



Declaration

I hereby declare that except where otherwise indicated, the work presented in this thesis is my own original work.

Regular Mapping of Multi-Dimensional Data on Parallel Processors

Peter Fletcher

May 1993

A thesis submitted for the degree of Doctor of Philosophy of
The Australian National University



Regular Mapping of Multi-Dimensional
Data on Parallel Processors

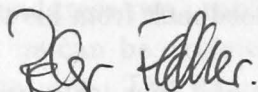
Peter Fletcher

May 1983

A thesis submitted for the degree of Doctor of Philosophy at
The Australian National University

Declaration

I hereby declare that except where otherwise explicitly stated, the work presented in this thesis is my own original work.



Peter Fletcher

Acknowledgements

Firstly many thanks to my supervisor, Phil Robertson, who provided me an opportunity to work in this area, convinced me of its importance, encouraged me to experiment and provided me with substantial feedback from his tireless reviewing of my drafts.

Special thanks to Guy Vézina, whose collaboration and friendship provided me with a great deal of enjoyment and motivation for this work, and whose software is responsible for many of the pretty pictures. Grateful thanks also to Ken Tsui for carefully reviewing the penultimate draft, finding many errors and omissions and assisting me in clarifying many dodgy areas.

Special thanks to people with whom I have had many valuable and stimulating discussions: Dave Abel, Don Bone, Oscar Bosman, Lisa de Ferrari, David Keightley, John Lilleyman, Scott Milton, Jonathon McCabe, Chris Moran, Heinz Schmidt, Kevin Smith, Duncan Stevenson, Ken Tsui and Andrew Vincent.

Thanks also to Don Fraser and Heiko Schröder who provided me with useful discussions and material at the start of my thesis; to John O'Callaghan who, in conjunction with the CSIRO, partially supported my work; to the members of my committee: Richard Brent, Iain Macleod and E. Krishnamurthy; to Tom Blank, Jeff Fier, Christopher Glaeser, Larry Levine and Russ Tuck who have helped me understand the MasPar; to Faye Baxendell who has helped me track down many an item; to Mike Sharrot for his help in producing videos; and to Peter Lamb who showed me how to produce postscript output anywhere.

Many people deserve thanks who have made my working environment a rich and enjoyable one: Stephen Barass, Arch Brayshaw, Dave Campbell, David Cook, Trish Devine, Kerry Douth, Peter Fox, Neale Fulton, Matthew Hutchins, Stuart Hungerford, Fei Jin, Steve Jones, Peter Milne, Kevin Moore, Richard Neville, Peter Nikitser, Mike Sharrot, Dione Smith (for the Strepsils too!), Roy Stockman, Paul Veldkamp, Kathy Visintin, Graham Williams and Steve Woods.

Thanks to those who have shown me that there is a life outside the computer: Helen, Catherine, Ben, Graham, and the Boys, Sally Kneebone, Dennis, Bruce, Kathy and Jamie, Chris and Nikki, Megan and Phil, Steve and Kirsty, Verena and Alex, Guy and Nathalie, Lindsay and Rod, Markus and Karen, Nicola and William, everyone at Aikido, and Chika, Sallie, Judy, Leigh, Graham, Tim, Peter and Carl at Boyce street.

Finally, thanks to Marcia and Bronwen for their love and forbearance, which gave me the freedom to work with enjoyment.

Abstract

This thesis presents a generalized framework for the mapping and remapping of large regularly-gridded multidimensional data sets on a parallel computer. We address two problems that influence the efficiency with which parallel computers can be exploited in image processing, visualization and simulation applications. The *data mapping problem* is the task of describing the layout of multi-dimensional data set on a parallel array. This layout has a significant effect on the choice and efficiency of processing algorithms. The *data remapping problem* is the task of moving data dynamically between data mappings to provide portability between applications, libraries and external devices, and allows the description of a class of data transformations of use in a variety of data processing algorithms.

We develop the *k*-Tile format, which provides a concise and flexible data mapping description for multidimensional data arrays on multidimensional devices, and allows the specification of many commonly used parallel data mappings, geometric transformations of these mappings, data replication and data padding.

Using the *k*-Tile format we define *Parallel mapping functions* (PMFs), which provide a general system for performing many remapping tasks. We introduce efficient algorithms for performing a subset of PMFs on a crossbar-connected parallel processing array with indirect addressing, and demonstrate an efficient implementation of these algorithms on a MasPar MP-1 computer. We also explore the problems involved in producing a complete implementation of PMFs on the MasPar, and suggest further work needed to produce such a system.

We show examples of the use of the *k*-Tile format and PMFs for the data mapping directives of High Performance Fortran, in image processing algorithms and in visualization applications.

Contents

1	Introduction	1
1.1	Multidimensional data	1
1.2	Parallel Architectures	2
1.3	A framework for data mapping	2
1.3.1	Storage and access requirements	2
1.3.2	Algorithm requirements	3
1.3.3	Portability requirements	3
1.3.4	A framework for data mapping	4
1.4	The structure of this thesis	4
1.4.1	Current approaches to data mapping	4
1.4.2	The k -Tile format	4
1.4.3	Radix 2 remapping	5
1.4.4	Implementation of radix 2 PMFs	5
1.4.5	Mixed radix remapping	5
1.4.6	The scope of PMFs	5
2	Multidimensional spaces, devices and data mapping techniques	7
2.1	Definitions	8
2.1.1	Multidimensional arrays	8
2.1.2	Data and index mappings	10
2.1.3	Multidimensional data arrays	12
2.1.4	Multidimensional devices	13
2.2	Data mapping on one-dimensional devices	15
2.2.1	The Multidimensional Tile Format	15
2.2.2	Remapping algorithms and applications	15
2.2.3	Hardware approaches to data mapping	16
2.3	Parallel architectures	16
2.3.1	Properties of Parallel Architectures	17
2.3.2	Three SIMD architectures	21
2.3.3	The MasPar and parallel programming in MPL	24
2.4	Current data mapping systems	32
2.4.1	Direct permutation	32
2.4.2	Dimension mapping	33

2.4.3	Data index computations	34
2.4.4	Index bit maps	35
2.4.5	Index digit maps	35
2.4.6	Blip schemes	36
2.4.7	Ad-hoc approaches	36
2.4.8	High Performance FORTRAN	37
2.5	Summary	38
3	The k-Tile format	41
3.1	The Basic k -Tile Format	42
3.1.1	The Data Type	42
3.1.2	The Data Space	42
3.1.3	The Device Space	43
3.1.4	The k -Tile Space	44
3.1.5	The k -Tile format, an overview	44
3.1.6	The k -Tile mapping	44
3.1.7	Specifying a k -Tile mapping	46
3.1.8	The implicit k -Tile mapping	48
3.1.9	The inverse k -Tile mapping	49
3.1.10	The k -Tile format, a definition	49
3.1.11	A basic k -Tile summary	50
3.1.12	Example mappings	51
3.2	Extending the k -Tile format	53
3.2.1	"Empty" k -Tile dimensions	54
3.2.2	Sense indicator	55
3.2.3	Templates	56
3.2.4	Offsets	59
3.2.5	Extended k -Tile offsets	63
3.2.6	Summary of extended k -Tile format	65
3.3	Parallel Mapping Functions	67
3.4	Summary	67
4	Radix 2 remapping	69
4.1	The index bit map	69
4.1.1	An overview of the index bit map	70
4.1.2	Definition of the index bit map	73
4.1.3	An index bit map notation	79
4.2	Radix 2 remapping	79
4.2.1	Definition of radix 2 remapping	81
4.2.2	A radix 2 remapping notation	82
4.3	Remapping with atomic operations	85
4.3.1	Assumed architectural features	85
4.3.2	Atomic index bit operations	86

4.3.3	Efficient use of atomic index bit operations	87
4.4	Optimal radix 2 remapping	88
4.4.1	Assumed architectural features	88
4.4.2	Types of cycles	89
4.4.3	A recursive approach	90
4.4.4	(p^*) cycles	90
4.4.5	(m^*) cycles	91
4.4.6	Simultaneous (m^*) and (p^*) cycles	95
4.4.7	Transforming mixed cycles into (mp^*) cycles	95
4.4.8	Even-parity (mp^*) cycles	97
4.4.9	Identity cycles	98
4.4.10	Remapping data while copying	101
4.5	Summary	102
5	Implementation of Radix 2 PMFs	105
5.1	The 2^k -Tile format	105
5.1.1	Converting a 2^k -Tile format to an index bit map	106
5.1.2	A canonical form of the 2^k -Tile format	107
5.2	Data types used by radix 2 PMFs	108
5.3	Functions used to access PMFs	110
5.3.1	k -Tile format manipulation	110
5.3.2	mtag manipulation	110
5.3.3	Remapping	111
5.3.4	Standard mappings	112
5.3.5	Geometrical transformations	113
5.4	Structure of the PMF system	114
5.5	PMFs using atomic index bit operations	114
5.6	PMFs using the optimal algorithm	114
5.6.1	Assembler coding	116
5.6.2	PE register usage	116
5.6.3	Chunking to larger data objects	116
5.6.4	Processor cluster optimizations	117
5.6.5	Using the xnet for P/M transpositions	119
5.7	Testing	119
5.7.1	Generation of random remappings	119
5.7.2	Checking performed remappings	120
5.7.3	Results of testing	120
5.7.4	A non-assembler library	121
5.8	Results	121
5.8.1	Execution time of radix 2 remapping	121
5.8.2	Hand coding vs. PMFs	124
5.9	Summary	127

6	Mixed radix remapping	133
6.1	The index digit map	133
6.1.1	Mixed radix numbers	133
6.1.2	Indexing with a mixed radix number	134
6.1.3	Specifying an index digit map	135
6.1.4	Specifying a mixed radix remapping	138
6.2	Aligned index digit remapping	141
6.2.1	Algorithm components	141
6.2.2	An algorithm for index digit permutation	146
6.3	Non-aligned index digit remapping	147
6.3.1	Re-signification of digits in an index digit map	147
6.3.2	Re-signification within device dimensions	148
6.3.3	"Brute force" remapping	150
6.3.4	Re-signification across device dimensions	153
6.3.5	Brute-force <i>P/M</i> re-signification	156
6.3.6	Brute force performance	157
6.3.7	Restricting to relatively prime digits	159
6.3.8	Skewing initial memory addresses	159
6.3.9	Restricting stack size	163
6.4	Cluster contention removal	166
6.4.1	A cluster contention removal algorithm	167
6.4.2	Removing contention in mixed-radix remapping	167
6.5	A system for mixed radix remapping	171
6.6	Summary	172
7	The scope of data mapping operations	175
7.1	High Performance Fortran	175
7.1.1	<i>ALIGN</i> and <i>REALIGN</i> directives	175
7.1.2	HPF <i>PROCESSORS</i> directive	179
7.1.3	Processor <i>VIEW</i> s	180
7.1.4	<i>DISTRIBUTE</i> and <i>REDISTRIBUTE</i> directives	180
7.1.5	<i>TEMPLATE</i> directive	180
7.1.6	PMFs \supset HPF	181
7.2	KIPS	181
7.3	Sample applications	182
7.3.1	Scan-line algorithms	182
7.3.2	2d rotation	182
7.3.3	Perspective viewing	185
7.3.4	2d scan-line virtualization	185
7.3.5	Volume rotation and rendering	188
7.3.6	The Fast Fourier Transform	193
7.3.7	Neighbourhood operations	194
7.3.8	Computing the Mandelbrot set	196

7.4	Summary	197
8	Conclusions	201
8.1	The data mapping problem	201
8.2	The data remapping problem	202
8.2.1	Radix 2 PMFs	202
8.2.2	Mixed radix remapping	203
8.3	Application of the approach	203
8.4	Limitations of the approach, and future work	204
8.4.1	Data structures	204
8.4.2	Data-dependent mappings and operations	204
8.4.3	A general PMF system	205
8.4.4	Human interaction with data mappings	205
	Bibliography	207
A	Cluster contention removal	213
B	Status of PMFs	221
B.1	Introduction	221
B.2	PMF system calls	223
B.2.1	The k -Tile format	223
B.2.2	The mtag	225
B.2.3	The remap	227
B.2.4	Standard mappings	229
B.2.5	Geometrical transformations	231
B.3	Examples of Using PMFs	231
B.3.1	Performing a simple remap	231
B.3.2	Error reporting	233
B.3.3	Fourier transform	236
B.3.4	Mandelbrot set generator	240
B.4	Some header files	244
B.4.1	gr.h	244
B.4.2	Extract from gp2.h	250
B.5	PMF Implementation notes	252
B.6	Using the PMF workbench	252
B.6.1	Declaring a k -Tile format	252
B.6.2	The kmap	253
B.6.3	Declaring an mtag	254
B.6.4	Declaring a pair	255
B.6.5	Data types attached to a pair	256
B.6.6	Performing a remap	259
B.6.7	Timing	259
C	Glossary of symbols	261

List of Figures

4.1	Representing a permutation in direct and cycle notation	74
4.2	An example index bit map	80
4.3	An example radix 2 remapping	83
4.4	Example remappings on index bits	85
4.5	Example atomic operations for a radix 2 remapping	87
4.6	A memory permutation as a linked list	92
4.7	Transformation of mixed cycle into $(m*)$ and $(mp*)$ cycles . . .	97
4.8	Parity masking to align $(mp*)$ cycles	99
5.1	Partial structure of a PMF implementation	115
5.2	Base time for MasPar instructions	123
5.3	Execution time of radix 2 PMFs	124
5.4	Lower bound time of radix 2 PMFs	125
5.5	PMFs vs. mpipl for 512×512 image	129
5.6	PMFs vs. mpipl for 1024×1024 image	130
5.7	PMFs vs. mpipl for 2048×2048 image	131
5.8	PMFs vs. mpipl for 512×2048 image	132
6.1	Aligning two index digit maps	139
6.2	Non-alignable index digit maps	139
6.3	P/M digit exchange	145
6.4	Re-signification within device dimensions	149
6.5	Brute-force re-signification	158
6.6	Brute-force re-signification without common factors	160
6.7	Inverse re-signification	161
6.8	Inverse re-signification with address skewing	162
6.9	Re-signification stack high-water-mark	164
6.10	Re-signification execution time with bounded stack	165
6.11	Router iterations required for random permutations	168
6.12	Router iterations required for index digit swaps	168
6.13	Contention removal time for random problems	169
6.14	Contention removal time for index digit swap problems	169
6.15	Finding a contention-free ordering of a processor permutation . .	170

7.1	Operations to rotate an image 45°	184
7.2	Perspective images generated on the MasPar MP-1	186
7.3	Volume renderings of human head and strange attractor	190
7.4	Volume remapping times for 1K PE MasPar	191
7.5	Volume remapping times for 8K PE MasPar	192
7.6	Comparison of Mandelbrot set calculation times	197
7.7	Test sections of the Mandelbrot set	198
A.1	Finding a contention-free ordering of a processor permutation	214
A.2	Execution time of scrambling contention removal	219

Chapter 1

Introduction

With the profusion of data being gathered, generated and processed today, more powerful computers and faster computational techniques are becoming essential. Because of physical constraints on the speed of electronic devices, significantly faster performance may only be obtained by pipelining or parallelizing computation.

Ultimately, this data must be presented in a form suitable for interpretation by a person. Many stages of processing may be required to present this data in a meaningful way, entailing the movement of data between and within storage, display and computational devices to allow various transformations to be applied.

The method used for assigning storage locations to data on any of these devices can have a significant effect on the efficiency of large-scale operations on data; we call this task the *data mapping problem*. Once storage locations have been assigned to data, it may be necessary to re-order the data on the device; we call this task the *data remapping problem*.

This thesis provides justification for exploring these two problems, and presents a framework for specifying and manipulating mappings of large regularly gridded data sets on parallel processors.

1.1 Multidimensional data

Large data sets can be generated from a variety of sources: optical scanners, remote satellite sensors, CT scanners, imaging spectrometers, video cameras and simulations are but a few. As sensor technology and computational power improve, the size of multidimensional data sets is growing rapidly.

Each type of data has associated with it some dimensionality. Spatial data is usually two or three dimensional; two dimensional data sets are commonly produced by remote sensing devices and optical scanners, and three dimensional data sets by CT scanners, MRI or by stacking two dimensional sections.

Dimensionality may also be increased by including information sampled along some other axis; for example, including spectral information at every data point may multiply the size of a data set hundreds of times, and sampling the data set temporally for an animation may multiply its size thousands of times again.

Many large data sets are regularly-gridded and rectilinear. By applying transformations to irregularly-gridded data, many data sets may be treated as regularly gridded and rectilinear. This form of data is ideally suitable for processing on a SIMD parallel processor.

1.2 Parallel Architectures

Although many of the techniques in this thesis are applicable to any class of computer, the algorithms in this thesis have been developed for a particular class of computer, a massively parallel distributed-memory SIMD computer. Machines of this class share a number of features which allow efficient algorithms to be found for many problems involving regular operations on a large dataset. These machines have many architectural variations which have a significant effect on the type of algorithms which can be implemented, and hence their efficiency.

The algorithms developed in this thesis were implemented on the MasPar MP-1. However, many of the ideas to be presented are also applicable to other parallel processors, and the specification techniques are applicable to any computer.

1.3 A framework for data mapping

When processing large multidimensional data sets, it is desirable that the computer architecture and software tools used satisfy several requirements: convenient storage and data access; appropriate positioning of the data to allow algorithms to operate efficiently (or efficient algorithms to be used); and portability of data and applications to other architectures with a minimum of decoding or recoding. These three requirements can be met by a suitable framework for data mapping.

1.3.1 Storage and access requirements

For fast processing and interaction with data we need to communicate data quickly between and within data source, viewing, bulk storage and processing devices. Dedicated devices such as frame buffers and scanners are often inflexible in their formats for accessing data, and standard data file formats also impose limitations on the order of access of multidimensional data. Many

processors, and massively parallel SIMD processors in particular, have specific requirements for the format of data storage, both for internal representation and access to data stored externally. Because these requirements are often imposed by system software packages, they cannot be changed to suit an application program's requirements; indeed, if several packages are being used together, their requirements can easily conflict.

To integrate all these devices in a data processing environment, and resolve the possible differences in data access requirements, the data handling problem may be consolidated into a generalized framework. This eases the task of programming data handling and allows compatibility to be provided between different programming efforts.

1.3.2 Algorithm requirements

When using a parallel processor to transform multidimensional data, the most convenient mapping of the data onto the processor might not be the most convenient mapping for actually performing the transformation, because different algorithms need to see different parts of the data during their execution. Two examples that illustrate this problem are neighbourhood filters and fast Fourier transforms.

When reading data from disk into a parallel processor the data is often stored simply in scan-line order, which makes it easiest to map scan-lines to processors. Unfortunately, the most efficient way of mapping a two dimensional data set to a processor array for the application of a neighbourhood filter is as a matrix of two dimensional tiles, and for a two dimensional Fourier transform the data may need to be index-digit reversed.

Rather than reading data from the disk in tiled or index-digit-reversed order, either of which is likely to be both slow and complicated (unless the data was originally stored in this order), it would seem more efficient to map the data onto the parallel processor in its natural order, and then use the parallel processor to remap the data using fast parallel techniques.

1.3.3 Portability requirements

Although most parallel computers have many features in common, there are several programming languages, data mapping techniques and terminologies for describing essentially the same data mappings.

To make life easier for the programmer, a consistent framework for describing and manipulating multidimensional data is needed to make it possible to both formally specify a convenient data mapping and provide a means to convert the specified mapping into any mapping used by other applications.

As both an aid to development and to enable software to be used on smaller systems, this scheme should not only be available on different parallel archi-

tectures, but also for use with sequential machines. In practice, using parallel techniques on sequential machines may have additional advantages, such as locally limiting and regularizing data access to local regions of memory, simplifying algorithms and reducing memory paging [61].

1.3.4 A framework for data mapping

A general framework is needed for describing data mappings conveniently in a way that is not constrained to a particular architecture. This framework must be high-level enough that efficient implementations are not restricted to a particular architecture or computing paradigm, and low-level enough that application programmers can participate in choosing data mappings appropriate for their particular problems.

As compiler technology improves these issues will increasingly be hidden from the programmer, but a good underlying model will still be necessary to enable compilers to handle large multidimensional data sets efficiently.

1.4 The structure of this thesis

The following paragraphs outline the structure of this thesis and the major focus for each chapter.

1.4.1 Current approaches to data mapping

The data mapping problem is not new; several systems for data mapping have been used for more than a decade, and schemes for improving data access efficiency on sequential computers and disk drives have been used for many years before that. Chapter 2 surveys some computer architectures, defines regular data structures more fully, and examines current approaches for data mapping. Limitations of current approaches are described.

1.4.2 The k -Tile format

The k -Tile format provides a basis for solutions to the data mapping problem which avoids many of the limitations associated with current approaches. The basic form of the k -Tile format allows many useful data mappings to be described, and extensions to the basic form allow even more flexibility.

Pairs of k -Tile formats may be used to define data remappings. We call this system for specifying remappings *parallel mapping functions* (PMFs). Chapter 3 defines the k -Tile format and PMFs.

1.4.3 Radix 2 remapping

Although the k -Tile format provides flexibility in specifying data mappings, it does not prescribe any algorithms for performing data remappings using parallel mapping functions. Indeed, it should not; to provide maximum portability, a variety of algorithms are necessary to suit the characteristics of different machine architectures.

Chapter 4 examines algorithms for performing parallel mapping functions based on k -Tile specifications with data-set dimensions restricted in length to powers of two. These operations are performed using the idea of *index bit permutation* [27,28], or *radix 2 index digit permutation*. A new set of algorithms with optimal communication characteristics is introduced for the MasPar MP-1

1.4.4 Implementation of radix 2 PMFs

Chapter 5 describes the implementation and interfacing of the algorithms described in chapter 4 for the MasPar MP-1. Architecture-dependent optimizations have been used to improve the speed of the implementation. Testing and timing results are shown and compared with other data remapping tools, showing that general data remapping algorithms compare favourably with purpose-built routines.

1.4.5 Mixed radix remapping

Although the radix 2 algorithms are efficient and regular, the restriction to powers of two causes a loss of flexibility. Chapter 6 introduces more general algorithms for performing *index digit permutation* and *re-signification*, again suitable for the MasPar MP-1, which may be used for performing remapping operations for general PMFs.

Several problems remain to be addressed in mixed radix remapping, and a mixed radix PMF system is outlined and suggested as future work.

1.4.6 The scope of PMFs

Chapter 7 explores the problems to be addressed by PMFs. The relationship between PMFs and the data-mapping directives of High Performance Fortran are examined. PMFs are demonstrated in a range of applications: scan-line algorithms used for affine transformations, surface view generation and volume visualization; neighbourhood operations such as filtering and mathematical morphology; the fast Fourier Transform; and pixel-local operations such as the generation of views of the Mandelbrot set.

Chapter 2

Multidimensional spaces, devices and data mapping techniques

As we have seen in the previous chapter, there is a good case for providing a framework for defining data mappings and algorithms for performing remappings efficiently. There are already many systems and techniques for specifying and manipulating the mapping of multidimensional data for both sequential and parallel computers.

To restrict the scope of this thesis, we will only be dealing with rectangularly gridded multidimensional objects with a fixed number of elements along each dimension. We will not examine irregularly shaped data sets or devices which cannot be made to fit within objects of this type. We will also assume that data mappings are *data-value independent*. These restrictions can increase the data storage required for an arbitrary data set, which in turn can reduce the efficiency of associated algorithms. However, the regularity obtained by these restrictions has many advantages, especially when using parallel architectures.

We will examine two problems associated with multidimensional data sets. When dealing with a multidimensional data set which must be stored in a storage or computation device, some strategy must be used to map the data elements onto the device. This is the data mapping problem referred to in the previous chapter. Once the data has been mapped to the device, we may wish to dynamically alter this mapping by permuting storage locations within the device. This is the data remapping problem referred to in the previous chapter.

2.1 Definitions

Before describing data mapping techniques, we describe a notation for multidimensional arrays and mappings.

2.1.1 Multidimensional arrays

A *multidimensional array* is a generalization of a 2d *matrix*. A matrix generally contains numbers; a multidimensional array may contain any data type. A matrix has only two dimensions, corresponding to rows and columns; a multidimensional array may have any number of dimensions. The number of elements in any row or column in a particular matrix is fixed; this is also true in a multidimensional array, in which the number of elements or *length* of each dimension is fixed.

Before providing a more formal definition of a multidimensional array, we must first define some related concepts:

- The *data element type* defines the elements within the array
- The *shape* defines dimensionality and the length of every dimension
- The *index space* defines the set of addresses of elements within an array

Data element type

A *data element type* \mathbb{T} is a space from which the elements of a multidimensional array are chosen. Because we wish to represent multidimensional arrays on a computer, \mathbb{T} will usually contain a finite number of elements allowing every element of \mathbb{T} to be represented in a fixed number of *bytes*, which we will treat as the smallest addressable unit of storage.

\mathbb{T} may also contain an *undefined* element, denoted \perp . When a multidimensional array is represented on a computer, \perp need not be a special value that may be distinguished from other data elements; it actually means that the element's value is undefined.

Shapes

A *shape* is a vector containing natural numbers. Letting

$$q \in \mathbb{N}$$

$$\mathbf{a} \in \mathbb{N}^q, \mathbf{a} = (a_0, \dots, a_{q-1}),$$

\mathbf{a} is a shape with dimensionality q , and

$$\text{dimensionality of } \mathbf{a} = \langle \mathbf{a} \rangle \triangleq q.$$

If $q = 0$, \mathbf{a} is represented by the empty set, \emptyset .

Index space

Given a shape \mathbf{a} with dimensionality q , the *index space* of \mathbf{a} is defined:

$$\text{index}(\mathbf{a}) \triangleq \{(u_0, \dots, u_{q-1}) \in \mathbb{N}^q : 0 \leq u_i < a_i\}.$$

If $\mathbf{a} = \emptyset$ or some element a_i of \mathbf{a} is 0, we define

$$\text{index}(\mathbf{a}) \triangleq \{\emptyset\}.$$

Multidimensional array

Letting

$$q \in \mathbb{N}$$

$$\mathbf{a} \in \mathbb{N}^q, \mathbf{a} = (a_0, \dots, a_{q-1})$$

\mathbb{T} be a set,

a multidimensional array \mathbf{A} of shape \mathbf{a} with data element type \mathbb{T} is a function

$$\mathbf{A} : \text{index}(\mathbf{a}) \cup \{\perp\} \rightarrow \mathbb{T} \cup \{\perp\}.$$

\mathbf{A} and \mathbf{a} have the following properties:

$$\text{dimensionality of } \mathbf{A} = \langle \mathbf{A} \rangle \triangleq q \quad (2.1)$$

$$\text{shape of } \mathbf{A} = [\mathbf{A}] \triangleq \mathbf{a} \quad (2.2)$$

$$\text{size of } \mathbf{A} = |\mathbf{A}| \triangleq |\text{index}(\mathbf{a})| \quad (2.3)$$

$$\text{length of dimension } i \text{ of } \mathbf{A} = [\mathbf{A}]_i \triangleq a_i \quad (2.4)$$

$$\text{index space of } \mathbf{A} = \text{index}(\mathbf{A}) \triangleq \text{index}(\mathbf{a}) \quad (2.5)$$

$$\mathbf{A}(\perp) \triangleq \perp \quad (2.6)$$

Note that in definition 2.3, unless $q = 0$ or an element a_i of \mathbf{a} is zero, the size of \mathbf{A} may be calculated

$$|\mathbf{A}| = |\text{index}(\mathbf{a})| = a_0 \times \dots \times a_{q-1}.$$

Multidimensional spaces

The set of all multidimensional arrays, the *multidimensional array space*, is denoted \mathbb{M} . Three subsets of \mathbb{M} may be defined

$$\text{arrays of shape } \mathbf{a} = \mathbb{M}_{\mathbf{a}} \triangleq \{\mathbf{A} \in \mathbb{M} : [\mathbf{A}] = \mathbf{a}\} \quad (2.7)$$

$$\text{arrays of type } \mathbb{T} = \mathbb{M}_{\mathbb{T}} \quad (2.8)$$

$$\triangleq \{\mathbf{A} \in \mathbb{M} : \forall u \in \text{index}([\mathbf{A}]), \mathbf{A}(u) \in \mathbb{T}\}$$

$$\text{arrays of shape } \mathbf{a}, \text{ type } \mathbb{T} = \mathbb{M}_{\mathbf{a}, \mathbb{T}} \triangleq \mathbb{M}_{\mathbf{a}} \cap \mathbb{M}_{\mathbb{T}} \quad (2.9)$$

For example, let

$$\mathbf{a} = (2, 3)$$

$$\mathbf{A} : \text{index}(\mathbf{a}) \cup \{\perp\} \rightarrow \mathbb{R} \cup \{\perp\}$$

$$\mathbf{A}(r, c) = \text{element } (r, c) \text{ of the matrix } \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

More simply, we can write

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix}$$

We can state the following properties of \mathbf{A} :

$$\mathbf{A} \in \mathbb{M}, \mathbf{A} \in \mathbb{M}_{\mathbb{R}}, \mathbf{A} \in \mathbb{M}_{\mathbf{a}}$$

$$\langle \mathbf{A} \rangle = 2$$

$$[\mathbf{A}] = (2, 3)$$

$$[\mathbf{A}]_0 = 2, [\mathbf{A}]_1 = 3$$

$$\text{index}(\mathbf{A}) = \{(0, 0)(0, 1)(0, 2)(1, 0)(1, 1)(1, 2)\}$$

$$|\mathbf{A}| = 2 \times 3 = 6$$

$$\mathbf{A}(1, 0) = 3$$

$$\mathbf{A}(\perp) = \perp.$$

2.1.2 Data and index mappings

Using the above definitions, functions may be defined between multidimensional arrays. For example, this function adds one to every element in a multidimensional array with shape \mathbf{a} :

$$f : \mathbb{M}_{\mathbf{a}, \mathbb{R}} \rightarrow \mathbb{M}_{\mathbf{a}, \mathbb{R}}$$

$$f(\mathbf{A})(\mathbf{u}) = \mathbf{A}(\mathbf{u}) + 1 \text{ for all } \mathbf{u} \in \text{index}(\mathbf{a}).$$

Data mappings

A *data mapping* is a function between two multidimensional arrays with fixed (but not necessarily identical) sizes and of the same type \mathbb{T} which copies elements from the argument into the result.

Letting

$$p \in \mathbb{N}, \mathbf{a} \in \mathbb{N}^p$$

$$q \in \mathbb{N}, \mathbf{b} \in \mathbb{N}^q,$$

we define a data mapping f :

$$f : \mathbb{M}_{\mathbf{a}, \mathbb{T} \cup \{\perp\}} \rightarrow \mathbb{M}_{\mathbf{b}, \mathbb{T} \cup \{\perp\}}$$

$$f(\mathbf{A})(\mathbf{v}) = \left\{ \begin{array}{ll} \mathbf{A}(\mathbf{u}) & \text{for some } \mathbf{u} \in \text{index}(\mathbf{a}) \\ \perp & \text{otherwise} \end{array} \right\} \text{ for all } \mathbf{v} \in \text{index}(\mathbf{b}).$$

Index mappings

Some data mappings may be expressed by defining an *index mapping* from one index space into another. Letting

$$p \in \mathbb{N}, \mathbf{a} \in \mathbb{N}^p$$

$$q \in \mathbb{N}, \mathbf{b} \in \mathbb{N}^q,$$

we define an index mapping g

$$g : \text{index}(\mathbf{b}) \cup \{\perp\} \rightarrow \text{index}(\mathbf{a}) \cup \{\perp\}.$$

From the index mapping g we may define a data mapping f :

$$f : \mathbb{M}_{\mathbf{a}, \mathbb{T} \cup \{\perp\}} \rightarrow \mathbb{M}_{\mathbf{b}, \mathbb{T} \cup \{\perp\}}$$

$$f(\mathbf{A})(\mathbf{v}) = \mathbf{A}(g(\mathbf{v})) \text{ for all } \mathbf{v} \in \text{index}(\mathbf{b}).$$

It is necessary to define the mapping g 'backwards' to ensure that only one element of \mathbf{A} is mapped to each element of the result. A mapping that may be defined in this way is *data independent* because the correspondence between elements depends only on positions of data elements and not their values. Conversely, any data independent mapping may be defined by an index mapping. A *data dependent mapping* can map elements to different positions depending on the contents of the array being mapped. Letting

$$a \in \mathbb{N},$$

an example of a data dependent mapping is a function which sorts real vectors of length a :

$$f : \mathbb{M}_{a, \mathbb{R}} \rightarrow \mathbb{M}_{a, \mathbb{R}}$$

$f(\mathbf{A})$ = the elements of \mathbf{A} in sorted order for all $\mathbf{A} \in \mathbb{M}_{a,\mathbb{R}}$.

Letting

$$(a_0, a_1) \in \mathbb{N}$$

$$\mathbf{a} = (a_0, a_1)$$

$$\mathbf{b} = (a_1, a_0),$$

an example of a data independent mapping is an array transposition:

$$f : \mathbb{M}_{a,\mathbb{R}} \rightarrow \mathbb{M}_{b,\mathbb{R}}$$

$$f(\mathbf{A})(u_1, u_0) = \mathbf{A}(u_0, u_1) \text{ for all } (u_1, u_0) \in \text{index}(\mathbf{b}).$$

Expressed using an index mapping, the transposition example may be written:

$$g : \text{index}(\mathbf{b}) \cup \{\perp\} \rightarrow \text{index}(\mathbf{a}) \cup \{\perp\}$$

$$g(u_1, u_0) = (u_0, u_1) \text{ for all } (u_1, u_0) \in \text{index}(\mathbf{b})$$

$$f : \mathbb{M}_{a,\mathbb{R}} \rightarrow \mathbb{M}_{b,\mathbb{R}}$$

$$f(\mathbf{A})(u_1, u_0) = \mathbf{A}(g(u_1, u_0)) = \mathbf{A}(u_0, u_1) \text{ for all } (u_1, u_0) \in \text{index}(\mathbf{b}).$$

To simplify this notation, \perp will often not be included in data and index mappings; it may be assumed to be present unless stated otherwise.

2.1.3 Multidimensional data arrays

A multidimensional data array is a multidimensional array with some meaning attached to each of the dimensions. Whether a dimension represents time, spatial dimensions, temperature, frequency or any other quantity, it can be treated identically when it is represented as a dimension of a multidimensional array. Our description of a multidimensional data array is derived from the description of a KIPS *image* [66]; thus, the term *image* will often be used interchangeably with *multidimensional data array*.

A distinction can be made between *implicit* dimensions of the data, whose values are implicitly defined by their positions in the multidimensional data array, and *explicit* dimensions, whose values are explicitly specified as part of the array data type [66].

The same data set may be represented in several ways depending on whether each dimension is stored explicitly or implicitly. As an example, a data set representing the elevations of a two-dimensional grid of points on a landscape may be stored in several ways:

- i. Two implicit dimensions representing spatial position (x, y) and one explicit dimension representing height z

- ii. Three implicit dimensions, two of which represent spatial (x, y) position and one of which has a binary value representing the presence or absence of earth
- iii. One implicit dimension, representing nothing physical, and three explicit dimensions giving (x, y, z) values

Note that not all combinations may be used; it is usually not possible to represent a data set with an implicit height dimension z and explicit spatial dimensions (x, y) because of the requirement that each data element has the same size.

The representation used will depend on the use to which the data is put; representation (i) is the more usual, as the data is in a suitable form for many common image processing and visualization operations. Representation (ii) is necessary if the surface has any overhangs (i.e. the height field is not single-valued), and representation (iii) might be used if the surface were to be approximated by polygons.

The storage representation chosen also has an effect on quantization error, storage requirements and algorithmic efficiency. Quantization error is introduced into both the explicit and implicit representations. Implicit dimensions are quantized by the regular grid used to sample the data; this error may be reduced by sampling to a finer grid with a proportional cost in storage requirements and processing time. Explicit dimensions are quantized by a finite data type used to represent data elements; this error may be reduced by an approximately logarithmic cost in storage requirements and processing time. However, converting an implicit dimension to an explicit dimension may have great costs in both the need to explicitly represent all data values, and the greater algorithmic complexity in searching for particular data values.

2.1.4 Multidimensional devices

Devices may also be treated as multi-dimensional arrays.

We use the term *storage device* to refer to any device upon which data may be written and retrieved. Thus, a disk drive, a sequential computer's memory, machine readable display devices and distributed-memory parallel processing arrays are all examples of storage devices. We will use the term *memory* specifically for the fast random-access memory attached to processors.

Common examples of one-dimensional storage devices are disk files and a sequential computer's memory, where the elements are addressable by a single index.

Multi-dimensional devices are not yet so common. A distributed memory parallel computer can be regarded as a two-dimensional array, where the memory forms one dimension and the processor array one or more dimensions. A

two-, or even three-, dimensional display can be regarded as multidimensional, and striped disks may be regarded as a very long and thin two-dimensional device, with a long axis being an address on disk and a short axis selecting the disk drive.

Disk files and sequential computers' memories also have some characteristics which are similar to distributed computers' memories which can in some circumstances make them behave as a two-dimensional storage device. Disk files are usually written as a series of *blocks*, where a block is the smallest unit of bytes that can be read or written at a time. Once a block has been read into a buffer, access to data in that block is very fast. This can make it appear as if the disk file has a two dimensional structure; access along the first dimension (inside the block) is very fast, and access along the second dimension (between blocks) is generally slower.

Similarly, most computers nowadays use *virtual memory*, which also segments memory into blocks. Access to blocks in physical memory is very fast, and access to blocks in secondary memory is slower. For computers with a fast data cache, a third level of access is added.

Although disk files and virtual memories do share some characteristics with multidimensional storage devices, there are many differences which limit the usefulness of the analogy; it is easy to find situations which cause poor performance in disk file access or virtual memory usage, but using these systems efficiently requires detailed knowledge about the underlying hardware and operating system.

The dimensionality of a distributed memory parallel processors is more evident and, unless programming in a parallel language which hides such things from the programmer, the dimensions are explicitly present in the programming language as memory references and communications operations.

In a multidimensional storage device, we will use the convention of numbering the dimensions in order of increasing access time. Usually, this will mean the memory dimension is device dimension zero. For the purposes of this thesis a memory array on a parallel device will be treated as if it were a data array, although there are physical restrictions which will affect the way data elements are accessed.

If a data array A is larger than the device array D we wish to map it onto, it will only be possible to map a subset of that data array onto the device. Usually we will assume that $|A| \leq |D|$, unless specified otherwise.

As an example, the smallest MasPar computer, the MasPar MP-1201-A, has 1024 processors, each with 16384 bytes of memory [4]. This could be treated as a multidimensional storage device with two dimensions, with the first dimension representing memory and containing 16384 elements, and the second dimension representing the processors and containing 1024 elements, giving a shape of [16384, 1024].

However, it is rarely possible or desirable to use all of a computer's memory

for storage of a single object. Specifying a smaller memory dimension allows many device arrays to be specified for the same physical device.

A MasPar can also be treated as a two-dimensional mesh computer. If our data array was 256 k-bytes or smaller, we could specify a device array with shape [256, 32, 32].

2.2 Data mapping on one-dimensional devices

When processing large multidimensional data sets on sequential computers and one-dimensional storage devices, the data mapping chosen can be important to the choice of algorithms and the efficiency of operations. Many data processing operations on these devices also require data remapping in the course of their execution. This section examines some current approaches to these issues.

2.2.1 The Multidimensional Tile Format

A method that allows flexibility in mapping multi-dimensional arrays onto a one-dimensional storage device, such as a disk file, is Fraser's *Multidimensional Tile Format* [29]. By hierarchically breaking down the image dimensions into a higher-dimensional space the tile format can describe many existing image storage formats, and allows scope for describing many more.

As well as providing flexibility in storage format specification, Fraser has derived efficient algorithms for reading and writing arbitrary blocks from a tile-format image file with only one pass over the file.

By streamlining the notation, adding some regularizing constructs, and including support for multidimensional devices, from the Multidimensional Tile Format we derived the parallel k -Tile format. A full description of the parallel k -Tile format is contained in section 3.1.

Some aspects of Fraser's work on the tile format have not been carried over to the k -Tile format, notably the application of the format to the storage of sparse data files.

2.2.2 Remapping algorithms and applications

When a data set is larger than the physical memory of a sequential computer, it is sometimes necessary to re-order the elements of the data set in a disk file to allow an image processing operation to be performed quickly. Even when an image will fit into the available memory, the properties of the algorithm may require the data to be remapped. Some examples of these operations are: scan-line operations [9, 31, 33, 44, 58, 61, 62, 71, 72], which require two- or three-dimensional images to be reflected and transposed about their axes; and different forms of the Fast Fourier Transform [8, 11, 15, 32, 35], which require

multi-dimensional images to be transposed about their axes and their indices bit-reversed (chapter 7 examines these problems in more detail).

Several efficient algorithms for performing these operations have been used for many years, and all of them use, or are equivalent to, index-bit and index-digit permutation algorithms. Fraser uses index-bit reversal techniques to allow images stored on disk or in an image display to be transposed or converted into a tiled format using only a small memory buffer [27, 28, 70]. Van Heel uses a mixed-radix perfect shuffle algorithm to perform transpositions of large multidimensional data sets using only small internal buffers, with the efficiency of the algorithm increasing with the size of the buffer [16, 30].

Many mapping and remapping techniques are *data dependent*, where the manner in which a pixel is stored depends upon its own values and those of its neighbours. Quad-trees and oct-trees [63], run-length encoding, numerical sorting and any form of data compression fall into this category. As we have limited the scope of this thesis to data-value independent mappings, we will not consider these further.

2.2.3 Hardware approaches to data mapping

Several hardware approaches have been taken to allow mapping-related tasks to be performed in hardware.

Lilleyman shows how the inclusion of a cross-bar switch between a micro-processor's address lines and its memory makes many implementations of the FFT more efficient by allowing data re-ordering to be performed in hardware with no movement of data between memory cells [45, 46].

Newman shows that with the use of a memory-management unit that pages data by tiles of a two-dimensional data array, rather than by rows (or parts of rows), the performance of many image-processing and graphics algorithms performed in a small physical memory can be greatly improved. The tiling is achieved by using the lowest-significant row- and column-bits within the data addresses to access data within a page, instead of the lowest-significant address bits. One example problem that began thrashing in 13 megabytes of RAM without tiling could be performed in an acceptable time using only two megabytes of tiled RAM [57].

Of course, either of these techniques could be simulated using software index bit permutation, but at a much greater cost in execution time and programming complexity.

2.3 Parallel architectures

There are a large number of parallel architectures available today, many of which have significantly varying characteristics. We outline some of these

characteristics and then examine one parallel machine, the MasPar MP-1, in more detail. For the problems addressed in this thesis, the characteristics of the MasPar MP-1 are ideal [4]. We will also mention properties of two other parallel machines, Active Memory Technology's Distributed Array Processor (AMT DAP 500) [59] and Connection Machines Connection Machine 2 (CM-2) [37, 67]; except for the algorithms, which are specifically for computers with characteristics similar to those of the MasPar, the specification techniques outlined in this thesis could be used with any computer architecture.

2.3.1 Properties of Parallel Architectures

Instruction execution

Parallel machines may be divided into two classes, *single instruction multiple data* (SIMD) and *multiple instruction multiple data* (MIMD). SIMD machines contain many processors or *processing elements* (PEs), all executing the same instructions simultaneously; conditional execution and looping must be performed by temporarily disabling and enabling groups of PEs. The processors in MIMD machines are able to execute instructions independently, and each processor is more similar to that of a sequential machine. MIMD machines are sometimes used restrictively, with every processor executing the same program independently; this class of machine has been dubbed *single program multiple data* (SPMD).

In general a MIMD machine is at least as powerful as the corresponding SIMD machine with the same number of processors and same processing power within each processor, because the MIMD machine used in SPMD mode would perform at least as well as the SIMD machine. However, in terms of financial and physical cost, the SIMD concept has a number of advantages; because the instruction stream need only be decoded once, more processors can be built in a smaller space. Similarly, because the instruction stream need only be stored in one place, many megabytes of memory are saved. Many operations such as synchronization, memory operations and communication also benefit from central control [50].

These properties enable SIMD machines to achieve very high performance at a low cost for some classes of problems. Because of the regularity of the problems we have chosen to address in this thesis, SIMD machines are ideal for their solution.

Memory addressing

Parallel machines may be further divided into two other classes, *shared memory* and *distributed memory* machines. In shared memory machines, every processor has access to the same address space. Because of the technical difficulty

in allowing many processors to communicate with a single memory, parallel shared memory machines have only been implemented with a few processors. This in turn makes the MIMD paradigm more appropriate for shared memory machines.

In distributed memory machines, every processor has access to an individual memory space. This is technically easier to implement, but means that some form of inter-processor communication must be included in the architecture to allow data stored in different memories to be integrated.

Memory addressing in SIMD architectures comes in two forms: *direct* addressing and *indirect* addressing. With direct addressing, the address to be accessed is part of the instruction stream and hence is the same for every processor. In indirect addressing, the address to be accessed is generated from within each processor, and hence every processor may access data from a different address in its local memory.

Direct addressing is technically easier to achieve, both because addresses may be generated from outside the PEs and because the RAM hardware may be used more efficiently. However, the use of indirect addressing offers a number of advantages: Load-balancing may be improved by allowing each processor to proceed through a set of tasks in memory at a different rate [65]; parallel linked lists of different lengths and containing different elements may be implemented; and blocks of data may be moved by different offsets in every processor, which is very important in *scan-line algorithms* (see section 7.3.1).

Interprocessor communication

Some form of inter-processor communication is essential when dealing with distributed memory computers. There are trade-offs to be made between communication speed and flexibility, and consequently there are many solutions that have been used to solve the communications problem. We will mention three mechanisms for inter-processor communication among N processors: the 2d mesh, the hypercube and the crossbar switch. A thorough exploration of the properties of, and algorithms for, many communication networks is found in Leighton's book [43], and forms the basis for the following comments.

Mesh communication is a simple form of inter-processor communication, as it only involves connections between neighbouring processors on a multidimensional grid. We will also assume that communication may only occur in one direction at a time. 1d and 2d meshes are the simplest to implement, because the mesh may easily be laid out with physically short connections between processors; 3d meshes could be implemented with a three-dimensional circuit structure, but there must always be longer wires for four-dimensional meshes and up.

Although 1d meshes are simplest to implement, there is a significant cost in communication when implementing any algorithms requiring inter-processor

communication beyond nearest-neighbour. Scan-line algorithms would appear to be ideal (see section 7.3.1), but require transposition of the image, which can be performed in a minimum of $N(N-1)/2$ steps.

The maximum distance between processors in 2d meshes is $2\sqrt{N}$, which is a great improvement over a maximum distance of N in 1d meshes. If diagonal and toroidal wrap-around connections are allowed, as in the MasPar MP-1, the maximum distance in a 2d mesh is reduced by a factor of four. However, algorithms for arbitrary processor permutations on the mesh require a queue of size at least $O(\log N)$ and $O(\sqrt{N})$ steps, and are quite complex.

The hypercube is a special case of a multidimensional mesh: in a hypercube with N processors, the processors are connected together in a $\log_2 N$ dimensional mesh with each dimension having a side-length of two processors. In addition, implementations of the hypercube usually have an important advantage over implementations of the mesh: different processors can communicate along different axes simultaneously.

The hypercube has a number of desirable properties; by carefully enumerating processors and providing the network with the ability for different processors to communicate along different axes of the hypercube simultaneously, lower-dimensional meshes whose dimensions are powers of two may be embedded within the hypercube. Thus, algorithms suitable for two-dimensional meshes may be used directly on the hypercube.

The connectivity of the hypercube is also extremely suitable for one component of the radix 2 FFT [11]; all the communications required for the butterfly communication component of the algorithm are between indices which differ by a power of two, which are stored in neighbouring processors. However, the bit-reversal component of the FFT using the simplest algorithm for hypercube routing, the "greedy algorithm", requires $O(\sqrt{N})$ steps, which is no better than the performance of a 2d mesh. For average routing problems only $O(\log N)$ steps are required, with very bad performance for the worst cases. Algorithms for sorting can be used to construct routing algorithms that are guaranteed to always perform well, but are quite complicated.

Although the hypercube is quite powerful, the number of connections to each processor can grow large as the number of processors increases; each processor would require 16 connections in a 65536-node hypercube. Some variations of the hypercube architecture with bounded degree exist that have similar computational properties.

The cross-bar switch provides a very simple communication model which allows any routing problem to be performed efficiently; any processor may connect to any other processor, and as long as there is no contention for processors, any communication operation may be performed in one step. The MasPar MP-1 includes a global router which is very similar to a crossbar switch in its operation, but has two important differences, mentioned in section 2.3.2.

Mixing sequential with parallel processing

Very few problems are soluble only with parallel computation, and many parallel computations require some form of data reduction from many values to one in the course of their execution. While such sequential computation is occurring, it may not be possible for parallel computation to proceed and many processors must be idle.

In order to ensure that this sequential time is minimized, sequential operations must be made as fast as possible. There are three approaches that have been used.

Firstly, in the case of MIMD machines, each processing node is approximately as powerful as the processor in a microcomputer or workstation, and is fast enough that sequential computation can be performed within a single processor (or redundantly in every processor) without causing a large bottleneck. The processors in massively parallel SIMD machines, however, are individually quite slow: execution of sequential code would be a large bottleneck.

A second approach is to perform all the sequential processing in the front end processor, which is usually a workstation. This approach allows fast sequential processing, but a bottleneck often remains in the relatively slow connection between the front end and the parallel processors. On a multi-user machine, job swapping may also degrade the performance of the parallel processor by preventing instructions being sent to the parallel processors as quickly as they could.

A third approach is to provide a dedicated sequential processor to control the SIMD array. This allows sequential processing to be performed very quickly and to be easily integrated with computations from the parallel processors. Because the sequential processor must be capable of both performing general sequential computation very quickly and controlling the parallel processing array, a custom architecture will usually be necessary, perhaps increasing the cost and complexity of the system.

Parallel register sets

For reasons of space, power consumption and cost, the processors in massively parallel SIMD machines are generally fabricated with many processors on a single chip. There is no room on these chips for large quantities of RAM, so that the bulk distributed memory must be kept off-chip. The number of pins available on a processor chip for memory access are limited, so memory access must either be performed with very narrow data paths or be multiplexed into the processors. Thus, memory access on a massively parallel processor is slow.

It is possible to limit the memory accesses of a parallel processor by including a large set of processor registers, so that intermediate results of computations may be kept on-chip rather than being transferred back and forth

between bulk memory and the processors [10].

Error checking

Because of the large number of components in a massively parallel processor, failure is more likely than in a smaller computer. Failure may also occur silently, because a complete failure of a single processor may only affect a tiny part of a large computation. Thus, designers of massively parallel processors include extensive error checking hardware in their machines, such as parity checking of memory and communications, error-correcting code in bulk storage and master/slave processor arrangements to verify computation [50, 59, 67]. Ideally, these measures should be invisible to the user of such a machine, and we will not examine them further.

2.3.2 Three SIMD architectures

The AMT DAP 500

The DAP 500 [59] is a massively-parallel SIMD architecture containing 1024 or 4096 PEs. The PEs are controlled by a master control unit (MCU), which takes instructions from a code memory of between 512 k-bytes and 2 M-bytes, interprets them and controls the PEs in the array.

The PEs are arranged in a square mesh of 32×32 or 64×64 elements. A bus system also connects processors by rows and columns, providing rapid broadcasting and fetching facilities.

Each processor in the DAP contains the following:

- between 32k-bits and 1M-bits per PE of bit-addressable memory
- seven 1-bit registers
- a mesh connection to its four nearest neighbours, communication being between either a register or a bit from its store
- fast connections to a row and a column bus
- a one-bit ALU

The DAP includes an I/O interface allowing 50M-bytes/second to be transferred over a fast data channel. Although this speed is substantially slower than the I/O speeds of the MasPar and the CM-2 presented in the following paragraphs, this I/O operation takes only 5% of the processor cycles.

The Connection Machines CM-2

The CM-2 [37, 67] is a massively-parallel SIMD architecture containing between 16384 and 65536 PEs. The PEs are controlled by nano-instructions from an instruction sequencer, which contains 64K 96-bit words of microcode storage, and is in turn controlled by instructions from a front end workstation. Up to four sequencers may be present in the machine, allowing four separate processes to be operating simultaneously on four banks of 16384 processors.

On the CM-1 there were two inter-processor communication networks: a 16-dimensional hypercube for global routing or a 2d mesh, dubbed NEWS. As a simplification, the CM-2 has only the hypercube network, with the ability in hardware to embed any multi-dimensional network within the hypercube network, thus preserving the functionality of the NEWS network. The hypercube router performs queueing, pipelining, routing decision-making, and combination of messages destined for the same address in hardware.

Each processor in the CM-2 contains the following:

- 64k-bits of bit-addressable memory
- four 1-bit registers
- a NEWS-grid interface to support multi-dimensional meshes
- a connection to the hypercube global routing network
- a connection to a global-or network for data reduction

In addition, an optional floating-point unit is available for sharing between each 32 processors. The CM-2 does not support indirect memory addressing directly, but by the use of the floating-point hardware, indirect addressing can be achieved with 16 processor sharing each indirect address.

The instruction sequencer also allows the CM-2 to be treated as a virtual machine, containing many more processors than physically exist. By segmenting the CM-2's memory into virtual processors, the machine may be treated as if it had any number of processors, as long as they can fit within memory.

An I/O interface from the CM-2 to an external group of disk drives, the DataVault, is capable of transferring data at a rate of 40 M-bytes per second. By operating eight data vaults in parallel, speeds of 320 M-bytes per second could be achieved.

The MasPar MP-1

The MasPar MP-1 [52] is a massively-parallel architecture containing between 1024 and 16384 PEs. The PEs are controlled by an *Array Control Unit* (ACU), which is itself a 32-bit RISC-style processor. A 2d toroidal mesh with diagonal

connections and a crossbar-like global router provide two mechanisms for inter-processor communication.

Each PE contains the following:

- A 32-bit accumulator for arithmetic and logical operations
- A 4-bit ALU
- 4-bit internal data paths
- A 32-bit status word, containing integer and floating point arithmetic, memory, execute enable, communication and error checking flags
- Forty 32-bit general-purpose registers
- An 8-bit wide connection by indirect or direct addressing to either 16K bytes (MP-12XXA) or 64K bytes (MP-12XXB) of RAM
- A 1-bit wide connection to the 2d mesh xnet network
- Access to a 1-bit wide global router connection, shared with 15 other processors
- A 4-bit wide connection to a global-or network, connected to the ACU
- A 4-bit wide connection to a global distribution network, allowing single values from the ACU to be transferred to every PE

The parallel instructions in the instruction set hide the four-bit nature of the PEs internal operations. Similarly, multiplication, division and floating-point operations are microcoded in the instruction set. A special M-machine queues PE memory requests to allow parallel processing to continue while memory operations are occurring; if a register involved in a memory access is read or written, computation stalls until the associated memory access is completed.

The global router is very similar to a crossbar switch connecting all processors, but there are two important differences. Firstly, each connection to the global router is shared by a *cluster* of sixteen processors; in many cases, this reduces performance by a constant factor of sixteen, as router links are assigned sequentially to processors within the clusters. For many processor permutations the performance is worse than this because the hardware cannot always assign router links to processors in an optimal order. Secondly, when there are more than 1024 processors in the MasPar, not all cluster permutations are supported by the hardware. This means that some processor permutations must take more than sixteen iterations to complete. Measurements of the properties of the global router and some partial solutions to these problems can be found in section 6.4 and in Preschelt's paper [60].

The ACU contains the following features:

- Thirty-two 32-bit general-purpose registers, with register 0 always returning a zero value
- Up to thirty-two special registers for status flags, timing, PE status flags and front end communication registers
- A 32-bit ALU for integer arithmetic, comparison, boolean and shift operations
- Connection to 128K bytes of data memory
- Connection to 1 M-byte of physical instruction memory, demand-paged in a 4G-byte address space

Because the ACU does not have any floating point capability, sequential floating-point operations must be performed on the PE array. The ACU has a RISC-style instruction set, allowing many sequential instructions to execute in one or two clock cycles. Because of the small data paths in the PEs, most parallel instructions operate at a rate of 1, 2 or 4 bits per clock cycle, which is 80nS.

Because there is only a single ACU, only one process may operate at a time on the MasPar. However, by either segmenting PE memory or paging data to disk, several processes may share the resources of the MasPar by time-slicing.

Communication between the MasPar and the front end processor is usually via the VME bus through the ACU. However, the facility to perform I/O directly with the PEs through the router links gives the prospect of extremely fast I/O: a 16384 PE machine has 1024 router links, each capable of a peak speed of one bit per clock, giving a peak data rate of more than a gigabyte per second. However, as yet the only external device which can utilize this speed is a RAM board, which may be used to simulate a fast disk-drive or to buffer data between the PEs and the front end.

The MasPar MP-2 has now been released. The MP-2 has a 32-bit ALU, 32-bit data paths within the PEs, a 16-bit wide connection to memory and hardware for faster arithmetic shifts and floating point operations. The clock-rate and inter-processor communications speeds are identical in the MP-2.

2.3.3 The MasPar and parallel programming in MPL

Much of the programming work described in this thesis was performed in the AMPL programming language on a MasPar MP-1 computer [4, 48, 49]. Similarly, all the example algorithms are presented in AMPL. AMPL is based upon both ANSI C and upon an earlier MasPar programming language, MPL, which

is closer to K&R C [34, 38, 48]. However, MPL is now regarded as obsolete, and we will follow MasPar's convention and refer to the ANSI compiler as MPL.

Although any C program for a sequential machine may be compiled and executed without modification for the MasPar using MPL (within the MasPar's memory limitations), MPL is a low-level parallel language in the sense that most of the language constructs map in a direct way to the underlying architecture of the MasPar. Although there are no concepts of virtualization or variant processor topologies in MPL, some language operations that appear to be 'atomic' actually require loops to be executed internally; these exceptions will be noted in the following paragraphs.

Memory model

Data objects in MPL are of two types: *singular* and *plural*. Singular data objects are stored in the data memory of the ACU. All the data objects declared in a normal C program would be treated as singular by MPL. Plural data objects are stored in the distributed memory of the PEs; a single plural declaration will allocate one data object in every PE.

This example declares an array of ten integers in the singular memory of the ACU:

```
int k[10];
```

This example declares as many integers as there are processors, one in each of the distributed memories of the PEs:

```
plural int m;
```

This example declares as many arrays as there are processors, one in each of the distributed memories of the PEs:

```
plural int p[10];
```

Thus, on a MasPar containing 1024 PEs, the previous example declares 10 integers per PE, or 10240 integers altogether.

As well as the singular ACU memory and the plural distributed PE memory, the MasPar includes a third memory address space containing instructions to be executed by the ACU and PEs. This address space is not readable (except for the reading of instructions) or writable except by the front end host. Except for disallowing the possibility of self-modifying code, this restriction has no effect on normal C programs.

Expressions

Singular expressions are computed in MPL just as they are in C. Purely plural expressions are also computed in MPL just as in C, except they are performed in parallel across all the active processors. Thus, using the previous declarations,

```
k[0] = k[1]+k[2];
```

would add the two values $k[1]$ and $k[2]$ and store the result in $k[0]$.

```
m = p[0]+p[1];
```

would add the two values $p[0]$ and $p[1]$ and store the result in m in all active processors. Singular and plural expressions may be mixed to produce a plural result, as the singular expressions are simply 'promoted' to the plural type; this is analogous to the automatic promotion of integers to floating point values:

```
m = p[0]+p[1]*k[0];
```

The promotion of values from singular to plural type is vital; in the expression just given, not only the value of $k[0]$ but the constants 0 and 1 and the address of the plural array p are all singular values.

It is not possible to automatically cast plural values to singular values. There are at least three reasons for this restriction:

- There is not a unique sensible way to combine multiple plural values into a singular value
- Most ways of combining multiple plural values into a singular value are quite slow, and an automatic conversion would hide this inefficiency from the programmer
- Most attempts to use a plural value in a singular context are the result of a programming error

However, the MPL language extension `globalor` uses the MasPar's global or network to convert a plural value to a singular by or'ing all of its bits together. Many library calls exist to find the singular minimum, maximum, sum, product, and boolean and, or and exclusive-or of plural values. By use of the `proc` keyword, a plural value may be extracted from a single processor. This example assigns the singular value of the plural variable b in processor 100 to the singular variable a :

```
int a;
plural int b;
```

```
a = proc[100].b;
```

When passing expressions on the stack for a function call, there are in reality two stacks, one for plural values and one for singular values. In the older MPL programming language, this caused many problems when attempting to pass singular values to plural parameters and vice versa, because there was no way for the language to automatically determine if the types matched; again, this is similar to the problem in K&R C when an attempt is made to pass an integer value into a floating point parameter, causing sometimes silent failures. However, as MPL is based upon ANSI C, function prototypes may be used to cause parameter types to be checked and allow casting to occur where appropriate. The use of prototypes also allows small data types such as chars and shorts to be passed in appropriately sized spaces on the stack, saving both space and time.

MPL also includes a number of predefined variables that contain useful values relating to the architecture of the MasPar. The variables and their meaning are as follows:

- `nproc` contains the number of PEs on the MasPar
- `nxproc` contains the number of PE columns
- `nyproc` contains the number of PE rows
- `lnproc` contains $\log_2 nproc$
- `lnxproc` contains $\log_2 nxproc$
- `lnyproc` contains $\log_2 nyproc$
- `iproc` is a plural value containing each PE's index
- `ixproc` is a plural value containing each PE's index within a row
- `iyproc` is a plural value containing each PE's index within a column

Pointers

One of the most flexible aspects of the C language is its handling of pointers. As we have seen, the MasPar contains two memories in which to store data which are distinguished by the use of the plural keyword when declaring variables. When dealing with pointers, the situation is complicated by the fact that both the memory being pointed to and the pointer itself may be singular or plural. There are four combinations of these possibilities, which are specified as follows:

- Singular pointer pointing to singular memory

This situation is identical to a standard pointer declaration in C:


```
int *a;
```

- Singular pointer pointing to plural memory

This situation is identical to a standard pointer declaration in C, except that the object being pointed to is plural:

```
plural int *a;
```

Because the address of the plural object pointed to by a singular pointer is the same on all PEs, direct addressing is used to access values.

- Plural pointer to plural memory

To declare a plural pointer to a plural object, it is necessary to specify that the pointer is plural as well as the appointed object:

```
plural int *plural a;
```

Because the address of the plural object may be different on every PE, indirect addressing must be used to access the data values. Accessing memory by indirect addressing takes three times as long as by direct addressing, and if the plural address computation must be performed in the PEs, may take even longer.

- Plural pointer to singular memory

A plural pointer to a singular object is specified thus:

```
int *plural a;
```

It would appear that this would allow a form of shared memory to be used, as every processor may access values from the ACU's data address space. Unfortunately, the MasPar's hardware does not support such an operation directly, and any operation using a plural pointer to a singular object is translated into an operation looping over every PE. However, the machine code generated for these operations is more efficient than could be programmed using MPL, making this kind of operation concise and useful.

