Development of a Mathematical Subroutine Library for Fujitsu Vector Parallel Processors

R. Brent, L. Grosz, D. Harrar II, M. Hegland, M. Kahn,
G. Keating, G. Mercer, O. Nielsen, M. Osborne, B. Zhou
Australian National University
Canberrra ACT, 0200, Australia
and
M. Nakanishi
High Performance Computing Group, Fujitsu Ltd.
Numazu-Shi Shizuoka, 410-03, Japan

Abstract

The Fujitsu-ANU Parallel Mathematical Subroutine Library Project is a joint research program involving staff at the Australian National University and Fujitsu Japan. The aim of the project is to produce a library of mathematical subroutines for the vector-parallel Fujitsu VPP300 which result in high performance and accuracy on large problems. In order to utilise the architecture of the VPP300 it is necessary to develop new algorithms for many of the standard numerical problems.

1 Introduction

The Fujitsu-ANU Parallel Mathematical Subroutine Library Project provides high performance mathematical subroutines for inclusion in the SSLII mathematical subroutine library for Fujitsu vector and vector/parallel supercomputers [12], in particular, for the Fujitsu VPP300. Because of the combination on the VPP300 of several high performance vector processors connected in a distinctive parallel architecture, standard parallel numerical algorithms may not perform well. Thus the major work on the project has been in the development of new algorithms which achieve high performance through optimal use of the computer architecture.

An announcement of the library was given in [7] in 1994. Since then the functionality of the library has been substantially extended. In this paper we will refer only briefly to the topics covered in [7] and concentrate on more recent developments. A series of working notes is available on the WWW at

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

The next section gives some details on the architecture of the VPP300 and a discussion of the issues which must be considered when writing high performance code.

In the remaining sections of this paper several research areas in the project are highlighted and a brief discussion of the algorithmic development required to produce high performance code for each is given.

2 The Fujitsu VPP300

2.1 Architecture

The ANU VPP300 has a peak speed of about 29 Gflops with 14 Gbytes of memory based on a uniform 7ns clock rate.

Each processing element (PE) consists of the following:

- A low performance scalar unit (SU) which has a long-instruction-word (LIW) RISC CPU achieving approximately 100Mflops peak using simultaneous scalar, VU and DTU instructions. Use of this unit must be minimzed whenever possible.
- 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 per 7 cycle). It is also possible to chain load-mult-add allowing a peak of 2.2 Gflops (matrix multiply achieves 2.18Gflops). The vector registers are configurable in "power-of-two" steps between 256 registers of length 64 words to 8 registers of length 2048 words. The actual configuration is optimised by the compiler depending on computational density within a particular loop.
- Memory (MSU) with 2GB of SDRAM memory on 5 of the PEs and 512MB on the remainder. This is a substantial amount of memory per processor and can lead to near peak performance on long vector lengths.
- A data transfer unit (DTU) for direct memory access data communication to the interprocessor network. Communication between processors is via a crossbar switch (all processors are "equidistant" from one another) with a peak bandwidth of 570MB/s bidirectional and an achievable latency of about 5 μsecs. Because of the crossbar connection parallel algorithms are not constrained to using communication to neighbouring processors only as all processors are equally "close". This removes a major restriction from coding parallel algorithms.

Parallelism in an algorithm is achieved in several ways. At one level parallelism is obtained from the vector operations. This can be further exploited by coding so that several of the different pipes are being used at the same time. At the top level, MIMD parallelism is used by having several of the processors working simultaneously.

Basically, the design of the Fujitsu VPP series favours algorithms which use vector operations with very long vector lengths and stride one data access. In addition, the crossbar switch provides a very flexible communication mechanism so that in many cases one gets best performance by regrouping communication such that the number of communication steps used is minimised.

Many standard techniques display performance degradation caused by short vector lengths, non-unit stride data acess or complicated communication patterns requiring many steps. Examples include the Stockham algorithms for FFTs, divide-and-conquer methods for the solution of banded linear systems, standard random number generators and eigenvalue algorithms. Other common algorithms do not vectorise at all like the tridiagonal QR algorithm for eigenvalues.

For all these cases new algorithms and implementations have been found which optimise the use of the hardware. Performance tests show that attention to these hardware characteristics leads to substantially improved performance for all the cases mentioned above. This hardware focus, however, is not reflected in the interface to the routines so that users experience the higher performance but use similar calling commands as required by standard parallel algorithms.

2.2 VPPFortran

Library code to be delivered as part of the project is written using VPPFortran. Although it is common practice to use message-passing languages such as PVM or MPI to develop parallel algorithms, the parallel routines of the SSLII library do not use these. VPPFortran is basically FORTRAN 90 plus compiler directives to promote parallel layout of data, communication between processors, global sums, synchronization and so on. There are some similarities between VPPFortran and HPF especially in the mapping of index domains to processors but the programmer has more control in VPPFortran. This is because the compiler directives can be used to achieve both data and task distribution. This provides extra flexibility similar to HPF2.0 with approved extensions [21].

