High Performance Algorithms for Numerical Linear Algebra

Yusaku Yamamoto

Contents

List of Figures

Acknowldgements

I would like to express my sincere gratitude to Professor Masaaki Sugihara for his support and guidance throughout the development of this thesis. I'm also grateful to Professor Masao Doi, Professor Katsuya Ishii, Professor Yukio Kaneda and Associate Professor Akio Ishiguro for valuable comments and encouragements.

In conducting the research, I have benefited greatly from the discussion with many professors and researchers in my field. Among them, I would like to express my special thanks to Professor Yasumasa Kanada, Associate Professor Reiji Suda and Dr. Takahiro Katagiri.

All of the works presented in this thesis were done while I was at the Central Research Laboratory, Hitachi Ltd. I am especially indebted to my former colleagues Mitsuyoshi Igai, Ken Naono and Hiroyuki Hirayama, without whose collaboration these works would have been impossible. I would like to thank Nobuhiro Ioki, Shin-ichi Tanaka and other members at the Software Development Division of Hitachi Ltd., Hiroki Kawamura at the Hitachi Software Engineering Corp. and Yasunori Ushiro at the Enterprise Server Division of Hitachi Ltd. for many useful discussions and for preparing the environments for computer experiments. I am also grateful to Dr. Mamoru Sugie, Dr. Yasuhiro Inagami, Dr. Chisato Konno, Dr. Sigeo Ihara, Dr. Satoshi Itoh, Hiroaki Fujii, Toshio Okochi and Nobutoshi Sagawa for kind guidance and for providing me with an ideal environment for pursuing this study.

Last, but not least, I wish to thank my parents Naohisa Yamamoto and Mariko Yamamoto for their constant and sincere support during the course of my research.

Chapter 1

Introduction

1.1 Background

Computer simulation has become an increasingly important tool in many areas of science and technology. For example, numerical simulation of atmospheric flows and ocean flows are essential for weather forecast and study on climate change. Electronic structure calculation and molecular dynamics simulation are used to develop new micro devices or to study the structure and functionality of proteins. Car crash simulation by finite element methods is an indispensable tool in the design of vehicles. All of these simulations require solution of mathematical problems with a large degree of freedom. Now, finite element methods or finite difference methods with millions of elements or grid points are commonly used. Electronic structure calculations often employ millions of basis functions and molecular dynamics simulations deal with millions of atoms. Accordingly, there are ever-growing needs for larger computational power and larger memory space.

To satisfy these needs, various types of high performance computers have been proposed and commercially shipped so far. Representative architectures adopted by these computers include vector processors, processors with hierarchical memory, symmetric multi-processors (SMPs) and distributed-memory parallel machines. Recently, combinations of these architectures such as distributed memory parallel machines of which each computational node is a vector processor or SMP have also appeared in pursuit of even higher performance.

However, it is not straightforward to fully exploit the performance of these complex machines. In the case of SMPs, the user must pay enough attention to distribute the work evenly among the processors and at the same time to minimize the number of interprocessor synchronization. To achieve high performance on distributed-memory parallel machines, one has to determine the distribution of data carefully so that both the frequency and volume of interprocessor data transfer is kept to minimum. In addition, techniques to extract the performance of a single processor which constitutes the parallel machine must be employed. All these considerations need a reconstruction of algorithms developed for a sequential computer, or sometimes development of new algorithms.

In many of the simulations listed above, the core of the computation is linear algebra calculations. For instance, both weather simulation based on the Regional Spectral Model and electronic structure calculation using the plane wave basis employ the fast Fourier transform as a key component. Electronic structure calculation also needs solution of eigenvalue problems. In the finite element method, most of the computing time is spent to solve linear simultaneous equations with a sparse coefficient matrix. It is therefore meaningful to develop efficient algorithms for these linear algebra calculations on the various types of machines.

1.2 Our goals

In this thesis, we focus on representative linear algebra computations such as solution of linear simultaneous equations, symmetric eigenvalue problem and the fast Fourier transform and develop algorithms for high performance computers. Our main targets are parallel machines, which include both SMPs and distributed-memory machines, and computers with hierarchical memory.

Our goals are development of efficient, accurate and stable algorithms. By efficiency, we mean that the algorithm requires the same order of computational work as its sequential counterpart and the speedup with P processors approaches P when the problem size is increased. By accuracy and stability, we mean the same level of accuracy and stability as those of the corresponding sequential algorithm. There are many parallel algorithms which do not posses the latter properties, but we believe that these properties are essential for a parallel algorithm to be practical.

For some of the linear algebra computations, algorithms which satisfy the above conditions can be constructed by restructuring the conventional sequential algorithms. This is the case with the direct solution of linear simultaneous equations with dense or sparse positive definite coefficient matrices and the fast Fourier transform. In other occasions, we have to devise a new algorithm. This is the case with the direct solution of linear simultaneous equations with unsymmetric tridiagonal coefficient matrices and the symmetric eigenproblems.

1.3 Outline of the thesis

The following summarizes the contents of this thesis.

1. In Chapter 2, which serves as the preliminary to the entire thesis, we begin by explaining typical architectures of high performance computers. As architectures for a single processor, we take up vector processors and processors with hierarchical memory. As architectures for parallel machines, we discuss symmetric multi-processors and distributed-memory parallel machines. We go into the characteristics of each architecture and discuss optimization techniques suited for each of them. Finally, we introduce the idea of BLAS, or Basic Linear Algebra Subprograms. Roughly speaking, BLAS 1 are a vector-vector operations such as inner product or addition of two vectors, BLAS 2 are matrix-vector operations such as matrix-vector multiplication and BLAS 3 are matrix-matrix operations such as matrix multiplications. We show that higher-level BLAS are desirable both from the viewpoint of single-processor performance and parallel efficiency.

2. In Chapter 3, we discuss direct solution of linear simultaneous equations with dense coefficient matrices. We begin with the basic Gaussian elimination and describe its variants and their properties. Next we introduce two conventional high performance algorithms for Gaussian elimination. One is the blocked Gaussian elimination, which is optimized to maximize data reuse and to achieve high single-processor performance on processors with hierarchical memory. The other is the parallel blocked Gaussian elimination, which was devised to minimize the number and volume of interprocessor communication and attain high parallel efficiency on distributed-memory parallel machines. However, it can be shown that if each node of the distributed-memory machine is a processor with hierarchical memory, it is difficult to choose the block size so that both high single-processor performance and high parallel efficiency are achieved. To solve this problem, we propose the double-blocked Gaussian elimination method and verify its effectiveness through analysis based on an analytical performance model and experiments. The idea of double blocking introduced here is shown to be useful in other fields of numerical linear algebra as well.

3. In Chapters 4 and 5, we deal with direct solution of linear simultaneous equations with sparse coefficient matrices. After introducing some preliminaries at the beginning of chapter 4, we study two problems, namely, parallel direct solution of linear simultaneous equations with sparse symmetric positive definite matrices and parallel direct solution of unsymmetric tridiagonal matrices. For the former problem, we improve the conventional algorithm for distributed-memory parallel machines so that it can also achieve high single-processor performance when the node of the distributed-memory machine is a processor with hierarchical memory. For the latter problem, many parallel algorithms without pivoting have been known. Though it

is essential for numerical accuracy and stability to incorporate pivoting, it has been difficult because pivoting destroys parallelism. We solve this problem by inventing a new reordering method of the tridiagonal matrix that preserves parallelism even when pivoting is introduced.

4. In Chapters 6 and 7, we treat the problem of computing eigenvalues and eigenvectors of a symmetric matrix. The typical algorithm for this consists of four steps, namely, tri-diagonalization of the input matrix by Householder transformations, computation of the eigenvalues of the resulting tri-diagonal matrix by the bisection-type method, computation of the eigenvectors of the tri-diagonal matrix by the inverse iteration method (IIM) and the computation of the eigenvectors of the original matrix by back-transformation. Among these steps, efficient parallel algorithms have been known for the first, the second and fourth steps. However, parallelization of the IIM has been difficult because of the so-called re-orthogonalization process. We propose a new procedure for re-orthogonalization that has a high degree of parallelism and attains the same level of accuracy as the conventional procedure. On the other hand, the tri-diagonalization step had the problem that it was difficult to obtain high single-processor performance on processors with hierarchical memory. This is because half of the operations are done with BLAS2, which is inferior to BLAS 3 in terms of performance. We propose a new algorithm that fully utilizes BLAS 3 and demonstrate that it can achieve much higher performance on this type of machines.

5. In Chapter 8, we discuss parallel algorithms for the 1-dimensional complex fast Fourier transform. We take up distributed-memory machines with vector-processing nodes as our target and aim at constructing algorithms that fully exploit the performance of these machines. To this end, we first describe algorithms based on the so-called 3-dimensional representation of 1-dimensional data. We classify these algorithms and demonstrate that one variant, which inputs and outputs data both using cyclic distribution and require only one all-to-all interprocessor data transfer, is the best one from the viewpoint of single-processor performance, parallel efficiency and usability. Next, we extend this variant to increase the freedom of data distribution. The resulting algorithm can input and output data both using block cyclic data distributions with user-specified block sizes. This obviates the need for data redistribution routines which was necessary with conventional algorithms and enhances the overall performance considerably.

6. Finally in Chapter 9, we give some conclusions and future directions.

Chapter 2

Architectures of High Performance Computers and Optimization Techniques

In this chapter, we explain the architectures of high performance computers which we use as target machines in the rest of this thesis. As architectures for a single processor, we take up vector processors and processors with hierarchical memory. As architectures for parallel machines, we discuss symmetric multi-processors and distributed-memory parallel machines. We go into the characteristics of each architecture and discuss optimization techniques suited for each of them. Finally, we introduce the idea of BLAS, or Basic Linear Algebra Subprograms, and show why the use of higher-level BLAS is desirable both for increasing single-processor performance and enhancing parallel efficiency.

2.1 Architectures of a single processor

2.1.1 Vector processors

A schematic diagram of a vector processor is shown in figure 2.1. It is composed of three main elements, namely, a *vector operation unit*, a *vector register* and *memory banks* [26][27]. The vector register is a large register which can hold dozens or hundreds of words of data. It sends the data to the vector operation unit at a rate of one word per cycle, and the vector operation unit performs the vector operation, that is, the same arithmetic operation on all of the data, at the same rate. The main memory consists of dozens or hundreds of banks and the addresses in the memory space are allocated to the banks cyclically. So, if the memory is accessed contiguously, the banks are used one by one in turn. This enables the data transfer rate from the main memory to the vector register to be increased by a factor of the number of banks, thereby hiding the large cycle time of dynamic RAMs composing the main memory. In the case of several machines, there is no distinct vector register or vector processing unit, but instead there is a software mechanism that makes general registers and general arithmetic processing unit behave like vector processing units. This is called pseudo-vector processor. Examples of machines classified as vector processors are CRAY Y-MP, NEC SX-7 and Fujitsu VPP 5000. Examples of pseudo-vector processors are Hitachi SR2201 and SR8000.

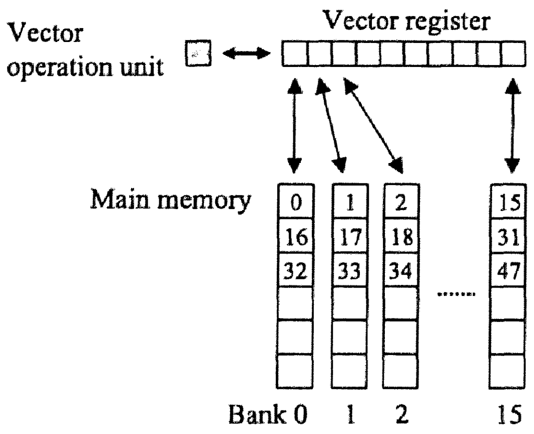


Figure 2.1: Schematic diagram of a vector processor.

When the vector processor starts a vector operation, it has to load the first data from the main memory to the vector register, and next from the vector register to the vector operation unit. This causes some latency or vector startup time. Because the startup time does not depend on the vector length, that is, the number of data on which the vector operation is performed, its ratio to the execution time of the vector operation decreases as the vector length increases. This leads to a guideline that one should make the vector length as long as possible to get high performance on vector processors. Because vectorization is usually done with respect to the innermost loop, this is equivalent to maximizing the length of the innermost loop. We show in Table. 2.1 how the performance of matrix multiplication on a single processor of the SR2201 varies with the vector length.

Table 2.1: Effect of vector length on the performance of matrix multiplication

Vector length	20	40	60	80	100	120	250
Performance (MFLOPS)	167	205	224	233	241	246	250

Another consideration is to use the memory banks efficiently. As we mentioned earlier, memory banks are designed so that they achieve the maximum performance when the memory is accessed contiguously. However, in real applications, it is frequently necessary to access a multi-dimensional array in a direction for which memory access is not contiguous. More specifically, consider a two dimensional array $A(NX, NY)$ (in FORTRAN) and suppose that the second index of the array is incremented one by one in the innermost loop. In this case, every NX-th elements in the memory is accessed in the innermost loop, so if NX is a multiple of the number of memory banks, or if these two have a large common divisor, only part of the banks are accessed. This situation is known as *bank conflict* and causes severe performance degradation. To prevent bank conflict, it is effective to change the leading dimension NX to $NX + \alpha$ so that $NX + \alpha$ and the number of banks do not have a large common divisor.

Finally, one should pay attention to the ratio of the number of load or store operation to the number of arithmetic operation. Many vector processors are designed so that the vector operation unit attains the maximum performance when this ratio is 1:1. As an example, we consider an inner product of two vectors, $c = \mathbf{x} \cdot \mathbf{y}$. In computing this, two words of data, x_i and y_i, are loaded from the main memory, their multiplication is computed and the result is added to the partial sum. So there are two loads and two arithmetic operations for each i and the vector operation unit can run at a full speed. In contrast, if one wants to compute the addition of two vectors $\mathbf{z} = \mathbf{x} + \mathbf{y}$, one needs to load two words of data, x_i and y_i, add them and store the result. In this case, the ratio of load/store operations and arithmetic operations is 3:1, and therefore the vector operation unit can attain at most one third of its peak performance due to the limitation of memory throughput. We will introduce techniques to overcome this situation in section 2.3.

2.1.2 Processors with hierarchical memory

Another representative architecture for a single processor is a processor with hierarchical memory [26][27]. The schematic diagram of this type of machine is illustrated in Fig. 2.2. There are at least three level of memory devices, namely, the *registers*, the *cache*, and the *main memory*. The cache is directly connected to the operation unit. It can contain only several to dozens of words, but is the fastest both in terms of latency and throughput. The data stored in it can be accessed with the latency of a few cycles and at a rate of two or more words per cycle. On the other hand, the main memory has the largest capacity, but is the slowest. The latency is several dozens to one hundred cycles, and the throughput is well below one word per cycle. The cache is situated between the register and both its capacity and speed is between those of the main memory and the registers. It retains the data used by the operation unit so that the second and the following access to the same data can be done at a much higher speed.

Most of the modern computers are equipped with cache, so the considerations we state

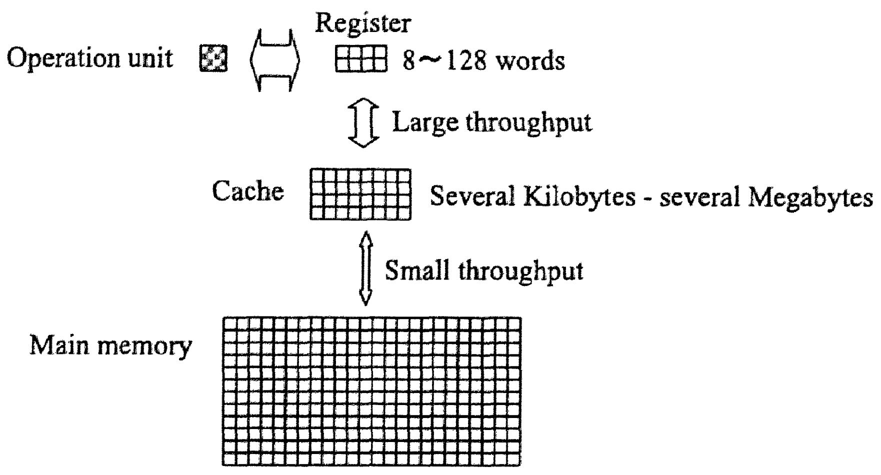


Figure 2.2: Schematic diagram of a processor with hierarchical memory.

here apply to them. Representative machines include Intel Pentium III, IBM Power 4, AMD Athlon and DEC Alpha. Some processors have more than one levels of cache, that is, the first cache integrated in the processor chip and a larger second cache. But the principles of optimization techniques for single level cache apply to them as well. Note also that processors with hierarchical memory and vector processors are not mutually exclusive concepts. Actually, there are machines such as the Hitachi SR8000 that have both pseudo-vector processing facility and the cache.

To attain high performance on processor with hierarchical memory, it is essential to increase the locality of data reference and use the data as many times as possible while it is in the cache. This reduces accesses to the slow main memory and enables the operation unit to run at a faster speed. Representative technique for this are *blocking* and use of higher-level BLAS, which we will explain in the following sections. In addition, even if the data is guaranteed to be in the cache, we still need to pay attention to the ratio of load/store operations to the arithmetic operations, because of the limitation of throughput between cache and the operation unit. To reduce this ratio, the same technique as in the case of vector processors can be employed.

2.2 Architectures of parallel machines

2.2.1 Symmetric multi-processors

The architectures of parallel machines can be divided into two types, namely, the *shared-memory machines* and the *distributed-memory machines* [26][27][80][102]. A simplified diagram of a shared-memory machine is shown in Fig. 2.3. There is a main memory system and a number of processors, and each of the processor is connected to the memory through a bus. On some machines, there are more than one memory system and the processors and the memory systems are connected via a crossbar switch instead of the bus. This type of machine is also called *symmetric multi-processors* or SMPs, because any portion of the memory can be accessed by any processor equally easily and there is no correspondence between a processor and a portion of the memory. There are many machines belonging to this category, such as NEC SX-7, one node of Hitachi SR8000 and IBM p-Series. Note that SX-7 is an SMP of which each processor is a vector processor. Similarly, one node of SR8000 is an SMP of which each processor is a pseudo-vector processor. Recently, personal computers with dual CPUs have become popular. These are also a kind of SMP machines.

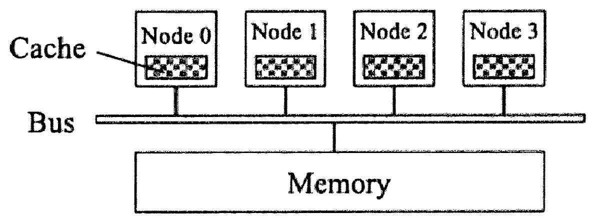


Figure 2.3: Schematic diagram of a shared-memory parallel machine.

One of the greatest advantages of the SMPs is that there is only one global memory space. This is the same characteristic as the sequential computer and makes parallel programming considerably easy. On the other hand, there is a disadvantage that if the number of processors is too large, say more than 20, bus contention in memory access occurs frequently and it becomes very difficult to increase the performance in proportion to the number of processors.

An important concept accompanying the SMP is *interprocessor synchronization.* This means that if a processor wants to use the result of another processor, it has to wait and make sure that the computation on the latter has completed. As an example, we show in Fig. 2.4 the behavior of processors when computing the inner product of two vectors using four processors.

There are several considerations to extract the potential performance of SMPs [80][102]. First, one has to make sure that computational loads are allocated to the processors evenly. If the workload balance is uneven, or considerable part of the program has to be executed sequentially, processor will become idle and parallel efficiency will be low. In fact, this is true of any type of parallel machines.

