Design of the Scientific Subroutine Library for the Fujitsu VPP300

R. Brent, L. Grosz, D. Harrar II,
M. Hegland, M. Kahn,G. Keating,
G. Mercer, M. Osborne, B. Zhou

Australian National University
Canberrra ACT, 0200, Australia

M. Nakanishi

High Performance Computing Group,
Fujitsu Ltd.
Numazu-Shi Shizuoka, 410-03, Japan

Abstract

A research agreement between the Australian National University and Fujitsu Japan has led to the development of a library of parallel mathematical subroutines and the extension of the library of single processor routines for the Fujitsu VPP300. The Fujitsu VPP300 provides a very sophisticated architecture combining vector processors in parallel by a crossbar switch. Very high performance can be obtained if carefully designed algorithms are used which exploit the full performance of the hardware. New algorithms have been developed and performance results are provided for eigenvalue problems, linear solvers and fast Fourier and wavelet transforms.

1. Introduction

With 13 vector processors, the Fujitsu VPP300 at the Australian National University has a peak spead of 29 Gflop/s and a main memory of 14 Gbytes. Thus the VPP300 is ideally suited to processing very large and demanding simulation and data analysis tasks. Each processor has a 7 ns cycle time and contains

- A scalar unit (SU) which has a long-instruction-word (LIW) RISC CPU with a 100Mflop/s peak performance. This unit simultaneously controls scalar, vector and data transfer instructions.
- A vector unit (VU) with 1 load, 1 store, 1 add, 1 multiply, 1 divide pipe each completing 8 operations per cycle (except divide which does 8 operations in 7 cycles). It is also possible to chain load-mult-add to get a peak performance of 2.2 Gflop/s (actual matrix multiply implementations achieve 2.18Gflop/s). The vector unit also contains 128 kB of register space which can be configured as 256 registers with 64 words (a 64 bits) down to 8 registers with 2048 words each.
- Memory (MSU) with 512MB of SDRAM memory on 8 of the PEs and 2GB on the remaining 5 PEs.
- Data transfer unit (DTU) for direct memory access data communication to the interprocessor network.

The basic challenge of programming the Fujitsu VPP300 is that both the memory and the processing elements have a hierarchical nature. The memory hierarchy with fast registers, local and global memory is now well understood, and programming models based on data locality are well established. On the lowest levels the compilers are able to handle registers and local memory while the programmer deals with the global data access using either message passing or data parallelism. The second hierarchy of the processing elements is less well understood. Here one has at the lowest level the segments of the pipelines which all run in parallel. On the next level there are the processors which consist of a collection of pipelines. The outermost level is the multiprocessing system. Distribution of computational tasks between these levels whilst maintaining a good load balance is a formidable challenge and vector-parallel algorithms which can exploit parallelism on all levels are few.

For this project the Fujitsu VPP300 has been programmed in VPP Fortran. This extension of Fortran 90 provides a concept of global memory which allows access to the DTU and thus to the memory of other processors. In addition, VPP Fortran provides directives to specify task parallelism and data distribution.

Thus a subroutine library, known as SSLIIVPP, in which the most frequent computational tasks are implemented is of great importance for the user who wishes to get the best performance from the Fujitsu VPP300 using their own programs. In addition, the subroutine library contains a collection of implementation ideas which can be made accessible to the user through publications and the SSLIIVPP library manual [12]. These implementation ideas provide an implicit basic model for vector-parallel programming. We would like to see the following exposition as such a collection of implementation ideas.

The remaining sections contain a description of parallel-vector matrix multiplication, and based on this, the solution of dense linear systems. It is seen that both operations are scalable achieving over 95 percent efficiency for the matrix products and over 80 percent efficiency for linear solvers. Similar high performance is achieved for the fast Fourier transforms and wavelets. The key here is the development of new algorithms with very long vector lengths, stride one data access and separate communication and computation. Multi-level techniques for the iterative solution of sparse linear systems provide the basis for scalable solvers. Here, as for many other algorithms, the parallelism of the algorithm is shared by vector units and distributed across processors. In many cases, especially 2D problems, direct solvers can be very competitive compared with iterative solvers. A major issue is the exploitation of parallelism and vectorisation while keeping fill-in low. In the final section we deal with eigenvalue solvers. The algorithms discussed here are an example of mapping different levels of parallelism to the different levels of the hierarchy. The different eigenvalues provide a natural parallelism where vectorisation is implemented in the basic tridiagonal solvers which need to be both fast and stable.

Further information of the Fujitsu VPP library developed jointly by Fujitsu Japan and the ANU can be found in [7, 6] and

`http://anusf.anu.edu.au/Area4_Working_Notes`

2. Matrix–Matrix Multiplication

Matrix multiplication is an essential part of any dense linear algebra library. The SSLII parallel matrix multiplication routine achieves completely scalable performance by overlapping the required data transfer between processors with the computation.

Let the $n \times n$ matrices A, B and C be partitioned as $A = (A_1, A_2, ..., A_p)$, $B = (B_1, B_2, ..., B_p) = (B_{ij})$, $C = (C_1, C_2, ..., C_p)$

where $B_i = (B_{1i}^T, B_{2i}^T, ..., B_{pi}^T)^T$

and p is the number of processors. The matrices are distributed such that A_i, B_i and C_i are allocated to the i-th processor.

Then $C = A \times B$ is calculated as

$C_i = \sum_{j=1}^{p} A_j B_{ji}$

On each processor the computation of $C_i = C_i + A_j B_{ji}$ is done concurrently. The different A_i on each processor can be transferred to the adjacent processor through the crossbar network until all the $A_i B_{ji}$ are summed. These data transfers can be overlapped with the computation with the use of an additional small work area [12]. Scalability is achieved as shown in Table 1.

#PE	Order	Gflops	Ratio
2	10,000	4.325	0.98
4	10,000	8.636	0.98
6	10,000	12.92	0.98
8	10,000	17.19	0.97
10	10,000	21.53	0.98
15	10,000	31.62	0.95
20	10,000	42.42	0.96
25	10,000	52.27	0.95

Table 1: Performance of Matrix Multiplication
Ratio is observed Gflops to theoretical peak ($2.2Gflops \times \#PE$)

3. Dense Linear Systems

A linear equation solver which achieves high performance on the VPP300 has been reported on in [26]. The high performance of this solver, based on an outer-product algorithm, is due to the cyclic column block layout of the data which simplifies communication, allows for high vectorisation rates and facilitates overlapping of communication and computation. The columnwise block cyclic layout also achieves internal load balance when executed in parallel. The data partitioning is dynamically changed within the routine from the input easy-to-use banded partition to the columnwise block cyclic partition by parallel data transfer.

To factorize an $n \times n$ matrix A as $A = LU$ a recursive algorithm is used such that, after the k-th block is decomposed into the LU decomposition as $(L_1^{(k)T}, L_2^{(k)T})^T U_1^{(k)}$ and the corresponding rowwise block is updated by $U_2^{(k)} = (L_1^{(k)})^{-1} U_2^{(k)}$, the submatrix is updated in outer product fashion as follows. p is the number of processors.

$A^{(k)} = A^{(k)} - L_2^{(k)} U_2^{(k)}$

$L_2^{(k)}$ is located on a processor and this block is partitioned into m pieces.

$L_2^{(k)} = (L_{12}^{(k)T}, L_{22}^{(k)T}, ..., L_{m2}^{(k)T})^T$

$U_2^{(k)}$ and $A^{(k)}$ are redefined combining the parts on each processor regardless of the cyclic layout.