3 Linear Systems of Equations

A variety of solvers for systems of linear equations have been included in the library. Different techniques are necessary to achieve the best possible performance for different systems. For banded matrices with a large bandwidth a torus-wrap mapping of the data to the processors is used to give high performance algorithms for both symmetric and nonsymmetric matrices. This mapping can also be used for dense matrices. An alternative approach for dense matrices uses a blocked modified Cholesky for the symmetric case and a blocked LU decomposition for the nonsymmetric case. Both of these approaches were summarised in [7].

For more sparse matrices such as narrow-band matrices, for example, block bidiagonal matrices, a wrap-around partitioning technique is used to achieve high performance on the VPP300.

A suite of iterative solvers for both symmetric and nonsymmetric matrices has been developed for different sparse matrix storage schemes as well as routines which use a reverse communication interface to allow the user to provide application dependent subroutines for matrix/vector multiplication and preconditioning. More recent work has concentrated on developing an algebraic multilevel preconditioned conjugate gradient method.

3.1 Wrap-around Partitioning

Wrap-around partitioning is a reordering of the unknowns in a system of linear equations into q blocks of p unknowns each for the purpose of highlighting groups of unknowns that can be eliminated independently of one another in order to permit vectorization of the elimination process [19]. Its natural formulation is for block bidiagonal matrices. It can be applied to narrow banded matrices of sufficiently regular structure with a minimum of reorganisation. In this case special structure can be exploited in the first pass only [11]. Speed ups of order 20 times scalar speed can be obtained fairly easily for small matrix subblock size m ($m = 2$ in the tridiagonal case) and $n > 1000$. Wrap-around partitioning has the important advantages:

- It is not necessary for p and q to be exact factors of n. This means that there is no requirement to use the power of 2 strides associated with cyclic reduction with consequent memory access degradation. Power of 3 stride is recommended;
- The block bidiagonal structure is preserved under the use of stable factorization techniques and so can be applied recursively. Our code employs orthogonal transformations because stability of Gaussian elimination with partial pivoting cannot be guaranteed for the full generality of block bidiagonal matrices [36].

Consider for example the case $n = 12$. The block bidiagonal matrix is illustrated in Figure 1. The first stage of reordering is illustrated for $q = 2$, $p = 5$ to make the point that exact factors of n are not required and the result of the transformation shown. The first stage of elimination removes the B blocks with suffices $\{1, 3, 5, 7, 9\}$. Each elimination is independent of the others and so can be vectorized. However, fill-in occurs if either pivoting is allowed in Gaussian elimination, or orthogonal factorization is used. The result is shown in Figure 1. Here blocks that remain nonzero are shown by *, those deleted by $\varnothing$, and those where fill is introduced by an integer indicating the stage fill first occurred. The key point is that the submatrix containing the uneliminated unknowns is again block bidiagonal so the sequence of wrap-around partitioning and elimination can be applied recursively. The recursion is terminated when the matrix of the remaining unknowns is too small to benefit from vectorization. This subproblem is solved sequentially.

3.2 Iterative Methods

The solution x of a large system of linear equations

$$Ax = b \tag{1}$$

with a sparse coefficient matrix $A \in R^{n \times n}$ is required within many practical applications (e.g. discretization of partial differential equations (PDEs)). Iterative methods of the conjugate gradient type (CG) have proved to be suitable algorithms as they are robust and can be efficiently implemented on parallel and vector computers. However, as there is no one CG method that is optimal for all matrices, several CG methods have been implemented (classical CG,

$$
\left|\begin{array}{cccccccccccc}
A_1 & & & & & & & & & & & D_1 \\
B_1 & A_2 & & & & & & & & & & \\
 & B_2 & A_3 & & & & & & & & & \\
 & & B_3 & A_4 & & & & & & & & \\
 & & & B_4 & A_5 & & & & & & & \\
 & & & & B_5 & A_6 & & & & & & \\
 & & & & & B_6 & A_7 & & & & & \\
 & & & & & & B_7 & A_8 & & & & \\
 & & & & & & & B_8 & A_9 & & & \\
 & & & & & & & & B_9 & A_{10} & & \\
 & & & & & & & & & B_{10} & A_{11} & \\
 & & & & & & & & & & B_{11} & A_{12}
\end{array}\right|
$$

$$
\left|\begin{array}{cccccccccccc}
A_1 & & & & & & & & & & & D_1 \\
 & A_3 & & & & B_2 & & & & & & \\
 & & A_5 & & & & B_4 & & & & & \\
 & & & A_7 & & & & B_6 & & & & \\
 & & & & A_9 & & & & B_8 & & & \\
B_1 & & & & & A_2 & & & & & & \\
 & B_3 & & & & & A_4 & & & & & \\
 & & B_5 & & & & & A_6 & & & & \\
 & & & B_7 & & & & & A_8 & & & \\
 & & & & B_9 & & & & & A_{10} & & \\
 & & & & & & & & & B_{10} & A_{11} & \\
 & & & & & & & & & & B_{11} & A_{12}
\end{array}\right|
$$

$$
\left|\begin{array}{cccccccccccc}
* & & & & & 1 & & & & & & * \\
 & * & & & & * & 1 & & & & & \\
 & & * & & & & * & 1 & & & & \\
 & & & * & & & & * & 1 & & & \\
 & & & & * & & & & * & 1 & & \\
\varnothing & & & & & * & & & & & & 1 \\
 & \varnothing & & & & 1 & * & & & & & \\
 & & \varnothing & & & & 1 & * & & & & \\
 & & & \varnothing & & & & 1 & * & & & \\
 & & & & \varnothing & & & & 1 & * & & \\
 & & & & & & & & & * & * & \\
 & & & & & & & & & & * & *
\end{array}\right|
$$

Figure 1: Steps in wrap-around partitioning for a block bidiagonal matrix.