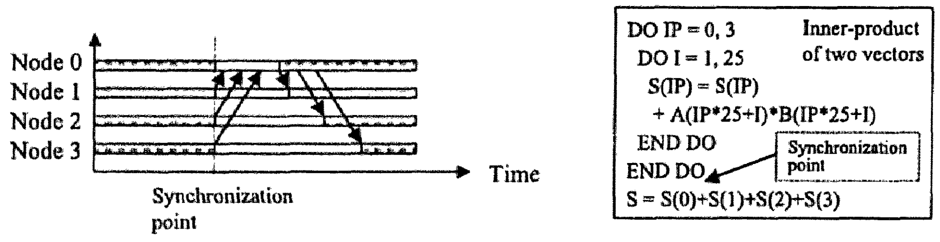
Second, one has to make effort to minimize the number of interprocessor synchronization. On many SMPs, interprocessor synchronization requires hundreds or even thousands of cycles, so excessive use of synchronization will easily spoil the effect of parallelization. In linear algebra algorithms, one of the best ways to enhance load balance and reduce the number of synchronizations is to use higher level of BLAS such as BLAS 2 and BLAS 3, as we will see in section 2.4.

Finally, one should reduce the bus contention. Many SMP machines have cache memories associated with each processor, so it is desirable to use these cache memories efficiently and reduce the access to the main memory. For this purpose, one can use the techniques used for processors with hierarchical memory.

2.2.2 Distributed-memory parallel machines

The other type of parallel machines is a distributed-memory parallel machine, which has a separate memory system for each processor. We show a schematic diagram of this type of machine in Fig. 2.5. Each computational node consists of a processor, memory system and (possibly) cache and these nodes are connected via interprocessor network. Machines classified into this category include CRAY T3E, IBM SP3, Fujitsu VPP5000, NEC SX-7 and Hitachi SR2201 and SR8000. Note that SX-7 and SR8000 have a complex architecture such that each node is again a parallel machine of SMP type. PC clusters, which have become increasingly popular recently, also belong to this category.

This architecture has a marked advantage that the number of nodes can be easily increased. Actually, *massively parallel machines* with thousands of nodes have been shipped commercially and are widely used. However, from the programmer's point of view, these machines are very different from sequential computers because the memory space is divided into as many subspaces as the number of nodes. So the programmer must always be aware to which subspace each data belongs to. If data having been processed by one node is to be processed by another node, it has to be first transferred to the latter node



```
DO IP = 0, 3
 DO I = 1, 25
  S(IP) = S(IP)
   + A(IP*25+I)*B(IP*25+I)
 END DO
END DO
S = S(0)+S(1)+S(2)+S(3)
```

Figure 2.4: Interprocessor synchronization in computing inner-product of two vectors.

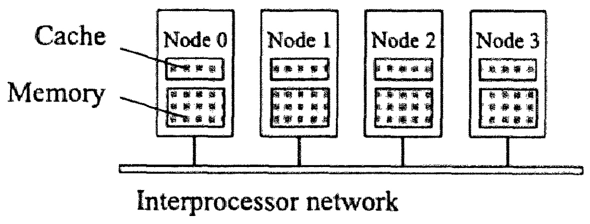


Figure 2.5: Schematic diagram of a distributed-memory parallel machine.

via the network. This often makes the computer program much lengthier than that for sequential machines.

To attain high performance on distributed machines, it is important as in the case of SMPs to make the load balance even among the nodes. In addition, it is critical to keep the frequency and the volume of interprocessor data transfer as small as possible [80][102]. On many machines, setup time of hundreds or even thousands of machine cycles is necessary to start one interprocessor data transfer. In addition, sending one word of data takes more than ten times as long as performing one arithmetic operation. Accordingly, unless one exercises enough care to minimize these costs, it is very likely that the speedup obtained by increasing the number of nodes is cancelled by the overhead of data transfer.

To reduce the volume of data transfer, it is effective to optimize the distribution of data to the nodes, or to devise an algorithm which inherently involves small volume of data transfer. These are part of the subjects of Chapters 3, 4 and 8 and will be treated there in detail. Another approach, which can be used together with the former approaches, is to hide the overhead of data transfer by overlapping it with arithmetic operations. This is discussed, for example, in [32]. Decreasing the number of interprocessor data transfer can be achieved by collecting together several transfers. We can see an example of this in subsection 3.2.2.

2.3 Optimization techniques

In this section, we summarize basic techniques to improve the performance of a program on a single processor [26][27]. Techniques for SMPs and distributed-memory parallel machines are more problem-dependent and will be treated in the following chapters.

2.3.1 Loop exchange

On vector processors, one of the key factors that determine the performance is the innermost loop length. In some cases, it is possible to increase this length by exchanging the innermost loop with another loop outside it. For example, we show here Stockham's algorithm for computing 1-dimensional fast Fourier transform (FFT) of $N = 2^p$ points [97].

```
[Algorithm 2.1 Stockham's algorithm]
do $L = 0, p-1$
  $\alpha_L = 2^L$
  $\beta_L = 2^{p-L-1}$
  do $k = 0,\ \alpha_L - 1$
    do $j = 0,\ \beta_L - 1$
      $X_{L+1}(j,k) = X_L(j,k) + X_L(j+\beta_L,k)\,\omega^{k\beta_L}$
      $X_{L+1}(j,k+\alpha_L) = X_L(j,k) - X_L(j+\beta_L,k)\,\omega^{k\beta_L}$
    end do
  end do
end do
```

Here, $X_L(j,k)$ is a two-dimensional array of size $2\beta_L \times \alpha_L$ and $\omega = \exp(-2\pi i/N)$. The algorithm consists of a three-fold loop and the length of the innermost loop changes as $2^{p-1}, 2^{p-2}, \ldots, 1$ as L increases.

If we notice that the computation of $X_{L+1}(j,k))$ are independent with respect to j and k and that the length of the second innermost loop increases as L increases, we know we can exchange the innermost and the second innermost loops at the value of L where $\alpha_L > \beta_L$ holds. This will keep the length of the innermost loop always over $O(\sqrt{N})$.

Loop exchange is also useful for reducing the ratio of load/store to arithmetic operations. For instance, Algorithm 2.2 and 2.3 below both compute the matrix product $\mathbf{C} := \mathbf{C} + \mathbf{AB}$ and only the order of loop nesting is different. While Algorithm 2.2 (outer-product form) requires two load operations ($C(i,j)$ and $A(i,k)$) and one store operation ($C(i,j)$) in the innermost loop, Algorithm 2.3 (inner-product form) require only two load operations. So from the viewpoint of minimizing load/store, we can say that Algorithm 2.2 is superior.

```
[Algorithm 2.2 Matrix multiplication (outer-product form)]
do $k = 1, L$
  do $i = 1, M$
    do $j = 1, N$
      $C(i,j) := C(i,j) + A(i,k) * B(k,j)$
    end do
  end do
end do
```

```
[Algorithm 2.3 Matrix multiplication (inner-product form)]
do $i = 1, M$
  do $j = 1, N$
    do $k = 1, L$
      $C(i,j) := C(i,j) + A(i,k) * B(k,j)$
    end do
  end do
end do
```

2.3.2 Loop merging

Suppose that there are two distinct loops and the same array variable A is accessed in both of the loops. In that case, we have a possibility to be able to reduce the load/store of A by merging these two loops.

As an example, let's consider matrix-vector multiplication $\mathbf{y} = \mathbf{Ax}$, where $\mathbf{A}$ is a N by N symmetric matrix. By using the symmetry of $\mathbf{A}$, the program can be written as follows.

```
[Algorithm 2.4 Matrix-vector multiplication (I)]
do $i = 1, N$
  $y(i) = A(i,i) * x(i)$
end do
do $i = 1, N$
  do $j = 1, i - 1$
    $y(i) := y(i) + A(i,j) * x(j)$
  end do
end do
do $j = 1, N$
  do $i = 1, j - 1$
    $y(i) := y(i) + A(j,i) * x(j)$
  end do
end do
```

By noting that the second loop accesses the same element of $A(i,j)$ if the notation of i and j is interchanged, we can merge these two loops and get the following code. This code reduces the load of A by hald and therefore should run faster than the original code.

[Algorithm 2.5 Matrix-vector multiplication (II)]

```
do $i = 1, N$
  $y(i) = A(i,i) * x(i)$
end do
do $i = 1, N$
  do $j = 1, i-1$
    $y(i) := y(i) + A(i,j) * x(j)$
    $y(j) := y(j) + A(i,j) * x(i)$
  end do
end do
```

It is also possible to collapse a double loop into a single loop. This is sometimes useful for increasing the length of the innermost loop. See [97] for an example in the case of FFT.

2.3.3 Loop unrolling

The technique most commonly used to decrease the ratio of load/store is *loop unrolling*. As an example, we show a code obtained by unrolling the loops of i and j in Algorithm 2.3.

[Algorithm 2.6 Matrix multiplication (inner-product form, (2,2) unrolled)]

```
do $i = 1, M, 2$
  do $j = 1, N, 2$
    do $k = 1, L$
      $C(i,j)         := C(i,j) + A(i,k) * B(k,j)$
      $C(i,j+1)       := C(i,j+1) + A(i,k) * B(k,j+1)$
      $C(i+1,j)       := C(i+1,j) + A(i+1,k) * B(k,j)$
      $C(i+1,j+1)     := C(i+1,j+1) + A(i+1,k) * B(k,j+1)$
    end do
  end do
end do
```

This is called (2,2) unrolling because the increments of both i and j loops are increased to 2 and the equations in the loop are copied 4 ($= 2 \times 2$) times. As a result, the number of arithmetic operations for each iteration of the innermost loop is increased to 8. However, the number of load operation is increased only twice, because each of $A(i,k)$, $A(i+1,k)$,

$B(k,j)$ and $B(k,j+1)$ can be used twice. Hence, the ratio of load/store to arithmetic operation is reduced by half and this code is expected to run at a faster speed than the original one. Note however that in the above reasoning, it is assumed that all the C's can be stored in the register. In general, the larger the size of unrolling, the higher the performance, on condition that variables do not spill out of the register. Note also that unrolling of the innermost loop is sometimes effective because it decreases the number of iterations of the innermost loop and thereby reduces the overheads. The performance of matrix multiplication on a single processor of the SR2201 as a function of unrolling size is shown in Table. 2.2. This clearly demonstrates the great impact of unrolling on the performance.

Table 2.2: Effect of loop unrolling on the performance of matrix multiplication

Unrolling of (i,j,k)	(1,1,1)	(1,1,5)	(2,2,1)	(2,2,2)	(5,2,2)
Performance (MFLOPS)	67	110	180	210	250

2.3.4 Blocking

Blocking means to partition the data into small size of blocks that can be fit into the cache, and change the order of arithmetic operations so that as many operations as possible can be performed on each block while it is in the cache. This increases the locality of data reference and makes it possible to use the cache efficiently.

In numerical linear algebra computations, this corresponds to partition the matrix into (usually square) submatrices of a size fitting into the cache [37][46]. Then, in many cases, it is possible to reformulate the algorithm so that the operations are done not on the individual matrix elements but on the submatrices. For example, in the matrix multiplication $\mathbf{C} = \mathbf{AB}$ shown in Algorithm 2.3, if we partition the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ into submatrices of size $P \times P$ and denote the submatrices by $\mathbf{A}_{IK}$, $\mathbf{B}_{KJ}$ and $\mathbf{C}_{IJ}$, respectively, we can derive the following blocked version of the algorithm.

```
[Algorithm 2.7 Matrix multiplication (blocked form)]
do $I = 1, M/P$
  do $J = 1, N/P$
    do $K = 1, L/P$
      $\mathbf{C}_{IJ} := \mathbf{C}_{IJ} + \mathbf{A}_{IK}\mathbf{B}_{KJ}$
    end do
  end do
end do
```

In this algorithm, the scalar multiplications and additions are replaced by matrix multiplications and additions of $P \times P$ matrices. Because a matrix multiplication performs $O(P^3)$ operations on $O(P^2)$ data, each data is used $O(P)$ times, and the locality of data reference is greatly enhanced.

Similar algorithms can be derived for Gaussian elimination to solve linear simultaneous equations and tri-diagonalization for eigenvalue/eigenvector computation as well and will be described in Chapters 3 and 6, respectively.

2.4 The basic linear algebra subprograms

By combining the above-mentioned techniques with optimized algorithms, one can maximize the performance of linear algebra programs on a single processor. However, it is too laborious to apply these techniques to each of the loops that constitute a linear algebra algorithm. Fortunately, most of the linear algebra algorithms can be decomposed into basic linear algebra operations such as inner product of two vectors or matrix-vector multiplication. So if optimized subroutines which performs these basic operations are provided for each machine, they can be used in common for many types of linear algebra computations.

Based on this idea, the BLAS, or Basic Linear Algebra Subprograms have been proposed [34][35][46] and are widely used. There are three levels of BLAS as follows:

1. Level 1 BLAS

   These are operations on one or two vectors and include inner product $c = \mathbf{x} \cdot \mathbf{y}$, AXPY operation $\mathbf{y} = \alpha\mathbf{x} + \mathbf{y}$ and L_2 norm $c = \| \mathbf{x} \|_2$. Typically, these BLAS 1 routines perform $O(N)$ operations on $O(N)$ data when the length of the input vector is N.

2. Level 2 BLAS

   These are operations on a matrix and one or two vectors and include matrix-vector multiplication $\mathbf{y} = \mathbf{A}\mathbf{x}$ and rank-1 modification of a matrix $\mathbf{A} = \mathbf{A} + \mathbf{x}\mathbf{y}^t$. Here, the t on the right shoulder denotes transpose of a matrix or a vector. Solution of linear simultaneous equation with an upper (or lower) triangular coefficient matrix $\mathbf{x} = \mathbf{U}^{-1}\mathbf{b}$ is also regarded as a member of BLAS2. When the size of matrix is $N \times N$, BLAS 2 routines perform $O(N^2)$ operations on $O(N^2)$ data. From the viewpoint of data reuse, BLAS2 is superior to BLAS1 because the input vectors are used many times. But the reduction of the number of load/store is at most by a factor of two, because each element of the input matrix is accessed only once. Regarding parallelization on the SMP machines, BLAS 2 are much preferable to

BLAS 1, because both matrix-vector multiplication and rank-1 modification of a matrix needs only one synchronization for $O(N^2)$ arithmetic operation. In contrast, BLAS 1 routines need one synchronization for $O(N)$ operation and the overhead due to it is far bigger.

3. Level 3 BLAS

   These are operations on one, two or three matrices and include matrix multiplication $\mathbf{C} = \mathbf{AB}$, LU decomposition $\mathbf{A} = \mathbf{LU}$ and solution of linear simultaneous with an upper (or lower) triangular coefficient matrix and multiple right-hand-sides $\mathbf{X} = \mathbf{U}^{-1}\mathbf{B}$. When the size of matrix is $N \times N$, BLAS 3 routines perform $O(N^3)$ operations on $O(N^2)$ data. So, each data can be reused $O(N)$ times in average. This is much better than the data reuse of BLAS1 and BLAS2. With regard to parallelization, matrix multiplication has an ideal property, because involves only one synchronization for $O(N^3)$ operations.

From what we have stated above, we know that higher-level BLAS, especially BLAS 3, have much more desirable properties than lower-level BLAS, both in terms of data reuse and parallelization. Consequently, if a linear algebra algorithm constructed from lower-level BLAS can be rewritten by higher-level BLAS by blocking or other means, it will have a much greater chance of achieving both high single-processor performance and parallel efficiency. We will show examples of this in Chapters 3, 6 and 7.

The BLAS routines can be made from scratch according to the functional and interface specification given in [34][35]. But many vendors provide a complete set of optimized BLAS for their machines. In addition, a variant of BLAS called ATLAS (Automatically Tuned Linear Algebra Software) [11], which automatically optimizes itself for a given machine, has been developed and distributed freely. ATLAS is reported to give better performance than vendor-supplied BLAS in some cases, and is therefore maybe the routine of choice when the latter is not fast enough.

Chapter 3

Direct Solution of Linear Simultaneous Equations with Dense Coefficient Matrices

3.1 Introduction

In this chapter, we deal with the problem of solving linear simultaneous equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.1}$$

where $\mathbf{A}$ is a dense unsymmetric matrix of order N and $\mathbf{x}$ and $\mathbf{b}$ are vectors of length N. This problem arises, for example, in the solution of partial differential equations with the boundary element method and in the scattering problem of electromagnetic waves. In addition, this is the most basic problem in numerical linear algebra and techniques developed for this problem can be applied to many other problems.

The most common procedure for solving eq. (3.1) is the *Gaussian elimination* or *LU-factorization* method [46][89][101]. In this method, one first factorizes $\mathbf{A}$ as $\mathbf{A} = \mathbf{L}\mathbf{U}$, where $\mathbf{L}$ and $\mathbf{U}$ are a lower triangular matrix with unit diagonal and an upper triangular matrix, respectively. One then solves two equations $\mathbf{L}\mathbf{y} = \mathbf{b}$ and $\mathbf{U}\mathbf{x} = \mathbf{y}$ in this order and finally obtain the solution $\mathbf{x}$. Because LU factorization involves $O(N^3)$ computations and solution of two triangular equations require $O(N^2)$ computations, we treat only the LU factorization part in this chapter.

3.1.1 The basic algorithm for LU factorization

The basic algorithm of LU factorization is shown below [46]. Here, $a_{ij}^{(k)}$ denotes the (i, j)-th element of $\mathbf{A}$ at the k-th stage.

```
[Algorithm 3.1 LU factorization ($kij$-form)]
do $k = 1, N$
  do $i = k + 1, N$
    $a_{ik}^{(k+1)} = a_{ik}^{(k)} / a_{kk}^{(k)}$
    do $j = k + 1, N$
      $a_{ij}^{(k+1)} = a_{ij}^{(k)} - a_{ik}^{(k+1)} * a_{kj}^{(k)}$
    end do
  end do
end do
```

This algorithm requires about $\frac{2}{3}N^3$ arithmetic operations. When the computation is finished, the upper triangular part of matrix $\mathbf{A}$ constitutes $\mathbf{U}$. The strictly lower triangular part of $\mathbf{A}$ constitutes the strictly lower triangular part of $\mathbf{L}$. The element $a_{kk}^{(k)}$, the column $\{a_{ik}^{(k+1)}\}_{i=k+1}^{N}$ and the row $\{a_{kj}^{(k)}\}_{j=k+1}^{N}$ are called the *pivot element*, *pivot column* and *pivot row* at the k-th stage, respectively. This algorithm is known as the *kij-form* (or *outer-product form*) of LU factorization because the triple loops are nested in this order.

The kij-form has the advantage that it has $O((N-k)^2)$ parallelism for each stage k. It is therefore used as a basis of parallel algorithms which we will introduce in the subsequent sections. However, it has the disadvantage that the innermost two loops have the form of rank-2 modification (outer product of two vectors) and need two loads and one store for each iteration.

3.1.2 Variants of the LU factorization algorithm

The triple loops in Algorithm 3.1 can be nested in any order, so there are six possibilities in total. For example, the jik-form can be written as follows.

```
[Algorithm 3.2 LU factorization ($jik$-form)]
do $j = 1, N$
  do $i = 2, N$
    do $k = 1, \min(i-1, j-1)$
      $a_{ij} := a_{ij} - a_{ik} * a_{kj}$
    end do
  end do
  do $i = j + 1, N$
    $a_{ij} := a_{ij} / a_{jj}$
  end do
end do
```

This form does not possess as much parallelism as the kij-form, because the innermost loop about k has a form of inner product and each iteration cannot be executed completely independently. However, because of the structure of inner product, it requires only two loads for each iteration. Hence this form is preferable when the ratio of load/store to arithmetic operation has a great impact on the performance. Other forms of LU factorization also have their own advantages and disadvantages and may be the algorithm of choice for certain type of machines. See [37][46] for details.

There is another variant of LU factorization called *Cholesky factorization*. It can be used when the matrix $\mathbf{A}$ is symmetric positive definite and factorizes $\mathbf{A}$ as $\mathbf{A} = \mathbf{L}\mathbf{L}^t$, where $\mathbf{L}$ is a lower triangular matrix. It has the advantage that it requires only half the computational work and memory compared with the LU factorization. The Cholesky factorization will be treated in more detail in the next chapter.