$U_2^{(k)} = (U_{21}^{(k)}, U_{22}^{(k)}, ..., U_{2p}^{(k)})$,

$A^{(k)} = (A_1^{(k)}, A_2^{(k)}, ..., A_p^{(k)})$
$A^{(k)}$ has $m \times p$ block structure.
$A^{(k)} = (A_{ij}^{(k)})$

Then $A_{ij}^{(k)}$ is computed in parallel on each processor using
$A_{ij}^{(k)} = A_{ij}^{(k)} - L_{i2}^{(k)} U_{2j}^{(k)}$

At first $L_{i2}^{(k)}$ is distributed to each processor by data transfer and then $L_{i2}^{(k)}$ is transferred to adjacent processor along the ring to compute the next update concurrently. This data transfer can be hidden behind the computation.

This method reduces the total amount of data transfer to about

$$\frac{1.5}{\log_2(n)}$$

of that required by broadcasting blocks of data.

Forward substitution and back substitution are parallelized after changing the columnwise banded distribution into rowwise banded distribution.

Performance data in Table 2 and Table 3 show the scalability of the linear equation solver.

The linear equation solver for a symmetric positive definite matrix can be parallelized in a similar fashion [26].

#PE	Order	Gflops	Ratio
2	15,000	3.788	0.86
4	20,000	7.562	0.86
6	25,000	11.26	0.85
8	30,000	15.02	0.85
10	32,000	18.65	0.85
15	37,000	27.33	0.82
20	42,000	36.12	0.82
25	47,500	44.89	0.81

Table 2: Performance of Linear Equation Solver. Ratio is observed Gflops to theoretical peak ($2.2Gflops \times \#PE$)

To calculate the inverse of a dense, real matrix , the Gauss-Jordan method is parallelized. In this method each column vector is eliminated and then the original matrix is reduced to a unit matrix. These operations are accumulated on the unit matrix simultaneously and finally produce the inverse of matrix. The array for the storage of matrix is equally partitioned in the second dimension.

The accumulation changes only the left part of matrix where the column vectors have already been eliminated and elimination and accumulation are done concurrently.

#PE	Order	Gflops	Ratio
2	10,000	3.689	0.84
4	10,000	7.145	0.81
6	10,000	10.22	0.78
8	10,000	13.23	0.75
10	10,000	16.20	0.74
15	10,000	20.87	0.63
20	10,000	26.23	0.59
25	10,000	31.61	0.57

Table 3: Performance of Linear Equation Solver
Ratio is observed Gflops to theoretical peak ($2.2Gflops \times \#PE$)

To reduce the overhead of parallel execution and BROADCAST, the elimination is done in columnwise block. Namely the column vectors in a block are eliminated but not accumulated at first. The information to be used in the elimination in a block, that is, column vectors to be eliminated are BROADCASTed to all the processors. Using them, the elimination and accumulation in the remainder and the accumulation in the block are done in parallel.

The exchange of the row vectors uses partial pivoting. Therefore the inverse of matrix is obtained after the exchange of column vectors according to the history of the pivoting.

$B = (PA)^{-1} = A^{-1}P^{-1}$ then $A^{-1} = BP$ where P is the permutation matrix.

After changing the columnwise blocked partition into rowwise blocked partition, which can be done by parallel data transfer using an additional small work area [12], these exchanges are done concurrently.

4. Wrap-around Partitioning

Wrap-around partitioning is the basis for vectorizing the narrow band linear equation solvers on the VPP300. It comprises a reordering of ($m \times m$ subblocks of) unknowns in a linear system into q blocks of p (subblocks of) unknowns, the purpose being to highlight groups of unknowns that can be eliminated independently of one another. If these fall in regular sequences then this enables vectorisation of the elimination process over p for each of the first $q-1$ partitioning blocks [19]. Its natural formulation is for block bidiagonal matrices, but it is applicable also to narrow-banded matrices by converting these to block bidiagonal form. In the block bidiagonal case it has the advantages that (a) wrap-around partitioning can be applied recursively in conjunction with stable elimination methods - the final subblock is solved sequentially when p becomes too small for effective vectorization; and (b) it does not require p and q to be exact factors of n - this permits the use of memory access strides which avoid contention (stride three is recommended). Speed-ups of about 20 times scalar speed are obtained on the VPP300 for small matrix subblock size m ($m = 2$ in the tridiagonal case) and $n > 1000$. To illustrate the transformation consider the case $n = 16$. An index array A details the addressing information (in practice it would be required only for nonconstant stride access).

$A \leftarrow \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$
step 1: $p(1) = 5, q(1) = 3, O(1) = 0$
$A \leftarrow \{1, 4, 7, 10, 13; 2, 5, 8, 11, 14; 3, 6, 9, 12, 15; 16\}$
step 2: $p(2) = 2, q(2) = 3, O(2) = 10$
$A \leftarrow \{1, 4, 7, 10, 13; 2, 5, 8, 11, 14; 3, 12; 6, 15; 9, 16\}$

Note that there is a case of nonconstant stride in step 2 which corresponds to the inexact factoring of n in step 1. Here the index O points to the start of the current transformation. A diagram is very helpful at this point (see [19]).

5. Fast Fourier Transforms

Two dimensional fast Fourier transforms are basically implemented as a 2-stage process: first a multiple 1D FFT is performed over the first dimension and then a second multiple 1D FFT is performed over the second dimension. The multiple transforms naturally split across processors. However, in order to maintain this "natural distribution" in the case of distributed memory computers such as the VPP300 the data has to be redistributed between the two stages. This redistribution amounts to a personalised all-to-all communication. Using Swarztrauber's "4-step" method this idea is also applied to the case of 1D data where again the main work consists of two multiple 1D FFTs.

If this algorithm is used, all the communication required is concentrated in the personalised all-to-all communication step. Essentially, this step corresponds to a matrix transpose. A big advantage of having all the communication done in one or two steps which is a standard procedure is the ease with which the software can be maintained. Depending on whether the latency is extremely large (like in networked computers) or not (like on the VPP300) one can plug in a different subroutine for the personalised all-to-all communication step. Note that personalised all-to-all communication is an MPI collective communication routine and the matrix transpose is also part of the Fortran 90 standard.

The performance of the FFT relies heavily on the availability of a very fast transpose routine. As the operation count is low, the communication time is much more important than, e.g., for the solution of linear systems of equations. One could consider overlapping the communication and computation. This, however, only gives at most a factor of two speedup and this only if the computation and communication times are approximately equal. One important reason why this overlapping was not implemented was that it substantially increases the software complexity and thus the maintainability and portability for a limited gain in performance. This is particularly relevant as at this stage the paradigm for distributed computing is still in flux and thus porting between environments like MPI, HPF and VPP Fortran should be an option for the user of the library but also for the developer.

The long vector lengths and mainly stride 1 was achieved through the 6 step algorithm described in [17] and [18]. For the case of $n = 2^r$ and radix 2 the complexity of this algorithm is larger than the $4n\log_2(n)$ required for the split-radix [10] algorithm [31]. However, by using a combination of radices $2, 4, 8$ and 16, the complexity is reduced to the same level as for split-radix case. Radices 3/5/7 are also included and the prime-factor idea is used repeatedly to get a very fast method with long vector lengths [18]. The software design of the resulting FFT library is modular and essentially based on three core functions: 1.Small Fourier transforms (orders 2/3/4/5/8/16), 2. Matrix transpose, and 3. Multiplication with a unitary diagonal matrix. In addition, there are some auxiliary routines and combinations of these routines required to enhance performance and functionality [6]. Besides the lower complexity, the higher radix routines have higher computational density which leads to less memory traffic.

The one processor performance of the routines is now compared with the performance of the corresponding NAG routines [25]. In Figure 1 the performance of the library routine based on the algorithm mentioned above is displayed together with the performance of the NAG routines "C06ECF" and "C06FCF" which both implement complex 1D FFTs. The routine C06FCF achieves slightly higher performance but requires more storage space. (The times for this routine are denoted by NAG^* in the diagram.) It can be seen, however, that the SSLII library routine is substantially faster than the NAG routine. It is thought that the higher performance is obtained almost exclusively using stride one data access and maximally long vectors.