GMRES, TFQMR, LSQR, for more details see [7]) to enable the user to select the optimal method for his problem.

The sparse coefficient matrix may be given in the diagonal storage scheme or the ELLPACK storage format (see [12]). In the diagonal storage scheme (typically used for discretization on a rectangular grid) any diagonal of the matrix A that contains a non-zero element is stored on the processors as a banded partioned vector. The matrix vector multiplication is the basic operation in all CG method and achieves more than 1Gflop per processor by unrolling over the number of diagonals. For a discretization on an unstructured grid the ELLPACK storage format has proved to be more efficient as it uses a more general concept of a diagonal by introducing an additional index vector. Although this slows down the matrix vector multiplication (because of indexed load operations in the linked triad) the ELLPACK storage format is more efficient than the diagonal storage scheme if the matrix contains a large number of sparse diagonals.

To improve the robustness and convergence behaviour of the CG iteration a preconditioner can be used, ie. instead of system (1) the iteration method is applied to the equivalent system

$$M^{-1}Ax = M^{-1}b \tag{2}$$

with a suitable, non-singular matrix M. The intention is that the new iteration matrix $M^{-1}A$ will have better numerical properties than A.

Recent developments for SSLII use the algebraic multilevel iteration (AMLI, see [2]). In the same way as multigrid methods, AMLI build up smaller coefficient matrices $(M^{(k)})_{k=0,\dots,l}$ with $M := M^{(0)}$ but, in contrast to multigrid techniques, only information available from the given matrix A is used.

For $k = 0,\dots,l-1$ a subset $C^{(k+1)}$ of the unknowns $C^{(k)}$ of level k is selected. The coefficient matrix $A^{(k)}$ for level k is split in the following way:

$$A^{(k)} = \left[\begin{array}{cc} A_{FF}^{(k+1)} & A_{FC}^{(k+1)} \\ A_{CF}^{(k+1)} & A_{CC}^{(k+1)} \end{array}\right]. \tag{3}$$

The columns and rows of the submatrix $A_{FF}^{(k+1)}$ belong to the unknowns in $C^{(k)}/C^{(k+1)}$ and those of the submatrix $A_{CC}^{(k+1)}$ to the unknowns in $C^{(k+1)}$. After approximating the inverse of the matrix $A_{FF}^{(k+1)}$ by a diagonal matrix $D_{FF}^{(k+1)}$ the approximate factorization

$$M^{(k)} = \left[\begin{array}{cc} (D_{FF}^{(k+1)})^{-1} & 0 \\ A_{CF}^{(k+1)} & I \end{array}\right] \left[\begin{array}{cc} I & D_{FF}^{(k+1)}A_{FC}^{(k+1)} \\ 0 & M^{(k+1)} \end{array}\right] \tag{4}$$

of the matrix $A^{(k)}$ can be calculated. The matrix $M^{(k+1)}$ becomes an approximation of the Schur complement

$$A^{(k+1)} := A_{CC}^{(k+1)} - A_{CF}^{(k+1)}D_{FF}^{(k+1)}A_{FC}^{(k+1)} \tag{5}$$

by again applying the approximate factorization (4) of the coefficient matrix $A^{(k+1)}$ or, on the lowest level (if $l = k+1$), an approximate solution of a linear system with coefficient matrix $A^{(l)}$.

Within the CG iteration, the procedure for calculating the vector $p := M^{-1}r$ for a given vector r consists of a sequence of block forward substitutions, the solution of a linear system on the lowest level k followed by a sequence of block backward substitutions. That is, $p = \mathrm{AMLI}(0, r)$ with

$$
\begin{array}{l}
\mathrm{AMLI}(k-1, r^{(k-1)}) \\
\quad \text{if } (k-1 = l) \\
\quad\quad \text{solve } A^{(k-1)}p^{(k-1)} = r^{(k-1)} \\
\quad \text{else} \\
\quad\quad (r_F^{(k)}, r_C^{(k)}) \leftarrow r^{(k-1)} \\
\quad\quad q_C^{(k)} \leftarrow r_C^{(k)} - A_{CF}^{(k)}D_{FF}^{(k)}r_F^{(k)} \\
\quad\quad p_C^{(k)} \leftarrow \mathrm{AMLI}(k, q_C^{(k)}) \\
\quad\quad p_F^{(k)} \leftarrow D_{FF}^{(k)}(r_F^{(k)} - A_{FC}^{(k)}p_C^{(k)}) \\
\quad\quad p^{(k-1)} \leftarrow (p_F^{(k)}, p_C^{(k)}) \\
\quad \text{end if} \\
\text{return } p^{(k-1)}\,.
\end{array} \tag{6}
$$

This V-cycle type method does not give a minimal condition number for the iteration matrix $M^{-1}A$ as does the W-cycle type method, but by solving a sufficiently large system at the lowest level l, the computing time for a V-cycle method is optimal, see [26].

