Area 4 Working Note 16: Implementation and Performance of Scalable Scientific Library Subroutines on Fujitsu's VPP500 Parallel-Vector Supercomputer

R. Brent, A. Cleary, M. Dow,
M. Hegland, J. Jenkinson, Z. Leyk,
M. Osborne, S. Roberts, D. Singleton

Australian National University
Canberra, ACT 2602
Australia

M. Nakanishi

Fujitsu Limited
Numazu-shi, Shizuoka 410-03
Japan

Abstract

We report progress to date on our project to implement high-impact scientific subroutines on Fujitsu's parallel-vector VPP500. Areas covered in the project are generally between the level of basic building blocks and complete applications, including such things as random number generators, fast Fourier transforms, various linear equation solvers, and eigenvalue solvers. Highlights so far include a suite of fast Fourier transform methods with extensive functionality and performance of approximately one third of peak; a parallel random number generator guaranteed to not repeat sequences on different processors, yet reproducible over separate runs, that produces randoms in 2.2 machine cycles; and a Gaussian elimination code that has achieved over a Gflop per processor for 32 processors of the VPP500, and 124.5 Gflops total on the Fujitsu-built Numerical Wind Tunnel, a machine very similar architecturally to the VPP500.

1 Introduction

The Australian National University has forged an impressive partnership with the Fujitsu Corporation on a number of scientific computing topics. One of the most recent of these is the Fujitsu-ANU Parallel Scientific Subroutine Library project. This project is concerned with the implementation of high-impact scientific subroutines for inclusion in Fujitsu's parallel version of their SSL-II Scientific Subroutine Library for the new generation VPP500 parallel-vector machine. The ANU's role in this partnership is to apply their expertise in state-of-the-art parallel algorithms to computations of particular importance to Fujitsu's users. Similar parallel libraries are being pursued commercially for other architectures by companies such as Thinking Machines, Cray, and IBM, and in public domain projects such as ScaLAPACK [4].

The VPP500 is Fujitsu's new entry into the high end of supercomputers. It consists of up to 222 highly powerful vector processors connected by a proprietary crossbar switch. Each processor is a multiple-pipe vector computer with a peak theoretical speed of 1.6 Gflops, giving a fully configured machine a peak of over 320 Gflops. The pipes are "fat" in the sense that they produce four add-multiply results per clock cycle. This increases the peak speed, but at the same time increases the length of vectors needed to achieve this speed by the same amount, making it difficult though not impossible to fully utilize its capabilities. The Numerical Wind Tunnel Machine of National Aerospace Laboratory in Japan, which is architecturally similar to the VPP500, reported the highest Linpack benchmark results as of the end of 1993 [10], 124.5 sustained Gflops for 140 processors. The programming model of the VPP500 is that of a single address space, though references to global memory are expensive and best used only for communication. Access to parallel facilities are given through VPP Fortran, a Fortran-77 based language with compiler directives to handle the parallel programming aspects (High Performance Fortran is planned for the future, and there are hopes that standard-interface message passing capabilities may be added as well). Recent indications are that the majority of the first wave of customers will be purchasing machines in the 20 to 40 processor range. The combination of a moderate number of processors, each of which requires very long vectors to achieve high efficiency, means that the VPP500 offers a different flavor to algorithm designers in terms of scalability. For example, if vector lengths of 500 are required to achieve an acceptable efficiency on each processor (which is approximately correct), then to issue a separate vector instruction of this length to each of 40 processors (normally considered fairly small in parallel computing circles) in parallel requires an algorithmic parallelism of 20,000 independent operations, which is a challenge to achieve for many scientific calculations.

Instead of attempting to provide an exhaustive library, we have concentrated on providing important user and application groups with routines that impact the most on their calculations. The technical areas of interest so far and some of their application areas are: fast fourier transforms for signal processing, random number generators for Monte Carlo simulations, iterative methods for symmetric and nonsymmetric linear systems for PDE solutions, least squares problems useful in statistics, sparse linear system solution used in structures calculations and circuit design, dense and banded linear system solution for boundary element methods and ODEs, and both symmetric and unsymmetric eigenvalue problems for chemistry and control problems. We note that to a large extent other vendors have followed the same sort of pattern, concentrating on certain key areas and applications.