The multiple 1D FFT routines have been specifically designed in NAG for efficiency on vector processors. However, even here the Fujitsu SSLII library routines

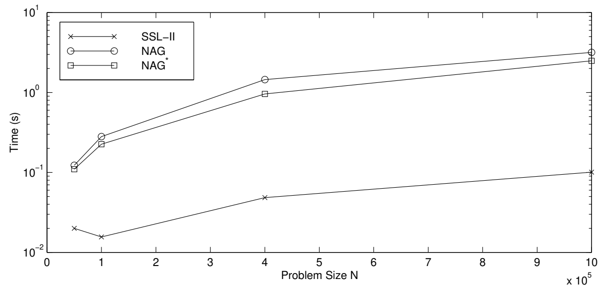

Figure 1: Performance of 1D Complex FFT Routines of the Fujitsu SSL-II library and the NAG library.

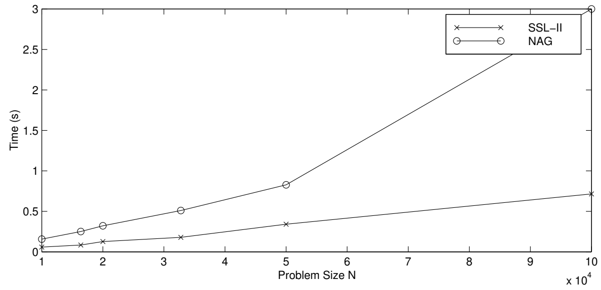

Figure 2: Performance of 1D Complex multiple FFT Routines of the Fujitsu SSL-II library and the NAG library.

implementing the algorithm described here have a clear performance advantage as can be seen from Figure 2

The performance of the 1D FFT is computed as $5n \log_2(n)/t$ where n is the problem size and t is the time required. From Figure 3 it can be seen that close to expected peak performance is achieved for large enough problem sizes on one processor. Furthermore, in Figure 4 the speedup of the parallel algorithm compared with execution on one processor shows the scalability of the algorithm. Further work on FFTs is reported on in [21].

6. Wavelet Transforms

Wavelets yield very good and sparse approximations for most practically occurring functions and data sets and are thus ideally suited for compression [8]. We consider 2D wavelet transforms which can be represented as matrix-matrix products

$$Y = W_M X W_N^T \tag{1}$$

where $X \in \mathbf{R}^{N \times M}$ is the data matrix and W_N (and W_M) are the wavelet transform matrices. These are not formed explicitly, instead the fast wavelet transform is

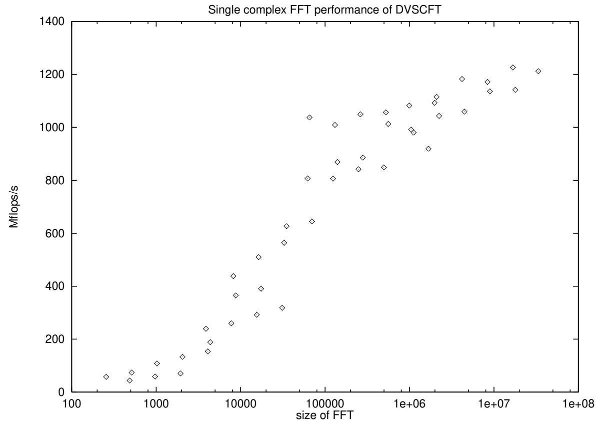

Figure 3: Performance of a 1D complex FFT on one processor.

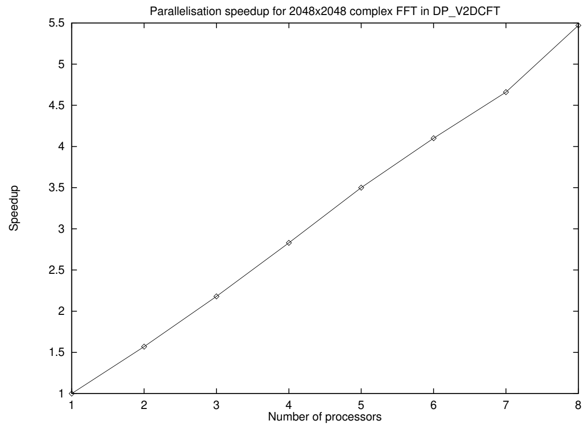

Figure 4: Scalability of FFT on the VPP300.

used which forms the matrix vector product $W_N x$ as a succession of steps of the form

$$c_i^{j-1} = \sum_{k=0}^{D-1} a_k c_{k+2i}^{j}.$$

The parallel implementation of (1) depends on the distribution of the data X and the result Y to the processors.

The wavelet transforms can be done in a similar way as the fast Fourier transforms. As for Fourier transforms, the amount of communication required for the intermediate transposition step is relatively high as the computational density is low. However, due to the local nature" of the wavelets, an alternative is feasible which does not rely on a transposition step. This reduces the amount of communication but requires an introduction of a "distributed wavelet transform" and is also more complex from the point of view of software maintenance. A comparison of the performance of these two approaches can be obtained from Table 1.

On the Fujitsu VPP 300 a wavelet transform with $D = 10$ coefficients gave the following performance:

P	M=N	replicated	combined
1	512	1.3 Gflop/s	1.3 Gflop/s
2	1024	1.3 Gflop/s	2.8 Gflop/s
4	2048	2.3 Gflop/s	5.9 Gflop/s
8	4096	3.8 Gflop/s	11.8 Gflop/s

In the "replicated" column the performance of the replicated, FFT-like algorithm is displayed and in the "combined" column the performance of the algorithm which uses horizontal blocking for X and vertical blocking for Y is given.

7. Sparse Matrix–Vector multiplication

The multiplication of a sparse and large coefficient matrix A of order `N` with a vector x is the basic operation in routines for the solution of large systems of linear systems and eigenvalue problems. In SSLII two storage formats for sparse matrices are considered: the diagonal storage scheme, which is suitable for problems with a regular structure, and the ELLPACK format for unstructured problems.

For the diagonal storage scheme only diagonals of the matrix A containing non–zero elements are stored in the two-dimensional array `MAT`, where the diagonals are extended by zeros in order to make all diagonals the same length. The integer array `IOFF` marks the distance of the diagonal from main diagonal. The calculation of the matrix–vector product $x = x + A \cdot y$ is executed by `ND` general linked triad operations with

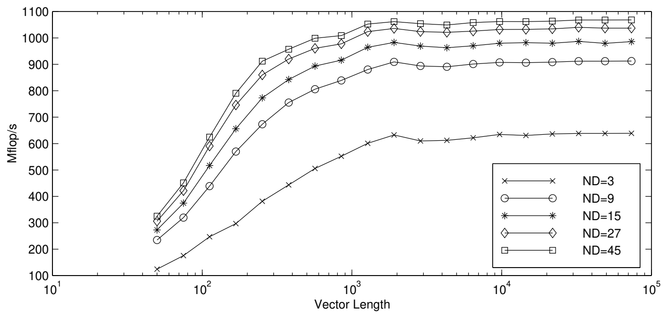

Figure 5: Performance of the matrix-vector multiplication using the diagonal storage scheme.

vector length `N`, where the number of occupied diagonals `ND` is assumed to be very small (< 100):

```
 DO ID=1,ND
    DO IV=1,N
y        X(IV)=X(IV)+MAT(IV,ID)*Y(IOFF(ID)+IV)
    END DO
 END DO
```

(2)

This operation needs three loads and one store operation per diagonal. By using strip mining of the `IV`–loop and two-times unrolling of the `ID`–loop a strip (of length 2048) of the result vector `X` can be kept in the vector register, see [29]. Thus, except for the initial load and final store operation for the result vector
t X, there are two load operations per diagonal. As one load, the multiply and the add pipe can be chained the operation (2) achieves maximal 1.1 Gflops. Only a multi–functional load–store pipe instead of a pure store pipe would achieve theoretical peak of 2.2 Gflops.