The current implementation considers the case for which the coefficient matrix is an M-matrix and is stored in diagonal storage format. Typically these matrices arise from finite difference schemes of order two and finite element schemes of order one on a rectangular grid. The routine independently defines the sets $(C^{(k)})_{k=1,\dots,l}$ of the unknowns for the coarse levels by using an alternating direction technique and includes an optimal selection of the number of levels l. The diagonal matrix $D_{FF}^{(k)}$ is constructed in such a way that all matrices $A^{(k)}$ are M-matrices again which ensures that there is no break down within the factorization (4) and the CG iteration.

No. of unknowns	$0.5 \cdot 10^6$	$1 \cdot 10^6$	$2 \cdot 10^6$	$4 \cdot 10^6$
No. of processors	1	2	4	8
Time in seconds	6.11	7.53	10.1	13.6

This table shows timings for solution of linear systems where the coefficient matrices arise from the finite difference discretization of the differential operator $-\nabla a \nabla$ on the 3-dimensional unit cube. The function a jumps at the surface of the unit sphere from 1 to 10^6. The accuracy of the level of equation is 10^{-6}. The optimal number of levels is about 8 and the number of CG steps about 50. However, in all cases a CG method with Jacobi preconditioner could not return a solution within 10 minutes of computing time.

Further work will improve the computing time for both gather and scatter operations which require communication and reduce the performance of the current implementation. Also ELLPACK format and more general types of matrices will be considered in future versions.

4 Eigenvalue Problems

The first algorithms developed for solving symmetric eigenvalue problems were based on the Jacobi method as this lends itself to parallel implementation. A one-sided Jacobi method to calculate the Singular Value Decomposition of the matrix was used which achieved high megaflop rates because of a specific rotation of blocks of data across processors [37]. The symmetric eigensolver was then built on top of this. The resulting eigensolver was robust and accurate at finding eigenvalues and vectors whatever the spectrum but suffered from a high operation count. Subsequent work has concentrated on developing symmetric eigensolvers based on an initial reduction of the symmetric matrix to tridiagonal form.

4.1 Tridiagonal Eigenvalue Problems

In order to compute eigendecompositions for tridiagonal matrices the conventional procedure is to determine eigenvalues using bisection, based on Sturm sign count evaluation (see, e.g., [28]), then to compute the corresponding eigenvectors via inverse iteration. Vectorization of the eigenvector computation requires that the solution of tridiagonal linear systems be vectorized; this is accomplished using a variant of the wrap-around partitioning described in Section 3.1 - note that a tridiagonal matrix is transformed to block bidiagonal form through a straightforward permutation. Also, since bisection is not optimal on vector processors [32], we use multisection, which entails subdividing an interval known to contain an eigenvalue into greater than two subintervals and/or locating more than one eigenvalue at a time. On the VPP300 computation of one eigenvalue via multisection can be forty times as fast as using bisection.

When all eigenvalues are distinct, the algorithm is embarassingly parallel; a copy of the matrix T is stored on each processor and each can compute eigenpairs without the need for any information from other processors. However, when numerically multiple or clustered eigenvalues have been detected, it is generally necessary to reorthogonalize eigenvector approximations after inverse iteration. In a parallel implementation this reorthogonalization entails considerable communication for vectors stored on different processors. Hence it is necessary to insure that all eigenvectors corresponding to a particular cluster reside on an individual processor. To this end, we attempt to detect clusters during multisection refinement. Depending on the size of the clusters and the initial allocation of subspectra, load-imbalance is likely, and, for large enough clusters, entire processors may be dropped from the computation. Also, as clusters become large, finite precision arithmetic can result in inconsistencies in the way in which different processors have determined clusters to be distributed; hence inter-processor communication of a few integer values is necessary in order to confirm the distribution of clusters. Further details of the multisection algorithm and its implementation can be found in [15], which also contains a brief performance comparison with the corresponding LAPACK and ScaLAPACK routines on a problem from computational quantum chemistry.

4.2 Symmetric/Hermitian Eigenvalue Problems

These routines are based directly on the tridiagonal eigensolver. The reduction to tridiagonal form is accomplished via Householder transformations using a cyclic distribution of columns; in the Hermitian case, the Hermitian matrix is first reduced to complex tridiagonal form, then to real tridiagonal form. These reduction routines are standard and hence are not discussed. Performance gains are attributable to the efficiency of the tridiagonal eigensolver, i.e., to the implementation of multisection and of inverse iteration using wrap-around partitioning.

The following shows typical elapsed times to compute the full eigendecomposition of a real symmetric matrix on multiple processors of the VPP300.