The VPP500 has only been accessible to us for a short period of time, and runs have been limited mostly to four processors. However, development has been aided by the use of two other Fujitsu computers residing at the ANU: the VP2200 is a vector computer with the same vectorizing compiler as that used on the VPP500 and has allowed us to experiment with vectorization aspects; and the AP1000 is a 128 node MIMD parallel computer that has a VPP Fortran compiler (actually a translater into node Fortran and message passing library calls) that has allowed us to debug parallel programs and test scalability in lieu of access to a larger VPP500. At this point our results are a mixture of results from the VPP500 and the AP1000.

We briefly describe our efforts for some of the specific technical areas listed above in the remainder of this paper, along with some more general comments concerning the implementation of library subroutines on the VPP500.

2 Random Number Generators

Brent [3] considered several popular classes of random number generators, including linear congruential generators, and concluded that the requirements of a parallel vector random number generator could best be met by a vectorized implementation of generalized Fibonacci generators. The same conclusion appears to have been reached independently by others [1, 21]. A single vector processor version, RANU4, was completed as part of this project for the Fujitsu VP 2200/10 vector processor in 1992.

The generalized Fibonacci generators are based on the recurrence

$$x_n = x_{n-r} + x_{n-s} \bmod 2^w$$

where w is related to the wordlength of the machine and is usually considerably larger than 1. In his recent paper [19] Marsaglia recommends a new class of generators, termed "very long period" (VLP) generators. These are similar to generalized Fibonacci generators but can achieve periods close to 2^{rw}, whereas the generalized Fibonacci generators can "only" achieve period $O(2^{r+w})$. Thus, for comparison purposes, we implemented one of Marsaglia's VLP generators (RANU5) on the VP 2200.

Tests on the VP 2200 indicate that RANU5 is about 3 times slower than RANU4. This is quite satisfactory, considering the complexity of the algorithm and the high speed of RANU4. Some improvements in the speed of RANU5 may be possible. Note that our implementation of RANU5 does not include a combination with another generator (as suggested by Marsaglia) – this would obviously slow it down.

We have implemented a multiprocessor version of RANU4 on the VPP500. We have developed theory that ensures that sequences of pseudo-random numbers returned for different initial seeds are separated by a distance greater than 10^{18} in the full periodic sequence. Thus, for all practical purposes, different initial seeds ensure different subsequences of pseudo-random numbers. The requirement that each processor generate disjoint subsequences is thus equivalent to the requirement that they use different initial seeds. In our implementation on the VPP500, then, each processor uses a different seed (based on the processor number), thus ensuring that there is no overlap between the sequences used on different processors.

Extensive statistical testing has been performed using a variety of testing methodologies. RANU4 has been shown to pass these at least as well as previous random number generators, and in some cases (i.e. linear congruential generators) the results are orders of magnitude better. Testing for performance was also done. On each processor, 1.6×10^9 pseudo-random numbers were generated in batches of 800,000.

With these settings, each processor generated random numbers at a rate of 0.581 cycles per random, or one random per 2.2 cycles. The rate of 0.581 cycles per random can also be expressed as 0.17×10^9 randoms per second per processor and this can be compared with the rate of 0.14×10^9 randoms per second for the VP2200 implementation.

3 Parallel fast Fourier transforms

We have implemented a suite of fast Fourier transform algorithms, including real and complex one-, two- and three-dimensional data. These were written in VPP Fortran and extensively tested on the AP1000 and the VPP500. In addition, to get a more realistic idea of parallel performance on the AP1000, communication steps as translated by the VPP Fortran compiler were replaced with much faster calls to the native AP1000 communication library, and the code was retested.