Figure 5 shows the performance for different number of diagonals due to finite difference discretizations of order two for several spatial dimensions. The optimal performance of 1.1 Gflops is almost reached for a larger number of diagonals `ND`. For a smaller number of diagonals the additional load operation for the output vector as well as the costs for the filling of the pipes create a significant overhead.

In many applications (eg. finite element problems) the number of diagonals is very large but the number of non–zero entries in the diagonals is rather small. In this case the ELLPACK storage scheme is a better choice: the non-zero entries in every row are stored in the two dimensional array `MAT` and in addition the corresponding column is stored in the two dimensional array `COL`. The matrix vector multiplication looks similar to the diagonal version (2) but with an indexed access to the

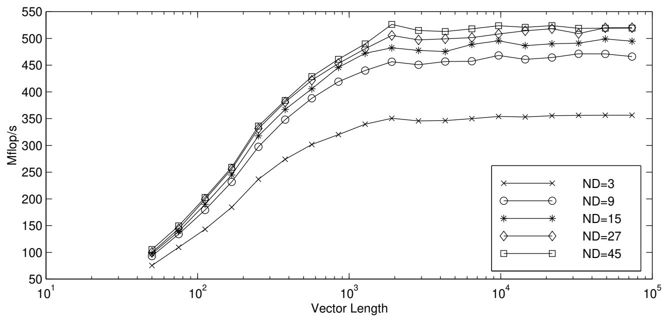

Figure 6: Performance of the matrix-vector multiplication using the ELLPACK storage scheme.

input vector:

```
DO ID=1,ND
   DO IV=1,N
      X(IV)=X(IV)+MAT(IV,ID)*Y(ICOL(IV,ID))
   END DO
END DO
```

(3)

Figure 3 shows the performance for a matrix-vector multiplication with ELLPACK format, where the test matrices are identical to the matrices used for the measurements in Figure 5. The maximal peak performance is 500 Gflops which is achieved only for large values of ND with the maximal number of non–zero entries per row. As the additional index vector ICOL has to be loaded and a non–continuous load operation needs twice as long as a continuous load only a quarter of the peak performance of 2.2 Gflops can be achieved.

Unfortunately the performance for operation (3) is very unlikely to be met in a really unstructured situation as the performance depends dramatically on the actual values of the index vector COL. It is not rare that for a fixed ID the vector ICOL(1:N,ID) produces directly consecutive access to the same memory bank (e.g. by containing two successive identical entries). Thus typically the curves in Figure 5 have heavy downwards deflections as low as 100 Mflop. However, in the case of unstructured grids the usage of the ELLPACK format allows faster matrix–vector multiplications as the number of floating point operations is smaller than for the diagonal storage scheme.

8. Iterative Solvers

For a given real $\mathtt{N}\times\mathtt{N}$ matrix A and given right hand side b the solution vector x of the linear system

$$Ax = b \tag{4}$$

is sought. It is assumed that the number of unknowns N is very large and the matrix A is sparse.

8.1 Conjugate Gradient Methods

Iterative methods of the conjugate gradient type (CG) are robust and parameter–free algorithms to solve these kind of problems on parallel and vector computers in a highly efficient way are given in [32]. However, there is no CG method that is optimal for all matrices. Thus several CG methods have been implemented in order to give the user the ability to select the best method for the problem.

To solve linear systems with a symmetric positive definite matrix, the preconditioned conjugate gradient method has been implemented. Preconditioning methods using Neumann series and incomplete Cholesky factorisation are available, see [7].

For the general case of a non-symmetric coefficient matrix several methods have been implemented. The truncated MGCR (which is a modification of GMRES) converges very fast but is not very robust since for a good convergence the eigenvalues must have positive real parts and be close to the real axis. A more robust but slower method is TFQMR which can successfully be applied to a matrix for which the spectrum is contained in the positive half plane. For the general case of scattered eigenvalues LSQR has been implemented. It is very robust but on the other hand has ver slow convergence.

All CG methods require scalar products and saxpy operations, which are standard and highly efficient operations on the VPP. In addition one matrix–vector multiplication per iteration step has to be executed. This operation is already discussed in section 7.. All CG methods are available for the diagonal as well as the ELLPACK storage scheme.

8.2 Multi-level Preconditioning

A special preconditioner for coefficient matrices arising from the discretisation of partial differential equations on rectangular grids was implemented using the algebraic multi–level iteration (AMLI, see [4]). Compared to classical multi–grid methods AMLI has a wider application range as it uses only information available from the given coefficient matrix. Moreover, it is more robust than a classical multi–grid methods. Naturally for some special applications a classical multi–grid method is faster, however the AMLI is still competitive in most cases.

The coefficient matrix A has to be given in the diagonal storage scheme which means that the underlying grid has to be rectangular but may have an arbitrary spatial dimension. This assumption is made and widely used in the implementation of the AMLI in order to get a highly efficient preconditioner for the relevant case of rectangular grids on the VPP, see below. The development of a version for unstructured grids has been started.

For the construction of the AMLI preconditioner the coefficient matrix A is reduced to a smaller coefficient matrix $A^{(1)}$ by subdividing the matrix A into

$$A = \left[\begin{array}{cc} A_{FF} & A_{FC} \\ A_{CF} & A_{CC} \end{array} \right] . \qquad (5)$$

The columns and rows of the sub–matrix A_{CC} belong to the so–called coarse unknowns of the reduced matrix and those of the sub–matrix A_{FF} to the unknowns that are removed. The inverse of the matrix A_{FF} is approximated by a diagonal matrix D_{FF}, eg. calculated from the condition $D_{FF}A_{FF}e = e$ with $e = (1, \ldots, 1)$. The coarse level matrix $A^{(1)}$ is defined by block–wise Gaussian elimination:

$$A^{(1)} := A_{CC} - A_{CF}D_{FF}A_{FC} \ . \qquad (6)$$

The reduction can be successively applied to the coarse level matrices until a final level l has been reached. On the coarsest level l a rather small linear system has to be solved with less accuracy (2-3 digits). That can be done very fast using iterative methods if the coefficient matrix contains only a small number of occupied diagonals.

Within the CG iteration the preconditioner delivers the image $p = \mathtt{AMLI}(0, r)$ of a given vector r by the following recursive algorithm:

$$\begin{array}{rcl}
1 & : & \mathtt{AMLI}(k, r) \\
2 & : & \quad \mathtt{IF}\ (k = l)\ \mathtt{THEN} \\
3 & : & \qquad \mathtt{SOLVE}\ Ap = r \\
4 & : & \quad \mathtt{ELSE} \\
5 & : & \qquad (r_F, r_C) \leftarrow r \\
6 & : & \qquad q_C \leftarrow r_C - A_{CF}D_{FF}r_F \\
7 & : & \qquad p_C \leftarrow \mathtt{AMLI}(k+1, q_C) \\
8 & : & \qquad p_F \leftarrow D_{FF}(r_F - A_{FC}p_C) \\
9 & : & \qquad p \leftarrow (p_F, p_C) \\
10 & : & \quad \mathtt{END\ IF} \\
11 & : & \mathtt{RETURN}\ p\ .
\end{array} \qquad (7)$$

The block forward substitution in step 6 and the block backward substitution in step 8 needs matrix–vector multiplications with the matrices A_{FC} and A_{CF}, respectively.