Matrix Size	No. of Processors	Time in secs
4000×4000	4	103
8000×8000	4	633

4.3 Nonsymmetric Eigenvalue Problems

Development work on routines for the solution of general nonsymmetric eigenvalue problems (EVPs)

$$Au = \lambda u, \quad A \in \mathcal{C}^{n\times n}, \ \ x \in \mathcal{C}^n, \ \ \lambda \in \mathcal{C}, \tag{7}$$

has only been initiated relatively recently. Thus far the primary focus is limited to consideration of Newton-based algorithms and Arnoldi methods. Very recently we have also considered the composition of these two methods, and we refer to these as "Arnoldi-Newton methods". More detailed descriptions of the Newton-based and Arnoldi-Newton methods can be found in [27] and [16], respectively; here we only briefly summarize the main ideas.

4.3.1 Newton-Based Methods

We restrict our attention to the standard EVP (7) (though the development in [27] is in terms of generalized EVPs $Au = \lambda Bu$). Let $M(\lambda) = (A - \lambda I) : \mathcal{C}^n \to \mathcal{C}^n$ so that (7) can be written $M(\lambda)\mathbf{v} = 0$; this is embedded in the more general family

$$M(\lambda)v = \beta(\lambda)x, \qquad s^*v = \kappa. \tag{8}$$

As λ approaches an eigenvalue $\bar{\lambda}$, $M(\lambda)$ becomes singular so that the solution v of the first of these equations becomes unbounded for almost all $\beta(\lambda)x$. Hence, the scaling condition, $s^*v = \kappa$, can only be satisfied if $\beta(\lambda) \to 0$ as $\lambda \to \bar{\lambda}$. Zeros of $\beta(\lambda)$ correspond to singularities of $M(\lambda)$ and therefore to eigenvalues of A. Differentiating equations (8) with respect to λ, rearranging terms and simplifying, the Newton update takes the form $\lambda \longleftarrow \lambda - s^*v/(s^*M^{-1}\frac{dM}{d\lambda}v)$, where for (7) $dM/d\lambda = -I$. The vectors s and x can be chosen dynamically; this engenders the possibility of exceeding the characteristic second-order convergence rate of Newton's method [27]. In order to compute more than one eigenvalue a variety of deflation strategies have been considered [27]. The major computational kernel - LU decomposition of M - is handled using extremely efficient software for linear systems [25].

4.3.2 Arnoldi Methods

Arnoldi's method was originally developed to reduce a matrix to upper Hessenberg form; it is a Krylov subspace projection method in which an exact eigenvector $u \in \mathcal{C}^n$ of (7) is approximated by some vector $\hat{u}$ residing in a Krylov subspace, $\mathcal{K}_m = \mathcal{K}_m(A, v_1) \equiv \text{span}\{v_1, Av_1, A^2v_1, \ldots, A^{m-1}v_1\}$, of dimension $m \ll n$.

An orthonormal basis $\{v_1, \ldots, v_j\}$ for $\mathcal{K}_m$ is constructed using modified Gram-Schmidt (MGS) orthogonalization. On forming the $n \times m$ matrix $V_m = [v_1|\ldots|v_m]$, one obtains the projected eigenproblem

$$Hy = V_m^* A V_m y = \hat{\lambda} y, \qquad (9)$$

where $H \in \mathcal{C}^{m\times m}$ is upper Hessenberg and $m \ll n$. The eigenvalues of H can be computed using, e.g. QR. Dominant eigenvalues of H approximate those of A, with the accuracy increasing as m does.

For a broad overview of the many modifications that are typically made to Arnoldi's method in order to increase efficiency, robustness, etc., see, e.g., [31]. Our current implementation uses restarting in conjunction with an implicit deflation process. In order that non-extremal eigenvalues can be computed, a shift-invert procedure is used; this amounts to applying Arnoldi's method to $\hat{A} = (A - \sigma I)^{-1}$ - extreme eigenvalues of $\hat{A}$ correspond to eigenvalues near to the shift σ. Unlike the implementation of the Newton procedures, the Arnoldi routine currently uses only real arithmetic and hence only real shifts.

4.3.3 Arnoldi-Newton Methods

Shift-inversion in the context of the Arnoldi method is exactly equivalent to inverse iteration (factorization of M) in the context of the Newton-based procedure. This is clearly the most computationally expensive portion of each of the algorithms. The Newton-based procedures, though capable of quadratic and even third-order convergence, suffer when good initial data are unavailable. The Arnoldi method drives eigen-estimates toward the exact values from the outset; however, achieving the ultimately desired convergence involves many issues such as what forms of restarting, deflation, spectral transformation, preconditioning, acceleration, etc. should be used, when, and how often? Unfortunately, the answers to these questions may be problem-dependent, and this hampers the development of robust and efficient software. Hence, the drawbacks of these two methods are in a sense orthogonal, and we have recently begun experimenting with Arnoldi-Newton methods. The basic idea so far is to use the Arnoldi method (in real arithmetic only) to get good initial estimates to the (generally complex) eigenvalues and Schur vectors of interest, then to use the Newton-based method to obtain the final eigendata. Ideally, the resulting Arnoldi estimates can be refined to machine precision with only one additional Newton step since this should effectively double the accuracy once quadratic convergence is obtainable. These methods have been implemented and applied to problems from the study of chemical reactions and from fiber optics [16]. Preliminary results are promising.