3.1.3 Pivoting for accuracy and numerical stability

During the course of LU factorization defined by Algorithm 3.1, it may occur that the pivot element $a_{kk}^{(k)}$ becomes zero or very small in modulus. In that case, we cannot continue the calculation any more, or even if we can, the subsequent matrix elements might lose accuracy. To prevent this, one usually incorporates *partial pivoting* [46][56][99] into the algorithm. This means that before computing $a_{ik}^{(k+1)} = a_{ik}^{(k)}/a_{kk}^{(k)}$, one chooses the element with the maximum modulus, say $a_{k'k}^{(k)}$, from $\{a_{ik}^{(k)}\}_{i=k}^{N}$ and exchange the k'-th row of the matrix with the k-th row. It can be shown that in exact arithmetic, $a_{k'k}^{(k)} = 0$ if and only if the matrix $\mathbf{A}$ is singular. Hence this procedure guarantees that the algorithm does not break down if $\mathbf{A}$ is nonsingular. Moreover, it ensures that all the elements of the pivot column have modulus smaller than 1. This is essential in retaining numerical accuracy in the solution of the linear simultaneous equations. See [46][56][99] for details.

Although partial pivoting is important in computing the LU decomposition of a general unsymmetric matrix $\mathbf{A}$, we will not mention it explicitly in the description of high performance algorithms to be treated. This is because it is straightforward to incorporate partial pivoting in the case of dense algorithms. In the case of sparse algorithms, in contrast, incorporation of partial pivoting needs development of new techniques. This will be the subject of Chapter 5.

3.2 Conventional high-performance algorithms

3.2.1 Blocked Gaussian elimination method

In this section, we introduce conventional high performance algorithms for LU factorization, the *blocked Gaussian elimination* [37][46] and its parallelization on distributed-memory parallel machines.

The blocked Gaussian elimination is designed for processors with hierarchical memory. In this method, the input matrix $\mathbf{A}$ is partitioned into submatrices of size $L \times L$. Here, L is determined so that three such submatrices can be stored in the cache. Then the algorithm of LU factorization defined in Algorithm 3.1 is applied to these submatrices, instead of individual matrix elements. The resulting algorithm can be written as follows. Here, $\mathbf{A}_{IJ}^{(K)}$ denotes the (I, J)-th submatrix of $\mathbf{A}$ at the K-th stage and we assumed for simplicity that N is divisible by L.

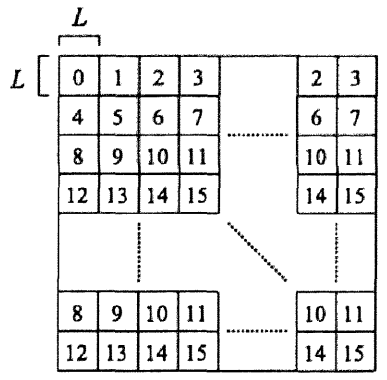
```
[Algorithm 3.3 Blocked Gaussian elimination]
do $K = 1, N/L$
  $\mathbf{A}_{KK}^{(K)} = \mathbf{L}_K\mathbf{U}_K$     (LU factorization of $\mathbf{A}_{KK}^{(K)}$)
  do $I = K + 1, N/L$
    $\mathbf{A}_{IK}^{(K+1)} = \mathbf{A}_{IK}^{(K)}\mathbf{U}_K^{-1}$
  end do
  do $J = K + 1, N/L$
    $\mathbf{A}_{KJ}^{(K+1)} = \mathbf{L}^{-1}\mathbf{A}_{KJ}^{(K)}$
  end do
  do $I = K + 1, N/L$
    do $J = K + 1, N/L$
      $\mathbf{A}_{IJ}^{(K+1)} = \mathbf{A}_{IJ}^{(K)} - \mathbf{A}_{IK}^{(K+1)}\mathbf{A}_{KJ}^{(K+1)}$
    end do
  end do
end do
```

The submatrix $\mathbf{A}_{KK}^{(K)}$, the column of submatrices $\{\mathbf{A}_{IK}^{(K+1)}\}_{I=K+1}^{N/L}$ and the row of submatrices $\{\mathbf{A}_{KJ}^{(K+1)}\}_{J=K+1}^{N/L}$ are called the *block pivot element*, *block pivot column* and *block pivot row* at the K-th stage, respectively. Note that in this algorithm, division by $a_{kk}^{(k)}$ is replaced with LU factorization of $\mathbf{A}_{KK}^{(K)}$ and multiplication by $\mathbf{L}_K^{-1}$ and $\mathbf{U}_K^{-1}$. We can prove that this algorithm actually computes the LU factorization of $\mathbf{A}$ by noting that the matrix at each stage K the leading K by K blocks of $\mathbf{A}$ form an upper triangular matrices and that the matrices used at each stage to transform $\mathbf{A}$ is a lower triangular matrix. See [37][46] for details.

This algorithm is optimal for machines with hierarchical memory because it is composed entirely of BLAS 3 operations such as the LU factorization of a diagonal submatrix and matrix multiplications. From the definition of the block size L, all of these operation can be done within the cache and access to the main memory can be reduced by a factor of $O(L)$.

3.2.2 Parallel blocked Gaussian elimination method

The blocked Gaussian elimination method can be naturally extended to execute on distributed-memory parallel machines [37]. Suppose that there are P processing nodes and P can be factorized as $P = qr$. In the parallel blocked Gaussian elimination method, the blocks are allocated to nodes periodically in both directions (row and column). More specifically, the (I, J)-th block is allocated to node MOD$(I-1, q)$*r + MOD$(J-1, r)$. The allocation in the case of $q = r = 4$ is shown in Fig. 3.1.



The number in the square denotes the node number of the processing node which takes charge of the block. The figure shows the case where there are 16 nodes.

Figure 3.1: Allocation of nodes in the blocked Gaussian elimination method.

Before going into the algorithm, we need some definitions. Let d_K denote the node that has the block pivot element $\mathrm{A}_{KK}^{(K)}$ at the K-th stage. Let R_K and C_K denote the group of nodes that have the block pivot row and block pivot column at the K-th stage, respectively. Then the algorithm can be stated as in Algorithm 3.4.

Because this algorithm is based on the blocked Gaussian elimination, each interprocessor data transfer appearing in Algorithm 3.4 occurs only once for each value of K. On the other hand, if parallelization is based on the basic algorithm shown in Algorithm 3.1, data transfer will occur once for each value of k. This means that blocking reduces the number of data transfers by a factor of L. This advantage, combined with the ability to use cache memory efficiently, makes the parallel blocked Gaussian elimination an ideal method for large scale problems such as the LINPACK benchmark [66] both from the viewpoint of single-processor performance and parallel efficiency.

```
[Algorithm 3.4 Parallel blocked Gaussian elimination]
do $K = 1, N/L$
  Node $d_K$ performs the LU factorization $\mathbf{A}_{KK}^{(K)} = \mathbf{L}_K\mathbf{U}_K$.
  Node $d_K$ broadcasts $\mathbf{L}_K$ in the row direction (to $R_K$).
  Node $d_K$ broadcasts $\mathbf{U}_K$ in the column direction (to $C_K$).
  Node group $C_K$ computes $\mathbf{A}_{IK}^{(K+1)} = \mathbf{A}_{IK}^{(K)}\mathbf{U}_K^{-1}$
      for $K+1 \le I \le N/L$.
  Node group $C_K$ broadcasts $\mathbf{A}_{IK}^{(K+1)}$ in the row direction.
  Node group $R_K$ computes $\mathbf{A}_{KJ}^{(K+1)} = \mathbf{L}^{-1}\mathbf{A}_{KJ}^{(K)}$
      for $K+1 \le J \le N/L$.
  Node group $R_K$ broadcasts $\mathbf{A}_{KJ}^{(K+1)}$ in the column direction.
  Each node computes $\mathbf{A}_{IJ}^{(K+1)} = \mathbf{A}_{IJ}^{(K)} - \mathbf{A}_{IK}^{(K+1)}\mathbf{A}_{KJ}^{(K+1)}$
      for the blocks $\mathbf{A}_{IJ}^{(K)}$ allocated to it.
end do
```

There are additional techniques to further enhance the performance of this method by, for example, overlapping the data transfer with computation and hiding the overhead of the former. See [32] for details.

3.3 The double blocked Gaussian elimination method

3.3.1 Difficulties with the conventional algorithm

In the preceding section, we concluded that the parallel blocked Gaussian elimination is effective for solving large problems on distributed-memory machines with hierarchical memory. In the reasoning, we implicitly assumed that the blocks are distributed to the nodes evenly, because the problem is large and there are a plenty number of blocks.

When the matrix is small or has inherently small parallelism, however, the above reasoning fails and the parallel blocked Gaussian elimination method cannot perform well on these machines. This is because smaller matrix size or smaller parallelism demands smaller block size for achieving good load valance, and as a result, the block size L optimal for single-processor performance differs from that optimal for high parallel efficiency.

To be concrete, let's consider a situation where we want to factorize a dense matrix of order 1000 on a distributed memory parallel machine with 64 nodes and suppose that each node has 256K Bytes of cache memory. To attain maximum single-processor performance, L should be as large as possible under the condition that three $L \times L$ blocks can be stored in the cache memory. So L should be around 80 from this criterion. To attain maximum concurrent efficiency, on the other hand, one must choose L by considering two factors, namely, the frequency of interprocessor communications and the load balance among the nodes. Increasing L will reduce the frequency of communication but will make it more difficult to balance the load because it will decrease the number of blocks to be distributed among the nodes. Decreasing L will give rise to an opposite effect. Usually, for the case under consideration, the value of L that gives the highest parallel efficiency is less than 10. So in this case, the value of L optimal for single-processor performance is very different from that optimal for parallel efficiency (See Fig. 3.2) and one cannot exploit the potentially high performance of the parallel machine no matter what value of L one may choose.

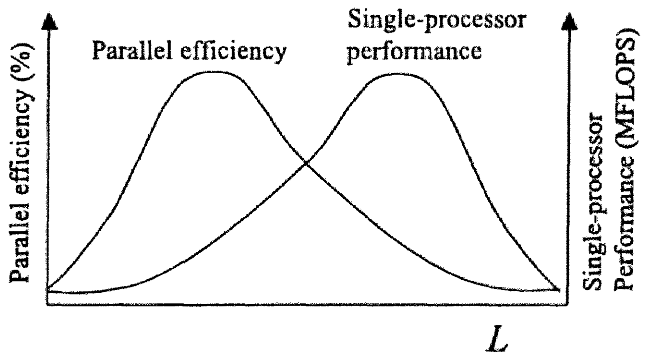


Figure 3.2: Single-processor performance and parallel efficiency as functions of the distribution block size L.

The above discussion applies not only small to dense matrices but also to large skyline or band matrices arising from finite element calculations. These matrices often have a size of more than one million, but it is not rare that their skyline length or bandwidth is only one or two thousand. Because the inherent parallelism of such matrices is determined by the skyline length or bandwidth rather than dimension, the conventional blocked Gaussian elimination can perform only poorly when applied to these problems. Actually, this has been one of the obstacles to applying distributed-memory parallel machines with hierarchical memory to finite element problems.

3.3.2 The double blocked Gaussian elimination method

To overcome this difficulty, we propose an extension of the conventional blocked Gaussian elimination, the double blocked Gaussian elimination method[104][105]. In this method, we use two block sizes: the distribution block size L, which is used for distributing the coefficient matrix among the processing nodes and the algorithmic block size M, which is used for matrix multiplication arising in the elimination operation. The values of L and

M are determined so as to maximize concurrent efficiency and single-processor performance, respectively, under the condition that M is divisible by L. Using two block sizes makes it possible to fully exploit the potential performance of distributed-memory parallel machines with hierarchical memory even for problems with small inherent concurrency. In order to use different block size for elimination, after calculating a block pivot row $\{\mathrm{A}_{IK}^{(K+1)}\}_{I=K+1}^{N/L}$ and a block pivot column $\{\mathrm{A}_{KJ}^{(K+1)}\}_{J=K+1}^{N/L}$, one postpones the elimination until M/L pivot rows and columns are completed. After that, the elimination operations corresponding to M/L block stages are performed at once using these block pivot rows and columns. This method is an extension of the multistage elimination method [68] developed for vector supercomputers to a case where each element of the coefficient matrix is not a real number but is itself an $L \times L$ submatrix.

To describe the double blocked Gaussian elimination method in detail, we assume that N is a multiple of M and that M is a multiple of L and set $m = M/L$. Then the algorithm can be stated as in Algorithm 3.5.

In this algorithm, the allocation of the blocks (of size $L \times L$) to nodes, the frequency of interprocessor communication, and the amount of data to be transferred between processors are the same as in the case of the conventional parallel blocked Gaussian elimination method with block size L. The concurrent efficiency of the both methods can thus be considered almost the same. On the other hand, the single-processor performance of the algorithm is governed by the performance of the main body of elimination (the last operation in the above algorithm). As can be seen from Fig. 3.3, in the double block method, this operation can be cast into matrix multiplication with inner-product length M. We can thus expect this method to make use of the potential performance of distributed-memory parallel machines with hierarchical memory if we optimize the distribution block size L to maximize concurrent efficiency and optimize the algorithmic block size M to maximize single processor performance.

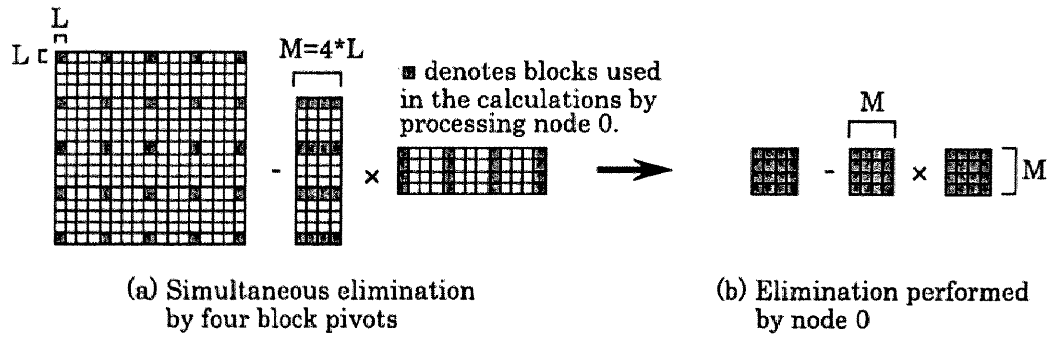

L
L
M=4*L
-
×
■ denotes blocks used in the calculations by processing node 0.
M
-
×
M
(a) Simultaneous elimination by four block pivots
(b) Elimination performed by node 0

Figure 3.3: The effect of double blocking.

```
[Algorithm 3.5 Double Blocked Gaussian Elimination Method]
do $K'$=1, $N/M$
  do $K$=$(K'-1)*m+1,\ K'*m$
    [LU factorization of the diagonal block]
    Node $d_K$ performs the LU factorization $\mathbf{A}_{KK}^{(K)} = \mathbf{L}_K\mathbf{U}_K$.
    [Broadcast of $\mathbf{L}_K$ and $\mathbf{U}_K$]
    Node $d_K$ broadcasts $\mathbf{L}_K$ in the row direction (to $R_K$).
    Node $d_K$ broadcasts $\mathbf{U}_K$ in the column direction (to $C_K$).
    [Formation of the $K$-th block pivot column]
    Node group $C_K$ computes $\mathbf{A}_{IK}^{(K+1)} = \mathbf{A}_{IK}^{(K)}\mathbf{U}_K^{-1}$
        for $K+1 \le I \le N/L$.
    [Broadcast of the $K$-th block pivot column]
    Node group $C_K$ broadcasts $\mathbf{A}_{IK}^{(K+1)}$ in the row direction.
    [Formation of the $K$-th block pivot row]
    Node group $R_K$ computes $\mathbf{A}_{KJ}^{(K+1)} = \mathbf{L}_K^{-1}\mathbf{A}_{KJ}^{(K)}$
        for $K+1 \le J \le N/L$.
    [Broadcast of the $K$-th block pivot row]
    Node group $R_K$ broadcasts $\mathbf{A}_{KJ}^{(K+1)}$ in the column direction.
    [Partial elimination to form next block pivot rows]
    $\mathbf{A}_{IJ}^{(K+1)} = \mathbf{A}_{IJ}^{(K)} - \mathbf{A}_{IK}^{(K+1)}\mathbf{A}_{KJ}^{(K+1)}$
        for $K+1 \le I \le K'*m$ and $K+1 \le J \le N/L$.
    [Partial elimination to form next block pivot columns]
    $\mathbf{A}_{IJ}^{(K+1)} = \mathbf{A}_{IJ}^{(K)} - \mathbf{A}_{IK}^{(K+1)}\mathbf{A}_{KJ}^{(K+1)}$
        for $K'*m+1 \le I \le N/L$ and $K+1 \le J \le K'*m$
  end do
  [Main body of the elimination operation]
  $\mathbf{A}_{IJ}^{(K'*m+1)} = \mathbf{A}_{IJ}^{((K'-1)*m+1)} - \mathbf{A}_{I,(K'-1)*m+1}^{((K'-1)*m+2)}\mathbf{A}_{(K'-1)*m+1,J}^{((K'-1)*m+2)}$
                               $-\mathbf{A}_{I,(K'-1)*m+2}^{((K'-1)*m+3)}\mathbf{A}_{(K'-1)*m+2,J}^{((K'-1)*m+3)}$
                               $-\dots-\mathbf{A}_{I,K'*m}^{(K'*m+1)}\mathbf{A}_{K'*m,J}^{(K'*m+1)}$
      for $K'*m+1 \le I,J \le N/L$.
end do
```

3.3.3 Analytical model for performance prediction

To evaluate the performance of the double blocked Gaussian elimination method, we first construct a model to predict performance when the dimension of the matrix, the number of processors, single-processor performance, interprocessor communication performance, and the two block sizes are given. Then, in the next subsection, we will predict the performance on two kinds of parallel computers, the nCUBE2 and the Hitachi SR2001, and confirm the prediction experimentally.

In constructing the model, we make the following four assumptions for simplicity:

1. Single-processor performance remains constant (s FLOPS) throughout the algorithm.

2. The time $t_{comm.}$ (sec) needed for interprocessor communication is a linear function of the amount of data (n bytes) to be transferred:

$$t_{comm} = t_{setup} + n/w, \qquad (3.2)$$

   where t_{setup} (sec) and w (bytes/sec) are setup time and throughput, respectively.

3. Only time for computation, time for communication, and idle time is taken into consideration.

4. The total execution time is divided into two kind of phases, the communication phase and the computation phase. In the former phase, each processor either joins in interprocessor communication or waits in an idle state. In the latter phase, each processor either performs computation or, if it has finished its computation, waits in an idle state.

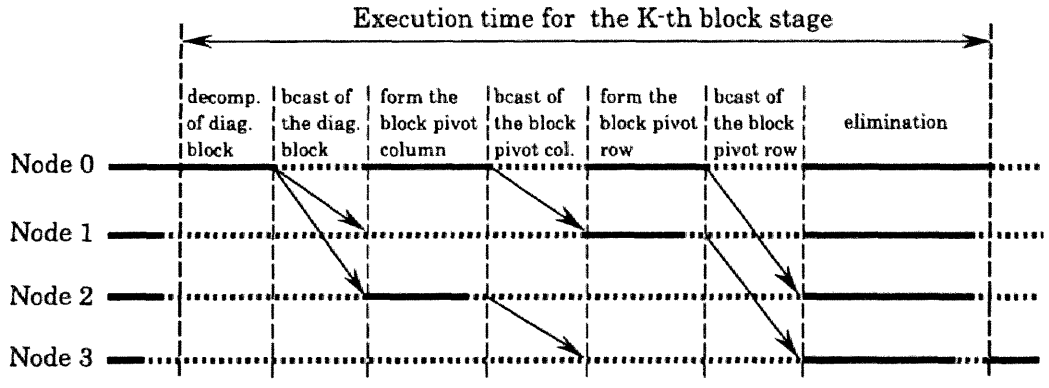
Because of the assumption 4, the length of each computation phase is governed by a node with the heaviest load. Under these assumptions, the time needed to perform one stage of the conventional blocked Gaussian elimination can be illustrated as in Fig. 3.4.