By declaring pointers to pointers, an unlimited number of combinations may be achieved. Although this can easily cause confusion, the type-checking C performs will usually produce warnings when a combination is used inappropriately.

Conditional execution

The flow of control in an MPL program, or indeed in any parallel programming language, is more complex than in standard C because when a branch-point is reached in which a test is evaluated in parallel, some processors may succeed and some may fail the test. In a MIMD or SPMD machine, this may result in the processors executing different portions of the same program at the same time. In a SIMD machine, because there is only one instruction stream, the alternatives must be executed sequentially. It is possible to create an illusion of conditional execution by disabling processors that fail conditional tests. However, the ACU continues executing all statements inside plural conditionals unless *all* processors have failed a conditional test.

The current set of *enabled* processors is the *active set*. Only enabled processors participate in memory accesses, computation and communication.

There are plural equivalents for all of the C conditional execution statements: *if*, *while*, *do*, *case* and *for*. There is also an additional statement, *all*, which enables all PEs in the following statement or block. There are also plural equivalents of the *break* and *continue* statements. The *?:* operator is similar to an *if* statement, except that it returns a value to an expression.

This example of a plural *if* statement decrements a plural variable iff it is positive and increments it iff it is negative:

```
plural int a;  
int b;  
  
if (a>0) {  
    a--;  
    b++;  
} else if (a<0) {  
    a++;  
    b--;  
}
```

If the value of *a* is negative in some PEs and positive in others, both clauses of the *if* statement will be executed, resulting in *b* being first incremented and then decremented. This can sometimes cause confusion, as in sequential versions of C only one clause in an *if* statement is ever executed.

The operation of the other conditional execution statements is similar to that of the *if* statement; if the conditional expression is plural, per-PE conditional execution is simulated by the manipulation of the active set. The ACU will still execute all singular code inside plural conditional blocks unless the active set is empty, in which case the block may be skipped.

Inter-processor communication

Inter-processor communication on the MasPar MP-1 may be performed by the *xnet*, a two-dimensional mesh; or by the *router*, a crossbar-like connection. Both mechanisms may be used on either the left- or right-hand side of an assignment statement. If the *xnet* or *router* expression is an r-value (i.e. is a non-assigning expression or appears on the right-hand side of an assignment statement), the plural expression is evaluated in the source processor, communicated by the *xnet* or the *router* and returned to every active processor. If the *xnet* or *router* expression is an l-value (i.e. an expression that is assigned to), the assigned values are computed in the active processors, sent to the destination processors where the l-value is computed and set to the received value.

An *xnet* expression contains three parts: a direction, a distance and a plural expression. Because the MasPar incorporates diagonal connections between processors, there are eight simple *xnet* expressions corresponding to the eight compass directions. The distance is a singular integer value indicating how far the message will be communicated.

This example shows how the *xnet* may be used for all active PEs to fetch the value of *a* from their eastern neighbours and assign this value to their own variable *a*:

```
a = xnetE[1].a;
```

This example shows how the *xnet* may be used for all active PEs to store their value of *a* into their eastern neighbours:

```
xnetE[1].a = a;
```

Variants of the *xnet* expression exist for fast copying and pipelining between processors, but these facilities are not used in this thesis.

A *router* expression contains a plural processor index and a plural expression. The semantics of the *router* expression are identical to those of the *xnet* expression, except that the connected processor is specified explicitly as a processor index instead of in a relative way with a direction and a distance.

This example shows how the *router* may be used for all active PEs to fetch the value of *a* from the processor at the same position in a transposed mesh and assign this value to their own variable *a*:

```
a = router[iyproc+ixproc*nyproc].a;
```

If several processors attempt to fetch data from a single processor, the operation will succeed but may take longer. This example shows how the *router* may be used for all active PEs to store their value of *a* into a scattered set of processors :


```
router[(iproc*273)%nproc].a = a;
```

If several processors attempt to send data to a single processor, only one will succeed. The successful processor is deterministic but not defined. It is not possible for a sending processor to detect if other processors are attempting to send data to the same address, which means that each processor sending data to the same address must make the connection in turn and only the last will be recorded. However, it is possible to accumulate multiple sends in many different ways through library calls.

Because the router is implemented in hardware as a crossbar-like switch connecting clusters of sixteen processors, the router expression is translated into a loop in which every processor repeatedly attempts to obtain a router link, sends or fetches data and closes the link until every processor has successfully transferred data. If only a small number of processors are active, only one iteration may be required. If all processors are active, a minimum of sixteen iterations is required; for many common processor permutations, this minimum is achieved. If many processors attempt to connect to a smaller subset of processors, many more iterations may be required; if all processors attempt to connect to the same processor, *nproc* iterations may be required.

Front end vs. back end

It is possible to perform nearly all of one's MasPar programming in MPL, as there are interfaces to input/output, file-systems and indeed many system calls available on the front end workstation. All the examples and utilities used in and produced for this thesis were written in MPL for the DPU, or back end. However, to avoid the limited ACU data memory or to communicate with other processors on the front end, it is possible to pass control back and forth between the back end and the front end using the *callRequest* library call.

Data may be passed between the back end and the front end in many ways. The simplest is to pass information as arguments in the *callRequest* call. Data may be transferred between memory in the front end and either ACU or PE memory using *copyIn*, *copyOut*, *blockIn* and *blockOut* calls. Because the Unix file-system is visible to the MasPar through system calls, data may also be transferred between the Unix file-system and the back end through data files.

Summary

Because MPL is based on a common programming language, ANSI C, many programs may be ported to the MasPar simply by compilation. However, no automatic parallelization is performed, and use of the language extensions must be used to take advantage of any data parallelism in the program. The extensions fall into three broad classes: a memory model which includes plural

data objects, stored in a distributed fashion in every PE; a different model for conditional execution using the concept of active sets to allow conditions and looping to be performed on a SIMD machine; and several inter-processor communication expressions to allow data to be transferred both between PEs and between PEs and the ACU.

The extensions to the language are highly dependent on the MasPar's architecture, and map very closely to it. Although this can make programming a complex task, there can be substantial advantages to efficiency because it is always easy to see where a program is likely to perform slowly.

Because MPL has been based upon the GNU C compiler, a C++ compiler for the MasPar is a possibility in the future. The additional flexibility of C++ may allow virtualization and higher-level language constructs to be built into the language.

2.4 Current data mapping systems

In order to map a data array onto a parallel device, we must find a flexible way of specifying this mapping. Several techniques will be examined, and we will attempt to pull the most useful concepts together under one format. We will make the following assumptions about the mapping:

- All of the data in the data array is mapped to the device
- The mapping is independent of the data contained in the data array, and thus can be expressed as an index mapping
- The data array and the storage device may be treated as multidimensional arrays, which we will refer to as **A** and **D** respectively, with shapes **a** and **d**, as defined in section 2.1.

Because it is often more convenient to express data-independent mappings as index mappings, we will often define mappings in terms of the index spaces of **A** and **D**:

$$\begin{aligned} A &= \text{index}(A) \\ D &= \text{index}(D) \end{aligned}$$

2.4.1 Direct permutation

The most general way of mapping a data array onto a parallel device is to map each data element individually to a different location on the device:

$$D(u_0, \dots, u_{(D)-1}) = A(g(u_0, \dots, u_{(D)-1}))$$

where g is an arbitrary index map:

$$g : D \rightarrow A$$

This allows any data-independent mapping to be expressed from a data array to a device, but unfortunately the mapping specification g may require as much or more storage than the data array, as the number of functions g is large:

$$|\{g : (g : D \rightarrow A)\}| \geq |A|!$$

Clearly a less general approach that does not compromise flexibility too much is required.

2.4.2 Dimension mapping

Rather than mapping data elements directly onto a device, the implicit structuring of regularly gridded data allows the mapping of the dimensions of the data array onto the parallel device.

If both the data set and the storage device are one-dimensional, a straightforward index mapping g from D to A sequentially assigns addresses in D to addresses in A :

$$g : D \rightarrow A$$

$$g(u) = \begin{cases} u & \text{for } u \in A \\ \perp & \text{otherwise} \end{cases}$$

If both the data set and the storage device have the same dimensionality and every dimension of A fits within a dimension of D , the corresponding dimensions of A may be mapped to D . For example:

$$g(u_0, u_1) = (u_0, u_1) \text{ for } (u_0, u_1) \in A, \perp \text{ otherwise}$$

Alternatively, some permutation of the dimensions can be mapped. For example:

$$g(u_1, u_2, u_0) = (u_0, u_1, u_2) \text{ for } (u_0, u_1, u_2) \in A, \perp \text{ otherwise}$$

If the product of the length of several data dimensions is no greater than the length of a device dimension, two or more data dimensions may be mapped into a single device dimension. This is a generalization of the common practice of mapping a multidimensional array into a one-dimensional memory.

For example, a two-dimensional data array can be assigned to a one-dimensional device in two ways, in a row-major ordering:

$$g(u_0 + [a]_0.u_1) = (u_0, u_1) \text{ for } (u_0 + [a]_0.u_1) \in D, \perp \text{ otherwise}$$

or a column major ordering:

$$g(u_1 + [a]_1.u_0) = g(u_0, u_1) \text{ for } (u_1 + [a]_1.u_0) \in D, \perp \text{ otherwise}$$

Similar schemes can be used for mapping multidimensional arrays to multidimensional devices, both of arbitrary dimensionality.

Dimension mapping provides some flexibility, but if there are too many data array dimensions, or a data array dimension is too large for one of the device dimensions, this mapping scheme fails. A different problem occurs if the data dimensions fit within the device dimensions with wasted space; many processors may not be allocated data from the array, and so cannot be utilized in any computation with the data.

The idea of dimension mapping is an important component of the k -Tile format, described in chapter 3.

2.4.3 Data index computations

A scheme which offers more flexibility than dimension mapping and can require less specification than direct permutation is data index computation. This method allows device indices to be computed as functions of the data indices.

As an example, assume we wish to map a 256×256 data array onto a two dimensional parallel device with 1024 processors. If we were to use dimension mapping, three quarters of the processors in the parallel device would be wasted. However, we could specify several different mappings which would use more processors in the processor array, and allow us to specify mappings containing differently shaped pieces in the array too.

One simple mapping (corresponding to one-dimensional hierarchical in Maspar's terminology) treats the data as a one-dimensional array and assigns an equal-sized portion of the array to each processor:

$$\text{Let } d = (64, 1024), a = (256, 256)$$

$$g : \text{index}(d) \rightarrow \text{index}(a)$$

$$g(v_0, v_1) = (v_0 + (v_1 \bmod 4), \lfloor v_1/4 \rfloor) \text{ for all } (v_0, v_1) \in D$$

Another simple mapping splits the input array into 8×8 tiles (corresponding to two-dimensional hierarchical in Maspar's terminology) and assigns them to individual processors thus:

$$g(v_0, v_1) = (v_0 \bmod 8 + 8 \times (v_1 \bmod 8), \lfloor v_0/8 \rfloor + 8 \times \lfloor v_1/8 \rfloor) \text{ for all } (v_0, v_1) \in D$$

Simple expressions could be used to reverse, shift, transpose, permute and otherwise rearrange the data indices to specify a mapping onto a device.

This method is very flexible, and a small expression interpreter could be used to move data onto a parallel device as specified. Unfortunately, there is no real limit to the complexity of expressions that could be represented, and the many different ways of representing similar mappings would limit the

possibility of implementing efficient approaches to taking advantage of any parallelism inherent in the mappings.

Another problem with this representation is that it is moderately difficult for a human to interpret the mapping actually represented by the expressions, and the length of the expressions are likely to grow very large.

2.4.4 Index bit maps

One successful technique for mapping a data array onto a parallel device is the *index bit map*, which, in a slightly modified form, forms the basis for *Parallel Data Transforms* [1, 20, 21]. This technique treats the data indices and the device indices as two one-dimensional vectors of bits, and uses a combination of bit permutations and inversions to define a mapping between the data array and the parallel device.

Once a data array has been mapped to a device, the data may be remapped by the use of *index bit permutation* [1, 13, 14, 20, 21, 27, 28, 56, 70].

Optimal algorithms for single elements stored on mesh-connected processors have been found [56], and these algorithms have been implemented with extensions and optimizations by Flanders for the AMT DAP [59] and Cruz for the *Reconfigurable Processor Array* (RPA) [13, 14]. A fuller description of the index bit map, index bit permutations and associated algorithms for the MasPar is given in chapter 4. Fier mentions that he has derived similar algorithms for the MasPar independently [17, 18], but has not published an account of these results.

There are several disadvantages to the index bit map. Because all operations are specified in terms of operations on bits, the dimensions of both the data array and the device are limited to powers of two. The specification method also requires all remapping operations to be specified as permutations of bits.

2.4.5 Index digit maps

A natural extension of index bit maps is mixed-radix index digit maps. This technique treats both the data indices and the device indices as mixed radix numbers, and uses operations on the digits of these numbers to define a mapping between the data array and the parallel device. A fuller description of the index digit map and associated algorithms is given in chapter 6.

Index digit maps have more flexibility than index bit maps, as the dimensions of the data array and parallel device are not limited to powers of two. However, the specification of mappings is more complex than using the index bit map.

2.4.6 Blip schemes

Boppana and Raghavendra have described a scheme for storing multidimensional arrays with powers-of-two dimensions in the memories of a distributed-memory parallel processor [6]. The emphasis of their paper is for computers with some form of connection network between the processor and the memories, in which it is desirable to be able to access *templates* of the data arrays without network contention. A template T of an $N \times N$ matrix is a set of N element positions including the element at position $(0,0)$. Important templates are row, column, diagonal and block templates. An *affine template* is a set of N element positions where each position is the exclusive-or of a template position with some constant.

By storing data elements in a way defined by a *blip scheme* it is possible to access various templates of a data array without network contention with a variety of interconnection networks. An element (i, j) stored according to a blip scheme is stored in location i of a memory unit with index $i \oplus \pi(j)$, where each bit of $\pi(j)$ is the exclusive-or of some selection of bits of j .

This may be contrasted with index bit permutation (see chapter 4), where each element i of a one-dimensional, or unwrapped, array is stored in location d of a parallel device (d incorporates both processor index and memory address) where $d = \pi(i) \oplus k$, where $\pi(i)$ is a permutation of the bits of i and k is a constant.

When applied for use with a distributed-memory SIMD computer with an interconnection network connecting processors, the blip scheme may be used to transform between mappings in which certain templates are accessible without contention. The blip schemes appear to have much in common with index bit permutation schemes, and may be able to describe a richer class of mappings. However, they are limited to square arrays with power-of-two dimensions, and Boppana and Raghavendra state that further work would be required to systematically obtain the linear permutation schemes $(\pi(j))$ for the templates of interest; as with index bit and index digit maps, the specification technique for an arbitrary remapping is complex.

2.4.7 Ad-hoc approaches

A simple way of describing data mappings is to describe a few mappings explicitly and to give them descriptive names. This approach has been used with some success with the DAP (for mappings with names like *sheet* and *crinkle*), and the MasPar. The MasPar naming scheme illustrates this approach [51]:

One-dimensional hierarchical

The data array is treated as a one-dimensional object and broken into P approximately equal pieces in order to be mapped onto P processors.

One-dimensional cut'n'stack

The data array is treated as a one-dimensional object and data elements are assigned separately to each processor. When all P processors have been assigned one data element, the process is repeated until the data array is distributed fully.

Two-dimensional hierarchical

The data array is broken into a set of P two-dimensional tiles each of which is assigned to a different processor. This corresponds to the DAP crinkle mapping.

Two-dimensional cut'n'stack

The data array is broken into two-dimensional tiles where each tile is the same size as the processor array, and the tiles are placed to cover the processor array. This corresponds to the DAP sheet mapping.

One problem with this method is the lack of flexibility that is offered. The four mappings described here are sufficient for many algorithms on two-dimensional images, but they do not address the greater complexity of data arrays containing three or more dimensions.

If all the different types of mappings are allowed, many distinct library calls for inter-converting between the mappings are required. The practical solution to this problem has been to restrict allowable remapping operations to only a subset of the possible operations. For example, in the MasPar image processing library [51], all the data remapping routines are to and from the two-dimensional hierarchical format.

These mappings do not allow the specification of any more geometrically oriented remappings, such as transposition of axes or axis reversals, so library calls must also be included to perform these operations also.

2.4.8 High Performance FORTRAN

In an attempt to give the programmer more power in specifying and redistributing data on a parallel processor array, the specification of High Performance Fortran (HPF) gives a powerful set of compiler directives designed to allow many different mappings to be specified. The approach is similar to the ad-hoc approaches in that descriptive names such as **block**, and **cyclic** are used to specify different types of mappings, but it has more flexibility because these modifiers may be applied to individual dimensions of a multidimensional array.

Although the mechanism for mapping specification appears quite different from the k -Tile format, the set of mappings that can be specified is similar. The k -Tile format is capable of specifying all the possible regular mapping

relationships between data arrays that HPF is capable of, and in many respects the k -Tile format is more powerful. A more complete comparison of the two schemes and a description of methods for their inter-conversion is contained in section 7.1.

However, the most significant limitation on HPF is that it is an integral part of just one programming language, and it remains to be seen how well these concepts will transplant to other languages such as C.

2.5 Summary

To provide a context in which to work in the rest of this thesis, this chapter defines multidimensional data arrays and the problem of mapping these arrays to one- or multidimensional storage devices. We show how the mapping problem has relevance to the efficiency of data access both within and between devices, the efficiency of algorithms and the portability of applications between different architectures. It is suggested that current approaches have shortcomings in portability, flexibility and ease of use. Thus, a consistent framework for data mapping would both provide a useful tool for programmers on a wide variety of architectures and for a large class of problems, and also provide a powerful framework on which to build tools which would allow the data mapping problem to be hidden from the programmer, such as compilers and data processing libraries.

Parallel architectures are very suitable for processing large multidimensional data sets, and much of the work in this thesis is based on development for the MasPar MP-1. In this chapter gave a broad overview of the properties of parallel architectures and outlines the structure of three massively parallel SIMD architectures: the AMT DAP 500 [59], the CM-2 [37, 67] and the MasPar MP-1 and MP-2 [4, 50]. Particular emphasis is given to the MasPar MP-1, and the MPL programming language is summarised to provide a vehicle for parallel algorithm descriptions in later chapters.

We also examine three broad classes of current data mapping systems; software techniques for both sequential and parallel architectures, and hardware techniques. Central to all of these systems is index digit permutation. Many data mapping systems are related to this approach:

- Lilleyman's crossbar switch to allow arbitrary index bit permutations for the Fast Fourier Transform in fast hardware
- Newman's tiling memory-management unit
- Nassimi and Sahni's optimal routing algorithm for mesh-connected computers using index-bit permutation

- Flander's substantial extensions to Nassimi and Sahni's work in Parallel Data Transforms for the AMT DAP
- Cruz' implementation of PDTs for the RPA [13, 14]
- Boppana and Raghavendra's Blip schemes [6]
- the data mapping directives of high-performance Fortran [36]
- Fraser's index-bit permutation techniques for scan-line algorithms and the Fast Fourier Transform [27, 28, 30, 31]
- Van Heel's transposition method for large images on disk
- Fraser's Multidimensional Tile format.

The value of being able to describe all these techniques within a single framework is emphasized.

Chapter 3

The k -Tile format

In the last chapter we examined many techniques currently being used for mapping multidimensional data sets to parallel computers, and showed the need for a flexible format for specifying these data mappings.

Specifying data mappings is not a trivial task. To simplify the writing of programs to access data, it is common to use the simplest data mapping strategies, which usually correspond to the structure of a disk file or of a particular computation. When dealing with large data sets, the use of an inappropriate mapping may greatly increase data communication costs, whether between devices or between processors within a device.

When trying to use one or more data mappings for a problem, there are several possible sources of inefficiency or unnecessary effort. When writing code for different problems with similar data mapping requirements there is the potential for duplication of effort, while attempting to integrate problems with different data mapping requirements may entail complex programming tasks.

A general framework for describing and manipulating data mappings would alleviate these problems by providing a consistent mechanism for specifying data mappings. Such a mechanism could be integrated with a system for performing data remapping using general but efficient algorithms.

Many of the mapping specification methods outlined in the previous chapter suffer from both a lack of flexibility in specification and a separation of the mapping method from the intuitive ideas of data and device dimensions.

In this chapter we describe the k -Tile format, which is a general framework for describing the mapping of multidimensional data arrays to a variety of storage devices. The k -Tile format addresses the mapping problem of chapter 1 for multidimensional arrays on parallel devices.

Two k -Tile formats implicitly specify a data remapping, and thus the k -Tile format also provides a means of specifying data remappings. Because this method is descriptive and does not specify how a data remapping is to be performed, algorithms must be developed for this task. This means of

specifying and performing data remappings is called *Parallel Mapping Functions* (PMFs), and later chapters show how the *k*-Tile format and PMFs may be implemented and used for data mapping and remapping applications. [7, 22, 25, 29, 55, 64, 66].

3.1 The Basic *k*-Tile Format

The *k*-Tile format defines a mapping between three multidimensional array spaces (see section 2.1): the *data space*, the *device space* and the *k*-Tile space. Each array space has the same element data type. The *k* in the name emphasizes the multi-dimensional nature of the tiles mapped through the *k*-Tile space.

The data space is a multidimensional space containing the data array. The device space contains the mapping of the image data to a physical storage device, the device array. The *k*-Tile space is an abstract space linking the image space and the device space.

3.1.1 The Data Type

Because the device array must be represented on a physical storage device, we assume that the data array data type contains a finite number of elements. Similarly, to allow computation of physical addresses from array indices, each element of the data type is represented in a fixed number of bytes. Thus, the data type of any multidimensional array may be treated as another dimension of the array with length the size of the data type in bytes.

In future we will assume that the data type of all multidimensional arrays is *byte* and treat the true data type as the first dimension in the array. This allows us to ignore the actual data type of an array and define mappings between any multidimensional array spaces.

3.1.2 The Data Space

The data space represents a physical or synthetic coordinate system in which the data was sampled or generated, and the dimensions of the data space represent the intrinsic dimensions and dimensionality of the data.

The dimensions in the data space generally offer some physical meaning; examples include regular samples from a map grid, three-dimensional samples from a CT scan at millimeter spacings or two spatial dimensions of a scanned photograph. Different dimensions will often have quite different physical interpretations; a single image may contain one dimension along which to store the pixel data type, three spatial dimensions, a temporal dimension and a spectral dimension to represent colour. As discussed in section 2.1.3, data dimensions

may be stored either implicitly or explicitly; for the purposes of remapping, we manipulate only the implicit dimensions of the data space and treat the elements as uninterpreted objects.

Because the data type of the data space is always treated as *byte*, the data space is completely specified by its shape, *a*. A few examples illustrate some data spaces:

- A single-channel 512×1024 byte image: $[512, 1024]$
- A three-channel 512×1024 image of 32-bit integers: $[4, 512, 1024, 3]$
- A 512×512 slice from a 3-dimensional image: $[512, 1, 512]$

3.1.3 The Device Space

The device space represents a part of the storage component of a device the data space will be mapped to. In a distributed memory parallel processor, dimension 0 corresponds to the computer's memory and subsequent dimensions to dimensions in the processor array. In a display device, dimension 0 may correspond to the *x*-axis of the display. In general, communication speed along device axes decreases with increasing device dimension number; the index of the highest device dimension may select different physical devices.

It is not necessary to specify or fill all the physical dimensions in the device, and it may be possible to combine certain dimensions in order to treat the device as a lower-dimensional device with longer dimensions.

The maximum dimensionality of the device space is determined by the architecture of the device: a serial computer or disk file has a device space containing at most one dimension; a "striped" disk file may have two dimensions; a 2d mesh machine a device space of at most three dimensions; and hypercube architectures such as the CM-2 [67] may have many more.

In some cases, the ability to treat a processor array as lower-dimensional allows the architecture to be optimized for the device space; as an example, when dealing with dimensions which are powers of two, by using a Gray-code indexing scheme the CM-2 may be configured to have local connectivity for any number of dimensions up to the dimensionality of its hypercube.

A few examples illustrate the some device spaces:

- A 1 megabyte file on disk: $[1048576]$
- A 67 byte array stored on each processor in the 128×128 mesh of a MasPar MP-1: $[67, 128, 128]$
- A 64 byte array stored on a CM-2 configured as a four-dimensional $16 \times 16 \times 16 \times 16$ processor array: $[64, 16, 16, 16, 16]$

- A 7 byte array stored on the first 7 processor along the Y -axis of a mesh:
[7, 1, 7]

3.1.4 The k -Tile Space

In section 2.4.2 it was shown how to define a data-to-device mapping by mapping data dimensions directly to device dimensions. The k -Tile format is based on a similar idea, but gains more flexibility than the dimension mapping method by the use of an intermediate higher-dimensional k -Tile space. A dimension mapping is used to map the k -Tile space onto both the data and the device spaces, thus defining a mapping from the data space to the device space. As with the data and device spaces, the k -Tile space is completely specified by its shape, k .

3.1.5 The k -Tile format, an overview

A k -Tile format f_K is a data mapping between a data space, with shape a , and a device space, with shape d . Using the notation in section 2.1:

$$f_K : \mathbb{M}_a \rightarrow \mathbb{M}_d.$$

The mapping f_K is data-independent, and may thus be defined by an index map g_K between the index spaces of d and a :

$$g_K : \text{index}(d) \rightarrow \text{index}(a)$$

$$f_K(A)(v) = A(g_K(v)) \text{ for all } A \in \mathbb{M}_a \text{ and } v \in \text{index}(d)$$

In turn, g_K is defined as two index mappings, g_{KA} and g_{DK} , passing through the k -Tile index space, with shape k :

$$g_{DK} : \text{index}(d) \rightarrow \text{index}(k)$$

$$g_{KA} : \text{index}(k) \rightarrow \text{index}(a)$$

$$g_K = g_{KA} \circ g_{DK}$$

The index mapping g_{DK} is one-to-one and onto, thus we may define g_{DK} in terms of its inverse, g_{KD} . Both g_{KD} and g_{KA} are defined as k -Tile mappings.

3.1.6 The k -Tile mapping

A k -Tile mapping from a k -Tile index space K with shape k and another index space Z with shape z is specified by mapping at least one k -Tile dimension into each dimension in Z . A group of k -Tile dimensions are mapped into a single Z dimension as if the k -Tile dimensions were a multidimensional array

mapped into a one-dimensional memory. A k -Tile mapping is also identical to the dimension mapping described in section 2.4.2.

Before defining this fully, we show some examples of k -Tile mappings. Firstly we show the two ways a 2d k -Tile index space K can be mapped to a 1d index space Z . These two examples map the dimensions of K into Z in column and row major order respectively. Letting

$$\mathbf{k} = (3, 4), K = \text{index}(\mathbf{k})$$

$$\mathbf{z} = (12), Z = \text{index}(\mathbf{z})$$

an example of a k -Tile mapping m from K to Z is

$$g : K \rightarrow Z$$

$$g(a_0, a_1) = (a_1 + k_1 \cdot a_0) = (a_1 + 4 \cdot a_0).$$

Another example of a k -Tile mapping for the same K and Z is

$$g : K \rightarrow Z$$

$$g(a_0, a_1) = (a_0 + k_0 \cdot a_1) = (a_0 + 3 \cdot a_1).$$

The data mapping associated with the second example is

$$f : \mathbb{M}_Z \rightarrow \mathbb{M}_K$$

$$f(\mathbf{A})(a_0, a_1) = \mathbf{A}(g(a_0, a_1)) = \mathbf{A}(a_0 + 3 \cdot a_1).$$

For the mapping to be one-to-one and onto, the product of the lengths of each group of k -Tile dimensions must be identical to the length of the associated dimensions of Z .

We do not restrict the assignment of k -Tile dimensions to dimensions of Z to be either row or column major ordering. Thus, in a 3d k -Tile space K mapped to a 1d space Z , there are six possible k -Tile mappings. Letting

$$\mathbf{k} = (k_0, k_1, k_2), K = \text{index}(\mathbf{k})$$

$$\mathbf{z} = (z), Z = \text{index}(\mathbf{z}),$$

the six possible k -Tile mappings from K to Z are:

$$\begin{aligned} g_1(a_0, a_1, a_2) &= a_0 + k_0 \cdot (a_1 + k_1 \cdot a_2) & g_4(a_0, a_1, a_2) &= a_0 + k_0 \cdot (a_2 + k_2 \cdot a_1) \\ g_2(a_0, a_1, a_2) &= a_1 + k_1 \cdot (a_0 + k_0 \cdot a_2) & g_5(a_0, a_1, a_2) &= a_1 + k_1 \cdot (a_2 + k_2 \cdot a_0) \\ g_3(a_0, a_1, a_2) &= a_2 + k_2 \cdot (a_0 + k_0 \cdot a_1) & g_6(a_0, a_1, a_2) &= a_2 + k_2 \cdot (a_1 + k_1 \cdot a_0). \end{aligned}$$

Thus, when specifying a k -Tile mapping we must provide a means to express any valid permutation of k -Tile dimensions to be assigned to each dimension of Z .

Some examples of possible k -Tile mappings to 2d spaces are:

- A mapping from a 12×12 k -Tile space to a 12×12 destination space

$$\mathbf{k} = (12, 12), K = \text{index}(\mathbf{k})$$

$$\mathbf{z} = (12, 12), Z = \text{index}(\mathbf{z})$$

$$g : K \rightarrow Z$$

$$g(a_0, a_1) = (a_1, a_0)$$

- A mapping from a $3 \times 3 \times 4 \times 4$ k -Tile space to a 12×12 destination space. k -Tile dimensions 0 and 2 are mapped to dimension 0 of Z , and k -Tile dimensions 3 and 1 are mapped to dimension 1 of Z

$$\mathbf{k} = (3, 3, 4, 4), K = \text{index}(\mathbf{k})$$

$$\mathbf{z} = (12, 12), Z = \text{index}(\mathbf{z})$$

$$g : K \rightarrow Z$$

$$g(a_0, a_1, a_2, a_3) = (a_0 + a_2 \cdot k_0, a_3 + a_1 \cdot k_3)$$

- A mapping from a $3 \times 12 \times 4$ k -Tile space to a 12×12 destination space. k -Tile dimensions 0 and 2 are mapped to dimension 0 of Z , and k -Tile dimension 1 is mapped to dimension 1 of Z

$$\mathbf{k} = (3, 12, 4), K = \text{index}(\mathbf{k})$$

$$\mathbf{z} = (12, 12), Z = \text{index}(\mathbf{z})$$

$$g : K \rightarrow Z$$

$$g(a_0, a_1, a_2) = (a_0 + a_2 \cdot k_0, a_1)$$

3.1.7 Specifying a k -Tile mapping

The assignment of dimensions of a k -Tile index space to dimensions of an arbitrary index space Z can be achieved by specifying a vector \mathbf{m} of length $\langle K \rangle$ which assigns k -Tile dimensions to dimensions of Z , from the lowest significant position in dimension 0 of Z upwards.

$$\mathbf{m} \in \mathbb{N}^{\langle K \rangle}, \mathbf{m} = (m_0, \dots, m_{\langle K \rangle - 1})$$

To provide a valid specification for a k -Tile mapping, \mathbf{m} must satisfy two conditions

- No dimension of K appears more than once in \mathbf{m} ; in other words, \mathbf{m} is a permutation of $[0, \dots, (\langle K \rangle - 1)]$. This ensures that every k -Tile dimension is only mapped once into the dimensions of Z . Equivalently, every k -Tile dimension appears exactly once.

- The product of the length of every group of *k*-Tile dimensions assigned to a dimension of *Z* must be equal to the length of that dimension.

To check the second condition we may calculate a vector *c* which sequentially assigns sections of the mapping vector *m* to dimensions of *Z*. *m* is a valid mapping vector for the *k*-Tile index space *K* and the index space *Z* iff

$$\exists \mathbf{c} \in \mathbb{N}^{\langle Z \rangle + 1}, \mathbf{c} = (c_0, \dots, c_{\langle Z \rangle})$$

such that for all *i*, $0 \leq i < \langle Z \rangle$

$$c_{i+1} \geq c_i \quad (3.1)$$

$$z_i = \prod_{j=c_i}^{c_{(i+1)}-1} k_{m_j} \quad (3.2)$$

$$c_{\langle Z \rangle} = \langle K \rangle \quad (3.3)$$

Equation 3.1 ensures that the elements of the mapping vector *m* are used only once to map the dimensions of *K* into *Z*. Equation 3.2 ensures that the assigned *k*-Tile dimensions fit snugly within the associated dimensions of *Z*. Equation 3.3 ensures that all the *k*-Tile dimensions are assigned to dimensions of *Z*.

These conditions ensure that the *k*-Tile mapping provides a map from *K* to every element of *Z*, and that every element of *K* maps to an element of *Z*.

From the three vectors *k*, *z* and *m*, we may define a *k*-Tile mapping *g* from *K* to *Z*. Letting

$$p \in \mathbb{N}, \mathbf{z} \in \mathbb{N}^p, Z = \text{index}(\mathbf{z}), \mathbf{z} = (z_0, \dots, z_{p-1})$$

$$q \in \mathbb{N}, \mathbf{k} \in \mathbb{N}^q, K = \text{index}(\mathbf{k}), \mathbf{k} = (k_0, \dots, k_{q-1})$$

$$\mathbf{m} \in \mathbb{N}^q, \mathbf{m} = (m_0, \dots, m_{q-1})$$

If *m* satisfies the conditions stated in section 3.1.7, a vector *c* may be derived as described in section 3.1.7,

$$\mathbf{c} \in \mathbb{N}^{p+1}$$

and the *k*-Tile mapping may be defined

$$\text{kmap}(\mathbf{k}, \mathbf{z}, \mathbf{m}) \triangleq g : K \rightarrow Z$$

$$g(w_0, \dots, w_{q-1}) = (u_0, \dots, u_{p-1})$$

$$\text{where } u_i = \sum_{l=c_i}^{c_{(i+1)}-1} w_l \cdot \prod_{j=c_i}^{l-1} k_{m_j}$$

For example, the following selection of values for the vectors \mathbf{k} , \mathbf{z} and \mathbf{m} , and a derived value for \mathbf{c} , define a valid k -Tile mapping:

$$\begin{aligned}\mathbf{z} &= (128, 64) \\ \mathbf{c} &= (0, 3, 5) \\ \mathbf{m} &= (2, 3, 0, 1, 4) \\ \mathbf{k} &= (2, 2, 8, 8, 32)\end{aligned}$$

3.1.8 The implicit k -Tile mapping

The k -Tile format is based on two k -Tile mappings from the same k -Tile index space to the data and device index spaces. Because the k -Tile space is internal to the k -Tile format and any valid permutations of k -Tile dimensions may be assigned to the dimensions of the data and device spaces, the numbering of the k -Tile dimensions is arbitrary.

To simplify the specification of the k -Tile format and to limit the number of k -Tile formats specifying identical mappings, we define the *implicit k -Tile mapping*. The implicit k -Tile mapping differs from the k -Tile mapping in that k -Tile dimensions are assigned implicitly to the dimensions of the destination space in sequential order, thus a mapping vector \mathbf{m} is not required. If one k -Tile mapping in the k -Tile format is implicit, arbitrary assignment of the k -Tile dimensions to the data and device index spaces may still be achieved by renumbering the k -Tile dimensions.

Given a k -Tile index space with shape \mathbf{k} and an index space Z with shape \mathbf{z} ,

$$p \in \mathbb{N}, \mathbf{z} \in \mathbb{N}^p, Z = \text{index}(\mathbf{z}), \mathbf{z} = (z_0, \dots, z_{p-1})$$

$$q \in \mathbb{N}, \mathbf{k} \in \mathbb{N}^q, K = \text{index}(\mathbf{k}), \mathbf{k} = (k_0, \dots, k_{q-1})$$

and assuming that a vector \mathbf{c} exists assigning k -Tile dimensions to dimensions of Z

$$\exists \mathbf{c} \in \mathbb{N}^{(K)+1}, \mathbf{c} = (c_0, \dots, c_{\langle K \rangle})$$

such that for all $i, 0 \leq i < \langle K \rangle$

$$c_{i+1} \geq c_i \tag{3.4}$$

$$z_i = \prod_{j=c_i}^{c_{(i+1)}-1} k_j \tag{3.5}$$

$$c_{\langle Z \rangle} = \langle K \rangle \tag{3.6}$$

the implicit k -Tile mapping may be defined

$$\begin{aligned}\text{ikmap}(\mathbf{k}, \mathbf{z}) &\triangleq g : K \rightarrow Z \\ g(w_0, \dots, w_{q-1}) &= (u_0, \dots, u_{q-1})\end{aligned}$$

$$\text{where } u_i = \sum_{l=c_i}^{c_{(i+1)}-1} w_l \cdot \prod_{j=c_i}^{l-1} k_j$$

Alternatively, the implicit k -Tile mapping may also be defined

$$\text{ikmap}(\mathbf{k}, \mathbf{z}) \triangleq \text{kmap}(\mathbf{k}, \mathbf{z}, (0, \dots, \langle \mathbf{k} \rangle - 1))$$

For example, the following selection of values for the vectors \mathbf{k} and \mathbf{z} , and a derived value for \mathbf{c} , define a valid implicit k -Tile mapping:

$$\begin{aligned} \mathbf{z} &= (128, 64) \\ \mathbf{c} &= (0, 3, 5) \\ \mathbf{k} &= (8, 8, 2, 2, 32) \end{aligned}$$

3.1.9 The inverse k -Tile mapping

Letting

$$g = \text{kmap}(\mathbf{k}, \mathbf{z}, \mathbf{m}) : \text{index}(\mathbf{k}) \rightarrow \text{index}(\mathbf{z})$$

We define the inverse k -Tile mapping

$$\begin{aligned} \text{kmap}^{-1}(\mathbf{k}, \mathbf{z}, \mathbf{m}) &\triangleq g^{-1} : \text{index}(\mathbf{z}) \rightarrow \text{index}(\mathbf{k}), \text{ where} \\ g(\mathbf{w}) &= \mathbf{u} \Rightarrow g^{-1}(\mathbf{u}) = \mathbf{w} \text{ for all } \mathbf{w} \in \text{index}(\mathbf{k}) \end{aligned}$$

Because the k -Tile mapping is one-to-one and onto, g^{-1} is uniquely defined everywhere in $\text{index}(\mathbf{z})$.

3.1.10 The k -Tile format, a definition

The k -Tile format is defined by an inverse k -Tile map from the device index space D to the k -Tile index space K and an implicit k -Tile map from the k -Tile index space K to the data index space A .

$$p \in \mathbb{N}, \mathbf{a} \in \mathbb{N}^p, A = \text{index}(\mathbf{a})$$

$$q \in \mathbb{N}, \mathbf{k} \in \mathbb{N}^q, K = \text{index}(\mathbf{k})$$

$$\mathbf{m} \in \mathbb{N}^q, \mathbf{m} = (m_0, \dots, m_{q-1})$$

$$r \in \mathbb{N}, \mathbf{d} \in \mathbb{N}^r, D = \text{index}(\mathbf{d})$$

Assuming \mathbf{a} , \mathbf{k} , \mathbf{m} and \mathbf{d} satisfy the conditions specified in section 3.1.7 to define k -Tile mappings from K to A and D respectively, we may define a

k -Tile index format from D to A

$$\begin{aligned} \text{ktileindex}(\mathbf{a}, \mathbf{k}, \mathbf{m}, \mathbf{d}) &\triangleq g_K : D \rightarrow A, \text{ where} \\ g_K &= g_{KA} \circ g_{DK} \\ g_{DK} &: D \rightarrow K \\ g_{DK} &= \text{kmap}^{-1}(\mathbf{k}, \mathbf{d}, \mathbf{m}) \\ g_{KA} &: K \rightarrow A \\ g_{KA} &= \text{ikmap}(\mathbf{k}, \mathbf{a}) \end{aligned}$$

The k -Tile format is the corresponding data mapping defined by the index mapping g_K

$$\begin{aligned} \text{ktile}(\mathbf{a}, \mathbf{k}, \mathbf{m}, \mathbf{d}) &\triangleq f_K : \mathbb{M}_{\mathbf{a}} \rightarrow \mathbb{M}_{\mathbf{d}} \\ f_K(\mathbf{A})(\mathbf{v}) &= \mathbf{A}(g_K(\mathbf{v})) \text{ for all } \mathbf{A} \in \mathbb{M}_{\mathbf{a}}, \mathbf{v} \in \text{index}(\mathbf{d}) \end{aligned}$$

3.1.11 A basic k -Tile summary

A k -Tile format is described by four vectors, \mathbf{a} , \mathbf{k} , \mathbf{m} and \mathbf{d} .

- \mathbf{a} The shape of the data index space
- \mathbf{k} The shape of the k -Tile index space
- \mathbf{m} The mapping between the k -Tile and device spaces
- \mathbf{d} The shape of the device space

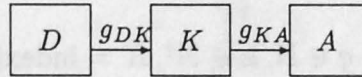
These vectors provide a specification for two index mappings g_{DK} and g_{KA} .

g_{DK} is an inverse k -Tile mapping between D and K , with the assignment of k -Tile dimensions to device dimensions specified by the vector \mathbf{m} . g_{KA} is an implicit k -Tile mapping between K and A .

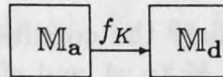
$$\begin{aligned} g_{DK} &= \text{kmap}^{-1}(\mathbf{k}, \mathbf{d}, \mathbf{m}) \\ g_{KA} &= \text{ikmap}(\mathbf{k}, \mathbf{d}, \mathbf{m}) \end{aligned}$$

Composing these two mappings defines an index mapping g_K between D and A ,

$$g_K = g_{KA} \circ g_{DK}$$



g_K may be directly transformed into a data mapping f_K from $\mathbb{M}_{\mathbf{a}}$ to $\mathbb{M}_{\mathbf{d}}$



3.1.12 Example mappings

To clarify the above description we show several mappings specified using the k -Tile format. The numbers shown in the two spaces represent data elements mapped from the data to the device space. A column-major ordering of these numbers has been used throughout for consistency of description; a row-major ordering could have been used instead. The specifications for a , k , m and d are linear arrays, but for clarity ';' is used between elements where the dimension changes in an associated space. In k , a semicolon indicates an increment in data dimension, and in m a semicolon indicates an increment in device dimension. Semicolons are used between all dimensions in a and d .

Simple 1d mapping

This k -Tile format defines a simple mapping between a one-dimensional image containing seven pixels and a one-dimensional storage device containing seven storage locations:

k -Tile	A	D
a : [7]		
k : [7]		
m : [0]	0 1 2 3 4 5 6	0 1 2 3 4 5 6
d : [7]		

Column-major ordering

This k -Tile format defines a column-major mapping between a 2×3 image and a one-dimensional storage device:

k -Tile	A	D
a : [3; 2]	$\leftarrow 0 \rightarrow$	
k : [3; 2]	\uparrow	
m : [0, 1]	0 1 2	0 1 2 3 4 5
d : [6]	3 4 5	
	\downarrow	

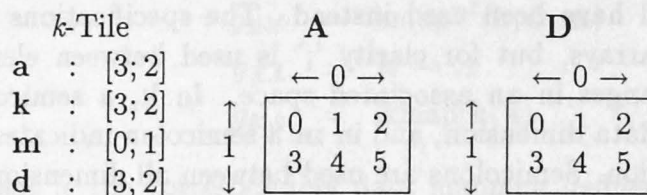
Row-major ordering

This k -Tile format defines a row-major mapping between a 2×3 image and a one-dimensional storage device:

k -Tile	A	D
a : [3; 2]	$\leftarrow 0 \rightarrow$	
k : [3; 2]	\uparrow	
m : [1, 0]	0 1 2	0 3 1 4 2 5
d : [6]	3 4 5	
	\downarrow	

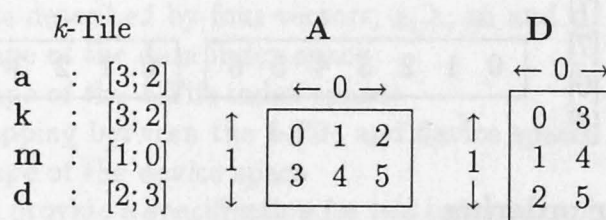
Simple 2d mapping

This k -Tile format defines a scan mapping between a 2×3 image and a two-dimensional storage device:



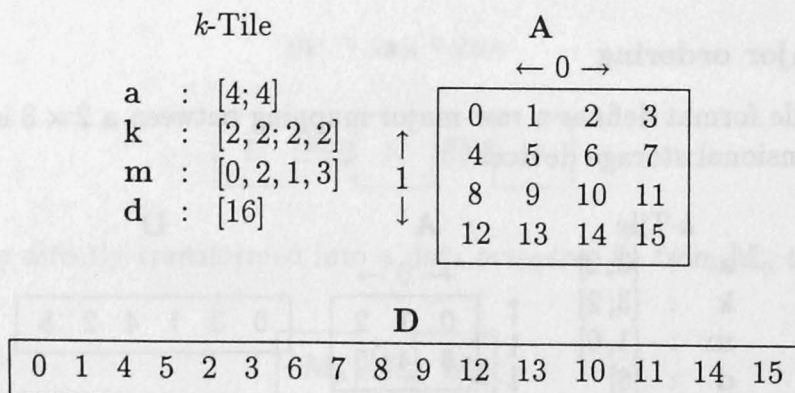
Transposed 2d mapping

This k -Tile format maps the image in the previous mapping transposed to the device.



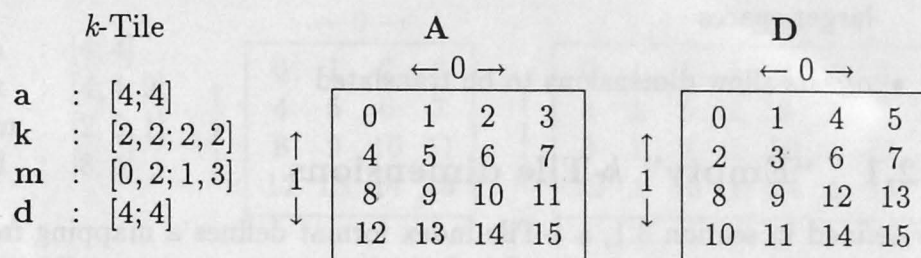
Tiled mapping to 1d device

This k -Tile format maps a 4×4 image to a one-dimensional device as a 2×2 group of 2×2 tiles.

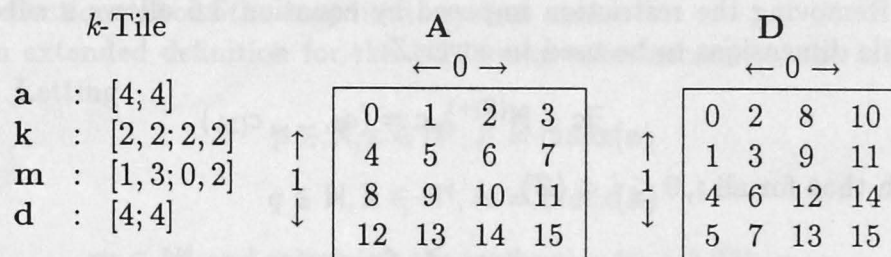


Tiled mapping to 2d device (2d hierarchical or crinkle mapping)

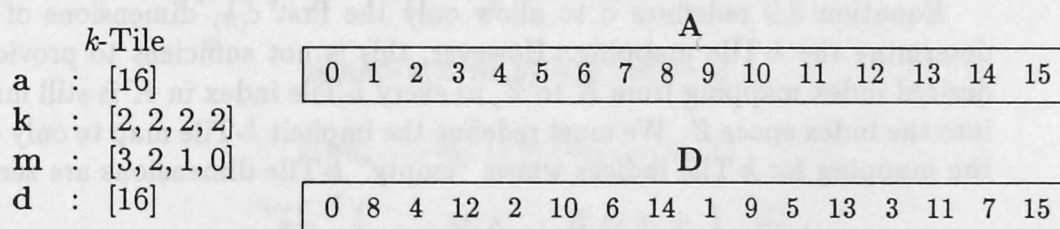
This *k*-Tile format maps a 4×4 image to a two-dimensional device as a 2×2 group of 2×2 tiles.

**Tiled mapping to 2d device (2d cut'n'stack or sheet mapping)**

This *k*-Tile format maps a 4×4 image to a one-dimensional device with a 2×2 group of 2×2 tiles stacked along the memory dimension.

**Index-bit-reversed mapping to 1d device**

This *k*-Tile format maps a 16 element image to a one-dimensional device in index-bit-reversed order, suitable for the application of a radix 2 FFT [11].



3.2 Extending the *k*-Tile format

The basic *k*-Tile format is a flexible and intuitive data mapping specification technique. However, there are many regular mappings it cannot specify. By adding extra concepts the basic *k*-Tile format may be extended to allow a wider range of useful mappings to be specified:

- *empty k*-Tile dimensions allow the declaration of empty storage locations on the device

- *sense* indicators allow the storage order of the k -Tile dimensions to be reversed
- *templates* allow the image, k -Tile and device spaces to be padded into larger spaces
- *offsets* allow dimensions to be translated

3.2.1 “Empty” k -Tile dimensions

As defined in section 3.1, a k -Tile index format defines a mapping from D to A that is one-to-one and onto. Similarly, the two component k -Tile mappings are also defined to be one-to-one and onto.

By modifying the definition of the implicit k -Tile mapping in section 3.1.8, it is possible to define an index mapping from a proper subset of K to an index space Z . This allows elements of the device array to be declared as undefined.

Removing the restriction imposed by equation 3.6 allows a subset of the k -Tile dimensions to be used to cover Z :

$$\exists c \in \mathbb{N}^{\langle Z \rangle + 1}, c = (c_0, \dots, c_{\langle Z \rangle})$$

such that for all $i, 0 \leq i < \langle Z \rangle$

$$c_{i+1} \geq c_i \tag{3.7}$$

$$z_i = \prod_{j=c_i}^{c_{i+1}-1} k_j \tag{3.8}$$

$$c_{\langle Z \rangle} \leq \langle K \rangle \tag{3.9}$$

Equation 3.9 redefines c to allow only the first $c_{\langle Z \rangle}$ dimensions of K to determine the k -Tile mapping. However, this is not sufficient to provide the desired index mapping from K to Z , as every k -Tile index in K is still mapped into the index space Z . We must redefine the implicit k -Tile map to only define the mapping for k -Tile indices whose “empty” k -Tile dimensions are zero:

$$\text{ikmap}(k, z) \triangleq g : K \rightarrow Z \cup \{\perp\}$$

$$g(w_0, \dots, w_{q-1}) = \begin{cases} \perp & \text{if } w_i > 0 \text{ for some } i \geq c_{\langle Z \rangle} \\ (u_0, \dots, u_{q-1}) & \text{otherwise} \end{cases}$$

$$\text{where } u_i = \sum_{l=c_i}^{c_{i+1}-1} w_l \cdot \prod_{j=c_i}^{l-1} k_j$$

The following example defines a k -Tile format between a 4×4 image and a two-dimensional storage device, with the k -Tile space larger than the data

space to allow empty space to be included between elements stored on the device. \perp represents undefined, or empty, storage locations on the device:

<i>k</i> -Tile		A		D
		$\leftarrow 0 \rightarrow$		$\leftarrow 0 \rightarrow$
a : [4; 4]		0 1 2 3		0 \perp 1 \perp 2 \perp 3 \perp
k : [4; 4; 2]	\uparrow	4 5 6 7	\uparrow	4 \perp 5 \perp 6 \perp 7 \perp
m : [2; 0; 1]	1	8 9 10 11	1	8 \perp 9 \perp 10 \perp 11 \perp
d : [8; 4]	\downarrow	12 13 14 15	\downarrow	12 \perp 13 \perp 14 \perp 15 \perp

3.2.2 Sense indicator

A “+” or “-” sense indicator for each *k*-Tile dimension may be used to reverse the order of storage of that *k*-Tile dimension on the device. This is useful for specifying geometrical transformations such as rotations by multiples of 90° and reflections about the coordinate system axes.

An extended definition for the *k*-Tile map takes account of the sense indicator. Letting

$$p \in \mathbb{N}, z \in \mathbb{N}^p, Z = \text{index}(z)$$

$$q \in \mathbb{N}, k \in \mathbb{N}^q, K = \text{index}(k)$$

$m \in \mathbb{N}^q$ and satisfying the properties for a *k*-Tile map

$c \in \mathbb{N}^{p+1}$ as described in section 3.1.7

$$s \in \{+, -\}^q, s = (s_0, \dots, s_{q-1})$$

The *k*-Tile mapping *g* with sense may be defined

$$\text{skmap}(k, z, m, s) \triangleq g : K \rightarrow Z$$

$$g(w_0, \dots, w_{q-1}) = (u_0, \dots, u_{p-1})$$

$$u_i = \sum_{l=c_i}^{c_{(i+1)}-1} \left\{ \begin{array}{ll} w_j & \text{if } s_j = + \\ k_j - w_j - 1 & \text{if } s_j = - \end{array} \right\} \cdot \prod_{j=c_i}^{l-1} k_{m_j}$$

Because inverting the sense of a data or device dimension is equivalent to inverting the sense of the associated *k*-Tile dimensions, sense indicators for the data and device dimensions are not required.

The *k*-Tile mapping with sense indication is included in the *k*-Tile format by adding an extra field, *s*, and replacing the inverse *k*-Tile mapping from *D* to *K* with an inverse *k*-Tile mapping from *D* to *K* incorporating sense. The following examples using the sense indicator shows four rotations of a 4 × 4 image. The *k*-Tile sense indicator is labelled *s*. A top left origin is assumed:

0° rotation:

k -Tile		A		D						
		$\leftarrow 0 \rightarrow$		$\leftarrow 0 \rightarrow$						
a : [4;4]	\uparrow 1 \downarrow	0	1	2	3	\uparrow 1 \downarrow	0	1	2	3
k : [4;4]		4	5	6	7		4	5	6	7
s : [+;+]		8	9	10	11		8	9	10	11
m : [0;1]		12	13	14	15		12	13	14	15
d : [4;4]										

90° clockwise rotation:

k -Tile		A		D						
		$\leftarrow 0 \rightarrow$		$\leftarrow 0 \rightarrow$						
a : [4;4]	\uparrow 1 \downarrow	0	1	2	3	\uparrow 1 \downarrow	12	8	4	0
k : [4;4]		4	5	6	7		13	9	5	1
s : [+,-]		8	9	10	11		14	10	6	2
m : [1;0]		12	13	14	15		15	11	7	3
d : [4;4]										

180° rotation:

k -Tile		A		D																																
		$\leftarrow 0 \rightarrow$		$\leftarrow 0 \rightarrow$																																
a : [4;4]	\uparrow 1 \downarrow	<table> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>4</td><td>5</td><td>6</td><td>7</td></tr> <tr><td>8</td><td>9</td><td>10</td><td>11</td></tr> <tr><td>12</td><td>13</td><td>14</td><td>15</td></tr> </table>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	\uparrow 1 \downarrow	<table> <tr><td>15</td><td>14</td><td>13</td><td>12</td></tr> <tr><td>11</td><td>10</td><td>9</td><td>8</td></tr> <tr><td>7</td><td>6</td><td>5</td><td>4</td></tr> <tr><td>3</td><td>2</td><td>1</td><td>0</td></tr> </table>	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0			1	2	3																															
4			5	6	7																															
8			9	10	11																															
12			13	14	15																															
15	14	13	12																																	
11	10	9	8																																	
7	6	5	4																																	
3	2	1	0																																	
k : [4;4]																																				
s : [-,-]																																				
m : [0;1]																																				
d : [4;4]																																				

90° counter-clockwise rotation:

k -Tile		A		D						
		$\leftarrow 0 \rightarrow$		$\leftarrow 0 \rightarrow$						
a : [4;4]	\uparrow 1 \downarrow	0	1	2	3	\uparrow 1 \downarrow	3	7	11	15
k : [4;4]		4	5	6	7		2	6	10	14
s : [-,+]		8	9	10	11		1	5	9	13
m : [1;0]		12	13	14	15		0	4	8	12
d : [4;4]										

3.2.3 Templates

One limitation of the k -Tile format is the lack of flexibility allowed in specifying where data is *not* stored; when processing large data sets it is often useful to include empty padding around the data.

Firstly, if the length of a data dimension is a prime number or has unsuitable factors, it may be impossible to split the dimension into regular tiles of a convenient size.

Secondly, it may be useful to include some empty space around the data to allow for working storage; one example would be to include a border around tiles when performing neighbourhood operations to allow inter-processor communication to be separated from computation (more details of this approach may be found in section 7.3.7).

One solution to both of these problems is to pad the data array by increasing the length of some dimensions at the expense of some wasted memory or processors. However, this approach is undesirable for several reasons:

- the actual shape of the data array is not preserved, which has the potential to cause confusion
- a different padding may be required for different mappings of the same data array, which could make it impossible to map between different k -Tile descriptions for the same data array
- it is not possible to optimize data movement by ignoring empty storage locations if they are not explicitly defined.

A solution to the padding problem is to allow each of the data, k -Tile and device spaces to be mapped into their own *template* space, which is effectively a padded image.

Given an index space Z ,

$$p \in \mathbb{N}, z \in \mathbb{N}^p, Z = \text{index}(z), z = (z_0, \dots, z_{p-1}),$$

a template space for Z is an index space T

$$q \in \mathbb{N}, t \in \mathbb{N}^q, T = \text{index}(t), t = (t_0, \dots, t_{q-1}),$$

where

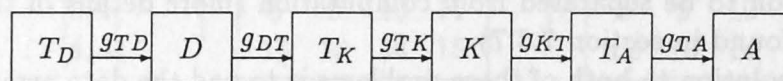
$$\begin{aligned} p &= q \text{ and} \\ t_i &\geq z_i \text{ for all } i, 0 \leq i < p. \end{aligned}$$

Elements of Z are mapped into T using a *template mapping*:

$$\text{template}(t, z) \triangleq g : T \rightarrow Z$$

$$g(u) = \left\{ \begin{array}{ll} \perp & \text{if } u \notin \text{index}(z) \\ u & \text{if } u \in \text{index}(z) \end{array} \right\} \text{ for all } u \in T.$$

The inclusion of template spaces in the k -Tile format adds three additional index spaces, T_D , T_K , and T_A with respective shapes t_D , t_K , and t_A , and three template mappings g_{TD} , g_{TK} and g_{TA} . Because the template spaces may be a different size from the data spaces the k -Tile mappings g_{DK} and g_{KA} are replaced by mappings to the appropriate template spaces, g_{DT} and g_{KT} .



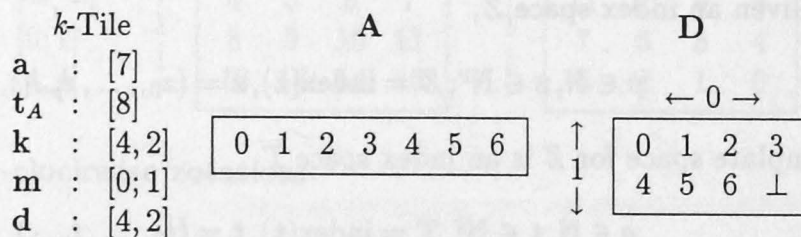
The template spaces for the data, k -Tile and device spaces can be used in quite different ways. The following sections give examples of the use of template spaces for three different purposes.

Data space template

When an image has dimensions which are not convenient for factorization into tiles, the data space may first be mapped into the data template space. Two example mappings of a 7 element image on two devices are shown:

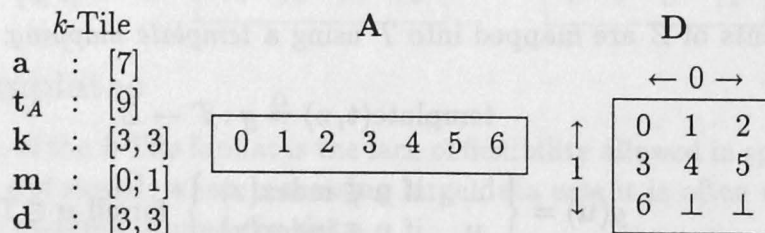
- 7-element image into 4×2 device

This mapping pads a 7-element input image to 8 elements and maps the resulting space to a 4×2 device.



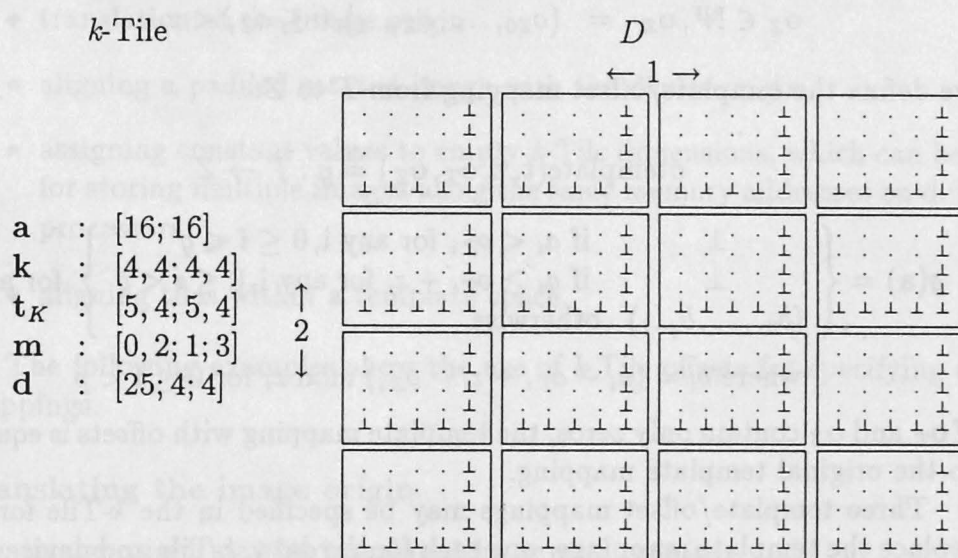
- 7-element image into 3×3 device

This mapping pads the 7-element input image to 9 elements and maps the resulting space to a 3×3 device.



k -Tile space template

When performing neighbourhood operations on the data elements in an image it may be useful to separate the inter-processor communication component of the operation from the computation component (see section 7.3.7). To avoid this communication, ‘overlapping’ tiles may be specified by using a k -Tile template space which allows empty work-space to be included around the edge of each tile. Before the commencement of computation data elements from each processor’s neighbours may be copied into the empty edges of the tile, allowing computation to be performed in exactly the same way for every data element. This k -Tile mapping defines a 16×16 image arranged as 4×4 tiles on 4×4 processors with extra space around the edge of each tile:



Device space template

Besides declaring storage space on the parallel device, the dimensions of the device space may have a role in setting up the connection configuration of the processor array (see section 3.1.3). It may be convenient to set up the processor array differently by allocating more space on the device than that represented by the k -Tile space by declaring a template device space.

3.2.4 Offsets

A geometrical transformation that is often used in image processing is the *translation* of spaces. Translation may be specified within the *k*-Tile format using an *offset* for each dimension of each space, including template spaces.