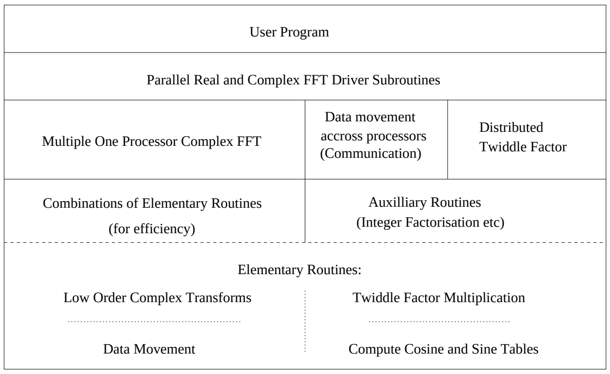


Figure 2: Design of the Fujitsu VPP FFT library

5 Fast Fourier Transforms

It was known to Gauss that an order 12 trigonometric interpolation can be performed by doing 3 order 4 interpolations followed by doing 4 order 3 interpolations [13]. This was generalized to an arbitrary product of relatively prime factors of the order n by Good [14] and to arbitrary factors by Cooley-Tukey [9]. They also suggested the first practical algorithms, the FFTs. The basic idea by Gauss and Good is now called *splitting* and underlies most high-performance FFT algorithms [34]. On the top level, it allows "blocking" and leads to the parallel *4 step algorithm* [33, 18]. Applied recursively, splitting provides the well-known "butterflies" of the FFTs. The butterflies do show the data flow but also expose the basic FFT building blocks which are identified in [17] to be:

- multiple Fourier transforms of small prime order
- simple index digit permutations
- unitary scaling transforms ("twiddle factors")

The Fujitsu FFT library has been systematically designed around these building blocks called the *BAFFS* (BAsic Fast Fourier transform Subroutines). The layered design of the FFT library is displayed in Figure 2.

To get high performance for power 2 transforms, elementary transform subroutines for the orders 2,4,8 and 16 are used. This reduces the number of memory accesses by reusing data in the registers but it also brings the floating point operation count from $5n\log_2(n)$ for the radix 2 algorithms down to $4\log_2(n)$ which is the same operation count as achieved by the split radix algorithm [10]. In the case of mixed radices the operation count was reduced by the application of the prime factor algorithm for small composite order transforms.

For highly composite numbers n there are many possible ways to choose the factors p and q of the order $n = pq$. Thus there are many different splittings possible. The common algorithms choose either p or q to be small [34]. This has the advantage that further splitting has to be done for only one of the factors p or q. However, from a computational point of view, this is not necessarily the best choice. For example, it is shown in [17] that if p and q are chosen to be the same up to a small factor, self-sorting and in-place algorithms result.

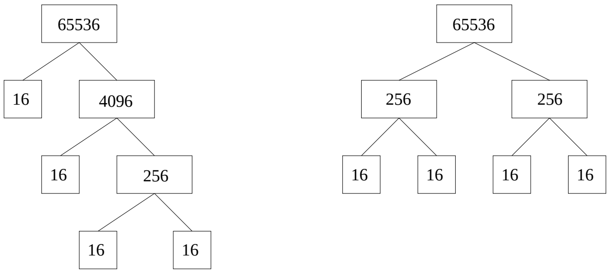


Figure 3: Two recursive splittings of 65536

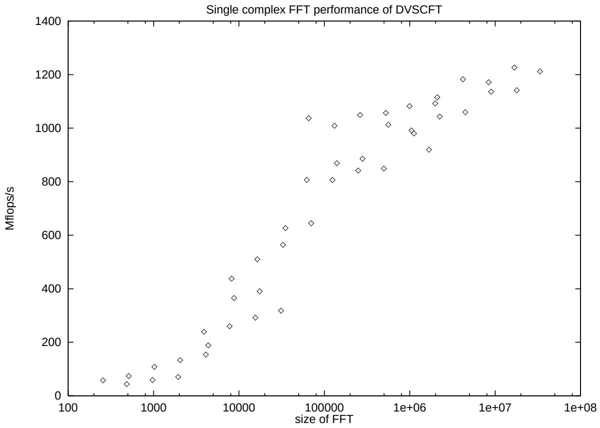


Figure 4: Performance of a 1D complex FFT on one processor

The recursive factorization of n and thus the splitting-based FFT can be represented as a tree as shown in Figure 3. The transform based on the tree on the right of Figure 3 can be done in a self-sorting and in-place way and, in addition, requires fewer operations if performed in parallel as locally only order 256 transforms are done. The method based on the left tree requires locally order 4096 transforms and thus higher order twiddle factors. There is also a possibility to exploit any relative primeness of factors of n [20]. A systematic exploitation of the advantages obtained by careful choice of the splitting is one of the most innovative aspects of the Fujitsu VPP FFT library [22].

