Parallel Algorithms for Linear Algebra

Richard P. Brent

Computer Sciences Laboratory

Australian National University

Outline

- Basic concepts
- Parallel architectures
- Practical design issues
- Programming styles
- Data movement and data distribution
- Solution of dense linear systems and least squares problems
- Solution of large sparse linear systems
- SVD and eigenvalue problems

Speedup and Efficiency

T_P = time to solve problem on a parallel machine with P processors

T_1 = time to solve problem on a single processor

$S = T_1/T_P$ is the speedup

$E = S/P$ is the efficiency

Usually $S \le P$ and $E \le 1$.
For effective use of a parallel machine we want S close to P and E close to 1.

Number of Processors and Problem Size

If we take a problem of fixed size N (for example, solving a system of N linear equations), then the efficiency $E \to 0$ as the number of processors $P \to \infty$ (Amdahl's Law).

However, for many classes of problems we can keep the efficiency $E > 0.5$ provided $N \to \infty$ as $P \to \infty$
This is reasonable because parallel machines are intended to solve large problems.

Parallel Computer Architectures

There are many different computer architectures.

We briefly consider the most important categories.

SIMD or MIMD

**Single Instruction Stream,
Multiple Data Stream.
All processors execute the same instructions on their local data (unless turned off),
e.g. CM1, CM2, Maspar.**

**Multiple Instruction Stream,
Multiple Data Stream.
Different processors may execute independent instructions and need not be synchronized.
e.g. AP 1000, CM5, Intel Delta.**

Local or Shared Memory

**Local memory is only accessible to one processor.
Shared (or global) memory is accessible to several (or all) processors.**

Usually local memory is cheaper than shared memory and access to it is faster.

Examples -

AP 1000	**-**	**local memory.**
VP 2600	**-**	**shared memory.**
VPP 500	**-**	**local and shared.**

Connection Topology

The processors may be physically connected in several ways - ring, grid, torus, 3D grid or torus, tree, hypercube, etc.

The connection topology may be visible or invisible to the programmer. If invisible then the programmer can assume that all processors are connected, because routing by hardware and/or system software is automatic and reasonably efficient.

Message Routing

On local memory machines, messages may be routed in several ways, such as -

Store and forward.
High latency (proportional to number of hops x message size). e.g. early hypercubes.

Wormhole routing (or similar).
Low latency (proportional to number of hops + message size). e.g. Fujitsu AP 1000.

Other Design Issues

Power of each processor versus number of processors. Should each processor have a floating-point unit, a vector unit, or several vector units ? (Extremes - CM1 and CM5).

Should a processor be able to emulate several virtual processors (convenient for portability and programming) ?

Should multiple users be handled by time-sharing or space-sharing (or both) ?

Programming Styles

Data-Parallel Programming -
Based on the SIMD model so popular on SIMD machines with languages such as C*. May be emulated on MIMD machines (e.g. CM5, AP 1000).

Single Program Multiple Data (SPMD) - often used on MIMD machines such as the
AP 1000. Sometimes one program is special, e.g. host program on AP 1000, or controller of "worker farm".

Message Passing Semantics

A program for a local-memory MIMD machine has to explicitly send and receive messages. Typically the sender provides the message and a destination (a real or virtual processor ID, sometimes also a task ID).

The send may be blocking (control returns only after message transmission complete) or nonblocking.

Semantics of Receive

The receiver typically calls a system routine to find out if a message has arrived. There may be restrictions on the type or source of the message.

Again, there are blocking and nonblocking versions.

Sometimes the message is copied immediately into the user’s area, sometimes this requires another system call.

Efficiency of Message Passing

For fast message passing, context switching and message copying should be minimized. On the AP 1000, the "synchronous" message passing routines (xy_send etc) avoid context switching.

"Active" messages contain a destination address in their header, so they can be transferred directly into the user address space, avoiding copying overheads.

Avoiding Explicit Message Passing

On shared memory machines the programmer does not have to worry about the semantics of message passing. However, synchronization is required to make sure that data is available before it is used.

Data-parallel languages remove the need for explicit message passing by the programmer. The compiler does whatever is necessary.

Broadcast and Combine

Broadcast of data from one processor to others, and combination of data on several processors (using an associative operator such as addition or concatenation) are very often required.