For the complex to complex transform we used the "4-step" or transpose algorithm [23]. A new algorithm based on the 4-step algorithm was developed for the real transforms [14]. The 4-step algorithm is well adapted to systems with a moderate number of processors like the Fujitsu VPP 500 and the AP1000.

One-dimensional data is stored in a two-dimensional array which is then partitioned over the first dimension. Two- and three-dimensional data is stored in two- and three-dimensional arrays which are partitioned over the first dimension as well. We chose blocked partioning, however, the algorithms do not depend on the partitioning.

The building blocks of the 4-step algorithms are completely local complex one-dimensional (multiple) Fourier transforms, and communication intensive matrix transpositions. The local complex transforms implement a new mixed-radix in-place and self-sorting algorithm with radices 2,3,4,5,8 and 16, see [15, 12, 13].

The performance on an 11 processor VPP 500 is 5.8 Gflop/s, about a third of peak performance, for a problem size $n = 2^{22}$. For the same sized problem on a 128 processor AP1000 we get about a sixth of peak performance. This compares favorably with other parallel FFT routines which in our tests achieve approximately 10% of peak speed. It also shows that the nodes of the VPP500 can sustainable high performance. This problem points to another major strength of the VPP500 architecture: the crossbar switch. The algorithms are organized so that all communication is done at one step, in a personalized

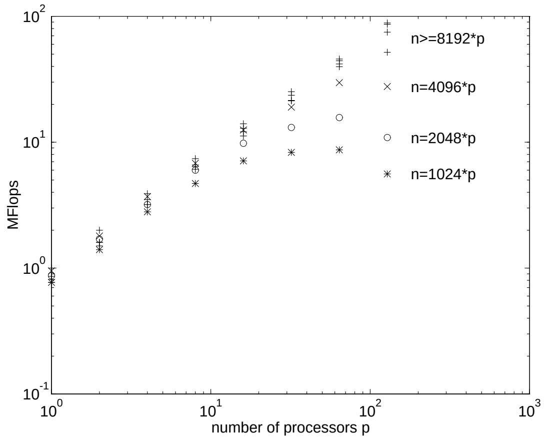

Figure 1: Performance of FFT algorithm on AP1000

all-to-all communication operation. This operation is done with large chunks of memory, which is typically more efficient for message passing machines (which at the bottom levels the VPP500 essentially is). But the fact that all processors are simultaneously sending and receiving data is a real test of the crossbar switch technology. Measurements of this communication phase show that with large blocks this operation can approach (to within a factor of 2/3) the theoretical maximum bandwidth of 400 Mbytes/second for a write operation (200 Mbytes/second for a read operation) even with all processors simultaneously sending messages. Testing this operation on a larger VPP500 should prove very interesting.

Theoretical analysis and practical tests [15] demonstrate the scalability of the algorithm. Thus on a fully equipped VPP 500 we would expect around 100 Gflop/s for large enough problem sizes.

4 Least Squares Problems

The linear least squares problem [20] is

$$\min_{\mathbf{x}} \mathbf{r}^{\mathbf{T}}\mathbf{r}; \quad \mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}, \tag{1}$$

where $A : R^p \rightarrow R^n$, $p < n$, and the solution is unique if $\text{rank}(A) = p$. If $\text{rank}(A) < p$ then the solution is not unique, but it can be made so by seeking the solution closest to the origin in R^p. Typically, the computational cost in solving (1) is $O(np^2)$ flops. Even for rather large data fitting problems – say $n = 100000$ and $p = 100$ – the processing load is of the order of a gigaflop and insufficient on its own to tax the VPP500.

It becomes more significant when it has to be called iteratively in performing a large nonlinear least squres analysis. However, the linear least squares problem does lend itself to parallel and vector implementation.