Under these assumptions, the concurrent efficiency of the double blocked Gaussian elimination method with distribution block size L can be considered almost the same as that of the conventional method with block size L, provided that the single-processor performance is the same. This is because the former differs from the latter only in that it postpones part of the elimination until M/L block pivots are completed. Therefore, we construct a model for the conventional method and estimate the performance of the double blocked method by using a formula for block size L but substituting the measured single-processor performance at block size M for the variable s in assumption 1.

In this model, the total execution time is given as the sum of the computation time and the communication time. The computation time consists of the following:

(a) $T_{LU}^{(o)}$: Time for the LU factorization of the diagonal block,

(b) $T_{PIVC}^{(o)}$: Time for formation of the block pivot column,



Execution time for the K-th block stage
decomp. of diag. block
bcast of the diag. block
form the block pivot column
bcast of the block pivot col.
form the block pivot row
bcast of the block pivot row
elimination
Node 0
Node 1
Node 2
Node 3

The above is a timing diagram for 4 nodes. The abscissa denotes time and solid lines show computation time. Dotted lines show communication time or idle time. Arrows denote data transfer between processing nodes.

Figure 3.4: Execution time of the blocked Gaussian elimination method.

(c) $T^{(o)}_{PIVR}$: Time for formation of the block pivot row, and

(d) $T^{(o)}_{MOD}$: Time for the main body of the elimination.

The communication time consists of the following:

(e) $T^{(c)}_{LU}$: Time for broadcasting the diagonal block,

(f) $T^{(c)}_{PIVC}$: Time for broadcasting the block pivot column, and

(g) $T^{(c)}_{PIVR}$: Time for broadcasting the block pivot row.

Let the dimension of the matrix, the block size, and the number of processing nodes be N, L, and P, respectively, and assume that nodes are allocated to blocks periodically in both directions with period $P^{1/2}$. For simplicity, let N be a multiple of $L * P^{1/2}$ and set $N' = N/L$ and $N'' = N/(L * P^{1/2})$. Then, for example, $T^{(o)}_{MOD}$ can be calculated as follows.

The number of floating point operations needed to eliminate one block is $2L^3$, and the maximum number of blocks taken charge of by one node is N''^2 for the first $P^{1/2}$ block stages, and $(N'' - 1)^2$ for the second $P^{1/2}$ block stages, and so on. Therefore, the total time needed for this part of calculation becomes

$$T^{(o)}_{MOD} = \frac{1}{3}L^3 * N'' * (N'' + 1) * (2N'' + 1) * P^{1/2}/s. \tag{3.3}$$

Times for other parts can be calculated in a similar way and the results are summarized in Table ??. The total execution time of the blocked Gaussian elimination can be expressed as the sum of these times:

$$T_{total} = T_{LU}^{(o)} + T_{PIVC}^{(o)} + T_{PIVR}^{(o)} + T_{MOD}^{(o)} + T_{LU}^{(c)} + T_{PIVC}^{(c)} + T_{PIVR}^{(c)}. \qquad (3.4)$$

Table 3.1: Execution time for each part of the blocked Gaussian elimination.

Item	Time
$T_{LU}^{(o)}$	$(2/3)L^3 * N'/s$
$T_{PIVC}^{(o)}$	$(1/2)L^3 * N'' * (N''+1) * P^{1/2}/s$
$T_{PIVR}^{(o)}$	$(1/2)L^3 * N'' * (N''+1) * P^{1/2}/s$
$T_{MOD}^{(o)}$	$(1/3)L^3 * N'' * (N''+1) * (2N''+1) * P^{1/2}/s$
$T_{LU}^{(c)}$	$N' * t_{setup} + L^2 * N'/w$
$T_{PIVC}^{(c)}$	$N' * t_{setup} + (1/2)L^2 * N'' * (N''+1) * P^{1/2}/w$
$T_{PIVR}^{(c)}$	$N' * t_{setup} + (1/2)L^2 * N'' * (N''+1) * P^{1/2}/w$

3.3.4 Experimental results

In this section, we evaluate the performance of our double blocked Gaussian elimination method using two types of distributed-memory parallel machines, namely, the nCUBE2 and the Hitachi SR2001 [104][105].

First, we predict the performance by using the model developed in subsection 3.3.3. The example problem is to solve linear simultaneous equations of size N=1024 with 64 nodes. We used t_{setup} = 120 (μs) and w = 0.85 (Mbytes/s), which are values obtained by approximating the measured communication time by a linear function. As single-processor performance, we used measured value listed in Table 3.2.

Table 3.2: Single-processor performance on the nCUBE2.

M	4	8	16	32	64
Performance (MFLOPS)	1.16	1.39	1.51	1.57	1.58

We estimated the performance for two series of parameters: (a) M is set equal to L and the both are varied from 4 to 64 (this corresponds to the conventional parallel blocked Gaussian elimination method), and (b) L is fixed to 4 and M is varied from 4 to 64. The results of prediction are shown in Fig. 3.5 by dotted lines. It is estimated that by setting L=4 and M=64, the new method can attain performance higher than the highest performance of the conventional method.

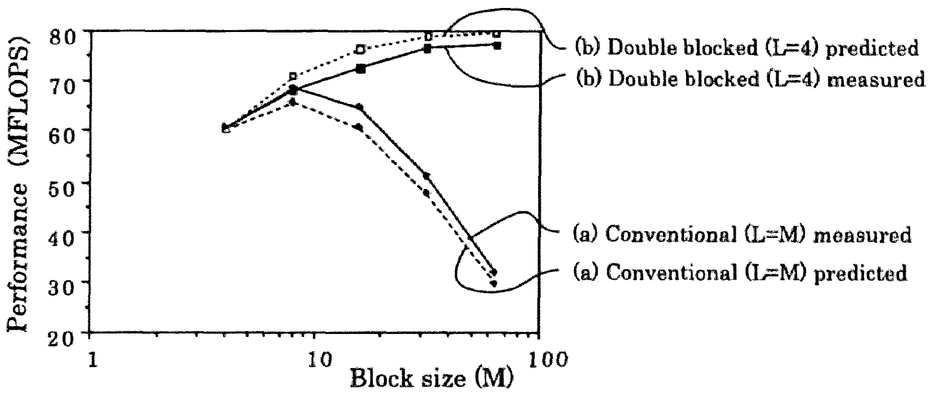


Figure 3.5: Parallel performance on the nCUBE2.

The measured performance is shown in Fig. 3.5 by solid lines. As estimated, the new method actually attained 77.14 MFLOPS when L=4 and M=64, which is 12% faster than the fastest conventional method (68.47 MFLOPS when L=M=8).

We also evaluated the performance of the new method on the HITACHI SR2001. Again, we first predicted the performance based on the model developed in subsection 3.3.3. The example problem is to solve linear simultaneous equations of size N=512 with 16 nodes. As t_{setup} and w, we used values in the hardware catalog. As single-processor performance, we used measured values listed in Table 3.3.

Table 3.3: Single-processor performance on the SR2001.

M	4	8	16	32	64
Performance (MFLOPS)	12.86	24.04	36.09	41.96	38.57

We estimated performance for two series of parameters: (a) M is set equal to L and the both are varied from 4 to 32 (this corresponds to the conventional method), and (b) L is fixed to 16 and M is varied from 16 to 32. The result of estimation are shown in Fig. 3.6 by dotted lines. It is estimated that by setting L=16 and M=32, the new method can attain performance higher than the highest performance of the traditional method.

The measured performance is shown in Fig. 3.6 by solid lines. As estimated, the new method actually attained 338 MFLOPS when L=16 and M=32, which is 19% faster than the fastest conventional method (283 MFLOPS when L=M=16). These results confirm the effectiveness of our new method on distributed-memory parallel machines with hierarchical memory.

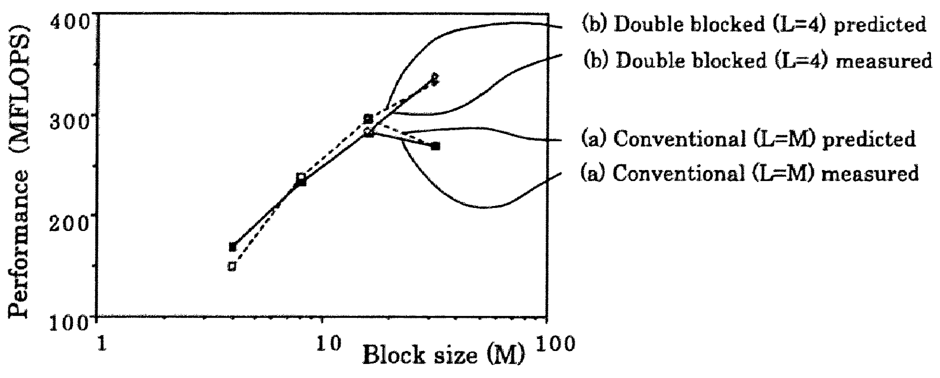


Figure 3.6: Parallel performance on the SR2001.

3.3.5 Application to other linear algebra algorithms

The idea of using two different block sizes for data distribution and algorithmic blocking can be applied to other linear algebra algorithms as well. For example, the algorithm presented in subsection 3.3.2 can be extended quite straightforwardly to incorporate partial pivoting. In fact, this extended algorithm has been implemented on CP-PACS [19], a massively parallel distributed-memory machine with 2048 processors, and contributed to winning the world's fastest record in the LINPACK benchmark [66] in 1996.

Another application is tri-diagonalization of real symmetric or complex Hermitian matrices by the Householder method [46][76]. In this case, the frequency of interprocessor communications does not depend on the distribution block size L and therefore the choice of L=1 is the best from the viewpoint of parallel efficiency. On the other hand, the algorithmic block size that attains the highest single-processor performance is determined from the capacity of the cache memory and is usually much larger than 1. Hence, the idea of double blocking is expected to work well. The readers are referred to [55][59] for more details.

Other potential applications include the QR factorization, reduction to the Hessenberg form of a unsymmetric matrix and bi-diagonalization for computing the singular value decomposition [46][89][90]. Application to sparse direct solvers [41][44][45] should also be explored as a future work.

Chapter 4

Direct Solution of Sparse Symmetric Positive Definite Matrices on Distributed-Memory Parallel Machines

4.1 Introduction

Structural analysis based on the Finite Element Methods is one of the most frequently used applications in the field of scientific computing. Recently, as the size of the problems increases, there is a growing need for using distributed-memory parallel machines which have high computational power and large memory spaces, and many software packages such as MARC and NASTRAN have been ported to this type of machines. In the structural analysis, much of the computation time is consumed to solve linear simultaneous equations. It is therefore crucial to develop an efficient linear equation solver for distributed-memory parallel machines.

In many structural analysis calculations, the coefficient matrix is a large sparse symmetric positive definite matrix with a large condition number. To solve linear simultaneous equations with such a coefficient matrix, sparse direct solvers have been widely used [41][44][45]. The sparse direct solver stores and computes only nonzero elements in the factorized matrix and thereby reduces the amount of memory and floating point operation compared with the skyline solvers.

There have been many studies to parallelize the sparse direct solver, and efficient distributed-memory parallel algorithms have been proposed for each part of the solver, including reordering [20][58][77], symbolic factorization [44], Cholesky factorization [6][9][49] [82][83], and forward and backward substitution [48]. However, there are not many distributed-memory parallel sparse solver programs which performs all of these phases; some examples of such solvers are CAPSS [51], which was developed as a part of the ScaLAPACK project, MUMPS [4][5], which was developed in a European joint project PARASOL [75], PSPACES [78], which was developed by Kumar et al. at Minnesota University.

We developed a complete sparse direct solver which performs all of the above four steps [106][107]. The features of our solver are as follows:

1. It can solve linear simultaneous equations whose coefficient matrix is symmetric positive definite and has a 3×3 block nonzero structure. Such matrices arise in 3-dimensional structural analysis problems.

2. It performs the reordering step on one processing node and performs the following steps on multiple nodes. By adopting this strategy, it can utilize the large memory space of a distributed-memory parallel machine, while retaining an easy-to-use interface.

3. The single-processor performance in the Cholesky factorization part is enhanced using a locally optimized loop unrolling technique. This is made possible by exploiting the block nonzero structure of the matrix.

Compared with the sparse solver programs listed above, advantages and disadvantages of our solver can be stated as follows:

- Comparison with MUMPS

  In MUMPS [4][5], one of the processing nodes is designated as the host node and the user is required to store the whole coefficient matrix on the memory of the host node when calling the solver. This interface has the advantage that it is easy to use, because it is the same interface as that of the sequential sparse solver. However, it has the disadvantage that the size of problems that can be solved is limited by the memory size of the host node. In contrast, our solver accepts a coefficient matrix that is scattered among the processing nodes, allowing the user to solve larger matrices. In addition, our solver has an optional matrix distribution routine which provides the user with the same interface as that of MUMPS.

- Comparison with CAPSS

  In CAPSS [51], all the four steps of the sparse solver including reordering are executed on distributed memory. As an reordering algorithm, it adopts the *Cartesian Nested Dissection method* [51], which is simple and easy to parallelize on distributed memory. However, it is known that this algorithm is not as effective as other popular reordering algorithms [20][58][77] in reducing the fill-ins. In contrast, our solver utilizes a more effective reordering algorithm called the *Multilevel Nested Dissection* (MND) [20][58]. Because MND is hard to parallelize, we chose to execute only the reordering step on a single node. Though this causes the possibility of limiting the size of solvable problems, we circumvent it by exploiting the 3×3 nonzero structure of the coefficient matrix and perform reordering on a smaller matrix with 1/3 columns and 1/3 rows. This reduces the memory required for reordering by 1/9 and contributes to extending the limit of solvable problems.

- Comparison with PSPACES

  In PSPACES [78], as in CAPSS, all the four steps of the sparse solver are executed on distributed memory. As an reordering algorithm, it uses a newly developed variant of the Multilevel Nested Dissection [78] which is designed to be both effective and easy to parallelize. This would be an ideal solution if the new reordering method is as powerful as the current one. In this study, however, we take a conservative approach and use a current version of the MND method that has proved effective for many types of problems.

The rest of this chapter is structured as follows: we explain basic concepts of the sparse direct solver in section 4.2 and details of parallelization in section 4.3. In sections 4.4 and 4.5, we will explain optimization techniques for enhancing single-processor performance and the result of performance evaluation on the SR2201 distributed memory parallel computer, respectively. Finally, we will conclude our study in section 4.6.

4.2 Basics of the sparse direct solver

4.2.1 Components of a sparse direct solver

A sparse direct solver finds a solution to a system of linear simultaneous equations $\mathbf{Ax} = \mathbf{b}$ by performing the four steps, namely reordering, symbolic factorization, Cholesky factorization, and forward and backward substitution in this order. We will explain briefly each of the four phases [41][44].

(1) Reordering

In the reordering phase, the rows and columns of the coefficient matrix $\mathbf{A}$ are permuted simultaneously using a permutation matrix $\mathbf{P}$, and $\mathbf{A}$ is transformed into $\mathbf{A}' = \mathbf{PAP}^t$. The permutation matrix $\mathbf{P}$ is chosen so that the number of fill-ins after the Cholesky decomposition $\mathbf{A}' = \mathbf{L}\mathbf{L}^t$ is as small as possible, and at the same time, parallelism in the Cholesky decomposition is as high as possible. Popular methods for reordering includes the *Minimum Degree (MD) method* [29], which is based on the idea of locally minimizing the number of fill-ins generated at each step, and the *Nested Dissection (ND) method* [45][20][58][77], which is based on recursive partitioning of the underlying mesh.

In the ND method, the FEM mesh is partitioned into two subregions A and B by a vertex set called *separator*. The separators are chosen so that A and B contain roughly equal number of vertices and there is no edge that connects a vertex belonging to A and that belonging to B. After the partition, the vertices are renumbered so that the vertices in A are numbered first, those in B next, and those in the separator last. From the definition of the separator, there are no matrix elements whose column belongs to set A and whose row belongs to set B, or vice versa, and the matrix is transformed into a *bordered block diagonal form*. By repeating this partitioning and renumbering recursively for each subregion, the matrix is transformed into a recursive bordered block diagonal matrix. A matrix arising from the five-point finite difference formula on a 5 by 5 lattice shown in Fig. 4.1 and reordered using the ND method is shown in Fig. 4.2.

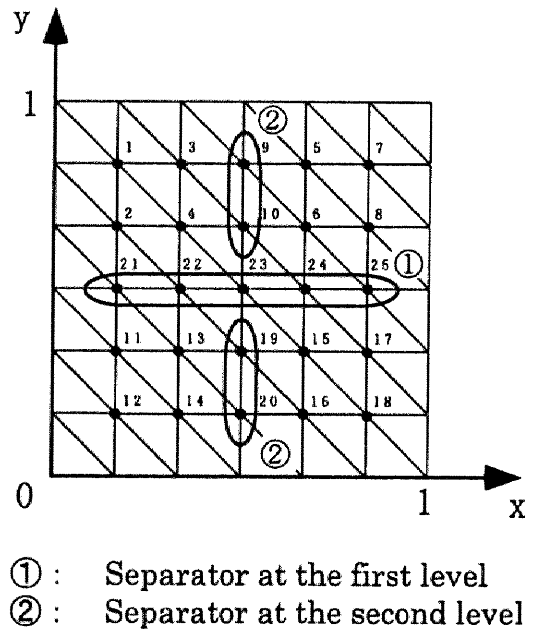


Figure 4.1: An example of finite element mesh and the separators.

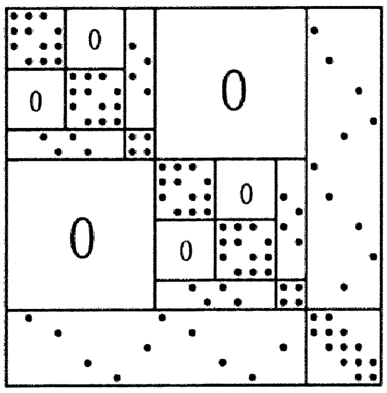


Figure 4.2: A matrix generated by the FEM and reordered by the ND method.

(2) Symbolic factorization

In this step, the positions of fill-ins are calculated prior to the Cholesky decomposition, and the storage area to store them is allocated. At the same time, the index list to access the nonzero elements is generated.

(3) Cholesky factorization

The Cholesky decomposition $\mathbf{A}' = \mathbf{L}\mathbf{L}^t$ is calculated using the index list generated in the symbolic factorization phase. In this phase, the central operation of the Gaussian elimination

$$a_{ij} = a_{ij} - a_{ik}a_{kj}/a_{kk} \tag{4.1}$$

is performed only for the lower triangular part, using the symmetry of the matrix. Also, only those elements which become nonzero after factorization are calculated. Variants of the algorithm include the *right-looking algorithm* (also referred to as *kij-form* or *outer-product form*), where the outermost loop index is k, the *left-looking algorithm* [9] (also referred to as *jik-form* or *inner-product form*), where the outermost loop index is j, and the *multifrontal algorithm* [40][67], which is similar to the right looking algorithm, except that it accumulates the updates to the matrix in a small full matrix called the *frontal matrix*, and perform the update later. Consult [10][44][81] for comparison of these three approaches.

(4) Forward and backward substitution

The solution of the linear simultaneous equation is calculated by solving $\mathbf{L}\mathbf{y} = \mathbf{b}$ and $\mathbf{L}^t\mathbf{x} = \mathbf{y}$ successively.

The flow chart of the sparse direct solver is shown in Fig. 4.3. By separating the symbolic factorization phase from the Cholesky factorization phase, it becomes unnecessary to repeatedly calculate the same index list, when many sets of equations with the same nonzero structure are to be solved, as in the case of Newton iteration in the nonlinear problems. Also, by separating the forward and backward substitution phase, when many sets of equations with the same coefficient matrix are to be solved, as in the case of time dependent problems, only the forward and backward substitution phase needs to be repeated.