In order to get efficient matrix–vector multiplications in the matrices should be stored using the diagonal storage scheme. This has to be considered in the

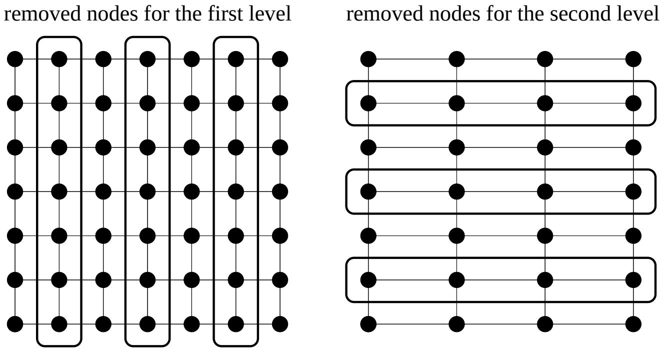

Figure 7: Two coarse levels on a 7×7 grid.

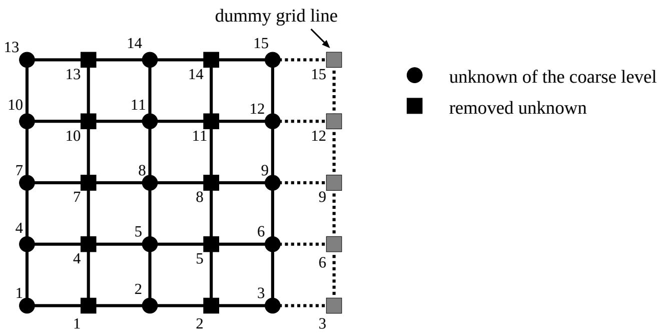

Figure 8: Enumeration of coarse and removed unknowns.

selection of the coarse level unknowns. The assumption that the matrix A arises from a discretisation procedure allows the recovery of the underlying rectangular grid from the given diagonals. The unknowns of the coarse level are selected by an alternating direction approach: every second grid line (or more general hyperplane) is removed in one fixed spatial direction where the spatial direction is cyclically changed from one level to the next. Figure 7 shows this strategy for a two dimensional 7×7 grid, where in the first level grid lines are removed in the x_1–direction, in the second level in the x_2–direction and in the third level in the x_1–direction again.

The coarse level unknowns are enumerated in a natural manner by dropping the removed nodes. Then the coarse level matrix mounted by formula (6) can be efficiently stored using the diagonal storage scheme In most relevant cases the number of diagonals is even equal to the number of diagonals in the original matrix. Also there is less fill–in as the the sub–matrix A_{FF} is condensed to a diagonal matrix D_{FF} otherwise more long–range couplings between coarse grid nodes would be created. In the grid of the removed unknowns an additional dummy grid line is put–up if the number grid node in the direction, in which grid lines have

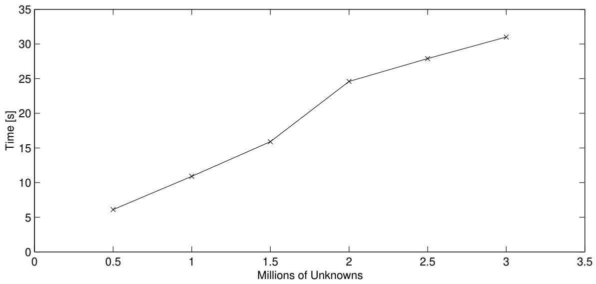

Figure 9: AMLI on one processor.

unknowns N	$0.5 \cdot 10^6$	$1 \cdot 10^6$	$2 \cdot 10^6$	$4 \cdot 10^6$
processors NP	1	2	4	8
levels l	4	5	5	6
time [sec]	6.11	7.53	10.1	13.5

Table 4: AMLI on several processors.

been removed, is odd, see Figure 8. By this trick the number of diagonals in the sub–matrices A_{FF}, A_{FC} and A_{CF} becomes not larger than the number of diagonals in the given matrix. Thus these matrices can be efficiently stored in the diagonal storage scheme.

Figure 9 shows the timings for the solution of linear systems which coefficient matrices arise from the finite difference discretisation of the differential operator $-\nabla a \nabla$ on a rectangular and equidistant grid on the 3–dimensional unit cube. The function a jumps at the surface of the unit sphere from 1 up to 10^6. The accuracy on the level of equation is 10^{-6}. The timing includes the computing time for the assemblage of the coarse level matrices. The number of levels is automatically selected by balancing the computational effort for the solution of the linear system on the coarsest level and the forward and backward substitutions. The growth of the computing time with the number of unknowns is not worse than linear. It is emphasised that in all cases the CG iteration with Jacobi preconditioner was not successful within 10 minutes computing time.

8.3 Parallelisation

In the parallel version all vectors of the length N of number of unknowns are stored distributed over the NP processors, where every processor holds a vector portion of the size $\frac{\mathtt{N}}{\mathtt{NP}}$. According to this data distribution the vectorised loops over the number of unknowns is split in order to execute the loop strips in parallel with a vector length $\frac{\mathtt{N}}{\mathtt{NP}}$ on every processor. Communication is necessary to calculate the scalar products and to gather vector portions from other processors in order to build up the input vector for the matrix-vector multiplication. In the forward and backward substitution (step 6 and step 8 in algorithm (7)) communication is needed to separate and combine the coarse and removed unknowns. The corresponding routines produce vectors which can be directly used as input vectors for the following matrix–vector multiplications, with A_{CF} and A_{FC}, respectively. Thus only one communication step per forward and per backward substitution is needed. Table 4 shows the timings when solving the example problem on several number of processors where the number of unknowns per processor $\frac{\mathtt{N}}{\mathtt{NP}}$ is constant.

9. Sparse Cholesky Factorisation

Direct solution of large sparse symmetric positive definite systems of linear equations has many important applications in science and engineering. A normal method consists of four steps. The first step is matrix reordering. In this step the rows and columns of the original matrix are reordered to achieve the fill reduction in the factor matrix. The second step is symbolic factorisation which will generate a compact data structure for the Cholesky factors. Since there is no numerical computation involved, the computation in these two steps is purely symbolic. The third and the fourth steps are numerical Cholesky factorisation and the solution of triangular systems.

On a conventional computer such as workstation the time taken for symbolic computation (mainly the time for matrix reordering) is usually considerably shorter than the time required for numerical computation. The algorithm design were mainly focused on the quality of the matrix reordering in terms of fills, that is, how to minimise the fills in the factor matrix to improve the performance for numerical computation, but the time spent for symbolic computation is considered as a less important factor. On more advanced machines which are equipped with vector computational units, however, the time for symbolic computation can be comparable to the time for numerical factorisation. Therefore, other factors such as the cost of indirect addressing, the vector length and the time for symbolic computation should also be considered in order to enhance the overall performance.

Since the result of symbolic computation is very crucial for achieving highly efficient sparse matrix solvers, in the following we only describe the algorithm adopted for symbolic computation on a single processor VPP300 and discuss the difficulties of parallelisation in the symbolic computational steps.

The most commonly used algorithm for performing reordering is the minimum degree algorithm [23]. This algorithm adopts a bottom-up method that uses only local information. For large size problems it may not perform, in terms of fills, as good as nested dissection and its variants which adopt a top-down method and use global information. As we discussed previously, however, the overall performance is not just determined by the number of fills. In general cases the algorithms using the top-down technique can take much longer time to complete than the minimum degree ordering. Thus we decide to use the minimum degree algorithm.