We use the so called Modified Gram-Schmidt (MGS) tableau-based algorithm, which computes

$$U^{-T}A^T \rightarrow Q_1^T, \quad \mathbf{x} = \mathbf{U^{-1}Q_1^T b}. \tag{2}$$

MGS proceeds by orthogonalizing the current column A_i to $A_{i+1}, \cdots, A_p$ followed by a normalising step, for $i = 1, 2, \cdots, p-1$. Consider the tableau

$$W = \begin{array}{cc} A^T & I_p \\ -b^T & 0 \end{array}. \tag{3}$$

The MGS orthogonalisation steps are applied to W in a sequence of p sweeps denoted by $\overrightarrow{(p)}$. This gives

$$W_{\overrightarrow{(p)}} = \begin{array}{cc} Q_1^T & U^{-T} \\ \mathbf{r^T} & \mathbf{x^T} \end{array}. \tag{4}$$

This is verified readily. Orthogonalising $\mathbf{b}$ to each of the columns of A in step with the MGS orthogonalisations necessarily produces $\mathbf{r^T}$ in the first n places of the last row of W. It follows from (2) that the orthogonalisation process must introduce U^{-T} into the tableau replacing the unit matrix. But from (1) $\mathbf{r^T}$ is given also by $[\mathbf{x^T}\ \mathbf{1}]\mathbf{W}$. Thus the remaining elements in the last row of W are just the components of $\mathbf{x^T}$.

This results in a very simple program, yet one that can be both parallelized and vectorized. Our parallel algorithm for the VPP500 divides the tableau vertically. The bulk of the computations can be performed without communication. The only sources of communication are global sums across all processors at each step. VPP Fortran provides an intrinsic construct for performing such a sum; however, at present this routine seems to be somewhat unoptimized. For this routine as well as several others, efficient implementation of this small-grained communication operation is very important.

5 Tridiagonal eigenvalue problem

We assume that the tridiagonal matrix can be stored on each processor, since each processor has either 128 or 256 Mbytes of storage. With this assumption, we attack this problem on the VPP500 in the following way:

1. Parallelism is sought in the separate eigenvalue calculations in order to spread the computation across the processors; and
2. Techniques for vectorizing the calculation of the distinct eigenvalues are applied on each processor.

This way there is *no* interprocessor communication once the tridiagonal matrix has been distributed to each processor until the results are collected. The scalability relation for this approach has the form

$$TC = S + \frac{n_{ev}}{n_p}C \tag{5}$$

where TC is the total cost of the computation, S is the start–up overhead (essentially broadcast time), C is the cost of a single eigenvalue computation on a single processor, n_{ev} is the number of eigenvalues required, and n_p is the number of processors. A key parameter here is n_{ev}. Typically what is required is a (small) subset of the eigenvalues (and certainly of the eigenvectors) of a tridiagonal matrix of a size that would justify sensible computation on a machine of the capability of the VPP500. For this reason methods which seek all the eigenvalues, such as the Cuppen algorithm [8], would seem *a priori* unattractive even if the problems of interprocessor communication could be solved as effectively.

The Sturm sequence property of the principal minors of $T - \lambda I$ provides a suitable method for isolating the eigenvalues of interest on each processor provided these are specified by order information. It is known [24] that a multisection technique in which the Sturm sequence is evaluated at k distinct values of λ in each pass in order to isolate each distinct eigenvalue is appropriate. The optimal value of k satisfies

$$k(\log k - 1) = n_{1/2}, \tag{6}$$

where $n_{1/2}$ refers to the vectorization over j, the index of the multisection points, of the recurrence for the ratio of the principal minors. That is, vectorization is over the different points at which the Sturm sequence is to be evaluated.

$$q(j) = (T_{ii} - T_{i(i-1)}T_{(i-1)i}/q(j)) - \lambda(j). \tag{7}$$

- It follows from 6 that k increases only marginally more slowly than $n_{1/2}$ which is typically large ($O(10^2)$ or more) on the Japanese supercomputers. In this case there is no great advantage in attempting to accelerate the convergence of the eigenvalues by using a higher ordered scheme once these have been isolated. Thus it suffices to apply the multisection procedure a fixed number of times to achieve full machine accuracy.
- The bracketing of the eigenvalues by counting the changes in sign in the sequence of Sturm polynomials from 7 can use a result of Kahan [9] that the

# of points	elapsed time (s.)
40	.0597
80	.0536
85	.0540
90	.0541
95	.0506
100	.0522
105	.0548
110	.0551
120	.0570
160	.0595

Table 1: Execution time on VPP500 vs. multisection points per processor for n=1000

> sign count is monotonic in λ in IEEE arithmetic provided the bracketed order of computation is imposed.