For example, we may wish to sum several vectors or the elements of one vector, or to find the maximum element of a vector, where the vectors are distributed over several local memories.

Efficiency of Broadcast and Combine

With hardware support, the broadcast and combine operations can be about as fast as simple processor to processor message passing.

Without hardware support they require a binary tree of communications (implemented by software) so are typically more expensive by a factor

$O(\log P)$

Aims of Numerical Computation on Parallel Computers

Speed (else why bother with parallel algorithms ?)

Numerical stability

Efficiency

Simplicity, generality etc

We illustrate with some examples involving linear algebra.

Typical Numerical Linear Algebra Problems

Solution of dense linear systems by Gaussian elimin-ation with partial pivoting (LINPACK Benchmark).

Solution of least squares problems by QR factorization.

Solution of large sparse linear systems by direct or iterative methods.

Eigenvalue and SVD problems.

The LINPACK Benchmark

A popular benchmark for floating-point performance.

Involves the solution of a nonsingular system of n equations in n unknowns by Gaussian elimination with partial pivoting.
(Pivoting is generally necessary for numerical stability).

Three Cases

n = 100

The original benchmark
(too easy for our purposes).

n = 1000

Often used to compare vector processors and parallel computers.

$n \gg 1000$

Often used to compare massively parallel computers.

Assumptions

Assume double-precision arithmetic (64-bit).

Interested in $n \geq 1000$.

Assume coefficient matrix available in processors.

Use C indexing conventions -

Indices 0, 1, ...
Row-major ordering

Results are for the AP 1000

Hardware

The *Fujitsu AP 1000* (also known as the *CAP II)* is a MIMD machine with up to 1024 independent 25 Mhz Sparc processors (called *cells*).

Each cell has 16 MB RAM, 128 KB cache, and Weitek floating-point unit capable of 5.56 Mflop for overlapped multiply and add.

Communication

The topology of the AP 1000 is a torus with wormhole routing. The theoretical bandwidth between any pair of cells is 25 MB/sec.

In practice, because of system overheads, copying of buffers, etc, about 6 MB/sec is attainable by user programs.

Data Distribution

Ways of storing matrices (data and results) on a local memory MIMD machine -

- column wrapped
- row wrapped
- scattered = torus wrapped = row and column wrapped
- blocked versions of these

We chose the *scattered* representation because of its good load-balancing and communication bandwidth properties.

Scattered Storage

On a 2 by 2 configuration

cell cell
cell cell

a 4 by 6 matrix would be stored as follows, where the color-coding indicates the cell where an element is stored -

```
00 01 02 03 04 05
10 11 12 13 14 15
20 21 22 23 24 25
30 31 32 33 34 35
```

Scattered Storage - Global $\leftrightarrow$ Local Mapping

On a machine configuration with *ncelx . ncely* cells *(x, y)*, $0 \le x < ncelx$, $0 \le y < ncely$, element $a_{i,j}$ is stored in cell

$$(j \bmod ncelx,\ i \bmod ncely)$$

with local indices[1]

$$i' = i \operatorname{div} ncely,$$
$$j' = j \operatorname{div} ncelx.$$

[1]Sorry about the confusing *(i,j)* and *(y,x)* conventions !

Blocked Storage

If the above definition of scattered storage is applied to a block matrix with *b* by *b* blocks, then we get the *blocked panel-wrapped* (blocked torus-wrapped) representation. Choosing larger *b* reduces the number of communication steps but worsens the load balance.

We use *b = 1* on the AP 1000, but *b > 1* has been used on other local-memory machines (e.g. Intel Delta).

Blocked Matrix Operations

The rank-1 updates in Gaussian elimination can be grouped into blocks of ω so rank-ω updates can be performed using level 3 BLAS (i.e. matrix-matrix operations).

The two possible forms of blocking are independent - we can have $b > 1$ or $\omega > 1$ or both. If both then $b = \omega$ is convenient but not necessary. In our implementation

$$b = 1, \quad \omega \geq 1.$$

Gaussian Elimination

The idea of Gaussian Elimination (G.E.) is to transform a nonsingular linear system

$$Ax = b$$

into an equivalent upper triangular system

$$Ux = b'$$

which is (relatively) easy to solve for x. It is also called LU Factorization because

$$PA = LU,$$

where P is a permutation matrix and L is lower triangular.