Two nodes x and y in an elimination graph are said to become indistinguishable if they satisfy

$$Adj(x) \cup \{x\} = Adj(y) \cup \{y\}.$$

Indistinguishable nodes can be grouped together to form a supernode and eliminated all at the same time. To group indistinguishable nodes can reduce the reordering complexity and the cost for indirect addressing during numerical computation. To achieve high efficiency for direct sparse Cholesky factorisation on modern vector computers, it is very important to effectively detect the indistinguishability during the matrix reordering. The conventional multiple minimum degree [23] includes a method to detect the indistinguishability. Though not expensive, the method is less aggressive. To improve the performance we adopt a greedy graph compression technique.

Many matrices have certain columns of identical nonzero pattern before the reordering. Thus the graph corresponding to an original sparse matrix may be compressed into a smaller graph by grouping indistinguishable nodes. This graph compression [2] usually takes only a small amount of time and the overall reordering time can be reduced if the size of the original graph can indeed be reduced.

We know that the graph compression applied at the beginning of the reordering may improve the performance. The question is whether the performance can further be improved if the compression is also applied during the reordering. In our algorithm with a greedy graph compression we keep all the good features of the original multiple minimum degree and the compressed graph for preprocessing. At each step we also do graph compression immediately after the degree update, that is, we try to find more indistinguishable nodes from those being just updated. Our experimental results show that the performance has been improved significantly after adopting the greedy graph compression technique [34].

Conventionally all logical zeros should be excluded from the data structure for Cholesky factors to reduce both the computational complexity and the size of storage space. The problems are that the length of columns can become very short and that the cost of indirect addressing be significantly increased. Our early experimental results show that short vector length and indirect addressing can cause performance problems on the Fujitsu VPP300.

Matrix	Method 1	Method 2	Improvement
bcsstk17	1.84	1.58	14.1
bcsstk25	3.49	2.96	15.2
bcsstk30	5.91	4.95	16.2
bcsstk31	8.47	6.94	18.1
bcsstk32	8.19	6.83	16.6
bcsstk33	3.31	2.69	18.7
bcsstk35	4.83	3.87	19.9
bcsstk36	3.73	3.30	11.5
bcsstk37	4.06	3.56	12.3
c3dkt3m2	19.1	15.8	17.3
c3dkq4m2	24.4	18.5	24.2
finan512	145	9.12	93.7

Table 5: Improvement in % with relaxed supernodes

To alleviate these problems we adopt an idea of relaxed supernodes which is similar to that described [3]. With relaxed supernodes we intentionally include a number of logical zeros in the factors by merging certain columns or supernodes together. This may result in an increased complexity in numerical computation, yet the overall performance can be enhanced because short columns are made longer and the cost of indirect addressing is decreased.

Some experimental results obtained on a single processor VPP300 are presented in Table 5. We compared two different methods. The same algorithm for numerical factorisation and the solution of triangular systems is used in both methods. The difference between the two methods is that method one uses only the conventional multiple minimum reordering algorithm and does not apply the relaxed supernode algorithm, while method two adopts both the greedy graph compression and the relaxed supernodes with the number of allowable zero entries set to 200 (which is certainly not an optimal number). It can be seen from the table that the improvement of using method two are significant.

The primary goal of designing sequential matrix reordering algorithms is simply to reduce the workload and memory space for the numerical factorisation. For parallel computing, however, we need algorithms which can also be executed efficiently in parallel. Algorithms based on the bottom-up technique such as multiple minimum degree are sequential in nature and very difficult to be efficiently implemented on parallel machines.

Some efforts have been made to convert the sequential nested dissection and its variants such as multisection into parallel algorithms. However, algorithms based on top-down technique are usually much more expensive than the minimum degree in general cases. Although some of these algorithms can be computed in parallel, the results are not very exciting when the total computational time is considered. Till now the success in designing very efficient parallel algorithms for matrix reordering is still limited. As the processing speed for numerical computation continues to increase on advanced computing systems and more efficient parallel numerical factorisation algorithms are developed, the symbolic computation will become a bottleneck. Efficient parallel implementation of symbolic computation thus becomes a necessary and crucial step in parallel computation of sparse linear systems.

For parallel reordering an efficient algorithm should produce a good quality in terms of fills, take the problem of load balancing into consideration and show the significant improvement in terms of time over the fastest sequential reordering algorithm. It should also be scalable, that is, the ordering time should be scaled down with the increase in the number of processors. That is a very challenging problem.

10. Eigenvalue Problems

The development and implementation of solution techniques for eigenvalue problems has been underway only for a few years. We have developed routines for tridiagonal, symmetric, and nonsymmetric matrices. The necessity of striving for both vectorization and parallelization in our routines has lead both to novel implementations of known methods and to the development of novel techniques for certain classes of problems.

In the next few sections we consider the solution of eigenvalue problems

$$Au = \lambda u, \tag{8}$$

under various assumptions as to specific properties of A (symmetric, nonsymmetric) and/or structural characteristics (tridiagonal, block bidiagonal, banded, etc.). The routines are at varying stages of development, and the amount of discussion devoted to each reflects this.

10.1 Tridiagonal EVPs

Consider the solution of the eigenvalue problem

$$Tu = \lambda u,$$

where T is tridiagonal. We assume that T is irreducible and symmetric. If T were irreducible, i.e. one of its off-diagonal entries were zero - say the k-th, then the matrix is separable into two tridiagonal matrices, of sizes k and $n-k$, each of which is irreducible. If the matrix were nonsymmetric then, under the assumption that symmetrically-positioned off-diagonal terms – $t_{i,i+1}$ and $t_{i+1,i}$ – have the same sign, it is symmetrizable via a diagonal similarity transformation, the elements of which depend on square roots of ratios of these off-diagonal entries. For the remainder of this section it is assumed that T in above is an irreducible, symmetric matrix.

Bisection and Multisection: One of the standard techniques for computing selected eigenvalues of tridiagonal matrices is based on the so-called *Sturm sign count* and *bisection*. This is the case, for example, for the relevant routines in in the widely used LAPACK [1] and Scalapack [5] software packages. Seeking brevity we omit the details which can be found in, for example, [28].

The main computational component of a typical implementation of bisection is evaluation of the sign count $\sigma_n(\lambda)$ defined by the recursion

$$\sigma_i(\lambda) = \left(\alpha_i - \frac{\beta_i^2}{\sigma_{i-1}(\lambda)}\right) - \lambda, \tag{9}$$

for $i = 1, 2, \cdots, n$. Zeros of $\sigma_n(\lambda)$ are eigenvalues of T, and the number of times $S_i(\lambda) < 0$, $i = 1, \cdots, n$, is equal to the number of eigenvalues of T less than the approximation λ.

The recurrence (9) is not vectorizable over the index i. However, evaluating (9) over m estimates λ_j, $j = 1, \cdots, m$, engenders the possibility of interchanging i- and j-loops and vectorizing over j. This is the fundamental idea behind *multisection*; it entails the subdivision of an interval into greater than two subintervals. Although this generally entails spurious computation – many (adjacent) intervals may contain no eigenvalues, the efficiency gained through vectorization should offset the penalty imposed by spurious sign count evaluation. Another way to enable vectorization, in the case of computing more than one eigenvalue, is to bisect multiple intervals; we refer to this as "multibisection".

In [30] it was shown that bisection is not optimal on vector processors. The machines considered in that study typically had values of $n_{1/2}$ of no more than about 20. However, many of today's vector processors have a considerably larger $n_{1/2}$. On the VPP300 it is measured in the (lower) hundreds, and it turns out that the larger is $n_{1/2}$ the less optimal is bisection [14]. We illustrate the nonoptimality of bisection on the VPP300 in Figure 10 where we plot the time to compute one eigenvalue of a tridiagonal matrix of size 1000 as a function of the vector length, i.e. number of multisection points. The tolerance is $\epsilon =$ `3.0d-16`, and, as with all plots in this section, times are averages from 25 runs. The bisection time is circled; this is also the time obtained using LAPACK.