The technique used for vectorizing the multisection computation over j is described by us in [7].

Table 1 shows the effect of varying the number of multisection points per processor on total elapsed time. The times given are for a tridiagonal matrix T, $T_{i(i-1)} = T_{i(i+1)} = 1$, $T_{ii} = 0$, of order $n = 1000$. The computations were carried out on 4 processors of the VPP500, with each processor finding 4 eigenvalues. There is a tradeoff between vectorization and convergence in deciding how many multisection points to use per processor, as can be see from the table. With more points, vectorization increases, but more Sturm sequences must be evaluated to achieve the same accuracy as compared to using less points.

Allowance is made for pathologically close eigenvalues which can be identified as numerically repeated eigenvalues. The first eigenvector associated with such a cluster is found by inverse iteration using the sign structure of the Sturm sequence to set an initial eigenvector. For each subsequent eigenvector computation associated with the cluster a starting vector is found by generating a vector with random elements and then orthogonalising this to the vectors already computed and rescaling. This has proved satisfactory for generating an independent basis for the eigenspace, but there can be very noticable loss of orthogonality. For example, the routine when applied to find the 6 largest eigenvalues of W_{35} [25] locates three numerically repeated pairs. The corresponding eigenvectors for each pair are found in one step of inverse iteration, but are orthogonal only to within 1%.

6 Iterative Methods

For the solution of sparse, symmetric linear systems, we are investigating a variety of preconditioned Conjugate Gradient methods. The methods implemented so far are Neumann, Block Neumann and block Incomplete Cholesky (IC). The block size is chosen so that there is no interprocessor communication. Thus if a matrix A is partitioned so that a processor has rows from n1 to n2, the preconditioning matrix M is set to zero outside that column range. This is just a block Jacobi, but with a large block size. Such preconditioners trade off better parallel properties against slower convergence, since information is not propagated from processor to processor in the preconditioning stage. This structure has the advantage for IC that the preprocessing steps, such as determination of a "wavefront" ordering and any re-ordering of equations, as well as the factorizations themselves, may all be done on each processor in parallel [22].

The sparse matrix storage format chosen was the "Purdue" format, which has long vector lengths (down each row) and partitions naturally, each processor having $n/nprocs$ rows. This format gives compatibility with other conjugate gradient packages as well as with our unsymmetric iterative solvers.

Results for a simple test problem, the 2D Laplace's equation on a square domain, are as follows:

```
n=512000 nprocs=4

Precond.      Iterations  Mflops    Time (s.)
none           2956        1737        15.7
Neumann        1479        1517        12.9
Block Neumann  1637        1541        14.1
" (order 3)    1223        1424        18.5
Block IC       991         713         20.6
```

These results are very encouraging, except for the Block IC. An examination of the time of each stage reveals, as expected, that it is the preconditioner solve that is slow:

```
             Times in seconds
Prec.        None   Neumann    IC       IC
Stage                                 Mflops
factor        -       .006     .5
wave          -       .6       .17
init         .003    4.4       .02
send p       1.3      .6       .4
Ap           8.9      .9      3.0      1358
p'Ap         1.3      .6       .4      2342
2 Saxpy's    1.9      .9       .6      3121
```

r'z	1.2	.6	.5	2018
1 saxpy	.9	.5	.3	3169
send prec	0	.6	0	
solve prec.	0	4.5	14.6	348

The slow solve step is due to too many levels of indirect addressing in the following code.

```
do k = 2, numwav
    do L = ist(k), ist(k+1) - 1
        i = ip_L
        y(i) = y(i) - A(i,j) * y( COL(i,j) )
```

The remedy here is to re-order the equations using the permutation vector ip, which has not yet been implemented.