Data elements that are translated beyond the end of a non-template dimension are wrapped around, ensuring that no data is lost. Data elements may not be translated beyond the end of a template dimension. Because the

use of offsets does not change the shape of the associated index spaces, offsets may be incorporated within the existing template mappings g_{TD} , g_{TK} and g_{TA} by redefining the template mapping as follows. Given an index space Z and a template space for Z , T ,

$$p \in \mathbb{N}, \mathbf{z} \in \mathbb{N}^p, \mathbf{z} = (z_0, \dots, z_{p-1}), Z = \text{index}(\mathbf{z})$$

$$\mathbf{t} \in \mathbb{N}^p, \mathbf{t} = (t_0, \dots, t_{p-1}), T = \text{index}(\mathbf{t})$$

and two vectors \mathbf{o}_T and \mathbf{o}_Z , which contain the offsets within T and Z respectively,

$$\mathbf{o}_T \in \mathbb{N}^p, \mathbf{o}_T = (o_{T0}, \dots, o_{Tp-1}), 0 \leq o_{Ti} \leq (t_i - z_i)$$

$$\mathbf{o}_Z \in \mathbb{N}^p, \mathbf{o}_Z = (o_{Z0}, \dots, o_{Zp-1}), 0 \leq o_{Zi} < z_i$$

we define the template/offset mapping from T to Z :

$$\text{otemplate}(\mathbf{t}, \mathbf{z}, \mathbf{o}_T, \mathbf{o}_Z) \triangleq g : T \rightarrow Z$$

$$g(\mathbf{a}) = \left\{ \begin{array}{ll} \perp & \text{if } a_i < o_{Ti} \text{ for any } i, 0 \leq i < p \\ \perp & \text{if } a_i \geq o_{Ti} + z_i \text{ for any } i, 0 \leq i < p \\ (b_0, \dots, b_{p-1}) & \text{otherwise} \end{array} \right\} \text{ for } \mathbf{a} \in T$$

$$\text{where } b_i = (a_i - o_{Ti} + z_i - o_{Zi}) \bmod z_i \text{ for } 0 \leq i < p$$

If \mathbf{o}_T and \mathbf{o}_Z contain only zeros, the template mapping with offsets is equivalent to the original template mapping.

Three template/offset mappings may be specified in the *k*-Tile format to replace the template mappings: one each for the data, *k*-Tile and device spaces. Unlike the sense indicator, offsets within the data and device spaces cannot always be specified by giving an offset in the *k*-Tile space. This is because the wrap-around within a single *k*-Tile dimension is independent of any other *k*-Tile dimensions associated with a particular data or device dimension. Two examples show the difference between offsets in the data space and the *k*-Tile space. The first example shows a *k*-Tile format with an offset of one position in the first *k*-Tile dimension:

<i>k</i> -Tile	A	D
a : [6]		
k : [3, 2]		
o_K : [1, 0]	0 1 2 3 4 5	2 0 1 5 3 4
m : [0, 1]		
d : [6]		

The second example shows a *k*-Tile format with an offset of one position in the data dimension:

k -Tile	A	D												
a : [6]														
o_A : [1]														
k : [3, 2]	<table border="1"> <tr> <td>0</td> <td>1</td> <td>2</td> <td>3</td> <td>4</td> <td>5</td> </tr> </table>	0	1	2	3	4	5	<table border="1"> <tr> <td>5</td> <td>0</td> <td>1</td> <td>2</td> <td>3</td> <td>4</td> </tr> </table>	5	0	1	2	3	4
0	1	2	3	4	5									
5	0	1	2	3	4									
m : [0, 1]														
d : [6]														

A carry generated in the first k -Tile dimension must be propagated to the second k -Tile dimension to allow an offset in the data dimension.

Offsets may be used for many purposes:

- translation of the image origin
- aligning a padded rotated image with the device origin
- assigning constant values to empty k -Tile dimensions, which can be used for storing multiple images using the same memory addresses on different processors
- aligning tiles within a template space.

The following examples show the use of k -Tile offsets for specifying useful mappings.

Translating the image origin

The image origin may be translated for several reasons:

- when viewing an $[N, M]$ image, such as a 2d Fourier Transform, which has the origin at position (0,0), it is usual to translate the image so that the origin appears at the centre of the image, $[\frac{N}{2}, \frac{M}{2}]$.
- when extracting a region of interest from an image, it may be more convenient to align a corner of the region with the device origin.

This example shows an image offset by one element along the first data dimension:

k -Tile		A				D			
		$\leftarrow 0 \rightarrow$				$\leftarrow 0 \rightarrow$			
a	: [4; 4]	<div> <div> <div> <div> <div>0</div> <div>1</div> <div>2</div> <div>3</div> </div> <div> <div>4</div> <div>5</div> <div>6</div> <div>7</div> </div> <div> <div>8</div> <div>9</div> <div>10</div> <div>11</div> </div> <div> <div>12</div> <div>13</div> <div>14</div> <div>15</div> </div> </div> <div> <div>↑</div> <div>1</div> <div>↓</div> </div> </div> </div>				<div> <div> <div> <div> <div>3</div> <div>0</div> <div>7</div> <div>4</div> </div> <div> <div>1</div> <div>2</div> <div>5</div> <div>6</div> </div> <div> <div>11</div> <div>8</div> <div>15</div> <div>12</div> </div> <div> <div>9</div> <div>10</div> <div>13</div> <div>14</div> </div> </div> <div> <div>↑</div> <div>1</div> <div>↓</div> </div> </div> </div>			
o_A	: [1, 0]								
k	: [2, 2; 2, 2]								
m	: [0, 2; 1, 3]								
d	: [4; 4]								

Aligning rotated images with the device origin

An image can be rotated by inverting the sense of, and permuting, k -Tile dimensions. If the image does not fill the data template space fully, before the image is rotated empty space will be present between the ends of the image dimensions and the edges of the template dimensions. After this image is rotated, this empty space will appear at the start of one or more data dimensions. Having the image data appearing in different positions on the device depending on its rotation state is generally not desirable. To ensure that a corner of a rotated image always appears at the device origin, a data offset can be used.

The following three example mappings show a 3×3 image mapped to a 4×4 device, the same image rotated 180° , and the rotated image shifted to the device origin:

3×3 image mapped to a 4×4 device

k -Tile		A		D
a : [3; 3]		$\leftarrow 0 \rightarrow$		$\leftarrow 0 \rightarrow$
t_A : [4; 4]				
k : [4; 4]	\uparrow	0 1 2	\uparrow	0 1 2 \perp
m : [0; 1]	\downarrow	3 4 5	\downarrow	3 4 5 \perp
d : [4; 4]		6 7 8		6 7 8 \perp
				$\perp \perp \perp \perp$

180° rotation

k -Tile		A		D
a : [3; 3]		$\leftarrow 0 \rightarrow$		$\leftarrow 0 \rightarrow$
t_A : [4; 4]				
k : [4; 4]	\uparrow	0 1 2	\uparrow	$\perp \perp \perp \perp$
s : [-, -]	\downarrow	3 4 5	\downarrow	\perp 8 7 6
m : [0; 1]		6 7 8		\perp 5 4 3
d : [4; 4]				\perp 2 1 0

180° rotation translated to device origin

<i>k</i> -Tile	A	D
a : [3; 3]		
t_A : [4; 4]		
o_{T_A} : [1; 1]		
k : [4; 4]		
s : [-, -]		
m : [0; 1]		
d : [4; 4]		

	← 0 →	← 0 →
↑	0 1 2	8 7 6 ⊥
1	3 4 5	5 4 3 ⊥
↓	6 7 8	2 1 0 ⊥
		⊥ ⊥ ⊥ ⊥

3.2.5 Extended *k*-Tile offsets

By assigning special meanings to offsets in empty *k*-Tile dimensions, extra functionality may be obtained.

Assigning constant values to empty *k*-Tile dimensions

Some *k*-Tile mappings, such as scan-line mappings, use only a subset of the available processors and potentially waste limited storage space. By setting an unused *k*-Tile dimension to a constant value, which is equivalent to adding an offset to the value of that dimension, two images may be stored on different processors using the same memory locations. The definition of template mappings with offsets is already consistent with this usage of offsets.

These two images could be stored using the same memory array:

<i>k</i> -Tile	A	D
a : [4; 4]		
k : [4; 4; 2]		
o_K : [0; 0; 0]		
m : [0; 1; 2]		
d : [4; 8]		

	← 0 →	← 0 →
↑	0 1 2 3	0 1 2 3
1	4 5 6 7	4 5 6 7
↓	8 9 10 11	8 9 10 11
	12 13 14 15	12 13 14 15
		⊥ ⊥ ⊥ ⊥
		⊥ ⊥ ⊥ ⊥
		⊥ ⊥ ⊥ ⊥
		⊥ ⊥ ⊥ ⊥

$$\mathbf{o}_K \in (\mathbb{N} \cup \{*\})^q, \mathbf{o}_Z = (o_{K0}, \dots, o_{Kq-1}), 0 \leq o_{Ki} < k_i, \text{ or } k_i = *$$

A template/offset mapping including data replication may be defined thus:

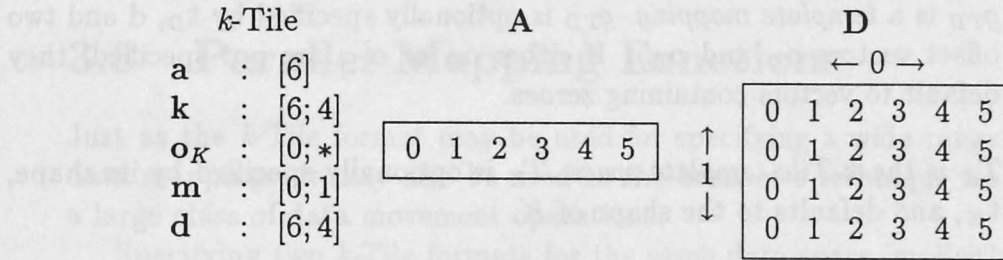
$$\text{otemplate}^*(t, \mathbf{k}, \mathbf{o}_T, \mathbf{o}_K) \triangleq g : T \rightarrow K$$

$$g(\mathbf{a}) = \begin{cases} \perp & \text{if } \exists i, a_i < o_{Ti} \text{ for any } i, 0 \leq i < q \\ \perp & \text{if } \exists i, a_i \geq o_{Ti} + k_i \text{ for any } i, 0 \leq i < q \\ (b_0, \dots, b_{(K)-1}) & \text{otherwise.} \end{cases}$$

$$\text{where } b_i = \begin{cases} 0 & \text{if } o_{Ki} = * \\ (a_i - o_{Ti} + k_i - o_{Zi}) \bmod k_i & \text{otherwise} \end{cases}$$

Use of the $*$ -offset is not allowable in a non-empty k -Tile dimension.

This example mapping shows how a look-up table could be stored locally on every processor in a processor array:



3.2.6 Summary of extended k -Tile format

The k -Tile format is a flexible data mapping between a data space \mathbb{M}_a and a device space \mathbb{M}_d , and is in turn specified as a composite index mapping between a device index space D and a data index space A .

Two types of index mappings are used to specify a k -Tile format: k -Tile mappings and template mappings. k -Tile mappings allow a lower-dimensional space to be mapped into a higher-dimensional space to allow dimensions to be tiled. Template mappings allow spaces to be translated and empty space to be declared around mapped data.

Two variations on the basic k -Tile mapping are used in the k -Tile format.

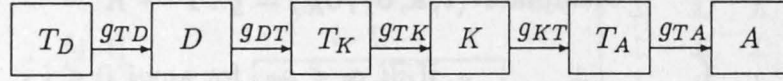
The index mapping between the k -Tile index space and the data index space is an implicit k -Tile mapping allowing empty k -Tile dimensions for declaring empty space.

The index mapping between the device index space and the k -Tile index space is an inverse k -Tile mapping including a sense indicator.

Three template mappings are used in the k -Tile format, one each for the device, k -Tile and data index spaces.

A variation on the basic template mapping allows k -Tile dimensions to be replicated across the device.

The mappings and spaces defined in a k -Tile format are summarized in this diagram showing the index spaces and index mappings included in a k -Tile index format:



where

- i. T_D is the *device template space*, containing the indices of an array in the physical memory of some storage device. T_D is optionally specified by its shape, t_D , and defaults to the shape of D .
- ii. D is the *device index space*. D must be specified by its shape, d .
- iii. g_{TD} is a *template mapping*. g_{TD} is optionally specified by t_D , d and two offset vectors o_D and o_{T_D} . If either o_D or o_{T_D} are not specified, they default to vectors containing zeroes.
- iv. T_K is the *k-Tile template space*. T_K is optionally specified by its shape, t_K , and defaults to the shape of K .
- v. g_{DT} is a an inverse k -Tile mapping with sense indication. g_{DT} must be specified by d , t_K and by a mapping vector m .
- vi. K is the *k-Tile index space*. K must be specified by its shape, k .
- vii. g_{TK} is a *template mapping*. g_{TK} is optionally specified by t_K , k and two offset vectors o_K and o_{T_K} . If either o_K or o_{T_K} are not specified, they default to vectors containing zeroes. g_{TK} may be used to replicate data using the special *-offset in o_K .
- viii. T_A is the *data template space*. T_A is optionally specified by its shape, t_A , and defaults to the shape of A .
- ix. g_{KT} is an implicit k -Tile mapping, specified by k and t_A .
- x. A is the *data index space*, representing the data array to be mapped to the device. A must be specified by its shape, a .
- xi. g_{TA} is a *template mapping*. g_{TA} is optionally specified by t_A , a and two offset vectors o_A and o_{T_A} . If either o_A or o_{T_A} are not specified, they default to vectors containing zeroes.

g_K may be written as the composition of several index maps:

$$\begin{aligned}
 g_K &: T_D \rightarrow A \\
 g_K &= \text{otemplate}(t_A, a, o_{T_A}, o_A) \circ \\
 &\quad \text{ikmap}(k, t_A) \circ \\
 &\quad \text{otemplate}^*(t_K, k, o_{T_K}, o_K) \circ \\
 &\quad \text{skmap}^{-1}(t_K, d, m, s) \circ \\
 &\quad \text{otemplate}(t_D, d, o_{T_D}, o_D)
 \end{aligned}$$

The smallest subset of the k -Tile format, specified by a , k , m and d is sufficient for specifying a large number of useful data mappings. Each of the additional k -Tile format fields defined here has been shown to have a useful role in defining data mappings, although only a small selection will usually be needed for any particular problem.

3.3 Parallel Mapping Functions

Just as the k -Tile format may be used for specifying a wide range of regular data mappings, it may also be used as the basis of a technique for specifying a large class of data movement operations.

Specifying two k -Tile formats for the same data space implicitly defines a *remapping* from one storage mapping to another; by specifying the current and desired mappings for a data array to a device, a mapping is defined between storage locations on the device through the data index space. The mapping between two device addresses defined by two k -Tile formats is a *parallel mapping function*.

Although the k -Tile format provides a way of specifying remappings, it is also necessary to have a system of algorithms to *perform* the remapping by taking the two k -Tile formats and performing the appropriate permutation of the data stored on the device. Chapters 4 and 6 describe systems for performing a subset of these k -Tile remappings using lower-level mapping specification techniques. Chapter 5 describes a system that has been implemented on the Maspar MP-1 for performing parallel mapping functions using a subset of the k -Tile format restricted to spaces with dimensions with powers-of-two lengths, and shows how these restricted parallel mapping functions may be converted into lower-level operations.

3.4 Summary

This thesis is intended to address two problems in computation with large multi-dimensional data sets: the *mapping problem*, or specifying the layout

of data on a storage or computational device, and the *remapping problem*, or dynamically rearranging data on a device from one mapping to another. This chapter demonstrates the *k*-Tile format, which is a flexible way of specifying the layout of multidimensional data arrays on parallel or sequential storage or computational devices, and hence is one solution to the mapping problem.

Fraser's tile format, which is a mechanism for specifying the layout of a multidimensional image on a one-dimensional device, forms the basis for the definition of the *k*-Tile format.

The *k*-Tile format is defined by a composition of index mappings between multidimensional array spaces, the data space, the device space, and the *k*-Tile space. The data and device spaces are representations of the data array and the storage device. The structure of the *k*-Tile format is device-independent, and may be used for any parallel architecture, or even sequential architectures. Many data mappings can be specified with the *k*-Tile format:

- common 1d and 2d storage mappings
- geometrical transformations such as rotations by multiples of 90° , reflections and transpositions
- a variety of 3d volume mappings
- arbitrary translations and rotations within the data and device coordinate systems
- index bit reversals for the Fast Fourier Transform
- data tiles bordered by empty space
- replication of data across many processors.

Specifying two *k*-Tile formats for the same data array implicitly defines a remapping from one storage mapping to another; this forms the basis for *Parallel Mapping Functions*, which use the flexibility of the *k*-Tile format to describe a class of data movement operations.

Chapter 4

Radix 2 remapping

The mapping of multidimensional data onto a processor array can have a significant effect on the selection, efficiency and programming complexity of algorithms used to solve many problems. Where a combination of several algorithms, input and output operations are required to solve a problem, it is useful to be able to alter the mapping of data on a processor array dynamically. This chapter describes methods for both mapping and remapping multidimensional arrays whose dimensions have lengths which are powers of two.

The *index bit map* is a flexible method of specifying a data mapping of such an array to a processor array.

A pair of index bit maps may be used to specify a data remapping. This remapping can be expressed as an *index bit permutation* and an *index bit inversion* [1, 13, 14, 20, 21, 27, 28, 56, 70].

Algorithms have already been developed for index bit permutation and inversion on mesh-connected processors, but by using a model of a more flexible architecture we introduce more efficient algorithms.

4.1 The index bit map

Using the definitions in section 2.1, the index bit map defines a data-independent data mapping from a multidimensional array space representing a data array to a multidimensional array space representing a device array. The data mapping is defined using an index mapping from the device array index space to the data array index space. The index mapping is defined by a mapping between bits of elements in the device array index space to the data array index space.

Many commonly used data mappings may be specified by the index bit map. Some examples are n -dimensional hierarchical, cut-and-stack, and scan-line mappings. Some geometrical transformations of these mappings may also be described, such as n -dimensional transpositions, reverses and rotations by

multiples of 90° . In the next chapter we will show that the index bit map is equivalent to a restricted form of the k -Tile format.

4.1.1 An overview of the index bit map

The index bit map is similar to the k -Tile format in that it defines a data mapping from a data array to a device array. However, there are two important differences between the index bit map and the k -Tile format: firstly, we assume that the size of the data and device arrays is a power of two; secondly, we assume that the data and device arrays are one dimensional. We show that the second restriction can be avoided by 'wrapping' multidimensional arrays into one-dimensional arrays.

The data and device array spaces

The data and device spaces we use to define the index bit map are identical to the data and device spaces used when defining the k -Tile format in chapter 3. We repeat some of the properties of these spaces:

- All arrays have data type *byte*. Data types requiring multiple bytes can be incorporated into an array by using the first dimension of the array to represent the data type.
- Distributed memory parallel processors are treated as multidimensional arrays, with the first dimension representing memory and subsequent dimensions representing dimensions in the processor array
- The dimensionality and shape of the device array need not be fixed, but both the dimensionality and the lengths of each dimension will be bounded by the constraints on the physical device. By treating devices as having lower dimensionality than they actually possesses, some may be optimized for a chosen shape.

In addition, in this chapter we assume that the size of any multidimensional array A is a power of two. With this restriction, it is often more convenient to treat sizes in terms of \log_2 of their absolute size. Using the definitions in section 2.1, we define

$$\log \text{ size of } A = \|A\| \triangleq \log_2 |A|$$

Similarly, we may define the log size of a set A ,

$$\log \text{ size of } A = \|A\| \triangleq \log_2 |A|$$

Wrapping the data and device array spaces

For convenience we assume in this chapter that both the data and device arrays are one dimensional. Although this restriction seems harsh, by defining a mapping between a multidimensional array and a one dimensional array, we may use one dimensional mapping techniques for multidimensional arrays.

Given an index space A with shape \mathbf{a} ,

$$p \in \mathbb{N}, \mathbf{a} \in \mathbb{N}^p, \mathbf{a} = (a_0, \dots, a_{p-1}), A = \text{index}(\mathbf{a})$$

and a one-dimensional destination index space Z ,

$$z = |A|, Z = \text{index}(z)$$

we define the *index wrapping mapping* from A to Z

$$\text{wrap}(\mathbf{a}) \triangleq g_w : A \rightarrow Z, \text{ where}$$

$$g_w(u_0, \dots, u_{p-1}) = (v), \text{ for all } (u_0, \dots, u_{p-1}) \in A, \text{ where}$$

$$v = \sum_{i=0}^{p-1} u_i \cdot \prod_{j=0}^{i-1} a_j$$

The mapping is one to one and onto, and thus may be inverted directly

$$\text{wrap}^{-1}(\mathbf{a}) \triangleq g_w^{-1} : Z \rightarrow A, \text{ where}$$

$$g_w(u_0, \dots, u_{p-1}) = v \Rightarrow g_w^{-1}(v) = (u_0, \dots, u_{p-1}) \text{ for all } (u_0, \dots, u_{p-1}) \in A$$

Given two multidimensional spaces of shape \mathbf{a} and \mathbf{d} and a one dimensional index map g ,

$$p \in \mathbb{N}, \mathbf{a} \in \mathbb{N}^p, \mathbf{a} = (a_0, \dots, a_{p-1}), A = \text{index}(\mathbf{a})$$

$$r \in \mathbb{N}, \mathbf{d} \in \mathbb{N}^r, \mathbf{d} = (d_0, \dots, d_{r-1}), D = \text{index}(\mathbf{d})$$

$$g : \text{index}(|D|) \rightarrow \text{index}(|A|)$$

we may define a multidimensional index map from D to A :

$$g_{DA} : D \rightarrow A$$

$$g_{DA} = \text{wrap}^{-1}(\mathbf{a}) \circ g \circ \text{wrap}(\mathbf{d})$$

Thus, given a data mapping for a one-dimensional data array and a one-dimensional device array, we may define a data mapping for multi-dimensional data and device arrays of the same size.

Binary index notation

The basis of the index bit map is the manipulation of the binary digits, or bits, used to represent data and device array indices. Following the notation in Nassimi [56], we may refer to an index a in the index space \mathbf{A} of a one dimensional data array by representing it in binary form:

$$a = a_{\|\mathbf{A}\|-1} \dots a_0 = \sum_{i=0}^{\|\mathbf{A}\|-1} a_i \cdot 2^i$$

This notation should not confuse a with the shape of \mathbf{A} ; the arrays we are dealing with in this chapter are one-dimensional and thus we shall not need to refer to the shape of \mathbf{A} separately from its size, because $[\mathbf{A}] = |\mathbf{A}|$.

Similarly, we refer to an index d in the index space \mathbf{D} of the device array in binary form also:

$$d \in D, d = d_{\|\mathbf{D}\|-1} \dots d_0 = \sum_{i=0}^{\|\mathbf{D}\|-1} d_i \cdot 2^i$$

Because the wrapped device index incorporates both memory indices and processor indices, we sometimes refer to an index d while distinguishing between the memory index bits and the processor index bits. Because the memory dimension is dimension zero in the device index space, the memory index bits form the lowest significant bits of the device index and the processor index bits the most significant bits. To distinguish between the memory index bits and the device index bits, we define two index spaces, P and M , which respectively contain indices for the processor and memory components of the device array. Given a multidimensional device shape \mathbf{b} and a wrapped device index space D we define P and M :

$$r \in \mathbb{N}, \mathbf{b} \in \mathbb{N}^r, \mathbf{b} = (b_0, \dots, b_{r-1})$$

$$D = \text{index}(|\text{index}(\mathbf{b})|)$$

we define the *memory index space*

$$M = \text{index}(b_0)$$

and the *processor index space*

$$P = \text{index}((b_1, \dots, b_{r-1}))$$

The number of processors is $2^{\|P\|}$ and the number of memory locations per processor is $2^{\|M\|}$.

We may now write an index d in the index space of the device array in binary form, using m_i for memory index bits and p_j for processor index bits:

$$d \in D, d = p_{\|P\|-1} \dots p_0 m_{\|M\|-1} \dots m_0 = \sum_{i=0}^{\|M\|-1} m_i \cdot 2^i + 2^{\|M\|} \cdot \sum_{i=0}^{\|P\|-1} p_i \cdot 2^i$$

Mapping device index bits to data index bits

An index bit map between the device index space and the data index space is defined by an index bit permutation followed by an index bit inversion. The index bit permutation defines a mapping between the device index bits and the data index bits, and the index bit inversion causes a fixed set of index bits to be inverted.

When a device index contains more bits than a data index, not all the device index bits can be assigned to data index bits. Any device index bit which is assigned a data index bit is said to be *enabled*. Any device index bit not assigned a data index bit is said to be *disabled* and is instead assigned to be either zero or one. A disabled index bit is said to be *inverted* if it is assigned to one.

Any device indices which are mapped into the data array by the index bit map are said to be *enabled*, and device addresses mapped to \perp are said to be *disabled*.

An *enabled processor* contains mapped data and a *disabled processor* contains none, and an *enabled memory address* contains mapped data on an enabled processor and a *disabled memory address* contains no mapped data on enabled processors.

4.1.2 Definition of the index bit map

Permutations

A *permutation* is an operation changing the order of a vector's elements [39]. Given a vector \mathbf{a} of length q ,

\mathbb{T} is a set

$$q \in \mathbb{N}$$

$$\mathbf{a} \in \mathbb{T}^q, \mathbf{a} = (a_0, \dots, a_{q-1})$$

we define a permutation using a mapping P :

$$P : \{0, \dots, q-1\} \rightarrow \{0, \dots, q-1\}$$

$$P(i) = P(j) \text{ only if } i = j$$

From P we define a permutation π :

$$\pi : \mathbb{T}^q \rightarrow \mathbb{T}^q$$

$$\pi(\mathbf{a}) = \mathbf{b}, \text{ for all } \mathbf{a} \in \mathbb{T}^q, \text{ where}$$

$$\mathbf{a} = (a_0, \dots, a_{q-1})$$

$$\mathbf{b} = (b_0, \dots, b_{q-1})$$

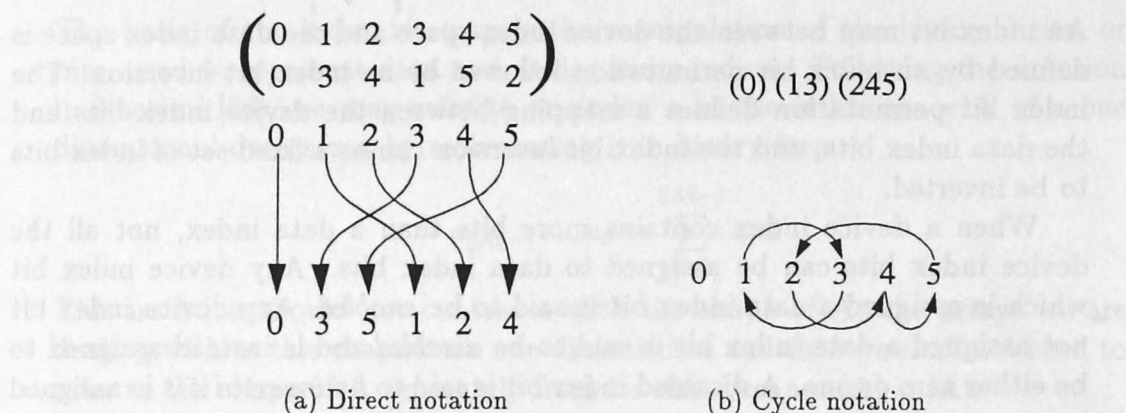


Figure 4.1: Two representations of an example permutation

$$b_{P(i)} = a_i \text{ for all } i, 0 \leq i < q$$

A permutation is usually represented in one of two ways, in direct notation or in cycle notation, as shown in figure 4.1.

The direct notation gives the destination position of each element written below its original position, for example

$$\pi = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 3 & 4 & 1 & 5 & 2 \end{pmatrix}.$$

Because in binary index notation bit zero appears at the right, it is sometimes more convenient to represent a permutation in direct notation with the original positions presented in decreasing order, for example

$$\pi = \begin{pmatrix} 5 & 4 & 3 & 2 & 1 & 0 \\ 2 & 5 & 1 & 4 & 3 & 0 \end{pmatrix}.$$

The cycle notation shows the permutation as *cycles*. A cycle is a list of element positions where each position is followed by its destination position. Because listing the elements in this way must always return to the first element in the list, the list is terminated by the last element in the cycle. Two cycles are said to be *disjoint* if they have no element positions in common. A permutation may be represented as a list of disjoint cycles,

$$\pi = (0)(13)(245)$$

A single element cycle is called an *identity cycle*, and need not be included in the cycle notation if its presence is clear by context. The identity permutation, in which no element changes position, is represented I. Because each cycle

represents a closed loop, it does not matter which element begins the cycle. Usually each cycle will be represented with the element with the smallest index first, and the cycles will be listed with the first elements in increasing order.

A permutation can also be represented as a *permutation matrix*, and the permutation performed by multiplying the vector by the permutation matrix. Using the same example again,

$$\pi(\mathbf{a}) = \mathbf{a} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Permutations thus may be composed by multiplication and are associative, but not commutative. For example:

$$\begin{aligned} (21).(01) &= (012) \\ (01).(21) &= (021) \\ (012345).(02) &= (01)(2345) \\ (0123).(01).(12).(23) &= \mathbf{I} \\ (012345).(01) &= (12345) \\ (012).(345).(03) &= (012345) \\ (012345).(543210) &= \mathbf{I} \\ (012345).(012345) &= (024)(135) \end{aligned}$$

Every permutation P has an inverse, P^{-1} , such that

$$P \circ P^{-1} = \mathbf{I}$$

Index bit permutations

An *index bit permutation* \mathcal{P} is an operation changing the order of q bits in an integer. Let

$$q \in \mathbb{N}$$

$$P : \{0, \dots, q-1\} \rightarrow \{0, \dots, q-1\}$$

$$P(i) = P(j) \text{ only if } i = j$$

Using the permutation map P , which specifies q destination positions of q bits, we define an index bit permutation \mathcal{P} :

$$\text{ibp}(q, P) \triangleq \mathcal{P} : \{0, \dots, 2^q - 1\} \rightarrow \{0, \dots, 2^q - 1\}$$

$\mathcal{P}(u) = v$ for all $u, v \in \{0, \dots, 2^q - 1\}$, where

$$u = \sum_{i=0}^{q-1} 2^i \cdot u_i, v = \sum_{i=0}^{q-1} 2^{P(i)} \cdot u_i$$

For example, let P be the permutation:

$$P = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 2 & 0 \end{pmatrix} = (012) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

from P we can define an index bit permutation \mathcal{P} :

$$\mathcal{P} = \text{ibp}(3, P)$$

$\mathcal{P}(a)$	$=$	$\mathcal{P}(a_2 a_1 a_0)$	$=$	$a_1 a_0 a_2$	
$\mathcal{P}(0)$	$=$	$\mathcal{P}(000_2)$	$=$	000_2	$= 0$
$\mathcal{P}(1)$	$=$	$\mathcal{P}(001_2)$	$=$	010_2	$= 2$
$\mathcal{P}(2)$	$=$	$\mathcal{P}(010_2)$	$=$	100_2	$= 4$
$\mathcal{P}(3)$	$=$	$\mathcal{P}(011_2)$	$=$	110_2	$= 6$
$\mathcal{P}(4)$	$=$	$\mathcal{P}(100_2)$	$=$	001_2	$= 1$
$\mathcal{P}(5)$	$=$	$\mathcal{P}(101_2)$	$=$	011_2	$= 3$
$\mathcal{P}(6)$	$=$	$\mathcal{P}(110_2)$	$=$	101_2	$= 5$
$\mathcal{P}(7)$	$=$	$\mathcal{P}(111_2)$	$=$	111_2	$= 7$

An index bit permutation is one to one and onto, therefore an inverse function exists. This inverse is

$$\text{ibp}^{-1}(q, P) \triangleq \mathcal{P}^{-1} : \{0, \dots, 2^q - 1\} \rightarrow \{0, \dots, 2^q - 1\}$$

$$\mathcal{P}^{-1} = \text{ibp}(q, P^{-1})$$

Index bit inversions

An *index bit inversion* \mathcal{S} is an operation inverting a selection of q bits in an integer. This can be performed by the use of the bitwise *exclusive-or* operator, \oplus . This operator performs the logical exclusive-or operation on corresponding bits in two integers to yield an integer result. Let

$$q \in \mathbb{N}$$

$$s \in \mathbb{N}, 0 \leq s < 2^q$$

Using the integer s , which specifies a fixed set of bits to invert, we define an index bit inversion \mathcal{S} :

$$\text{inv}(q, s) \triangleq \mathcal{S} : \{0, \dots, 2^q - 1\} \rightarrow \{0, \dots, 2^q - 1\}$$

$$\mathcal{S}(u) = u \oplus s$$

In binary notation, the effect of this operation is clearer:

$$s = s_{q-1} \dots s_0$$

$$\mathcal{S}(u) = v, \text{ where}$$

$$u = u_{q-1} \dots u_0$$

$$v = v_{q-1} \dots v_0$$

$$v_i = u_i \oplus s_i, \text{ for all } i, 0 \leq i < q$$

Two properties of index bit permutation and index bit inversion are used in the following sections: \mathcal{S} is its own inverse:

$$\mathcal{S}(\mathcal{S}(u)) = s \oplus (s \oplus u) = (s \oplus s) \oplus u = 0 \oplus u = u;$$

and index bit permutation is distributive over index bit inversion

$$\mathcal{P}(s_1 \oplus s_2) = \mathcal{P}(s_1) \oplus \mathcal{P}(s_2)$$

Index bit map

The index bit map is defined using an index bit permutation and an index bit inversion. Letting

$$r, q \in \mathbb{N}, r \geq q$$

$$A = \text{index}(2^q)$$

$$D = \text{index}(2^r)$$

$$s \in \mathbb{N}, 0 \leq s < 2^q$$

$$P : \{0, \dots, r-1\} \rightarrow \{0, \dots, r-1\}$$

$$P(i) = P(j) \text{ only if } i = j$$

we can define an index mapping g_x :

$$g_x : D \rightarrow D$$

$$g_x = \text{inv}(r, s) \circ \text{ibp}(r, P)$$

The function g_x almost defines a mapping from device indices to data indices. However, if the device array index contains more bits than the data array index, some device index bits are mapped to bit positions not present in the data index. Thus, we must define data addresses as undefined where device addresses are mapped outside the data index space. Device index bits that are mapped to bit positions not present in the data index are said to be *disabled*,

and those that are mapped are said to be *enabled*. Correcting this deficiency in g_x , we define the index bit map g :

$$\text{ibm}(r, q, P, s) \triangleq g : D \rightarrow A \cup \{\perp\}$$

$$g(d) = \begin{cases} \text{inv}(r, s) \circ \text{ibp}(r, P)(d) & \text{if } \text{inv}(r, s) \circ \text{ibp}(r, P)(d) \in A \\ \perp & \text{otherwise} \end{cases} \text{ for all } d \in D$$

From g we may directly define a data mapping, f

$$f : \mathbb{M}_{2^q} \rightarrow \mathbb{M}_{2^r}$$

$$f(A)(d) = A(g(d)) \text{ for all } d \in D$$

When there are two or more disabled device bits, the choice of \mathcal{P} and \mathcal{S} is not unique, because disabled device index bits can be assigned to non-existent data index bits in any order.

An inversion of the index bit index map, g^{-1} , from data indices to enabled device indices may be derived as follows. Letting

$$E = \{d \in D : g(d) \in A\},$$

E is the set of enabled device addresses. g is one to one and onto from E to A , therefore an inverse function of g exists. Letting

$$\mathcal{P} = \text{ibp}(q, P)$$

$$s' \in \mathbb{N}, s' = \mathcal{P}^{-1}(s)$$

$$g^{-1} : A \rightarrow E$$

$$g^{-1}(g(d)) = d \text{ for all } d \in E$$

The inverse index bit map g^{-1} from A to E may be derived as follows:

$$g(d) = \text{ibm}(r, q, P, s)(d) \text{ for all } d \in E$$

$$g(g^{-1}(a)) = \text{inv}(r, s) \circ \text{ibp}(r, P)(g^{-1}(a)) \text{ for all } a \in A$$

$$a = s \oplus \mathcal{P}(g^{-1}(a))$$

$$\mathcal{P}^{-1}(a) = \mathcal{P}^{-1}(s \oplus \mathcal{P}(g^{-1}(a)))$$

$$\mathcal{P}^{-1}(a) = \mathcal{P}^{-1}(s) \oplus \mathcal{P}^{-1}(\mathcal{P}(g^{-1}(a)))$$

$$\mathcal{P}^{-1}(s) \oplus \mathcal{P}^{-1}(a) = g^{-1}(a)$$

$$\text{inv}(r, s') \circ \text{ibp}(r, P^{-1})(a) = g^{-1}(a)$$

Thus, the inverse index bit mapping from A to a subset of D has the same form as the index bit map.

4.1.3 An index bit map notation

An index bit permutation and index bit inversion may be represented in a simple notation, providing a way to directly specify any index bit map.

- The permutation P , which directly assigns device index bits to data index bits, is specified by showing the destination data index bit for each device index bit position. The data index is represented by the binary number $a_{\|A\|-1} \dots a_0$, with zeroes assumed to be in bit positions more significant than $\|A\| - 1$. For example,

$$P = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 1 & 0 \end{pmatrix} = (0)(1)(2) \Rightarrow \text{ibm}(3, 3, P, 0) \equiv a_2 a_1 a_0$$

$$P = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} = (012) \Rightarrow \text{ibm}(3, 3, P, 0) \equiv a_0 a_2 a_1$$

$$P = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 2 & 1 & 3 & 0 \end{pmatrix} = (0)(132) \Rightarrow \text{ibm}(4, 3, P, 0) \equiv a_2 a_1 0 a_0$$

- The sense is specified by putting an overbar over enabled inverted device bits, and using 1 instead of 0 for disabled inverted device bits.

For example,

$$P = \begin{pmatrix} 2 & 1 & 0 \\ 2 & 1 & 0 \end{pmatrix} = (0)(1)(2) \left. \vphantom{\begin{pmatrix} 2 & 1 & 0 \\ 2 & 1 & 0 \end{pmatrix}} \right\} \begin{matrix} s = 2_{10} = 010_2 \end{matrix} \Rightarrow \text{ibm}(3, 3, P, s) \equiv a_2 \bar{a}_1 a_0$$

$$P = \begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix} = (012) \left. \vphantom{\begin{pmatrix} 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}} \right\} \begin{matrix} s = 2_{10} = 010_2 \end{matrix} \Rightarrow \text{ibm}(3, 3, P, s) \equiv a_0 \bar{a}_2 a_1$$

$$P = \begin{pmatrix} 3 & 2 & 1 & 0 \\ 2 & 1 & 3 & 0 \end{pmatrix} = (0)(132) \left. \vphantom{\begin{pmatrix} 3 & 2 & 1 & 0 \\ 2 & 1 & 3 & 0 \end{pmatrix}} \right\} \begin{matrix} s = 3_{10} = 0011_2 \end{matrix} \Rightarrow \text{ibm}(4, 3, P, s) \equiv a_2 a_1 1 \bar{a}_0$$

An example mapping from a data array to a device is shown in Figure 4.2.

4.2 Radix 2 remapping

Once we have a data array mapped onto a device in one index bit map, we may wish to re-order the data on the device to correspond to another index

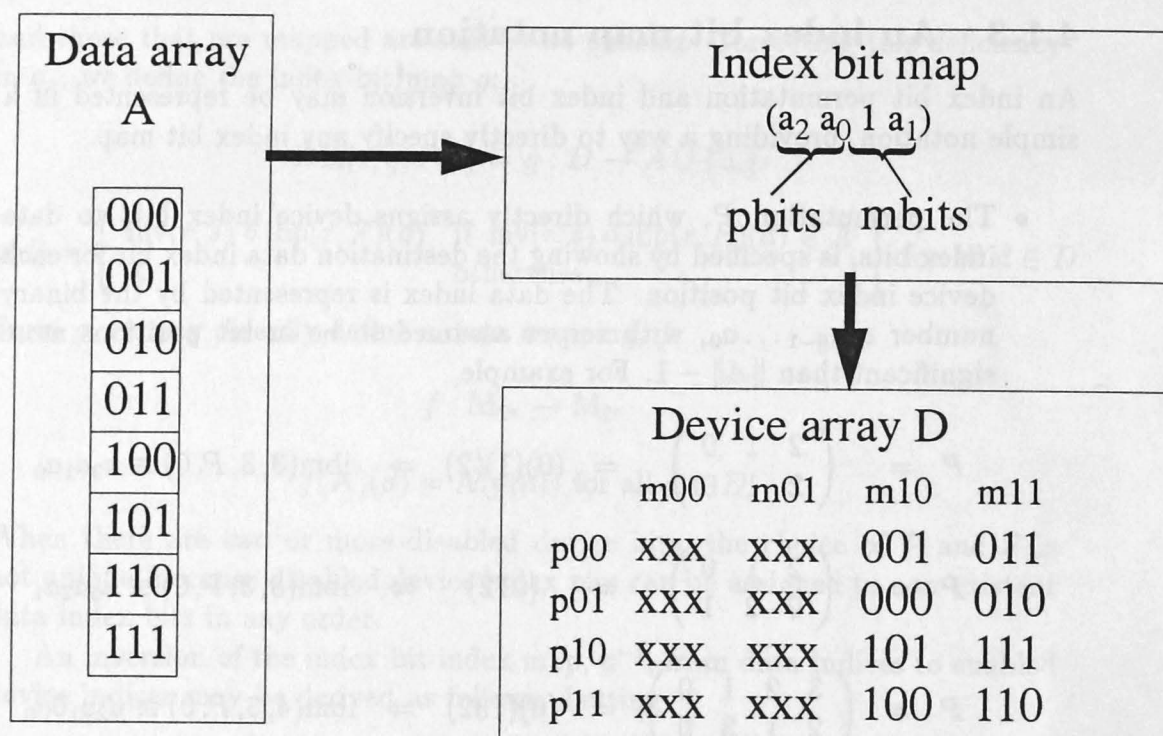


Figure 4.2: Example index bit map of a data array onto a parallel device

bit map. Such a remapping is specified given the current *source* index bit map and the desired *destination* index bit map.

Where two mappings may be specified by an index bit map, data mapped to the array in one mapping may be *remapped* to the other using the technique of *index-bit permutation* and *index bit inversion*. Index bit permutation is a special case of *index-digit permutation* [27, 28, 70], which is itself related to general *parallel mapping functions*.

Efficient algorithms have been designed to perform operations similar to radix 2 remapping on SIMD mesh-connected computers by sequences and combinations of inversions of single bits and swaps of pairs of bits. The difference between radix 2 remapping and the previous work is the inclusion of disabled device index bits in radix 2 remapping. An optimal mesh algorithm for processor-only index bit permutation and inversion is described by Nassimi [56], and a system for processor/memory index bit permutation and inversion, Parallel Data Transforms, has been created for the AMT DAP [1, 20, 21, 59]. Algorithms for radix 2 remapping are efficient on a mesh connected processor, but require multiple communication steps and multiple passes over data stored in the processor array.

In this chapter we develop generalizations of these earlier methods. In section 4.3 we show how radix 2 remapping may be performed using a set of *atomic index bit* operations. In section 4.4 we develop an optimal algorithm for

radix 2 remapping for an indirect addressing machine with a crossbar switch.

4.2.1 Definition of radix 2 remapping

Usually when defining index mappings we define a mapping from destination indices to source indices to allow data to be replicated in several locations on a device. In a radix 2 remapping all data is mapped and data replication is not allowed, ensuring that a mapping we define for remapping between enabled source addresses and enabled destination addresses is one to one and onto. Thus, we may define a radix 2 remapping as a mapping from source device indices to destination device indices.

This mapping is defined by using the two index bit maps defining the radix 2 remapping. Using the source index bit map we map the source device index into a data index. The data index can then be mapped to the desired destination address using the inverse of the destination index bit map.

Assume we have two index bit maps g_s and g_d for a data array with index space A to a device array with index space D . Using the definitions in section 4.1.2 we can derive a mapping from source device addresses to destination device addresses. Letting

$$q, r \in \mathbb{N}, q \leq r$$

$$P_s : \{0, \dots, r-1\} \rightarrow \{0, \dots, r-1\}, P(i) = P(j) \Rightarrow i = j$$

$$P_d : \{0, \dots, r-1\} \rightarrow \{0, \dots, r-1\}, P(i) = P(j) \Rightarrow i = j$$

$$\mathcal{P}_s = \text{ibp}(r, P_s), \mathcal{P}_d = \text{ibp}(r, P_d)$$

$$\mathcal{P}' = \mathcal{P}_d^{-1} \circ \mathcal{P}_s$$

$$s_s \in \mathbb{N}, 0 \leq s_s < 2^r, s_d \in \mathbb{N}, 0 \leq s_d < 2^r$$

$$g_s : D \rightarrow A \cup \{\perp\}, g_s = \text{ibm}(q, r, P_s, s_s)$$

$$g_d : D \rightarrow A \cup \{\perp\}, g_d = \text{ibm}(q, r, P_d, s_d)$$

$$E_s = \{d \in D : g_s(d) \in A\}$$

$$E_d = \{d \in D : g_d(d) \in A\}$$

$$s' = \mathcal{P}_d^{-1}(s_d \oplus s_s)$$

we define the *radix 2 remapping* from the source mapping to the destination mapping

$$\text{remap}(q, r, P_s, s_s, P_d, s_d) \triangleq g : E_s \rightarrow E_d$$

$$g(d_s) = g_d^{-1}(g_s(d_s)) \text{ for all } d_s \in E_s$$

Using a similar derivation to that of the inverse index bit map, the radix 2 remap can be derived:

$$\begin{aligned}
 g(d_s) &= g_d^{-1}(g_s(d_s)) \text{ for all } d_s \in E_s \\
 &= \text{ibm}^{-1}(r, P_d, s_d)(\text{ibm}(r, P_s, s_s)(d_s)) \\
 &= \text{ibm}^{-1}(r, P_d, s_d)(s_s \oplus P_s(d_s)) \\
 &= P_d^{-1}(s_d) \oplus P_d^{-1}(s_s \oplus P_s(d_s)) \\
 &= P_d^{-1}(s_d) \oplus P_d^{-1}(s_s) \oplus P_d^{-1}(P_s(d_s)) \\
 &= P_d^{-1}(s_d \oplus s_s) \oplus P_d^{-1} \circ P_s(d_s) \\
 &= s' \oplus P'(d_s)
 \end{aligned}$$

Thus, the transformation from source to destination address consists of an index bit permutation and bit inversions. We will refer to this form of device index mapping as a *radix 2 remapping*.

Although the transformation need only be applied to enabled source device addresses to define an index remapping, because the component mappings are one to one and onto in the whole of D , the domain of the mapping can be extended over the whole of D . An example of using index bit maps to define a radix 2 remapping is shown in figure 4.3.

4.2.2 A radix 2 remapping notation

The form of the radix 2 remapping is identical to the index bit mapping, and a similar notation could have been used. However, we have chosen to represent radix 2 remappings with an extended form of *cycle notation*. This notation is convenient when examining the operation of the parallel algorithms to be presented in section 4.4.

An index bit permutation is represented as a list of cycles to be performed from left to right. Unlisted device bits are assumed to be disabled in both the source and destination index bit maps, unless they can be seen to be enabled from their context, in which case they may be omitted for brevity.

As an example, given a source index bit mapping:

$$a_3 a_2 a_1 a_0$$

and a destination index bit mapping:

$$a_3 a_0 a_2 a_1$$

the resulting radix two remapping can be specified in cycle notation as:

$$(d_0, d_2, d_1)(d_3)$$

Source index bit map: $(a_2 \bar{a}_0 1 a_1)$

$$g_s(d_3 d_2 d_1 d_0) = d_1 d_3 d_0 d_2 \oplus 1001 = \bar{d}_1 d_3 d_0 \bar{d}_2$$

$$g_s^{-1}(0 a_2 a_1 a_0) = a_2 a_0 0 a_1 \oplus 0110 = a_2 \bar{a}_0 1 a_1$$

Destination index bit map: $(\bar{a}_2 \bar{a}_1 \bar{a}_0 1)$

$$g_d(d_3 d_2 d_1 d_0) = d_0 d_3 d_2 d_1 \oplus (1111)$$

$$g_d^{-1}(0 a_2 a_1 a_0) = a_2 a_1 a_0 0 \oplus (1111)$$

Map from source to destination

$$g_d^{-1}(g_s(d_3 d_2 d_1 d_0)) = g_d^{-1}(d_1 d_3 d_0 d_2 \oplus 1001)$$

$$= d_3 d_0 d_2 d_1 \oplus 0011 \oplus 1111$$

$$= \bar{d}_3 \bar{d}_0 d_2 d_1$$

$$= (m_0 \bar{p}_0 m_1 \sim)(\bar{p}_1) \text{ (cycle notation)}$$

Device data permutation ($|M|=2, |P|=2$)

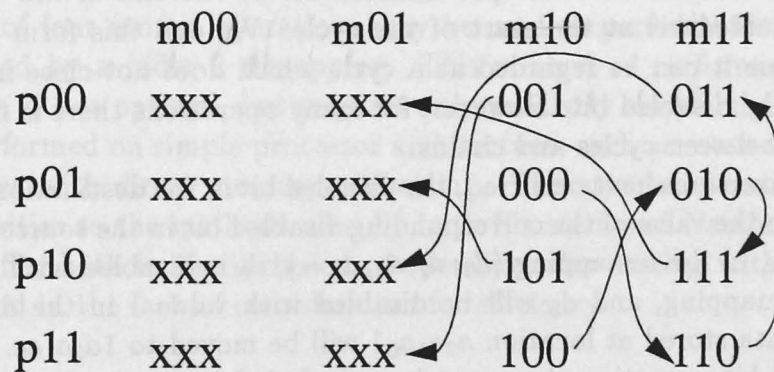


Figure 4.3: Two index bit maps define a radix 2 remapping

Inversion of device index bits

Inversion of a device index bit is indicated by an overbar, for example \bar{d}_0 . Inversions are performed *after* the enclosing permutation cycle has been performed; thus, (d_0, d_1, \bar{d}_2) is equivalent to both $(d_0, d_1, d_2)(\bar{d}_2)$ and $(\bar{d}_1)(d_0, d_1, d_2)$.

Disabled device index bits and index bit chains

A bit which is disabled in the source mapping is indicated by a trailing $.$ or \sim on the bit if it is set to zero or one respectively. Device index bits which are disabled in both the source and the destination map are indicated $(\bar{d}_i.)$ if d_i is set to zero, or $(d_i \sim)$ if d_i is set to one.

A disabled bit may be inverted in the remapping; $(\bar{\bar{d}}_i.)$ indicates d_i is set to zero in the source map and one in the destination map, and $(\bar{\bar{d}}_i \sim)$ indicates d_i is set to one in the source map and zero in the destination map.

A remapping may include some bits which are disabled in the source mapping and enabled in the destination mapping. The remapping may still be represented in cycle notation, because the defining index bit permutation can be used to write the cycles. When there is more than one disabled device bit, the defining index permutation is not unique. However, there is exactly one description of the remapping in which there is at most one disabled device bit in each index bit cycle, and we always choose this permutation to represent the remapping.

The disabled bit is moved by the index bit permutation just as the enabled bits are, thus the disabled bit in the source mapping is moved to a disabled bit in the destination mapping. We always represent a cycle containing a disabled bit in cycle notation with the pre-disabled bit at the end of the cycle, and the post-disabled bit at the start of the cycle. We call this form of cycle a *chain* because it can be regarded as a cycle which does not close in on itself because of the disabled bit. However, for many operations there is no need to distinguish between cycles and chains.

Before inversions have occurred, the disabled bit in the destination mapping will be set to the value of the corresponding disabled bit in the source mapping. For example, in the remapping $(d_3, d_2, d_1, d_0 \sim)$, d_0 is disabled with value 1 in the source mapping, and d_3 will be disabled with value 1 in the destination mapping; data stored at location $a_2a_1a_01$ will be moved to $1a_2a_1a_0$.

Disabled bit operations have not been included in previous expositions of radix 2 remapping. A table showing a selection of the 36 possible remappings that can be specified with two index bits are shown in Figure 4.4.

Index bit cycles vs. Data cycles

It is important to distinguish between *index bit cycles*, which are cycles in the conceptual movement of index bits, versus *data cycles*, which are cycles

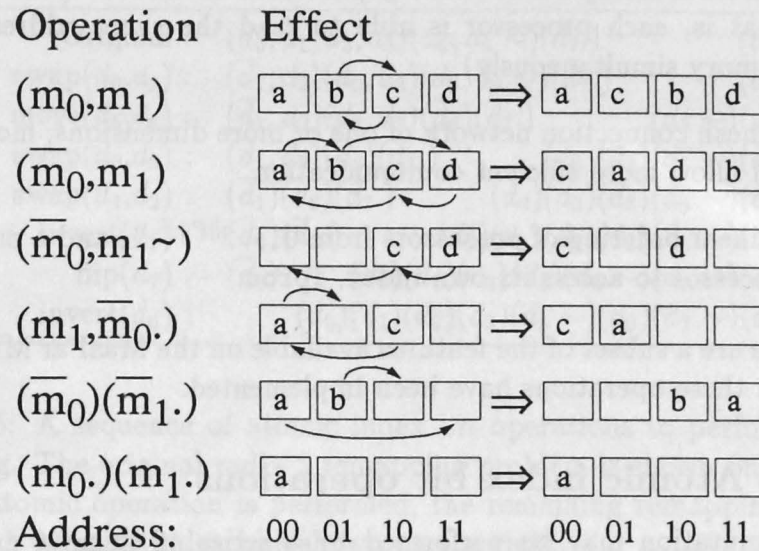


Figure 4.4: Some remappings on two index bits. a, b, c, d are data items stored in memory; blanks indicate disabled addresses

in the movement of data elements on the device. Similarly, it is important to distinguish between *index bit chains* in the remapping and *data chains* in the actual movement of data values.

4.3 Remapping with atomic operations

A set of four atomic operations may be used to perform the data movement specified by a radix 2 remapping. This method of performing the transformation is not optimal, but requires only relatively simple algorithms and can be performed on simple processor architectures such as mesh connected architectures with direct memory addressing. The atomic operations have many similarities to the methods used by both Flanders and Nassimi et al. [20, 56], but differ both in the inclusion of operations for dealing with disabled index bits and in the lack of consideration for efficiency.

4.3.1 Assumed architectural features

The operations outlined in the following sections may be performed on a processor architecture with the following features:

- A tightly coupled uni-processor to control the SIMD array to allow recursion and global data broadcasting

- A local memory for each array processor accessible by direct addressing (that is, each processor is able to read the same address in its local memory simultaneously)
- A mesh connection network of one or more dimensions; more dimensions will allow more efficient communication
- A linear ordering of processors from $0 \dots 2^{\|P\|} - 1$, and a means for every processor to access its own index, *iproc*.

These are a subset of the features available on the MasPar MP-1 computer, on which these operations have been implemented.

4.3.2 Atomic index bit operations

Any permutation may be performed by a series of element exchanges, and similarly, an index bit permutation may be performed as series of index bit exchanges. Thus, a radix 2 remapping may be performed as a series of index bit exchanges followed by a sequence of single index bit inversions.

The four atomic operations used to perform the radix 2 remapping are:

swap Swaps two enabled index bits d_i and d_j :

$$\begin{aligned} D_{dst} &= \text{swap}(D_{src}, i, j) \\ &\equiv \\ D_{dst}(d_{\|D\|-1} \dots d_j \dots d_i \dots d_0) &= D_{src}(d_{\|D\|-1} \dots d_i \dots d_j \dots d_0) \end{aligned}$$

invert Inverts an enabled index bit d_i :

$$\begin{aligned} D_{dst} &= \text{invert}(D_{src}, i) \\ &\equiv \\ D_{dst}(d_{\|D\|-1} \dots \bar{d}_i \dots d_0) &= D_{src}(d_{\|D\|-1} \dots d_i \dots d_0) \end{aligned}$$

move Swaps an enabled index bit d_i with a disabled (and therefore constant) index bit d_j :

$$\begin{aligned} D_{dst} &= \text{move}(D_{src}, i, j) \\ &\equiv \\ D_{dst}(d_{\|D\|-1} \dots d_j \dots d_i \dots d_0) &= D_{src}(d_{\|D\|-1} \dots d_i \dots d_j \dots d_0) \end{aligned}$$

After this operation, d_i will be disabled with the pre-operation value of d_j and d_j enabled.

Original :	$(d_0, \bar{d}_1, d_2, d_3)(d_5, \bar{d}_6 \sim)(\bar{d}_7.)$	(d_8)
swap(d_0, d_2) :	$(\bar{d}_1, d_2)(d_0, d_3)(d_5, \bar{d}_6 \sim)(\bar{d}_7.)$	(d_8)
move(d_5, d_6) :	$(\bar{d}_1, d_2)(d_0, d_3)(\bar{d}_6)(\bar{d}_7.)$	$(d_5 \sim)(d_8)$
swap(d_0, d_3) :	$(\bar{d}_1, d_2)(\bar{d}_6)(\bar{d}_7.)$	$(d_0)(d_3)(d_5 \sim)(d_8)$
swap(d_1, d_2) :	$(\bar{d}_1)(\bar{d}_6)(\bar{d}_7.)$	$(d_0)(d_2)(d_3)(d_5 \sim)(d_8)$
invert(d_1) :	$(\bar{d}_6)(\bar{d}_7.)$	$(d_0)(d_1)(d_2)(d_3)(d_5 \sim)(d_8)$
flip(d_7) :	(\bar{d}_6)	$(d_0)(d_1)(d_2)(d_3)(d_5 \sim)(d_7 \sim)(d_8)$
invert(d_6) :		$(d_0)(d_1)(d_2)(d_3)(d_5 \sim)(d_6)(d_7 \sim)(d_8)$

Figure 4.5: A sequence of atomic index bit operations to perform a radix 2 remapping. The original radix 2 remapping problem is shown on the top line. As each atomic operation is performed, the remaining remapping problem is shown with identity index bits shown on the right.

flip Inverts a disabled (and therefore constant) index bit d_i :

$$\begin{aligned}
 D_{dst} &= \text{flip}(D_{src}, i) \\
 &\equiv \\
 D_{dst}(d_{||D||-1} \dots \bar{d}_i \dots d_0) &= D_{src}(d_{||D||-1} \dots d_i \dots d_0)
 \end{aligned}$$

Although each operation to be described is only mentioned in terms of one or two index bits, it must be remembered that the actual data movement operation to be performed also depends on which device index bits are enabled at the time of the operation.

Note that **swap** could be used instead of **move**, and **invert** used instead of **flip** to perform these operations, but this would entail unnecessarily moving data from disabled addresses; although **swap** and **invert** are sufficient for performing arbitrary radix 2 remapping operations, they are not by themselves atomic because **move** and **flip** cannot be defined in terms of **swap** and **invert**. Figure 4.5 shows an example sequence of atomic operations used to simplify a radix 2 remapping.

4.3.3 Efficient use of atomic index bit operations

Because many radix 2 remappings may be performed by more than one sequence of atomic operations, there is great scope for finding efficient sequences of atomic operations for performing them. Improvements may also be achieved by combining sequences of atomic operations into more efficient compound operations. Ideally, all the data movement in a remapping should occur in a single step per memory address, and the algorithms to be presented in the

next section achieve this goal in many cases. For simpler architectures, however, more steps are required; Nassimi and Sahni's algorithm is optimal on a multidimensional mesh connected computer, but only handles processor index bit permutations and inversions [56]. In the implementation of Parallel Data Transforms on the AMT-DAP, many operations are combined to improve efficiency [1, 20, 21].

With the handling of disabled index bits, specifically the **move** operation, finding an optimal sequence of operations becomes much more difficult. After a **move** operation the data access requirements change; for example, after moving a memory index bit to a processor index bit any subsequent operations will require half the number of memory accesses. Another complicating factor is that in a real implementation of atomic bit operations, the time taken by the various operations is dependent on the enabled bits in the mapping and quite a complicated quantity to calculate.

However, good descriptions of atomic index bit permutation algorithms can be found in the work of Flanders and Nassimi et al. [20, 56], and they will not be considered further here. A description of an implementation of atomic radix 2 remapping operations can be found in chapter 5.

4.4 Optimal radix 2 remapping

Parallel memory accesses and communication operations in a massively-parallel distributed memory computer are often a large cost compared to simple intra-processor or uni-processor operations. As long as the computational overhead is modest, the most effective way to ensure that radix 2 remapping can be performed quickly is to limit the number of memory accesses and uses of the communication network. Ideally, all the processors containing data should be operating continuously, and each data item should be read from memory, passed through the communication network, and written to memory exactly once. We develop algorithms which are optimal in usage of the communications network, as each data element is passed through the network at most once. The use of memory may be sub-optimal, because data elements may sometimes be read from, and written to, memory more than once. Algorithms for in-place remapping are developed first, and it is shown how they may be modified for remapping by copying from one array to another.

4.4.1 Assumed architectural features

The optimal algorithms described in the following sections require the following architectural features:

- A tightly coupled uni-processor to control the SIMD array to allow recursion

- A fast mechanism for broadcasting data from the uni-processor to the SIMD array
- A fast mechanism for transferring data from a single SIMD processor to the uniprocessor to allow distributed storage of large lists
- An indirect addressing capability, allowing each processor in the array to access a different address
- A flexible interconnection network, such as a cross-bar switch, to allow processors to be connected in arbitrary permutations
- A large set of fast parallel registers to minimize processor array memory accesses to often-used data; each PE on the MasPar MP-1 has 48 registers of 32 bits each, of which 40 registers are available to programmers.
- A linear ordering of processors from $0 \dots 2^{\|P\|} - 1$, and a means for every processor to access its own index, *iproc*.

These are a subset of the features available on the MasPar MP-1 computer, on which the algorithms were developed. On the MasPar, each crossbar connection may be shared by a *cluster* of several processors. This limitation will only increase communication time by a constant factor, as is shown in chapter 5.

4.4.2 Types of cycles

When examining lists of cycles used to specify a radix 2 remapping, it will be useful to distinguish between various types of index bit permutation cycle.

- (*m*) represents a memory index bit (*m*-bit) identity cycle
- (*p*) represents a processor index bit (*p*-bit) identity cycle
- (*m**) represents a cycle containing only *m*-bit permutations and/or inversions
- (*p**) represents a cycle containing only *p*-bit permutations and/or inversions
- (*mp**) represents a cycle containing exactly one *m*-bit and multiple *p*-bits
- *mixed* cycles contain a mixture of *m*-bits and *p*-bits and do not fall into any of the other categories.

It will also be useful to distinguish between *even-parity* (*mp**) cycles, containing an even number (including zero) of index bit inversions, and *odd-parity* (*mp**) cycles.

4.4.3 A recursive approach

By working up recursively from components designed to handle simpler types of permutation cycle, we will show how efficient radix 2 remapping algorithms can be built.

The components in the remapping algorithm, in approximately bottom-up order, are as follows:

- Processor bit cycles (p^*), including single processor bit inversions (\bar{p})
- Memory bit cycles (m^*), including single memory bit inversions (\bar{m})
- Combined (p^*) and (m^*) cycles
- Conversion of mixed cycles to (m^*) and (mp^*) cycles
- Single memory bit/multiple processor bit cycles (mp^*)
- Identity cycles (m) and (p)

4.4.4 (p^*) cycles

(p^*) cycles are easy to perform using a computer with a crossbar switch connecting all the processors. Because no m -bits participate in the permutation, only one byte per processor will be affected. To perform the permutation, every pre-enabled processor reads a single byte from memory and sends it through the communications network to its destination processor; every post-enabled processor then writes the received byte back to memory.