A Typical Step of G.E.

$$\begin{array}{ccccccc}
x & x & x & x & x & x & x \\
 & x & x & x & x & x & x \\
 & & x & x & x & x & x \\
 & & x & x & x & x & x \\
 & & x & x & x & x & x \\
 & & x & x & x & x & x
\end{array}$$

by row operations $\Rightarrow$

$$\begin{array}{ccccccc}
x & x & x & x & x & x & x \\
 & x & x & x & x & x & x \\
 & & x & x & x & x & x \\
 & & 0 & x' & x' & x' & x' \\
 & & 0 & x' & x' & x' & x' \\
 & & 0 & x' & x' & x' & x'
\end{array}$$

Comments

x is a nonzero element,
x is the pivot element,
x is an element to be zeroed,
x is in the pivot row,
$x \rightarrow x'$ is in the active region.

Row interchanges are generally necessary to bring the pivot element x into the correct position.

The right-hand side vector has been stored as the last column of the (augmented) matrix.

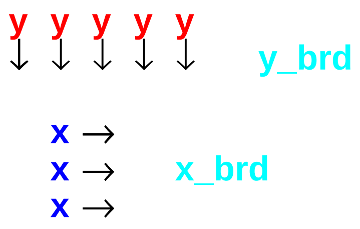
Communication Requirements for G.E.

Pivot selection requires finding the largest element in (part of) a column;
then, if necessary, two rows are interchanged. We do this explicitly to avoid later load-balancing problems.

The rank-1 update requires vertical broadcast (y_brd) of the pivot row and horizontal broadcast (x_brd) of the multiplier column.

x_brd and y_brd

The AP 1000 has hardware support for x_brd and y_brd, so these can be performed in the same time as a single cell to cell communication.
(A binary tree with *O(*log *P)* communication overheads is not required.)



Memory Refs per Flop

The ratio

$$R = (\text{loads and stores})/(\text{flops})$$

is important because it is impossible to keep the floating-point unit busy unless $R < 1$. Rank-1 updates

$$a_{ij} \leftarrow a_{ij} + u_i \times v_j$$

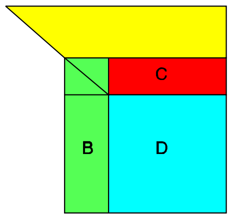
have $R \geq 1$. To reduce R and improve performance, need blocking. (ω rank-1 updates $\rightarrow$ one rank-ω update.)

G.E. with Blocking

Defer operations on the region labelled D until ω steps of G.E. have been performed. Then the rank-ω update is simply

$$D \leftarrow D - BC$$

and can be performed by level-3 BLAS without inter-cell communication.



Choice of ω

Operations in the vertical strip of width ω and the horizontal strip of depth ω are done using rank-1 updates (slow) so want ω to be small. However, level-3 BLAS for rank-ω updates are slow unless ω is large. The optimum choice is usually

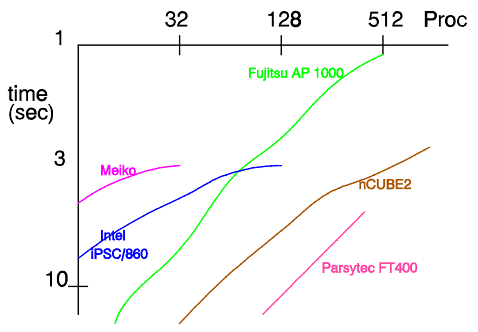
$$\omega \sim n^{1/2}$$

However, ω should be small enough that the parts of B and C stored in each cell are smaller than the cache size.

LINPACK Benchmark Results *(n = 1000)* on the AP 1000

cells	time (sec)	speedup	efficiency
512	1.10	147	0.29
256	1.50	108	0.42
128	2.42	66.5	0.52
64	3.51	46.0	0.72
32	6.71	24.0	0.75
16	11.5	13.9	0.87
8	22.6	7.12	0.89
4	41.3	3.90	0.97
2	81.4	1.98	0.99
1	160	1.00	1.00