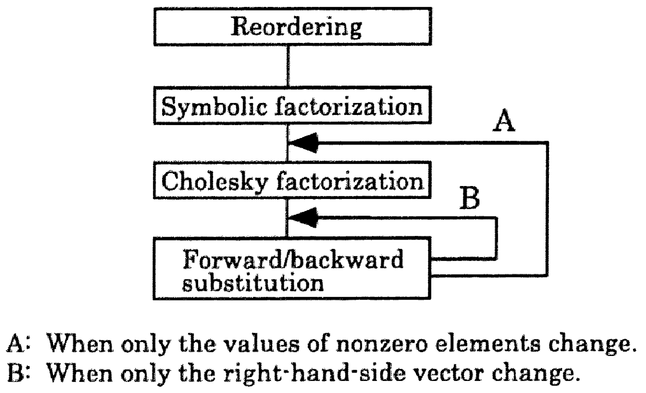

Reordering
Symbolic factorization
A
Cholesky factorization
B
Forward/backward substitution
A: When only the values of nonzero elements change.
B: When only the right-hand-side vector change.

Figure 4.3: Flow chart of the sparse solver.

4.2.2 The elimination tree and the parallelism in the Cholesky decomposition

Definition of the elimination tree

Next we will explain the concept of the *elimination tree* [44] which plays an important role in parallelizing the sparse solver. The elimination tree is a rooted tree defined using the nonzero structure of the Cholesky factor **L** and each vertex of the tree corresponds to a column of **L**. For two vertices i and j $(i > j)$ of the tree, i is defined as the parent of j if $i = \min\{k > j | L_{kj} \neq 0\}$ and is denoted as $i = p(j)$. When the matrix is irreducible, i.e., if it cannot be transformed into a block diagonal matrix with the simultaneous permutation of its rows and columns, there is only one tree that has the vertex N as its root. The nonzero structure of the Cholesky factor of the matrix shown in Fig 4.2 is illustrated in Fig. 4.4, while the corresponding elimination tree is shown in Fig. 4.5. The number in the circle denotes the column number. The recursive structure of the tree reflects the recursive bordered block-diagonal structure of the matrix; the subtrees correspond to the diagonal blocks, while the chains between branches correspond to the border of the matrix.

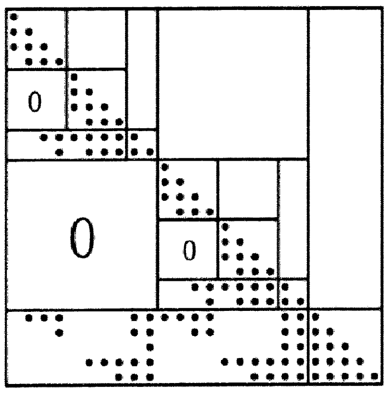


Figure 4.4: The nonzero structure of the matrix in Fig. 4.2 after factorization.

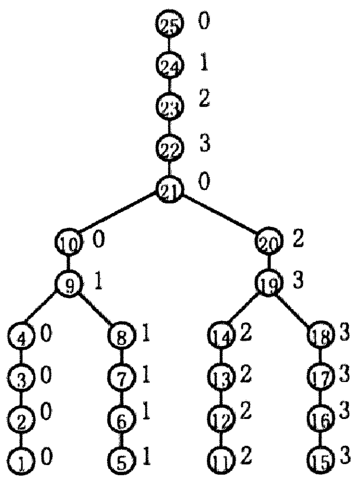


Figure 4.5: The elimination tree corresponding to the matrix in Fig. 4.4.

Parallelization of Cholesky decomposition using the elimination tree

One of the important properties of the elimination tree is that elimination by column j affects column i only if node i is an ancestor of node j in the tree. Consequently, if the left-looking algorithm is used, all the columns used in the elimination of column i are descendants of node i. This means that if two nodes belong to two different subtrees which do not share any nodes, the elimination operations for them can be done independently.

Using this fact, a matrix partitioning strategy called *subtree-to-subcube mapping* [44] has been proposed and is widely used. In this strategy, the number of processing nodes is assumed to be a power of two, and the nodes are allocated cyclically to the vertices of the tree from the root downwards. At the first branch, half of the processing nodes are allocated to the left subtree, while the other half are allocated to the right subtree. This process is continued recursively until there is only one processing node in each group, and then the whole subtree is allocated to that node. In Fig. 4.5, we show the node number of the processing node allocated to each vertex when the subtree-to-subcube mapping is applied to the elimination tree.

By using this mapping, the elimination operation for a subtree allocated to a processing node can be done without interprocessor communication. Consequently, to maximize the parallelism, it is necessary to reorder the matrix so that the number of vertices in these subtrees is maximized (namely, the number of vertices in the separators are minimized) and at the same time, the difference in the sizes of the subtrees is minimized.

4.2.3 Data structures

The partial matrix data allocated to each processing node by the subtree-to subcube mapping is stored using the following three arrays.

1. A 1-dimensional array containing all the nonzero elements of the partial matrix. The elements are stored in the ascending order of the column number, and within each column, in the ascending order of the row number.

2. A 1-dimensional array containing the row numbers of all the nonzero elements of the partial matrix. The elements are stored in the ascending order of the column number, and within each column, in the ascending order of the row number.

3. A 1-dimensional array of size $N + 1$ containing pointers to the first elements of each column in arrays 1 and 2.

In addition to these, we use a 1-dimensional array of size N which stores a mapping from a column number to the node number which takes charge of it, and another 1-dimensional array which stores a mapping from a column number to the column number of its parent. All the processing nodes have copies of these two arrays.

4.3 Paralleization of the sparse direct solver

4.3.1 The target of parallelization

In this subsection, we will describe how to parallelize each part of the sparse direct solver.

Among the four phases of the sparse direct solver, our program executes only the reordering phase on one processing node. It then distributes the reordered matrix to nodes using a utility routine and executes the remaining phases on all the nodes. The reason for adopting this strategy is as follows:

1. It is more convenient for the user to pass the whole matrix to the solver, especially because the standard interface for passing partitioned matrices from the FEM program to the solver has not yet been established.

2. Because the number of nonzero elements in the coefficient matrix is about an order of magnitude smaller than that in the factorized matrix, storing the whole coefficient matrix does not become a severe constraint on the size of the problem that can be solved, as long as the number of nodes is ten or so.

By adopting this strategy, it becomes possible to efficiently use the large address space of a distributed-memory parallel machine, while retaining an easy-to-use interface.

4.3.2 Parallelization of each part

(1) Reordering

We used a variant of the ND method called the Multilevel ND method [20][58]. This algorithm approximates the input mesh by a coarser mesh, partitions it, and maps the partitioning again to the original mesh. The partitioning of the coarser mesh is done recursively using the same procedure. The MND method has a desirable property that a high quality partitioning with small separator and equally-sized subtrees can be obtained at a relatively small cost, compared with other methods based on the ND approach, such as the spectral ND method [77].

After partitioning the mesh using the MND method, our program reorders the nodes within each subregion using the Approximate Minimum Degree method [29]. By doing this, the number of operations needed for the Cholesky decomposition can be further reduced.

(2) Partitioning of the matrix

In this part, an elimination tree for the reordered coefficient matrix is generated, and the columns of the matrix are allocated to processing nodes using the subtree-to-subcube mapping. For the columns corresponding to the chains of the tree, a block cyclic mapping is used. In determining the block size L, we have to consider two factors: increasing L will reduce the frequency of interprocessor communication and the overhead due to it, but may cause load imbalance among the nodes. Decreasing L will give rise to the opposite effect. In our study, we chose $L = 12$ from the result of preliminary experiments.

(3) Symbolic factorization

The position of nonzero elements after decomposition is calculated, and an index list to access the nonzero elements of the decomposed matrix is generated. The index list is expressed as a list of row numbers of nonzero elements in each row.

(4) Cholesky factorization

The left-looking algorithm is used for the Cholesky decomposition due to its relatively small communication requirement. In this algorithm, as we stated in subsection 4.2.2, elimination operation within a subtree allocated to one processing node can be done without communication. As the processing proceeds, the number of nodes which work together increases to two, four, eight, and so on. For calculation above the uppermost branch, all the nodes cooperate.

(5) Forward and backward substitution

In terms of the elimination tree, the forward elimination is a process in which the elements of solution corresponding to the leaves are calculated first, and the other solution are calculated upward. While the backward substitution is a process in which the elements of the solution corresponding to the root is calculated first, and the other solutions are calculated downward.

There are two variants of the forward elimination, depending on whether the index of the inner loop is i or j, in the calculation of $y_i = v_i - L_{ij}y_j$. Here, because the index list expresses the row number i of nonzero elements in each row j, the outer-product algorithm, in which the inner loop index is i, is more desirable. But, if the outer-product algorithm is applied straightforwardly, it will incur much interprocessor communication, because even when y_j is in a subtree allocated to one node, the y_i's, which is to be modified by y_j, is in general allocated to other processors.

In our program, we solve this problem by accumulating the modification by y_i in a temporary array within the processor, and make a update to y_j allocated to another node only after the processing of the subtree allocated to the former processor has been finished. Thus, the processing of subtrees allocated to each node can be done without communication.

On the other hand, in the backward substitution step, the inner-product form, in which the inner loop index in the calculation of $y_i = y_i - L_{ji}y_j$ is j, is more preferable. In this algorithm, to calculate y_i, the y_j's corresponding to the ancestors of y_i are required. So, by making sure that all the solutions obtained so far have been sent to the processor prior to the calculation of the subtree, the calculation within that subtree can be done without communication.

4.4 Optimization for enhancing the single-processor performance

Among the five phases of the parallel sparse solver stated in the last section, the most time-consuming part is the Cholesky decomposition, whose central operation is a matrix-vector multiplication $\mathbf{c}_k = \mathbf{c}_k - \mathbf{A}_k\mathbf{b}_k$ as shown in Fig 4.6. Actually, because the algorithm is blocked, $\mathbf{b}_k$ and $\mathbf{c}_k$ are also matrices whose width is equal to the block size L, and the operation becomes a matrix multiplication $\mathbf{C}_k = \mathbf{C}_k - \mathbf{A}_k\mathbf{B}_k$.

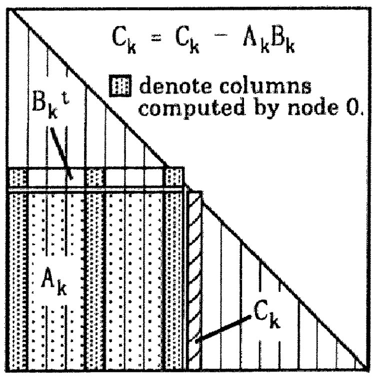


Figure 4.6: Kernel operation of the Cholesky factorization.

In this operation, because all the matrices $\mathbf{A}_k$, $\mathbf{B}_k$ and $\mathbf{C}_k$ are sparse, it is necessary to access the elements using the index list. This lowers the single-processor performance if, as is the case with most microprocessors, the processor has no special hardware for index list addressing. Moreover, the loop unrolling technique used to reduce the load/store and increase the performance cannot be applied straightforwardly, because the number and the position of nonzero elements differs for each iteration. A straightforward application would reduce the performance, for it would increase the operation count by operating also on zero elements.

To solve these problems, we use locally optimal loop unrolling patterns for each part of the matrix. In case of three-dimensional structural analysis problem, the nonzero elements appear in a 3 by 3 block. So, we decompose the nonzero pattern of the matrix $\mathbf{B}_k$ into a direct sum of the three kind of blocks, whose sizes are 6 by 2, 3 by 3, and 6 by 1 each, as shown in Fig. 4.7. Then we apply loop unrolling of (6, 2), (3, 3), or (6, 1) to (J, I) of Fig. 4.7, according to the chosen block. In this way, the loop unrolling technique can be applied without increasing the operation count. Because the (6, 2) unrolling attains the highest speed for the machine we used in the experiment, we decompose $\mathbf{B}_k$ so that the number of 6 by 2 blocks is maximized. The decomposition of the nonzero structure into these blocks is done in the symbolic factorization step, so its overhead is negligible when many sets of equations with the same nonzero structure are to be solved.

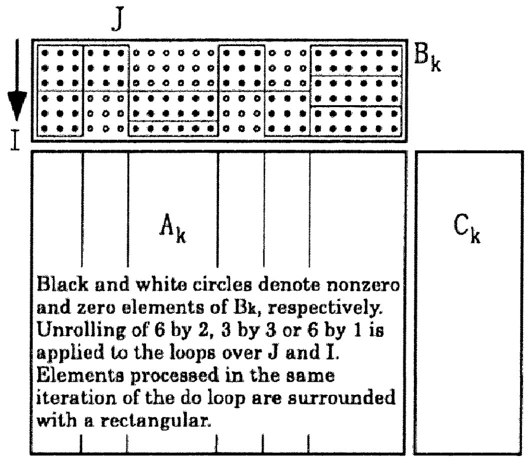

J
I
B_k
A_k
C_k
Black and white circles denote nonzero and zero elements of Bk, respectively. Unrolling of 6 by 2, 3 by 3 or 6 by 1 is applied to the loops over J and I. Elements processed in the same iteration of the do loop are surrounded with a rectangular.

Figure 4.7: Optimized loop unrolling for the local nonzero structure of the matrix.

4.5 Performance evaluation

We evaluated the performance of our parallel sparse solver on the Hitachi SR2201 distributed-memory parallel computer. We used a three-dimensional structural analysis problem of size N=70,032 as a test problem and varied the number of processing nodes P from 4 to 16. We also measured the performance for P=1 and 2, but in these cases, we used a smaller problem of N=32,274 due to memory limitation.

4.5.1 Parallel speedup

The execution times for the symbolic factorization, Cholesky factorization and forward/backward substitution are shown in Table 4.1 as a function of P. Here, we used the technique introduced in the previous section to enhance the single-processor performance in the Cholesky factorization part.

From the table, it can be seen that the Cholesky factorization part is sped up by 2.5 times when P is increased from 4 to 16, while the speedup of the symbolic factorization and forward/backward substitution parts is less than twice. This is because the latter two parts require the same number of interprocessor communication as the Cholesky factorization part, although they have far less computational work to distribute among the processing nodes. Note, however, that the relatively small speedup of these two parts does not affect the total parallel performance severely, for their contribution to the total execution time is small.

Table 4.1: Execution time for each part of the sparse solver (in seconds).

	P=4	P=8	P=16
Symbolic factorization	0.55	0.48	0.29
Cholesky factorization	23.86	16.17	9.68
Forward/backward substitution	0.65	0.57	0.35
Total	25.06	17.22	10.32

We can further analyze the parallel performance of the Cholesky factorization part by using the elimination tree. The elimination tree corresponding to this problem is illustrated in Fig. 4.8. In the Figure, we show only branching vertices and omitted vertices below the fourth level, where the tree is divided into 16 subtrees. The numbers to the right and left of a branching vertex denote the number of vertices belonging to the right and left subtrees, respectively, right below the branching vertex.

From these numbers, we know that the distribution of vertices among the subtrees is highly equal up to the second level (which corresponds to parallelization with 4 processing nodes), but a severe imbalance occurs at the third level, at a vertex marked as α. As a result, one of the nodes has to take charge of 16 thousand vertices, which is about twice the number allocated to other nodes. We can deduce that this is the reason why the speedup from P=4 to P=8 is modest (23.86/16.17=1.48). In contrast, vertices are distributed relatively equally at the fourth level. This is in accordance with the observation that the speed up from P=8 to P=16 is higher (16.17/9.68=1.67).

The imbalance at vertex α is caused because the finite element mesh in this problem is irregular and is difficult to partition automatically by the MND method. It remains our future work to improve the MND method so that it can generate high quality partitioning for such irregular problems. Another possibility is to modify the subtree-to-subcube mapping strategy so that it can allocate equal number of vertices to each node even if the elimination tree is unbalanced, by exploiting tree structures at finer levels.

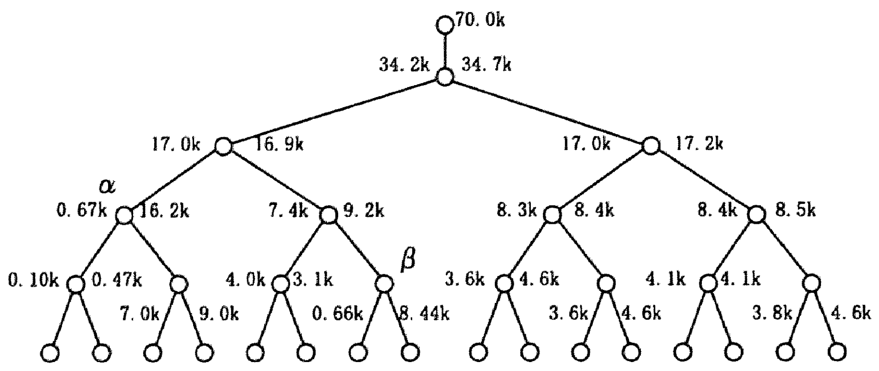


Figure 4.8: The elimination tree for the 3-dimensional structural analysis problem.

4.5.2 Effect of optimization

Next we compared the performance of the Cholesky factorization part with and without optimization explained in the section 4.4. The result of measurement is shown in Fig. 4.9. Here we used the problem of N=32,274 for the cases of P=1 and 2 and the problem of N=70,032 for the cases of P=4, 8 and 16. The single-processor performance with optimization is about 100MFLOPS, which is 20% higher than that without optimization. When the number of nodes is increased to 16, speedup of about 10 times was obtained.

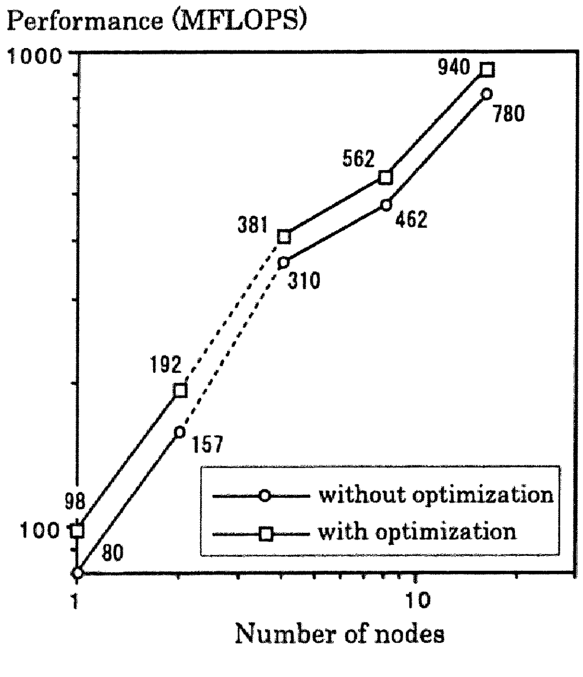


Figure 4.9: Parallel performance for the 3-dimensional structural analysis problem.

4.6 Conclusion

We developed a sparse direct solver for distributed-memory parallel machines and evaluated its performance on the Hitachi SR2201 parallel computer. This solver is designed to deal with sparse symmetric positive definite matrices with 3×3 block nonzero structure, and adopts a locally optimized loop unrolling technique in the Cholesky factorization part for enhancing single-processor performance. For a structural analysis problem of size N=70,032, our program attained 100MFLOPS on one processing node and speedup of about 10 times when the number nodes is 16.

Our future work will include optimization of the other phases of the solver including the symbolic factorization and forward and backward substitution, improvement of the reordering and allocation algorithms, and performance comparison with other parallel sparse solver algorithms such as the parallel multifrontal method.

Chapter 5

Direct Solution of Unsymmetric Tridiagonal Matrices on Shared-Memory Machines

5.1 Introduction

The problem of solving linear simultaneous equations with a tridiagonal coefficient matrix arises in many areas of scientific computing. Typical applications include computation of eigenvectors of a tridiagonal matrix by the inverse iteration method [46][101], solution of a partial differential equation by the ADI method [98] and interpolation by spline functions [79]. When the size of the matrix is large, it is appropriate to accelerate the solution with the use of parallel computers. Many approaches for the parallel solution of tridiagonal matrices have been proposed so far, including the dissection method [50], the cyclic reduction method [54][91] and a method based on the QR decomposition [7].