A function to perform (p^*) cycles, `permute_p`, is shown in function 4.1. Global variables are used in all routines to make it clear that all variables are stored in processor registers, and to ensure that no parameter stack is necessary. A stack is necessary for storing return addresses to enable recursion to take place, but these stack operations can be performed wholly within the fast uniprocessor.

In order to accommodate the offset corresponding to the constant values associated with disabled memory bits, and to allow the integration of other types of cycle, an offset, `moffset`, may be passed to access bytes from the memory array other than the first.

The pre-enabled flag `svalid` can be pre-computed by calculating processors containing enabled addresses in E_{src} , as defined in section 4.2.1. When `moffset` contains only enabled addresses, it is sufficient to limit the calculation of `svalid` and `dvalid` to calculate enabled processors only.

The destination processor, `dproc`, may be computed by permuting and inverting the bits of `iproc` as defined by the remapping. As with the valid flags, `dproc` may be computed once before performing the permutation.

Function 4.1 (p^*) permutations

```

plural char *m;           /* Base address of memory array */
plural int moffset;       /* Offset for flipped memory bits etc. */
plural int dproc;         /* Destination processor */
plural int svalid;        /* 1 if proc is enabled in source map */
plural int dvalid;        /* 1 if proc is enabled in destination map */
plural char z;            /* Temporary data storage */

permute_p()
{
    if (svalid) {
        z = *(m+moffset);
        router[dproc].z = z;
    }
    if (dvalid) {
        *(m+moffset) = z;
    }
}

```

4.4.5 (m^*) cycles

When performing memory index permutations, every data element to be moved must be part of either a data cycle or a data chain, so we may perform a memory permutation with at most one read/write per memory address in the data array by traversing a linked list containing lists of permutation cycles and chains, as shown in figure 4.6. A function to perform memory permutations is shown in function 4.2, `permute_m`. The code for chains is not shown; moving data in a chain is performed in the same way as moving data in a cycle, except a read at the start of the chain and a write at the end of the chain are not necessary.

The function `permute_m` uses the processor array as a linear array to store a linked list of data permutation cycles in the plural variables `offset`, `next_off` and `next_cyc`. Because processor bits are not being permuted, the values of the variables `svalid` and `dvalid` are actually the same.

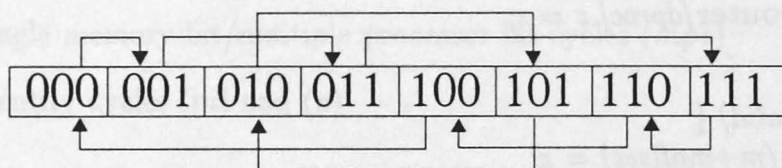
In order to compute the linked list representing the data permutation cycles, a mapping is first defined between data storage locations and individual processors by assigning m -bits participating in the permutation to processor index bits (not to be confused with the p -bits involved in the remapping). All processors thus mapped now contain a source offset into the memory array,

4.4.3 A recursive approach

Function 4.1 (p) permutation
 By writing an array in memory and reading it back in a different order, a permutation can be achieved. For example, if the array [0,1,2,3,4,5,6,7] is read in the order [4,6,7,3,1,0,5,2], the permutation is (4,6,7,3,1,0,5,2). This can be represented as a linked list where each node contains a number and a pointer to the next node in the sequence. The sequence of nodes is 4 → 6 → 7 → 3 → 1 → 0 → 5 → 2, with the last node (2) pointing to a null pointer (represented by a ground symbol).

Bit cycle: mm^* cycle $(\overline{m_0}, m_1, m_2)$

Data cycle: (0,1,3,7,6,4)(2,5)



Permutation as linked list:

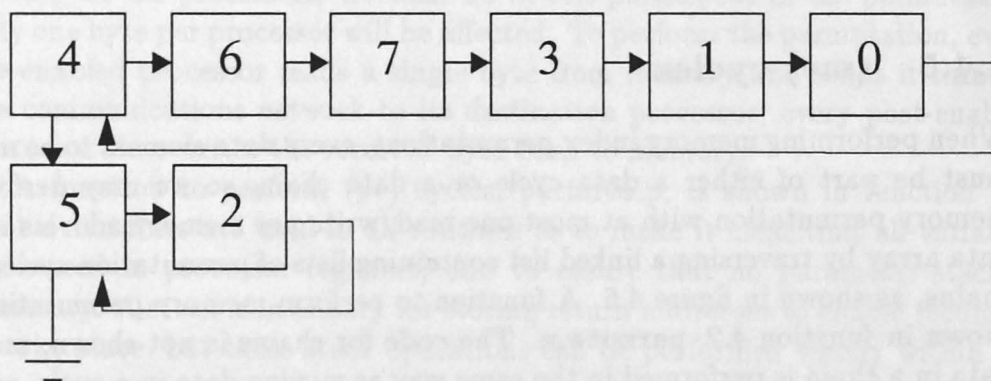


Figure 4.6: A memory permutation as a linked list

Function 4.2 (*m**) permutations

```

plural char *m;                /* Base address of memory array */
plural int moffset;            /* Offset for flipped memory bits etc. */
plural int svalid;             /* 1 if proc is enabled in source map */
plural int dvalid;             /* 1 if proc is enabled in destination map */
plural char z0;                /* Storage for data bytes */
plural char z1;
plural int o0;                 /* Storage for data offsets */
plural int o1;
plural int offset;             /* All offsets used in memory permutation */
plural int next_off;           /* proc with next offset in cycle or chain */
plural int next_cyc;           /* proc with next cycle or chain */
int cycles_start;              /* proc with first cycle */
int chains_start;              /* proc with first chain */

permute_m()
{
    int cycles_current = cycles_start;

    while (cycles_current != -1) {
        int elt_current = cycles_current;        /* Start of data cycle */

        o0 = proc[elt_current].offset;           /* Get first elt in cycle */

        if (svalid) z0 = *(m+moffset+o0);        /* Make space */
        elt_current = proc[elt_current].next_off; /* Next elt */

        while (elt_current != cycles_current) {
            o1 = o0;
            o0 = proc[elt_current].offset;

            if (svalid) z1 = *(m+moffset+o0);
            if (dvalid) *(m+moffset+o1) = z1;
            elt_current = proc[elt_current].next_off;
        }

        if (dvalid) *(m+moffset+o0) = z0;

        cycles_current = proc[cycles_current].next_cyc;
    }
    /* Code for chains here */
}

```


offset. The destination offset may be computed from the source offset by applying to it the (m^*) index bit permutations. Every processor can compute the index of the processor containing the destination location corresponding to its source offset by mapping the bits of the computed destination offset back to processor bits, every processor. As the direction of each linked list representing a cycle is from destination offset to source offset, this pointer can be reversed and stored in `next_off`, which can be accomplished in one router step.

All the cycles and chains have now been generated as cyclic linked lists. To apply them it is necessary to make a list of them by finding the first element in each.

It does not matter which element of a cycle is regarded as the first. For simplicity, we will regard the processor with the minimum value of `iproc` as being the first element in a cycle.

For each chain, however, the first element is always a disabled source offset and the final element will contain the corresponding disabled destination offset.

The first element in every cycle and chain can be propagated to every processor in a maximum of $\|P\|$ steps by the use of a recursive doubling algorithm, where at each step the head of the list is propagated to a doubling number of processors. The longest data cycle occurring in an index bit permutation containing $\|P\|$ bits is $2\|P\|$ (corresponding to $(\bar{m}_0, m_1 \dots m_{\|P\|-1})$), thus the number of doubles required is at most $\lceil \log_2 2\|P\| \rceil$.

Once we have a group of processors that can identify themselves as the head of a cycle or chain, a further recursive doubling can be used to form a linked list of chains and cycles in at most $\|P\|$ steps, which is stored in `next_cyc`.

The communication required for this recursive doubling may be performed using the crossbar switch.

The first element in either the lists of cycles or chains can be found by determining the minimum value of `iproc` containing the head of a cycle or chain list, and stored in `cycles_start` and `chains_start`.

If the number of m -bits appearing in (m^*) permutations is k , the number of data memory locations to be permuted can be no greater than 2^k . If $k > \|P\|$, it is necessary to break the memory permutation into several parts.

There are two ways to do this. The easiest is to break the index bit permutation into several parts to be performed separately, each with $\|P\|$ m -bits or fewer. As an example, if $\|P\| = 10$ the (m^*) permutation

$$(m_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9, m_{10}, m_{12}, m_{13}, m_{14})$$

could be performed by application of the two (m^*) permutations

$$(m_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9) \text{ and } (m_0, m_{10}, m_{12}, m_{13}, m_{14})$$

An alternative method, instead of splitting the index bit cycles, splits the data cycles into several data permutations. This method has the advantage

that only part of the data array need be permuted at each step. Unfortunately, it also has the disadvantage that the number of data permutations increases substantially; in the example given, instead of two data permutations being computed and possibly retained, thirty-two are required.

4.4.6 Simultaneous (m^*) and (p^*) cycles

Memory and processor permutations may be combined into one function, `permute_mp`, by sending each data element to its destination processor while performing the memory permutation.

4.4.7 Transforming mixed cycles into (mp^*) cycles

The algorithms we have presented for performing (m^*) and (p^*) are optimal, in that they may be performed using a minimum of memory accesses and uses of the communications network. An optimal algorithm for performing mixed cycles would also be desirable. Given enough registers for storage of temporary values and data destinations, such algorithms do exist (see section 6.4). Given only a small number of processor registers, these algorithms cannot be used.

However, by converting any mixed cycle into an (m^*) cycle followed by a combination of (m^*) and even-parity (mp^*) cycles optimal performance can be achieved in the use of the communications network, if not in the use of memory. The following steps show how this transformation may be effected.

Isolating memory bits

Assume we have several mixed cycles. Each cycle which contains groups of at least two adjacent memory bits can be modified to contain isolated memory bits as follows:

- Write the cycle so that it begins with a group of memory bits and ends with a processor bit, i.e. as

$$(m_a, m_b \dots m_c, p_d \dots p_e)$$

where the bits between m_a and m_c are memory bits and the bits between p_d and p_e are arbitrary.

- Split the cycle into three; if any bit is inverted in the original cycle and is factored into several cycles, it should only be inverted in the final cycle.
 - i. (m_a, m_c)
 - ii. $(m_a, p_d \dots p_e)$
 - iii. $(m_b \dots m_c)$

- Repeat the above on cycle (ii) until it contains only isolated memory bits

Now we have an index bit permutation consisting of several memory permutations and several mixed permutations containing isolated memory bits. All the type (i) cycles appear in front of the type (ii) cycles, and can be performed in a separate step. Type (iii) memory cycles contain no bits in common with the mixed cycles, and so may be performed independently of, and simultaneously with, them.

Splitting into $(mp*)$ and $(m*)$ cycles

The mixed cycles which contain more than one isolated memory bit can be modified to contain exactly one memory bit as follows:

- Write the cycle so that it begins with a single memory bit, i.e. as

$$(m_a, p_c \dots p_d, m_b, p_e \dots p_f)$$

where the bits between p_c and p_d are processor bits and the bits between p_e and p_f are arbitrary

- Split the cycle into three, handling inverted bits as before:
 - i. (m_a, m_b)
 - ii. $(m_b, p_c \dots p_d)$
 - iii. $(m_a, p_e \dots p_f)$
- Repeat the above until the cycle (iii) has been split into isolated $(mp*)$ cycles

Now we have an index permutation consisting of several $(m*)$ cycles and several $(mp*)$ cycles. As with the conversion from mixed cycles to mixed cycles with one isolated memory bit, type (i) memory permutations can be moved to the front of the $(mp*)$ cycles and performed in a separate step.

Converting to even-parity $(mp*)$ cycles

Finally, if an $(mp*)$ cycle does not contain an even number of inversions, it can be modified as follows:

- Write the cycle so that it begins with its single memory bit, i.e. as

$$(m_a, p_b \dots p_c)$$

where the bits between p_b and p_c are processor bits.

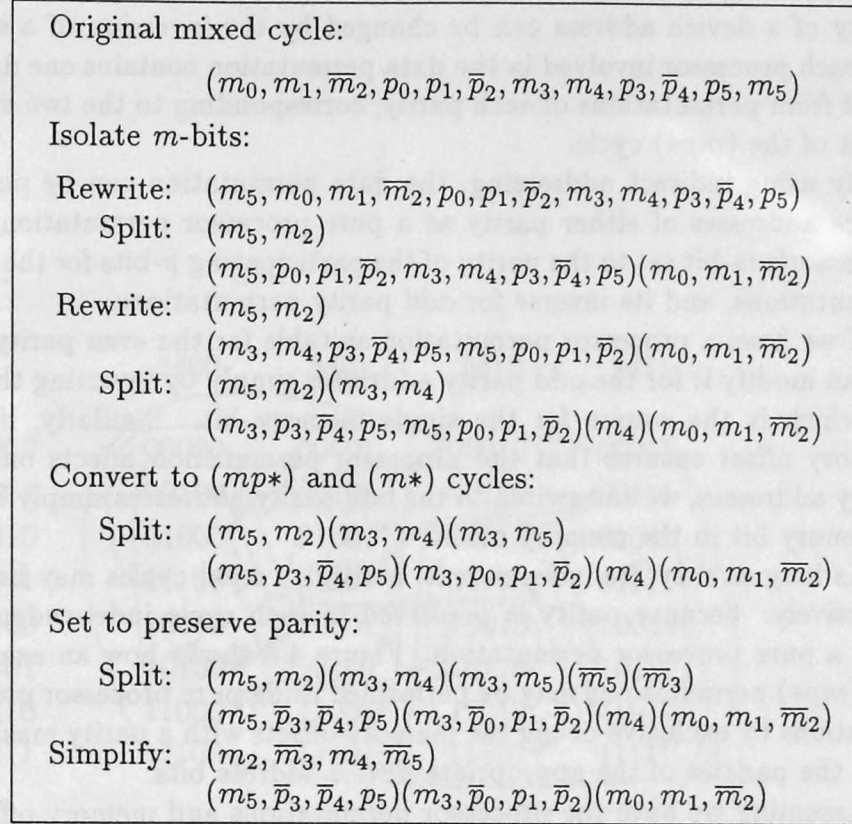


Figure 4.7: Transformation of mixed cycle into (m^*) and (mp^*) cycles (with identity cycles omitted for brevity)

- Split the cycle into two cycles:

- (\bar{m}_a)
- $(m_a, \bar{p}_b \dots p_c)$

As before, each memory bit may be inverted with all the other memory permutations in a pre-processing step. An example of splitting a mixed cycle into (m^*) and (mp^*) cycles is shown in figure 4.7.

4.4.8 Even-parity (mp^*) cycles

An even-parity (mp^*) cycle has several desirable properties. Because only one memory bit is present in the cycle, only two memory elements in each processor are affected by the permutation. Because the cycle contains an even number of bit inversions, the parity of the permuted bits is preserved by the permutation. This allows us to split the data permutation into two disjoint permutations,

corresponding to permuted device bits of odd and even parity. Because the parity of a device address can be changed by the inversion of a single index bit, each processor involved in the data permutation contains one data element offset from permutations of each parity, corresponding to the two values of the m -bit of the (mp^*) cycle.

By using indirect addressing, the data permutation can be performed on device addresses of either parity as a pure processor permutation, with each processor's m -bit set to the parity of the participating p -bits for the even parity permutations, and its inverse for odd parity permutations.

If we have a processor permutation suitable for the even parity addresses, we can modify it for the odd parity addresses simply by inverting the processor bit which is the source for the single memory bit. Similarly, if the initial memory offset ensures that the processor permutation affects only the even parity addresses, we can switch to the odd parity addresses simply by inverting a memory bit in the memory offset.

As long as they are independent, multiple (mp^*) cycles may be performed recursively: because parity is preserved in each cycle independently we still have a pure processor permutation. Figure 4.8 shows how an example set of two (mp^*) permutations may be performed using pure processor processor permutations by exclusive-or'ing the memory offsets with a parity mask generated from the parities of the appropriate device address bits.

Assuming we have the processor permutations and memory offsets set up, function 4.3, `permute_x`, will perform (mp^*) permutations.

In order for this algorithm to operate, `moffset` and `dproc` must be pre-computed. `moffset` is set so that all memory locations initially read are part of the even parity offsets of (mp^*) cycles. To perform the even parity processor permutation first, we need to set each bit in `moffset` that corresponds to a memory bit in an (mp^*) cycle to the parity of the processor bits in that cycle.

The calculation of `dproc` will include (p^*) cycles, as described for the function `permute_p`, supplemented with (mp^*) cycles. The bits in `dproc` participating in (mp^*) cycles are calculated exactly as for (p^*) cycles, except that the value of the processor bit appearing after the memory bit in the (mp^*) cycle is determined by the value of that memory bit, which is initially just the parity of the processor bits in the cycle.

4.4.9 Identity cycles

Identity cycles may be included with the (m^*) and (p^*) cycles, and this is the approach taken with identity processor bits.

A different approach is taken with identity memory bits; by keeping the number of memory bits in the memory cycles small, it may be possible to ensure that the length of the list of offsets in the memory permutation is smaller than the number of processors. Function 4.4, `permute_i`, will recursively call the

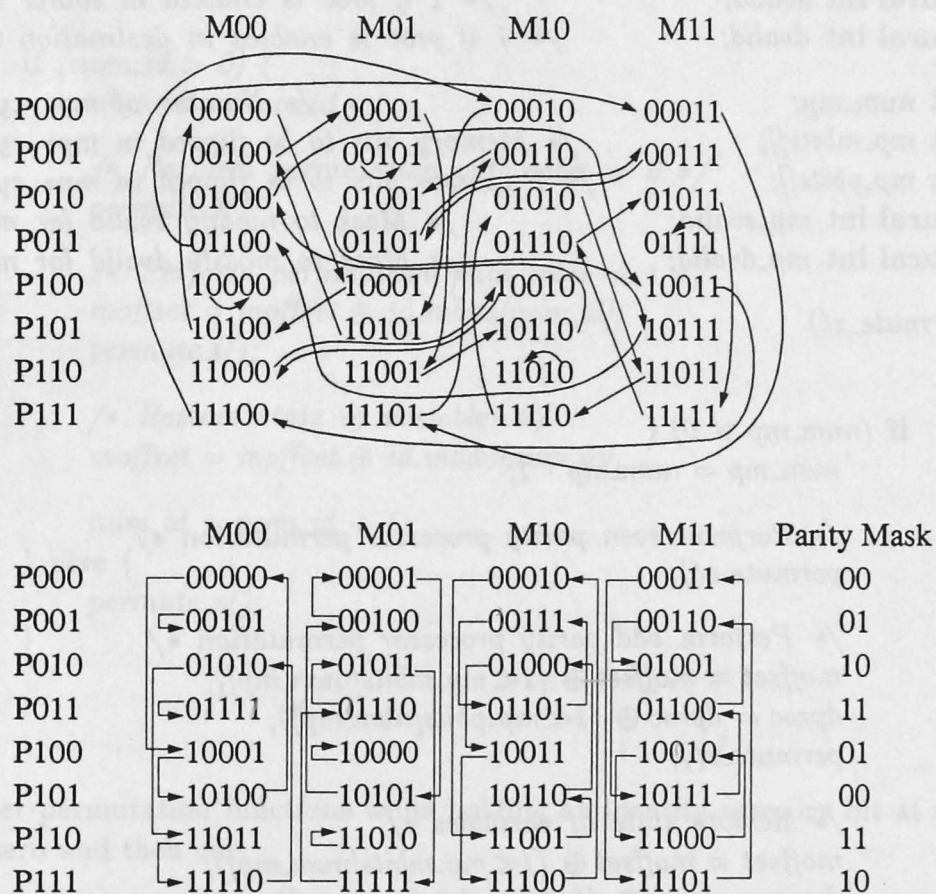


Figure 4.8: Parity masking to align the (mp^*) cycles $(\overline{m}_0, p_0, \overline{p}_2)(m_1, p_1)$

Function 4.3 (*mp**) permutations

```

plural char *m;                /* Base address of memory array */
plural int moffset;            /* Offset for flipped memory bits etc. */
plural int dproc;              /* Destination processor */
plural int svalid;             /* 1 if proc is enabled in source map */
plural int dvalid;            /* 1 if proc is enabled in destination map */

int num_mp;                    /* Number of mp* cycles */
int mp_mbits[];               /* Memory bits to be flipped in mp* cycles */
int mp_pbits[];              /* Processor bits to be flipped in mp* cycles */
plural int mp_svalid;         /* Mask to modify svalid for mbits */
plural int mp_dvalid;         /* Mask to modify dvalid for mbits */

permute_x()
{
    if (num_mp > 0) {
        num_mp = num_mp - 1;

        /* Perform even parity processor permutation */
        permute_x();

        /* Perform odd parity processor permutation */
        moffset = moffset  $\oplus$  (1 $\ll$ mp_mbits[num_mp]);
        dproc = dproc  $\oplus$  (1 $\ll$ mp_pbits[num_mp]);
        permute_x();

        /* Restore state of variables */
        moffset = moffset  $\oplus$  (1 $\ll$ mp_mbits[num_mp]);
        dproc = dproc  $\oplus$  (1 $\ll$ mp_pbits[num_mp]);

        num_mp = num_mp + 1;
    } else {
        permute_mp();
    }
}

```

Function 4.4 *Identity cycles*

```

plural char *m;                /* Base address of memory array */
plural int moffset;            /* Offset for flipped memory bits etc. */

int num_id;                    /* Number of memory identity cycles */
int id_mbits[];                /* Identity bit numbers */

permute_i()

{
    if (num_id > 0) {
        num_id = num_id - 1;

        /* Perform permutation with mbit = 0 */
        permute_i();

        /* Perform permutation with mbit = 1 */
        moffset = moffset  $\oplus$  id_mbits[num_id];
        permute_i();

        /* Restore state of variables */
        moffset = moffset  $\oplus$  id_mbits[num_id];

        num_id = num_id + 1;
    } else {
        permute_x();
    }
}

```

other permutation functions while holding an identity memory bit at a value of zero and then one.

4.4.10 Remapping data while copying

With a set of algorithms to remap an array in-place on a device, it is possible to perform a remap copying one array to another by reading the data from the source array and writing it into the destination array.

In some situations, care must be taken to ensure that intermediate states in the remapping process do not overflow the bounds of the memory array. As an example, when remapping a data array that is stored in a long memory array in only a few processors to a short memory array stored in a large number of processors, it may be necessary to perform an in-place remapping to a short-

representation in the long array with many processors before copying the data into the short memory array and remapping the data back again.

A similar problem occurs if trying to obtain speed-ups by remapping a subset of an array while wishing to preserve data in disabled storage locations; the nature of the transformations for mixed cycles do not guarantee that disabled storage locations will remain untouched during the intermediate stages of data remapping.

4.5 Summary

This thesis is intended to address two problems in computation with large multi-dimensional data sets: the *mapping problem*, or specifying the layout of data on a storage or computational device, and the *remapping problem*, or dynamically rearranging data on a device from one mapping to another. This chapter introduces efficient algorithms for performing data remapping on a restricted set of multidimensional data arrays.

The *index bit map* provides a method for specifying data mappings for data arrays, the length of whose dimensions are powers of two, to multidimensional devices whose dimensions are similarly constrained. This mapping technique is similar to the mapping vectors of Flanders' PDTs, except that the index bit map allows the inclusion of disabled device index bits. It is shown that the data movement required to remap a data array between two index bit maps may be performed using the technique of index bit permutation, including index bit inversion. Methods for performing this technique on mesh-connected parallel processors have been substantially explored by Nassimi and Sahni, Flanders, and Cruz. Using similar algorithms, we describe *Atomic index bit operations* which could be used to perform a remapping between two index bit maps using the mesh connections and direct memory addressing capabilities of a parallel processor.

The MasPar has some architectural features that differ from earlier SIMD machines, and we introduce an optimal algorithm for index bit permutation on a parallel computer with indirect memory addressing and a fixed-time global router network. The algorithm operates by breaking an arbitrary index bit permutation into six types of index bit permutation cycle, all of which may be performed simultaneously by a structure of nested recursive routines performing a different type of index bit permutation cycle:

- identity memory index bits (m) may be performed in two separate parts corresponding to a bit value of 0 or 1
- identity processor bits (p) require no overhead whatsoever
- memory index bit permutations ($m*$) may be performed using a distributed linked list

- processor index bit permutations (p^*) may be performed directly using the global router network
- single m -bit, multiple p -bit permutations (mp^*) may be performed using the global router and indirect memory addressing, using a base memory address modified by the parity of the device index bits included in the permutation cycle.

Some index bit permutations may need to be performed as a memory permutation followed by an index bit permutation consisting of cycles of these five types. The algorithm is equally applicable to in-place or copying remaps.

By combining the remapping algorithms described in Chapter 4 with a simplified form of the k -Tile format and PMPs described in Chapter 5, an efficient and useful tool can be implemented. This chapter describes an implementation of radix 2 PMPs for the MasPar MP-2 computer, and gives some related testing and performance results.

5.1 The 2^k -Tile format

The form of the 2^k -Tile format to be used for radix 2 parallel mapping functions, the radix 2 k -Tile or 2^k -Tile format, is defined in several ways to allow the algorithms of Chapter 4 to be used. All dimensions must have lengths which are powers of two, and only the following 2^k -Tile fields are included in the 2^k -Tile format:

a — The shape of the image space

k — The shape of the k -Tile space

s — The k -Tile source indicator

m — The mapping between the k -Tile and device spaces

d — The shape of the device space

The 2^k -Tile format is equivalent to the index bit map of Chapter 4. This may be shown by describing how to convert a 2^k -Tile format into an index bit map.

Chapter 5

Implementation of Radix 2 PMFs

By combining the remapping algorithms described in chapter 4 with a simplified form of the k -Tile format and PMFs described in chapter 3, an efficient and useful tool can be implemented. This chapter describes an implementation of radix 2 PMFs for the MasPar MP-1 computer, and gives some related testing and performance results.

5.1 The 2^k -Tile format

The form of the k -Tile format to be used for radix 2 parallel mapping functions, the *radix 2 ktile* or 2^k -Tile format, is restricted in several ways to allow the algorithms of chapter 4 to be used. All dimensions must have lengths which are powers of two, and only the following k -Tile fields are included in the 2^k -Tile format:

- a — The shape of the image space
- k — The shape of the k -Tile space
- s — The k -Tile sense indicator
- m — The mapping between the k -Tile and device spaces
- d — The shape of the device space

The 2^k -Tile format is equivalent to the index bit map of chapter 4. This may be shown by describing how to convert a 2^k -Tile format into an index bit map.

We have earlier described how to use two index bit maps to define a radix 2 remapping. This allows us to use 2^k -Tile formats to describe radix 2 remappings. With the use of the algorithms described in chapter 4, we can create a complete radix 2 PMF system.

5.1.1 Converting a 2^k -Tile format to an index bit map

A 2^k -Tile format may be converted to an index bit map using the following steps:

- Check the 2^k -Tile format for internal consistency
- Re-write the 2^k -Tile dimensions as k -Tile index bits
- Re-write the data and device dimensions using the k -Tile index bits
- Re-write the 2^k -Tile sense indicator as an inverting bit mask
- Pass the data index bits through the corresponding 2^k -Tile index bits to form a map from data index bits to device index bits
- Invert the device index bits corresponding to inverted 2^k -Tile index bits

As an example, the 2^k -Tile format:

a : [256; 256]
 k : [16, 16, 16, 16]
 s : [+, +, -, -]
 m : [0, 2, 1, 3]
 d : [256; 256]

may be written in index bits as:

a : [$k_7k_6k_5k_4k_3k_2k_1k_0$, $k_{15}k_{14}k_{13}k_{12}k_{11}k_{10}k_9k_8$]
 k : [$k_3k_2k_1k_0$, $k_7k_6k_5k_4$, $k_{11}k_{10}k_9k_8$, $k_{15}k_{14}k_{13}k_{12}$]
 s : [++++, +++++, ----, ----]
 m : [0, 2, 1, 3]
 d : [$k_3k_2k_1k_0$, $k_{11}k_{10}k_9k_8$, $k_7k_6k_5k_4$, $k_{15}k_{14}k_{13}k_{12}$]

Note that the assignment of 2^k -Tile index bits of lower significance to the first dimensions in the data and 2^k -Tile spaces is arbitrary, and a mapping that gives 2^k -Tile index bits of higher significance to the first dimensions would be equally valid. However, this arrangement has been chosen because the first device dimensions are of lowest significance in the device index.

From this form of the 2^k -Tile mapping, an index bit map may be derived:

$$\bar{a}_{15}\bar{a}_{14}\bar{a}_{13}\bar{a}_{12}a_7a_6a_5a_4\bar{a}_{11}\bar{a}_{10}\bar{a}_9\bar{a}_8a_3a_2a_1a_0$$

Any empty 2^k -Tile dimensions simply correspond to disabled device index bits, and the sense indicators of empty 2^k -Tile dimensions correspond to the sense of the disabled index bits.

5.1.2 A canonical form of the 2^k -Tile format

Although the 2^k -Tile format was intended to be as compact as possible, it is still possible that many 2^k -Tile formats may describe exactly the same mapping from the image space to the device space. This can make it difficult to determine if two 2^k -Tile formats are in fact identical. This can occur in the following situations:

- i. Empty 2^k -Tile dimensions occurring in the most significant positions in the device dimensions. For example:

$$\begin{array}{ll} \mathbf{a} : [128; 128] & \mathbf{a} : [128; 128] \\ \mathbf{k} : [128; 128; 8, 8] & \mathbf{k} : [128; 128] \\ \mathbf{s} : [+, +, +, +] & \equiv \mathbf{s} : [+, +] \\ \mathbf{m} : [0, 2; 1, 3] & \mathbf{m} : [0; 1] \\ \mathbf{d} : [1024; 1024] & \mathbf{d} : [128; 128] \end{array}$$

- ii. Data dimensions needlessly split. For example:

$$\begin{array}{ll} \mathbf{a} : [128; 128] & \mathbf{a} : [128; 128] \\ \mathbf{k} : [16, 8; 16, 8] & \mathbf{k} : [128; 128] \\ \mathbf{s} : [+, +, +, +] & \equiv \mathbf{s} : [+, +] \\ \mathbf{m} : [0, 1; 2, 3] & \mathbf{m} : [0; 1] \\ \mathbf{d} : [128, 128] & \mathbf{d} : [128; 128] \end{array}$$

- iii. Empty dimensions listed in varying order. For example:

$$\begin{array}{ll} \mathbf{a} : [128; 128] & \mathbf{a} : [128; 128] \\ \mathbf{k} : [128; 128; 2, 2] & \mathbf{k} : [128; 128; 2, 2] \\ \mathbf{s} : [+, +, +, +] & \equiv \mathbf{s} : [+, +, +, +] \\ \mathbf{m} : [2, 0; 3, 1] & \mathbf{m} : [3, 0; 2, 1] \\ \mathbf{d} : [256, 256] & \mathbf{d} : [256; 256] \end{array}$$

- iv. A three or more dimensional device treated as sub-dimensional. For example, on a 32×32 processor array:

$$\begin{array}{ll} \mathbf{a} : [128; 128] & \mathbf{a} : [128; 128] \\ \mathbf{k} : [32, 4; 32, 4] & \mathbf{k} : [32, 4; 32, 4] \\ \mathbf{s} : [+, +, +, +] & \equiv \mathbf{s} : [+, +, +, +] \\ \mathbf{m} : [1, 3; 0, 2] & \mathbf{m} : [1, 3; 0, 2] \\ \mathbf{d} : [16, 32, 32] & \mathbf{d} : [16; 1024] \end{array}$$

However, in some implementations the processor array may behave differently when configured as a two-dimensional 32×32 processor array as opposed to a linear array of 1024 elements, so this situation must be treated with care.

A 2^k -Tile format is called *canonical* only if it has the following properties:

- No empty k -Tile dimension appears in the most significant part of a device space dimension
- No adjacent k -Tile dimensions appear in increasing order within the same device dimension and the same data dimension
- Empty k -Tile dimensions are referenced in increasing order in the k -Tile/device map, \mathbf{m}
- Where appropriate, a device of three or more dimensions can be treated as a device of lower dimension.

Rather than applying multiple transformations to an existing 2^k -Tile format to convert it into canonical form, it is simpler to convert the 2^k -Tile format into an index bit map, and then back-convert this into canonical form. We have not undertaken to prove that the above four properties are sufficient for defining a canonical form of the 2^k -Tile format. As the representation of an index bit map is unique, the canonical form of a 2^k -Tile format may be defined as the 2^k -Tile format generated by converting the original 2^k -Tile format into an index bit map and then back-converting into a unique 2^k -Tile format.

5.2 Data types used by radix 2 PMFs

In order to represent 2^k -Tile formats, index bit maps, radix 2 remaps and algorithm-specific information for performing the remap, several data types are used within the radix 2 PMF system:

k -Tile format

The k -Tile format data structure 'ktile' contains four arrays to represent \mathbf{a} , \mathbf{k} , \mathbf{m} and \mathbf{d} , and a bit vector to represent \mathbf{s} . Because the lengths of the dimensions of the three spaces are powers of two, they may be stored as \log_2 of their actual value. This reduces storage space for the array and allows many computations on the dimensions to be performed using addition and subtraction instead of the slower multiplication and division.

Memory tag

The memory tag data structure 'mtag' represents a block of memory stored on the processor array: the size of the array, the address of the block of memory, and the k -Tile format currently associated with the memory.

Index bit map

The index bit map data structure 'kmap' represents an index bit map: two arrays define a bidirectional map between data index bits and device index bits, and two bit vectors represent device bit sense and device bit enable.

Radix 2 remap

The radix 2 remap data structure 'gmap' represents a radix 2 remap: two arrays define a bidirectional map between source and destination device index bits, and three bit vectors define pre-enabled, post-enabled, and post-inverted device index bits.

Radix 2 remap in cycle notation

The cycle notation data structure 'cyc' represents the same information as the gmap with supplementary information to define the index bit cycles: arrays list the start of each bit permutation cycle of each type (pp^*), (mm^*) and (mp^*)), and three bit vectors indicate identity device bits, device bits involved in simple permutations (m -bit-only and p -bit-only), and device bits involved in (mp^*) permutations.

Atomic remap operations

The atomic operations data structure 'rop' contains a list of atomic index bit operations for performing a radix 2 remapping. Because the implementation can use the optimal radix 2 algorithms, atomic remap operations are no longer required.

Remapping implementation

The remapping implementation data structure 'remap' contains all the information needed to perform an optimal radix 2 remapping operation. Because this operation can have multiple components, this structure is more complex than the other data structures. Information which is local to a single processor and linked lists are stored in distributed memory.

5.3 Functions used to access PMFs

To use the PMF system, several functions are provided for specifying k -Tile formats, declaring data memory, initializing data remaps and remapping data in-place or by copying.

Other useful functions are provided for generating k -Tile formats representing some standard mappings and for performing some useful transformations on existing k -Tile formats.

The following functions are a subset of those available in our MasPar MP-1 implementation of radix 2 PMFs.

5.3.1 k -Tile format manipulation

Declaring k -Tile format - *new_ktile(in ktile, out kid)*

This function takes a k -Tile format as an argument, which it checks and converts to a canonical form. If the format has already been specified, a usage count is incremented and no memory storage is required. If valid, a k -Tile identifier is output, otherwise a descriptive error message is printed and an error indicator returned.

Inquiring k -Tile format - *inquire_ktile(in kid, out ktile)*

This function takes a k -Tile identifier as an argument. If the identifier is valid, the appropriate k -Tile format is output. If invalid, a descriptive error message is printed and an error indicator returned.

Freeing k -Tile format - *free_ktile(in kid)*

This function takes a k -Tile identifier as an argument. If the identifier is invalid or if the k -Tile identifier is referred to by an mtag or a pair, a descriptive error message is printed and an error indicator returned. Otherwise, the usage count of the kid is decremented. If the usage count reaches zero the storage for the k -Tile is freed.

5.3.2 mtag manipulation

Declaring an mtag - *new_mtag(in mem, in size, out mid)*

This function takes a memory address and its size as arguments. If the memory defined is valid and does not overlap other declared mtags, an mtag id is output. Otherwise, a descriptive error message is printed and an error indicator returned.

Freeing an mtag - *free_mtag(in mid)*

This function takes an mtag identifier as an argument. If the identifier is valid, the mtag is freed. Otherwise, a descriptive error message is printed and an error indicator returned.

Inquiring an mtag - *inquire_mtag(in mid, out mtag)*

This function takes an mtag identifier as an argument. If the identifier is valid, the mtag is output. Otherwise, a descriptive error message is printed and an error indicator returned.

Associating a mapping with an mtag - *set_mtag_kid(in mid, in kid)*

This function takes an mtag identifier and a *k*-Tile identifier as an argument. A null *k*-Tile identifier may be passed to indicate that no mapping is attached to the specified memory. If either identifier is invalid or the mtag is too small for the mapping described by kid, a descriptive error message is printed and an error indicator returned. Otherwise, the ktile identifier is associated with the mtag identifier.

5.3.3 Remapping

Initializing a remap - *init_remap(in kid1, in kid2)*

This function takes as arguments two *k*-Tile identifiers. If the two identifiers describe valid mappings between arrays of the same shape and enough memory is available to represent the remapping, a mapping is generated to remap data from format kid1 to format kid2 (but not vice-versa). Otherwise, a descriptive error message is printed and an error indicator returned. If an attempt is made to initialize a remapping more than once, only one copy of the remapping is kept and an internal usage count ensures that the remapping is retained while it is needed.

Freeing a remap - *free_remap(in kid1, in kid2)*

This function takes as arguments two *k*-Tile identifiers. If either identifier is invalid or if the map between the *k*-Tile identifiers has not been initialized, a descriptive error message is printed and an error indicator returned. Otherwise, the usage count of the remap is decremented. If the usage count reaches zero the storage for the remap is freed.

Performing an in-place remap - *remap(in mid, in kid)*

This function takes as arguments an mtag identifier and a *k*-Tile identifier. If either identifier is invalid or if the *k*-Tile identifier is too large for the mtag memory, a descriptive error message is printed and an error indicator returned. Otherwise, the data stored at the address referred to by the mtag is remapped in place and the mtag's kid is updated to reflect the new mapping. Note that if a remap has previously been initialized for the two *k*-Tile formats, the operation will require no pre-initialization and will occur much faster.

Performing a copy remap - *copy(in mid1, in mid2, in kid)*

This function takes as arguments two mtag identifiers and a *k*-Tile identifier. If any identifier is invalid or if the *k*-Tile identifier is too large for the mtag memory, a descriptive error message is printed and an error indicator returned. Otherwise, the data stored at the address referred to by mtag1 is simultaneously remapped and copied to the memory referred to by mtag2, and mtag2's kid is updated to reflect the new mapping. Note that if a remap has previously been initialized for the two *k*-Tile formats, the operation will require no pre-initialization and will be faster.

5.3.4 Standard mappings

To ease the burden of programmers creating their own *k*-Tile formats, several functions have been provided to generate *k*-Tile descriptions of some standard mappings.

2d cut'n'stack mapping - *make_2dcs(in size, in x, in y, out ktile)*

This function takes as arguments the shape of a 2d image and a *k*-Tile structure. If the shape is valid, a 2d cut'n'stack *k*-Tile format is generated and output. If the shape is invalid, a descriptive error message is printed and an error indicator returned.

2d hierarchical mapping - *make_2dh(in size, in x, in y, out ktile)*

This function takes as arguments the shape of a 2d image and a *k*-Tile structure. If the shape is valid, a 2d hierarchical *k*-Tile format is generated and output. If the shape is invalid, a descriptive error message is printed and an error indicator returned.

1d cut'n'stack mapping - *make_1dcs(in size, in x, in y, out ktile)*

This function takes as arguments the shape of a 2d image and a *k*-Tile structure. If the shape is valid, a 1d cut'n'stack *k*-Tile format is generated and

output. If the shape is invalid, a descriptive error message is printed and an error indicator returned.

1d hierarchical mapping - *make_1dh(in size, in x, in y, out ktile)*

This function takes as arguments the shape of a 2d image and a *k*-Tile structure. If the shape is valid, a 1d hierarchical *k*-Tile format is generated and output. If the shape is invalid, a descriptive error message is printed and an error indicator returned.

1d scan mapping - *make_1dscan(in size, in x, in y, out ktile)*

This function takes as arguments the shape of a 1d image and a *k*-Tile structure. If *y* is smaller than or equal to *nproc*, a 1d scan mapping, else a 1d hierarchical mapping, is generated and output. If the shape is invalid, a descriptive error message is printed and an error indicator returned.

5.3.5 Geometrical transformations

Once a *k*-Tile format has been generated, it is possible to perform several geometric transformations on the dimensions within the *k*-Tile format. Note that these functions will work on any valid *k*-Tile formats, not just those generated with the *make_...* functions. The operations may also be composed, but it must be remembered that because the operations are performed on the image dimensions rather than the device dimensions, they will appear to act as pre-operations rather than post-operations.

Transposition of axes - *xpose(in dim1, in dim2, inout ktile)*

This function will transpose two image dimensions in a *k*-Tile format. Because there are several possible ways of transposing rectangular mappings, the two dimensions to be transposed must be the same length.

Reversal of axis - *reverse(in dim, inout ktile)*

This function will reverse an image dimension in the *k*-Tile format.

Index bit reversal of axis - *bitrev(in dim, inout ktile)*

This function will bit-reverse the index bits of an image dimension in the *k*-Tile format. This function is useful for the remapping prior to a radix 2 FFT.

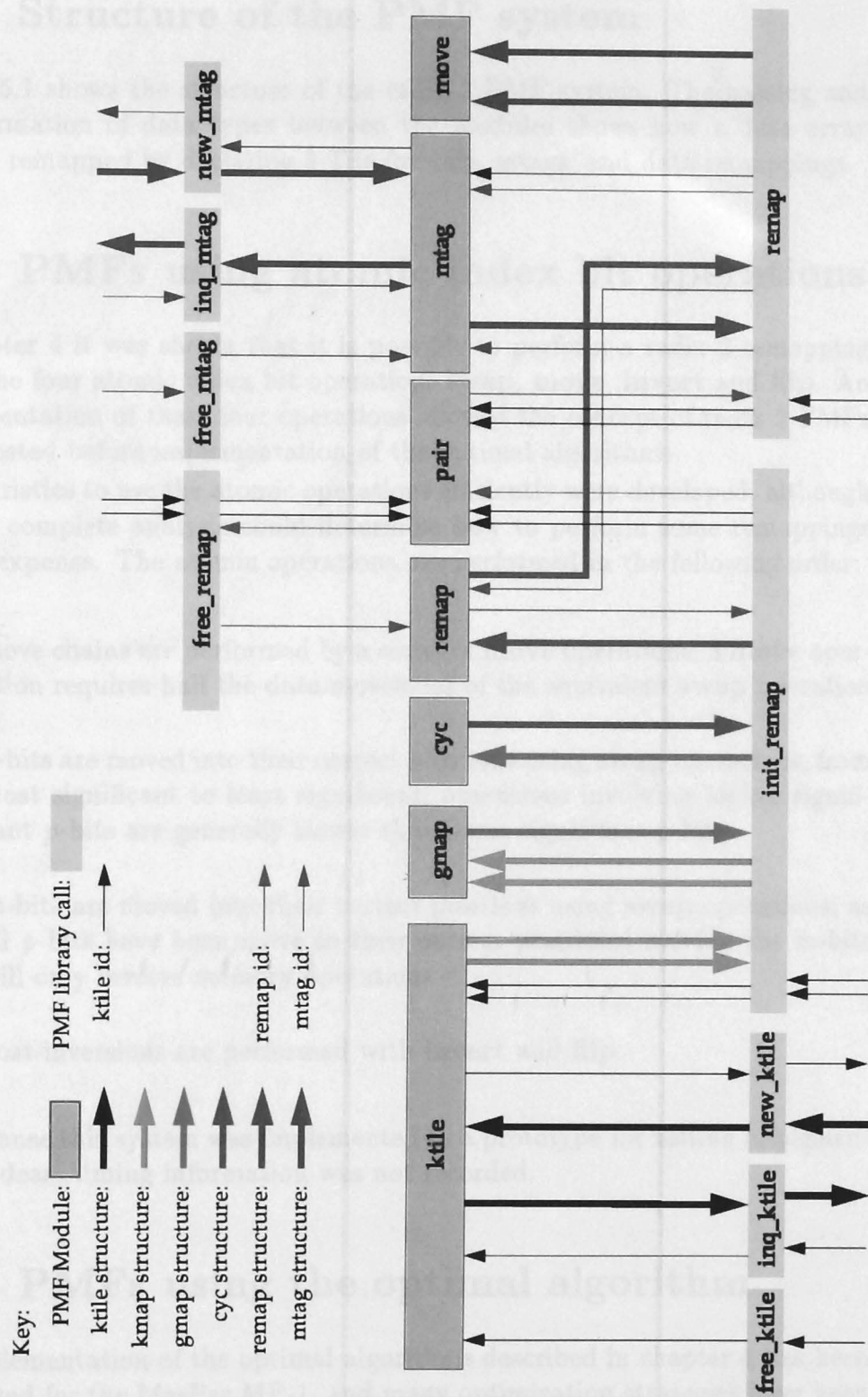


Figure 5.1: The partial structure of a radix 2 PMF implementation on the MasPar MP-1



Figure 2.1: The partial structure of a radix-2 FFT implementation on the MPP-1.

5.4 Structure of the PMF system

Figure 5.1 shows the structure of the radix 2 PMF system. The passing and transformation of data types between the modules shows how a data array may be remapped by declaring k -Tile formats, mtags, and data remappings.

5.5 PMFs using atomic index bit operations

In chapter 4 it was shown that it is possible to perform a radix 2 remapping using the four atomic index bit operations *swap*, *move*, *invert* and *flip*. An implementation of these four operations allowed the concept of radix 2 PMFs to be tested before implementation of the optimal algorithms.

Heuristics to use the atomic operations efficiently were developed, although a more complete analysis could determine how to perform some remappings at less expense. The atomic operations are performed in the following order:

- i. move chains are performed by a series of *move* operations; a *move* operation requires half the data movement of the equivalent *swap* operation
- ii. p -bits are moved into their correct positions using *swap* operations, from most significant to least significant; operations involving higher significant p -bits are generally slower than lower significant p -bits
- iii. m -bits are moved into their correct positions using *swap* operations; as all p -bits have been move to their correct positions, moving the m -bits will only involve memory operations
- iv. post-inversions are performed with *invert* and *flip*.

Because this system was implemented as a prototype for testing implementation ideas, timing information was not recorded.

5.6 PMFs using the optimal algorithm

An implementation of the optimal algorithms described in chapter 4 has been completed for the MasPar MP-1, and many optimization strategies have been used. Because of the recursive nature of the algorithms the implementation needed some care. Several architectural features peculiar to the MasPar can give greater improvements in performance.

5.6.1 Assembler coding

In many computer languages such as C, recursive algorithms are expensive because of the need to pass parameters and save machine registers on a data stack. On the MasPar, recursion can be especially expensive; the memory attached to the processors is relatively slow, and 40 32-bit registers are available to the programmer.

To avoid these difficulties, the data remapping functions were hand-coded in assembler with parameters passed in machine registers. This avoids the need to use a stack within the processing elements at all, and greatly limits the use of the ACU stack with its much faster memory.

Coding in assembler also allowed more flexibility in taking advantage of some special machine instructions, instruction scheduling to allow memory operations to occur in the background, and more efficient use of the router links.

At the time the PMF system was first implemented no optimizing compiler was available for the MasPar and hand-coded assembler was more efficient than compiler-generated code. An optimizing MPL compiler based on the GNU C compiler has now been introduced, making the gains available by the use of hand-coded assembler less marked; the reasons given here for hand-coding are still applicable, however [47, 48].

5.6.2 PE register usage

Every processing element in the MasPar contains 48 data registers of 32 bits each, of which 40 are available to the assembly language programmer. This is more than sufficient to store all the distributed information required to perform a remapping; except in the initialization stages, the slow parallel memories need only be accessed for the data to be remapped.

The ACU has fewer registers than the PEs but, because the overhead in accessing its own memory is so small (two clocks to access memory instead of one clock to access an internal register), keeping remapping information in its memory does not significantly affect performance.

5.6.3 Chunking to larger data objects

Most large sequential machines nowadays have a wide data path between the processor and its memory which is 32 or even 64 bits wide. This affects the performance of the processor when dealing with different data types; many operations take an identical time whether for a single byte or for a 32- or 64-bit integer.

For this reason, when moving large amounts of data on a sequential machine, it is most efficient to move the data in chunks of the largest available

data type.

In a massively parallel processor array, the data path between a processing element and its memory is typically not so large; the MasPar MP-1 has an eight-bit data path between processors and memory, and some machines have only a one-bit data path. Although data transfers are limited by hardware to a certain width, the MasPar instruction set includes instructions for data types of up to 64 bits.

Although PE/memory communication is limited to an eight-bit data path, there are several good reasons for chunking the data movement to larger data types:

- Using larger data types allows the use of fewer instructions, reducing instruction startup time and decreasing the number of times a loop needs to be executed.
- Treating the index bit permutation as an operation on 16, 32 or 64-bit integers instead of 8-bit integers allows 1, 2 or 3 memory index bits to be ignored
- When using the global router to send a single byte, 88% of the time is spent opening and closing the router connection and only 12% of the time is used for actual data communication. When transferring a 64 bit object, only 48% of the time is used for opening and closing the communication and 52% of the time can be used for sending data (see figure 5.2).

The last point is the most significant; for this reason, it is faster to perform extra memory permutations before and after the router remapping step to allow router communications to be performed using as large a data type as possible.

5.6.4 Processor cluster optimizations

The processors in the MasPar are not individually connected to a crossbar switch, but in clusters of sixteen processors per router connection. Clusters correspond to processing element chips, each of which contain two clusters. Similarly, each block of 32×32 processors correspond to a single board on the MasPar. These hardware details change the view of processor index bits from a contiguous sequence of bits to a more complex address with different bits selecting board, cluster and processor within the cluster. Viewed in this light, the 14-bit address of a processor in a 16384 element MasPar would look like this:

$$i_{proc} = \overset{b}{p}_{13} \overset{b}{p}_{12} \overset{c}{p}_{11} \overset{c}{p}_{10} \overset{c}{p}_9 \overset{i}{p}_8 \overset{i}{p}_7 \overset{b}{p}_6 \overset{b}{p}_5 \overset{c}{p}_4 \overset{c}{p}_3 \overset{c}{p}_2 \overset{i}{p}_1 \overset{i}{p}_0,$$

with "b" representing board-selection bits, "c" representing cluster-selection bits and "i" representing intra-cluster processor selection bits [5].

Opening a router connection does not connect individual processors, but individual clusters of sixteen processors. To give the appearance of a link to every processor, the communication must be iterated until every processor has obtained a link and sent its data. This iteration often has the effect of decreasing router performance by a factor of sixteen, but in many situations different numbers of iterations are required. If only a few processors in each cluster are attempting to communicate, fewer than sixteen iterations may be required. When performing complex processor permutations, the hardware may not be able to find an optimal connection order for processors within each cluster, resulting in more than sixteen iterations being required; section 6.4 describes a software method for obtaining an optimal connection order.

On a single board MasPar with 1024 processing elements, the connection network associated with the router may be treated as a crossbar switch, as it is capable of connecting clusters in any permutation. However, on a multi-board MasPar, some cluster permutations involving processor bit permutations between cluster and board bits cannot be connected in a single step. It is not known how to predict when this will occur, so that it is necessary to check which processors succeed in a router communication and repeat it if some fail [17].

Once a router connection is open it is possible for any processor in that cluster to send data through the opened connection to any processor in the connected cluster.

If the data remapping to be performed contains a p -bit permutation in which intra-cluster index bits are only permuted with intra-cluster index bits, sixteen data transfers may be performed using only one router open/close. Similarly, if the data remapping contains a p -bit permutation involving three intra-cluster index bits, eight data transfers may be performed using a single router open/close.

An example of such an operation is the transposition of a 128×128 array stored on a 128×128 processor array; the complete remapping may be performed with the opening of only one router connection per cluster:

Source k -Tile	Destination k -Tile	Radix 2 remapping
a : [128; 128]	a : [128; 128]	
k : [128; 128]	k : [128; 128]	$(\overset{i}{p}_0, \overset{i}{p}_7)(\overset{i}{p}_1, \overset{i}{p}_8)(\overset{c}{p}_2, \overset{c}{p}_9)(\overset{c}{p}_3, \overset{c}{p}_{10})$
m : [0; 1]	m : [1; 0]	$(\overset{c}{p}_4, \overset{c}{p}_{11})(\overset{b}{p}_5, \overset{b}{p}_{12})(\overset{b}{p}_6, \overset{b}{p}_{13})$
d : [1; 128; 128]	d : [1; 128; 128]	

Because it is necessary to synchronize the sending with the receiving processors, it is more complex to perform this cluster optimization if any intra-cluster bits take part in an (mp^*) cycle, so this is not performed.

5.6.5 Using the xnet for P/M transpositions

As well as a router network, the architecture of the MasPar includes a toroidal mesh network, the *xnet*. This network allows every processor in the array to communicate simultaneously in the same direction, so it is at least sixteen times faster than the router for nearest-neighbour communication (figure 5.2).

When performing a radix 2 remapping using the algorithms and optimizations previously described, remapping can be performed faster using the router than using the *xnet* because communication can be performed in one step per data element, whereas the *xnet* often requires several steps with time proportional to distance.

However, in some special cases the remapping to be performed is especially suitable for the *xnet*. One special case is detected by the PMF system to be performed using the *xnet*: the exchange of a sequential set of pairs of m -bits and p -bits.

The algorithm for performing this operation is similar to Kuszmaul's, and is described in section 6.2.1 [42].

As with the chunking optimization, it is possible to convert many remappings to use the *xnet* transposition algorithm by performing memory permutations before and after the *xnet* algorithm.

5.7 Testing

Because the complete radix 2 remapping system is moderately large and contains a quantity of hard-to-check assembler code, we took the approach of generating a large number of random problems for checking the system.

5.7.1 Generation of random remappings

In order to generate a representative sample of remappings, several considerations were taken into account. When generating random bit-fields of length n , the simplest strategy would be to take the binary representation of a binary number x , $0 \leq x < 2^n$. However, this would give an undesirable distribution, with all bits or no bits appearing with the quite small probability $1/2^n$. Similarly, bit-fields with $n/2$ bits would be more common than the other distributions.

Similar problems arise in providing fair distributions for testing the different algorithms, which are divided into mixtures of the groupings: (m^*) , (p^*) , (mp^*) , identity, in-place/copy and 8/16/32/64 bits. An approach which provides a good coverage of all these groupings was obtained by generating a pair of random kmaps (described in section 5.2) with the following properties:

- Keep 0, 1, 2 or 3 initial memory bits as identity bits to exercise routines for the four data types evenly
- Randomly generate the number of m -bits, p -bits, pre- and post-inverted m -bits, and pre- and post-inverted p -bits
- Randomly enable and invert the appropriate number of bits in the first kmap
- Because remaps including bit flips are performed with a copy remap, allow flips to occur only with approximately a 50% probability
- Set the second kmap to be a random permutation of the bits in the first kmap

Once each pair of kmaps is generated, a remap may be generated and tested. If an error occurs (and several did), the kmaps may be back-converted to k -Tile formats in canonical form and printed with the generated remap for manual inspection.

5.7.2 Checking performed remappings

Once a random remap has been generated, it must be tested on a sample data set to ensure that it is performing the required data movement. If the data type to be remapped is large enough (i.e., the lowest significant memory bits are identity cycles), it is possible to assign a different value to every data element; a simple value to test is the data array index of the element.

This strategy tests the algorithms but does not exercise all the functions, as the 8- and 16-bit remapping functions will never be called. When testing these it is not possible to ensure that every enabled data address contains a different data value, so we must choose data values that are unlikely to appear correct after a failed remapping. The strategy used is to set each data value to the exclusive-or of all the bytes in the source address; most errors are manifested as missing data elements or incorrect index bit permutations, and this strategy will detect such errors because only the correct index-bit permutation can give the correct ordering of data elements.

5.7.3 Results of testing

These random mappings were used extensively for uncovering several errors in the early implementations. This testing also uncovered the problem with cluster permutations on multi-board machines (see section 5.6.4), which occurs rarely with application-oriented remappings.

In the later stages of development, error-rates were reduced to one error in several hundred random remappings. The system has now been passed

through 15000 remappings of 0 to 15 m -bits and 0 to 14 p -bits with no errors detected and is considered sufficiently stable.

5.7.4 A non-assembler library

One unfortunate aspect of a relatively new architecture, such as the MasPar MP-1, is that details of both the assembly language and its interface to higher-level language code may change. If such a change were to occur, it might be very difficult to re-create a working version of the radix 2 PMF library.

To alleviate any problems if this were to occur, an MPL language version of the radix 2 PMF library was created, containing MPL-language versions of all the functions in the assembler implementation. If the assembler language version ceased working due to a change in the MasPar's instruction set or the assembler/MPL language interface, the MPL language version could be used in its place. When generating a new assembler version, the operation of the new version could be also be compared against the MPL version.

The MPL language version has been subjected to the same testing as the assembler version, and executes at approximately half the speed of the assembler version.

5.8 Results

The results given in this section compare the actual times required for performing several remapping operations on the MasPar MP-1 against a lower bound for the times based only on the speed of memory and communication operations on the MasPar.

The speed of this implementation is also compared against several remapping routines supplied by MasPar for performing a small set of remapping operations in MasPar's image processing library. Because these routines were individually written by MasPar, presumably with efficiency in mind, they should provide a comparison of the speed of the general PMF system with handwritten code. However, it should be noted that the MasPar routines are not limited to data sets with power-of-two dimensions.

More performance measurements for the use of PMFs in actual applications are given in chapter 7.

5.8.1 Execution time of radix 2 remapping

Several remapping problems were chosen to exercise the different algorithms used in the PMF implementation:

- memory bit cycles (m^*)

- processor bit cycles (p^*)
- combined memory/processor bit cycles (mp^*)
- mixed bit cycles.

Several variations of these problems were also chosen to show the effect of the architecture-dependent optimizations:

- chunking to larger data objects
- processor cluster optimizations
- xnet usage for transposition

Figure 5.2 shows the base timings for memory and communication operations on the MasPar MP-1 computer [47]. These timings can be used to determine a lower bound on the running time of any algorithms for radix 2 PMFs implemented using the MasPar's router.

Algorithms using the xnet may in some cases be faster; however, this would require a more detailed analysis of possible xnet algorithms, and this has not been undertaken. Because the use of the router requires a similar time to a single xnet transmission over a distance of 16 processors, it is to be expected that only highly regular remappings or those involving the lowest-significance processor grid bits could be performed with a substantial increase in efficiency. In any case, remappings involving transpositions between contiguous pairs of processor and memory bits are performed with a more efficient xnet algorithm.

The MasPar architecture is capable of performing register/memory operations in parallel with any other operations which do not conflict with the affected processor registers. This effect has not been taken into account in the lower bound; it would be possible to unroll some of the loops to ensure that multiple memory reads and writes were interlaced with communication operations to allow both to proceed in parallel.

It is not possible to perform (mp^*) remappings in a single pass without some recourse to indirect addressing, so its use is assumed for (mp^*) and mixed remappings.

The timings for operations on data types of different sizes can vary widely but, because many operations may be rearranged to use the largest possible data type, remappings involving inter-processor communication are assumed to be performed using the largest chunk possible of 8 bytes, except in cases where every chunk is sent to a different processor. Similarly, because of the possibility for cluster optimizations, it is assumed that a router connection need only be opened once per data chunk per cluster unless all communications from within each cluster occur to different clusters.

Instruction		Time (ticks)	Time (nS)
Open router connection,	ropen	42	3360
Send data through router,	rsend8	25	2000
	rsend16	33	2640
	rsend32	49	3920
	rsend64	81	6480
XNET transmission,	xnet8	$13 + d \times 10$	$1040 + d \times 800$
	xnet16	$15 + d \times 18$	$1200 + d \times 1440$
	xnet32	$19 + d \times 34$	$1520 + d \times 2720$
	xnet64	$27 + d \times 66$	$2160 + d \times 5280$
Direct load from memory,	ld8	20	1600
	ld16	38	3040
	ld32	74	5920
	ld64	146	11680
Indirect load from memory,	ld8*	55	4400
	ld16*	103	8240
	ld32*	199	15920
	ld64*	391	31280

Figure 5.2: Base time for MasPar MP-1 memory and communication instructions; 1 tick = 80nS. Memory store instructions take a similar time to memory load instructions. An ropen and an rsend may be performed simultaneously for a saving in time approximately equal to the cost of an rclose

These assumptions give a lower bound on runtime that in some cases may be lower than any possible implementation, but give an indication of where overheads could be reduced or a more efficient algorithm used.

Figure 5.3 shows the execution time for several in-place radix 2 remappings. Figure 5.4 shows a computed lower bound for the same problems. Some remappings come close to achieving 100% of the lower bound/actual time ratio, but several are substantially slower. The overheads are lowest for the largest data types, which is to be expected as the parallel execution of memory operations is able to hide non-memory and non-communication costs.

The (*mp**) remappings perform most poorly, but the simple (*mp**) remapping may be performed more efficiently using the xnet algorithm; comparing the actual xnet execution time with the router lower bound shows the xnet algorithm to be superior in this case.

Cumulatively, the lower bound on execution time is 64% of the time achieved without using the xnet, and 77% of the time achieved with. Thus, a 30%–50% average performance increase could be obtained using an improved implementation or superior algorithms. Improvements in implementation could be obtained in at least three ways: by unrolling the memory permutation loops to

Permutation	Time (μS)			
	8 bit	16 bit	32 bit	64 bit
Identity cycle (m_0)	166	-	-	-
Short (m^*) cycles (m_0, m_1)(m_2, m_3)(m_4, m_5)(m_6, m_7)(m_8, m_9)	6222	6536	12232	23797
Long (m^*) cycle ($\overline{m}_0, m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8, m_9$)	5251	6660	12629	24572
(p^*) cycles (clusters preserved) ($m_1 \dots (m_{10})(p_0, p_1)(p_2, p_3)(p_4, p_9)(p_5, p_6)(p_7, p_8)$)	59228	79916	122262	141475
(p^*) cycles (no clusters preserved) ($m_1 \dots (m_{10})(p_0, p_3, p_2, p_1, p_4, p_9, p_5, p_7, p_6, p_8)$)	82565	102551	145199	187339
Simple (mp^*) (m_0, p_0)(m_1, p_1)(m_2, p_2) \dots (m_9, p_9)	119756	133103	163282	232268
Simple (mp^*) using xnet (m_0, p_0)(m_1, p_1)(m_2, p_2) \dots (m_9, p_9)	35837	45470	71465	131340
Long mixed cycle ($m_8, m_6, m_4, m_0, p_6, p_4, m_7, p_5, p_7, p_3$, $m_1, p_0, p_9, m_3, p_8, m_9, p_1, p_2, m_2, m_5$)	38116	68529	127576	253146

Figure 5.3: Actual in-place execution time of radix 2 PMFs on a variety of problems. 16-, 32-, and 64-bit problems have identity lower-significant m -bits with permuted m -bits moved up to make room

allow memory reads and writes to be back-grounded, by calculating multiple cluster permutation synchronizations for (mp^*) permutations, and only opening a router connection once for a complete remap where there is only one cluster permutation. As the data is only passed through the global router once, algorithmic improvements using the router could only reduce the number of memory operations. Any actual gains obtainable may be even smaller considering the conservative nature of the lower bound estimate. There is still much scope for improved algorithms using the xnet, however.