Another problem on the VPP is to implement the indirect addressing in the matrix vector product $A \times x$:

$$y_i = y_i + A_{i,j} * x(COL_{i,j})$$

The matrix A is partitioned row-wise between processors, which causes no problems, but the vector x is partitioned in the same way, and since we know nothing about the structure of the matrix A we cannot determine what communication pattern will be needed to implement x($COL_{i,j}$). Thus, there is no way of efficiently moving data between processors based on such indirect addressing. Under investigation are more sophisticated techniques for this operation which may require a different mapping of the matrix A to the processors; see e.g. [17].

Unsymmetric Problems:

As a preliminary part of this project, we have developed a variant of conjugate-gradient type methods for nonsymmetric systems of equations called the MGCR method [18]. MGCR is always faster than GMRES, i.e. at each iterative step MGCR performs one scalar-by-vector multiplication and one vector-to-vector addition less than GMRES, and it is as robust as GMRES. MGCR cannot break down unless it has already converged or the exact solution is found. As with the symmetric iterative methods, the matrix A is represented in Purdue format but preliminary computations indicate that the product A*x is slow. Achieving good performance on both of these problems will depend on finding an efficient implementation of parallel sparse matrix-vector multiplication.

7 Banded and Dense Linear Systems

The solution of dense and banded linear systems with no interior sparsity is a standard functionality for scientific subroutine libraries. On the VPP500, we have chosen to use the so-called *torus wrap mapping* [2, 16] which was first described for banded matrices by Fox [11], as our main data mapping for implementing these programs. The torus wrap mapping assumes that the processors are configured into a two-dimensional grid of size $NPCOL \times NPROW$ and that each processor is labelled by its coordinates in this grid (starting with zero). The mapping is determined in terms of the processor grid by the following rule: element i, j of the matrix is assigned to processor $(i \bmod NPCOL, j \bmod NPROW)$. In effect this scatters both the rows and columns of the matrix. This is a good match with the ability in VPP Fortran to declare data to be distributed with the CYCLIC characteristic.

The torus wrap mapping provides a theoretical savings in volume of communication and an improvement in scalability of a factor of $\sqrt{P}$ over row and column methods [2], and has been shown in practice to provide highly scalable and efficient programs for a variety of architectures. Thus, it has been selected as the primary mapping for the ScaLAPACK project [4]. Though the VPP500 currently in use commercially do not yet have enough processors to make this theoretical savings applicable, we chose the torus wrap mapping both because we anticipate that Fujitsu's successors to the VPP500 will emphasize more parallelism than the current generation, as well as to maintain consistency with other projects using the torus wrap mapping like ScaLAPACK.

For current users with machines in the low to middle number of processors, an efficient LU factorization code has been written by Mr. Makoto Nakanishi. This code divides the matrix into column strips which are each assigned to a single processor. Computation and communication are done in block portions of these strips, allowing both the overlapping of communication and computation by configuring the processors logically in a ring, as well as efficient matrix-matrix single processor kernels. For the LINPACK benchmark problem, results for this code include 1.26 Gflops on a single processor and 32.93 Gflops on 32 processors, a parallel efficiency of 77%. See Table 7.