In the cyclic reduction method, the odd-numbered variables in the equations are eliminated first and the number of equations is reduced by half. This procedure is repeated until the number of equations is sufficiently small. This method has a large degree of parallelism and has been successfully implemented on vector processors [92]. Also, extensions to block tridiagonal matrices [54] and band matrices [39] have been proposed. In this method, however, the order in which the variables are eliminated is predetermined and pivoting for numerical stabilization cannot be incorporated. This makes it difficult to apply this method to general unsymmetric tridiagonal matrices.

The dissection method is based on the Cholesky decomposition and extracts the parallelism in the elimination operation by renumbering the variables and equations [45][50]. It can be extended to unsymmetric tridiagonal matrices by using the LU decomposition instead of the Cholesky decomposition. However, because the order of variable elimination is fixed also in this method, the type of matrices to which it is applicable is limited to symmetric positive definite matrices or diagonal dominant matrices.

The approach based on the QR decomposition [7], on the other hand, can be applied to general unsymmetric tridiagonal matrices. However, it is known that the QR decomposition also needs pivoting to produce accurate solution when the matrix is nearly singular [30]. Hence, the applicability of this method without pivoting is also limited.

In this chapter, we propose a new parallel direct solver for unsymmetric tridiagonal matrices [109][111]. Our method is a variant of the dissection method that can incorporate partial pivoting and can solve linear simultaneous equations with general unsymmetric tridiagonal coefficient matrices on parallel machines efficiently and accurately.

A parallel direct solver for unsymmetric tridiagonal matrices with pivoting has also been proposed in [52]. However, in this method, there is a tradeoff between the ratio of the sequential part in the algorithm and the number of interprocessor synchronizations. More specifically, if we denote the number of columns that have to be eliminated sequentially by $2P$ and the number of interprocessor synchronizations by Q, PQ is equal to the matrix size N. In contrast, our method has the advantage that it needs only one interprocessor synchronization and the number of columns that have to be eliminated sequentially is independent of N.

This chapter is organized as follows: In section 5.2, we briefly review the conventional dissection method applied to tridiagonal matrices along with the difficulty arising in the case of unsymmetric matrices. Our new parallel direct solver which incorporates partial pivoting is introduced in section 5.3. Numerical results on the Hitachi SR8000, a shared-memory parallel computer with 8 processors, can be found in section 5.4. Concluding remarks are given in the final section.

5.2 The Dissection Method and Its Limitation

5.2.1 Tridiagonal solver based on the dissection method

We consider a problem of solving a linear simultaneous equation $\mathbf{Tx} = \mathbf{b}$, where $\mathbf{T}$ is a unsymmetric tridiagonal matrix of order N. In the dissection method, we first transform $\mathbf{T}$ to $\mathbf{T}' = \mathbf{PTP}^t$ with a permutation matrix $\mathbf{P}$ and then solve a new equation $(\mathbf{PTP}^t)(\mathbf{Px}) = \mathbf{Pb}$ by Cholesky decomposition. $\mathbf{P}$ is determined so that the parallelism in the decomposition phase is maximized.

As is well known [45][50], a matrix $\mathbf{A}$ with symmetric nonzero pattern can be represented by a non-directed graph $G_\mathbf{A}$. $G_\mathbf{A}$ has N vertices that correspond to rows of $\mathbf{A}$ and $G_\mathbf{A}$ has an edge between two vertices i and j if and only if $A_{ij} \neq 0$. The graph $G_\mathbf{T}$

corresponding to $\mathbf{T}$ is a chain, as shown in Fig. 5.1(a). We can identify the simultaneous permutation of rows and columns $\mathbf{T}' = \mathbf{PTP}^t$ with renumbering of $G_\mathbf{T}$.

To solve $\mathbf{Tx} = \mathbf{b}$ on a parallel computer with P processors using the dissection method, we divide $G_\mathbf{T}$ into P subregions and $P - 1$ boundary vertices. Then we renumber the vertices so that the vertices in the first subregion are numbered first, those in the second subregion are numbered second, and so on, and the $P - 1$ boundary vertices are given numbers from $N - P + 2$ to N. The graph $G_\mathbf{T}$ with new vertex numbers is shown in Fig. 5.1(b) for the case of $P = 3$. Here, the boundary vertices are represented by shaded circles.

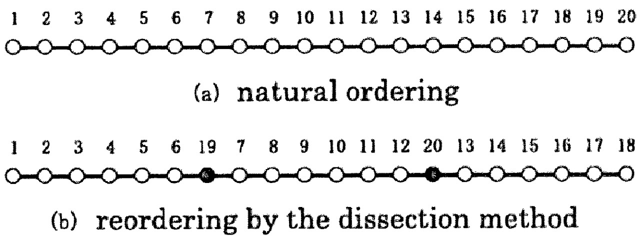


Figure 5.1: A graph associated with a tridiagonal matrix.

By applying the corresponding permutation of rows and columns to $\mathbf{T}$, we obtain a matrix shown in Fig. 5.2. It can be seen from the figure that the original tridiagonal matrix T is transformed into a bordered block diagonal matrix with three diagonal blocks. When pivoting is not used, the Cholesky decomposition of each diagonal block can be performed independently. Thus we can solve the tridiagonal equation $\mathbf{Tx} = \mathbf{b}$ in parallel using p processors.

5.2.2 Problems in the case of unsymmetric matrices

When solving linear simultaneous equations with unsymmetric coefficient matrix using direct methods, it is in general necessary to perform pivoting to ensure accuracy and numerical stability [30][46]. The most commonly used method for pivoting is the partial pivoting, which chooses the element in the pivot column with the largest modulus as the pivot element. In this subsection, we study how the parallelism in the dissection method for $\mathbf{Tx} = \mathbf{b}$ is affected when the partial pivoting is introduced.

Assume we apply Gaussian elimination with partial pivoting to a matrix shown in Fig. 5.2. The nonzero pattern after the first 6 columns (which corresponds to the vertices in the first subregion in the graph of Fig. 5.1(b)) have been eliminated is shown in Fig. 5.3. The actual nonzero pattern depends on the sequence of the row numbers of the pivot elements chosen; $\sigma = (n_1, n_2, \ldots, n_6)$, where $i \leq n_i \leq N$. The nonzero pattern displayed in the figure is the union of nonzero patterns over all possible σ's. In the figure, the elements modified by the elimination operation are denoted by squares with oblique lines, while the elements generated by fill-ins are denoted by black squares.

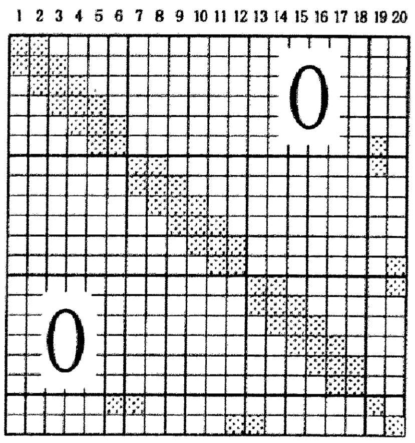


Figure 5.2: A tridiagonal matrix reordered by the dissection method.

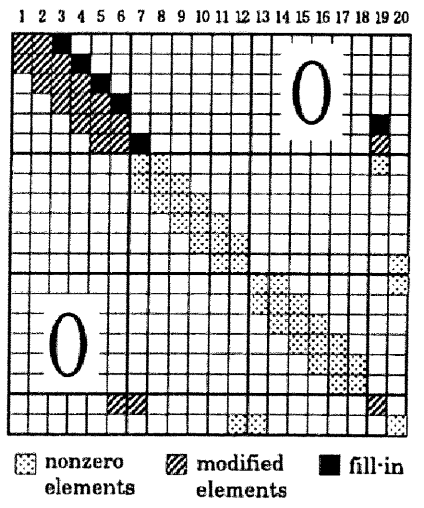


Figure 5.3: A tridiagonal matrix reordered by the dissection method.

From the figure, we can see that the element in the 7th column and the 2nd row from the last has been modified due to the elimination. However, in the elimination of the 7th column, this element is one of the candidates of the pivot element, because we choose the pivot element from all the elements in the column below the diagonal. This means that we cannot start the elimination of the 7th column until the value of this element has been determined, that is, until the first six columns have been eliminated. Hence introduction of partial pivoting causes dependence of the elimination operations in the second subregion on those in the first subregion, thereby destroying the parallelism.

5.3 A Parallel Tridiagonal Solver with Partial Pivoting

5.3.1 The basic idea

In this section, we propose a new parallel direct solver for unsymmetric tridiagonal matrices that can incorporate partial pivoting [109][111]. We achieve this by modifying the reordering scheme in the conventional dissection method.

In the example shown in the previous subsection, the dependence of the elimination operations was caused due to the existence of the nonzero element in the 7th column and the 2nd row from the last. This element is nonzero because the rightmost vertex in the first subregion is connected with the leftmost vertex in the second subregion through a boundary vertex (vertex 19 in Fig. 5.1(b)). In our algorithm, we dissolve this dependence by renumbering the vertices again in each subregion of Fig. 5.1(b). More specifically, in each subregion, the vertex numbers of all the "purely inner" vertices, which are not adjacent to any boundary vertices, are decremented by one, and the leftmost vertex in the subregion is given the second largest number in the subregion. By reordering all the subregions in this manner, we obtain the numbering of the vertices shown in Fig. 5.4.

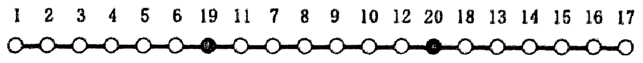


Figure 5.4: Reordering of the nodes by the proposed method.

The new matrix corresponding this renumbering is shown in Fig. 5.5. Because the leftmost vertex in each subregion is given the second largest number in the subregion, the element that caused the dependence of elimination operation is moved to the second last column in the subregion.

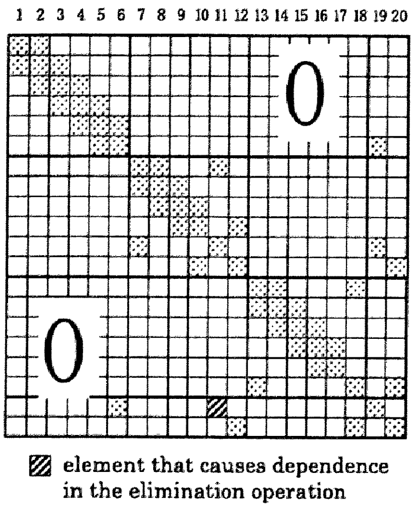


Figure 5.5: A tridiagonal matrix reordered by the proposed method.

5.3.2 Parallelism in the elimination operation

In Fig. 5.6, we show the block structure of the nonzero pattern of the matrix shown in Fig. 5.5. Here, the columns of the matrix are divided into sets that correspond to purely inner vertices (A, B and C), inner vertices that are connected to boundary vertices (the thin column sets right after the sets A, B and C) and the boundary vertices (the last set). Likewise, the rows are divided into sets that correspond to three subregions and the boundary vertices. Each block of the matrix is shaded if there are at least one nonzero element in the block, and is white otherwise.

Now we focus on the column set B. Among the four blocks in B, only the second one contains nonzero elements and the blocks left to this block contain no nonzero elements. As a result, the columns in B are not modified by the elimination of columns left to B, even if partial pivoting is introduced. This means that we can start elimination of columns in B before elimination of columns in A has been completed. Similarly, because the only nonzero block in C is the third one and all the blocks left to this block is zero, the columns in C is not modified by the elimination of columns left to C. So we can start eliminating columns in C without waiting for the completion of the elimination of columns in A and B.

Fig. 5.7 illustrates the nonzero structure of the matrix shown in Fig. 5.5 after elimination by the 6th column. As we have explained now, the columns in B (the 7th to the 10th column) have not been modified by the elimination of the 1st to the 6th columns. By using this new renumbering, we can eliminate the columns in A, B and C in parallel

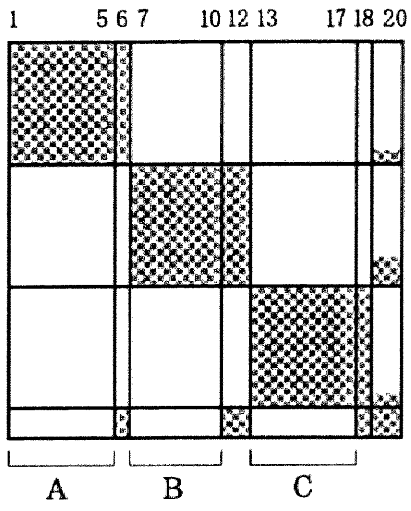


Figure 5.6: Block structure of nonzero elements.

using 3 processors.

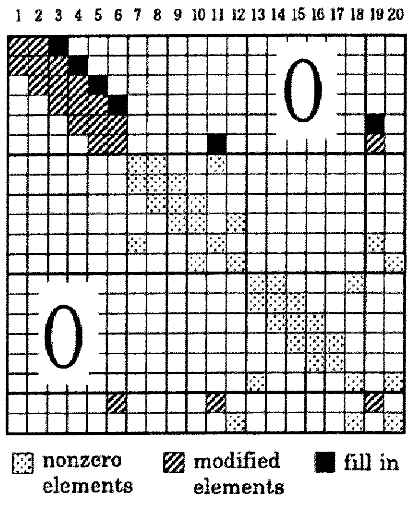


Figure 5.7: Nonzero structure after elimination by the 6th column.

In Fig. 5.8, we show the nonzero structure of the second block in B during elimination. Here, we consider the situation where there are 12 columns in the set B and the elimination by the 5th column in the set have been completed. When we apply Gaussian elimination with partial pivoting to a tridiagonal matrix, it can be easily seen that fill-ins appear at positions two elements above the diagonal. In our method, in addition to these, we also have fill-ins in the second last rows and columns in the block. As a result, the number of elements involved in the elimination in each step increases from 6 to 12 and the number of pivot candidates increases from two to three. This almost doubles the number of arithmetic operations and it is the price we have to pay for parallelization.

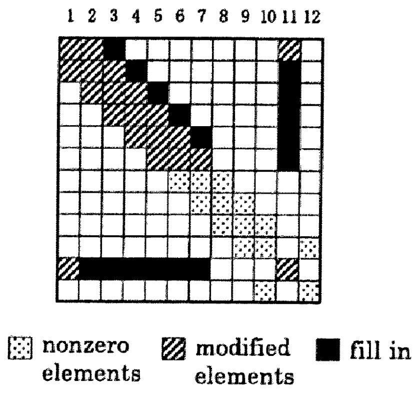


Figure 5.8: Nonzero structure of block B during elimination.

So far, we have described our algorithm for the case of $P = 3$. However, it can be easily generalized to use any number of processors. Our algorithm needs only one inter-processor synchronization, which occurs when all the "purely inner" columns allocated to each processor have been eliminated. The remaining columns, which correspond to the boundary vertices and vertices adjacent to them, needs to be eliminated sequentially, but the number of such columns is $3(P-1)$ and is independent of the matrix size N.

5.4 Numerical Results

We implemented our method on the Hitachi SR8000/F1, a shared memory parallel machine with 8 processors [95], and compared its performance and numerical accuracy with that of the conventional methods.

5.4.1 Parallel performance

To evaluate the parallel performance, we used random tridiagonal matrices whose elements were extracted from uniform random numbers in [0,1] (matrices of type (a)). The matrix size N was varied from 500 to 8,000 and the number of processors P was varied from 1 to 8. We used a sequential tridiagonal solver based on Gaussian elimination with partial pivoting when $P = 1$ and used our method when $P \geq 2$.

The execution times for the LU decomposition part are shown in Table 5.1 and Fig. 5.9. As can be seen from the table, our method achieves speedup of 5.5 times compared with the sequential method when $N = 8000$ and $P = 8$. It is also faster than the sequential method when $P = 2$ or $P = 4$.

Fig. 5.10 shows the details of the execution time for the case of $P = 8$. The white area and the shaded area denote the execution time for the parallel part (elimination of the purely inner columns) and the sequential part (elimination of the remaining columns), respectively. We can see from the graph that the execution time for the latter is almost constant and its percentage decreases as N increases. This is in consistent with the observation we made at the end of subsection 5.3.2 and means that the parallel efficiency of our method increases with N.

We also measured the execution times for matrices of type (b) and (c) which we will define in the next subsection. But here we omit the results because they were almost the same as those for matrices of type (a).

Table 5.1: Execution time of the LU decomposition.

Matrix size N	Sequential (1PU)	Ours (2PU)	Ours (4PU)	Ours (8PU)	Speedup (8PU)
500	3.26E-4	2.04E-4	1.85E-4	2.58E-4	1.26
1000	6.24E-4	3.49E-4	2.66E-4	3.05E-4	2.04
2000	1.21E-3	6.63E-4	4.32E-4	3.84E-4	3.15
4000	2.44E-3	1.32E-3	7.42E-4	5.48E-4	4.45
8000	4.79E-3	2.59E-3	1.38E-3	8.75E-4	5.47

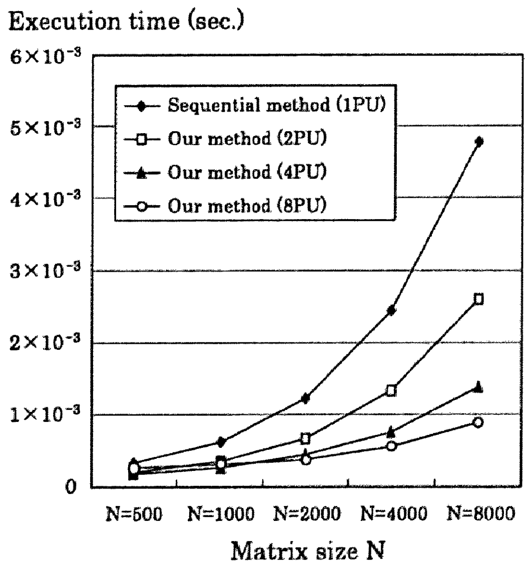


Figure 5.9: Execution time of the LU decomposition.

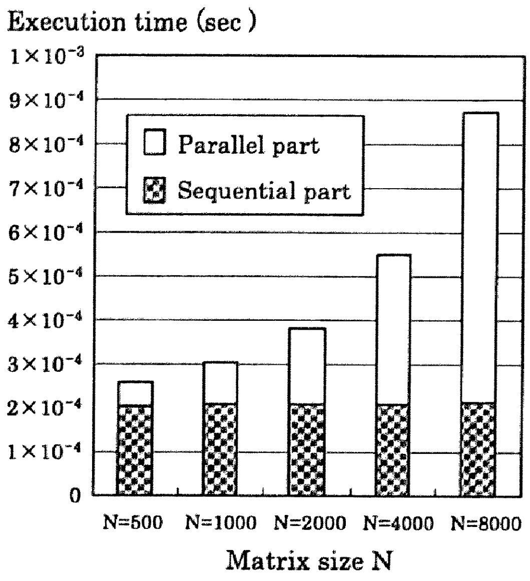


Figure 5.10: Details of the execution time.

5.4.2 Numerical accuracy

Next we compared the accuracy of our method with that of the sequential tridiagonal solver with partial pivoting and that of the dissection method without pivoting. The problems we used are (a) the random matrices which we used in the previous subsection, (b) random matrices which are the same as (a), except that the diagonal elements are multiplied by 10^{-4}, and (c) matrices obtained by tridiagonalizing the Frank matrices $A_{ij} = \min(i, j)$ and subtracting their smallest eigenvalue from the diagonal elements. Matrices of type (c) arise in the computation of eigenvectors using the inverse iteration method. The matrix size N was varied from 500 to 8000 as in the previous subsection, and the accuracy was measured in terms of the residual $\| \mathbf{Tx} - \mathbf{b} \|_\infty$.