Comparison for *n = 1000* using Dongarra's Table 2



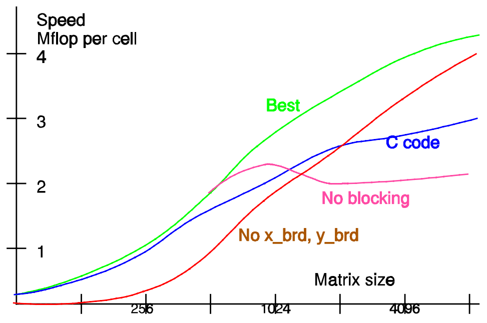
The AP 1000 is fastest for $\geq$ 128 processors and shows little "tailoff" as their number $\uparrow$

LINPACK Benchmark Results *(n large)* on the AP 1000

cells	r_{max} Gflop	n_{max}	n_{half}	r_{max}/r_{peak}
512	2.251	25600	2500	0.79
256	1.162	18000	1600	0.82
128	0.566	12800	1100	0.80
64	0.291	10000	648	0.82
32	0.143	7000	520	0.80
16	0.073	5000	320	0.82

Note the high ratio r_{max}/r_{peak} and the large ratio n_{max}/n_{half}

Comparison of Options on 64-cell AP 1000



The graph shows the effect of turning off blocking, hardware x_brd, y_brd, or assembler BLAS 3 inner loops.

Conclusions from LINPACK Benchmark

The AP 1000 is a well-balanced machine for linear algebra. We can attain at least 50% of peak performance over a wide range of problem sizes.

The communication speed is high and startup costs are low relative to the floating-point speed[2]. Hardware support for x- and y-broadcast is an excellent feature.

[2]Floating-point is slow by current standards

QR Factorization

Gaussian elimination gives an triangular factorization of the matrix A. For some purposes an orthogonal factorization $A = QR$ is preferable, although it requires more arithmetic operations.

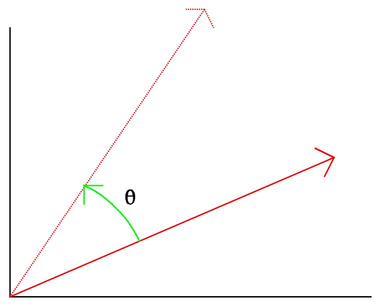
The QR factorization is more stable, does not usually require pivoting, and can be used to solve over-determined linear systems by the method of least squares.

Parallel Algorithms for QR Factorization

The idea is to transform the input matrix A to the upper triangular matrix R using simple orthogonal trans-formations which may be accumulated to give Q (if it is needed).

The simple orthogonal transformations are usually plane rotations (Givens transformations) or elementary reflectors (Householder transformations).

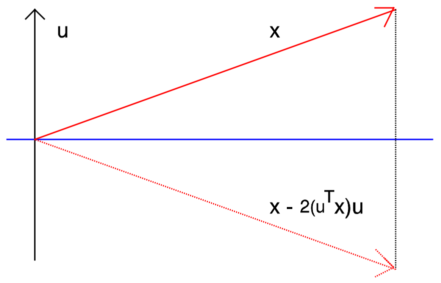
Plane Rotations



Represented by the matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

Elementary Reflectors



Represented by the matrix

$$I - 2uu^T$$

where u is a unit vector.

Application of Plane Rotations

Plane rotations are numerically stable, use local data, and can be implemented on a systolic array or SIMD machine.

They involve more multiplications than additions, unless the "Fast Givens" variant is used. Thus, the speed in Mflop may be significantly less than for the LINPACK Benchmark, and there are more operations.

Application of Elementary Reflectors

Elementary reflectors are suitable for vector processors and MIMD machines. They may be grouped into blocks via the

$$I - WY$$

representation (or variants) to make use of level-3 BLAS.

The communication requirements and speed in Mflop are close to those of the LINPACK Benchmark.

Large Sparse Linear Systems

Large linear systems which arise in practice are more often sparse than dense. By "sparse" we mean that most of the coefficients are zero.

For example, large sparse linear systems arise in the solution of structural problems by finite elements,
the solution of partial differential equations (PDEs),
and optimization problems (linear programming).

Direct Methods for Large Sparse Systems

For structured sparse systems (e.g. band, block tridiagonal) the methods used for dense linear systems can be adapted.

For example, the scattered storage representation is good for band matrices provided the bandwidth w is at least sqrt(P)

Iterative Methods for Large Sparse Systems

Direct methods are often impractical for very large, sparse linear systems, so iterative methods are necessary. Most iterative methods involve -

Preconditioning - problem-dependent, aims to increase the speed of convergence.