In the FFT library all the steps involving communication have been collected together. This has the advantage of simplifying software maintenance, in particular it facilitates movement between different paradigms like task parallelism (VPP Fortran), message passing (MPI) and data parallelism (HPF). This comes at some performance cost due to lower overlap of communication with computation. This conflict between ease of software maintenance and performance is being investigated.

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

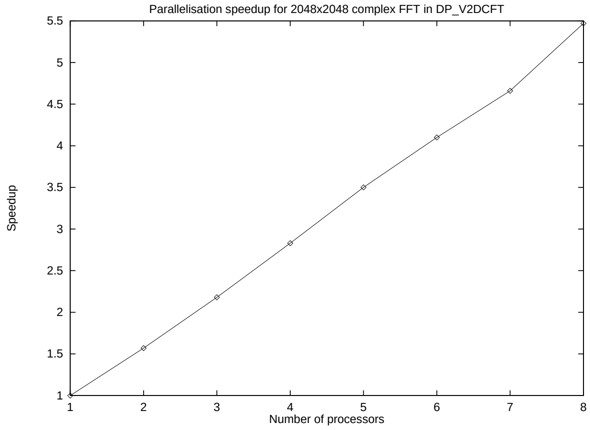


Figure 5: Scalability of FFT on the VPP 300

6 Wavelet Transforms

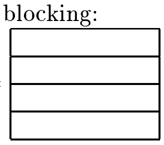
Wavelets yield very good and sparse approximations for most practically occurring functions and data sets and are thus ideally suited for compression [8]. We consider 2D wavelet transforms which can be represented as matrix-matrix products

$$Y = W_M X W_N^T \tag{10}$$

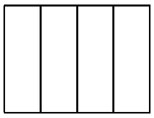
where $X \in \mathbf{R}^{\mathbf{N}\times\mathbf{M}}$ is the data matrix and W_N (and W_M) are the wavelet transform matrices. These are not formed explicitly, instead the fast wavelet transform is used which forms the matrix vector product $W_N x$ as a succession of steps of the form

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

The parallel implementation of (10) depends on the distribution of the data X and the result Y to the processors. We have considered vertical blocking:

$X =$ 

or horizontal blocking:

$X =$ 

where each "slice" is associated with the memory of a different processor. The basic method to compute (10) is, as for FFTs, the split-transform algorithm:

$$\begin{array}{ll} 1) & Z := XW_N^T \\ 2) & Y := W_M Z \end{array}$$

If X is blocked vertically, then the first step is done without communication. The second step requires no communication if Z (and Y) is blocked horizontally. However, a personalised all-to-all communication step in the form of a matrix transpose is required in between. This algorithm is called the "replicated" transform and is in structure identical with that used in the FFT transforms described above.

As for FFTs the great advantage of this method is that all communication is contained in a separate step which makes the software more manageable.

If X is blocked vertically then a small amount of communication is required. Furthermore the vertical blocking of the resulting Z allows for application of the communication-less transform for the computation of Y. Thus the costly personalised all-to-all step is completely avoided. However, the two steps require different software and the first step does require communication which complicates the software engineering.

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 algorithm with both X and Y blocked vertically 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 Pseudo-Random Number Generators

7.1 Uniform Distribution

Several authors [1, 3, 30] have considered the generation of uniformly distributed pseudo-random numbers on vector and parallel computers. The method which we implemented on the Fujitsu VP and VPP series machines is based on the generalized Fibonacci recurrence

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

where w depends on the floating-point fraction length, $r > s > 0$, and $x^r + x^s + 1$ is a primitive polynomial (mod 2). For example, we could choose $w = 52$, $r = 132049$, and $s = 79500$. As shown in [5], the period of such a generator is $2^{w-1}(2^r - 1)$. If the code is written to take advantage of the vector units, each random number can be generated in less than three processor cycles. The generator can easily be implemented on a parallel machine by using a suitable (different) initialization on each processor. For details see [3, 7].

7.2 Normal Distribution

In many applications, random numbers from specified non-uniform distributions are required. A common requirement is for the normal distribution with given mean and variance.

The most efficient methods for generating normally distributed random numbers on sequential machines [23, 24] involve the use of different approximations on different intervals, and/or the use of "rejection" methods, so they do not vectorize well. Other methods are preferable on vector processors. The Box-Muller and Polar methods [23] were considered in [4]. The Polar method was implemented as `RANN3` and was the fastest vectorized method for normally distributed numbers known at the time [29], although much slower than the best uniform random number generators.

Recently Wallace [35] proposed a new class of pseudo-random generators for normal variates. These generators do not require a stream of uniform pseudo-random numbers (except for initialization) or the evaluation of elementary functions such as log, sqrt, sin or cos (needed by the Box-Muller and Polar methods). The crucial observation is that, if x is an n-vector of normally distributed random numbers, and A is an $n \times n$ orthogonal matrix, then $y = Ax$ is another n-vector of normally distributed numbers. Thus, given a pool of nN normally distributed numbers, we can generate another pool of nN normally distributed numbers by performing N matrix-vector multiplications. The inner loops are very suitable for implementation on vector processors. The vector lengths are proportional to N, and the number of arithmetic operations per normally distributed number is proportional to n. Typically we choose n to be small, say $2 \le n \le 4$, and N to be large.