The residual for the matrices (a), (b) and (c) are shown in Figs. 5.11, 5.12 and 5.13, respectively. It can be seen from the figures that our method achieves higher accuracy than the dissection method in all cases except for one, which is the $N = 2000$ case for matrix (c). The accuracy of our method is up to two orders of magnitude higher than the dissection method for matrices of type (b). This suggests that pivoting is indispensable for matrices which do not have diagonal dominance. When compared with the sequential Gaussian elimination with partial pivoting, our method attains much the same accuracy. The results shown in figures 5.11 and 5.12 are for a specific seed of the random number generator, but the difference of the accuracy of the three methods showed almost the same tendency for other values of the seed. In Fig. 5.14, we show how the residual changes when the seed of the random number generator is changed for the case of $N = 2000$ and matrix type (b). As can be seen from the graph, the accuracy of our method is almost the same as that of the sequential method and about two orders of magnitude better than that of the dissection method. This agrees with the results shown in Fig. 5.12.

From these numerical results, we can say that our method is a good choice when one wants to solve general unsymmetric tridiagonal matrices on parallel computers efficiently and accurately.

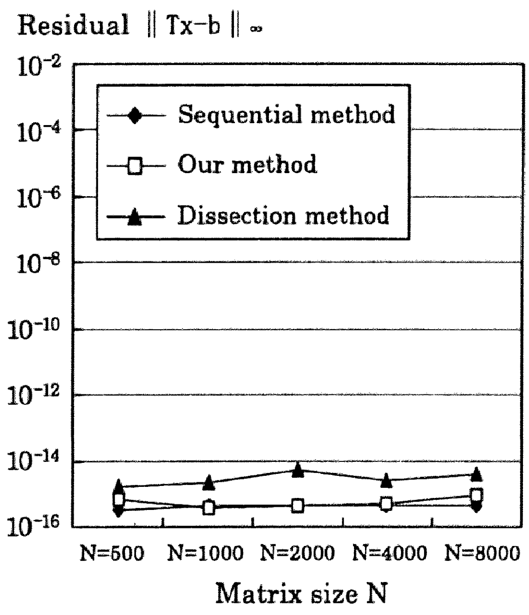


Figure 5.11: Residual of the three methods for random matrices.

5.5 Conclusion

In this chapter, we proposed a new parallel direct solver for unsymmetric tridiagonal matrices that can incorporate partial pivoting. We implemented our algorithm on one node of the Hitachi SR8000/F1 and obtained speedup of 5.5 times compared with the sequential tridiagonal solver with partial pivoting when the matrix size is 8000 and the number of processor is 8. The accuracy of our method is almost the same as that of the sequential solver and is up to two orders of magnitude better than that of the parallel solver based on the dissection method without pivoting. Our future work will include implementation of this algorithm on distributed memory parallel machines and incorporation of this algorithm into real applications such as the inverse iteration method for eigenvalue computation.

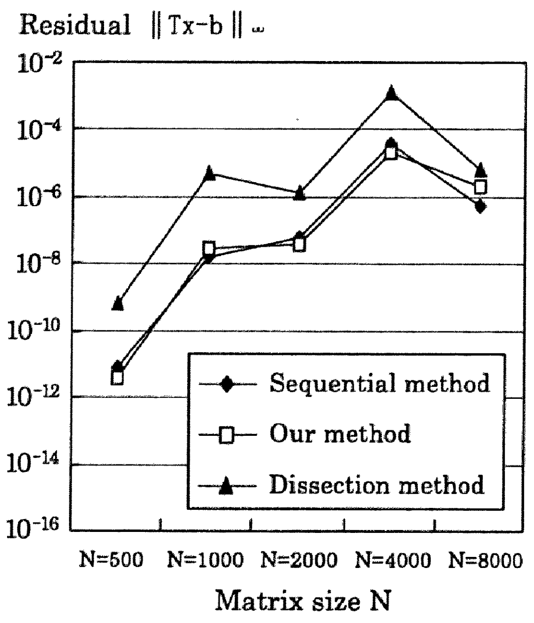


Figure 5.12: Residual of the three methods for random matrices with diagonal elements multiplied by 10^{-4}.

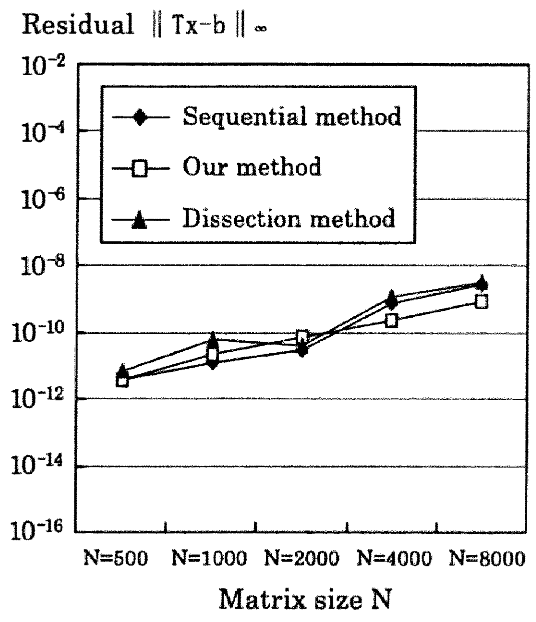


Figure 5.13: Residual of the three methods for matrices obtained by tridiagonalizing the Frank matrices.

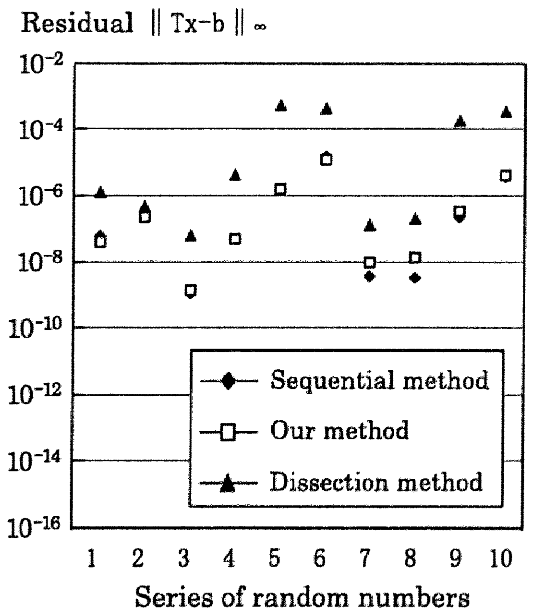


Figure 5.14: Residual of the three methods when the seed of the random number generator is changed.

Chapter 6

Computation of Eigenvalues of Real Symmetric Matrices on Processors with Hierarchical Memory

6.1 Introduction

In this chapter and the next, we deal with the problem of computing the eigenvalues and the corresponding eigenvectors of an N by N real symmetric or complex Hermitian matrix $\mathbf{A}$, that is, a set of a scalar e_i and a vector $\mathbf{x}_i$ that satisfy

$$\mathbf{A}\mathbf{x}_i = e_i\mathbf{x}_i. \tag{6.1}$$

It is well known that $\mathbf{A}$ has N eigenvalues including multiplicity and we are interested in computing all or part of the eigenvalues and, in some cases, the corresponding eigenvectors. This is one of the most basic linear algebra calculations and has wide applications to scientific computing such as structural analysis and electronic structure calculation [103].

The standard procedure for this problem consists of the following four steps [46][76][100] [101]:

1. Reduction of $\mathbf{A}$ to a real symmetric tridiagonal matrix $\mathbf{T}$ by Householder transformations.

2. Computation of the eigenvalues $\{e_1, e_2, \ldots, e_N\}$ of $\mathbf{T}$.

3. Computation of the eigenvectors $\{\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_N\}$ of $\mathbf{T}$.

4. Back-transformation of these eigenvectors into those of the original matrix, $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$.

The steps are shown in Fig. 6.1 in more detail. In this chapter, we deal with the computation of eigenvalues (steps 1 and 2) on high performance architectures. Computation of eigenvectors will be covered in the next chapter.

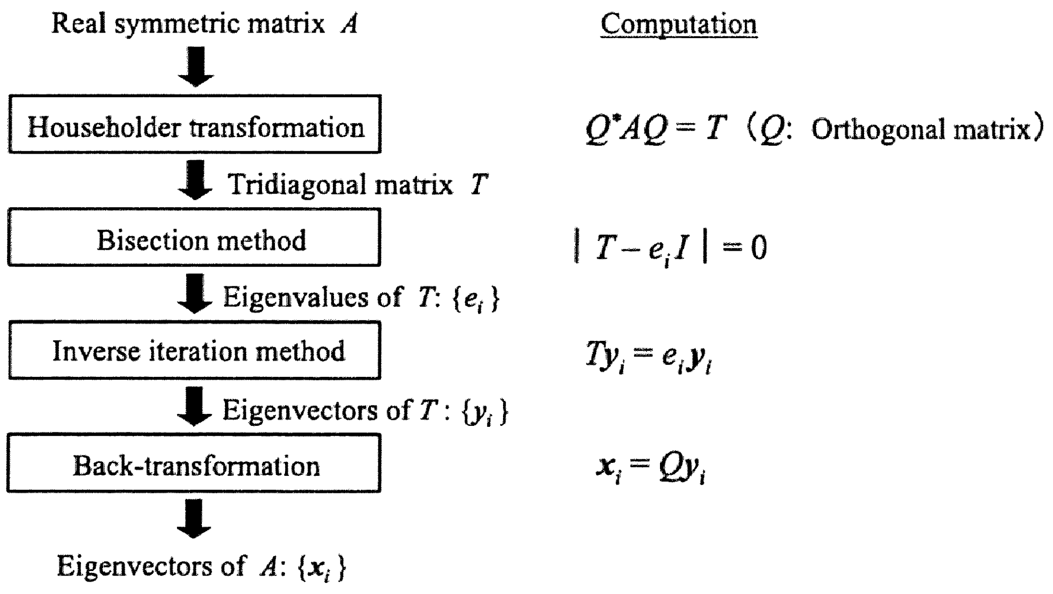
Of the two steps for computing the eigenvalues, we focus on the tri-diagonalization step. This is because it requires $O(N^3)$ computational work and dominates the computing time in large scale problems. On the other hand, computation of the eigenvalues of tri-diagonal matrix $\mathbf{T}$ by, for example, the bisection method requires only $O(N^2)$ computation [46][76].

Tri-diagonalization by the Householder transformations has inherently large parallelism of $O(N^2)$ and its efficient implementations on parallel computers have been well studied. In fact, there are algorithms for shared-memory parallel computers [84] and distributed-memory parallel computers [22][23][36][59][60][63][70].

Optimizing the single-processor performance of this algorithm on processors with hierarchical memory is more difficult, however, because it consists mainly of matrix-vector multiplications and rank-2 updates of a matrix [46][76], both of which are BLAS 2 operations that provide only small opportunity for reusing the matrix data. To solve the problem, Dongarra and van de Geijn [36] proposed a blocked algorithm for tri-diagonalization. By deferring the application of the Householder transformations and applying L transformations at once, they show that the rank-2 update operation, which occupies half of the computational work, can be replaced with rank-$2L$ update, a BLAS 3 operation. However, in their algorithm, the data reuse rate in the other half of the algorithm remains low. Bishof et al. [16][17], on the other hand, devised a two-step algorithm for tri-diagonalization which first transforms the matrix into a band matrix of half bandwidth L and then reduces it into a tridiagonal matrix. Their algorithm has the advantage that the former step can be done using entirely BLAS 3 operations, while the latter step requires only $O(N^2L)$ work. Nevertheless, it still has the difficulty that if one chooses L so that the cache memory can be fully utilized in the former step, L tends to become large and the computational work of the latter step becomes not negligible.

In this chapter, we improve Bishof's algorithm by combining it with Dongarra el al.'s blocking technique. By deferring the application of transformations in the former step of Bishof's algorithm and applying L' transformations at once, we can effectively increase the size of matrix multiplications and use the cache efficiently without increasing the work in the latter step. In addition, we employ loop merging techniques to further enhance data reuse.

The rest of this chapter is organized as follows. In section 6.2, we describe the basic algorithm of Householder tri-diagonalization and its blocked version due to Dongarra et al. In section 6.3, we explain the two-step algorithm of Bishof et al. Our improvement on this algorithm is introduced in section 6.4 and the results of numerical experiments are given in 6.5. Finally we conclude with some prospect of future work.



Real symmetric matrix A
Computation
Householder transformation
$Q^{*}AQ = T$ (Q: Orthogonal matrix)
Tridiagonal matrix T
Bisection method
$|T - e_i I| = 0$
Eigenvalues of T: $\{e_i\}$
Inverse iteration method
$Ty_i = e_i y_i$
Eigenvectors of T: $\{y_i\}$
Back-transformation
$x_i = Qy_i$
Eigenvectors of A: $\{x_i\}$

Figure 6.1: Standard procedure for computing the eigenvalues and eigenvectors of a real symmetric matrix.

6.2 The conventional Householder tri-diagonalization and its blocked variant

6.2.1 Basic algorithm for Householder tri-diagonalization

The basic algorithm for Householder tri-diagonalization is shown in Algorithm 6.1. The computation consists of $N-2$ stages. Let's denote the matrix at the k-th stage ($1 \leq k \leq N-2$) by $\mathbf{A}^{(k)}$, the column vector which consists of the $(k+1, k)$-th through (N, k)-th elements of $\mathbf{A}^{(k)}$ by $\mathbf{d}^{(k)}$ and the lower-right $N-k$ by $N-k$ submatrix of $\mathbf{A}^{(k)}$ by $\mathbf{C}^{(k)}$, as shown in Fig. 6.2. At the k-th stage, we first generate the *reflector vector* $\mathbf{u}^{(k)}$ from $\mathbf{d}^{(k)}$ and, by multiplying it with $\mathbf{C}^{(k)}$, obtain vectors $\mathbf{p}^{(k)}$ and $\mathbf{q}^{(k)}$. $\mathbf{u}^{(k)}$ and $\mathbf{q}^{(k)}$ are called the *pivot column* and *pivot row*, respectively. Finally, we update $\mathbf{C}^{(k)}$ using the pivot column and the pivot row. This operation is called *rank-2 modification* of the matrix. See [46][76] for the proof that this algorithm actually tri-diagonalizes the input matrix $\mathbf{A}$.

```
[Algorithm 6.1 Tri-diagonalization by the Householder transformations]
do $k = 1, N-2$
   [Generation of the reflector vector $\mathbf{u}^{(k)}$]
   $\sigma^{(k)} = \sqrt{\mathbf{d}^{(k)t}\mathbf{d}^{(k)}}$
   $\mathbf{u}^{(k)} = (d_1^{(k)} - \mathrm{sgn}(d_1^{(k)})\sigma^{(k)}, d_2^{(k)}, \ldots, d_{N-k}^{(k)})$
   $\alpha^{(k)} = 2/ \parallel \mathbf{u}^{(k)} \parallel_2$
   [Matrix-vector multiplication]
   $\mathbf{p}^{(k)} = \alpha^{(k)}\mathbf{C}^{(k)}\mathbf{u}^{(k)}$
   $\beta^{(k)} = \alpha^{(k)}\mathbf{u}^{(k)t}\mathbf{p}^{(k)}/2$
   $\mathbf{q}^{(k)} = \mathbf{p}^{(k)} - \beta^{(k)}\mathbf{u}^{(k)}$
   [Rank-2 update of the matrix]
   $\mathbf{C}^{(k)} := \mathbf{C}^{(k)} - \mathbf{u}^{(k)}\mathbf{q}^{(k)t} - \mathbf{q}^{(k)}\mathbf{u}^{(k)t}$
end do
```

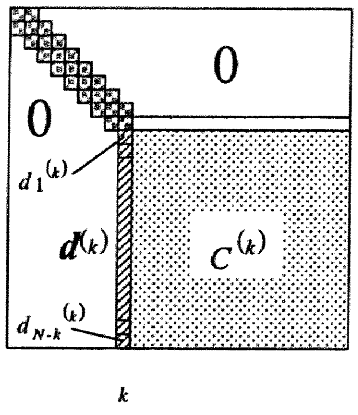


Figure 6.2: The k-th stage of Householder tri-diagonalization.

6.2.2 Blocked version of the Householder tri-diagonalization

In Algorithm 6.1, each of the matrix-vector multiplication and the rank-2 update of the matrix require $2/3N^3$ floating point operations and these two constitute most of the total computational work. However, both of these are BLAS 2 operations that provide only small opportunity for data reuse. As a result, it cannot attain high performance on processors with hierarchical memory.

To overcome this difficulty, Dongarra et al. proposed a blocked algorithm for the Householder tri-diagonalization [36]. This is shown as Algorithm 6.2. Here, $(\mathbf{X})_i$ denotes the i-th column vector of matrix $\mathbf{X}$. In the k-th stage of this algorithm, after computing the pivot column $\mathbf{u}^{(k)}$ and pivot row $\mathbf{q}^{(k)}$, we defer the rank-2 update and instead store $\mathbf{u}^{(k)}$ and $\mathbf{q}^{(k)}$ as a column of matrix $\mathbf{U}^{(k)}$ and $\mathbf{Q}^{(k)}$, respectively. After L pivot columns and rows have been generated, we perform L updates at once. Hence the rank-2 update is replaced by rank-$2L$ update or matrix multiplication, as illustrated in Fig. 6.3, and the ratio of data reuse of $\mathbf{C}^{(k)}$ is increased by a factor of L. This algorithm is adopted in the LAPACK routine *dsytrd* [8].

```
[Algorithm 6.2 Tri-diagonalization by the blocked Householder transformations]
do $K = 1,\ N/L$
  $\mathbf{U}^{((K-1)*L)} = \phi,\ \mathbf{Q}^{((K-1)*L)} = \phi$
  do $k = (K-1)*L+1,\ K*L$
    [Partial Householder transformation]
    $\mathbf{d}^{(k)} := \mathbf{d}^{(k)} - \mathbf{U}^{(k-1)}(\mathbf{Q}^{(k-1)t})_{k-(K-1)*L}$
                 $- \mathbf{Q}^{(k-1)}(\mathbf{U}^{(k-1)t})_{k-(K-1)*L}$
    [Generation of the reflector vector $\mathbf{u}^{(k)}$]
    $\sigma^{(k)} = \sqrt{\mathbf{d}^{(k)t}\mathbf{d}^{(k)}}$
    $\mathbf{u}^{(k)} = (d_1^{(k)} - \mathrm{sgn}(d_1^{(k)})\sigma^{(k)}, d_2^{(k)}, \ldots, d_{N-k}^{(k)})$
    $\alpha^{(k)} = 2/ \parallel \mathbf{u}^{(k)} \parallel_2$
    [Matrix-vector multiplication]
    $\mathbf{p}^{(k)} = \alpha^{(k)}(\mathbf{C}^{(k)} - \mathbf{U}^{(k-1)}\mathbf{Q}^{(k-1)t} - \mathbf{Q}^{(k-1)}\mathbf{U}^{(k-1)t}\mathbf{u}^{(k)}$
    $\beta^{(k)} = \alpha^{(k)}\mathbf{u}^{(k)t}\mathbf{p}^{(k)}/2$
    $\mathbf{q}^{(k)} = \mathbf{p}^{(k)} - \beta^{(k)}\mathbf{u}^{(k)}$
    $\mathbf{U}^{(k)} = [\mathbf{U}^{(k-1)}|\mathbf{u}^{(k)}],\ \mathbf{Q}^{(k)} = [\mathbf{Q}^{(k-1)}|\mathbf{q}^{(k)}]$
  end do
  [Rank-$2L$ update of the matrix]
  $\mathbf{C}^{(K*L)} := \mathbf{C}^{((K-1)*L)} - \mathbf{U}^{(K*L)}\mathbf{Q}^{(K*L)t} - \mathbf{Q}^{(K*L)}\mathbf{U}^{(K*L)t}$
end do
```

Although this algorithm can replace half of the BLAS 2 operations with BLAS 3, the other half, the matrix-vector multiplication part, still has to be done with BLAS 2. The algorithm of Bishof et al. [16][17] which we will explain in the next section was proposed to remedy this problem.