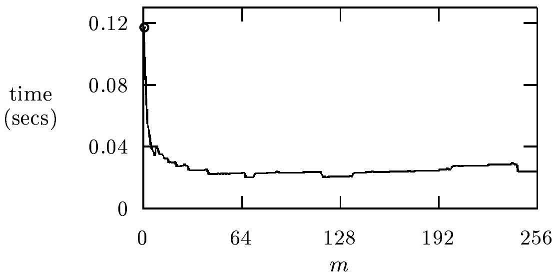

Figure 10: Time vs. number of multisection points.

Obviously, bisection is not optimal. The optimal multisection time, which is achieved using 70 points, is approximately one-sixth the time for bisection. LAPACK and ScaLAPACK use bisection and hence are considerably slower than our routine on the VPP300; so much so in fact that the performance of one of our symmetric eigensolvers (based on Householder reduction to tridiagonal form) is faster than their equivalent routine. We note that the codes used there can be modified so that they will perform multibisection when $r > 1$ eigenvalues are requested and the differences in performance are reduced as r increases (since multi-bisection becomes optimal [13]).

Once it has been decided that multisection is to be used as opposed to bisection there still remains a critical question: What vector length – number of multisection points – should be used? This is considered in detail in [13] for the computation of a single eigenvalue or for $r > 1$ eigenvalues; here we only briefly elucidate a few relevant points.

The optimal vector length is a function of the number r of eigenvalues and the desired accuracy ϵ; dependence on ϵ is not noted in [30] but on the VPP300 for $\epsilon \in [3 \times 10^{-16}, 10^{-7}]$ we find $m_{\mathrm{opt}} \in [47, 250]$ [13]. These values of m_{opt} are roughly five to twenty times those determined in [30], manifesting the effect of the significantly larger $n_{1/2}$ of the VPP300 in comparison to the computers used in that study.

Generally the optimum it is a minimal number of points which will effect convergence in some number ν of multisection steps; using a few more points than this minimum only results in additional operations on each of the ν steps. However, extensive performance analysis, as performed in [13], reveals a vector length performance anomaly: There is a noticeable increaase – of 10-20% – in execution time in going from a vector length of $64i+8$ to $64i+9$, $i = 0, 1, 2, \ldots$. This is illustrated in Figure 11 where we plot the time to compute one eigenvalue as a function of the number of multisection points $m = 1, \ldots, 1024$. The plot "wraps around" - the bottom curve is for $m = 1, \ldots, 256$, the next for $m = 257, \ldots, 512$, etc. Precision is $\epsilon = 3 \times 10^{-10}$. Note the slight increases in time for vector lengths falling on the dotted lines at $m = 64i + 9$.

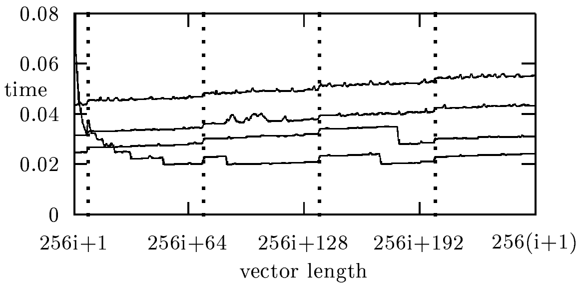

Figure 11: Time vs. vector lengths $m = 1, \ldots, 1024$. $i = 0$ for the bottom curve, $i = 1$ for the next, etc. Dotted lines are for $m = 64i + 9$.

The point to be made is that only by initiating a detailed performance analysis was this odd behaviour noticed. Although the savings is relatively small – typically 10-20% – avoidance of these vector lengths is still worthwhile, particularly in the case of developing library-quality software which may be used over a long period of time, in which case even small savings in execution time may cumulatively offset the time required to perform the analysis.

Parallelization Issues: Eigenvectors are calculated via the standard technique of inverse iteration (see, e.g., [33]). Letting λ_i denote a converged eigenvalue, choose v^0 and iterate

$$(T - \lambda_i I)v^k = v^{k-1}, \quad k = 1, 2, \cdots .$$

Usually one step suffices. Vectorization is enabled using wrap-around partitioning [11, 19] as discussed Section 4..

If all eigenvalues are distinct, computation of eigenpairs is embarassingly parallel. However, eigenvectors corresponding to multiple or clustered eigenvalues generally require orthogonalization, a communication-intensive operation for eigenvectors distributed on different PEs.

Ways around this include computing all eigenvalues on each PE or distributing the computation and initiating all-to-all communication afterwards; in either case complete spectral information is available to all PEs and work allocation can be decided. Our approach entails a distributed computation and the communication of only a few integer values (sign counts). We detect clustering *during* the multisection refinement. Once the subinterval width is the size of the user-defined cluster tolerance the smallest and largest sign counts on each

n	LAPACK	new	old1	old2
2068	72.00	38.73	125.8	113.19
2254	92.88	48.51	153.03	135.97
4709	766.63	397.73	834	687

Table 6: Symmetric eigensolver - one PE.

n	new	ScaLAPACK
2068	19.87	70 - 87
2254	24.88	100 - 115
4709	295.38	Execution aborted

Table 7: Symmetric eigensolver - four PEs.

PE are communicated to the PEs which were initially allocated eigenvalues with those indices, and the decision as to which PE is to keep the clustered eigenvalues is made; this PE then refines these eigenvalues to the desired accuracy and computes corresponding invariant subspaces. This process will generally result in load-imbalance, the effect worsening as cluster size increases; if clusters extend across more than two PEs some will drop entirely from the computation.

10.2 Symmetric EVPs

Since the methods we have adopted for symmetric (and Hermitian) EVPs are relatively standard, we do not describe them in detail. Each uses the tridiagonal eigensolver just described, and often the overall performance of the symmetric eigensolvers benefit from the efficiency of that highly optimized routine.

The primary method we use for symmetric and Hermitian matrices is based on the usual Householder reduction to tridiagonal form; the reduction is parallelized using a panel-wrapped storage scheme [9]. In Tables 10.2 and 10.2 we tabulate some performance data obtained for the solution of a problem from quantum chemistry [15] on one and four PEs, respectively.

The data in colums labelled 'old1' and 'old2' are for previous symmetric eigensolvers which were included in SSLIIVP, the former based on the QL method, the latter on bisection and inverse iteration. The efficiency of the new routine is evident.

ScaLAPACK times depend on the choice of block size. Most of the superiority of our implementation is due to that of the tridiagonal eigensolver.

For sparse matrices we use a Lanczos method. Performance depends primarily on efficient matrix-vector multiplication; the routine uses a diagonal storage format, loop unrolling, and a block distribution of matrix columns for parallelization. The (reduced) tridiagonal eigenproblem is solved redundantly on each processor. Another routine is based on a new one-sided reduction algorithm see [20].

10.3 Nonsymmetric EVPs

The nonsymmetric eigensolvers are still at an early stage of development. For this reason no performance are data are given. We merely give a short summary of work in progress.

An implementation of an Arnoldi method is underway. Thus far we have incorporated restarting, an implicit deflation scheme, and shift-invert transformation. In order to be better able to compute multiple/clustered eigenvalues, a block version is also being developed; for the block version matrix-vector multiplication is replaced by matrix-matrix multiplication enabling the use of level-3 BLAS.

Since the projected eigensystems should be small, it is probably not worthwhile parallelizing their eigensolution, and we will focus instead on parallelization of the reduction, as was done for the Lanczos solver. This is also the approach taken with P_ARPACK [24], a parallel implementation of the popular Arnoldi software package ARPACK [22].