5.8.2 Hand coding vs. PMFs

Another test of the relevance of a general system for performing remapping, as opposed to purpose-built routines, is to measure the performance of the general system against routines written for the same task. MasPar have included a set of data remapping routines in their image processing library, mpipl [51], which provides a large set of routines against which to test; section 2.4.7 outlines the advantages and shortcomings of MasPar's approach.

Figures 5.5-5.8 compare the execution times of version 2.1 of the mpipl remapping routines against the equivalent PMF operations for four image shapes. Version 3.1 of the MasPar operating system has been installed since these timings were taken, but very similar results were obtained with the newer version. The PMF operations have the advantage of allowing a remapping to

Permutation	Size	Lower bound	$\frac{\text{low}}{\text{actual}}\%$
Identity cycle	any	-	0%
Short (m^*) cycles	8bit	$992(T_{ld8} + T_{st8})$	50%
	16bit	$992(T_{ld16} + T_{st16})$	91%
	32bit	$992(T_{ld32} + T_{st32})$	95%
	64bit	$992(T_{ld64} + T_{st64})$	97%
Long (m^*) cycle	8bit	$1024(T_{ld8} + T_{st8})$	61%
	16bit	$1024(T_{ld16} + T_{st16})$	92%
	32bit	$1024(T_{ld32} + T_{st32})$	95%
	64bit	$1024(T_{ld64} + T_{st64})$	97%
(p*) cycles (clusters preserved)	8bit	$1024(T_{ld8} + T_{st8}) + T_{ropen} + 128.16T_{rsend64}$	28%
	16bit	$1024(T_{ld16} + T_{st16}) + T_{ropen} + 256.16T_{rsend64}$	41%
	32bit	$1024(T_{ld32} + T_{st32}) + T_{ropen} + 512.16T_{rsend64}$	53%
	64bit	$1024(T_{ld64} + T_{st64}) + T_{ropen} + 1024.16T_{rsend64}$	92%
(p*) cycles (no clusters preserved)	8bit	$1024(T_{ld8} + T_{st8}) + 16.128(T_{ropen} + T_{rsend64})$	28%
	16bit	$1024(T_{ld16} + T_{st16}) + 16.256(T_{ropen} + T_{rsend64})$	45%
	32bit	$1024(T_{ld32} + T_{st32}) + 16.512(T_{ropen} + T_{rsend64})$	64%
	64bit	$1024(T_{ld64} + T_{st64}) + 16.1024(T_{ropen} + T_{rsend64})$	99%
Simple (mp^*)	8bit	$1024(T_{ld8*} + T_{st8*}) + 128T_{ropen} + 16.1024T_{rsend8}$	35%
	16bit	$1024(T_{ld16*} + T_{st16*}) + 256T_{ropen} + 16.1024T_{rsend16}$	46%
	32bit	$1024(T_{ld32*} + T_{st32*}) + 512T_{ropen} + 16.1024T_{rsend32}$	60%
	64bit	$1024(T_{ld64*} + T_{st64*}) + 1024T_{ropen} + 16.1024T_{rsend64}$	75%
Simple (mp^*) using xnet	8bit	-	(117%)
	16bit	-	(134%)
	32bit	-	(138%)
	64bit	-	(132%)
Long mixed cycle	8bit	$1024(T_{ld8*} + T_{st8*}) + 128(T_{ropen} + 16T_{rsend64})$	59%
	16bit	$1024(T_{ld16*} + T_{st16*}) + 256(T_{ropen} + 16T_{rsend64})$	64%
	32bit	$1024(T_{ld32*} + T_{st32*}) + 512(T_{ropen} + 16T_{rsend64})$	68%
	64bit	$1024(T_{ld64*} + T_{st64*}) + 1024(T_{ropen} + 16T_{rsend64})$	69%

Figure 5.4: A lower bound on the execution times of problems in figure 5.3, compared with actual execution times. The xnet percentages are actually compared with the router lower bound

be pre-initialized, but this is still a reasonable comparison; if a remapping is to be performed only once, small differences in execution time are inconsequential, and if a remapping is to be performed many times a small initialization time is insignificant. The slowest remapping initialization took approximately .12 seconds, and the average time was .06 seconds; these times are similar to the time required for a single remapping.

The mpipl routines do handle non power-of-two image dimensions, giving them an increased generality over the PMF routines, but this need not cause them to execute more slowly. There is evidence that the MasPar routines are optimized for powers of two in any case; many remapping routines require a large working storage for non power-of-two remappings which is not required for power-of-two remappings.

The mpipl routines do not perform in-place remapping, which halves the size of image that may be processed within memory; in these tests, PMFs are also used to perform remapping by copying, but could be used for in-place remapping with little difference in execution time.

The cumulative difference in timings,

$$\frac{\sum \text{PMF timings}}{\sum \text{mpipl timings}}$$

is 36%, showing that the general PMF routines are almost three times faster than the image processing library routines. Almost all remappings are performed faster by PMFs; examination of a router counter internal to the MasPar indicates that the exceptions are remappings in which mpipl uses the xnet.

Fier has hand-coded a single index bit permutation problem partially in assembler, a perfect shuffle of 128 32-bit data values in a 16384 processor MasPar [18]:

$$(m_0)(m_1)(m_2, m_3, m_4, m_5, m_6, m_7, m_8, p_0, p_1, \dots p_{12}, p_{13})$$

He reports a timing of 0.037441 seconds to perform this mapping as a copy. The same problem was tried using radix 2 PMFs as both an in-place and a copy remapping with timings of 0.026299 and 0.026883 seconds respectively. The timing difference is not simply due to a difference in coding and algorithms, however; PMFs automatically transformed the problem to require half the number of router connections to be made. The original mixed remapping was converted into an (m^*) remapping followed by an $(m^*) + (mp^*)$ remapping, of which the (mp^*) remapping could use 64-bit router transfers:

$$\begin{aligned} & (m_0)(m_1)(m_2)(m_3, m_4, m_5, m_6, m_7)(m_8, p_0, p_1, \dots p_{13}) \\ & (m_0)(m_1)(m_2, m_3, m_8)(m_4)(m_5)(m_6)(m_7)(p_0)(p_1) \dots (p_{13}) \end{aligned}$$

5.9 Summary

This chapter shows the relationship between the k -Tile format of chapter 3 and the index bit map and algorithms of chapter 4: a restricted form of the k -Tile format, the 2^k -Tile format, is shown to be equivalent to the index bit map. This allows the use of the optimal radix 2 algorithms for implementation of a radix 2 PMF system.

The implementation of a radix 2 PMF system requires many data types, including the 2^k -Tile format, the index bit map, a memory tag and a remap structure for representing the algorithm generated by the methods in chapter 4. Several intermediate data structures are also required. A number of functions in an application/program interface is also required to allow the functionality of PMFs to be accessible to an application programmer; functionality is also included to allow the 2^k -Tile formats for many common data mappings and transformations to be generated automatically.

Two versions of the radix 2 PMF system were implemented. The first was based on the atomic index bit permutation operations. This version was really only for proof-of-concept, and although it was correct, it did not take full advantage of the MasPar's architectural features and was relatively slow.

The second version was based on the optimal algorithms introduced in chapter 4. To achieve the greatest efficiency, many aspects of the MasPar architecture were used:

- all the data movement code was hand-written in MasPar assembler
- the large register file within the PEs was used to avoid all unnecessary memory accesses
- data movements were chunked into 8-bit, 16-bit, 32-bit or 64 bit transfers to optimized memory and, far more importantly, communications speeds
- openings of the global router network were reduced by taking note of the clusterings of sixteen processors to each router link
- the mesh network was used for a common class of processor/memory transpositions. A non-assembler version of the data movement code was also written to safeguard against any changes in the MasPar assembler language or the C/assembler interface.

Because of the quantity of assembler code in the system, care was taken in the generation of test cases for the system. A system for generating and testing a set of random remappings was used which give all the components in the system approximately the same workout. Once the 'last' bug was found, 15000 separate remappings of data arrays up to the maximum size of the system were tested without error.

Several sample remapping problems that exercised different components of the radix 2 PMF system were timed and compared with a lower-bound on router-based remapping algorithms. The cumulative time of the lower bound on these problems was 77% of the time used by the PMF system, indicating that little can be done to improve the performance of router-based approaches. No computation of a lower-bound on xnet algorithms was attempted, but reasons are given for the expectation that only a small class of remappings would benefit from further xnet usage.

Several comparisons were made between the execution time of the radix 2 PMF system with hand-coded remapping operations. In a comparison of the PMF system with a set of remappings in the MasPar image-processing library (mpipl), PMFs executed the remappings in a cumulative 36% of the time required for the mpipl routines. In a comparison of PMF times against a single perfect-shuffle problem hand-coded by Fier in assembler, PMFs were 30% faster, even when an in-place instead of a copy remapping were performed.

The major shortcoming of radix 2 PMFs is the restriction of image dimensions to powers of two; to offer more flexibility, the ability to perform arbitrary k -Tile remappings is desirable. Chapter 6 explores *mixed radix remapping*, which is a step towards this goal.

Image shape: (512 × 512)								
Src	Dst		Time (μS)	$\frac{pmf}{ipl}\%$	Time (μS)	$\frac{pmf}{ipl}\%$	Time (μS)	$\frac{pmf}{ipl}\%$
			8bit		16bit		32bit	
1dcs	2dh	pmf ipl	9773	20%	17355	28%	33333	47%
			49728		62229		71385	
1dh	2dh	pmf ipl	9258	31%	17251	137%	33228	140%
			30281		12583		23804	
2dcs	2dh	pmf ipl	38540	47%	41922	50%	50885	59%
			82035		83317		85634	
2dh	1dcs	pmf ipl	9985	16%	17364	28%	33282	47%
			61527		61970		71264	
2dh	1dh	pmf ipl	9258	37%	17249	93%	33229	112%
			24703		18529		29761	
2dh	2dcs	pmf ipl	32641	40%	36099	44%	45100	53%
			82366		82392		85100	
2dh	e/w	pmf ipl	7691	25%	12568	30%	22452	35%
			30389		41354		63424	
2dh	n/s	pmf ipl	5410	15%	9827	23%	18667	29%
			37068		42114		63605	
2dh	xpos	pmf ipl	7421	34%	12053	137%	22313	127%
			21669		8819		17577	

Figure 5.5: A comparison of the execution times of PMFs versus the MasPar image processing library, mpipl, for a 512×512 image

Image shape: (1024 × 1024)								
Src	Dst		Time (μS)	$\frac{pmf}{ipl}$ %	Time (μS)	$\frac{pmf}{ipl}$ %	Time (μS)	$\frac{pmf}{ipl}$ %
			8bit		16bit		32bit	
1dcs	2dh	pmf ipl	36258	26%	65576	40%	129312	69%
			142158		163251		187398	
1dh	2dh	pmf ipl	22759	92%	26232	56%	51408	56%
			24627		47258		92139	
2dcs	2dh	pmf ipl	44199	19%	46194	18%	79679	27%
			235689		259824		294528	
2dh	1dcs	pmf ipl	36234	25%	65638	40%	129346	69%
			142132		163249		187402	
2dh	1dh	pmf ipl	13635	56%	26224	56%	51410	56%
			24555		47030		91727	
2dh	2dcs	pmf ipl	36552	16%	46189	18%	72205	25%
			235604		258231		292507	
2dh	e/w	pmf ipl	26760	22%	46230	28%	85775	34%
			121120		164830		253516	
2dh	n/s	pmf ipl	18661	15%	36328	22%	71662	28%
			123395		167900		253891	
2dh	xpos	pmf ipl	25385	144%	43968	125%	84996	121%
			17614		35127		70194	

Figure 5.6: A comparison of the execution times of PMFs versus the MasPar image processing library, mpipl, for a 1024×1024 image

Image shape: (2048 × 2048)								
Src	Dst		Time (μS) $\frac{pmf}{ipl}\%$		Time (μS) $\frac{pmf}{ipl}\%$		Time (μS) $\frac{pmf}{ipl}\%$	
			8bit		16bit		32bit	
1dcs	2dh	pmf ipl	179449	24%	309952	37%	587974	60%
			739126		833273		978107	
1dh	2dh	pmf ipl	62919	67%	124548	68%	247776	68%
			94011		183784		363318	
2dcs	2dh	pmf ipl	119816	13%	204752	20%	398300	34%
			949087		1041304		1177369	
2dh	1dcs	pmf ipl	180264	24%	313280	38%	590532	61%
			739123		833278		970864	
2dh	1dh	pmf ipl	62937	67%	124535	68%	247747	68%
			93440		183029		361810	
2dh	2dcs	pmf ipl	123616	13%	210469	20%	410004	35%
			942775		1038580		1177572	
2dh	e/w	pmf ipl	102752	21%	180604	27%	338798	33%
			483491		658599		1013891	
2dh	n/s	pmf ipl	71632	15%	142274	21%	283544	28%
			493119		670670		1014647	
2dh	xpos	pmf ipl	96902	138%	171356	122%	335409	120%
			70235		140359		280660	

Figure 5.7: A comparison of the execution times of PMFs versus the MasPar image processing library, mpipl, for a 2048 × 2048 image

Image shape: (512 × 2048)								
Src	Dst		Time (μS)	$\frac{pmf}{ipl} \%$	Time (μS)	$\frac{pmf}{ipl} \%$	Time (μS)	$\frac{pmf}{ipl} \%$
			8bit		16bit		32bit	
1dcs	2dh	pmf ipl	35454 197952	18%	65674 223600	29%	129652 259817	50%
1dh	2dh	pmf ipl	16714 29133	57%	32112 49210	65%	62912 94026	67%
2dcs	2dh	pmf ipl	80427 236228	34%	96158 260187	37%	136496 294614	46%
2dh	1dcs	pmf ipl	36234 197820	18%	65638 223599	29%	129341 259821	50%
2dh	1dh	pmf ipl	16727 28757	58%	32132 48981	66%	62930 93641	67%
2dh	2dcs	pmf ipl	80490 235775	34%	96254 259879	37%	136583 294599	46%
2dh	e/w	pmf ipl	27047 121403	22%	46513 165248	28%	86055 253520	34%
2dh	n/s	pmf ipl	18651 123320	15%	36312 167711	22%	71633 253706	28%
2dh	xpos	pmf ipl	25147 17611	143%	43714 35128	124%	85160 70186	121%

Figure 5.8: A comparison of the execution times of PMFs versus the MasPar image processing library, mpipl, for a 512×2048 image

Chapter 6

Mixed radix remapping

Although it is possible to perform multidimensional data processing operations on data sets whose dimensions are powers of two, being required to do so causes a great loss of flexibility. Real-world data sets rarely have such dimensions, forcing the programmer to pad data sets at the expense of both memory and processing time or, if memory is insufficient, at the expense of resolution.

As dimensionality increases, the wastage of memory increases markedly; scaling a one dimensional data set whose linear dimensions are 25% smaller than a power of two up to higher dimensions wastes 44% of memory in a two dimensional data set, 58% in three dimensions and 68% in four.

We have demonstrated in earlier chapters a class of data remapping algorithms for powers-of-two dimensions that are fast and flexible. In this chapter will show that it is possible to perform similar operations for mixed-radix dimensions, albeit with some performance losses.

6.1 The index digit map

An *index digit map* between a data array and a parallel device may be defined in a similar way to an index bit map. Instead of describing a mapping as a permutation of index bits, we use a permutation of mixed-radix digits. Because of the added complication of mixed radices, the notation used for index bit maps will be extended.

6.1.1 Mixed radix numbers

A mixed radix number is a representation of a number a using a vector containing k digits (a_0, \dots, a_{k-1}) , each with an associated base, (b_0, \dots, b_{k-1}) , where

$$0 \leq a_i < b_i, \text{ for } i, 0 \leq i < k$$

and

$$a = \sum_{i=0}^{k-1} a_i \prod_{j=0}^{i-1} b_j.$$

Indices represented as binary numbers are easily specified as a concatenation of bits, for example

$$a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0.$$

When specifying indices as a mixed-radix number, it is necessary to indicate the base associated with every digit. Normally with fixed radix numbers, the usual place for this specification is as a subscript at the end of the digits, for example

$$153_8 = 107_{10}.$$

However, as with bits it is useful to number digits in their order of *significance* in the indices of the initial array, which conventionally also uses the subscripts.

To separate the two numbering schemes for base and significance, we have chosen to indicate the base of each digit in a mixed-radix number by specifying it in base 10 above the digit, and the significance of each digit by a subscript. Thus,

$$\begin{matrix} 2 & 7 & 3 \\ 1 & 4 & 2 \end{matrix} = 2 + 3 \times (4 + 7 \times (1 + 2 \times 0)) = \begin{matrix} 10 & 10 \\ 3 & 5 \end{matrix} = 35$$

In a fixed radix number, every digit has a *place*, which indicates the weight that would be assigned to a 1 in that digit; for example, in the number

$$\begin{matrix} 8 & 8 & 8 \\ 7 & 5 & 2 \end{matrix}$$

2 is in the one's place, 5 is in the eight's place and 7 is in the sixty-four's place. For any number represented with a fixed radix b , the place associated with each digit of significance i is fixed with value b^i . Similarly, we can define a place for every digit in a mixed radix number. The place of digit i in the mixed radix number

$$\begin{matrix} b_{k-1} & & b_0 \\ a_{k-1} & \dots & a_0 \end{matrix}$$

is

$$\prod_{j=0}^{i-1} b_j.$$

Thus, the place of any digit in a mixed radix number depends on the bases of all digits of lower significance.

6.1.2 Indexing with a mixed radix number

The purpose of representing an array index as a mixed radix number is to provide a flexible way of splitting the data in the array. As an example, if we have a one dimensional data array A of shape (b_0, b_1) mapped in column-major

ordering, we can represent a one dimensional index into that array with the mixed radix number

$$\begin{matrix} b_1 & b_0 \\ a_1 & a_0, \end{matrix}$$

where the values of the digits a_0 and a_1 represent the row and column number respectively in the original two dimensional array.

The mixed-radix representation is very useful for mapping a one-dimensional data array A onto a multidimensional parallel device, represented as a one dimensional device array D .

6.1.3 Specifying an index digit map

The index digit map is specified analogously to the index bit map; the digits of the data array index, a , are rearranged to provide a representation of the device index, d , which directly specifies the data index associated with every enabled device address.

Because the base and significance of every digit in the factorization of a is present in the index digit map, it is not necessary to explicitly state the factorization of a being used. Sometimes the map from the device address, d , to the data array index, a , may be many-to-one to allow the data array to be replicated across the device.

The digits used in the factorization of a data index a into a mixed radix number will also be used as digits in the mixed radix representation of the device address d . It will therefore sometimes be necessary to distinguish between the *data index place* and the *device index place* of a digit, and also the *data index significance* and *device index significance* of a digit.

However, a_i is always used to represent a data index digit of significance i in the current factorization.

Digit permutation

In its simplest form, an index digit map is simply a permutation of the digits in a data index. The only restriction placed on such a map is that no digit may be placed across the boundary between two device dimensions to limit the complexity of the mapping. An example shows a simple index digit map from a data array to a one-dimensional device:

$$\begin{matrix} 3 & 5 \\ a_0 & a_1 \end{matrix} \Rightarrow \begin{array}{|c|c|c|c|} \hline 0 & 3 & 6 & 9 & 12 \\ \hline 1 & 4 & 7 & 10 & 13 \\ \hline 2 & 5 & 8 & 11 & 14 \\ \hline \end{array}$$

The box on the right hand side represents a one-dimensional device, with nested boxes around data array indices showing the arrangement of the device index digits.

Constant or Disabled digits

By specifying a constant digit in an index digit map, disabled digits may be included in the device map analogously to disabled bits in an index bit map. For example,

$${}^2_{a_0} {}^{33}_{a_1} \Rightarrow \begin{array}{|c|c|c|c|c|c|} \hline \boxed{X \ X \ X} & \boxed{0 \ 2 \ 4} & \boxed{X \ X \ X} & \boxed{X \ X \ X} & \boxed{1 \ 3 \ 5} & \boxed{X \ X \ X} \\ \hline \end{array}$$

Replicating digits

By specifying a 'star' digit (*-digit) in an index digit map, the data array may be replicated across the device. Given a valid device address, any value for the *-digit will contain the same data item. For example,

$${}^2_{a_0} {}^{33}_{a_1} \Rightarrow \begin{array}{|c|c|c|c|c|c|} \hline \boxed{0 \ 2 \ 4} & \boxed{0 \ 2 \ 4} & \boxed{0 \ 2 \ 4} & \boxed{1 \ 3 \ 5} & \boxed{1 \ 3 \ 5} & \boxed{1 \ 3 \ 5} \\ \hline \end{array}$$

Compound digits

Because we are dealing with mixed radices, it is possible that the number of elements stored along either the processor or memory axes will not exactly divide into their actual length. Thus it is not always possible to define empty space by inserting disabled digits. One solution is to treat the device's memory or processor array as if it were smaller, which means that the actual lengths of the device dimensions must be specified where they are not clear from the context. The lengths of the device dimensions may be stated explicitly by bracketing multiple digits together into a larger digit. For example,

$$\begin{array}{cc} \overbrace{31 \ 33}^{1024} & \overbrace{18 \ 14}^{256} \\ a_3 \ a_0 & a_1 \ a_2 \end{array}$$

In this example, a_1 and a_2 take up $18 \times 14 = 252$ bytes in the memory array, which has an actual size of 256 bytes. a_3 and a_0 take up 31×33 processors in the processor array, which actually contains 1024 processors. A simpler example mapping shows how a 3×3 data array may be mapped onto a 4×4 device:

$$\begin{array}{cc} \overbrace{3}^4 & \overbrace{3}^4 \\ a_1 & a_0 \end{array} \Rightarrow \begin{array}{|c|c|c|c|} \hline & P0 & P1 & P2 & P3 \\ \hline M0 & 0 & 1 & 2 & X \\ M1 & 3 & 4 & 5 & X \\ M2 & 6 & 7 & 8 & X \\ M3 & X & X & X & X \\ \hline \end{array}$$

In a similar way to Flander's notation, where the size of a device dimension is clear from the context or does not matter a bar may be placed in the index

digit map showing breaks between device dimensions. For example, in the mapping

$$\overset{7}{a_1} \mid \overset{3}{a_2} \overset{5}{a_0}$$

the device is treated as containing $5 \times 3 = 15$ memory elements and 7 processing elements.

In order to align digits between P and M , in some situations it is useful to be able to group several digits into a larger *compound* digit within other digits in an index digit mapping. In the mapping

$$\overset{5}{a_5} \overset{\overset{16}{2 \ 8}}{a_3 a_0} \mid \overset{3}{a_1} \overset{\overset{16}{3 \ 5 \ 4}}{a_6 a_2 a_4}$$

the compound digit containing a_3 and a_0 and the compound digit containing a_6 and a_2 are of the same size, making it a simpler task to exchange them.

Digit sense

The idea of bit sense in index bit maps may be generalized for index digit maps in several ways. The sense in an index bit map may be interpreted as either a reversal of storage order for that bit or a rotation by one position; which of these interpretations is placed on bit sense does not make any difference. For index digits, however, the two interpretations are not the same.

The idea of sense as a reversing operation has been retained for index digit maps for two reasons. Firstly, it is more consistent with the usual reversal interpretation of bit sense, and secondly, the functionality implied by the rotation interpretation can still be obtained with an implementation of k -Tile offsetting.

As with index bit maps, an inverted digit is indicated by an overbar, for example:

$$\overset{2}{a_0} \overset{3}{\bar{a}_1} \overset{3}{a_2} \Rightarrow \boxed{\boxed{4 \ 10 \ 16} \boxed{2 \ 8 \ 14} \boxed{0 \ 6 \ 12} \boxed{5 \ 11 \ 17} \boxed{3 \ 9 \ 15} \boxed{1 \ 7 \ 13}}$$

Compact mappings

Because there may be many factorizations of $|A|$, there may be many ways of specifying the same index digit map. For example, the index digit map

$$\overset{b_7}{a_7} \overset{b_6}{a_6} \overset{10}{6} \overset{10}{5} \overset{b_2}{a_2} \overset{b_1}{a_1} \overset{b_0}{a_0} \overset{b_5}{a_5} \overset{b_4}{a_4}$$

can be simplified to

$$\overset{b'_2}{a'_2} \overset{100}{65} \overset{b'_0}{a'_0} \overset{b'_1}{a'_1}$$

where

$$\begin{aligned} b'_0 &= b_0.b_1.b_2 & a'_0 &= a_0 + b_0.(a_1 + b_1.a_2) \\ b'_1 &= b_3.b_4.b_5 & a'_1 &= a_3 + b_3.(a_4 + b_4.a_5) \\ b'_2 &= b_6.b_7 & a'_2 &= a_6 + b_6.a_7 \end{aligned}$$

A representation of a mapping is defined to be *compact* if it contains the smallest number of digits.

Aligned index digit maps

Two index digit maps are said to be *aligned* if the factorization of the data array index, a , is the same in both mappings. For example, the two mappings

$$\begin{matrix} b_2 & b_1 & b_0 \\ a_2 & a_1 & a_0 \end{matrix} \text{ and } \begin{matrix} b_0 & b_2 & b_1 \\ a_0 & a_2 & a_1 \end{matrix}$$

are aligned because they both use the factorization

$$a = \begin{matrix} b_2 & b_1 & b_0 \\ a_2 & a_1 & a_0 \end{matrix}$$

Even if the factorization of the data index in two mappings differs, it may be possible to align the two mappings by re-factorizing the data index. Figure 6.1 shows how this may be performed for two example mappings.

However, it is not always possible to align two mappings; the two mappings shown in figure 6.2 cannot be aligned because the bases of digit 0 in the two mappings are relatively prime.

6.1.4 Specifying a mixed radix remapping

Once we have a mixed-radix data array mapped onto a device in one index digit map, we may wish to permute the data on the device to correspond to another. As an index digit map is a generalization of index bit map, and two index bit maps may be inter-converted by index bit permutation, it would be convenient if it were possible to convert one index digit mapping into another by index digit permutation.

Unfortunately, there are several complicating factors of mixed radix operations over radix 2 operations. Some are technology driven and may disappear in future computer architectures, but others are more troublesome and far-reaching in their effects.

Processor array size

As the binary number system pervades all common computer architectures, it makes sense to build parallel processor arrays in meshes whose dimensions are powers of two, which allows data sets with power-of-two dimensions to both fit snugly within the array and allows communications along the processor mesh

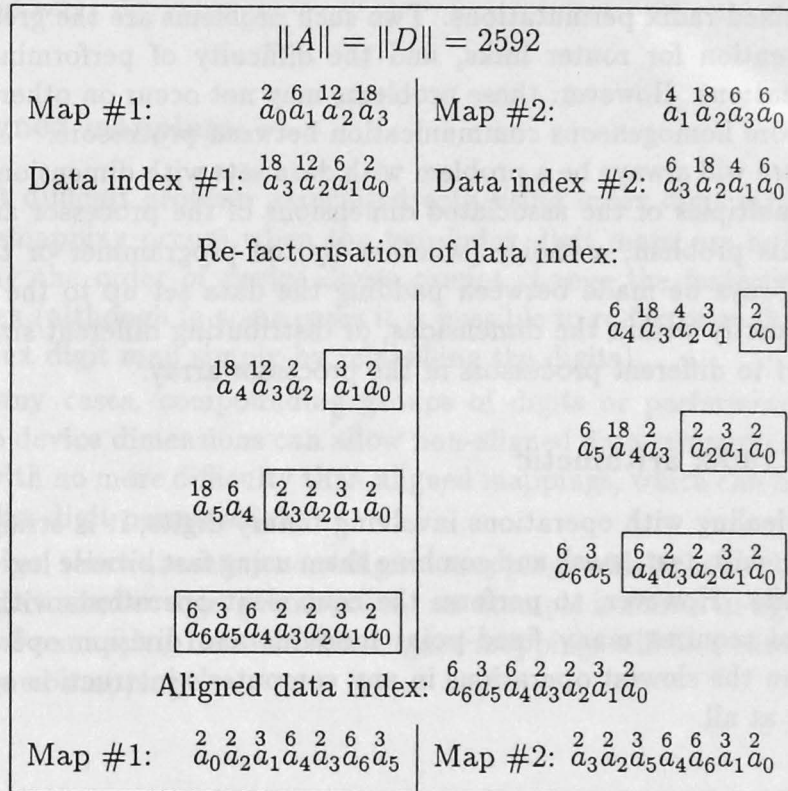


Figure 6.1: Aligning two index digit maps by factorizing data index

		M0	M1
$\begin{smallmatrix} 3 \\ a_1 \end{smallmatrix} \mid \begin{smallmatrix} 2 \\ a_0 \end{smallmatrix} \Rightarrow$	P0	0	1
	P1	2	3
	P2	4	5

		M0	M1
$\begin{smallmatrix} 3 \\ a_0 \end{smallmatrix} \mid \begin{smallmatrix} 2 \\ a_1 \end{smallmatrix} \Rightarrow$	P0	0	3
	P1	1	4
	P2	2	5

Figure 6.2: Two index digit maps that cannot be aligned

axes to correspond with the axes of the data set. Even where there is the appearance of more flexibility, such as the crossbar switch on the Maspar MP-1, there are often hidden costs associated with the communications resulting from mixed-radix permutations. Two such problems are the greater incidence of contention for router links, and the difficulty of performing any cluster optimizations. However, these problems may not occur on other architectures with more homogeneous communication between processors.

There will always be a problem with data sets with dimensions that are not exact multiples of the associated dimensions of the processor array. To cope with this problem, a choice (whether by the programmer or the computer) must always be made between padding the data set up to the first multiple to fit exactly within the dimensions, or distributing different sized pieces of a data set to different processors in the processor array.

Mixed radix arithmetic

When dealing with operations involving binary digits, it is straightforward to extract, shift, test, mask and combine them using fast bitwise logical operations and shifts. However, to perform the equivalent operations with mixed radix numbers requires many fixed-point modulus and division operations, which are often the slowest operations in any computer's instruction set, if they are present at all.

No independence of digits

When exchanging index bits in an index bit permutation, because the device index place of every bit is fixed, every bit whose position does not change during the permutation creates two identical independent sets of cycles in the corresponding data permutation. However, in index digit permutation, even if the positions of most digits do not change during the permutation, the device index places of all digits can change, creating complicated data permutations.

As an example, assume we have the two mappings:

$$\begin{matrix} 3 & 2 & 2 \\ a_2 & a_1 & a_0 \end{matrix} \quad \text{and} \quad \begin{matrix} 2 & 2 & 3 \\ a_0 & a_1 & a_2 \end{matrix}$$

When exchanging digits 0 and 2, the device index place of digit a_1^2 changes from 2 to 3. Although the value of this digit remains constant through the permutation, the address offset associated with that digit will change. For example, the source address

$$\begin{matrix} 322 \\ 010 \end{matrix} = 2$$

will be mapped to

$$\begin{matrix} 223 \\ 010 \end{matrix} = 3$$

Thus, representing the index digit permutation by cycle notation unfortunately does not clarify the corresponding data permutation operations, because the data permutation cycles specified by independent digit cycles are not separable in a useful way.

Non-aligned mappings

The most difficult problem associated with using index digit maps to specify a data remapping occurs when the two index digit maps are not aligned, as permuting the order of device digits cannot change the factorization of the data index (although in some cases it is possible to re-factorize the data index in an index digit map simply by relabelling the digits).

In many cases, compounding groups of digits or performing operations along the device dimensions can allow non-aligned data remappings to be performed with no more difficulty than aligned mappings, which can be performed as an index digit permutation.

However, there are many non-aligned mappings which cannot be performed as easily as an index digit permutation; an example is shown in figure 6.2. The problem of remapping between non-aligned mappings will be treated separately from aligned mappings.

6.2 Aligned index digit remapping

Many useful data remappings can be described by aligned index digit maps, so we first examine techniques to perform them.

Because the same factorization of the data index is used for the pair of maps we are converting between, remapping between aligned maps can be performed by index digit permutation.

As with the radix 2 algorithm, any algorithm for performing regular data remapping should aim to keep the number of memory accesses and uses of the communications network to a minimum. However, we must also take into account the problems we have mentioned previously, and try to minimize the use of multiplication, division and modulus operations.

6.2.1 Algorithm components

We first show how to perform an arbitrary index digit permutation with two different remapping operations, an arbitrary memory/processor permutation and a single digit exchange between processor and memory.

Memory permutation

The algorithm used in the radix 2 case for (m^*) permutations can be used unchanged for performing arbitrary memory permutations, unless the number of memory elements to be permuted is larger than the number of processors in the processor array.

If this is the case, either the index digit permutation or the memory data permutation may be split into several parts to be performed sequentially. In both cases, function 4.2, `permute_m`, may be used unchanged to perform the memory index digit permutation.

An example of a memory index digit permutation would be to convert the mapping

$$\begin{array}{c} 2 \quad 5 \\ a_3 a_2 \end{array} \mid \begin{array}{c} 3 \quad 2 \\ a_1 a_0 \end{array}$$

to

$$\begin{array}{c} 2 \quad 5 \\ a_3 a_2 \end{array} \mid \begin{array}{c} 2 \quad 3 \\ a_0 a_1 \end{array}$$

Processor permutation

The algorithm used in the radix 2 case for (p^*) permutations (function 4.1, `permute_p`) can be used unchanged for performing arbitrary processor permutations.

An example of a processor index digit permutation would be to convert the mapping

$$\begin{array}{c} 2 \quad 5 \\ a_3 a_2 \end{array} \mid \begin{array}{c} 3 \quad 2 \\ a_1 a_0 \end{array}$$

to

$$\begin{array}{c} 5 \quad 2 \\ a_2 a_3 \end{array} \mid \begin{array}{c} 3 \quad 2 \\ a_1 a_0 \end{array}$$

However, on a MasPar MP-1, each router link is shared between sixteen processors in a cluster. This will cause router link contention when performing mixed radix operations of this type if the router hardware is unable to distribute router links to processors to form a cluster permutation. Methods for pre-computing a contention-free allocation order of processors within clusters to router links is described in section 6.4.

Simultaneous memory and processor permutation

The algorithm described in section 4.4.6, `permute_mp`, can be used unchanged for performing arbitrary combined processor and memory permutations.

An example of a processor index digit permutation would be to convert the mapping

$$\begin{array}{c} 2 \quad 5 \\ a_3 a_2 \end{array} \mid \begin{array}{c} 3 \quad 2 \\ a_1 a_0 \end{array}$$

to

$$\begin{array}{c} 5 \quad 2 \\ a_2 a_3 \end{array} \mid \begin{array}{c} 2 \quad 3 \\ a_0 a_1 \end{array}$$

Exchanging a pair of digits between P and M

Exchanging a pair of base b index digits between P and M is equivalent to performing a transpose of a $b \times b$ square matrix, repeated several times to move over the identity memory digits and over several clusters of processors for the identity processor digits.

An example of an exchange of a pair of index digits between P and M would be to convert the mapping

$$\begin{matrix} 2 & 5 \\ a_3 & a_2 \end{matrix} \mid \begin{matrix} 3 & 5 \\ a_1 & a_0 \end{matrix}$$

to

$$\begin{matrix} 2 & 5 \\ a_3 & a_0 \end{matrix} \mid \begin{matrix} 3 & 5 \\ a_1 & a_2 \end{matrix}$$

As there are $b-1$ data elements to be moved per processor, the most efficient algorithm would perform the permutation in $b-1$ read/communicate/write steps. Function 6.1, `exchange_dig`, performs the permutation using the optimal number of steps. This algorithm is similar to an algorithm of Kuszmaul's [42] for transposition of a square array the size of the processor array, and operates by exchanging elements around the main diagonal with a wrap-around to ensure that all the processors are operating continuously. As shown, the algorithm works only if the digits to be exchanged are in the least significant digits of the device dimensions, but it is straightforward to modify it to allow a pair of P/M digits in any position to be exchanged. Figure 6.3 shows the data movement of the algorithm for an exchange of base 6 digits.

Data replication

When a $*$ -digit appears in a source or destination index digit map, data may be ignored or replicated respectively. If a $*$ -digit appears in the source index digit map and not in the destination map, data values may be ignored and convenient values of the $*$ -digit be used. If a $*$ -digit appears only in the destination index digit map, data values must be replicated; one method could use recursive doubling to create k copies of the data in $\lceil \log_2 k \rceil$ steps.

Resizing/compounding index digits

In order to make space for an exchange of index digits between P and M we may wish to group together neighbouring digits in the device index into a larger compound digit. This operation opens up space along either the memory or processor dimensions. As the operation only affects the places of either the processor indices or the memory indices, it can be performed with either a memory or a processor permutation.

Function 6.1 *Exchanging a pair of digits between P and M*

```

int base;                                /* Base of digits being exchanged */
plural int digit;                        /* Digit value of processor digit */
plural char *m;                          /* Base address of memory array */
plural char z0, z1;                      /* Temps used for reading, writing data */
plural int moff0, moff1;                 /* Offsets to lhs and rhs of diagonal */

int i;

exchange_dig()
{
    moff0 = digit;                        /* Begin with offsets on diagonal */
    moff1 = digit;

    for (i=1; i<=(digit-1)/2; i++)
    {
        moff0++; moff1--;                /* Move away from diagonal */

        if (moff0 ≥ base) moff0 = 0;      /* Wrap around off rhs */
        if (moff1 < 0) moff1 = base-1;    /* Wrap around off lhs */

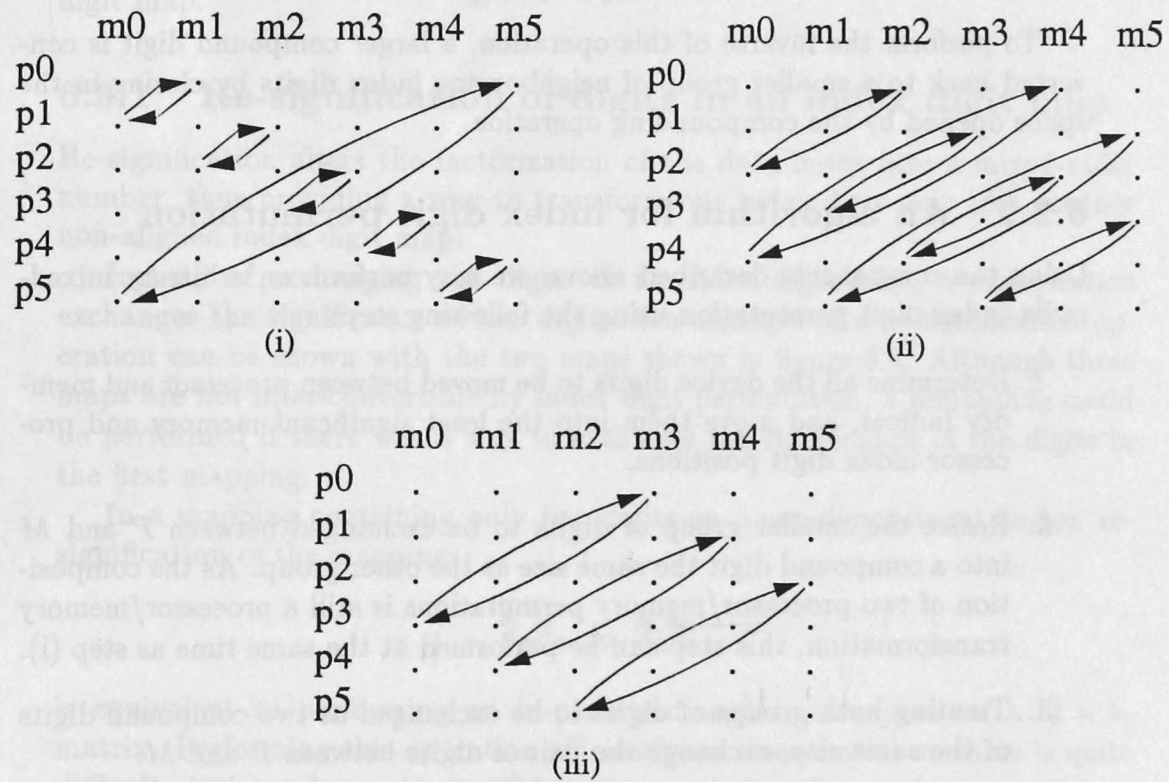
        z0 = *(m+moff0); z1 = *(m+moff1); /* Read data */
        router[moff0].z0 = z0; router[moff1].z1 = z1; /* Send data */
        *(m+moff0) = z1; *(m+moff1) = z0; /* Write data */
    }

    if ((digit & 1) == 0)                 /* Cope with even width case */
    {
        moff0++;

        if (moff0 ≥ base) moff0 = 0;      /* Wrap around */

        z0 = *(m+moff0);
        router[moff0].z0 = z0;
        *(m+moff0) = z0;
    }
}

```

Figure 6.3: Data movement in base 6 P/M digit exchange

Of course, there must be enough space along the device dimension to perform the compounding operation. However, if digits are to be exchanged between two device dimensions, both dimensions will have enough space to hold the larger.

An example of a compounding of a group of index digits would be to convert the mapping

$$\begin{matrix} 16 \\ a_3 \end{matrix} \mid \begin{matrix} 16 & 5 & 3 \\ a_2 & a_1 & a_0 \end{matrix}$$

to

$$\begin{matrix} 16 \\ a_3 \end{matrix} \mid \begin{matrix} 16 & \overbrace{5 & 3}^{16} \\ a_2 & a_1 a_0 \end{matrix}$$

To perform the inverse of this operation, a larger compound digit is converted back to a smaller group of neighbouring index digits by closing in the space opened by the compounding operation.

6.2.2 An algorithm for index digit permutation

Using the components described above, we may perform an arbitrary mixed-radix index digit permutation using the following steps:

- i. Determine all the device digits to be moved between processor and memory indices, and move them into the least significant memory and processor index digit positions.
- ii. Resize the smaller group of digits to be exchanged between P and M into a compound digit the same size as the other group. As the composition of two processor/memory permutations is still a processor/memory transformation, this step can be performed at the same time as step (i).
- iii. Treating both groups of digits to be exchanged as two compound digits of the same size, exchange the pair of digits between P and M .
- iv. Resize the compound digit produced in step (ii) to its original size.
- v. All the digits are now stored on the appropriate device dimensions, so a combined processor/memory remapping will complete the operation.

Steps (i) and (ii) both involve only processor/memory permutations, so can be composed and performed with one application of the function `permute_mp` described in section 4.4.6, with the usual restrictions and fixes if the number of elements to be permuted is greater than the number of processors.

Step (iii) can be performed using the function `exchange_dig`.

Steps (iv) and (v) can be composed in the same way as steps (i) and (ii).

Although this algorithm is not optimal, it is only three times slower than the radix 2 algorithm.

6.3 Non-aligned index digit remapping

The problem of remapping between two non-aligned maps cannot be performed using index digit permutation, thus we cannot use the remapping algorithm outlined in section 6.2.2. Three alternative approaches have been examined, but all approaches have undesirable characteristics and their analysis has been largely empirical. However, by trading off execution time with memory requirements, at least these operations can be performed.

A new operation on index digit maps, *re-signification*, expresses a way of altering an index digit map to bring it closer to another non-aligned index digit map.

6.3.1 Re-signification of digits in an index digit map

Re-signification alters the factorization of the data index into a mixed-radix number, thus providing a way to transform one index digit map into another non-aligned index digit map.

Instead of exchanging the digits in an index digit map, *re-signification* exchanges the significance of two digits. An example of a re-signification operation can be shown with the two maps shown in figure 6.2. Although these maps are not inter-convertible by index digit permutation, a remapping could be performed if there was a way to exchange the significance of the digits in the first mapping.

In a mapping containing only two digits on a one-dimensional device, re-signification of the mapping

$$\begin{matrix} b_1 & b_0 \\ a_1 & a_0 \end{matrix} \text{ to } \begin{matrix} b_1 & b_0 \\ a_0 & a_1 \end{matrix}$$

is equivalent to performing an in-place rectangular transpose on a $b_0 \times b_1$ matrix. Performing this operation efficiently on a sequential machine is quite difficult, and involves a trade-off between execution time and memory usage [69]. Finding any regular parallelism in this operation is also very hard.

It is worth noting that for a fixed-radix factorization of the data index, permuting the index digits and permuting their significance are identical operations, so that re-signification is a natural generalization from fixed radix index digit permutation to mixed radix operations.

Re-signification of two digits a_i and a_j of differing bases changes the data index places of all digits with significance between i and j . This problem is harder to resolve than the change of device index place that occurs when two digits of differing bases are exchanged within the same device dimension. When digits are exchanged within the same device dimension, the places of the device index only change in digits within that device dimension, requiring memory-only or processor-only remapping operations. However, when significance is

exchanged between two digits mapped to the same device dimension, the places of intervening digits mapped to any other device dimensions will also change, entailing a more complicated data permutation. For example, inter-converting between the two maps shown here cannot be performed by a memory-only or processor-only remapping operation, or even a composition of the two:

		M0	M1	M2	M3	M4	M5
$\begin{smallmatrix} 2 \\ a_1 \end{smallmatrix} \mid \begin{smallmatrix} 2 & 3 \\ a_2 & a_0 \end{smallmatrix} \Rightarrow$	P0	0	1	2	6	7	8
	P1	3	4	5	9	10	11

		M0	M1	M2	M3	M4	M5
$\begin{smallmatrix} 2 \\ a_1 \end{smallmatrix} \mid \begin{smallmatrix} 2 & 3 \\ a_0 & a_2 \end{smallmatrix} \Rightarrow$	P0	0	4	8	1	5	9
	P1	2	6	10	3	7	11

However, this is not a problem if either the two digits are adjacent in the data index or all the intervening digits are mapped to the same device dimension.

6.3.2 Re-signification within device dimensions

If the memory array is large enough or there are sufficient unused processors on the device, it is often possible to perform re-signification by using a combination of index digit permutation operations and processor/memory operations. Depending on the amount of memory available, different numbers of operations may be required. Figure 6.4 shows a possible sequence of operations to transform the mapping

$$\begin{smallmatrix} 32 & 32 \\ a_3 & a_2 \end{smallmatrix} \mid \begin{smallmatrix} 31 & 31 \\ a_1 & a_0 \end{smallmatrix} \text{ to } \begin{smallmatrix} 32 & 32 \\ a_1 & a_0 \end{smallmatrix} \mid \begin{smallmatrix} 31 & 31 \\ a_3 & a_2 \end{smallmatrix}$$

This is a conversion from two-dimensional hierarchical to two-dimensional cut'n'stack. The original index digit map uses $31 \times 31 = 961$ bytes per processing element, but some intermediate stages require $31 \times 32 = 992$ bytes per processing element.

We have not developed heuristics to perform this type of operation in a small number of steps, nor established how to decide if a given remapping is possible in the available memory and number of processors. However, it is clearly less efficient to perform some unaligned data remappings by this method than general index digit permutations. Unfortunately, some mappings cannot be interconverted by this method, such as

$$\begin{smallmatrix} 1024 \\ a_1 \end{smallmatrix} \mid \begin{smallmatrix} 1023 \\ a_0 \end{smallmatrix} \text{ to } \begin{smallmatrix} 1024 \\ a_0 \end{smallmatrix} \mid \begin{smallmatrix} 1023 \\ a_1 \end{smallmatrix}$$

This remapping would require $1024 \times 1023 = 1047552$ bytes in a single processor to be performed in memory, which is not only unrealistic but would take a long time.

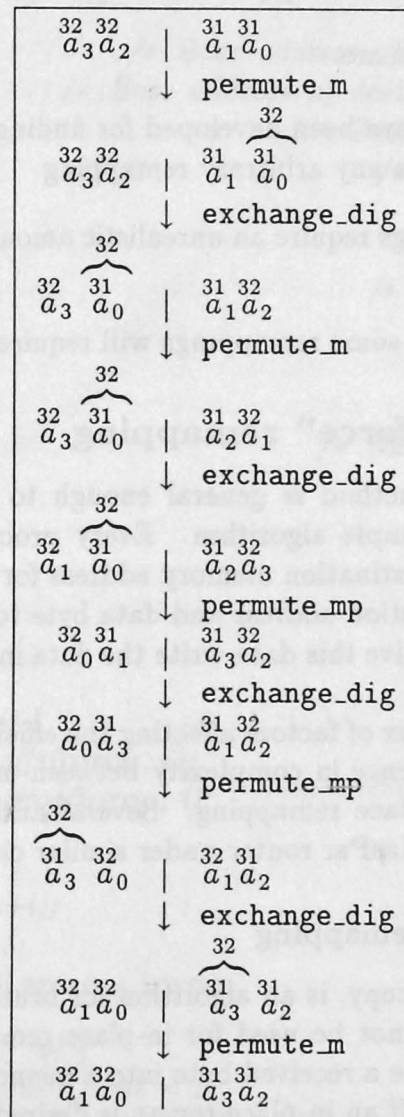


Figure 6.4: Example re-signification within device dimensions

To summarize, the advantages of this technique are:

- It uses the same efficient algorithms as general index digit permutation
- Some remappings may be performed with little memory overhead
- Some remappings may be performed in only a few steps

The disadvantages are:

- No heuristics have been developed for finding a good sequence of operations to perform any arbitrary remapping
- Some remappings require an unrealistic amount of working memory storage
- It appears that some remappings will require a large number of steps

6.3.3 “Brute force” remapping

The “brute force” method is general enough to perform any regular data remapping with a simple algorithm. Every processor calculates a destination processor and destination memory address for each byte in its data array and sends the destination address and data byte to the destination processor. Processors which receive this data write the data into the appropriate memory location.

There are a number of factors affecting the efficiency of this technique, and there is a large difference in complexity between using brute force remapping for copying and in-place remapping. Several authors have investigated the performance of the MasPar router under similar circumstances [18, 54, 60].

Brute force copy remapping

Function 6.2, `brute_copy`, is an algorithm for brute force copy remapping.

This function cannot be used for in-place remaps, because a destination processor cannot write a received byte into a memory location still containing untransmitted data. If an in-place remap is desired and a temporary buffer is used, the memory overhead is the size of the data array.

Because there may be a great deal of regularity in the data permutation being performed, there is the possibility that a large amount of contention could occur when a large number of processors attempt to talk to only a few. A pathological case occurs when using the brute force method to transpose a square $|P| \times |P|$ array, with one dimension stored along the memory axis and one along the processor axis. In the first step, all processors attempt to send their data to processor 0, and only one can succeed. Because so much time

Function 6.2 *Brute force copying*

```

/* Returns dest offset for byte stored at proc b, address offset m */
extern plural int dest_offs(plural int p,plural int m);
/* Returns dest processor for byte stored at proc b, address offset m */
extern plural int dest_proc(plural int p,plural int m);

plural char *m_src;          /* Base address of source memory array */
plural char *m_dst;          /* Base address of destination memory array */
plural int offs0;             /* Destination offset to send */
plural int proc0;             /* Processor to send to */
plural int offs1;             /* Received offset */
plural int proc1;             /* Sending processor */
plural int sent;              /* Set if send successful */
plural char z0;               /* Data to send */
plural char z1;               /* Received data */
plural int i;                 /* Loop variable */
int nbytes;                   /* Number of bytes to copy */

brute_copy()
{
    sent = 0;
    i = 0;

    while (i < nbytes) {
        offs0 = dest_offs(iproc, i);
        proc0 = dest_proc(iproc, i);
        sent = 0;

        z0 = *(m_src+i);
        all proc1 = -1;
        router[proc0].proc1 = iproc;

        all if (proc1 ≥ 0) {
            router[proc1].sent = 1;
            offs1 = router[proc1].offs0;
            z1 = router[proc1].z0;
            *(m_dst+offs1) = z1;
        }

        if (sent) {
            i++;
        }
    }
}

```


is wasted sending data which is overwritten, the brute force algorithm in this case is worse than a sequential algorithm reading and writing one data item at a time.

One approach which can greatly improve the efficiency of a brute-force copy is to *skew* the initial addresses read by every processor in an attempt to evenly distribute the destination processors. One very simple skewing scheme is to set the initial memory offset to the value of *iproc*. In the case of our square transposition example, this would distribute the communications load perfectly. Its effectiveness in more complex situations is unclear, and a random skewing method has been used with some success. However, in well-controlled problems skewing can be extremely effective, as will be shown in the next section.

The efficiency of the brute-force remap could also be substantially affected by the cost of computing the destination address and processor for each byte read from memory. If a moderately complicated remap is being performed, several dozen multiplication/division/modulus operations may need to be performed between each communication step, and these may take a substantial fraction of the execution time of the algorithm.

Brute force in-place remapping

An algorithm for brute force in-place remapping requires more careful attention than a copy remapping. Because received bytes must be stored in the same memory array as they are sent from, each processor must be careful not to overwrite data that have not yet been sent, which means that each processor must keep track of the memory locations that have been freed.

A flag array, T , may be used to mark freed memory locations and would require an overhead of $|A|/8$ bytes.

Because several processors may attempt to send data to a single processor simultaneously, it is necessary for each processor to maintain a stack of items to be written to memory; it is not possible to clear the items immediately as they are received, because clearing an item usually involves sending another. It is also not easy to disallow sends to processors whose stacks are full, because deadlock situations arise.

It is also not easy to ensure that the amount of stack required will always be substantially smaller than the number of bytes in the array being sent, but a method is shown in the next section to perform a brute-force remapping with a stack of fixed size.

As with copy remapping, any regularity in the data permutation being performed may cause a large amount of contention for a small number of processors. The problem of maintaining a data stack compounds this problem, as distributed data cannot be handled immediately, and may accumulate in small sets of processors at a time.

6.3.4 Re-signification across device dimensions

Instead of using the brute force method to perform a complete data remapping, it can be used to perform a smaller part of the problem: the re-signification of a pair of digits adjacent in the data index and mapped to different device dimensions. An example of such an operation would be to change the mapping

$$\overset{3}{a}_1 \mid \overset{2}{a}_0 \text{ to } \overset{3}{a}_0 \mid \overset{2}{a}_1$$

Re-signifying a pair of index digits by this method has a number of advantages which make the brute-force method appear more attractive.

Computing the data permutation

Computing the destination address for a data element during an arbitrary mixed index digit remapping may require dozens of multiply, divide and modulus operations, which may be a much greater overhead than memory-access and communication times. However, computing the data permutation of a significance exchange costs significantly less.

As has been mentioned previously, exchanging the significance of a pair of digits $\overset{b_1}{a}_1$ and $\overset{b_0}{a}_0$ is equivalent to performing a rectangular transpose of a $b_1 \times b_0$ array. When performing the data permutation within a single device dimension, a one-dimensional destination address can be computed from a source address by the expression:

$$d_{dst} = b_0 \cdot (d_{src} \bmod b_1) + (d_{src} / b_1)$$

An apparently more simple expression is given by Knuth [39]:

$$d_{dst} = (b_0 \cdot d_{src}) \bmod (b_0 \cdot b_1 - 1)$$

Unfortunately, the intermediate values in this expression are often too large to be represented in a machine-address sized word-length, causing overflows.

When a pair of digits is stored across two dimensions, the device address is more conveniently manipulated as two components, a memory offset m and a processor index p , where (neglecting any other digits, and assuming $|M| = b_0$):

$$d_{src} = m + b_0 \cdot p$$

The destination address can be found as two destination components by combining the source components into a one-dimensional device address and extracting the destination components:

$$\text{For the remap } \overset{b_1}{a}_1 \mid \overset{b_0}{a}_0 \Rightarrow \overset{b_1}{a}'_0 \mid \overset{b_0}{a}'_1$$

Letting:

$$\begin{aligned}
 a &= \overset{b_1}{a_1} \overset{b_0}{a_0} = \overset{b_0}{a'_1} \overset{b_1}{a'_0} \\
 &= b_0.a_1 + a_0 = b_1.a'_1 + a'_0 \\
 d_{src} &= \overset{b_1}{a_1} | \overset{b_0}{a_0} = b_0.a_1 + a_0 \\
 d_{dst} &= \overset{b_1}{a'_1} | \overset{b_0}{a'_0} = b_0.a'_1 + a'_0 \\
 p_{src} &= d_{src}/b_0 \\
 m_{src} &= d_{src} \bmod b_0 \\
 p_{dst} &= d_{dst}/b_0 \\
 m_{dst} &= d_{dst} \bmod b_0
 \end{aligned}$$

Now:

$$\begin{aligned}
 d_{dst} &= b_0.a'_0 + a'_1 \\
 &= b_0.(d_{src} \bmod b_1) + (d_{src}/b_1) \\
 p_{dst} &= d_{dst}/b_0 \\
 &= (b_0.(d_{src} \bmod b_1) + (d_{src}/b_1))/b_0 \\
 &= d_{src} \bmod b_1 \\
 &= (b_0.a_1 + a_0) \bmod b_1 \\
 &= (b_0.p_{src} + m_{src}) \bmod b_1 \\
 m_{dst} &= d_{dst} \bmod b_0 \\
 &= (b_0.(d_{src} \bmod b_1) + (d_{src}/b_1)) \bmod b_0 \\
 &= d_{src}/b_1 \\
 &= (b_0.a_1 + a_0)/b_1 \\
 &= (b_0.p_{src} + m_{src})/b_1
 \end{aligned}$$

Although this derivation is shown for a_0 and a_1 , it could be used for any re-signification of two adjacent data digits where the data digit of smaller significance was initially stored on the memory array. When the situation is reversed, the following derivation is obtained:

$$\text{For the inverse } \overset{b_1}{a'_0} | \overset{b_0}{a'_1} \Rightarrow \overset{b_1}{a_1} | \overset{b_0}{a_0}$$

$$\begin{aligned}
 p_{src} &= (b_0.m_{dst} + p_{dst})/b_1 \\
 m_{src} &= (b_0.m_{dst} + p_{dst}) \bmod b_0
 \end{aligned}$$

Thus, computing the destination address for each data element requires a multiply, a division and a modulus. Unless there is plenty of memory to spare,

it is not realistic to pre-compute these values (except the value of $b_0.p_{src}$), and there is no obvious way to perform strength reduction. If the digits being exchanged are not the least-significant digits in the device dimensions, one or two more multiplies may be required to put the digit in its correct place.

Smaller memory overhead

Because the size of the memory array being permuted is the size of the memory digit being re-signified rather than that of the whole array, the memory overhead will usually be much smaller, unless the digit represents the whole memory array.

Common factors

If the two digits to be re-signified have any common factors, the re-signification can be combined with an index digit permutation to reduce the size of the memory digit and hence the memory overhead. This also has the effect of linearizing the cost of a re-signification, as will be shown later. Initially let us assume we have a function to perform a re-signification of adjacent data digits across device dimensions, `resig_mp`.

As an example, assume we wish to perform a re-signification

$$a_1^{c_1} | a_0^{c_0} \Rightarrow a_0^{c_1} | a_1^{c_0}$$

with both c_0 and c_1 divisible by b_0 . Letting

$$b_1 = c_0/b_0$$

$$b_2 = c_1/b_0$$

the re-signification may be performed with the following steps:

$$\begin{array}{rcl}
 a_1^{c_1} & | & a_0^{c_0} \\
 \downarrow & & \text{relabel} \\
 a_3^{b_0} a_2^{b_2} & | & a_1^{b_1} a_0^{b_0} \\
 \downarrow & & \text{resig_mp} \\
 a_3^{b_0} a_1^{b_2} & | & a_2^{b_1} a_0^{b_0} \\
 \downarrow & & \text{exchange_dig} \\
 a_0^{b_0} a_1^{b_2} & | & a_2^{b_1} a_3^{b_0} \\
 \downarrow & & \text{permute_mp} \\
 a_1^{b_2} a_0^{b_0} & | & a_3^{b_1} a_2^{b_0} \\
 \downarrow & & \text{relabel} \\
 a_0^{c_1} & | & a_1^{c_0}
 \end{array}$$

6.3.5 Brute-force P/M re-signification

Because the behaviour of the brute-force algorithm is hard to analyse in a formal way, we have performed empirical tests to examine its behaviour on the re-signification problem. These tests show that in the absence of a more clever technique, the brute-force algorithm performs reasonably well in terms of both number of communication steps and size of working memory storage, and performs best in those situations least amenable to application of the other remapping techniques.

Execution time versus memory size

Because the programs used for these experiments were written as experimental algorithms without optimization and with the inclusion of debugging and profiling information, the timing information should be viewed in a relative way; by tuning the code, it would be possible to obtain speed-ups of a factor of at least two.

If the algorithm is well parallelized, the execution time should be directly proportional to memory size, and the size of the processor digit should not greatly affect the execution time.

However, on the MasPar MP-1, performing remappings with small processor digits would lower the execution time unrealistically, because only a subset of processors in a cluster would need to communicate. To make the execution times more realistic, a fixed processor digit that rounds up the size of the problem as closely as possible to the processor array size is added to ensure that most processors are active. For example, in a 1024-element processor array, instead of performing the remapping

$$\begin{array}{c|c} 27 & 673 \\ a_1 & a_0 \end{array} \Rightarrow \begin{array}{c|c} 27 & 673 \\ a_0 & a_1 \end{array}$$

the mapping

$$\begin{array}{c|c} 37 & 27 \\ a_2 & a_1 \end{array} \mid \begin{array}{c|c} 673 & \\ a_0 & \end{array} \Rightarrow \begin{array}{c|c} 37 & 27 \\ a_2 & a_0 \end{array} \mid \begin{array}{c} 673 \\ a_1 \end{array}$$

would be performed.

Memory overhead

In an ideal remapping algorithm, there should be no memory overhead and all temporary storage and scratch values should be carried in processor registers and computed quickly as needed. The radix 2 algorithms are almost ideal, requiring only a small fixed amount of memory per processor to guide their operations, which can be performed with all intermediate storage in processor registers.

The brute-force method is greatly inferior, as it requires a large amount of memory storage which in some cases may be larger than the data array being remapped.

There are two components to memory use: a tag array, indicating which bytes have been sent to their destinations leaving their sources safe to write into, and a stack, which contains data items awaiting a write to memory.

The size of the tag array is determined solely by the number of elements to be transferred, and can be implemented with one bit per element.

Each data element transferred requires at least two bytes for the stack: at least one byte for the data element and at least one byte for the memory offset. The size of the stack required is much harder to predict; if its size is not restricted in any way, the actual size required is almost always very reasonable but occasionally can be as large as the memory array.

6.3.6 Brute force performance

Test problems

Because of the lack of formal analytical methods, we have analysed the brute-force algorithm with brute-force analysis, which consists of generating many random remappings and measuring the time and space needed to perform them.