Wallace implemented variants of his new method on a scalar RISC workstation, and found that its speed was comparable to that of a fast uniform generator. The same performance relative to a fast uniform generator is achievable on a vector processor, although some care has to be taken with the implementation. Details of an implementation (`RANN4`) of Wallace's method on the VPP are given in [6]. `RANN4` is several times faster than `RANN3`.

Because Wallace's class of methods is new, there is little knowledge of their statistical properties. However, statistical tests performed on `RANN4` have been satisfactory.

8 Conclusion

Several areas of research in the ANU-Fujitsu Parallel Mathematical Subroutine Library project have been discussed. High performance has been achieved by developing algorithms amenable to vectorization and parallelisation and by devoting a great deal of attention to implementation details.

There exist a considerable number of other routines in the library which have not been addressed here; some of these were discussed in [7]. Also, we note that as part of the development of the parallel library (SSLIIVPP) many single processor routines have been developed to be included in Fujitsu's SSLIIVP, an extensive mathematical subroutine library for vector processors.

References

[1] S. L. Anderson, *Random Number Generators on Vector Supercomputers and other Advanced Architectures*, SIAM Review, **32**, 221–251, 1990.

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

[3] R. P. Brent, *Uniform Random Number Generators for Supercomputers*, Proc. Fifth Australian Supercomputer Conference, Melbourne, 95–104, 1992.

[4] R. P. Brent, *Fast Normal Random Number Generators for Vector Processors*, Area 4 working note #4, 1993.

[5] R. P. Brent, *On the Periods of Generalized Fibonacci Recurrences*, Math. Comp., **63**, 389–401, 1994.

[6] R. P. Brent, *A fast Vectorised Implementation of Wallace's Normal Random Number Generator,* Area 4 Working Note #21, 1997.

[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, 526-533, 1994.

[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.W. Cooley and J.W. Tukey, *An algorithm for the machine calculation of complex Fourier series*, Math. Comp. **19**, 297–301, 1965.

[10] P. Duhamel and H. Hollman, *Split radix FFT algorithms*, Electron. Lett., **20**, 14–16, 1984.

[11] C.Dun, M.Hegland, and M.Osborne, *Stable parallel solution methods for tridiagonal systems of linear equations*, CTAC95 Proceedings, 1996.

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

[13] C. Gauss, *Theoria interpolationis methodo novo tractata*, vol. 3, Königliche Gesellschaft der Wissenschaften, Göttingen, 1866.

[14] I.J. Good, *The interaction algorithm and practical Fourier analysis*, J. Roy. Stat. Soc. Ser. B **22**, 372–375, 1958

[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*, JIMA Num. Anal., To appear.

[20] M. Hegland and W. Wheeler, *Linear bijections and the fast Fourier transform*, Applicable Algebra in Engineering, Communication and Computing **8**, no. 2, 143–163, 1997.

[21] High Performance Fortran, High Performance Fortran Forum, Jan 31, 1997.

[22] G. Keating, *Choosing trees for FFTs*, Parallel Computing Workshop '97 Proceedings, Australian National University, P2–X–1 – P2–X–6, 1997.

[23] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms* (third edition). Addison-Wesley, Menlo Park, 1997.

[24] J. L. Leva, *A Fast Normal Random Number Generator*, ACM Transactions on Mathematical Software, **18**, 449–453, 1992.

[25] 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.

[26] M. Neytcheva, *Experience in implementing the algebraic multilevel iteration method on a SIMD–type computer*, Appl. Numer. Math., **19**,71–90, 1995.

[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. P. Petersen, *Some Vectorized Random Number Generators for Uniform, Normal, and Poisson Distributions for CRAY X-MP*, J. Supercomputing, **1**, 327–335, 1988.

[30] W. P. Petersen, *Lagged Fibonacci Series Random Number Generators for the NEC SX-3*, International J. of High Speed Computing, **6**, 387–398, 1994.

[31] Y. Saad, *Numerical Methods for Large Eigenvalue Problems*, Manchester University Press (Series in Algorithms and Architectures for Advanced Scientific Computing), Manchester, 1992.

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

[33] P. N. Swarztrauber, *Multiprocessor FFTs*, Parallel Comput., **5**, 197–210, 1987.

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

[35] C. S. Wallace, *Fast Pseudo-Random Generators for Normal and Exponential Variates*, ACM Trans. on Mathematical Software, **22**, 119–127, 1996.

[36] S.J.Wright, *A collection of problems for which Gaussian elimination with partial pivoting is unstable*, SISSC, **14**, 231–238,1993.

[37] B.B.Zhou and R.P. Brent, *A Parallel Ring Ordering for Efficient One-sided Jacobi SVD Computations*, Journal of Parallel and Distributed Computing, **42**, pp.1–10, 1997.