We have chosen to use the same basic algorithm for the banded matrices as for the dense systems. The assumption here is of medium to large bandwidths, such that vectorization/parallelization within the band is reasonably efficient. At the other end of the band-

P	N	Gflops	Efficiency
1	1000	1.26	1.00
2	2000	2.09	0.83
4	5200	4.46	0.88
8	7600	9.22	0.90
16	9984	16.89	0.84
32	14720	32.93	0.77

Table 2: Linpack Benchmark performance on VPP500

width spectrum, there are plans to implement special cyclic reduction type algorithms for tridiagonal and pentadiagonal matrices, and perhaps block bidiagonal matrices. In the middle of the spectrum are small to medium bandwidths for which the best option seems to be partitioning methods (of which cyclic reduction is a special example). Cleary [5] showed that for symmetric positive definite matrices, any partitioning method will have an operation count of at least four times that of the sequential method. Thus, unless the partitioning algorithm allows a computational rate (including parallelism) that is four times the sequential method, it cannot be superior. Matrices with such a small bandwidth are extremely sparse, and thus are better dealt with in the context of sparse matrices. A continuing area of this project is the solution of sparse linear systems. To handle narrowly banded matrices, we need only provide a special reordering module for these matrices and the sparse matrix programs will affect the partitioned factorization automatically.

First versions of our Gaussian elimination and Cholesky factorization codes haves shown that the computational kernels on each processor vectorize well and are completely load-balanced across processors, as expected. For $N = 4000, \beta = 1000$, the computational kernel for Gaussian elimination runs at 1.2 Gflops on a single processor. This is consistent with a peak speed of 1.6 Gflops, $n_{1/2} \approx 450$, and an average vector length of $(2/3)N$. However, whereas a sequential code has essentially no overheads, the parallel code running on a single processor has overheads that slow the code down to an average .8 Gflops. These overheads are accurate reflections of the cost of communication and synchronization in VPP Fortran, as the compiler does not recognize at compile time that only one processor is being used and thus generates full communication and synchronization calls even though they are not needed.

Naturally it is impossible to show near optimal speedups with such overheads in the parallel code. However, our code does show speedup on 2 and 4 processors when compared to the parallel code running on a single processor. The problem is centered around VPP Fortran and implementing message-passing type operations. VPP Fortran was originally designed to be a data-parallel type language, but for library software in which the number of processors is not known at compile time, most of the data-parallel facilities are unusable. Thus, programming practice degenerates into a message-passing type structure, but without the full support of message passing instructions. The result is that some operations are difficult to implement. In particular, the notion of sending a nonblocking message to another processor is particularly cumbersome, involving substantial handshaking and thus stripping asynchronicity out of the algorithm. While we are working on solutions and improvements within the confines of VPP Fortran, we also have hope that continuing efforts by Fujitsu may lead at least to HPF and possibly to a standard message passing functionality.

8 Summary

We have described our efforts at providing high impact scientific subroutines in the form of library software for Fujitsu's VPP500 parallel-vector supercomputer. The choice of target problem areas is determined both by users both directly via their input and indirectly by comparison with other parallel libraries. Several areas have been largely completed including random number generators, fast fourier transforms, and dense least squares problems, while many others are in various stages of completion. Though time with the VPP500 has been limited and even then mainly to a four processor machine, results to date have been encouraging. The excellent performance of the node processors, rated at 1.6 Gflop/s, allows very large and complex problems to be attacked, while the unmatched bandwidth of the communications crossbar switch (800 Mbyte/s/processor) prevents large blocks of communication from becoming a bottleneck. We need more experience with the machine to make conclusions about the communication of small chunks of data as well as synchronization mechanisms useful in smaller grained applications. Likewise, besides for analyses and the parameters we have calculated so far, we cannot comment on the possible performance of the VPP500 with large numbers of processors.

In addition to the areas we have mentioned already, we currently have preliminary efforts underway in other areas. A large effort at providing direct solvers for sparse matrices has just recently started but is basically still in the investigation stages. As part of the preliminary research in this project, a new mapping for sparse matrix factorization was developed that offer attractive communication advantages for massively parallel machines [6], but there are still questions as to whether this mapping is appropriate for the smaller number of processors on the VPP500 since communication is not such a dominant concern. Further efforts at various eigenvalue problems have been made, including the implementation of Jacobi methods for symmetric and unsymmetric problems and development of localization methods for small subsets of the eigenvalues of unsymmetric matrices, but these codes are not yet complete.