Execution time

Figure 6.5 shows the execution time of brute-force forward re-signification over 2000 random re-signification problems. It can be seen that performance is very nearly linearly related to memory digit size, which means the behaviour of the algorithm is reasonably predictable.

However, there are several timings far above the line which are much worse than would be expected for their memory size. Upon closer examination, most of these timings are associated with processor/memory digits with common factors. This behaviour may be caused by the increased regularity introduced into the problem because of the common factors, as discussed earlier.

If two digits have common factors, re-signification can be performed with an index digit permutation and a smaller re-signification of digits with no common factors. Because the digits in the smaller problem have no common factors, its execution time should be closer to linear. Of course, the extra memory digit will mean that the re-signification will need to be performed multiple times, but the total execution time will be smaller.

There are also several timings which lie far below the line which are much better than would be expected for their memory size. These timings are associated with very small processor digits, in which there can only be a small

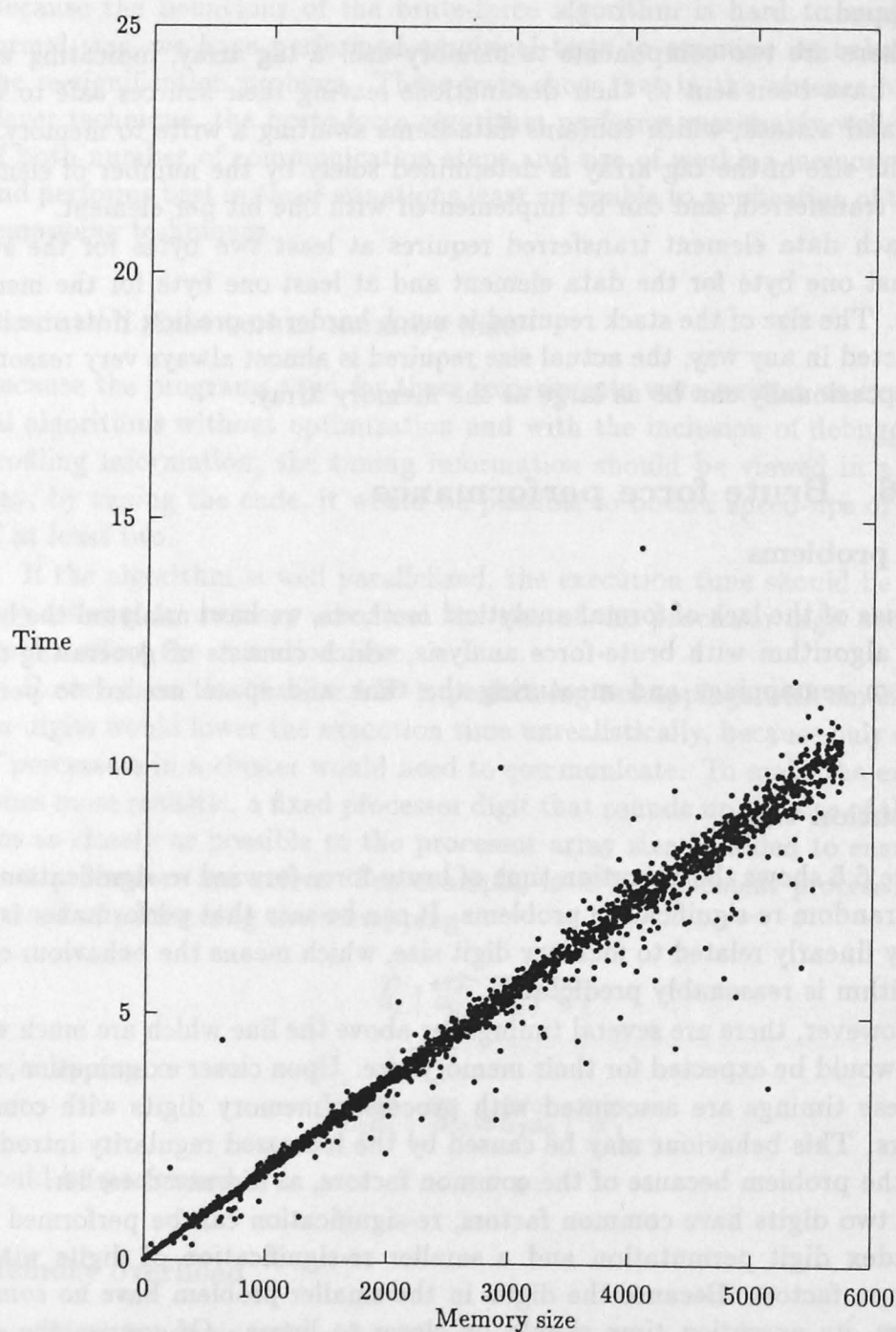


Figure 6.5: Brute force re-signification on 2000 random problems. Times are in seconds, memory size in bytes. A single data point at (2502, 98.7) has been removed to preserve the graph's aspect ratio

number of processors attempting to send to another, and hence a very small amount of processor contention.

6.3.7 Restricting to relatively prime digits

Figure 6.6 shows the execution time of brute-force forward re-signification over 2000 random re-signification problems which have been selected to ensure that no common factors are present between the digits involved.

This table shows that there are many fewer remapping problems in this set which cause unpredictably poor performance. However, there is still one remapping problem which takes much longer than would be predicted; it is not known why it causes such poor performance.

6.3.8 Skewing initial memory addresses

Figure 6.7 shows the execution time of brute-force reverse re-signification using the first hundred digits of those used for in figure 6.6. These timings are clearly much worse than for the forward case: The different form for the forward and inverse re-signification has a significant effect on the effectiveness of the brute-force approach. If $|M| > |P|$, the first step in the forward re-signification involves communication to several different processors. However, in the reverse re-signification, the first step involves every processor attempting to communicate with processor 0, resulting in very poor performance. This problem can be cured with a skewing scheme.

To prevent every processor attempting to send to a small set of processors in the first step of the algorithm, the processors should be given an initial set of addresses to read whose destination processors are well distributed over the array.

If $|M| > |P|$, for all processors p the processor's array element

$$\lfloor p \cdot (|M| - 1) / (|P| - 1) \rfloor$$

is sent to a different processor, with the array indices well spread over the array.

If $|M| < |P|$, it is harder to distribute the array entries to ensure that their destinations are distributed across every processor. However, every processor p using a skew factor of

$$p \bmod |M|$$

gives good results.

Figure 6.8 shows the execution time of brute-force reverse re-signification using these skewing strategies, and can be seen to give results very similar to the forward re-signification times.

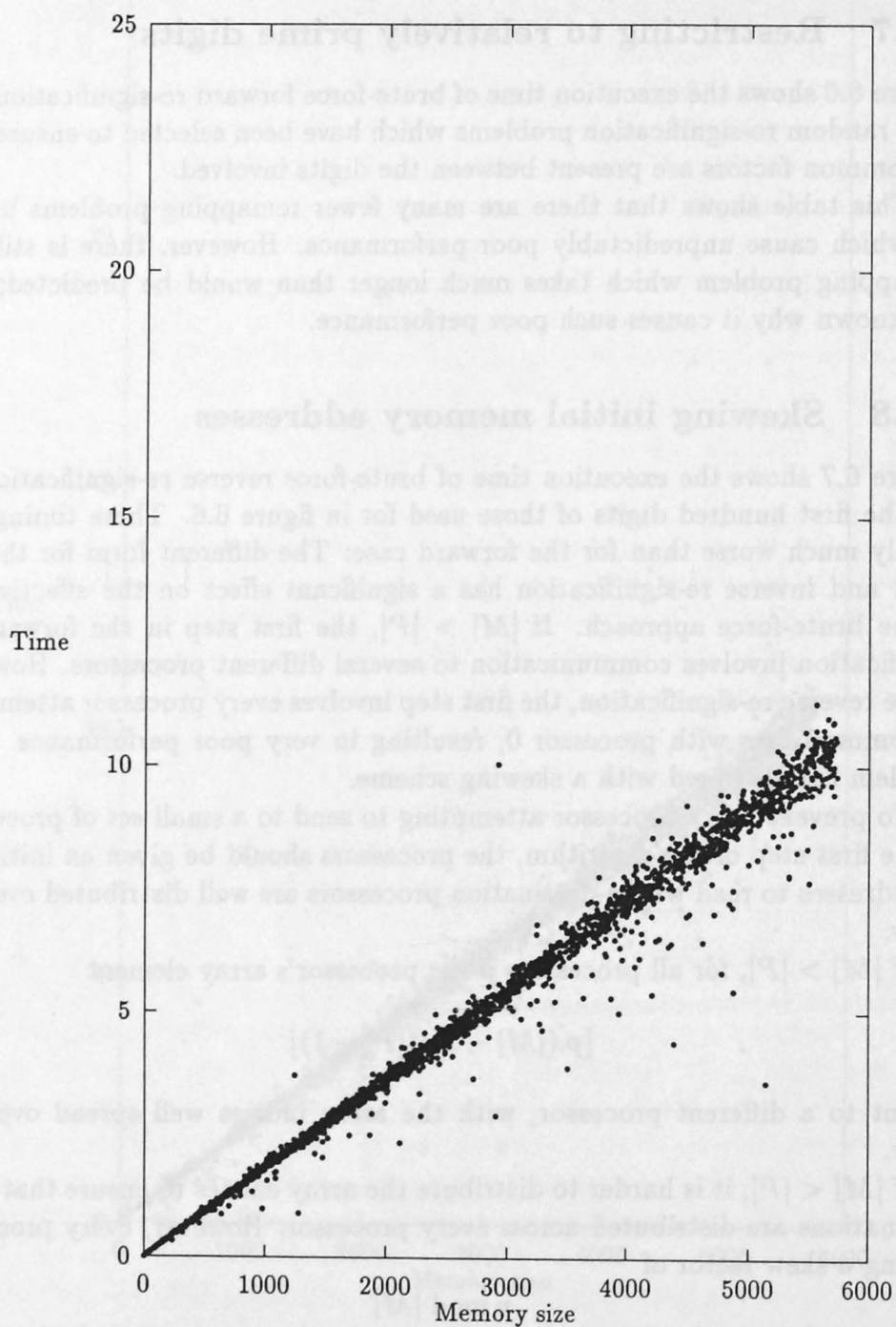


Figure 6.6: Re-signification with no common factors

6.3.1 Restricting stack size

Figure 6.3 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

Figure 6.4 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

Figure 6.5 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

Figure 6.6 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

Figure 6.7 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

Figure 6.8 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

Figure 6.9 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

Figure 6.10 shows the high-water-mark stack usage for the forward remapping problem. It is clear that the stack usage is very high, and that it is not much smaller than the data array to be mapped. This is due to the fact that the stack usage is not restricted, and that the stack usage is not restricted to a fixed size.

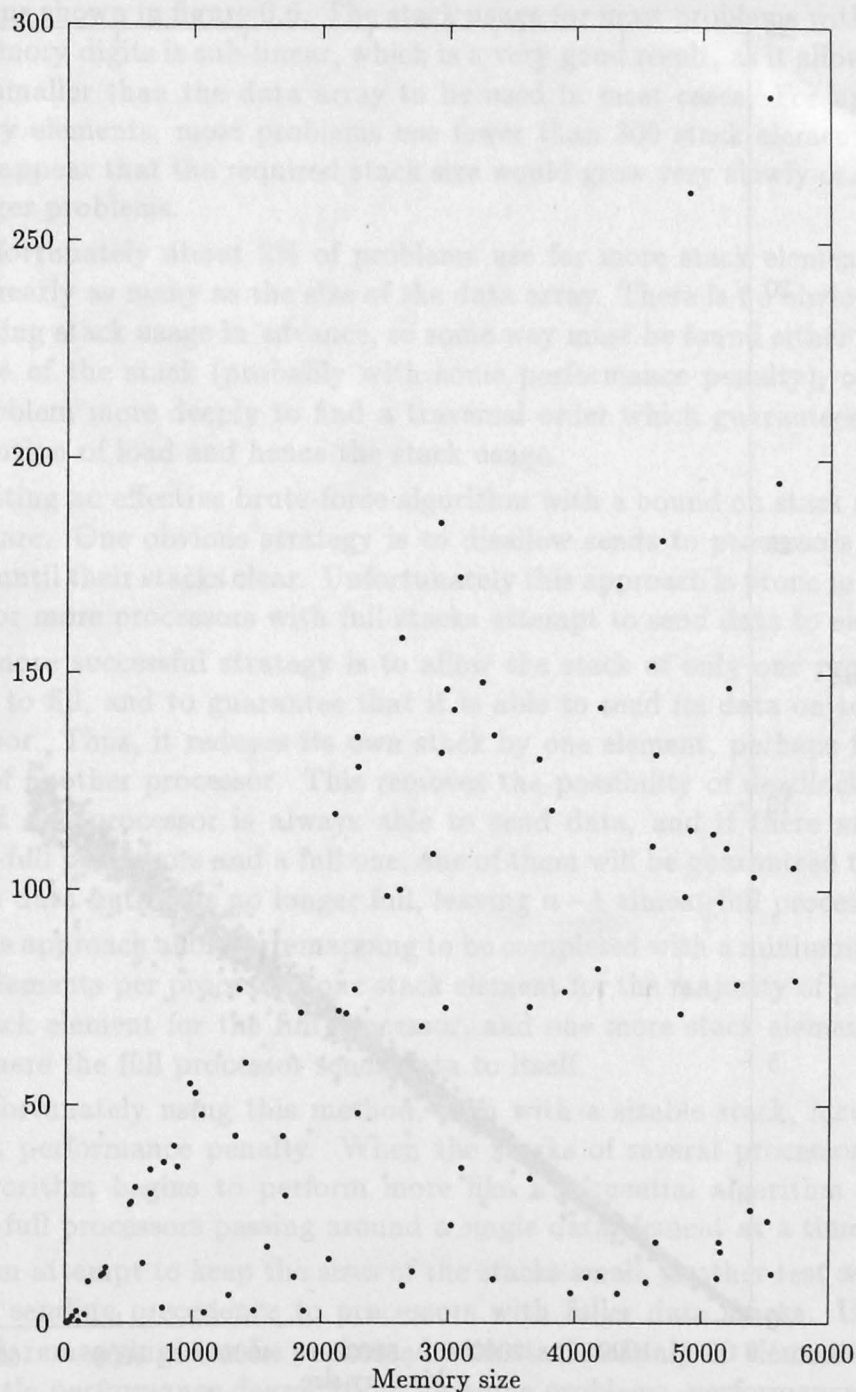


Figure 6.7: Inverse re-signification

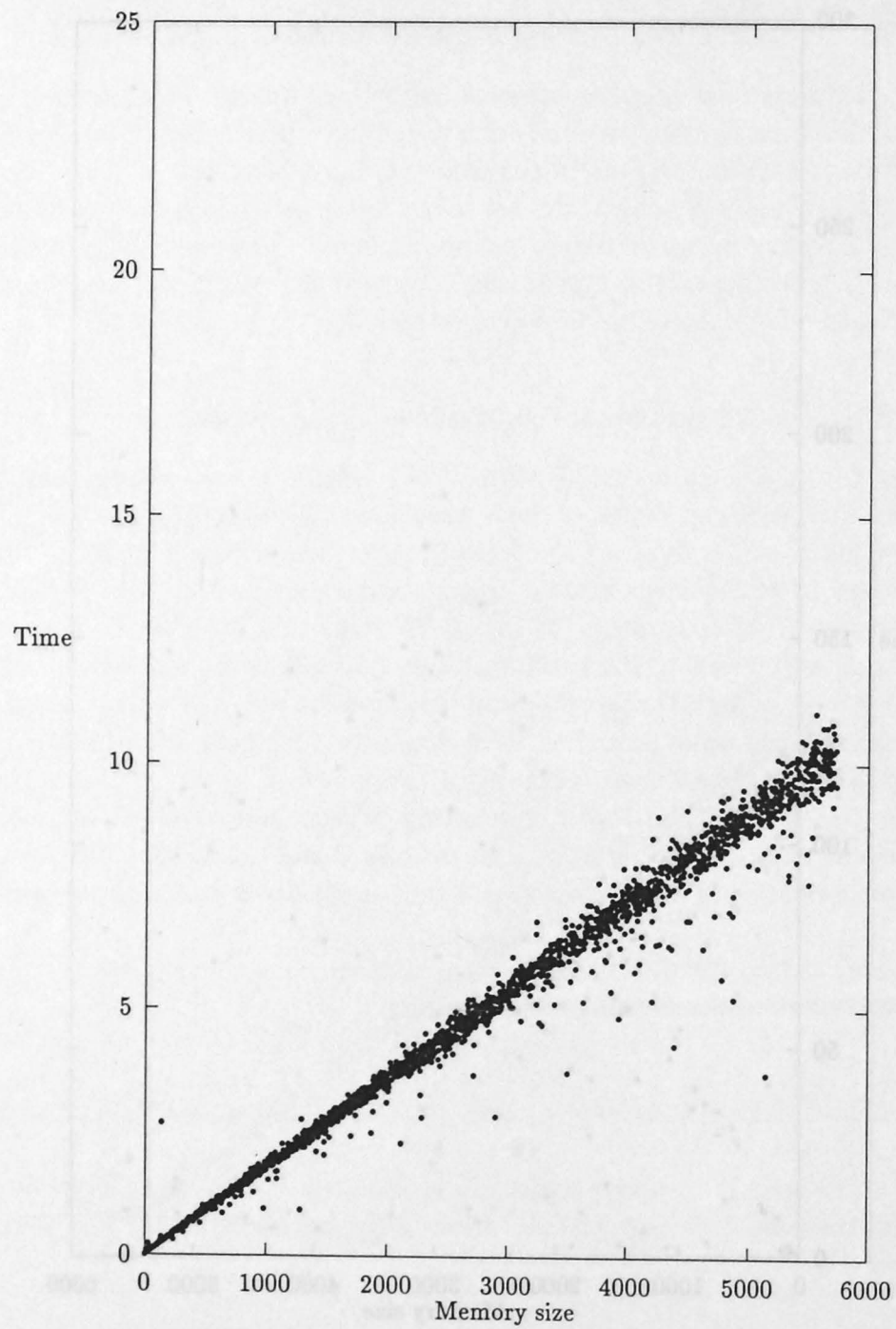


Figure 6.8: Inverse re-signification with address skewing

6.3.9 Restricting stack size

Figure 6.9 shows the high-water-mark stack usage for the forward remapping problems shown in figure 6.6. The stack usage for most problems with increasing memory digits is sub-linear, which is a very good result, as it allows a stack much smaller than the data array to be used in most cases. For up to 6000 memory elements, most problems use fewer than 300 stack elements, and it would appear that the required stack size would grow very slowly or not at all for larger problems.

Unfortunately about 2% of problems use far more stack elements, sometimes nearly as many as the size of the data array. There is no obvious way of predicting stack usage in advance, so some way must be found either to bound the size of the stack (probably with some performance penalty), or analyse the problem more deeply to find a traversal order which guarantees a better distribution of load and hence the stack usage.

Writing an effective brute-force algorithm with a bound on stack size takes some care. One obvious strategy is to disallow sends to processors with full stacks until their stacks clear. Unfortunately this approach is prone to deadlock if two or more processors with full stacks attempt to send data to each other.

A more successful strategy is to allow the stack of only one processor at a time to fill, and to guarantee that it is able to send its data on to another processor. Thus, it reduces its own stack by one element, perhaps filling the stack of another processor. This removes the possibility of deadlock because at least one processor is always able to send data, and if there are $n > 0$ almost-full processors and a full one, one of them will be guaranteed to be able to send data until it is no longer full, leaving $n - 1$ almost-full processors.

This approach allows a remapping to be completed with a minimum of three stack elements per processor; one stack element for the majority of processors, one stack element for the full processor, and one more stack element for the case where the full processor sends data to itself.

Unfortunately using this method, even with a sizable stack, incurs a significant performance penalty. When the stacks of several processors fill up, the algorithm begins to perform more like a sequential algorithm with the almost-full processors passing around a single data element at a time.

In an attempt to keep the sizes of the stacks small, another test was added to give sending precedence to processors with fuller data stacks. Using this method, remappings can be performed with stacks of only 20 elements, usually with little performance degradation; for some problems, performance actually improves because less time is required to empty the processor's stacks. However, there are several problems which show a significant performance penalty.

Figure 6.10 shows the execution time of brute-force forward re-signification with a limit of 50 stack entries.

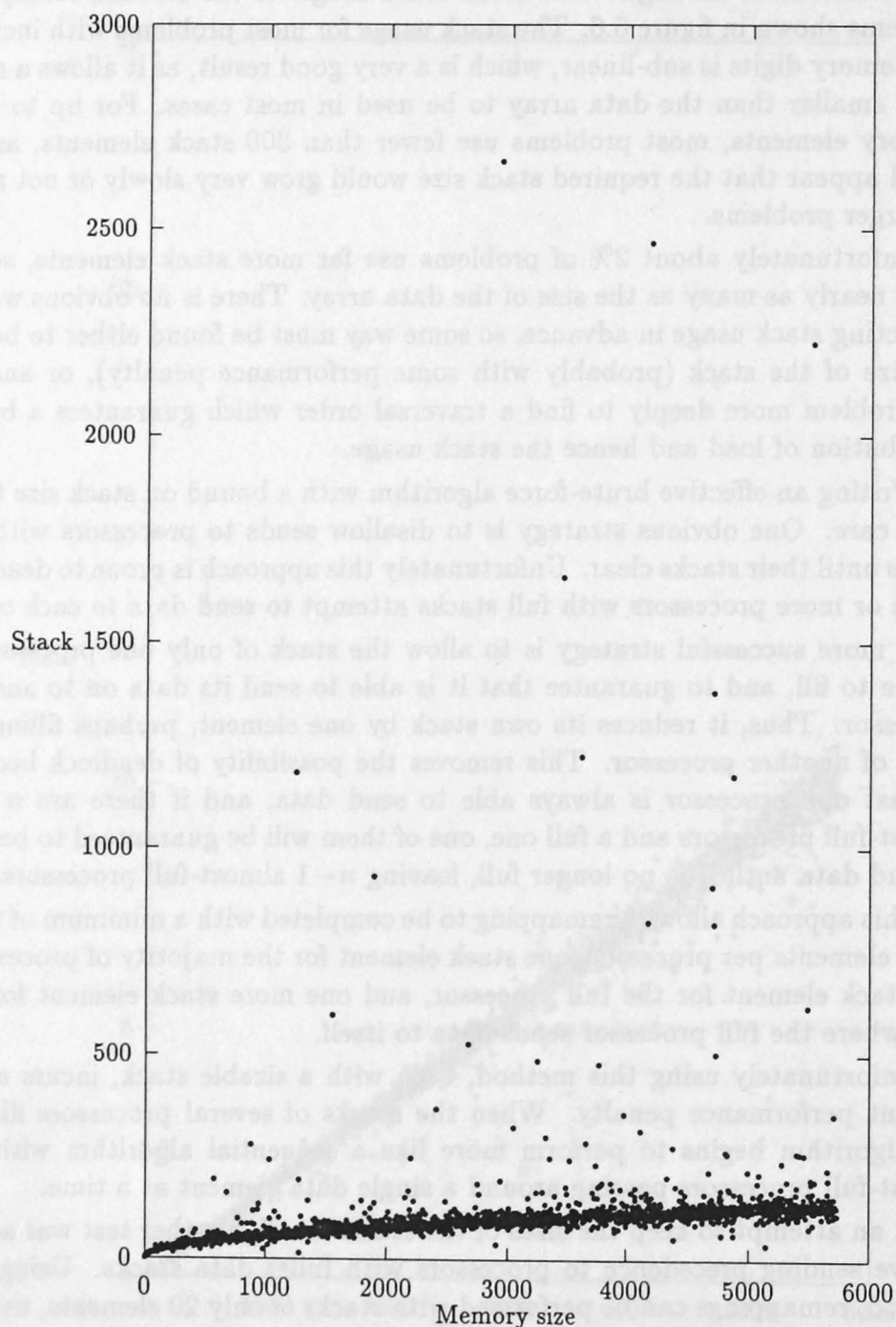


Figure 6.9: Stack high-water-mark with re-signification (stack is measured in number of entries)

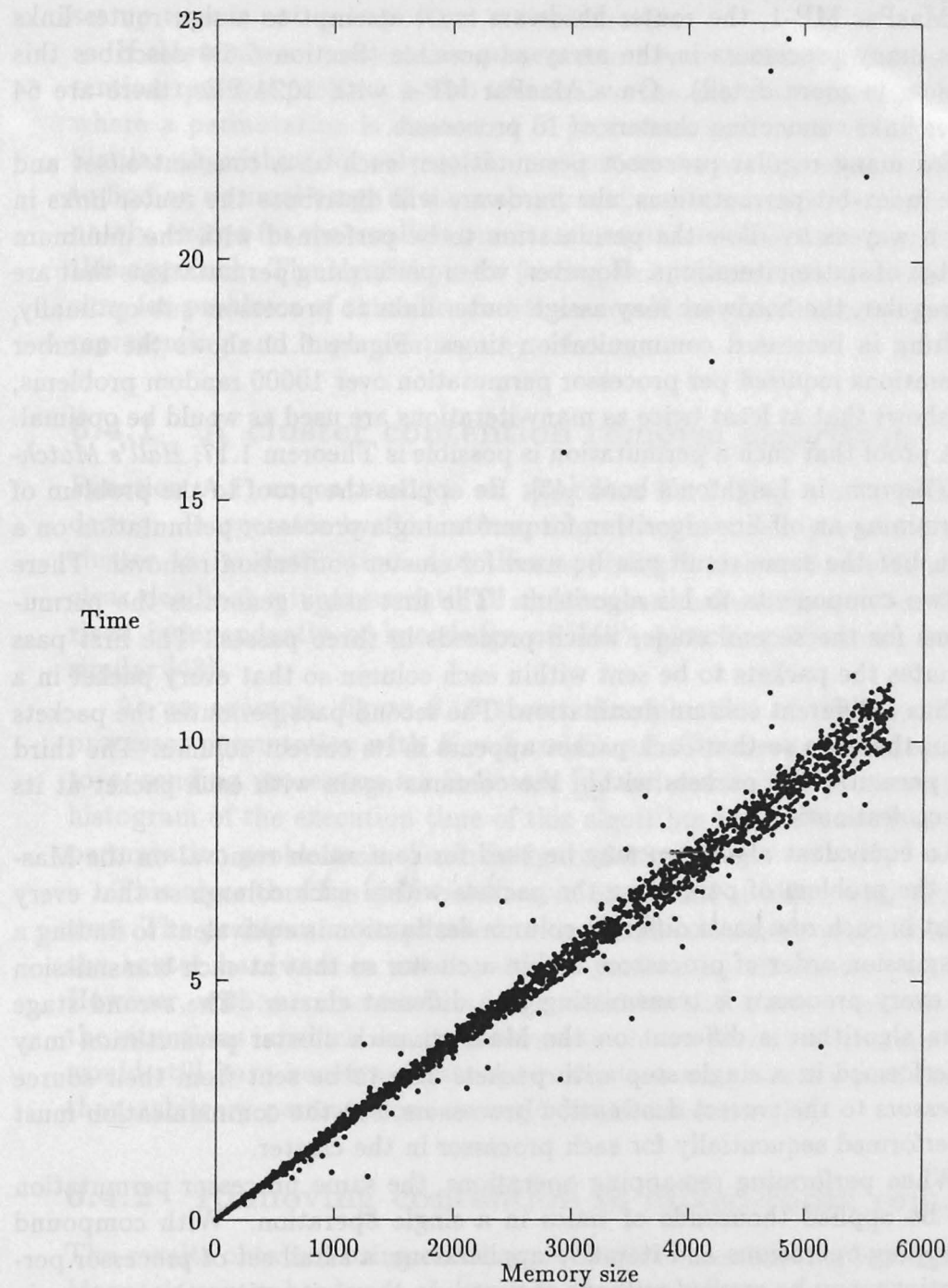


Figure 6.10: Execution time with stack limited to 50 elements

6.4 Cluster contention removal

When performing an arbitrary data permutation between all the processors on the MasPar MP-1, the router hardware must attempt to assign router links to as many processors in the array as possible (Section 5.6.4 describes this problem in more detail). On a MasPar MP-1 with 1024 PEs, there are 64 router links connecting clusters of 16 processors.

For many regular processor permutations, such as a constant offset and some index-bit permutations, the hardware will distribute the router links in such a way as to allow the permutation to be performed with the minimum number of sixteen iterations. However, when performing permutations that are less regular, the hardware may assign router links to processors sub-optimally, resulting in increased communication times. Figure 6.11 shows the number of iterations required per processor permutation over 10000 random problems, and shows that at least twice as many iterations are used as would be optimal.

A proof that such a permutation is possible is Theorem 1.17, *Hall's Matching Theorem*, in Leighton's book [43]. He applies the proof to the problem of determining an off-line algorithm for performing a processor permutation on a mesh, but the same result can be used for cluster contention removal. There are two components to his algorithm. The first stage generates the permutations for the second stage, which proceeds in three passes. The first pass permutes the packets to be sent within each column so that every packet in a row has a different column destination. The second pass permutes the packets within the rows so that each packet appears in its correct column. The third pass permutes the packets within the columns again with each packet at its correct destination.

An equivalent algorithm may be used for contention removal on the MasPar: the problem of permuting the packets within each column so that every packet in each row has a different column destination is equivalent to finding a transmission order of processors within a cluster so that at each transmission step every processor is transmitting to a different cluster. The second stage of the algorithm is different on the MasPar: each cluster permutation may be performed in a single step with packets able to be sent from their source processors to the correct destination processors, but the communication must be performed sequentially for each processor in the cluster.

When performing remapping operations, the same processor permutation may be applied thousands of times in a single operation. With compound remapping operations and iterative applications, a small set of processor permutations may be applied millions of times. In these situations the increase in speed obtained by using a contention-free processor permutation could more than balance the cost of computing it, even if this required several seconds; if a contention-free ordering takes one second to compute, the break-even point occurs with approximately 8000 applications of a processor permutation.

Another justification for performing a large amount of work for modest gains is in applications where human interaction or animation is desired. Such applications require maximum speed in the central body of the program, and set-up time is not as important a consideration.

If clusters were processors and processors were memory locations, the contention removal problem is also identical to the arbitrary remapping problem, where a permutation is desired of C processors each with K data elements. Similar algorithms for solving the contention removal problem could be used to find an optimal usage of the memory and communication network. Unfortunately, except for the smallest permutations, there would be two problems with this approach. The algorithms for finding an optimal solution would be far too slow for problems of this size, and there would be great memory overheads in representing and implementing any solution.

6.4.1 A cluster contention removal algorithm

Function A.1, `uncontend1`, is an algorithm for finding a contention-free ordering. It operates by first attempting to connect one processor from each cluster to its destination, and then applying a series of transformations to clear deadlock situations until all clusters can make a connection. It was derived independently of knowledge of Hall's Matching Theorem, but is very similar [43].

As an example, figure 6.15 shows the operation of the algorithm for a processor permutation with $K = 3$ and $C = 7$. Clusters are labelled a through g , sending processors are indicated \boxed{a} and deadlocked processors as α . A histogram of the execution time of this algorithm for 10000 random processor permutation problems is shown in figure 6.13.

This algorithm has only been tested on a MasPar MP-1 with 1024 processors. The problem is complicated on machines with more processors, because the router hardware is not capable of performing all cluster permutations. However, with more access to a larger machine and a better understanding of the situations in which deadlock occurs in cluster permutations, this approach would still have benefits; cluster permutations are generally performed faster than arbitrary permutations on multi-board machines [18].

6.4.2 Removing contention in mixed-radix remapping

The results obtained using random problems to examine the behaviour of the Maspar's router hardware and to test the contention removal algorithm may not be applicable for index digit permutation problems. For this reason a different group of tests using all the possible exchanges of digits on a 1024 processor MasPar was used to test both the Maspar's router hardware and to test the performance of the contention removal algorithm. As with the

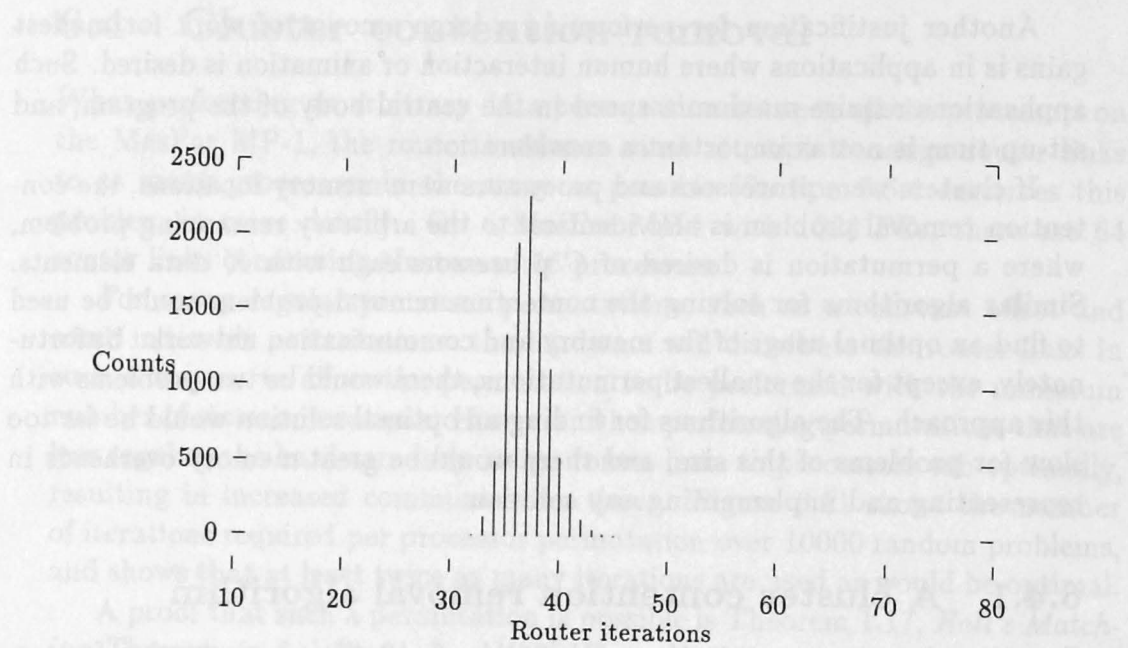


Figure 6.11: Number of router iterations required over 1000 random processor permutations

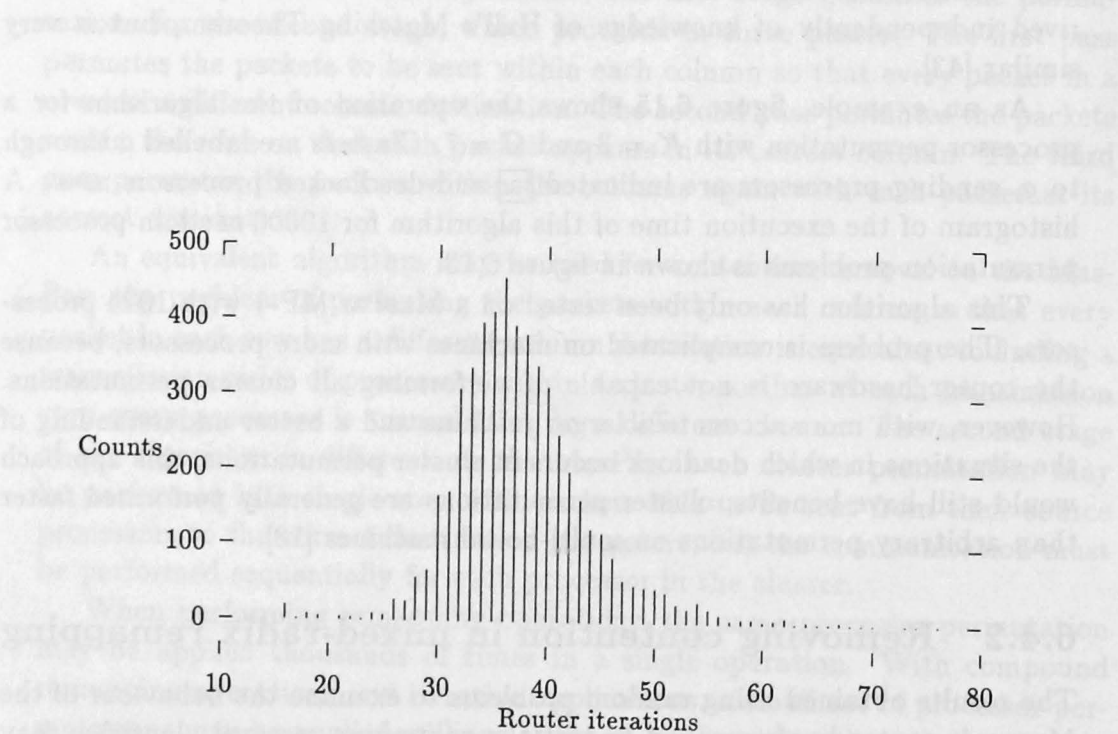


Figure 6.12: Number of router iterations required over 5216 digit swaps

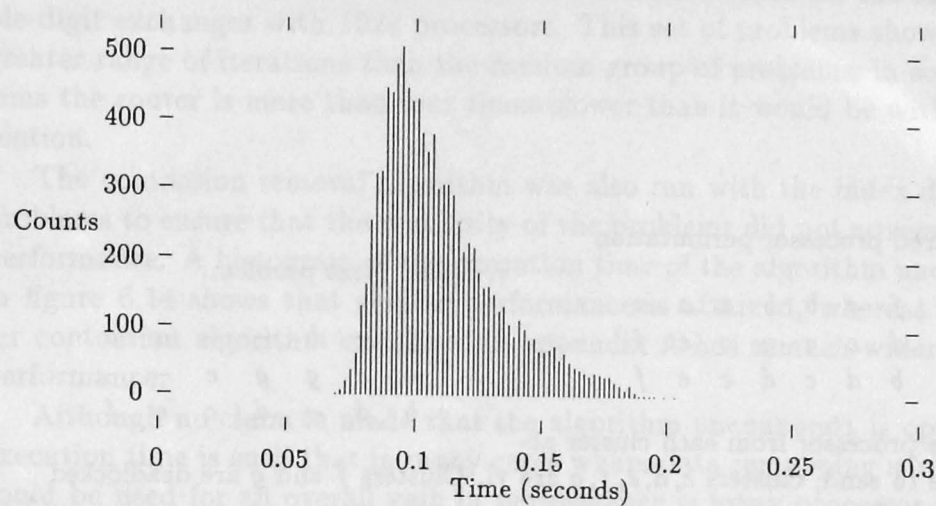


Figure 6.13: Execution time of contention removal algorithm `uncontend1` over 10000 random problems

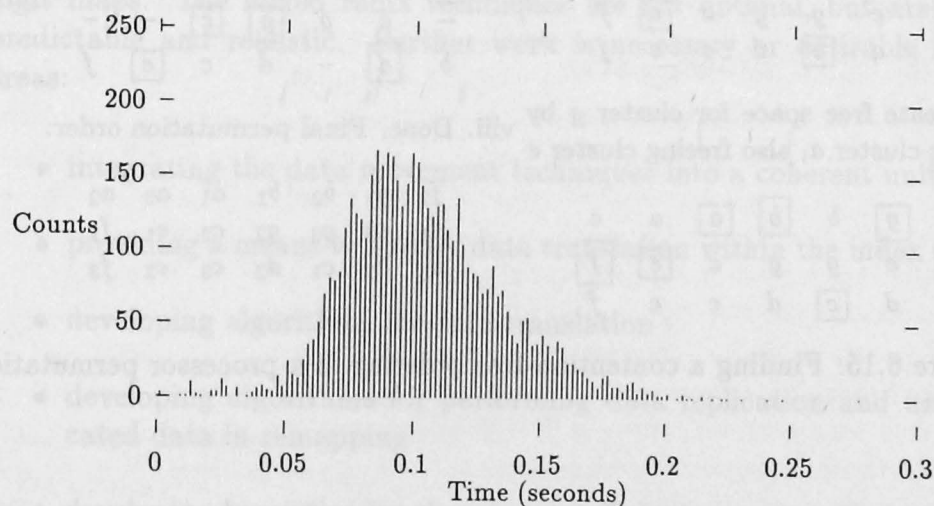


Figure 6.14: Execution time of contention removal algorithm `uncontend1` over all 5216 index digit swap problems

i. Desired processor permutation

<i>f</i>	<i>g</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>d</i>	<i>e</i>	<i>g</i>	<i>g</i>	<i>c</i>	<i>e</i>	<i>f</i>
<i>b</i>	<i>d</i>	<i>c</i>	<i>d</i>	<i>c</i>	<i>e</i>	<i>f</i>

v. Done. Next problem:

<i>f</i>	-	<i>b</i>	-	-	<i>a</i>	<i>a</i>
-	<i>e</i>	<i>g</i>	<i>g</i>	<i>c</i>	-	-
<i>b</i>	<i>d</i>	-	<i>d</i>	<i>c</i>	<i>e</i>	<i>f</i>

ii. One processor from each cluster attempts to send; clusters *c, d, e, f, g* are deadlocked

<i>f</i>	<i>g</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>d</i>	<i>e</i>	<i>g</i>	<i>g</i>	<i>c</i>	<i>e</i>	<i>f</i>
<i>b</i>	<i>d</i>	<i>c</i>	<i>d</i>	<i>c</i>	<i>e</i>	<i>f</i>

<i>f</i>	-	<i>b</i>	-	-	<i>a</i>	<i>a</i>
-	<i>e</i>	<i>g</i>	<i>g</i>	<i>c</i>	-	-
<i>b</i>	<i>d</i>	-	<i>d</i>	<i>c</i>	<i>e</i>	<i>f</i>

iii. Un-deadlock clusters *c* and *f*, also freeing cluster *d*

vii. Create free space for cluster *f* by moving cluster *b*, also freeing cluster *g*

<i>f</i>	<i>g</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>d</i>	<i>e</i>	<i>g</i>	<i>g</i>	<i>c</i>	<i>e</i>	<i>f</i>
<i>b</i>	<i>d</i>	<i>c</i>	<i>d</i>	<i>c</i>	<i>e</i>	<i>f</i>

<i>f</i>	-	<i>b</i>	-	-	<i>a</i>	<i>a</i>
-	<i>e</i>	<i>g</i>	<i>g</i>	<i>c</i>	-	-
<i>b</i>	<i>d</i>	-	<i>d</i>	<i>c</i>	<i>e</i>	<i>f</i>

iv. Create free space for cluster *g* by moving cluster *a*, also freeing cluster *e*

viii. Done. Final permutation order:

<i>f</i>	<i>g</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>d</i>	<i>e</i>	<i>g</i>	<i>g</i>	<i>c</i>	<i>e</i>	<i>f</i>
<i>b</i>	<i>d</i>	<i>c</i>	<i>d</i>	<i>c</i>	<i>e</i>	<i>f</i>

<i>f</i> ₂	<i>g</i> ₁	<i>b</i> ₂	<i>b</i> ₁	<i>a</i> ₁	<i>a</i> ₃	<i>a</i> ₂
<i>d</i> ₁	<i>e</i> ₃	<i>g</i> ₃	<i>g</i> ₂	<i>c</i> ₂	<i>e</i> ₁	<i>f</i> ₁
<i>b</i> ₃	<i>d</i> ₂	<i>c</i> ₁	<i>d</i> ₃	<i>c</i> ₃	<i>e</i> ₂	<i>f</i> ₃

Figure 6.15: Finding a contention-free ordering of a processor permutation

re-signification tests, an identity digit was included to ensure that as many processors as possible were communicating.

Figure 6.12 shows the number of router iterations used for the 5216 possible digit exchanges with 1024 processors. This set of problems shows a much greater range of iterations than the random group of problems; in some problems the router is more than four times slower than it would be without contention.

The contention removal algorithm was also run with the index digit swap problems to ensure that the regularity of the problems did not adversely affect performance. A histogram of the execution time of the algorithm `uncontend1` in figure 6.14 shows that similar performance is attained, whereas the cluster contention algorithm examined in appendix A has a much wider range of performance.

Although no claim is made that the algorithm `uncontend1` is optimal, its execution time is such that in many cases where data remapping is required it could be used for an overall gain in performance in many processor permutations were required, or for a speed-up in a core program section in cases where animation or interactive rates are desirable.

6.5 A system for mixed radix remapping

Using the components we have introduced in this chapter, it would be possible to implement a complete system for inter-converting between arbitrary index digit maps. The mixed radix techniques are not optimal, but are generally predictable and realistic. Further work is necessary or desirable in several areas:

- integrating the data movement techniques into a coherent unit
- providing a means to specify data translation within the index digit map
- developing algorithms for data translation
- developing algorithms for performing data replication and using replicated data in remapping
- developing heuristics for choosing an efficient sequence of operations for performing any given remapping
- integrating index digit maps with the k -Tile format
- integrating radix 2 techniques where appropriate

- analysing the brute-force approaches in more depth, using both empirical and mathematical techniques, to extract further performance improvements
- using more sophisticated analysis to determine non-brute-force algorithms for re-signification problems
- examining the cluster contention removal problem with machines with more than 1024 processors
- optimizing any resulting source code to obtain further speed and size improvements

6.6 Summary

In this chapter we go some way beyond the work of chapters 4 and 5, which examine data mapping on arrays and devices where the lengths of the dimensions are powers of two. We introduce the index digit map, which is a generalization of the index bit map.

The most important difference between the index bit map and the index digit map is that the index digit map may contain index digits of any mixed radices, thus allowing any length of array dimensions to be represented. A consequence of this is that by factorizing the length of the data array dimensions in different ways, many different ways exist of representing the same mapping.

Two other additions to the ideas in the index bit map provide further flexibility: by specifying *compound* digits, empty space may be included around digits; by the use of a **-digit*, data may be replicated.

The problem of remapping data between index digit maps is not straightforward. If two index digit maps are *aligned*, the remapping may be performed by re-factorizing their mixed-radix number representations and performing index digit permutations, for which a straightforward algorithm using similar techniques to the radix 2 case may be used.

However, if the two index digit maps are non-aligned, the data movement can be more complex. Some non-aligned remappings may be performed using a series of index-digit *re-significations* within the device dimensions. However, unless a more clever technique is found, remappings that are not amenable to these approaches must be remapped using a “brute-force” approach, where all processors attempt to send their data to their destination processors. Problems with this approach include as expensive computation, router network contention and data queues filling up, but we have found ways for most remappings to be performed with linear performance and moderate memory requirements.

One other aspect of mixed radix remapping on the MasPar is the problem of contention for router links when performing arbitrary processor permutations.

It is possible on a 1024 PE MasPar to perform every processor permutation in 16 router iterations, yet the hardware often takes many more. By choosing an appropriate transmission order, the minimum of 16 iterations can always be achieved. An algorithm for performing this task, based on *Halls Matching Theorem*, is described by Leighton [43]. We have implemented a similar algorithm which makes it possible to perform arbitrary router permutations a factor of up to five times faster than the unadvised hardware, although there is a set-up cost.

The techniques for mixed radix remapping have a very close relationship to the problem of performing arbitrary PMF remappings. However, there are problems remaining to be solved associated with these two systems, and we suggest many areas of possibly fruitful future study.

The previous chapters have examined the problems of specifying and implementation of data mapping and remapping in some detail. However, except for general statements of requirements, no concrete examples have yet been offered to justify the construction of a data mapping framework.

This chapter provides several concrete examples, showing how the A-File format and PMFs can be used in a variety of contexts.

7.1 High Performance Fortran

In section 2.4.3 it was suggested that the A-File format has a number of similarities to the data mapping directives of High Performance Fortran (HPF). In this section the capabilities of both the A-File format for mapping specification and PMFs for remapping are compared directly with the HPF regular data alignment and distribution directives to show that a complete PMF system could be used to implement the HPF directives. It is assumed that the reader is familiar with chapter 3 of the High Performance Fortran Language Specification [36], which in turn is based on Fortran 90 [4, 35] and on Fortran D [26]. Some examples of HPF are taken from the draft specification [36].

7.1.1 ALIGN and REALIGN directives

The ALIGN directive provides a way to specify a correspondence between the mappings of pairs of data objects (data arrays) so that values in separate data

While High Performance Fortran Language Specification [36] is ©1991 Rice University, Houston, Texas, and permission is given without fee or charge for this material to be printed, provided the Rice University copyright notice and the title of the document appear, and other is given that copying is by permission of Rice University.

The techniques for mixed radix remapping are very similar to those for the problem of performing arbitrary radix remapping. However, there are problems remaining to be solved associated with these two extensions, and we suggest many areas of possibly fruitful future study.

6.6 Summary

In this chapter we go some way toward the task of chapter 5, which is to design a radix remapping algorithm which is efficient and which is easy to implement. We now turn to the task of the index bit map.

The most important difference between the index bit map and the radix map is that the index bit map may contain any number of 1's, while the radix map contains only 0's and 1's. This is due to the fact that the index bit map is a binary vector, while the radix map is a binary vector of length n , where n is the number of bits in the radix.

Two other differences between the index bit map and the radix map are that the index bit map is a binary vector of length n , while the radix map is a binary vector of length n , where n is the number of bits in the radix.

The problem of performing arbitrary radix remapping is a difficult one, and it is not clear that there is a simple solution. However, the problem of performing mixed radix remapping is a much easier one, and it is possible to design an efficient algorithm for it.

However, it is not clear that there is a simple solution. However, the problem of performing mixed radix remapping is a much easier one, and it is possible to design an efficient algorithm for it. The problem of performing arbitrary radix remapping is a difficult one, and it is not clear that there is a simple solution. However, the problem of performing mixed radix remapping is a much easier one, and it is possible to design an efficient algorithm for it.

Chapter 7

The scope of data mapping operations

The previous chapters have examined the problems of specification and implementation of data mapping and remapping in some detail. However, except for general statements of requirements, no concrete examples have yet been offered to justify the construction of a data mapping framework.

This chapter provides several concrete examples, showing how the *k*-Tile format and PMFs can be used in a variety of contexts.

7.1 High Performance Fortran

In section 2.4.8 it was suggested that the *k*-Tile format has a number of similarities to the data mapping directives of High Performance Fortran (HPF). In this section the capabilities of both the *k*-Tile format for mapping specification and PMFs for remapping are compared directly with the HPF regular data alignment and distribution directives to show that a complete PMF system could be used to implement the HPF directives. It is assumed that the reader is familiar with chapter 3 of the High Performance Fortran Language Specification [36], which in turn is based on Fortran 90 [2, 53] and on Fortran D [26]. Some examples of HPF are taken from the draft specification [36]¹.

7.1.1 *ALIGN* and *REALIGN* directives

The *ALIGN* directive provides a way to specify a correspondence between the mappings of pairs of data objects (data arrays) so that values in separate data

¹The High Performance Fortran Language Specification is ©1992 Rice University, Houston Texas, and permission to copy without fee all or part of this material is granted, provided the Rice University copyright notice and the title of this document appear, and notice is given that copying is by permission of Rice University

arrays that are used in the same computations may be kept close together in the same or neighbouring processors.

The *REALIGN* directive is used to remap a data array dynamically to align it with another.

As well as allowing a direct correspondence between like elements in data arrays of the same shape, the *ALIGN* directive allows elements in different positions and elements in arrays of differing shapes to be aligned.

In terms of *k*-Tile mappings and PMFs, one data array, *B* (the *alignee*), may be aligned to another, *A*, by taking the *k*-Tile mapping of *A*, K_A , performing the appropriate alignment transformations on K as specified by the *ALIGN* directive, and using the transformed *k*-Tile mapping, K_B , to specify a mapping for *B*. If a complete PMF system were implemented on the target machine, the *k*-Tile mappings could be used to specify the data movement required for the *REALIGN* directive.

We will examine each of the mapping transformations that can be specified by the *ALIGN* directive, and show how they may be duplicated by *k*-Tile mapping transformations.

Simple *ALIGN* directives

The simplest form of the *ALIGN* directive allows two data arrays of the same shape to be assigned the same data mapping. For example, if a mapping has been specified for the two-dimensional data array *A*, another data array *B* of the same shape as *A* may be assigned an identical mapping by the declaration:

```
!HPF$ ALIGN B(:, :) WITH A(:, :)
```

The colons (:) indicate anonymous dummy variables assigned left-to-right; the same declaration could be written:

```
!HPF$ ALIGN B(i, j) WITH A(i, j)
```

The same semantics would be obtained with PMFs by using *A*'s *k*-Tile mapping for *B*:

<i>A k</i> -Tile		<i>B k</i> -Tile
a : [X; Y]		a : [X; Y]
k : [$K_{X_0}, \dots; K_{Y_0}, \dots$]	\Rightarrow	k : [$K_{X_0}, \dots; K_{Y_0}, \dots$]
m : [M_0, \dots]		m : [M_0, \dots]
d : [D_0, \dots]		d : [D_0, \dots]

Data dimension permutations

By permuting the dummy variables on the left-hand-side of the *WITH* in the *ALIGN* directive, the data dimensions of *B* array may be transposed with respect to *A*. The following example assigns *B* a mapping with the first two dimensions transposed with respect to *A*:

```
!HPF$ ALIGN B(i,j,k) WITH A(j,i,k)
```

The same semantics would be obtained with PMFs by taking the *k*-Tile mapping for *A*, transposing the data and *k*-Tile dimensions of *B*, and modifying the *k*-Tile mapping appropriately:

<i>A k</i> -Tile			<i>B k</i> -Tile	
<i>a</i>	: [<i>X</i> ; <i>Y</i> ; <i>Z</i>]		<i>a</i>	: [<i>Y</i> ; <i>X</i> ; <i>Z</i>]
<i>k</i>	: [<i>K_{X0}</i> , ..., <i>K_{Y0}</i> , ..., <i>K_{Z0}</i> , ...]	⇒	<i>k</i>	: [<i>K_{Y0}</i> , ..., <i>K_{X0}</i> , ..., <i>K_{Z0}</i> , ...]
<i>m</i>	: [<i>M₀</i> , ...]		<i>m</i>	: [<i>M₀</i> , ...]
<i>d</i>	: [<i>D₀</i> , ...]		<i>d</i>	: [<i>D₀</i> , ...]

For example, performing the same alignment with *A* representing a 2-dimensional hierarchical mapping of a $128 \times 256 \times 256$ volume yields the following mapping for *B*:

<i>A k</i> -Tile			<i>B k</i> -Tile	
<i>a</i>	: [128; 256; 256]		<i>a</i>	: [256; 128; 256]
<i>k</i>	: [4, 32; 8, 32; 256]	⇒	<i>k</i>	: [8, 32; 4, 32; 256]
<i>m</i>	: [0, 2, 4; 1; 3]		<i>m</i>	: [2, 0, 4; 3; 1]
<i>d</i>	: [8192; 32; 32]		<i>d</i>	: [8192; 32; 32]

Index arithmetic

By using arithmetic expressions involving index variables on the right-hand side of the *WITH* in the *ALIGN* directive, more complicated relationships may be declared between the aligned arrays. Three combinations of operators may be used to express these relationships: addition, multiplication and negation.

Index addition

A constant offset may be added to each of the dimensions of the alignee. The following example aligns position (0,0) of the array *B* with position (1,1) of array *A*:

```
!HPF$ ALIGN B(I,J) WITH A(I+1,J+1)
```

Note that there must be conditions on the dimensions of B relative to A to avoid indices being out of range. The same relationship could be expressed by PMFs by taking the k -Tile mapping for A , altering the data dimensions appropriately, supplying a data dimension offset, and using the transformed mapping for B :

$$\begin{array}{ll}
 \text{A } k\text{-Tile} & \text{B } k\text{-Tile} \\
 \begin{array}{l}
 \mathbf{a} : [R; C] \\
 \mathbf{k} : [K_{R_0}, \dots; K_{C_0}, \dots] \\
 \mathbf{m} : [M_0, \dots] \\
 \mathbf{d} : [D_0; \dots]
 \end{array} & \Rightarrow \begin{array}{l}
 \mathbf{a} : [R - 1; C - 1] \\
 \mathbf{t}_A : [R; C] \\
 \mathbf{o}_{T_A} : [1; 1] \\
 \mathbf{k} : [K_{R_0}, \dots; K_{C_0}, \dots] \\
 \mathbf{o}_K : [0; *] \\
 \mathbf{m} : [M_0, \dots] \\
 \mathbf{d} : [D_0; \dots]
 \end{array}
 \end{array}$$

Index negation

Indices may be negated to reverse the axes of an aligned array. The following example aligns the array B with A rotated by 180° :

```
!HPF$ ALIGN B(I,J) WITH A(R-I+1,C-J+1)
```

The same relationship could be expressed by PMFs by setting the k -Tile sense indicator:

$$\begin{array}{ll}
 \text{A } k\text{-Tile} & \text{B } k\text{-Tile} \\
 \begin{array}{l}
 \mathbf{a} : [R; C] \\
 \mathbf{k} : [K_{R_0}, \dots; K_{C_0}, \dots] \\
 \mathbf{m} : [M_0, \dots] \\
 \mathbf{d} : [D_0; \dots]
 \end{array} & \Rightarrow \begin{array}{l}
 \mathbf{a} : [R; C] \\
 \mathbf{k} : [K_{R_0}, \dots; K_{C_0}, \dots] \\
 \mathbf{s} : [-, -, \dots, -] \\
 \mathbf{m} : [M_0, \dots] \\
 \mathbf{d} : [D_0; \dots]
 \end{array}
 \end{array}$$

Index multiplication

Indices may be multiplied by a constant factor to leave storage gaps between elements. The following example aligns the elements of an array B , which is a quarter of the size of A , over the same storage as A :

```
!HPF$ ALIGN B(I,J) WITH A(I*2,J*2)
```

The same relationship could be expressed by PMFs by using empty k -Tile dimensions to spread the values of B :

$$\begin{array}{ll}
 \text{A } k\text{-Tile} & \text{B } k\text{-Tile} \\
 \mathbf{a} : [R; C] & \mathbf{a} : [R/2; C/2] \\
 \mathbf{k} : [K_{R_0}, \dots; K_{C_0}, \dots] & \Rightarrow \mathbf{k} : [K'_{R_0}, \dots; K'_{C_0} \dots; 2, 2] \\
 \mathbf{m} : [M_0, \dots] & \mathbf{m} : [M'_0, \dots] \\
 \mathbf{d} : [D_0; \dots] & \mathbf{d} : [D'_0; \dots]
 \end{array}$$

As a concrete example, performing the same alignment with A representing a scan-line mapping of a 256×256 image yields the following mapping for B :

$$\begin{array}{ll}
 \text{A } k\text{-Tile} & \text{B } k\text{-Tile} \\
 \mathbf{a} : [256; 256] & \mathbf{a} : [128; 128] \\
 \mathbf{k} : [256; 256] & \Rightarrow \mathbf{k} : [128; 128; 2, 2] \\
 \mathbf{m} : [0; 1] & \mathbf{m} : [2, 0; 3, 1] \\
 \mathbf{d} : [256; 256] & \mathbf{d} : [256; 256]
 \end{array}$$

Replicated arrays

Specifying '*' in an align-subscript indicates a replicated representation, in which a data array B may be aligned with every index in one or more dimensions of A . This is useful for maintaining a look-up table in every processor.

For example, the following alignment declares that a copy of B is to be kept with every column of A :

```
!HPF$ ALIGN B(:) WITH A(:,*)
```

The same relationship could be expressed with PMFs by taking the k -Tile mapping of A , converting the k -Tile dimensions corresponding to A 's column dimension to empty, replicating k -Tile dimensions and using the transformed mapping for B :

$$\begin{array}{ll}
 \text{A } k\text{-Tile} & \text{B } k\text{-Tile} \\
 \mathbf{a} : [R; C] & \mathbf{a} : [R] \\
 \mathbf{k} : [K_{R_0}, \dots; K_{C_0}, \dots] & \Rightarrow \mathbf{k} : [K_{R_0}, \dots; K_{C_0} \dots] \\
 \mathbf{m} : [M_0, \dots] & \mathbf{o}_K : [0; *] \\
 \mathbf{d} : [D_0; \dots] & \mathbf{m} : [M_0, \dots] \\
 & \mathbf{d} : [D_0; \dots]
 \end{array}$$

7.1.2 HPF PROCESSORS directive

The *PROCESSORS* directive declares rectilinear processor arrangements. It is equivalent to dimensions one and above in the k -Tile device space. Unlike HPF, the k -Tile format cannot represent processors with dimensionality above

that of the physical device, but these may be moved to memory by a specified distribution.

The *k*-Tile format *always* refers to the processors on a physical device with no concept of virtualization, and HPF refers to 'abstract' processors. However, it is permissible for an HPF compiler to reject processor arrangements with more processors than are available. In any case, the concept of virtual processors may not be a useful one for low-level implementation of algorithms [10].

7.1.3 Processor VIEWS

The HPF *VIEW* attribute allows a mapping to be defined between different arrangements of processors defined with the *PROCESSORS* directive by permutation of the processor dimensions. This may be duplicated in the *k*-Tile format by appropriate permutation of the device space and *k*-Tile dimensions.

7.1.4 *DISTRIBUTE* and *REDISTRIBUTE* directives

The *DISTRIBUTE* directive specifies a mapping of data objects to abstract processors in a processor arrangement. The *REDISTRIBUTE* directive specifies a dynamic remapping of data objects to another mapping.

It is possible to declare for each dimension in the array appearing in the *DISTRIBUTE* directive the mapping of that dimension as *BLOCK*, *CYCLIC* or *. *BLOCK* distributes data in a dimension as 1d hierarchical, *CYCLIC* distributes the data as 1d cut'n'stack and * stores the dimension in memory.

For example, the following HPF fragment declares a processor array and declares a three-dimensional array *A* whose dimensions are mapped *BLOCK*, *CYCLIC* and *:

```

      INT A(256,256,256)
      !HPF$ PROCESSORS P(32,32)
      !HPF$ DISTRIBUTE A(BLOCK, CYCLIC, *) ONTO P

```

The same mapping may be expressed with PMFs:

```

a  : [256;256;256]
k  : [8,32;32,8;256]
m  : [0,3,4;1;2]
d  : [16384;32;32]

```

7.1.5 *TEMPLATE* directive

The *TEMPLATE* directive is useful for declaring distributions and alignments with respect to an object which does not exist but has a well-defined mapping.

If PMFs were used as the basis for the other mapping directives, the template directive could easily be incorporated.

7.1.6 PMFs \supset HPF

By giving examples of the use of the HPF mapping directives, we have shown that the k -Tile format is capable of representing all the regular mapping relationships that can be specified within by HPF. There are, moreover, many mappings that can be specified by PMFs that cannot be represented within HPF.

Memory usage

HPF does not allow the programmer to specify the arrangement of array dimensions in memory. This may cause some problems in generating optimized code, as index address calculations can involve expensive multiplications that can sometimes be avoided. The lack of specification also makes it difficult to export the array from the HPF program: when calling a routine from another language, the arrangement of data is implementation-dependent and hence non-portable; when performing input/output, it is not possible to define the mapping to arrange memory to match the format of a file on disk, or some other device.

Hierarchical tiles

By use of the *BLOCK* and *CYCLIC* directives, an array dimension is only split into two sub-dimensions. With the k -Tile format, dimensions may be split into as many sub-dimensions as there are prime factors in that dimension. Extreme but useful examples include a bit-reversed mapping for the FFT, or pyramidal storage order in which dimension bits are interleaved.