Another technique we use for nonsymmetric EVPs is based on the use of Newton's method. It is in essence an inverse iteration technique, and matrix inversion is effected using the highly efficient linear solver described previously. The method is capable not only of the usual quadratic convergence rate of Newton's method, but of higher-order convergence in certain circumstances. This is the case, for example, for a separate version of this routine designed specifically for block bidiagonal matrices; it uses the wrap-around partitioning techniques of 4.. For more details on the method, convergence rates, and detailed discussion of issues such as deflation see [27]; the presentation there is in terms of generalized EVPs.

Although Newton methods are ultimately capable of quadratic – and even third-order – convergence, there are problems when good initial data are unavailable, and this is often the case when solving EVPs. On the other hand, Arnoldi methods seem nearly always to move in the right direction at the outset, but may stall or breakdown as the iteration continues; many heuristics are required to develop an efficient and robust Arnoldi eigensolver. In a sense the Newton and Arnoldi procedures have orthogonal difficulties, Newton methods suffering at the outset and ultimately performing very well, and Arnoldi methods starting off well but perhaps running into difficulties as the iteration proceeds. Hence we are considering a composite method: The Arnoldi method is used to obtain initial eigenestimates for the Newton-based procedures. Work on this method has only recently commenced, but results appear promising [16].

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov, and D. Sorensen, *LAPACK: Linear Algebra PACKage*, software available from `http://www.netlib.org` under directory "lapack".

[2] C. Ashcraft, Compressed Graphs and the Minimum Degree Algorithm, *SIAM J. Sci. Comput.*, Vol. 16, No. 6, 1993, pp.1404-1411.

[3] C. Ashcraft and R. Grimes, The Influence of Relaxed Supernode Partitions on the Multifrontal Method, *ACM Trans. Math. Software*, Vol. 15, 1989, pp.291-309.

[4] O. Axelsson and M. Neytcheva. Algebraic multilevel iteration method for Stieltjes matrices. *Num. Lin. Alg. Appl.*, 1:213–236, 1994.

[5] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R.C. Whaley, *ScaLAPACK: Scalable Linear Algebra PACKage*, software available from `http://www.netlib.org` under directory "scalapack".

[6] R. Brent, L. Grosz, D. Harrar II, M. Hegland, M. Kahn, G. Keating, G. Mercer, O. Nielsen, M. Osborne, B. Zhou and M. Nakanishi, "Development of a Mathematical Subroutine Library for Fujitsu Vector Parallel Processors", submitted to International Conference on Supercomputing, Melbourne, Australia, July 1998.

[7] R. Brent, A. Cleary, M. Hegland, J. Jenkinson, Z. Leyk, M. Osborne, P. Price, S. Roberts, D. Singleton and M. Nakanishi, "Implementation and Performance of Scalable Scientific Library Subroutines on Fujitsu's VPP500 Parallel-Vector Supercomputer", Proceedings of the Scalable High Performance Computing Conference, Knoxville, Tennesee, May 1994, IEEE/CS Press, 1994, pp 526-533

[8] C. K. Chui, "Wavelets: a mathematical tool for signal processing", SIAM Monographs on Mathematical Modeling and Computation, Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, With a foreword by Gilbert Strang, 1997.

[9] J.J. Dongarra and R.A. Van de Geijn, *Reduction to condensed form for the eigenvalue problem on distributed memory architectures*, Parallel Computing **18** (1992), 973–982.

[10] P. Duhamel and H. Hollman, "Split radix FFT algorithms", **20**, 1985, pp 14–16.

[11] C.R. Dun, M. Hegland, and M.R. Osborne, *Parallel stable solution methods for tridiagonal linear systems of equations*, Computational Techniques and Applications: CTAC '95 (Melbourne, Australia) (R.L. May and A.K. Easton, eds.), World Scientific, 1996, pp. 267–274.

[12] "Fujitsu SSLII/VPP User's Guide (Scientific Subroutine Library)", Fujitsu, Japan.

[13] D.L. Harrar II, *Determining optimal vector lengths for multisection on vector processors*, In preparation.

[14] D.L. Harrar II, *Multisection vs. bisection on vector processors*, In preparation.

[15] D.L. Harrar II and M.H. Kahn, *On the efficient solution of symmetric/Hermitian eigenvalue problems on parallel arrays of vector processors*, Computational Techniques and Applications: CTAC '97 (Adelaide, Australia) (J. Noye, M. Teubner, and A. Gill, eds.), World Scientific, To appear 1998.

[16] D.L. Harrar II and M.R. Osborne, *Composite Arnoldi-Newton methods for large nonsymmetric eigenvalue problems*, Computational Techniques and Applications: CTAC '97 (Adelaide, Australia) (J. Noye, M. Teubner, and A. Gill, eds.), World Scientific, To appear 1998.

[17] M. Hegland, *A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing*, Numerische Mathematik **68**, no. 4, 507–547, 1994.

[18] M. Hegland, *An implementation of multiple and multivariate Fourier transforms on vector processors*, SIAM J. Sci. Comp. **16**, no. 2, 271–288, 1995.

[19] M.Hegland, and M.Osborne, *Wrap-around partitioning for block bidiagonal linear systems*, IMA J. Num. Anal., To appear.

[20] M. Hegland, M. Kahn and M. Osborne, *A Parallel Algorithm for the Reduction to Tridiagonal Form for Eigendecomposition*, Submitted.

[21] G. Keating, *Multidimensional Vector FFTs Using Only Square Transpositions*, Proceedings HPCASia, 1998.

[22] R.B. Lehoucq, D.C. Sorensen, and P. Vu, *ARPACK: An implementation of the Implicitly Restarted Arnoldi Iteration that computes some of the eigenvalues and eigenvectors of a large sparse matrix*, 1995.

[23] Liu, J. W. H., Modification of the Minimum Degree Algorithm by Multiple elimination, *ACM Trans. Math. Software*, Vol. 11, 1985, pp.141-153.

[24] K. Maschoff and D. Sorensen, *P_ARPACK: An efficient portable large scale eigenvalue package for distributed memory parallel architectures parallel supercomputer*, Proc. of the Third Int. Workshop on Appl. Parallel Comp. (PARA96) (Denmark), 1996.

[25] NAG Fortran Library Manual, Numerical Algorithms Group, Wilkinson House, Jordan Hill Road, Oxford, UK.

[26] M. Nakanishi, H. Ina and K. Miura, *A High Performance Linear Equation Solver on the VPP500 Parallel Supercomputer*, Proceedings of Supercomputing 94, Washington D.C., Nov. 1994.

[27] M.R. Osborne and D.L. Harrar II, *Inverse iteration and deflation in general eigenvalue problems*, Tech. Report Mathematics Research Report No. MRR 012-97, Australian National University, 1997.

[28] B.N. Parlett, *The Symmetric Eigenvalue Problem*, Prentice Hall, Englewood Cliffs, 1980.

[29] W. Schönauer. *Scientific Computing on Vector Computers.* North-Holland, Amsterdam, New York, Oxford, Tokyo, 1987.

[30] H.D. Simon, *Bisection is not optimal on vector processors*, SIAM J. Sci. Stat. Comput. **10**, 205–209, 1989.

[31] C. Van Loan, *Computational frameworks for the fast Fourier transform*, SIAM, 1992.

[32] R. Weiss. *Parameter-Free Iterative Linear Solvers.* Mathematical Research, vol. 97. Akademie Verlag, Berlin, 1996.

[33] J.H. Wilkinson, *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.

[34] B.B. Zhou, *The Multiple Minimum Degree Ordering with Greedy Graph Compression for Vector Computers*, Computational Techniques and Applications: CTAC '97 (Adelaide, Australia) (J. Noye, M. Teubner, and A. Gill, eds.), World Scientific, To appear 1998.