References

[1] S. L. Anderson, "Random number generators on vector supercomputers and other advanced architectures". *SIAM Review* 32 (1990), 221-251.

[2] C. Ashcraft. The distributed solution of linear systems using the torus wrap data mapping. Technical Report ECA-TR-147, Boeing Computer Services, October 1990.

[3] R. P. Brent, Uniform random number generators for supercomputers. *Proc. Fifth Australian Supercomputer Conference*, Melbourne, December 1992, 95-104.

[4] J. Choi, J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. Technical Report 53, LAPACK Working Note, 1993.

[5] A. Cleary. Parallelism and fill-in in the Cholesky factorization of reordered banded matrices. Technical Report SAND90-2757, Sandia National Labs' MPCRL, 1990.

[6] A. Cleary. A new torus-like mapping of sparse matrices for parallel matrix factorization. In *Proceedings of the 6th SIAM Conference on Parallel Processing for Scientific Computing*, 1993.

[7] A. Cleary and M. Osborne. Eigenvalue solvers for large problems. In *Proceedings of the Computational Techniques and Applications Conference*, July 1993. Canberra, Australia.

[8] J.J.M. Cuppen. A divide and conquer method for the symmetric eigenproblem. *Numer. Math.*, 36:177–195, 1981.

[9] J. Demmel and W. Kahan. Accurate singular values of bidiagonal matrices. *SIAM J. Sci. Stat. Comput.*, 11(5):873–912, 1990.

[10] J. Dongarra. Performance of various computers using standard linear equations software. July 1993.

[11] G. Fox. Square matrix decomposition — Symmetric, local, scattered. CalTech Publication Hm-97, California Institute of Technology, Pasadena, CA, 1985.

[12] Markus Hegland. An implementation of multiple and multi-variate Fourier transforms on vector processors. *SIAM J. Sci. Comput.*, 1993. accepted.

[13] Markus Hegland. On some block algorithms for fast Fourier transforms. Technical Report CMA-MR51-93, CMA, Australian National University, 1993.

[14] Markus Hegland. Parallel FFTs for real and complex sequences. Technical report, CMA, Australian National University, 1994.

[15] Markus Hegland. A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing. *Numerische Mathematik*, 1994. accepted.

[16] B. Hendrickson and D. Womble. The torus-wrap mapping for dense matrix calculations on massively parallel computers. SAND Report SAND 92-0792, Sandia National Laboratories, Albuquerque, NM, 1992.

[17] J. Lewis and R. van de Geijn. Distributed memory matrix-vector multiplication and conjugate gradient algorithms. In *Proceedings of Supercomputing '93*, pages 484–492. ACM, 1993.

[18] Z. Leyk. *Modified generalized conjugate residuals method for nonsymmetric systems of linear equations*, ANU Centre for Mathematics and its Applications Research Report no. CMA-MR33-93. 1993.

[19] G. Marsaglia and A. Zaman. A new class of random number generators. *The Annals of Applied Probability*, 1:462–480, 1991.

[20] M. R. Osborne, *Gram–Schmidt for Least Squares Regression Problems,* Rep. SMS-015-90, School of Mathematical Sciences, Australian National University, 1990.

[21] W. P. Petersen. *Lagged Fibonacci Series Random Number Generators for the NEC SX-3.* IPS Research Report No. 92-08, IPS, ETH-Zentrum, Zurich, April 1992.

[22] M. Seager. *Parallelizing the conjugate gradient for the CRAY X-MP.* Parallel computing 3 (1986), p35.

[23] Charles Van Loan. *Computational Frameworks for the Fast Fourier Transform.* SIAM, 1992.

[24] H. Simon. Bisection is not optimal on vector processors. *SIAM J. Sci. Stat. Comput.*, 10(1):205–209, 1989.

[25] J.H. Wilkinson. *The Algebraic Eigenvalue Problem.* Claredon Press, Oxford, England, 1965.