7.2 KIPS

The CSIRO Division of Information Technology progressively developed a library for image processing, called the Kernel Image Processing Software (or KIPS) [7, 55, 66]. The aim of this system is to make many common tasks associated with image processing easy to perform, such as storage, display and computation on external devices. The storage model for KIPS will be based around the multidimensional tile format [28, 29]. Because the tile format is a subset of the k -Tile format, a PMF system will provide much of the required specification and code for implementing the KIPS storage model. As the computational framework for KIPS is expanded, PMFs will be ideal for bridging

the gap between specification of storage formats for sequential devices and the specification of mappings for parallel devices.

This thesis has addressed the algorithms associated with remapping data on a parallel computer similar to the MasPar MP-1. However, there are many other problems associated with the k -Tile format which will need to be addressed for a full implementation of KIPS. One such problem, the accessing of rectilinear blocks from an image stored according to a k -Tile format, has been solved by Fraser [29]. The problems of remapping data stored on sequential devices and moving blocks of data between devices while altering their k -Tile mappings must still be addressed.

7.3 Sample applications

The following examples show the benefits of remapping using PMFs for performing various internal data movement operations [12, 22, 72].

7.3.1 Scan-line algorithms

A significant cost in parallel computation is interprocessor communication, because providing a fast connection network capable of providing a large class of processor permutations is very expensive. A compromise is usually found between expense, flexibility and speed: mesh networks provide fast communication between neighbouring processors, but are inflexible and slow for irregular or long-distance communication; crossbar switches provide more flexibility, but generally provide a low bandwidth because of the high cost. Many other routing schemes have been created, but all are slow compared to intra-processor communication.

In order to minimize communication costs, *scan-line algorithms* perform their computation on local neighbourhoods of an image, such as a scan-line (i.e. scan-row or scan-column), with inter-processor communication confined to a set of regular remappings which are designed to use the communication network as efficiently as possible.

The resulting algorithms are designed to keep all processors busy at close to full capacity and, because of their simple structure, may easily be optimized. This simplicity also makes their performance predictable and consistent.

7.3.2 2d rotation

Rotation of a two dimensional image may be converted into a scan-line operation by separating the rotation matrix into 'skew' operations which are performed only along either the x-axis or the y-axis. Two formulations have been used; Paeth [58] uses three skew operations with no scaling, and both

Fraser and O'Brien [31] and Catmull and Smith [9] use two skew operations with scaling.

Firstly we will present Paeth's method. Because of storage requirements and a singularity at 180° , θ should only range from -45° to 45° :

$$\begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} = \begin{pmatrix} 1 & \frac{\cos \theta - 1}{\sin \theta} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\sin \theta & 1 \end{pmatrix} \begin{pmatrix} 1 & \frac{\cos \theta - 1}{\sin \theta} \\ 0 & 1 \end{pmatrix}$$

By performing transpositions between each of the three skew operations, the complete rotation may be performed with three 'skew' operations and two array transpositions:

$$\begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} = \begin{pmatrix} 1 & \frac{\cos \theta - 1}{\sin \theta} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & -\sin \theta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & \frac{\cos \theta - 1}{\sin \theta} \\ 0 & 1 \end{pmatrix}$$

For angles other than $-45^\circ < \theta < 45^\circ$, the rotation may be performed by a pre- or post-rotation by some multiple of 90° . This example PMF remapping describes a scan-line transposition for a 1024×1024 image on a 1024-PE machine:

$$\begin{array}{ll} \mathbf{a} : [1024; 1024] & \mathbf{a} : [1024; 1024] \\ \mathbf{k} : [1024; 1024] & \mathbf{k} : [1024; 1024] \\ \mathbf{m} : [0; 1] & \mathbf{m} : [1; 0] \\ \mathbf{d} : [1024; 1024] & \mathbf{d} : [1024; 1024] \end{array} \Rightarrow$$

Similarly, by the use of the sense indicator a 90° rotation may be specified:

$$\begin{array}{ll} \mathbf{a} : [1024; 1024] & \mathbf{a} : [1024; 1024] \\ \mathbf{k} : [1024; 1024] & \mathbf{k} : [1024; 1024] \\ \mathbf{m} : [0; 1] & \mathbf{s} : [-, +] \\ \mathbf{d} : [1024; 1024] & \mathbf{m} : [1; 0] \\ & \mathbf{d} : [1024; 1024] \end{array} \Rightarrow$$

Figure 7.1 shows the operation of Paeth's algorithm on a 600×600 test image, *aspens*. Linear interpolation was used in the skewing operations.

An alternative separation of the rotation matrix is proposed both by Fraser, and by Catmull and Smith:

$$\begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} = \begin{pmatrix} \cos \theta + \frac{\sin^2 \theta}{\cos \theta} & \tan \theta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\sin \theta & \cos \theta \end{pmatrix}$$

As with Paeth's algorithm, this may be performed with skews only along the x -axis by performing transpositions between the stages:

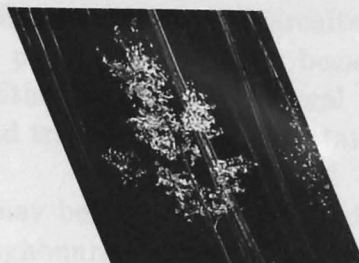
$$\begin{pmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{pmatrix} = \begin{pmatrix} \cos \theta + \frac{\sin^2 \theta}{\cos \theta} & \tan \theta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} \cos \theta & -\sin \theta \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

7.3.3 Perspective viewing

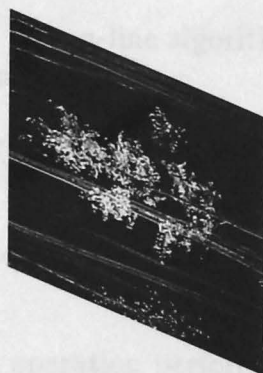
Viewing a height field in perspective may also be achieved using scanning operations, as described in [51]. The scan-line algorithm for perspective viewing is more complex than the scan-line algorithm for orthographic viewing. The scan-line algorithm for perspective viewing is more complex than the scan-line algorithm for orthographic viewing. The scan-line algorithm for perspective viewing is more complex than the scan-line algorithm for orthographic viewing.



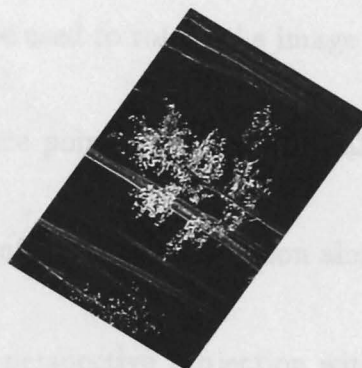
(a) Original image



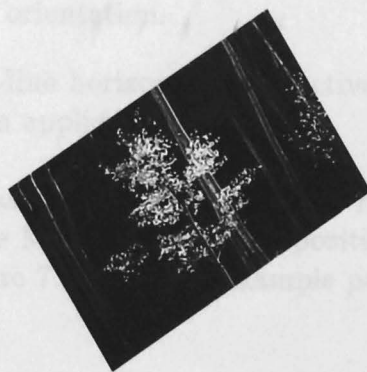
(b) First skew



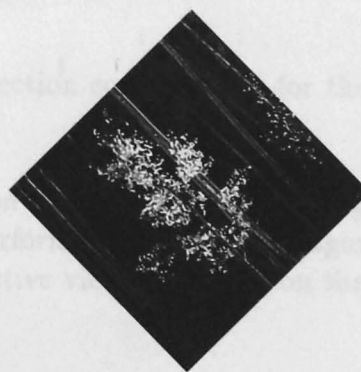
(c) Transposed



(d) Second skew



(e) Transposed



(f) Final skew

7.3.4 2d scan-line calibration

The scan-line approach to many image processing tasks is ideal if the number of scan-lines to be processed equals the number of processors. If the number of scan-lines is less than the number of processors, the processors must be idle. If the number of scan-lines is greater than the number of processors, the processors must be idle. If the number of scan-lines is less than the number of processors, the processors must be idle. If the number of scan-lines is greater than the number of processors, the processors must be idle.

Figure 7.1: Skewing and transposition operations used to rotate *aspens* 45°

7.3.3 Perspective viewing

Viewing a height field in perspective may also be performed using scan-line operations, as explained by Robertson [61]. Even on sequential architectures the scan-line approach is a fast solution for perspective viewing because of its regularity. Robertson's perspective algorithm proceeds in several passes alternating between scan-line skews/scales and transpositions/90° rotations:

- i. If the image has not been textured, it may be textured by a shading or shadowing algorithm. Shading is a neighbourhood operation (see section 7.3.7), and shadowing is performed in a similar way to the perspective algorithm [62]. Alternatively, the elevation image may be used alone for a fast simple texture. The pixels of the texture image remain with the corresponding pixels in the elevation image.
- ii. If required, a scan-line algorithm may be used to rotate the image to the desired frontal view.
- iii. A scan-line squeeze aligns all the surface points that can mutually occlude.
- iv. Transposition or rotation by a multiple of 90° allows operation along the columns.
- v. A scan-line operation performs vertical perspective projection with hidden surface removal by overwriting pixels from back to front.
- vi. Transposition or rotation by a multiple of 90° returns the image to its correct orientation.
- vii. A scan-line horizontal perspective projection compensates for the compression applied in stage (iii)

Vézina has implemented scan-line rotation and perspective viewing algorithms on the MasPar, with transposition performed by the PMF algorithms [72, 73]. Figure 7.2 shows an example perspective view generated on the MasPar MP-1.

7.3.4 2d scan-line virtualization

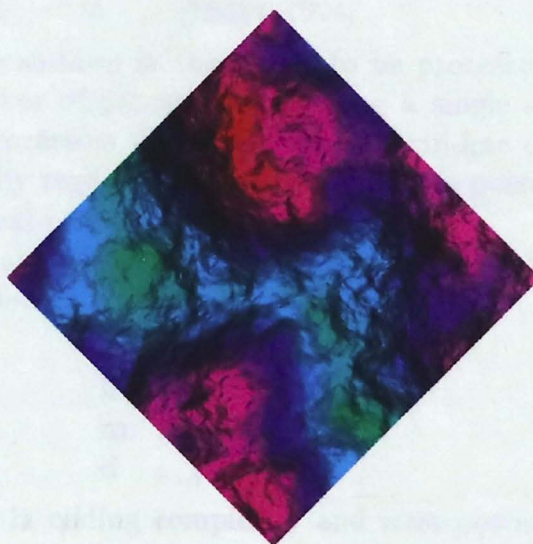
The scan-line approach to many image processing tasks is ideal if the number of scan-lines to be processed equals the number of processors. If the number of scan-lines is greater than the number of processors, several scan-lines may be distributed to each processor and processed sequentially. If the time to process each scan-line is constant and some inter-scan-line communication is required,



(a) Fractal height data



(b) Shaded



(c) Shaded and rotated



(d) Perspective view

Figure 7.2: Four stages in the generation of a perspective view of fractal height data, generated and processed on the MasPar MP-1

a hierarchical mapping of scan-lines to processors would be appropriate. If the time to process each scan-line is spatially dependent, a cut'n'stack mapping will give each processor a widely-spaced selection of scan-lines from the image.

These two *k*-Tile formats describe a hierarchical mapping of a 4096×4096 image to a 1024-PE processor,

```
a : [4096; 4096]
k : [4096; 4, 1024]
m : [0, 1; 2]
d : [16384; 1024]
```

and a cut'n'stack mapping of the same image:

```
a : [4096; 4096]
k : [4096; 1024, 4]
m : [0, 2; 1]
d : [16384; 1024]
```

If the number of scan-lines in the image to be processed is significantly smaller than the number of processors, mapping a single scan-line to each processor will leave processors underutilized. In scan-line operations where the data access is highly regular, such as rotation, it is possible to split each scan-line between several processors.

This *k*-Tile format shows a 512×512 image mapped with half a scan-line per PE in a 1024-PE device:

```
a : [512; 512]
k : [256, 2; 512]
m : [0; 1, 2]
d : [256; 1024]
```

The resultant increase in coding complexity and inter-processor communication requirements may negate gains due to load redistribution. In scan-line operations where the data access within scan-lines is both non-regular and non-local, such as vertical perspective projection, splitting scan-lines may offer no advantages.

Another approach to making better use of available processors for images with a small number of scan-lines is to generate several images simultaneously. This method can increase interaction speeds and allow animation sequences to be generated more quickly.

This *k*-Tile format declares space for 32 512×512 images on a 16384-PE device, with the third image dimension selecting the image number and each image stored in a block of 512 processors:

```
a : [512; 512; 32]
k : [512; 512; 32]
m : [0; 1, 2]
d : [512; 16384]
```


7.3.5 Volume rotation and rendering

Three-dimensional scan-line algorithms can be used for the visualization of three-dimensional volume data. Three dimensional rotations can be expressed as an alternating sequence of skews through the processors' memory axis and transpositions/rotations by multiples of 90° [33, 44, 71].

Vézina has implemented a volume visualization system for the MasPar that uses scan-line algorithms for performing both rotation [33] and for perspective and iso-surface calculations [71], and uses radix 2 PMFs for inter-processor data movement operations. Figure 7.3 shows output from Vézina's system using two data sets, an MRI image of a human head ² and a three-dimensional phase space diagram of a strange attractor [3].

Three dimensional rotations require many more remappings than those required for two. For two dimensional rotation only eight states are obtained from combinations of 90° rotations/transpositions: four rotations \times two transpositions. For three dimensions 48 states are obtained from these operations: one of six planes can face forward, each of which may be in four orientations, and may also be reflected by transposition, yielding $6 \times 4 \times 2 = 48$ states. In a later implementation of Vézina's rendering system [71], operations consisting of multiple transpositions and rotations by multiples of 90° are concatenated. This requires only one remapping operation to be performed between non-remapping processing steps, and allows the data movement to be optimized; inverse operations appearing in two compound operations need not be performed.

Except for the smallest images to be processed, the larger quantity of data processed in volume visualization requires some form of scan-line virtualization to be used. There are many more choices for virtualization of three dimensional images than for two because any 2d mapping may be used for any two of the volume dimensions and the third dimension may be mapped in several ways also. Vézina chose to map the viewing coordinate system with Z -scan-lines along the processors' memories with the X and Y axes in a 2d hierarchical mapping. This allows for fast access along the Z -axis of the data for scan-line operations and keeps many pixel neighbourhoods in memory for fast gradient calculations.

This example shows the k -Tile mapping for a $256 \times 256 \times 256$ image on a 1024-PE MasPar:

a	:	[256; 256; 256]
k	:	[8, 32; 8, 32; 256]
m	:	[4, 0, 2; 1; 3]
d	:	[16384; 32; 32]

Because the processor array in an 8192-PE MasPar is rectangular, rectangular

²Data taken on the Siemens Magnetom and provided courtesy of Siemens Medical Systems, Inc., Iselin, NJ., and obtained from the University of North Carolina

tiles must be stored to represent a cubical volume, as this example shows:

a : [256; 256; 256]
 k : [2, 128; 4, 64; 256]
 m : [4, 0, 2; 1; 3]
 d : [2048; 128; 64]

The other k -Tile mappings are defined similarly.

Figures 7.4 and 7.5 shows the execution time required for performing the 48 volume remappings for different image sizes on processor arrays of different size. Transpositions are *around* the indicated axis, e.g. X means 'transpose Y and Z axes', whereas reverses indicate the actual axis reversed. An XY transposition is an X transposition followed by a Y transposition. An XY transposition is equivalent to a YZ or ZX transposition, and an XZ transposition is equivalent to a ZY or YX transposition.

The timings for the volume remapping operations are similar on two processor arrays of different size with the same number of data elements per processor, thus the time taken for the data movement for volume rendering is very nearly linearly related to the number of processors. For example, an XZ transposition and an XYZ reverse takes 597 mS on a $256 \times 256 \times 256$ volume on a 1024-PE machine, and the same problem on a $512 \times 512 \times 512$ image on an 8192-PE machine takes 564 mS. Both these problems involve the movement of 16384 bytes in every processor.

Given a full crossbar switch connecting the processors and the algorithms presented in chapter 4, this would be expected. Strictly speaking the Mas-Par does not have a full crossbar switch (see section 5.6.4), yet equivalent performance is obtained because the implementation of PMFs prevents router contention from occurring.

Aside from manipulations on the volume data set, PMFs are also used when displaying the two-dimensional result of the volume rendering. Because of the mapping chosen for the three-dimensional volume, the two-dimensional rendering is generated in 2d hierarchical mapping. In order to transfer this image back to the front end workstation for display, a scan-line mapping is more convenient. If the image is to be displayed on a workstation with an 8-bit frame-buffer, direct colour quantization and dithering is performed [68], for which a scan-line mapping is also more convenient. This example shows the k -Tile mapping for the 2d hierarchical and 2d scan-line mapping of a $256 \times 256 \times 4$ generated colour volume image. A $256 \times 256 \times 3$ colour image would be more desirable, but remapping of this image could not be performed using radix 2 PMFs:

a	:	[4; 256; 256]		a	:	[4; 256; 256]
k	:	[4; 8, 32; 8, 32]	\Rightarrow	k	:	[4; 256; 256]
m	:	[0, 1, 3; 2, 4]		m	:	[0, 1; 2]
d	:	[256; 1024]		d	:	[1024; 256]



Figure 7.3: Example volume renderings of: (a, b, c): a human head and (d, e, f): a three-dimensional view of the phase space of a damped pendulum with a sinusoidal driving force.

Volume operation times on 1024 PE MasPar MP-1					
transposes	reverses	32 × 32 × 32 32 bytes/PE	64 × 64 × 64 256 bytes/PE	128 × 128 × 128 2048 bytes/PE	256 × 256 × 256 16384 bytes/PE
		Time (mS)	Time (mS)	Time (mS)	Time (mS)
-	-	1.0	1.0	1.0	1.0
-	X	1.6	5.7	36.9	284.3
-	Y	1.6	5.6	36.8	284.0
-	XY	1.6	5.6	36.3	282.0
-	Z	1.3	2.7	13.6	100.3
-	XZ	2.1	7.7	52.1	406.8
-	YZ	2.2	7.7	52.1	406.8
-	XYZ	2.2	7.7	52.1	455.6
X	-	2.0	7.5	39.2	288.7
X	X	4.9	11.1	77.1	545.4
X	Y	2.5	7.3	39.3	337.6
X	XY	5.3	11.0	76.0	595.4
X	Z	2.5	7.3	39.3	337.6
X	XZ	5.3	11.6	83.0	592.7
X	YZ	2.9	6.8	50.1	387.6
X	XYZ	4.9	11.7	82.6	544.5
Y	-	2.0	7.8	39.5	289.0
Y	X	2.5	7.3	39.2	338.0
Y	Y	4.9	11.1	73.1	545.8
Y	XY	5.3	11.0	75.4	595.9
Y	Z	2.5	7.3	39.2	338.0
Y	XZ	2.9	6.8	50.5	381.8
Y	YZ	5.3	11.6	80.6	593.0
Y	XYZ	4.9	11.7	82.1	546.3
Z	-	1.6	5.7	36.9	283.7
Z	X	1.6	5.6	36.4	282.1
Z	Y	1.6	5.6	36.4	282.1
Z	XY	1.6	5.7	36.9	283.8
Z	Z	2.2	7.7	52.1	455.6
Z	XZ	2.2	7.7	52.1	455.6
Z	YZ	2.2	7.7	52.1	455.6
Z	XYZ	2.2	7.7	52.1	455.6
XY	-	4.9	11.0	74.9	546.9
XY	X	5.3	11.0	76.4	595.6
XY	Y	5.3	11.0	76.1	600.0
XY	XY	4.9	11.1	76.2	546.9
XY	Z	5.3	11.6	82.3	595.3
XY	XZ	4.9	11.7	83.0	542.4
XY	YZ	4.9	11.7	84.1	542.0
XY	XYZ	5.3	11.8	82.0	596.5
XZ	-	4.9	11.0	74.9	546.6
XZ	X	5.3	11.0	76.4	600.5
XZ	Y	5.3	11.0	76.2	596.3
XZ	XY	4.9	11.1	76.2	546.6
XZ	Z	5.3	11.6	82.3	595.6
XZ	XZ	4.9	11.7	83.0	542.4
XZ	YZ	4.9	11.7	84.1	542.0
XZ	XYZ	5.3	11.8	82.0	597.1

Figure 7.4: Execution times in milliseconds of all 48 volume remapping operations on a 1024 PE MasPar MP-1.

Volume operation times on 8192 PE MasPar MP-1				
transposes	reverses	128 × 128 × 128 256 bytes/PE	256 × 256 × 256 2048 bytes/PE	512 × 512 × 512 16384 bytes/PE
		Time (mS)	Time (mS)	Time (mS)
-	-	1.1	1.1	1.1
-	X	5.6	37.6	287.0
-	Y	5.7	36.8	284.1
-	XY	5.7	36.6	282.4
-	Z	3.0	15.8	117.4
-	XZ	8.1	54.3	424.0
-	YZ	8.0	54.3	424.0
-	XYZ	8.0	54.3	424.0
X	-	9.1	51.6	389.0
X	X	14.5	79.4	562.6
X	Y	8.9	52.0	387.7
X	XY	20.7	78.1	561.5
X	Z	8.9	52.0	387.7
X	XZ	15.3	87.2	570.5
X	YZ	8.7	55.4	488.4
X	XYZ	22.2	86.7	561.3
Y	-	13.3	86.8	580.1
Y	X	15.4	84.4	582.0
Y	Y	22.6	79.3	584.0
Y	XY	21.9	78.4	604.4
Y	Z	15.4	84.4	582.0
Y	XZ	13.4	76.7	663.2
Y	YZ	21.9	84.9	659.5
Y	XYZ	22.6	86.0	671.5
Z	-	8.3	59.5	460.1
Z	X	9.4	64.8	506.2
Z	Y	9.4	64.8	506.2
Z	XY	8.3	59.5	460.1
Z	Z	10.8	75.8	595.9
Z	XZ	10.6	74.5	585.7
Z	YZ	10.6	74.5	585.7
Z	XYZ	10.8	75.8	595.9
XY	-	19.9	79.6	565.2
XY	X	19.7	79.9	561.1
XY	Y	19.7	79.6	564.1
XY	XY	19.9	80.6	564.0
XY	Z	19.7	86.5	561.1
XY	XZ	19.9	87.5	565.3
XY	YZ	19.9	88.8	564.0
XY	XYZ	19.7	86.2	564.1
XZ	-	19.9	76.2	565.2
XZ	X	19.7	78.4	561.1
XZ	Y	19.7	75.1	561.1
XZ	XY	19.9	78.8	565.3
XZ	Z	19.7	88.1	564.1
XZ	XZ	19.9	87.8	564.3
XZ	YZ	19.9	89.1	564.3
XZ	XYZ	19.7	86.2	564.1

Figure 7.5: Execution times in milliseconds of all 48 volume remapping operations on an 8192 PE MasPar MP-1.

7.3.6 The Fast Fourier Transform

The Fourier Transform of a two-dimensional image may be viewed as a scan-line algorithm, as it may be performed by transforming the rows, transposing, and transforming the columns [8, 11, 32]. However, there are several other aspects of the FFT which allow it to benefit by the use of PMFs for other components of the algorithm.

For the radix 2 FFT [11], the data array must be a power of two in length and is re-ordered so that each data element is moved to another position whose index is the bit-reversal of its own, whether before or after the FFT computation. This reordering may be specified by the k -Tile format; this example shows the PMF transformations that could be used for a 1024×1024 FFT. A complex number is assumed to consist of eight bytes:

- Bit-reversal prior to row FFT:

$$\begin{array}{ll} \mathbf{a} : [8; 1024; 1024] & \mathbf{a} : [8; 1024; 1024] \\ \mathbf{k} : [8; 1024; 1024] & \Rightarrow \mathbf{k} : [8; 2, 2, 2, 2, 2, 2, 2, 2; 1024] \\ \mathbf{m} : [0, 1; 2] & \mathbf{m} : [0, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1; 11] \\ \mathbf{d} : [8192; 1024] & \mathbf{d} : [8192; 1024] \end{array}$$

$$\begin{array}{c} (m_0)(m_1)(m_2) \\ \text{Radix 2 remapping: } (m_3, m_{12})(m_4, m_{11})(m_5, m_{10})(m_6, m_9)(m_7, m_8) \\ (p_0)(p_1)(p_2)(p_3)(p_4)(p_5)(p_6)(p_7)(p_8)(p_9) \end{array}$$

- Transposition and bit-reversal prior to column FFT:

$$\begin{array}{ll} \mathbf{a} : [8; 1024; 1024] & \mathbf{a} : [8; 1024; 1024] \\ \mathbf{k} : [8; 1024; 1024] & \Rightarrow \mathbf{k} : [8; 1024; 2, 2, 2, 2, 2, 2, 2, 2; 2] \\ \mathbf{m} : [0, 1; 2] & \mathbf{m} : [0, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2; 1] \\ \mathbf{d} : [8192; 1024] & \mathbf{d} : [8192; 1024] \end{array}$$

$$\begin{array}{c} (m_0)(m_1)(m_2) \\ \text{Radix 2 remapping: } (m_3, p_0, m_{12}, p_9)(m_4, p_1, m_{11}, p_8) \\ (m_5, p_2, m_{10}, p_7)(m_6, p_3, m_9, p_6) \\ (m_7, p_4, m_8, p_5) \end{array}$$

- Transposition and offset of origin to center for viewing:

$$\begin{array}{ll} \mathbf{a} : [8; 1024; 1024] & \mathbf{a} : [8; 1024; 1024] \\ \mathbf{k} : [8; 1024; 1024] & \Rightarrow \mathbf{o}_A : [0, 512, 512] \\ \mathbf{m} : [0, 1; 2] & \mathbf{k} : [8; 1024; 1024] \\ \mathbf{d} : [8192; 1024] & \mathbf{m} : [0, 2; 1] \\ & \mathbf{d} : [8192; 1024] \end{array}$$

Variations of the FFT have been developed for arrays with dimensions which are not powers of two and require mixed-radix index digit permutation. The vector-radix fast Fourier transform [15, 35] performs multi-dimensional FFTs by computing and combining multiple smaller FFTs of the same dimensionality as the desired FFT.

The data reordering required for both these forms of the FFT may be specified using PMFs. The fractal surface shown in figure 7.2 was generated by performing an inverse Fourier transform on noise with a $1/f^\beta$ distribution, as described by Voss, with a fractal dimension of 2.4 specified [74]. A scan-line mapping was used with the index-bit reversals and transposition performed by the radix 2 implementation of PMFs.

Fier has examined many implementations of the radix 2 FFT and has found that "the most efficient implementation consists of an alternating sequence of ... sequential FFTs followed by data redistribution" [19]. The data distribution required for his examples may also be specified by PMFs. Kuszmaul has also examined FFTs for the MasPar, and PMFs could be used for many of the data mappings he proposes. However, he also specifies an *embedded cube mapping*, which cannot be described by PMFs [40, 41].

7.3.7 Neighbourhood operations

A large class of image processing operations can be expressed as neighbourhood operations, in which every pixel is replaced by some function of its closest neighbours. Some examples of neighbourhood operations are filtering by kernel convolution, mathematical morphology, simple edge detection, gradient determination and shading. In this section we examine the number of communication operations required for performing neighbourhood operations on data stored in several data mappings. We will only examine 2d neighbourhood operations, but the same analysis could be carried out on higher-dimensional images with similar conclusions.

We examine a lower bound on communications cost for a 3×3 neighbourhood operation on an $N \times N$ image on a square $P \times P$ -element processor array, where N is divisible by P , for each of three mappings: 1d hierarchical, 2d hierarchical and 2d cut'n'stack. If $P^2 > N$, a partial scan-line from the image will be stored in each PE in the 1d hierarchical mapping, and nearest-neighbour communication is not sufficient for finding neighbours in the 1d hierarchical mapping.

In each of these mappings each processor will contain N^2/P^2 elements. We assume that the neighbourhood algorithm is intelligent enough to read each data element only once, thus the number of memory and computation cycles will be the same for each of the three mappings. We ignore image edge effects, which could be handled by element duplication or substitution of a padding value. We express the number of communication steps in terms of a

'2d' virtualization ratio, $V = N/P$ (the true virtualization ratio is N^2/P^2).

The algorithm using the 1d hierarchical mapping will need to communicate with each processor's neighbours to obtain the pixels from the scan-lines above the first and below the last in its memory, and to the left and right of a partial scan-line if $N < P$. The algorithm using the 2d hierarchical mapping needs communication to obtain the pixels on the four sides of each 2d tile. The algorithm using the cut'n'stack mapping needs communication to obtain the neighbours of every pixel:

- 1d hierarchical: $C = \begin{cases} 2 \times N = 2 \times P \times V & \text{if } N \geq P^2 \\ 6 + 2 \times N^2/P^2 = 6 + V^2 & \text{if } N < P^2 \end{cases}$
- 2d hierarchical: $C = 4 \times (1 + N/P) = 4 \times (1 + V)$
- 2d cut'n'stack: $C = 9 \times N^2/P^2 = 9 \times V^2$

Clearly a 2d hierarchical mapping requires the fewest communication steps when there are at least four processors.

Unfortunately, an algorithm using a 2d hierarchical mapping to perform a 3×3 neighbourhood operation which combines communication with computation operations is complex, as there are nine distinct types of communication associated with pixels in the tile, corresponding to the four corners, the four sides and the interior. If the algorithm is designed to cope with small virtualization ratios, there is one more type of communication associated with a virtualization ratio of 1. Thus, either the code for computation must be duplicated ten times or the containing loop must test for the distinct cases for an efficient implementation of a 3×3 neighbourhood operation. Operations on larger neighbourhoods contain many more distinct communications types.

An alternative method for implementing this type of operation is to separate the communication and computation components by allowing a region of padding around each tile. Before computation begins, the pixels neighbouring each tile are copied into the padding area. The computation may now be performed in exactly the same manner for each pixel.

This technique causes an overhead in memory use, and further investigation is necessary to determine if any time saving may be achieved. However, it may provide gains through the combining of all communications operations and simplification of the computational loop.

Although no algorithms have yet been developed for performing this type of PMF operation, it may be specified using k -Tile template offsets. This example shows how a 1024×1024 image on a 32×32 mesh may be padded to allow a 3×3 neighbourhood operation to be performed using the simplified communication and computation operations:

$$\begin{array}{ll}
 \mathbf{a} & : [1024; 1024] \\
 \mathbf{k} & : [32, 32; 32, 32] \\
 \mathbf{m} & : [0, 2; 1, 3] \\
 \mathbf{d} & : 1024; 1024] \\
 \Rightarrow & \\
 \mathbf{a} & : [1024; 1024] \\
 \mathbf{k} & : [32, 32; 32, 32] \\
 \mathbf{t}_K & : [34, 32; 34, 32] \\
 \mathbf{o}_{T_K} & : [1, 0; 1, 0] \\
 \mathbf{m} & : [0, 2; 1, 3] \\
 \mathbf{d} & : [1156; 1024]
 \end{array}$$

7.3.8 Computing the Mandelbrot set

The computation of the Mandelbrot set by iteration on each pixel has some properties that allow the computation to be optimized by choosing the appropriate data mapping. The calculation of each pixel is independent of every other pixel: only its position in the complex plane needs to be known. However, the cost of computing a pixel is generally similar to that of its neighbours, because every iteration contour N surrounding the Mandelbrot set is connected and is surrounded by iteration contours $N \pm 1$. This situation breaks down near the Mandelbrot set where the contours are closer together than a single pixel, but is usually a good guide.

This spatial dependence of the cost of computing a pixel may be used to balance the load per processor when computing the Mandelbrot set. If each processor has a similar spatial distribution of pixels, the computational load per processor should be similar. A 2d cut'n'stack mapping gives each processor a widely and evenly spaced collection of pixels, and therefore should have good load-balancing characteristics. This characteristic can be used to perform load-balancing of any algorithm in which the generation of pixels is spatially dependent and neighbourhood-independent; load balancing is improved in the merge-and box colour quantization algorithm with a well-spread spatial distribution [23, 24], and other problems such as ray-tracing could also benefit from this approach.

Tombouliau and Pappas have performed experiments in load-balancing Mandelbrot set calculations on the MasPar [65]. The results they obtain with a regular mapping are poor, and they attribute this to the fact that every processor contains a non-uniform distribution of the array. Although they describe their regular mapping as cut'n'stack, their results suggest that it is in fact a 2d hierarchical mapping. However, they obtained good results with a randomly skewed mapping. Further details about the mechanism for load-balancing may be obtained in their paper.

Good results can also be obtained using a regular 2d cut'n'stack mapping; figure 7.6 compares the best results of Tombouliau and Pappas using a randomly skewed mapping with those we obtained using a 2d cut'n'stack mapping, and figure 7.7 shows the regions of the Mandelbrot used for the tests. These figures show that the 2d cut'n'stack mapping is superior to the random skewing

Iteration counts and time for solving 512×512 Mandelbrot with 4K PEs					
real	imaginary	Random skew		2d cut'n'stack	
		iter	time (sec)	iter	time (sec)
-2.0,0.5	-1.25,1.25	25949	8.39	20280	5.80
-1.251,-1.25	0.024,0.025	11995	3.85	11952	3.57
-0.6,-0.5	-0.6,-0.5	33080	10.66	29376	8.38
0.26,0.27	0.00,0.01	52574	16.93	48672	13.73
-1.26,-1.24	0.01,0.03	61114	19.68	58416	16.22
0.2732,0.2741	-0.0064,-0.0055	13123	4.22	11952	3.58

Figure 7.6: Mandelbrot set calculation comparison between Tombouliau and Papert's random skewing and a 2d cut'n'stack mapping

method in both time and number of iterations required: although the source code for Tombouliau and Papert's is not the same as ours, the same compiler and run-time parameters were used for both tests.

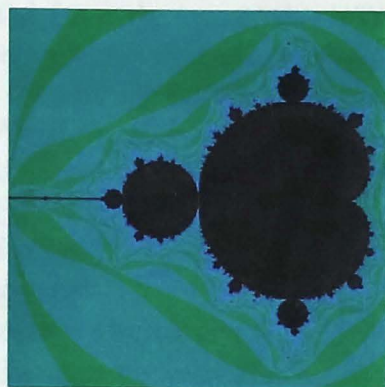
Another consideration in the choice of mapping is the set-up cost in calculating the mapping and the recovery cost in either writing the computed data to disk or to a display. The start-up cost in the random skewing method is small, involving the computation of 64 random numbers. Because it is a regular mapping, the start-up cost for the 2d cut'n'stack method is negligible. The recovery cost for the random skewing method will take one or two steps; communicating each of the 64 pixels a random distance along the x axis of the MasPar's mesh, and if the resultant regular mapping is not suitable for display, remapping it. These costs are negligible compared to the computation time. The recovery cost for the 2d cut'n'stack method is a single remapping, from a 2d cut'n'stack mapping to a 1d hierarchical mapping suitable for a fast write to disk:

$$\begin{array}{ll}
 \mathbf{a} & : [512; 512] \\
 \mathbf{k} & : [64, 8; 64, 8] \\
 \mathbf{m} & : [1, 3; 0, 2] \\
 \mathbf{d} & : [64; 4096]
 \end{array}
 \Rightarrow
 \begin{array}{ll}
 \mathbf{a} & : [512; 512] \\
 \mathbf{k} & : [64, 8; 512] \\
 \mathbf{m} & : [0; 1, 2] \\
 \mathbf{d} & : [64; 4096]
 \end{array}$$

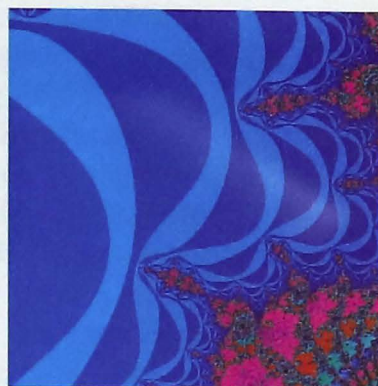
This remapping takes 14mS, and so is also insignificant compared to the cost of calculating pixels in the Mandelbrot set.

7.4 Summary

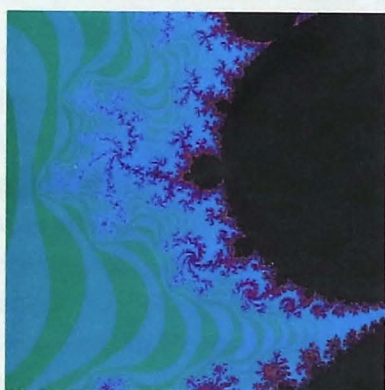
This chapter examines the scope for the usage of a PMF system by giving examples of its use for: the implementation of a compiler incorporating data



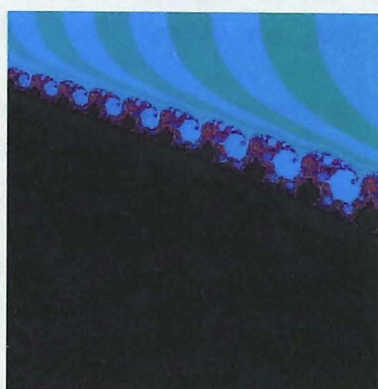
(a)



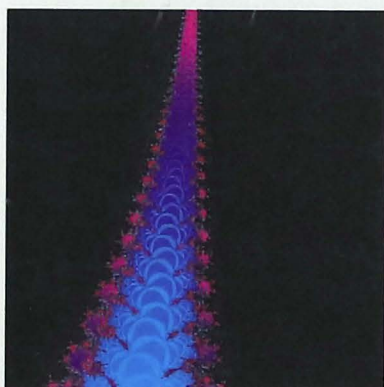
(b)



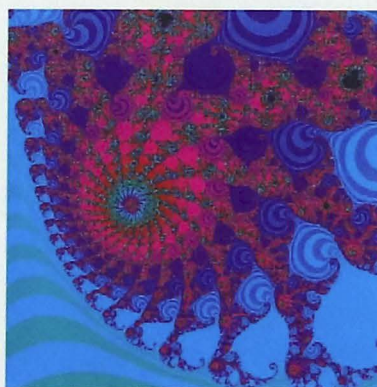
(c)



(d)



(e)



(f)

Figure 7.7: Sections of the Mandelbrot set computed with a 2d cut'n'stack mapping

mapping constructs; an image processing kernel; and several applications that have benefitted from the use of the k -Tile format and PMFs.

HPF is perhaps the first programming language which includes a rich class of directives for specifying the data mapping of a data array onto a parallel processor. We show that all the mapping directives specified by HPF may also be specified using the k -Tile format, hence all the remapping operations specified by HPF may be performed by a complete PMF system. PMFs provide a richer class of mapping specifications than HPF, and PMFs are not defined within a language such as Fortran 90, giving them a wider applicability. At present, however, only a radix 2 PMF system has been implemented.

The relationship between the k -Tile format and KIPS has been examined briefly. PMFs were designed for computation on parallel devices, and will provide the basis for implementing data tiling for KIPS on parallel architectures. The functionality of both PMFs and KIPS could be merged in the future to provide a powerful data mapping system for both parallel and sequential architectures.

Several visualization algorithms have benefitted from the implementation of radix 2 PMFs on the MasPar MP-1. A large class of scan-line algorithms, such as 2- and 3- dimensional rotation, perspective surface view generation and volume visualization have been implemented using PMFs for axis transposition/reflection and remapping to make display and disk operations both simpler and faster. Other operations such as the multidimensional FFT, neighbourhood filters and the computation of the Mandelbrot set have also been shown to benefit from approaches using data remapping.

8.1 The data mapping problem

To address the data mapping problem we developed the k -Tile format, based on Fraser's multidimensional tile format. The k -Tile format provides a flexible method for specifying the mapping of multidimensional images to multidimensional storage devices. Many useful data mapping techniques are incorporated in the k -Tile format.

mapping constructs, an image processing kernel, and several applications that have benefited from the use of the b-Tile format and PMPs.

HIPF is perhaps the first programming language which includes a rich class of directives for specifying the data mapping of a data array onto a parallel processor. We show that all the mapping directives specified by HIPF may also be specified using the b-Tile format, hence all the remapping operations specified by HIPF may be performed by a complete PMP system. PMPs provide a richer class of mapping specifications than HIPF, and PMPs are not defined within a language such as Fortran 90, giving them a wider applicability. At present, however, only a radix 2 PMP system has been implemented.

The relationship between the b-Tile format and KIPs has been examined briefly. PMPs were designed for computation on parallel devices, and will provide the basis for implementing data tiling for KIPs on parallel architectures. The functionality of both PMPs and KIPs could be merged in the future to provide a powerful data mapping system for both parallel and sequential architectures.

Several visualization algorithms have benefited from the implementation of radix 2 PMPs on the Maspar MFP-1. A large class of scan-line algorithms such as 2- and 3- dimensional rotation, perspective surface view generation and volume visualization have been implemented using PMPs for axis transformation/rotation and resampling to make display and disk operations both simpler and faster. Other operations such as the multidimensional FFT, neighborhood filters and the computation of the Mandelbrot set have also been shown to benefit from approaches using data remapping.

Chapter 8

Conclusions

This thesis addresses two problems becoming increasingly relevant as more processing is applied to larger multidimensional data arrays on increasingly massively parallel architectures:

- The task of mapping a data array onto a device, which we have termed the *data mapping problem*
- The task of altering the mapping of a data array already mapped to a device by rearranging the data, which we have termed the *data remapping problem*.

In this thesis we have examined some aspects of the mapping and remapping problems in detail. The approach we have chosen is to define a concise and flexible framework within which data mappings can be defined, and using this framework as a basis for a system of algorithms to perform data remappings. We have limited the scope of this thesis to examine regular mappings of multidimensional data arrays, but many problems in visualization, simulation and image processing can be addressed within this framework.

In this chapter we look at the effectiveness of the approach and our proposed solutions to the problems raised, outline limitations in the approach and suggest possible directions for future work.

8.1 The data mapping problem

To address the data mapping problem we developed the *k*-Tile format, based on Fraser's multidimensional tile format. The *k*-Tile format provides a flexible method for specifying the mapping of multidimensional images to multidimensional storage devices. Many useful data mapping techniques are incorporated in the *k*-Tile format:

- hierarchical decomposition of image dimensions into tiles to allow the shape of an image to be matched to the storage device, and the layout of the image to be matched to the algorithms;
- reversal and translation of image dimensions to allow the specification of geometrical transformations;
- padding of dimensions to allow images with inconvenient sizes to be mapped more flexibly, and to allow empty space to be declared around data tiles;
- replication of data across distributed memories to allow a local copy of data to be stored with every processor.

The basic form of the k -Tile format requires the specification of four vectors. Additional flexibility is provided by the specification of a selection of several other vectors.

8.2 The data remapping problem

A data remapping may be specified using a pair of k -Tile formats, one representing the source data mapping and the other representing a destination data mapping. This method of specification does not prescribe how the data movement is to be performed, however. To provide a solution to the data remapping problem, algorithms must be created to take two k -Tile formats describing a data remapping and move data on the storage device to perform the remapping.

A system to perform this task is called *Parallel Mapping Functions* (PMFs). We have developed *radix 2 PMFs*, which are a restricted form of PMFs, and have developed algorithms for performing some components of a general PMF system.

8.2.1 Radix 2 PMFs

A restricted form of the k -Tile format, in which data sets have dimension lengths a power of two and not incorporating translation, padding, or replication, defines the 2^k -Tile format. Using the 2^k -Tile format we implemented a restricted form of PMFs, *radix 2 PMFs*.

We implemented radix 2 PMFs using a generalized form of *index bit permutation*, which forms the basis for the work of Flander's Parallel Data Transforms (PDTs) on the DAP. PDTs perform index bit permutation on the DAP using a mesh connection network and direct memory addressing.

We have designed algorithms for performing index bit permutation on parallel architectures incorporating indirect memory addressing and a crossbar-like connection network. The algorithms are optimal in their use of the communications network, and use a small factor times the minimum number of memory operations. The algorithms operate by breaking an index bit permutation into a set of index bit permutation *cycles*, which may be performed independently and often simultaneously.

We have implemented these algorithms on the MasPar MP-1 within a radix 2 PMF system, and have incorporated many additional optimizations for the MasPar architecture. We have tested the speed and reliability of our implementation against many random remapping problems, a computed lower bound, and remapping routines written by others. The results obtained show our radix 2 PMF system to be fast, reliable and flexible.

8.2.2 Mixed radix remapping

The restriction of radix 2 PMFs to images whose dimension lengths must be powers of two is unfortunately a significant one. We have explored a generalization of index bit permutation, *mixed radix remapping*. We have outlined and implemented algorithms for performing several components of mixed radix remapping, which would be used in the implementation of a general PMF system:

- Mixed-radix index digit permutation
- Mixed-radix index digit re-signification, using two approaches:
 - Using a series of index digit permutations to allow re-signification within device dimensions
 - A brute-force approach performing index digit re-signification by using a general but inefficient permutation algorithm
- Processor cluster contention removal to optimize usage of crossbar connections shared by clusters of processors

Our results show that many components of a general PMF system could be implemented efficiently, although some components need more attention.

8.3 Application of the approach

To further justify the importance of finding solutions to the data mapping and remapping problems, we have examined several areas which could benefit, and have benefited, from these ideas:

- High Performance Fortran is an extension to the Fortran '90 language which incorporates many data mapping directives. We show that these directives may be specified and performed using PMFs;
- KIPS is an image processing kernel used for the storage and manipulation of multidimensional images on storage and computational devices. The k -Tile format and PMFs will be incorporated within the storage model of KIPS, and will provide a bridge between sequential storage of images in disk files and parallel computation on distributed memory processor arrays;
- Many visualization, image processing and image generation systems have been implemented using PMFs for image format remapping, 2d and 3d geometrical image transformations, and algorithm optimization.

These examples show that the specification of the k -Tile format and the implementation of radix 2 PMFs provide both a useful tool for applications being created now, and a general framework for future systems.

8.4 Limitations of the approach, and future work

We have demonstrated that the k -Tile format and a general PMF system covers the needs for data mapping and remapping of many visualization and image processing algorithms. This approach has several limitations, some of which could be addressed in the future.

8.4.1 Data structures

Because we have limited the scope of this thesis to mappings of multidimensional data arrays, many commonly used data structures and operations are not addressed.

Although the multidimensional array is a suitable data structure for a large range of data storage and data processing tasks, it is not suitable for data structures such as data-bases, linked lists, graphical objects, quad-trees and sparse arrays. Integration of these objects within the k -Tile format would be desirable.

8.4.2 Data-dependent mappings and operations

Another limitation of the scope of this thesis is the requirement that data mappings be data-independent.

This requirement has many advantages, because many of the resulting data mapping algorithms are highly regular and suitable for implementation on SIMD architectures. However, this requirement also precludes the use of data compression, such as run-length encoding or the use of oct trees.

Many commonly used algorithms such as sorting and clustering are also related to data-dependent mappings. Full integration of data-dependent mappings within the k -Tile format may not be realistic, but some form of generalization to data dependency would be desirable.

8.4.3 A general PMF system

Although the implementation of a radix 2 PMF system was successful, there are still many problems to be solved before the implementation of a general PMF system would be possible. The experiments we performed with mixed radix remapping showed that a number of tasks associated with a general PMF system could be performed using similar algorithms to the radix 2 PMF system.

However, an implementation of index digit re-signification using the suggested brute-force approach is likely to be disproportionately slower than the other components of mixed radix remapping, and the use of a more intelligent approach would be desirable.

Before it would be possible to integrate mixed radix remapping into a general PMF system additional tasks, outlined in section 6.5, must be completed. However, we believe there are no conceptual barriers to prevent the implementation of a general PMF system.

8.4.4 Human interaction with data mappings

An unfortunate aspect of the data mapping problem is the difficulty humans have in visualizing and interpreting multidimensional data mappings. Although the k -Tile format is a simple structure of one-dimensional vectors, it is difficult to extract the meaning of a data mapping from this abstract description.

Ad-hoc approaches have an advantage over general approaches for describing simple data mappings because descriptive names are more easily recognized than values in vectors. However, the large number of mappings available with the k -Tile format make the incorporation of the ad-hoc approach within a general data mapping system unrealistic except for a few special cases.

A possible solution to the problem of learning to visualize data mappings is the provision of a tool that could give an easily interpreted visual representation of any data mapping. By providing an experimenter with practice and experience in specifying and visualizing data mappings, the selection of a data mapping for an arbitrary problem could become an easier task.

Bibliography

- [1] Active Memory Technology Ltd. *Parallel Data Transforms*. 65 Suttons Park Avenue, Reading, Berks, UK, 2 edition, November 1988. (man022.02).
- [2] ANSI. *Fortran 90*. ANSI, 1991. X3J3 internal document S8.118 Submitted as Text for ANSI X3.198-1991.
- [3] Gregory L. Baker and Jerry P. Gollub. *Chaotic dynamics : an introduction*. Cambridge University Press, 1990.
- [4] T. Blank. The MasPar MP-1 architecture. In *CompCon '90 San Francisco, California*, San Francisco, California, 1990.
- [5] Tom Blank. Personal Communication.
- [6] Rajendra V. Boppana and C. S. Raghavendra. Generalized schemes for access and alignment of data in parallel processors with self-routing interconnection networks. *Journal of Parallel and Distributed Computing*, 11:97-111, 1991.
- [7] Oscar Bosman, Peter Fletcher, and Kenneth Tsui. K-tiling: A structure to support regular ordering and mapping of image data. In *APRS Workshop on Two and Three Dimensional Spatial Data: Representation and Standards*, Perth, December 7-8 1992.
- [8] Ron Bracewell. *The Fourier Transform and its Applications*. McGraw-Hill, Inc., New York, 1965.
- [9] Ed Catmull and Alvy Ray Smith. 3d transformations of images in scanline order. In *SIGGRAPH '80 Conference Proceedings*, pages 279-285, 1980.
- [10] P. Christy. Virtual processors considered harmful. In *DMCC6*, Portland, Oregon, 1991.
- [11] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematics of Computation*, 19(90):297-301, 1965.

- [12] L. De Ferrari. Scientific visualisation on a massively parallel SIMD supercomputer: Approaches to image-based visualisation. In *4ASC QLD*, Bond University, Gold Coast, Australia, 1991.
- [13] Adriano J. de O. Cruz. Parallel algorithms for SIMD computers. *Microprocessing and Microprogramming*, 28:85-90, 1989.
- [14] Adriano Joaui de Oliveira Cruz. *The Design of a Control Unit and Parallel Algorithms for a SIMD computer*. PhD thesis, University of Southampton, 1988.
- [15] Dan E. Dudgeon and Russel M. Mersereau. *Multidimensional Digital Signal Processing*. Prentice-Hall International, Inc., London, 1984.
- [16] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Transactions on Computers*, C-21(7), July 1972.
- [17] Jeff Fier. Personal Communication.
- [18] Jeff Fier. Efficient router-based permutations on the MasPar MP-1/MP-2. Technical report, MasPar Computer Corporation, 749 North Mary Avenue Sunnyvale, California, 1993.
- [19] Jeff Fier. Ordered fast Fourier transforms on the MasPar MP-1/MP-2. Technical report, MasPar Computer Corporation, 749 North Mary Avenue Sunnyvale, California, 1993.
- [20] P. Flanders. *Languages and Techniques for Parallel Array Processing*. PhD thesis, Queen Mary College, 1982.
- [21] P. M. Flanders. The effective use of SIMD processor arrays. In Dennis Parkinson and John Litt, editors, *Massively Parallel Computing with the DAP*, pages 119-129. The MIT Press, Cambridge, Massachusetts, 1990.
- [22] Peter Fletcher. Scientific visualisation on a massively parallel SIMD supercomputer: Regular mapping and handling of multidimensional data on SIMD architectures. In *The Fourth Australian Supercomputing Conference*, Bond University, Gold Coast, Queensland, 1991.
- [23] Peter Fletcher. A SIMD parallel colour quantization algorithm. *Computers and Graphics*, 15(3):365-373, 1991.
- [24] Peter A. Fletcher. Adaptive selection of optimised colour subsets for displaying 24-bit colour images on 8-bit graphics workstations. In *Image Processing and the Impact of New Technologies*, pages 193-196, Australian Defence Force Academy, Canberra, Australia, December 18-20 1989.

- [25] Peter A. Fletcher and Phillip K. Robertson. A generalised framework for parallel data mapping in multidimensional signal and image processing. In *International Symposium on Signal Processing and its Applications*, pages 614-617, Gold Coast, Australia, 1992.
- [26] Geoffrey Fox, Seema Hiranandani, Ken Kennedy, Charles Koelbel, Uli Kremer, Chau-Wen Tseng, and Min-You Wu. Fortran D language specification. Technical report, Department of Computer Science, Rice University, P.O. Box 1892, Houston, Texas, April 1991.
- [27] D. Fraser. Array permutation by index digit permutation. *Journal of the ACM*, 23:298-309, 1976.
- [28] D. Fraser. Bit reversal and generalized sorting of multidimensional arrays. *Signal Processing*, 9(3):163-176, 1985.
- [29] D. Fraser. Multidimensional image data formats. Consultation report, CSIRO Division of Information Technology, 1988.
- [30] Donald Fraser. Rectification of multichannel images in mass storage using image transposition. *Computer Vision, Graphics, and Image Processing*, 29:23-36, 1985.
- [31] Donald Fraser and Eddie O'Brian. Fast image rotation techniques using a colour display. In *Proceedings of the Digital Equipment Computer Users Society*, pages 1601-1604, Christchurch, New Zealand, August 1979.
- [32] Rafael C. Gonzalez and Paul Wintz. *Digital Image Processing (Second Edition)*. Addison-Wesley Publishing Company, Inc., 1987.
- [33] Pat Hanrahan. Three-pass affine transforms for volume rendering. *Computer Graphics*, 24(5):71-78, 1990.
- [34] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Prentice-Hall, 2 edition, 1987.
- [35] David B. Harris, James H. McClellan, David S. K. Chan, and Hans W. Schuessler. Vector radix fast Fourier transform. In *IEEE International Conference on Acoustics Speech and Signal Processing*, pages 548-541, Hartford, Conn., May 1977.
- [36] High Performance Fortran Forum. *DRAFT High Performance Fortran Language Specification*. Rice University, Houston Texas, 1992. <ftp://anonymous@titan.cs.rice.edu:public/HPFF>.
- [37] W. D. Hillis. *The Connection Machine*. MIT press, Cambridge, Massachusetts, 1985.

- [38] Brian Kernighan and Dennis Ritchie. *The C Programming Language*. Prentice-Hall, 1978.
- [39] D. Knuth. *The Art of Computer Programming, Volume 1 - Fundamental Algorithms*. Addison-Wesley, 1973.
- [40] Christopher Lee Kuszmaul. Fast Fourier transform. Technical report, MasPar Computer Corporation, 749 North Mary Avenue Sunnyvale, California, March 1990.
- [41] Christopher Lee Kuszmaul. FFT communications requirement optimizations on massively parallel architectures with local and global interprocessor communications facilities. In *The International Society for Optical Engineering*, San Diego, California, July 1990.
- [42] Christopher Lee Kuszmaul. Rapid transpose methods on massively parallel SIMD computers. In *The Society for Industrial and Applied Mathematics Annual Meeting*, Chicago, Illinois, July 1990.
- [43] F. Thomson Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays-Trees-Hypercubes*. Morgan Kaufmann, 2929 Campus Drive, Suite 260, San Mateo, CA, 1992.
- [44] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29-37, 1988.
- [45] John M. H. Lilleyman and G. H. Allen. Dynamic bit-mapping of address space by hardware. In *International Symposium on Signal Processing and its Applications*, pages 781-784, Brisbane, Australia, 1987.
- [46] John Michael Lilleyman. *Bit Mapping of Address Space by Hardware for General Purpose Microprocessor Systems*. PhD thesis, James Cook University, 1991.
- [47] MasPar Computer Corporation. *MasPar Assembly Language (MPAS) Reference Manual*. 749 North Mary Avenue Sunnyvale, California, 1990.
- [48] MasPar Computer Corporation. *MasPar Programming Language (ANSI C Compatible MPL) Reference Manual*. 749 North Mary Avenue Sunnyvale, California, software version 2.2 edition, December 1991. Document Part Number 9302-0001, Revision A1, December 1991.
- [49] MasPar Computer Corporation. *MasPar Programming Language (ANSI C Compatible MPL) User Guide*. 749 North Mary Avenue Sunnyvale, California, software version 2.2 edition, December 1991. Document Part Number 9302-0101, Revision A1, December 1991.

- [50] MasPar Computer Corporation. The design of the MasPar MP-2: A cost effective massively parallel computer. Technical Report TW057.1092, MasPar Computer Corporation, 749 North Mary Avenue Sunnyvale, California, 1992.
- [51] MasPar Computer Corporation. *MasPar Image Processing Library (MPIPL) Reference Manual*. MasPar Computer Corporation, 749 North Mary Avenue Sunnyvale, California, 1992.
- [52] MasPar Computer Corporation. *MasPar MP-1 and MP-2 Architecture Specification*. 749 North Mary Avenue Sunnyvale, California, 1992.
- [53] M. Metcalf and J. Reid. *Fortran '90 explained*. Oxford University Press, Oxford, 1990.
- [54] Scott Milton. Irregular routing strategies on the MasPar MP-1 architecture. CSIRO Division of Information Technology, 1992.
- [55] Chris J. Moran, Dale C. Sutcliffe, and Lisa G. De Ferrari. Kernel image processing software (KIPS) functional specification version 1.2. Technical report, CSIRO Division of Information Technology, May 1990.
- [56] D. Nassimi and S. Sahni. An optimal routing algorithm for mesh-connected parallel computers. *Journal of the ACM*, 27(1):6-29, 1980.
- [57] Gary Newman. Memory management support for tiled array organization. Technical memorandum, Kodak Electronic Printing Systems, 164 Lexington Road, Billerica, MA, 1992.
- [58] Alan W. Paeth. A fast algorithm for general raster rotation. In *Graphics Interface '86*, pages 77-81, 1986.
- [59] D. Parkinson, D. J. Hunt, and K. S. MacQueen. The AMT DAP 500. In Dennis Parkinson and John Litt, editors, *Massively Parallel Computing with the DAP*, pages 77-84. The MIT Press, Cambridge, Massachusetts, 1990.
- [60] Lutz Prechelt. Measurements of MasPar MP-1216A communication operations. Technical Report 01/93, Universität Karlsruhe, Postfach 6980 D-7500 Karlsruhe, Germany, 1993.
- [61] Philip K. Robertson. Fast perspective views of images using one-dimensional operations. *IEEE Computer Graphics and Applications*, 7(2):47-56, February 1987.

- [62] Philip K. Robertson. Spatial transformations for rapid scanline surface shadowing. *IEEE Computer Graphics and Applications*, 9(3):47-56, March 1989.
- [63] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2), 1984.
- [64] Kevin A. Smith and Peter A. Fletcher. Status of PMFs. Working paper HJ/2/6-1, CSIRO Division of Information Technology, 1992.
- [65] Sherryl Tombouliau and Matthew Pappas. Indirect addressing and load balancing for faster solution to Mandelbrot set on SIMD architectures. In *Third Symposium on the Frontiers of Massively Parallel Computation*, College Park, MD, October 1990.
- [66] K. Tsui, P. Fletcher, and S. Hungerford. A flexible kernel image model. In *Proc. DICTA-91*, pages 502-509, Melbourne, Australia, December 1991.
- [67] Lewis W. Tucker and George G. Robertson. Architecture and applications of the connection machine. *Computer*, 21:26-38, August 1988.
- [68] Robert Ulichney. *Digital Halftoning*. MIT Press, Cambridge, Massachusetts, 1990.
- [69] Unknown. Algorithm 513. *Transactions on Maths Software*, January 1977.
- [70] Marin van Heel. A fast algorithm for transposing large multidimensional data sets. *Ultramicroscopy*, 38(1):75-83, October 1991.
- [71] G. Vézina, Peter A. Fletcher, and Philip K. Robertson. Volume rendering on the MasPar MP-1. In *1992 Workshop on Volume Visualisation*, pages 3-8, 1992.
- [72] G. Vézina and P. Robertson. Scientific visualisation on a massively parallel SIMD supercomputer: Viewing and processing. In *4ASC QLD*, 1991.
- [73] Guy Vézina and Philip K. Robertson. Terrain perspectives on a massively parallel SIMD computer. In N. M. Patrikalakis, editor, *Scientific Visualization of Physical Phenomena*. Springer-Verlag, 1991.
- [74] Richard F. Voss. Fractals in nature. In Heinz-Otto Peitgen and Dietmar Saupe, editors, *The Science of Fractal Images*, chapter 1, pages 49-51. Springer-Verlag New York Inc., 1988.

Appendix A

Cluster contention removal

In section 6.4 a successful algorithm for cluster contention removal was introduced. The code for the algorithm is introduced here because of its length. Another algorithm for cluster contention removal was developed, but had many undesirable characteristics. Because a large amount of effort was spent in developing this algorithm, it is also included here.

Function A.1, `uncontend1`, was described in section 6.4 and finds a contention-free ordering by applying transformations to a sub-optimal ordering.

Function A.2, `uncontend2`, will find such an ordering by recursively trying each processor in each cluster in turn. By the use of parallelism, the algorithm is able to detect any deadlock situations as they occur.

As an example, figure A.1 shows the operation of the second algorithm for a processor permutation with $K = 3$ and $C = 7$. Clusters are labelled a through to g , sending processors are indicated \boxed{a} and disabled processors as $-$.

The worst-case running time of function A.2 is not known, but its behaviour varied considerably when applied to different random processor permutations. Contention-free orderings were obtained for most permutations within about a second, but for some permutations no solutions appeared even after 100 seconds.

These situations are assumed to occur when a processor is chosen as a sender which force a deadlock many levels deeper in the tree of choices, forcing the traversal of many unrewarding pathways. Because they are uncommon, it is possible that a different traversal order may avoid the slow situations completely.