Iteration by a recurrence involving matrix x vector multiplications.

Sparse Matrix x Vector Multiplication

The key to fast parallel solution of sparse linear systems by iterative methods is an efficient implementation of matrix x vector multiplication. This is not so easy as it sounds !

Consider storing sparse rows (or columns) on each cell,
or using scattered storage.
Communication of some sort is unavoidable (e.g. for vector sums or combine operations).

The Symmetric Eigenvalue Problem

The eigenvalue problem for a given symmetric matrix A is to find an orthogonal matrix Q and diagonal matrix Λ such that

$$Q^TAQ = \Lambda$$

The columns of Q are the eigenvectors of A, and the diagonal elements of Λ are the eigenvalues.

The Singular Value Decomposition (SVD)

The singular value decomposition of a real m by n matrix A is its factorization into the product of three matrices

$$A = U\Sigma V^T$$

where U and V have orthonormal columns, and Σ is a nonnegative diagonal matrix. The n diagonal elements of Σ are called the singular values of A. (We assume $m \geq n$)

Connection Between the SVD and Symmetric Eigenvalue Problems

The eigenvalues of A^TA are just the squares of the singular values of A.

However, it is not usually recommended that singular values be computed in this way, because forming A^TA squares the condition number of the problem and causes numerical difficulties if this condition number is large.

The Golub-Kahan-Reinsch-Chan Algorithm for the SVD

On a serial computer the SVD is usually computed by a QR factorization, followed by reduction of R to bidiagonal form by a two-sided orthogonalization process. The singular values are then found by an iterative method.

This process is complicated and difficult to implement on a parallel computer.

A Parallel Algorithm for the SVD

The 1-sided orthogonalization algorithm of Hestenes is simple and easy to implement on parallel computers. The idea is to generate an orthogonal matrix V such that AV has orthogonal columns. The SVD is then easily found.

V is found by an iterative process. Pairs of columns are orthogonalized using plane rotations, and V is built up as the product of the rotations.

Parallel Sweeps

For convergence we must orthogonalize each pair (i, j) of columns, $1 \le i < j \le n.$ This is a total of $N = n(n-1)/2$ pairs, called a sweep. Several[3] sweeps are necessary.

A parallel algorithm is based on the fact that $\lfloor n/2 \rfloor$ pairs of columns can be processed simultaneously, using $O(n)$ processors which need only be connected in a ring.

[3] $O(\log n)$?

Parallel Algorithms for the Eigenvalue Problem

The Jacobi algorithm for the symmetric eigenvalue problem is closely related to the Hestenes SVD algorithm.[4] The significant differences are that rotations are chosen to zero off-diagonal elements of A, and are applied on both sides. $2\lfloor n/2 \rfloor$ off-diagonal elements can be zeroed simultaneously, using $O(n^2)$ processors.

[4] Recall the relationship between the SVD and eigenvalue problems.

The Hestenes and Jacobi Algorithms on the AP 1000

The Jacobi and Hestenes algorithms are ideal for systolic arrays, but not for machines like the AP 1000, because they involve too much communication.

For good performance on the AP 1000 it is necessary to group blocks of adjacent columns (and rows) to reduce communication requirements.

Other Parallel Eigenvalue Algorithms

Dongarra & Sorensen suggest a "divide and conquer" algorithm which is attractive on shared-memory computers.

They first reduce *A* to tridiagonal form using a parallel version of the usual algorithm, then split the tridiagonal matrix into two pieces by making a rank-1 modification whose effect can later be reversed. The splitting can be repeated recursively.

An Alternative - Sturm Sequences

The rank-1 modification in the Dongarra-Sorensen algorithm causes numerical difficulties. An alternative is to find all the eigenvalues of the tridiagonal matrix in parallel, using the Sturm sequence method to isolate each eigenvalue.

If required, the eigenvectors can then be found in parallel using inverse iteration. It is not clear how to guarantee orthogonality in all cases.

Conclusions

- **Parallel computers provide many new opportunities and challenges.**
- **Good serial algorithms do not always give good parallel algorithms.**
- **The best parallel algorithm for a problem may depend on the machine architecture.**
- **Similar observations apply to non-numerical problems.**