The algorithm was modified to give up on a solution that was taking too much time, randomly re-ordering the traversal order of both the clusters and the processors within the clusters, and starting again with the new traversal order. This scrambling step increased the speed of the previously intractable permutations, making all computations take a similar amount of time. Figure A.2 shows the execution time of a scrambling `uncontend2` on 10000 random

i. Desired processor permutation

<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>f</i>	<i>g</i>	<i>b</i>	<i>b</i>	<i>f</i>	<i>e</i>	<i>e</i>
<i>b</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>g</i>	<i>d</i>	<i>d</i>

ii. Try sending to *d* from cluster *a*

d	<i>e</i>	<i>f</i>	<i>g</i>	<i>a</i>	<i>a</i>	<i>a</i>
-	<i>g</i>	<i>b</i>	<i>b</i>	<i>f</i>	<i>e</i>	<i>e</i>
-	<i>c</i>	<i>c</i>	<i>c</i>	<i>g</i>	-	-

iii. Try sending to *e* from cluster *b*

d	e	<i>f</i>	<i>g</i>	<i>a</i>	<i>a</i>	<i>a</i>
-	-	<i>b</i>	<i>b</i>	<i>f</i>	-	-
-	-	<i>c</i>	<i>c</i>	<i>g</i>	-	-

iv. Force send from clusters *f*, *g*

d	e	<i>f</i>	<i>g</i>	-	a	a
-	-	<i>b</i>	<i>b</i>	<i>f</i>	-	-
-	-	<i>c</i>	<i>c</i>	<i>g</i>	-	-

v. Failure; unroll

d	-	<i>f</i>	<i>g</i>	<i>a</i>	<i>a</i>	<i>a</i>
-	<i>g</i>	<i>b</i>	<i>b</i>	<i>f</i>	<i>e</i>	<i>e</i>
-	<i>c</i>	<i>c</i>	<i>c</i>	<i>g</i>	-	-

vi. Try sending to *g* from cluster *b*

d	-	<i>f</i>	-	<i>a</i>	<i>a</i>	<i>a</i>
-	g	<i>b</i>	<i>b</i>	<i>f</i>	<i>e</i>	<i>e</i>
-	-	<i>c</i>	<i>c</i>	-	-	-

vii. Try sending to *f* from cluster *c*

d	-	f	-	<i>a</i>	<i>a</i>	<i>a</i>
-	g	-	<i>b</i>	-	<i>e</i>	<i>e</i>
-	-	-	<i>c</i>	-	-	-

viii. Forced sends from *d*, *e*

d	-	f	-	a	-	-
-	g	-	b	-	<i>e</i>	<i>e</i>
-	-	-	c	-	-	-

ix. Forced sends from *f*, *g*

d	-	f	-	a	-	-
-	g	-	b	-	e	e
-	-	-	c	-	-	-

x. Failure; unroll

d	-	-	-	<i>a</i>	<i>a</i>	<i>a</i>
-	g	<i>b</i>	<i>b</i>	<i>f</i>	<i>e</i>	<i>e</i>
-	-	<i>c</i>	<i>c</i>	-	-	-

xi. Try sending to *b* from cluster *c*

d	-	-	-	<i>a</i>	<i>a</i>	<i>a</i>
-	g	b	-	<i>f</i>	<i>e</i>	<i>e</i>
-	-	-	<i>c</i>	-	-	-

xii. Forced sends from *d*, *e*

d	-	-	-	-	<i>a</i>	<i>a</i>
-	g	b	-	f	<i>e</i>	<i>e</i>
-	-	-	c	-	-	-

xiii. Try sending to *a* from *f*

d	-	-	-	-	a	-
-	g	b	-	f	-	<i>e</i>
-	-	-	c	-	-	-

xiv. Forced send from *g*

d	-	-	-	-	a	-
-	g	b	-	f	-	e
-	-	-	c	-	-	-

xv. Done. Next problem:

-	<i>e</i>	<i>f</i>	<i>g</i>	<i>a</i>	-	<i>a</i>
<i>f</i>	-	-	<i>b</i>	-	<i>e</i>	-
<i>b</i>	<i>c</i>	<i>c</i>	-	<i>g</i>	<i>d</i>	<i>d</i>

Figure A.1: Finding a contention-free ordering of a processor permutation

processor permutations.

Unfortunately, the algorithm has two faults. It is, at its heart, a sequential algorithm, so the in the best case execution still requires $K \times C$ iterations. In some cases, especially when dealing with index digit permutations, many thousands of scrambles may be performed with no solution being obtained.

Function A.1 *Generation of contention-free communication ordering*

```

plural int order = ⟨index of processor within cluster⟩;
plural int ibase = ⟨base processor within cluster⟩;
plural int dbase = ⟨base processor within destination cluster⟩;

uncontend1()
{
    int iorder;

    for (iorder=0; iorder<⟨number of processors in cluster⟩-1; iorder++)
    {
        do {
            /* Processor attempting to send from cluster, in base cluster */
            plural int current = -1;
            /* Base processor in current's destination cluster, in base processor */
            plural int cdbase = -1;
            /* Set to 1 in base processor if sending proc involved in deadlock */
            plural int deadlock = 0;
            /* Set to 1 in destination base proc if someone attempting to send */
            plural int touched = 0;
            /* Set to 1 if we should try to change sender */
            plural unsigned char tryfree = 0;

            /* If order==iorder, try to perform permutation */
            if (order==iorder) {
                router[ibase].current = iproc;
                router[ibase].cdbase = dbase;
            }

            /* Now check for deadlock */
            if (iproc == ibase) {
                plural unsigned char locked;
                plural int tmp;

                /* Attempt to connect */
                all tmp = -1;
                router[cdbase].tmp = iproc;
                all touched = (tmp ≥ 0);

                /* Did we fail to get there first ? */
                deadlock = !(router[cdbase].tmp == iproc);

                if (!globalor(deadlock)) {
                    /* We're done! Everyone connected */
                    break;
                }
            }
        } while (tryfree == 0);
    }
}

```

```

/* Identify deadlockers from deadlockees */
all locked = 0;
if (deadlock) {
    router[cdbase].locked = 1;
}
if (router[cdbase].locked == 1) {
    deadlock = 1;
}
}

/* Undeadlock by finding free proc to connect to */
fixed = 0;
tryfree = deadlock;
do {
    /* Guard allowing only one proc to grab free cluster */
    plural int grabfree;
    /* Guard allowing only one proc per cluster to grab free cluster */
    plural int gotfree;
    /* Base proc contains id of proc connected to free cluster */
    plural int canfree = -1;
    /* Base destination proc contains id of connecting processor */
    plural int canbefree = -1;
    /* Freeable proc */
    plural int freeforme = -1;
    /* Flag cluster that someone wants to talk to */
    plural char freeme = 0;

    subiter++;

    if ( (order > iorder) & router[ibase].tryfree ) {
        /* Possible sender in deadlock cluster */
        if (router[ibase].touched == 0) {
            /* Definite sender in deadlock cluster */
            router[ibase].grabfree = iproc;
            /* Try and grab free cluster */
            if (router[ibase].grabfree == iproc) {
                /* Got it! */
                router[ibase].gotfree = iproc;
                /* Tell base proc we're OK */
                if (router[ibase].gotfree == iproc) {
                    /* Got it; flag a fix */
                    fixed = 1;
                    /* Swap order with deadlocking processor */
                    router[router[ibase].current].order = order;
                    order = iorder;
                }
            }
        }
    }
    all {
        freeme = 0;
        tryfree = 0;
    }
}

```

```

        router[dbase].freeme = 1;
        router[ibase].deadlock = 1;
    }
    /* Things are a little easier now */
    if (fixed) break;

    /* Couldn't find free sender. */
    /* Try to free a cluster to allow another cluster to be freed */
    if ( (iproc == ibase) && (!deadlock) && router[cdbase].freeme ) {
        tryfree = 1;
    }
    } while (1);
} while (1);
}

/* Now order gives contention-free routing order */
}

```

Function A.2 Generation of contention-free communication ordering

```

#define NUM_CLUSTERS 64
#define NUM_IN_CLUSTERS 16

uncontend2(plural int dest,                                /* Destination processor */
           plural int *order                             /* Computed order of sending */
           )
{
    int iorder;

    for (iorder = 0; iorder < NUM_IN_CLUSTERS; iorder++) {
        int i=0;                                           /* Current cluster to test */

        (Disable processors with order < iorder)
        (If several procs in cluster sending to same cluster, disable all but one)

        do {
            if ((Cluster i has exactly one sender)) {
                i++;
                continue;                                /* This cluster OK, move to next cluster */
            } else if ((There is some untried processor P in cluster i)) {
                while ((P is newly next sending processor) ||
                       (A cluster newly has only one sender P) ||
                       (A cluster newly is sent to by only one P) ) {
                    (Disable other processors in same clusters as Ps)
                    (Disable processors sending to same clusters as Ps)
                    (Ps are senders)
                }

                if ((No more than one sender from each cluster) &&
                    (No more than one sender to each cluster) &&
                    (Every cluster has at least one potential sender) &&
                    (Every cluster has at least one potential receiver) ) {
                    i++;
                    continue;                             /* This cluster OK, move to next cluster */
                }
            }
        } while (1);
    }
}

```



```

    }
  }

  /* Something has failed */
  (Unroll i back to last processor tried)
} while (i < NUM_CLUSTERS);

(Set order = iorder in successfully sending processors)
}

/* order now gives contention-free ordering within cluster */
}

```

Appendix B

Status of PMFs

This working paper was written by Kevin Smith and Peter Fletcher and provides a formal set of notes and design instructions about the PMF system.

B.1 Introduction

Over the last few years the general concept of Parallel Mapping Functions (PMFs) that can be used for efficient data distribution on both serial and parallel computers has been developed. This work was based in turn on several years of previous work on generating and distributing image mappings and storage and distribution facilities. The PMF was chosen as the computing system for the implementation because of the fact that data re-ordering is a critical requirement for the PMF.

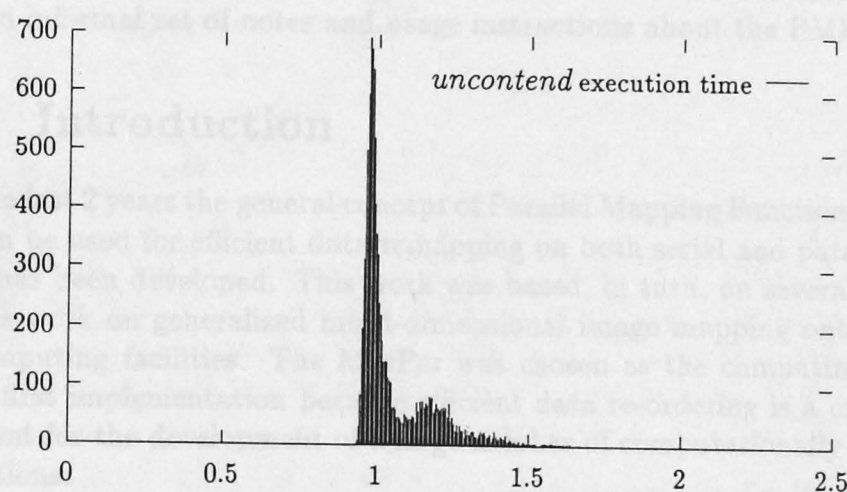


Figure A.2: Execution time of scrambling uncontend2 over 10000 random problems

A description of the data set is what is known as the Image Coordinate System (ICS). This is an abstract description of the multi-dimensional data set (usually an image) in a mathematical space. That is, a 16 bit 1024 by 1024 image is simply described as:

$$ICS: [2, 1024, 1024]$$

where the length of dimension 0 is 2, dimension 1 is 1024 and dimension 2 is 1024.

A segmentation of the image into tiles described by the ICS Coordinate System (KCS). For example, the 16 bit 1024 by 1024 system described above can be covered by 1024 16 bit 32 by 32 tiles. This would be represented as:

Appendix B

Status of PMFs

This working paper was written by Kevin Smith and Peter Fletcher and provides an informal set of notes and usage instructions about the PMF system.

B.1 Introduction

Over the last 2 years the general concept of Parallel Mapping Functions (PMFs) that can be used for efficient data remapping on both serial and parallel computers has been developed. This work was based, in turn, on several years of previous work on generalized multi-dimensional image mapping onto storage and computing facilities. The MasPar was chosen as the computing system for the first implementation because efficient data re-ordering is a critical requirement for the development of a large number of computationally intensive applications.

The mapping of a large range of regular multi-dimensional data sets onto a MasPar can be described using a "*k*-Tile Format". A *k*-Tile Format is made up of a number of components:

- A description of the data set in what is known as the Image Coordinate System (ICS). This is an abstract description of the multi-dimensional data set (usually an image) in a mathematical sense. That is, a 16 bit 1024 by 1024 image is simply described as:

ICS: [2, 1024, 1024]

where the length of dimension 0 is 2, dimension 1 is 1024 and dimension 2 is 1024.

- A segmentation of the image into tiles described by the *k*-Tile Coordinate System (KCS). For example, the 16 bit 1024 by 1024 system described above can be covered by 1024 16 bit 32 by 32 tiles. This would be represented as:

KCS: [2, 32, 32, 32, 32]

That is, the image can now be thought of as having been mapped into a 5 dimensional space. As with the ICS, the KCS is still an abstract concept.

- A model of the physical device that the image is to be stored on is required. In the case of the MasPar MP-1201B, there are several models that could be used for the same physical device. These different models are represented via a Device Coordinate System (DCS). For our 16 bit 1024 by 1024 image we could have many models; here are two examples:

DCS: [2048, 1024]

which views the device as being 2048 bytes of memory in each of a linear sequence of 1024 PE's,

DCS: [2048, 32, 32]

which views the device as being 2048 bytes of memory in each of a 2 dimensional array of 32 by 32 PE's.

- A mapping between the KCS and the DCS to describe how the image is actually stored on the device. This is called the KDMAP. For example, a 16 bit 1024 by 1024 image that is stored on a MP-1201 in a 2D Cut'n'stack Format (traditionally known as 2D sheet format) would have the following *k*-Tile format:

ICS: [2, 1024, 1024]

KCS: [2, 32, 32, 32, 32]

KDMAP: [0, 2, 4, 1, 3]

DCS: [2048, 1024]

Additionally, a full *k*-Tile format also contains a vector called KSENSE which allows the ordering of bytes to be reversed in any *k*-Tile dimension.

A library has been developed which allows:

- the *k*-Tile format of an image to be declared,
- the 'new' *k*-Tile format of an image to be declared,
- a remap function that takes an image, its existing *k*-Tile specification and a new *k*-Tile specification and, depending on which version of the library is used, will automatically generate either: a sequence of index bit permutations implemented with both xnet operations and direct PE memory addressing; or a sequence of indirect PE memory addressing and router operations

- once a remap sequence of operations is generated it does not need to be recalculated during the execution of the particular MasPar process.

There is also a command line based "PMF Workbench" interface that allows a user to declare k -Tile formats and perform remappings. The Workbench gives the user the timing on the MasPar for a particular remap and could be modified to let the user store the sequence of remapping operations which could then be imported directly into a MasPar program. This would allow a user developing signal processing applications to firstly find the optimal set of k -Tile to k -Tile remaps for, say, an FFT and then to store the resultant set of low-level remap operations in a macro assembler format which could then be imported as "in-line" code within a MasPar program.

Furthermore this PMF Workbench could be easily extended to have a GUI interface.

Currently, PMFs have only been implemented for power-of-two k -Tile dimensions. However, there is no limitation on the utility of the k -Tile format for specifying non power-of-two data mappings, and much of the low-level data movement code could be used with arbitrary mixed radix data remapping with little change.

By allowing an offset to be specified for each image dimension in the k -Tile mapping, it will in future be possible to use PMFs for the translation of multi-dimensional data, and will give PMFs the full functionality required for the regular FORTRAN D 'DISTRIBUTE' and 'ALIGN' statements; currently, PMFs would only support the 'DISTRIBUTE' statement alone.

B.2 PMF system calls

In order to access the functionality of the PMF system, several functions are provided for specifying k -Tile formats, declaring data memory, initializing data remaps and remapping data in-place or copying. Calls are also provided for initializing any of the standard one- and two-dimensional parallel mappings and performing geometric transformations such as transpositions and reversals of data axes.

B.2.1 The k -Tile format

The k -Tile format is used to describe a mapping of an image onto the MasPar processor array. The meaning of the elements of this structure are described in other documents, but its contents are:

- ICS - Image Coordinate System
- KCS - k -Tile Coordinate system

- DCS - Device Coordinate system
- KDMAP - Map between KCS and DCS
- KSENSE - Direction indicator for KCS dimensions

Declaring *k*-Tile format - `gr_new_ktile(in ktile, out kid)`

The function 'gr_new_ktile' takes a *k*-Tile format as an argument, which it checks and converts to a canonical form. If the format has already been specified, a usage count is incremented and no memory storage is required. If valid, the a *k*-Tile identifier is output and 0 returned. If the format is invalid, a descriptive error message is printed and -1 returned.

Errors

- ics dims in *k*-Tile format is bad
- kcs dims in *k*-Tile format is bad
- dcs dims in *k*-Tile format is bad
- ics dim in *k*-Tile format is bad
- kcs dim in *k*-Tile format is bad
- dcs dim in *k*-Tile format is bad
- kdmmap out of range
- dcs does not fit within pmem
- dcs does not fit in processor array
- dcs does not fit in processor x array
- dcs does not fit in processor y array
- dcs must have 0, 1, 2 or 3 dcs dimensions
- kcs dimension mentioned twice in kdmmap
- too many kcs dims for ics
- kcs dim overflows dcs dim
- kcs dims do not cover dcs dims
- bad ksense

- kcs dims do not fit in ics dims
- kcs dims do not cover ics dims
- no free k -Tile ids

Inquiring k -Tile format - `gr_inquire_ktile(in kid, out ktile)`

The function 'gr_inquire_ktile' takes a k -Tile identifier as an argument. If the identifier is valid, the appropriate k -Tile format is output and 0 returned. If invalid, a descriptive error message is printed and -1 returned.

Errors

- bad k -Tile id
- k -Tile id not allocated

Freeing k -Tile format - `gr_free_ktile(in kid)`

The function 'gr_free_ktile' takes a k -Tile identifier as an argument. If the identifier is invalid or if the k -Tile identifier is referred to by an mtag or a pair, a descriptive error message is printed and -1 returned. Otherwise, the usage count of the kid is decremented, and if this is zero the storage for the k -Tile is freed, and zero is returned.

Errors

- bad k -Tile id
- k -Tile id not allocated
- mtag still references kid
- pair still references kid

B.2.2 The mtag

The mtag is used to declare memory storage to the PMF system. It is the application's responsibility to allocate and release memory, but the PMF system will maintain information about the size and mapping associated with a piece of memory. The contents of an mtag are:

- mem - pointer to data memory
- size - size (in bytes) of memory
- kid - k -Tile id associated with this memory (-1 if none)

Declaring an mtag - `gr_new_mtag(in mem, in size, out mid)`

The function '`gr_new_mtag`' takes a memory address and a size as arguments. If the memory defined is valid, a mtag id is output and zero returned. Otherwise, a descriptive error message is printed and -1 returned.

Errors

- mem is null
- mem is too high
- size is negative or zero
- size is too large
- size must be power of two
- mem+size is too high
- mem overlaps another mtag
- invalid *k*-Tile id
- size too small for kid
- no free mtag ids

Freeing an mtag - `gr_free_mtag(in mid)`

The function '`gr_free_mtag`' takes an mtag identifier as an argument. If the identifier is invalid a descriptive error message is printed and -1 returned. Otherwise, the mtag is freed and zero is returned.

Errors

- bad mtag id
- mtag id not allocated

Inquiring an mtag - `gr_inquire_mtag(in mid, out mtag)`

The function '`gr_inquire_mtag`' takes an mtag identifier as an argument. If the identifier is invalid a descriptive error message is printed and -1 returned. Otherwise, the mtag is output and zero is returned.

Errors

- invalid mtag id

Associating a mapping with an mtag - `gr_set`(in mid, in kid)

The function '`gr_set`' takes an mtag identifier and a *k*-Tile identifier as an argument. If either identifier is invalid or the mtag is too small for the mapping described by kid, a descriptive error message is printed and -1 returned. Otherwise, the the kid is associated with mid and 0 returned.

Errors

- invalid mtag id
- invalid *k*-Tile id
- size too small

B.2.3 The remap

Once you have declared data mappings with `gr_declare_ktile` and associated them with memory in `gr_declare_mtag`, you will want to actually perform data remappings. This can be performed directly by simply specifying a new *k*-Tile id for an existing mtag, but if the mapping is to be performed many times it is advisable to initialize the remap so that the data movements for the remap operation need only be computed once.

Initializing a remap - `gr_init_remap`(in kid1, in kid2)

The function '`gr_init_remap`' takes as arguments two *k*-Tile identifiers. If the two identifiers describe mappings of the same size, there is enough memory and no horrible bugs appear, a mapping is generated to remap data from format kid1 to format kid2 (but not vice-versa). Otherwise, a descriptive error message is printed and -1 returned. If a attempt is made to initialize a mapping more than once, only one copy of the mapping is kept and an internal usage count ensures that the mapping is retained while it is needed.

Errors

- kid #1 is invalid
- kid #2 is invalid
- image sizes do not match
- no free pair ids
- no free remap ids
- `p_malloc(remap-ps)` failed

Freeing a remap - `gr_free_remap(in kid1, in kid2)`

The function '`gr_free_remap`' takes as arguments two k -Tile identifiers. If either identifier is invalid or if the map between the k -Tile identifiers has not been initialized, a descriptive error message is printed and -1 returned. Otherwise, the usage count of the remap is decremented, and if this is zero the storage for the remap is freed, and zero is returned.

Errors

- kid #1 is invalid
- kid #2 is invalid
- invalid pair id

Performing an in-place remap - `gr_remap(in mid, in kid)`

The function '`gr_remap`' takes as arguments an mtag identifier and a k -Tile identifier. If either identifier is invalid or if the k -Tile identifier is too large for the mtag memory, a descriptive error message is printed and -1 returned. Otherwise, the data stored at the address referred to by the mtag is remapped in place and the mtag's kid updated to reflect the new mapping. Note that if a map has previously been initialized for this mapping, the operation will require no pre-initialization and will occur much faster.

Errors

- mid is invalid
- destination kid is invalid
- size is too small

Performing a copy remap - `gr_copy(in mid1, in mid2, in kid)`

The function '`gr_copy`' takes as arguments two mtag identifiers and a k -Tile identifier. If any identifier is invalid or if the k -Tile identifier is too large for the mtag memory, a descriptive error message is printed and -1 returned. Otherwise, the data stored at the address referred to by mtag1 is simultaneously remapped and copied to the memory referred to by mtag2, and mtag2's kid updated to reflect the new mapping. Note that if a map has previously been initialized for this mapping, the operation will require no pre-initialization and will occur much faster.

Errors

- mid #1 is invalid

- mid #2 is invalid
- destination kid is invalid
- size is too small

B.2.4 Standard mappings

Several routines have been provided to initialize *k*-Tile formats to standard mappings.

2 dimensional cut'n'stack mapping - `gr_make_2dcs`(in size, in x, in y, out ktile)

The function '`gr_make_2dcs`' takes as arguments an image size and a *k*-Tile structure. If the size is valid, a 2d cut'n'stack *k*-Tile format is generated and output and 0 returned. If the size is invalid, a descriptive error message is printed and -1 returned.

Errors

- a dimension is negative or zero
- x dim is not a multiple of `nxproc`
- y dim is not a multiple of `nyproc`

2 dimensional hierarchical mapping - `gr_make_2dh`(in size, in x, in y, out ktile)

The function '`gr_make_2dh`' takes as arguments an image size and a *k*-Tile structure. If the size is valid, a 2d hierarchical *k*-Tile format is generated and output and 0 returned. If the size is invalid, a descriptive error message is printed and -1 returned.

Errors

- a dimension is negative or zero
- x dim is not a multiple of `nxproc`
- y dim is not a multiple of `nyproc`

1 dimensional cut'n'stack mapping - `gr_make_1dcs`(in size, in x, in y, out ktile)

The function '`gr_make_1dcs`' takes as arguments an image size and a *k*-Tile structure. If the size is valid, a 1d cut'n'stack *k*-Tile format is generated and output and 0 returned. If the size is invalid, a descriptive error message is printed and -1 returned.

Errors

- a dimension is negative or zero
- cannot split y to cover processors
- cannot combine x to cover processors

1 dimensional hierarchical mapping - `gr_make_1dh`(in size, in x, in y, out ktile)

The function '`gr_make_1dh`' takes as arguments an image size and a *k*-Tile structure. If the size is valid, a 1d hierarchical *k*-Tile format is generated and output and 0 returned. If the size is invalid, a descriptive error message is printed and -1 returned.

Errors

- a dimension is negative or zero
- cannot split y to cover processors
- cannot combine x to cover processors

1 dimensional scan mapping - `gr_make_1dscan`(in size, in x, in y, out ktile)

The function '`gr_make_1dscan`' takes as arguments an image size and a *k*-Tile structure. If y is smaller than or equal to nproc, a 1d scan mapping, else a 1d hierarchical mapping, is generated and output and 0 returned. If the size is invalid, a descriptive error message is printed and -1 returned.

Errors

- none

B.2.5 Geometrical transformations

Once a *k*-Tile format has been generated, it is possible to perform several geometric transformations on the dimensions within the *k*-Tile format. Note that these functions will work on any valid *k*-Tile formats, not just those generated with `gr_make_*`. The operations may also be composed, but it must be remembered that because the operations are performed on the image dimensions rather than the device dimensions, they will appear to act as pre-operations rather than post-operations.

Transposition of axes - `gr_xpose(in dim1, in dim2, inout ktile)`

The function '`gr_xpose`' will transpose two image dimensions in a *k*-Tile format. Because there are several possible ways of transposing rectangular mappings, the two dimensions to be transposed must be the same size.

Errors

- dimension #1 is bad
- dimension #2 is bad
- dimensions must have same size

Reversal of axis - `gr_reverse(in dim, inout ktile)`

The function '`gr_xpose`' will reverse an image dimension in the *k*-Tile format.

Errors

- dimension is bad

Index bit reversal of axis - `gr_bitrev(in dim, inout ktile)`

The function '`gr_xpose`' will bit-reverse the index bits of an image dimension in the *k*-Tile format. This function is useful for the remapping prior to a radix 2 FFT.

Errors

- dimension is bad

B.3 Examples of Using PMFs

B.3.1 Performing a simple remap

Simple remap code: `ex1.m`

This example declares a 1024 x 1024 image and uses PMFs to transpose it. To ensure that all executable code is loaded into memory and timing is accurate,


```

/* (It would now be usual to fill image with something ... */
/* ... but we won't bother) */

gr_set(mid, kscan_id);          /* Attach a ktile id to the image */
gr_remap(mid, kxpos_id);        /* Transpose the image */

gr_set(mid, kscan_id);          /* Attach a ktile id to the image */
tstart();                       /* Timing is now valid (all code should be loaded) */
gr_remap(mid, kxpos_id);

/* Transpose the image again (so it's back to the start!) */
tfin("Time to do transpose");   /* Timing message */

/* That's all folks! */
}

```

Sample run of ex1.m on an MP-1201B

```

garnet% ex1
Time to do transpose : 0.037156
garnet%

```

B.3.2 Error reporting

“Buggy” transpose code: ex2.m

This example shows the style of error reporting that can be expected when using PMFs

```

#include "timing.h"
#include "gr.h"

plural void *p_malloc();

/*
 * Example 1:
 *
 * Transposing a 1kx1kx8bit image
 *
 */

void main()
{
    plural unsigned char *img;
    ktile_s ktile;

```

```

int kscan_id;
int kxpos_id;
int mid;

img = p_malloc(1024);                                /* Allocate image */

gr_make_1dscan( /* Create a ktile structure corresponding to image */
               1,                                /* One byte pixels */
               1023,                            /* x=1023 will create an error! */
               1024,                            /* y=1024 */
               &ktile);

gr_new_ktile(&ktile, &kscan_id);                    /* Declare new ktile format */

gr_xpose(1,2,&ktile); /* Modify ktile structure for transposed image */
gr_new_ktile(&ktile, &kxpos_id);                    /* Declare ktile format */

gr_init_remap(kscan_id, kxpos_id);
/* Create data structures for remap scan->transposed */
gr_init_remap(kxpos_id, kscan_id);
/* Create data structures for remap transposed->scan */

gr_new_mtag(img, 1024, &mid);                      /* Declare image to remap */

/* (It would now be usual to fill image with something ... */
/* ... but we won't bother) */

gr_set(mid, kscan_id);                             /* Attach a ktile id to the image */
gr_remap(mid, kxpos_id);                           /* Transpose the image */

gr_set(mid, kscan_id);                             /* Attach a ktile id to the image */
tstart();                                           /* Timing is now valid (all code should be loaded) */
gr_remap(mid, kxpos_id);

/* Transpose the image again (so it's back to the start!) */
tfin("Time to do transpose");                     /* Timing message */

/* That's all folks! */
}

```

Sample run of ex2.m on an MP-1201B

```

maspar% ex2
kmath.m:63: gp2_log : ??? log(1023)

```

```
ktiles.m:364: gp2_ktol : ics dim in ktile format is bad
*** ktile **
ICS: [1, 1023, 1024]
KCS: [1, 1023, 1024]
DCS: [1023, 1024]
KDMAP: [0, 1, 2]
KSENSE: [+++]
ktiles.m:583: gp2_convert_ktile : failed
alloc_ktiles.m:216: gp2_declare_ktile : failed
gr.m:97: gr_new_ktile : failed
kmath.m:63: gp2_log : ??? log(1023)
ktiles.m:364: gp2_ktol : ics dim in ktile format is bad
*** ktile **
ICS: [1, 1023, 1024]
KCS: [1, 1023, 1024]
DCS: [1023, 1024]
KDMAP: [0, 1, 2]
KSENSE: [+++]
stdmap.m:114: gr_xpose : failed
kmath.m:63: gp2_log : ??? log(1023)
ktiles.m:364: gp2_ktol : ics dim in ktile format is bad
*** ktile **
ICS: [1, 1023, 1024]
KCS: [1, 1023, 1024]
DCS: [1023, 1024]
KDMAP: [0, 1, 2]
KSENSE: [+++]
ktiles.m:583: gp2_convert_ktile : failed
alloc_ktiles.m:216: gp2_declare_ktile : failed
gr.m:97: gr_new_ktile : failed
alloc_ktiles.m:236: gp2_ktile_valid : ktile id not allocated
pairs.m:140: gp2_declare_pair : kid #1 invalid
gr.m:238: gr_init_remap : failed
alloc_ktiles.m:236: gp2_ktile_valid : ktile id not allocated
pairs.m:140: gp2_declare_pair : kid #1 invalid
gr.m:238: gr_init_remap : failed
alloc_ktiles.m:236: gp2_ktile_valid : ktile id not allocated
mtags.m:249: gp2_mtag_set_kid : invalid ktile id
gr.m:281: gr_set : failed
alloc_ktiles.m:235: gp2_ktile_valid : bad ktile id
gr.m:309: gr_remap : mtag kid is bad
alloc_ktiles.m:236: gp2_ktile_valid : ktile id not allocated
mtags.m:249: gp2_mtag_set_kid : invalid ktile id
```



```

gr.m:281: gr_set : failed
alloc_ktiles.m:235: gp2_ktile_valid : bad ktile id
gr.m:309: gr_remap : mtag kid is bad
Time to do transpose : 0.042832

```

B.3.3 Fourier transform

Fourier transform code: ex3.m

This example shows the use of PMFs for performing a radix 2 FFT. The one dimensional FFT code in the routine `fft()` is not shown for brevity.

```

#include <mpl.h>
#include <stdio.h>
#include "gr.h"

/*
 *
 * Example 3:
 *
 * Raw ktile formats for doing FFT
 *
 */

plural void *p_malloc();

extern void initroots();           /* Initializes roots-of-unity tables */
extern void randomize();           /* Fills image with random goop */
extern void fft();                 /* Perform an FFT */
extern void ifft();               /* Perform an inverse FFT */

/*
 * Define a scan-line mapping of complex numbers
 *
 * Storage layout:
 *
 * PE0:    real[0,0] real[0,1] .. real[0,1023] imag[0,0] ... imag[0,1023]
 * PE1:    real[1,0] real[1,1] .. real[1,1023] imag[1,0] ... imag[1,1023]
 * .. .. .
 * PE1023: real[1023,0]. . . . . imag[1023,1023]
 *
 * ICS dim 0 is float
 * ICS dim 1 makes pair of float
 * ICS dim 2 is x
 * ICS dim 3 is y

```

```

*
*/
ktile_s k_scan =
{
    { 4, { 4,2,1024,1024 } },
    { 4, { 4,2,1024,1024 } },
    { 2, { 8192,1024 } },
    { 4, { 0,2,1,3 } },
    0
};
/*
* Bit-reverse of dimension 2 (x)
*/
ktile_s k_br =
{
    { 4, { 4,2,1024,1024 } },
    { 13, { 4,2,2,2,2,2,2,2,2,2,2,2,1024 } },
    { 2, { 8192,1024 } },
    { 13, { 0,11,10,9,8,7,6,5,4,3,2,1,12 } },
    0
};
/*
* Transpose dimensions 2 and 3
*/
ktile_s k_tr =
{
    { 4, { 4,2,1024,1024 } },
    { 4, { 4,2,1024,1024 } },
    { 2, { 8192,1024 } },
    { 4, { 0,3,1,2 } },
    0
};
/*
* Transpose dimensions 2 and 3, and bitreverse dimension 3
*/
ktile_s k_trbr =
{
    { 4, { 4,2,1024,1024 } },

```

/* ICS */
/* KCS */
/* DCS */
/* KDMAP */
/* KSENSE */

/* ICS */
/* KCS */
/* DCS */
/* KDMAP */
/* KSENSE */

/* ICS */
/* KCS */
/* DCS */
/* KDMAP */
/* KSENSE */

/* ICS */

```

    { 13, { 4,2,1024,2,2,2,2,2,2,2,2 } },          /* KCS */
    { 2, { 8192,1024 } },                          /* DCS */
    { 13, { 0,12,11,10,9,8,7,6,5,4,3,1,2 } },      /* KDMAP */
    0                                              /* KSENSE */
};

plural float f[1024+1024];
plural float *ar = f;
plural float *ai = f+1024;

void main(int argc, char *argv[])

{
    int numt;                                     /* Number of FFT's */
    int i;

    int kid_scan;                                /* Ktile identifiers */
    int kid_br;
    int kid_tr;
    int kid_trbr;

    int mid;                                     /* Mtag identifier */

    numt = 2;

    gr_new_ktile(&k_scan, &kid_scan);             /* Declare kTile identifiers */
    gr_new_ktile(&k_br, &kid_br);
    gr_new_ktile(&k_tr, &kid_tr);
    gr_new_ktile(&k_trbr, &kid_trbr);

    gr_new_mtag(f, 2048*sizeof(float), mid);      /* Declare memory */

    tstart();                                    /* Initialize roots */
    initroots();
    randomize(ar);
    tfin("FFT initialization time");

    tstart();                                    /* Initialize remaps */
    gr_init_remap(kid_scan, kid_br);
    tfin("scan -> br initialization time");

    tstart();
    gr_init_remap(kid_scan, kid_trbr);

```



```

tfin("scan -> trbr initialization time");

for (i=0; i<numt; i++)
{
    fprintf(DOUT, "---\n");

    gr_set(mid, kid_scan);          /* Give memory a ktile id */

    tstart();
    gr_remap(mid, kid_br);          /* Bitreverse */
    tfin("Time for bitrev");

    tstart();
    fft();                          /* Do row FFT */
    gr_set(mid, kid_scan);

    /* Set mtag ktile id back to default scan map */
    tfin("Time for FFT #1");

    tstart();
    gr_remap(mid, kid_trbr);        /* Transpose and bitreverse */
    tfin("Time for transpose+bitrev");

    tstart();
    fft();                          /* Do column FFT */
    gr_set(mid, kid_scan);          /* Set back to scan map */
    tfin("Time for FFT #2");

    tstart();
    gr_remap(mid, kid_br);          /* Bitreverse */
    tfin("Time for bitrev");

    tstart();
    ifft();                         /* Inverse FFT columns */

    gr_set(mid, kid_scan);          /* Back to scan map */
    tfin("Time for inverse #1");

    tstart();
    gr_remap(mid, kid_trbr);        /* Transpose and bitreverse */
    tfin("Time for transpose+bitrev");

    tstart();
    ifft();                         /* Inverse FFT rows */

```

```

        tfin("Time for inverse #2");
    }

    printf("Goodbye!\n");
}

```

Sample run of ex3.m on an MP-1201B

```

garnet% ex3
FFT initialization time : 3.132999
scan -> br initialization time : 0.232343
scan -> trbr initialization time : 0.037523
---
Time for bitrev : 0.037081
Time for FFT #1 : 0.913147
Time for transpose+bitrev : 0.200400
Time for FFT #2 : 0.913144
Time for bitrev : 0.025950
Time for inverse #1 : 0.913118
Time for transpose+bitrev : 0.170086
Time for inverse #2 : 0.912824
---
Time for bitrev : 0.025947
Time for FFT #1 : 0.913146
Time for transpose+bitrev : 0.170092
Time for FFT #2 : 0.913144
Time for bitrev : 0.025949
Time for inverse #1 : 0.913127
Time for transpose+bitrev : 0.170085
Time for inverse #2 : 0.912818
Goodbye!
garnet%

```

B.3.4 Mandelbrot set generator

Mandelbrot code (abridged): ex4.m

This example shows the use of PMFs to convert a 2 dimensional cut'n'stack image (a very efficient mapping for computing the Mandelbrot set) to 1 dimensional hierarchical (efficient for writing the image to disk). Some of the early timings are slower than expected because the instructions for the program must be pages from disk.

```
#include <stdlib.h>
```

```

#include <stdio.h>
#include <sys/file.h>
#include <math.h>
#include <mpl.h>
#include <reduce.h>
#include <ppeio.h>
#include <strings.h>
#include <maspar/mp_libc.h>
#include "rasterfile.h"
#include "gr.h"

/*
 *
 * Example 4:
 *
 * Mandelbrot set generator
 *
 */

plural unsigned short counts[64][64];
unsigned char outarray[32*64];
extern int fix_wheader(struct rasterfile *h);
extern void iterate_indirect(
    double xs,
    double ys,
    double xo,
    double yo,
    int ni,
    int gridsize,
    int print,
    plural unsigned char *image[64],
    plural unsigned short counts[64][64]);
extern void iterate_direct(
    double xs,
    double ys,
    double xo,
    double yo,
    int ni,
    int gridsize,
    int print,
    plural unsigned char *image[64],
    plural unsigned short counts[64][64]);

main(argc, argv)

```



```

    int argc;
    char *argv[];

{
    /* (mandelbrot definitions ...) */

    /* kTile definitions */
    ktile_s cutnstack;
    ktile_s onedh;
    int cnsid;
    int odhid;
    int mid;
    int ncols;
    int nrows;

    {
        /* (process arguments...) */
    }

    /* (initialize mandelbrot window...) */

    /* Create kTile formats to be used */
    gr_make_2dcs(1, gridsize*nxproc, gridsize*nyproc, &cutnstack);
    gr_make_1dh(1, gridsize*nxproc, gridsize*nyproc, &onedh);

    /* Declare them */
    tstart();
    gr_new_ktile(&cutnstack, &cnsid);
    if (print) tfin("Time for gr_new_ktile(cutnstack)");
    if (print) fprintf(stderr, "cnsid = %d\n", cnsid);

    tstart();
    gr_new_ktile(&onedh, &odhid);
    if (print) tfin("Time for gr_new_ktile(onedh)");
    if (print) fprintf(stderr, "odhid = %d\n", odhid);

    /* Initialize mapping */
    tstart();
    gr_init_remap(cnsid, odhid);
    if (print) tfin("Time for init_remap");

    /* Declare memory */

```

```

tstart();
gr_new_mtag(image[0], gridsize*gridsize, &mid);
if (print) tfin("Time for new_mtag");

/* Loop over all frames to be generated */
for (frameno = 0; frameno < nframes; frameno++)
{
    /* (generate mandelbrot set) */

    /* Write out image */
    if (fn)
    {
        int i, j, k, l;
        int f;

        /* (prepare filename) */

        /* Map 2d cut'n'stack to 1d hierarchical */
        tstart();
        gr_set(mid, cnsid);
        gr_remap(mid, odhid);
        if (print) tfin("Time for remap");

        /* (write file) */
    }

    /* (prepare for next image...) */
}
}

```

Sample run of ex4.m on an MP-1201B

```

garnet% ex4 -print -nframes 2 -gridsize 16 -outfile zit # 512x512 output image
Time for gr_new_ktile(cutnstack) : 0.207896
cnsid = 0
Time for gr_new_ktile(onedh) : 0.003296
odhid = 1

Time for init_remap : 0.410082
Time for new_mtag : 0.006236
Frame #0
Time to calculate Mandelbrot set : 18.262710
[ -2.0000000000000000, -1.5000000000000000]

```

```
[ 3.0000000000000000, 3.0000000000000000]
Time for remap : 0.081849
Frame #1
Time to calculate Mandelbrot set : 38.584498
[ -1.9084009000000000, -1.4770855000000000]
[ 2.8500000000000000, 2.8500000000000000]
Time for remap : 0.010209
garnet%
```

B.4 Some header files

B.4.1 gr.h

The gr module is a high-level interface to underlying remapping routines, for both convenience and to hide the actual remapping mechanism used.

```
/* -----
 * gr
 * -----
 */

/* Declare a new ktile format,
 * returning ktile id if successful.
 * -1 returned if failure occurs because
 * 1. ktile format is invalid
 * 2. ktile format does not fit on current PE array
 * 3. no more ktile identifiers
 */
int gr_new_ktile(
    ktile_s *k,
    int *kid_);

/*
 * Inquire a ktile format,
 * returning 0 if successful.
 * -1 returned if failure occurs because
 * 1. ktile id is invalid
 */
int gr_inquire_ktile(
    int kid,
    ktile_s *ktile);

/* Free a ktile format,
 * returning 0 if successful.
```



```

* -1 returned if failure occurs because
* 1. ktile format is invalid
* 2. an mtag still references kid
* 3. a pair still references kid
*/
int gr_free_ktile(
    int kid);

/*
* Declare a new memory tag,
* returning mtag id if successful.
* -1 returned if failure occurs because
* 1. size must be a power of two
* 2. memory is invalid or overlaps another mtag
* 3. no more mtag identifiers
*
* Initial ktile mapping is -1 (bad).
* You must malloc/declare mem yourself
*/
int gr_new_mtag(
    plural void *mem,
    int size,
    int *mid_);

/*
* Inquire an mtag,
* returning 0 if successful.
* -1 returned if failure occurs because
* 1. mtag id is invalid
*/
int gr_inquire_mtag(
    int mid,
    mtag_s *mtag);

/*
* Free existing mtag id.
* -1 returned if failure occurs
* due to mid being invalid
*/
int gr_free_mtag(
    int mid);

/*
* Generate and remember a pair mapping

```

```

* between ktile formats k1 and k2,
* returning 0 if successful
* -1 returned if failure occurs because
* 1. ktile id is invalid
* 2. no more pair identifiers
* 3. k1 and k2 have incompatible ICS
* 4. there are still a few special cases
* which cannot be dealt with
*/

```

```

int gr_init_remap(
    int k1,
    int k2);

```

```

/*
* Free and remember a pair mapping
* between ktile formats k1 and k2,
* returning 0 if successful
* -1 returned if failure occurs because
* 1. ktile id is invalid
* 2. no pair has been allocated to k1 and k2
*/

```

```

int gr_free_remap(
    int k1,
    int k2);

```

```

/* Set ktile format of mtag mid */

```

```

int gr_set(
    int mid,
    int kid2);

```

```

/* Remap image with mtag mid
* inplace to ktile format kid2,
* returning 0 if successful
* -1 returned if failure occurs because
* 1. mid is invalid
* 2. kid2 is invalid
* 3. mid has no kid attached to it
* 4. kid2 is incompatible with ICS of kid attached to mid
* 5. DCS of kid2 is incompatible with size of mid
*
* If no pair has been declared
* with init_remap, the generated

```

```

* remap will be ephemeral, so unless
* a remap will only be used once,
* it is better to call init_remap first
*/
int gr_remap(
    int mid,
    int kid2);

/* Copy image from mtag mid1 to
* mtag mid2, simultaneously
* remapping it to ktile format kid2
* 0 is returned if successful,
* -1 if failure occurs because
* 1. mid1 is invalid
* 2. mid2 is invalid
* 3. kid2 is invalid
* 4. mid1 has no kid attached to it
* 5. kid2 is incompatible with ICS of kid attached to mid1
* 6. DCS of kid2 is incompatible with size of mid2
*
* If no pair has been declared with
* init_remap, the generated remap
* will be ephemeral, so unless a
* remap will only be used once, it
* is better to call init_remap first
*
* If dimension 0 (memory) of the DCS
* of the kTile attached to mid1 is
* larger than the size of mid2, it is
* possible that data will be written
* past the size of mid2, silently corrupting
* everything! It is safer (but painful)
* to use gr_remap to map the data in mid1
* in-place, then copy the data to mid2,
* the map mid1 back to its original format
*/
int gr_copy(
    int mid1,
    int mid2,
    int kid2);

/* -----
* stdmap

```



```

* -----
*/
/* Given a ktile format,
 * transpose two of its dimensions
 */
int gr_xpose(
    int d1,
    int d2,
    ktile_s *k);

/* Given a ktile format,
 * reverse one of its dimensions
 */
int gr_reverse(
    int d,
    ktile_s *k);

/* Given a ktile format,
 * bitreverse one of its dimensions
 * (prior to an FFT)
 */
int gr_bitrev(
    int d,
    ktile_s *k);

/* Make a ktile format for
 * 2d cut'n'stack mapping for
 * an x by y image with size-byte pixels
 * If x<nxproc or y<nyproc,
 * use only part of the array
 */
int gr_make_2dcs(
    int size,
    int x,
    int y,
    ktile_s *k);

/* Make a ktile format for
 * 2d hierarchical mapping for
 * an x by y image with size-byte pixels
 * If x<nxproc or y<nyproc,
 * use only part of the array
 */

```

```

int gr_make_2dh(
    int size,
    int x,
    int y,
    ktile_s *k);

/* Make a ktile format for
 * 1d cut'n'stack mapping for
 * an x by y image with size-byte pixels
 * If x*y<nproc, use only part of the array
 */
int gr_make_1dcs(
    int size,
    int x,
    int y,
    ktile_s *k);

/* Make a ktile format for
 * 1d hierarchical mapping for
 * an x by y image with size-byte pixels
 * If x*y<nproc, use only part of the array
 */
int gr_make_1dh(
    int size,
    int x,
    int y,
    ktile_s *k);

/* Make a ktile format for
 * 1d scan mapping for
 * an x by y image with size-byte pixels
 * If y>nproc, store multiple scan-lines per PE
 * (equivalent to 1dh)
 */
int gr_make_1dscan(
    int size,
    int x,
    int y,
    ktile_s *k);

#endif

```

B.4.2 Extract from gp2.h

The gp2 module defines the *k*-Tile data structure along with many other associated lower-level data structures, and defines routines for generating index bit permutations from *k*-Tile definitions. Only the relevant parts of this include file are shown.

```

/*
 * Generic coordinate system
 */

typedef struct
{
    int num_dims;
    int dim[0];
} cs_s;

/*
 * kTile format structure
 */

typedef struct
{
    int num_dims;
    int dim[MAX_IDIMS];
} ics_s;

typedef struct
{
    int num_dims;
    int dim[MAX_KDIMS];
} kcs_s;

typedef struct
{
    int num_dims;
    int dim[MAX_DDIMS];
} dcs_s;

typedef struct
{
    int num_dims;
    int dim[MAX_KDIMS];
} map_s;

typedef struct

```



```

{
    ics_s ics;
    kcs_s kcs;
    dcs_s dcs;
    map_s kdmap;
    int ksense;
} ktile_s;

/*
 * Memory tag structure
 */

typedef struct
{
    plural unsigned char *mem;
    int kid;
    int size;
} mtag_s;

/*
 * Example kTile map: 256x256x256 volume on 32x32 array
 * with z-axis and low x, low y in mem
 *
 * With non-log sizes:
 *
 * ics = [256, 256, 256]
 * kcs = [32, 8, 32, 8, 256]
 * dcs = [32, 32, 16384]
 * kdmap = [0, 2, 1, 3, 4]
 * ksense = %000000
 *
 * With log sizes:
 *
 * ics = [8, 8, 8]
 * kcs = [5, 3, 5, 3, 8]
 * dcs = [5, 5, 14]
 * kdmap = [0, 2, 1, 3, 4]
 * ksense = %000000
 *
 */

```

B.5 PMF Implementation notes

PMFs use either the router or the xnet to implement general index bit permutation operations. The xnet is used for general processor/memory transpose operations whenever contiguous Pbits are exchanged with contiguous Mbits, and the router is used for all other inter-processor communication.

The following optimizations ensure that respectable performance can be achieved from the router:

- Index bit permutations are rearranged to ensure that each data element is sent through the router a maximum of once.
- Index bit permutations are rearranged to enable data to be sent in large chunks (up to 64 bits) through the router to minimize latency and the number of opened router connections.
- Router contention is minimized by pre-calculating the order in which router connections are opened.
- In most cases where several processors within a cluster wish to communicate with the same external cluster, unnecessary ropen instructions are not executed. This is not yet performed where processor/memory bit exchanges complicate the issue.
- Most memory accesses use the faster direct addressing, except when processor/memory bit exchanges are required.
- All data movement code is handwritten MasPar assembler, with slow *ampl* versions available for testing purposes

B.6 Using the PMF workbench

A tool called 'gpt' for testing and exploring PMFs has been written. All the internal and application-visible data-types are visible and a large set of commands allows them to be altered and applied. The following examples show the use of this tool for experimentation and delectation.

B.6.1 Declaring a *k*-Tile format

A *k*-Tile format may be declared in up to four ways:

- By directly specifying all fields

```

gpt>ktile k1 [64][8,8][8,8][0,1][++]
gpt>print k1
*** ktile **
ICS: [64]
KCS: [8, 8]
DCS: [8, 8]
KDMAP: [0, 1]
KSENSE: [++]

```

- By initializing to a standard mapping

```

gpt>ktile k2 2dhi 1 256 256
gpt>print k2
*** ktile **
ICS: [1, 256, 256]
KCS: [8, 32, 8, 32]
DCS: [64, 1024]
KDMAP: [0, 2, 1, 3]
KSENSE: [++++]

```

- By applying a transformation to an existing *k*-Tile format

```

gpt>ktile k3 txpose k2 1 2
gpt>print k3
*** ktile **
ICS: [1, 256, 256]
KCS: [8, 32, 8, 32]
DCS: [64, 1024]
KDMAP: [2, 0, 3, 1]
KSENSE: [++++]

```

- By specifying an index-bit permutation in cycle notation. This will be shown later with the pair data type.

B.6.2 The kmap

Attached to every *k*-Tile format is a kmap, which is an internal representation of an index bit map. It may be examined from the *k*-Tile format as follows:

```

gpt>ktile kp [8,8][8,8,2][16,8][2,1,0][+--]
gpt>print kp
*** ktile **
ICS: [8, 8]

```



```

KCS: [8, 8, 2]
DCS: [16, 8]
KDMAP: [2, 1, 0]
KSENSE: [--+]
gpt>print kp.kmap
*** kmap ***
Device sense : [--++] [---]
Enable : [.EEE] [EEE]
ICS->DCS : [p0, p1, p2, m1, m2, m3]
DCS->ICS : [XXX, 3, 4, 5][0, 1, 2]

```

B.6.3 Declaring an mtag

An mtag is declared with a specified amount of memory using the mtag command. Initially, no mapping is attached to the mtag. A mapping may be attached to the mtag using the kset or the kfill commands. The kfill command fills the memory with something meaningful so that it can be checked after a remapping operation.

```

gpt>mtag m 8
gpt>kfill m k1
gpt>print m
*** mtag ***
Size = 8
kTile = k1
gpt>print m.data
0: 0001020304050607
1: 08090a0b0c0d0e0f
2: 1011121314151617
3: 18191a1b1c1d1e1f
4: 2021222324252627
5: 28292a2b2c2d2e2f
6: 3031323334353637
7: 38393a3b3c3d3e3f
gpt>kfill m zap
gpt>print m
*** mtag ***
Size = 8
kTile = NONE

```

B.6.4 Declaring a pair

A pair contains all the information needed to perform remappings using PMFs. It is declared in one of three ways:

- Declaring the pair directly from two *k*-Tile formats:

```
gpt>ktile ka 2dhi 1 64 64
gpt>ktile kb 2dcs 1 64 64
gpt>print ka
*** ktile **
ICS: [1, 64, 64]
KCS: [2, 32, 2, 32]
DCS: [4, 1024]
KDMAP: [0, 2, 1, 3]
KSENSE: [++++]
gpt>print kb
*** ktile **
ICS: [1, 64, 64]
KCS: [32, 2, 32, 2]
DCS: [4, 1024]
KDMAP: [1, 3, 0, 2]
KSENSE: [++++]
gpt>pair p ka kb
gpt>print p
*** pair ***
kid pair = (ka,kb)
```

- Declaring the pair as an index bit permutation with cycle notation. This also creates two *k*-Tile formats generated from the cycles:

```
gpt>cycle pr kra krb [m0,m1,m2,m3,p0,p1,p2,p3][m4][p4~]
gpt>print pr
*** pair ***
kid pair = (kra,krb)
gpt>print kra
*** ktile **
ICS: [512]
KCS: [32, 16, 2]
DCS: [32, 32]
KDMAP: [0, 1, 2]
KSENSE: [++-]
gpt>print krb
```

```

*** ktile **
ICS: [512]
KCS: [8, 2, 2, 8, 2, 2]
DCS: [32, 32]
KDMAP: [4, 0, 2, 1, 3, 5]
KSENSE: [+++++-]

```

- A pair may also be generated automatically when a remapping is attempted between two *k*-Tile formats for which a pair has not yet been initialized.

B.6.5 Data types attached to a pair

Several data types may be seen attached to a pair. These are used in the generation and application of remapping operations:

gmap

The gmap is a representation of an index bit permutation:

```

gpt>print pr.gmap
*** gmap ***
  Pre enabled : [EEEEEE] [EEEE.]
  Pre flipped : [+++++] [++++-]
  Post enabled : [EEEEEE] [EEEE.]
  Post inverted : [+++++] [+++++]
  Forward permutation: [m1, m2, m3, p0, I][p1, p2, p3, m0]
  Reverse permutation: [p3, m0, m1, m2, I][m3, p0, p1, p2]
  cycles --> [m0,m1,m2,m3,p0,p1,p2,p3][m4][p4~]

```

cyc

The cyc is a representation of an index bit permutation with embedded index bit permutation cycle information and transformed into (m^*) , (p^*) , (mp^*) and identity cycles.

```

gpt>print pr.cyc
*** Cyc #0
*** gmap ***
  Pre enabled : [EEEEEE] [EEEE.]
  Pre flipped : [+++++] [++++-]
  Post enabled : [EEEEEE] [EEEE.]
  Post inverted : [+++++] [+++++]
  Forward permutation: [I, I, I, p0, I][p1, p2, p3, m3]

```



```

Reverse permutation: [I, I, I, p3, I][m3, p0, p1, p2]
cycles --> [m0][m1][m2][m3,p0,p1,p2,p3][m4][p4~]
*** MM* cycles ***
*** PP* cycles ***
*** MP* cycles ***
[m3, p0, p1, p2, p3]
*** gmap identities: [EEE.E] [....]
*** gmap MM*/PP* bits: [.....] [....]
*** gmap MP* bits bits: [...E.] [EEEE]
*** Cyc #1
*** gmap ***
    Pre enabled : [EEEEEE] [EEEE.]
    Pre flipped : [+++++] [++++-]
    Post enabled : [EEEEEE] [EEEE.]
    Post inverted : [+++++] [+++++]
Forward permutation: [m1, m2, m3, m0, I][I, I, I, I]
Reverse permutation: [m3, m0, m1, m2, I][I, I, I, I]
cycles --> [m0,m1,m2,m3][m4][p0][p1][p2][p3][p4~]
*** MM* cycles ***
[m0, m1, m2, m3]
*** PP* cycles ***
*** MP* cycles ***
*** gmap identities: [....E] [EEEE]
*** gmap MM*/PP* bits: [EEEE.] [....]
*** gmap MP* bits bits: [.....] [....]

```

remap

The remap contains all the information needed to perform an optimal radix 2 remapping. The following fields are printed:

Remap type One of (*mm**), (*pp**), (*mp**) or *tx*

Steprange An array of pairs of steps/ranges for identity mbits

ipsfunc Function to be called from steprange loop when performing in-place remappings

cpsfunc Function to be called from steprange loop when performing copy remappings

Pvalid Valid destination processors

memory copies Linked list representing memory permutation for copy remap

memory swaps Linked list representing memory permutation cycles for in-place remap

Base rx Base source processor from which to fetch data

rorder Order of processor fetching for cluster contention removal

num_routes Number of transmissions required for permutation

torder Order of processor transmission for cluster contention removal

num_cluster Number of transmissions possible per ropen

roufunc Function to call to perform router operations

Base mx Base memory offset calculated from (*mp**) parity masks

(*mp**) **masks** Array of xor masks for recursive parity masking

iprfunc Function to call after recursive parity masking for in-place remapping

cprfunc Function to call after recursive parity masking for copy remapping

```
gpt>print pr.remap
```

```
*** Remap #0
```

```
----- remap -----
```

```
mp* remap.
```

```
Steprange:
```

```
[16,32]
```

```
ipsfunc = op2_ip_routes()
```

```
cpsfunc = op2_cp_routes()
```

```
Pvalid:
```

```
.....XXXXXXXXXXXXXXXXX
```

```
Base rx:
```

```
***   XXX   XXX   XXX   XXX   XXX   XXX   XXX   XXX
```

```
      XXX   XXX   XXX   XXX   XXX   XXX   XXX   XXX
```

```
        16    24    25    17    26    18    19    27
```

```
        28    20    21    29    22    30    31    23
```

```
rorder (num_routes = 4):
```

```
.....0213021302130213
```

```
torder (num_cluster = 1):
```

```
<none>
```

```
roufunc = op2_2rc0_64()
```

```
Base mx:
```

```

***   XXX   XXX   XXX   XXX   XXX   XXX   XXX   XXX
      XXX   XXX   XXX   XXX   XXX   XXX   XXX   XXX
          0     8     8     0     8     0     0     8
          8     0     0     8     0     8     8     0

```

mp* masks:

[8,8]

iprfunc = op2_ip_ix_64()

cprfunc = op2_cp_ix_64()

*** Remap #1

----- remap -----

mm* remap.

Steprange:

[16,32]

ipsfunc = op2_ip_mi_08()

cpsfunc = op2_cp_mi_08()

Pvalid:

.....XXXXXXXXXXXXXXXXXX

memory copies

0-0 1-2 2-4 3-6 4-8 5-10 6-12 7-14 8-1 9-3 10-5 11-7

12-9 13-11 14-13 15-15

memory swap cycles

* 1 8 4 2 * 3 9 12 6 * 5 10 * 7 11 13 14

gpt>

B.6.6 Performing a remap

We have now defined all the commands necessary for defining *k*-Tile formats, memory and remappings. We can now use them for performing a remapping; the following example remaps the memory referenced by mtag *m* from the *k*-Tile format *kra* to *krb*:

gpt>mtag m 32

gpt>kfill m kra

gpt>ip m krb

gpt>kchk m

Checked out OK.

B.6.7 Timing

To determine how efficient the PMF system is at performing a remapping, many of the most expensive PMF operations may be timed:


```

gpt>time 1
Timing = 1
gpt>ip m kra
new pair id = p000
Time to declare gmap : 0.003583
Time to declare cycles : 0.009387
Time to declare remaps : 0.068689
Time to perform inplace remap : 0.001125
gpt>ip m krb
Time to perform inplace remap : 0.001134
gpt>ip m kra
Time to perform inplace remap : 0.001122
gpt>kchk m
Checked out OK.

```

Appendix C

Glossary of symbols

Logical relations

$a \Rightarrow b$	a implies b ; b if a ; a only if b
$a \triangleq b$	a is defined to be b
$a \equiv b$	a is equivalent to b
$a \text{ iff } b$	a if and only if b
$\exists a$	There exists an a
$\forall a$	For all a

General operators

$a \oplus b$	a exclusive-or b
Π	Product
Σ	Sum

Sets

$\{a, b, c\}$	A set containing a , b and c
\emptyset	The <i>empty set</i> , $\{\}$
$a \in B$	a is an element of B
$a \notin B$	a is not an element of B
$B \setminus a$	B without a
$A \cup B$	A union B
$A \cap B$	A intersection B
$A \subset B$	A is a subset of B

$\{a \in X : P(a)\}$ The subset of the set X satisfying the property P

$|A|$ The *cardinality*, *size*, or number of elements in the set A

Functions

$f : X \rightarrow Y$ A function f assigning a value from Y to every element in X

$\text{domain}(f)$ The set of values for which f is defined

$\text{range}(f)$ The set of values f maps to

f^{-1} The inverse function of f

one to one f is *one to one* iff $f(a)$ is different for every value of a

onto A function f is *onto* a set Y iff $\text{range}(f) = Y$

$f \circ g$ f composed with g ; $f \circ g(x) = f(g(x))$

Multidimensional arrays

p, q, r Integers used for defining dimensionality

\mathbf{a} A general shape vector

$\langle \mathbf{a} \rangle$ The *dimensionality* of \mathbf{a}

$\text{index}(\mathbf{a})$ The *index space* of \mathbf{a}

$[\mathbf{A}]$ The shape of the multidimensional array \mathbf{A}

$[\mathbf{A}]_i$ The length of dimension i of \mathbf{A}

\perp 'perp', the undefined data value or array index

$|\mathbf{A}|$ The *cardinality* or *size* of \mathbf{A}

General spaces

\mathbb{R} The set of all real numbers

\mathbb{N} The set of all non-negative integers, including zero

\mathbb{N}^q The set of all \mathbb{N} -vectors of length q

\mathbb{T} A set representing an arbitrary data type

\mathbb{M} The set of all multidimensional arrays

$\mathbb{M}_{\mathbb{T}}$ The set of all multidimensional arrays with data type \mathbb{T}

$\mathbb{M}_{\mathbf{a}}$ The set of all multidimensional arrays with shape \mathbf{a}

***k*-Tile format**

a	The data space shape
k	The k -Tile space shape
d	The device space shape
A	The data array index space
K	The k -Tile array index space
D	The device array index space
m	The mapping vector assigning k -Tile dimensions
c	The computed vector defining k -Tile dimension assignment
s	The k -Tile sense indicator
t_D, t_K, t_A	Template space shapes
T_D, T_K, T_A	Template index spaces
o_D, o_K, o_A	Space offsets
$o_{T_D}, o_{T_K}, o_{T_A}$	Template space offsets
f_K	k -Tile format data mapping
g_K	k -Tile index format index mapping
$g_{T_D}, g_{T_K}, g_{T_A}$	Offset template index mappings
g_{DT}	Inverse sense k -Tile mapping from D to T_K
g_{KT}	Implicit k -Tile mapping from K to T_A

Radix 2 remapping

$\ A\ $	Log size of $A = \log_2 A $
a	General data array index
a	General device array index
$a_q \dots a_0$	Binary representation of a
A	Index space of 1d data array
D	Index space of 1d device array
M	Memory address index space

$\begin{pmatrix} 0 & 1 & 2 \\ 2 & 1 & 0 \end{pmatrix}$	Sample permutation in direct notation
$(02)(1)$	Sample permutation in cycle notation
P	Processor number index space, also a general permutation mapping
π	General permutation
\mathcal{P}	General index bit permutation
s	General inversion mask
S	General index bit inversion
$1a_0\bar{a}_1a_2$	Sample index bit map
E	Enabled subset of device indices
E_s	Enabled subset of device indices in source mapping
E_d	Enabled subset of device indices in destination mapping
$(d_0, \bar{d}_1)(d_0.)$	Sample radix 2 remapping
(m)	Arbitrary m -bit identity cycle
(p)	Arbitrary p -bit identity cycle
$(m*)$	m -bit permutation cycle
$(p*)$	p -bit permutation cycle
$(mp*)$	Permutation cycle with one m -bit, multiple p -bits