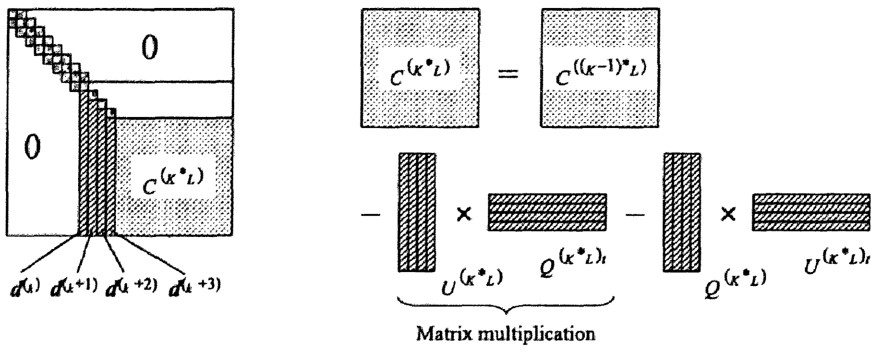


Figure 6.3: Blocked Householder tri-diagonalization with block size L=4.

6.3 The two-step algorithm for tri-diagonalization

6.3.1 The basic idea

Bishof's algorithm of transforms the input matrix **A** to a tridiagonal matrix **T** in two steps, namely, reduction of **A** to a band matrix **B** of half bandwidth L and reduction of **B** to a tridiagonal matrix **T**, as illustrated in Fig. 6.4.

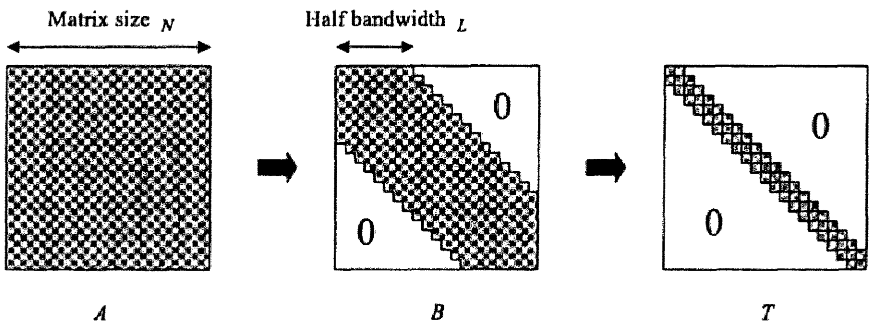


Figure 6.4: The two-step algorithm for tri-diagonalization.

Of the two steps, the former step can be done entirely with BLAS 3 operations and requires about $(4/3)N^3$ floating point operations [16][17], while the latter can be done with about $6N^2L$ operations. Accordingly, when $N \gg L$, the algorithm needs almost the same number of operations as the basic algorithm (Algorithm 6.1) and most of them are done with BLAS 3 routines.

6.3.2 Reduction of the input matrix to a band matrix

We show the algorithm for reducing the input matrix to a band matrix as Algorithm 6.3. Here, it is assumed for simplicity that N is divisible by the half bandwidth L. To describe the algorithm, we divide matrices into blocks of size $L \times L$ and use these blocks and rows/columns of the blocks as the basic components.

The algorithm consists of $N/L - 1$ stages. It is very similar to the basic Householder tri-diagonalization algorithm shown in Algorithm 6.1, except that the vectors are replaced with block vectors of width L and the scalars are replaced with $L \times L$ matrices. Specifically, vectors $\mathbf{d}^{(k)}$, $\mathbf{u}^{(k)}$, $\mathbf{p}^{(k)}$, $\mathbf{q}^{(k)}$ are replaced with block vectors $\mathbf{D}^{(K)}$, $\mathbf{U}^{(K)}$, $\mathbf{P}^{(K)}$, $\mathbf{Q}^{(K)}$ of width L, while scalars $\alpha^{(k)}$ and $\beta^{(k)}$ are replaced with $L \times L$ matrices $\alpha^{(K)}$ and $\beta^{(K)}$.

At the K-th stage, we focus on the block vector $\mathbf{D}^{(K)}$ which consists of the $(K+1, K)$-th to $(N/L, K)$-th block elements of $\mathbf{A}^{(K)}$. In a spirit similar to that of the basic Householder tri-diagonalization, we try to find a block Householder transformation $\mathbf{I} - \mathbf{U}^{(K)}\alpha^{(K)}\mathbf{U}^{(K)}$ that transforms $\mathbf{D}^{(K)}$ into a block vector whose first block is an upper triangular matrix and the following blocks are zero (Fig. 6.5). We can obtain such a block Householder transformation easily by computing the QR decomposition of $\mathbf{D}^{(K)}$ using Householder transformations [46] and combining these transformations using the *WY representation* [15][46][87]. Next, by multiplying it with $\mathbf{C}^{(K)}$, we obtain block vectors $\mathbf{P}^{(K)}$ and $\mathbf{Q}^{(K)}$. $\mathbf{U}^{(K)}$ and $\mathbf{Q}^{(K)}$ are called the *block pivot column* and *block pivot row* at the K-th stage, respectively. Finally, we update $\mathbf{C}^{(K)}$ using the block pivot column and the block pivot row. This is a rank-$2L$ update of $\mathbf{C}^{(K)}$. This completes the transformation of the leading $(K+1)L$ by $(K+1)L$ submatrix of $\mathbf{A}$ to a band matrix.

```
[Algorithm 6.3 Reduction of the input matrix to a band matrix]
do $K = 1, N/L - 1$
  [Generation of a block Householder transformation]
  Compute a block Householder transformation
  $\mathbf{I} - \mathbf{U}^{(K)}\alpha^{(K)}\mathbf{U}^{(K)}$ that transforms the block vector $\mathbf{D}^{(K)}$
  into a block vector whose first block is an upper triangular
  matrix and the following blocks are zero.
  [Matrix-block vector multiplication]
  $\mathbf{P}^{(K)} = \mathbf{C}^{(K)}\mathbf{U}^{(K)}\alpha^{(K)}$
  $\beta^{(K)} = \alpha^{(K)}\mathbf{U}^{(K)t}\mathbf{P}^{(K)}/2$
  $\mathbf{Q}^{(K)} = \mathbf{P}^{(K)} - \mathbf{U}^{(K)}\beta^{(K)}$
  [Rank-2L update of the matrix]
  $\mathbf{C}^{(K)} := \mathbf{C}^{(K)} - \mathbf{U}^{(K)}\mathbf{Q}^{(K)t} - \mathbf{Q}^{(K)}\mathbf{U}^{(K)t}$
end do
```

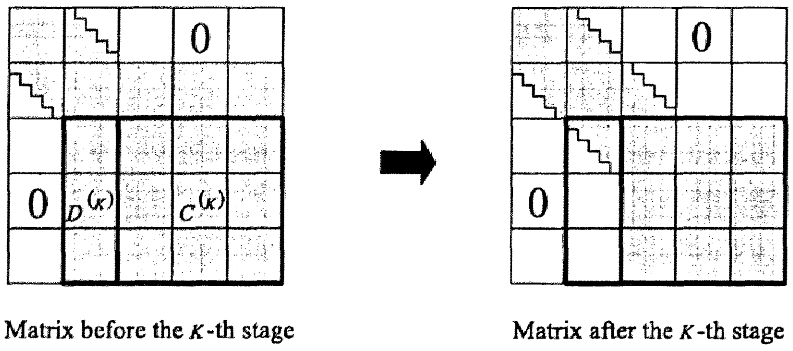


Figure 6.5: Reduction of the matrix at the second ($K = 2$) stage.

In this algorithm, both the matrix-block vector multiplication and the rank-$2L$ update account for nearly half of the total computational work. As is clear from the algorithm, both of these operation are matrix multiplications and their rate of data reuse increases with the half bandwidth L. It is therefore optimal for maximizing the single-processor performance in this step to make L as large as possible under the condition that all the blocks used in the matrix multiplications can be stored in the cache.

6.3.3 Reduction of the band matrix to a tridiagonal matrix

In the second step of the Bishof's algorithm, the band matrix obtained in the first step is further reduced to a tridiagonal matrix. This can be achieved using the algorithm explained in [68] or [101] and requires about $6N^2L$ floating point operations. Note that the computational work of this step increases proportionally with L.

6.4 Improvement on the two-step algorithm

6.4.1 Limitations of the two-step algorithm

As we have stated in subsection 6.3.2, we have to use moderately large L to achieve high single-processor performance in the first. On the other hand, the computational work in the second step increases with L. To be concrete, let's consider a situation of tri-diagonalizing a matrix of $N = 7200$ on a processor with 64KB of cache memory. If we choose $L = 12$, the computational work in the second step is kept to less than 3% of the total work, but we cannot fully exploit the cache in the first step because matrix multiplication $\mathbf{C} = \mathbf{AB}$ of size $L \times L$ require only 3.5KB of memory. In contrast, if we choose $L = 48$, we can fully utilize the cache and increase the rate of data reuse by four times, but the computational cost of the second step increases to about 10%. As can be seen from this example, it is sometimes difficult to choose L properly, especially when N is not large enough.

To solve this problem, we propose two techniques to enhance data reuse in the first step of Bishof's algorithm without increasing L. One of the techniques is for the matrix-block vector multiplication part and the other is for the rank-$2L$ update part. These two techniques can be used together to improve the overall performance of Bishof's algorithm.

6.4.2 Improvement of the matrix-block vector multiplication

We consider the computation of $\mathrm{C}^{(K)}\mathrm{U}^{(K)}$ in Algorithm 6.3. Because the original matrix $\mathbf{A}$ is symmetric and the rank-$2L$ updates keep the symmetry, we know that $\mathbf{C}^{(K)}$ is symmetric for every K. Let's assume that the multiplication is performed block-by-block and focus on two blocks $\mathbf{F}_1$ and $\mathbf{F}_2$ of $\mathrm{C}^{(K)}$ which are at mirror positions of each other with respect to the diagonal (Fig. 6.6). As can be seen from the Figure, $\mathbf{F}_1$ is multiplied with $\mathbf{G}_1$ and added to $\mathbf{H}_1$, while $\mathbf{F}_2$ is multiplied with $\mathbf{G}_2$ and added to $\mathbf{H}_2$. In the standard implementation, these two operations are done in separate loops. However, if we merge them, we can eliminate the load of $\mathbf{F}_2$ by exploiting the fact that $\mathbf{F}_2 = \mathbf{F}_1^t$. This increases the data reuse rate for $\mathrm{C}^{(K)}$ twice.

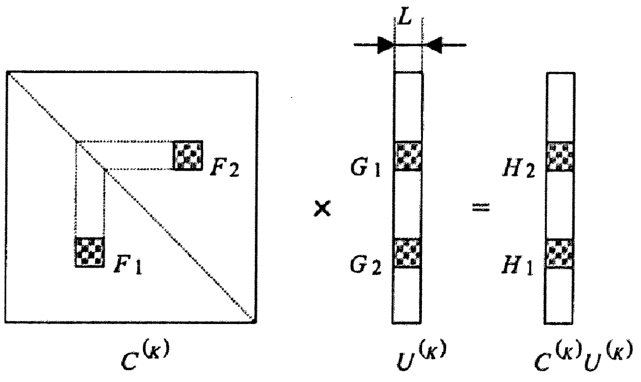


Figure 6.6: Exploiting the symmetry of $\mathrm{C}^{(K)}$.

6.4.3 Improvement of the rank-$2L$ update

For the rank-$2L$ update part, we can apply the idea of the blocked Householder transformation introduced in subsection 6.2.2. The algorithm based on this idea is shown as Algorithm 6.4. Here, L' is some integer and Q^{K} and U^{K} are matrices whose columns are block vectors. $(\mathsf{Q}^{\mathsf{K}})_i$ and $(\mathsf{U}^{\mathsf{K}})_i$ denote the i-th column (block vector) of Q^{K} and U^{K}, respectively.

In this algorithm, after computing the block Householder transformation $\mathbf{I}-\mathbf{U}^{(K)}\alpha^{(K)}\mathbf{U}^{(K)}$ and generating the block pivot rows and columns, we defer the rank-$2L$ update and instead store the block pivot rows and columns as a block column of matrices Q^{K} and U^{K}. After L' block pivot rows/columns have been generated, we apply these L' rank-L updates as a rank-$2LL'$ update on matrix $C^{(K)}$. This modification increases the data reuse rate of $C^{(K)}$ in this part by L' times without increasing the computational work in the second step of Bishof's algorithm.

```
[Algorithm 6.4 Improved version of Bishof's algorithm]
do $\mathsf{K} = 1,\ N/(LL') - 1$
  $\mathsf{U}^{((\mathsf{K}-1)*LL')} = \phi,\ \mathsf{Q}^{((\mathsf{K}-1)*LL')} = \phi$
  do $K = (\mathsf{K}-1)*L'+1,\ \mathsf{K}*L'$
    **[Partial Householder transformation]**
    $\mathbf{D}^{(K)} := \mathbf{D}^{(K)} - \mathsf{U}^{(K-1)}(\mathsf{Q}^{(K-1)t})_{K-(\mathsf{K}-1)*L'}$
                $- \mathsf{Q}^{(K-1)}(\mathsf{U}^{(K-1)t})_{K-(\mathsf{K}-1)*L'}$
    **[Generation of a block Householder transformation]**
    Compute a block Householder transformation
    $\mathbf{I} - \mathbf{U}^{(K)}\alpha^{(K)}\mathbf{U}^{(K)}$ that transforms the block vector $\mathbf{D}^{(K)}$
    into a block vector whose first block is an upper triangular
    matrix and the following blocks are zero.
    **[Matrix-block vector multiplication]**
    $\mathbf{P}^{(K)} = (\mathbf{C}^{(K)} - \mathsf{U}^{(K-1)}\mathsf{Q}^{(K-1)t} - \mathsf{Q}^{(K-1)}\mathsf{U}^{(K-1)t})\mathbf{U}^{(K)}\alpha^{(K)}$
    $\beta^{(K)} = \alpha^{(K)}\mathbf{U}^{(K)t}\mathbf{P}^{(K)}/2$
    $\mathbf{Q}^{(K)} = \mathbf{P}^{(K)} - \beta^{(K)}\mathbf{U}^{(K)}$
    $\mathsf{U}^{(K)} = [\mathsf{U}^{(K-1)}|\mathbf{U}^{(K)}],\ \mathsf{Q}^{(K)} = [\mathsf{Q}^{(K-1)}|\mathbf{Q}^{(K)}]$
  end do
  **[Rank-$2LL'$ update of the matrix]**
  $\mathbf{C}^{(K*L')} := \mathbf{C}^{((K-1)*L')} - \mathsf{U}^{(\mathsf{K}*L')}\mathsf{Q}^{(\mathsf{K}*L')t} - \mathsf{Q}^{(\mathsf{K}*L')}\mathsf{U}^{(\mathsf{K}*L')t}$
end do
```

6.5 Numerical results

6.5.1 Performance on a processor with hierarchical memory

We implemented the following three methods on the Hitachi EP8000 and evaluated their performance.

- Dongarra et al.'s blocked Householder tri-diagonalization described in subsection 6.2.2.
- Bishof et al.'s two-step tri-diagonalization algorithm described in section 6.3
- Our improvement on Bishof et al.'s algorithm proposed in section 6.4.

The EP8000 uses IBM Power 4 as its processor and have a peak performance of 5.2GFLOPS. As a compiler, we used Hitachi FORTRAN90 01-01 version.

First, we show the performance comparison of Dongarra's algorithm and Bishof's algorithm in Table 6.1 and Fig. 6.7. In the latter algorithm, we chose the maximum value of the half bandwidth as $L = 48$ so that all the data used in the algorithm can be stored in the first cache. From these results, we can make following observations:

1. Dongarra's algorithm, which uses BLAS 3 for only half of the total computational work and uses BLAS2 for the rest, can attain at most 20% of the peak performance.
2. Bishof's algorithm can attain 1.6 to 1.9 times the performance of Dongarra's algorithm.
3. The performance of Bishof's algorithm is highest when $L = 12$ and decreases as L increases.

The last point is due to the fact that the computational work in the second step increases proportionally with L.

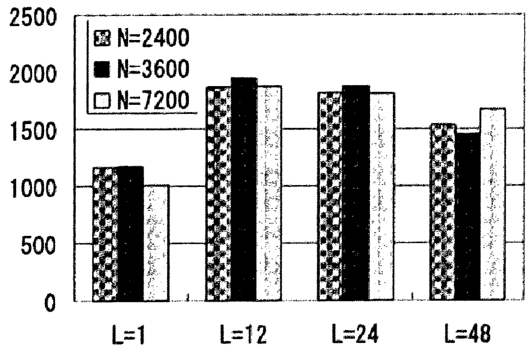


Figure 6.7: Performance comparison of Dongarra's and Bishof's algorithm.

Next, we investigated the effect of our improvements proposed in section 6.4. The results are given in Table 6.2. Here, we used $L' = 4$. The results show that our improvements increase the performance of Bishof's algorithm by 10 to 15% and achieve 1.7

Table 6.1: Performance comparison of Dongarra's and Bishof's algorithm.

Algorithm	L	N	Execution time (sec.)	Performance (MFLOPS)	% of peak	Relative performance
Dongarra et al.	1	2400	15.87	1162	22.3	1.0
		3600	53.35	1166	22.4	1.0
		7200	492.18	1011	19.4	1.0
Bishof et al.	12	2400	9.91	1860	35.8	1.60
		3600	32.02	1943	37.4	1.67
		7200	265.33	1875	36.1	1.86
	24	2400	10.17	1813	34.9	1.56
		3600	33.24	1872	36.0	1.61
		7200	275.75	1808	34.8	1.80
	48	2400	12.02	1534	29.5	1.32
		3600	42.77	1455	28.0	1.25
		7200	298.55	1667	32.1	1.65

to 2.1 times the performance compared with Dongarra's algorithm, or about 40% of the peak performance. Thus we can say that these improvements are effective for enhancing the single-processor performance of Householder tri-diagonalization on processors with hierarchical memory.

Table 6.2: Performance comparison of the original and improved version of Bishof's algorithm.

Algorithm	N	Execution time (sec.)	Performance (MFLOPS)	% of peak	Relative performance
Original	2400	9.91	1860	35.8	1.0
$L = 12$	3600	32.02	1943	37.4	1.0
	7200	265.33	1875	36.1	1.0
Improved	2400	9.01	2047	39.4	1.10
$L = 12$	3600	28.37	2192	42.2	1.13
$L' = 4$	7200	230.80	2155	41.5	1.15

6.5.2 Numerical accuracy

To study the numerical accuracy of the three algorithms, we used a problem of computing all eigenvalues of a symmetric matrix and compared the accuracy of eigenvalues obtained by Dongarra's and Bishof's algorithm. The accuracy of our algorithm is considered almost the same as that of Bishof's, because it is known that the blocking of Householder transformations, which we used in subsection 6.4.3, does not affect numerical accuracy [46]. Of course, the loop merging technique introduced in subsection 6.4.2 does not change numerical properties.

As test matrices, we used Frank matrices of N=2400, 3600 and 7200 which are defined by $A_{ij} = \min(i, j)$. This type of matrices are frequently used to evaluate the accuracy of eigensolvers [63][71][112] because their eigenvalues can be computed analytically. As an indicator of the accuracy, we used the sum of relative errors of computed eigenvalues:

$$\sum_{i=1}^{N} \left| \frac{e_i - e_i'}{e_i} \right|, \tag{6.2}$$

where e_i and e_i' are the i-th exact and computed eigenvalues, respectively.

The results are given in Table 6.3. They show that the accuracy of these two algorithms are almost the same and dividing the tri-diagonalization into two steps has little impact on numerical accuracy.

Table 6.3: Accuracy comparison of Dongarra's and Bishof's algorithm.

N	Dongarra's algorithm	Bishof's algorithm
2400	0.399×10^{-6}	0.440×10^{-6}
3600	0.199×10^{-5}	0.217×10^{-5}
7200	0.193×10^{-4}	0.116×10^{-4}

6.6 Conclusion

In this chapter, we dealt with the problem of computing the eigenvalues of real symmetric matrices. We studied three algorithms developed for processors with hierarchical memory, namely, Dongarra et al.'s blocked Householder tri-diagonalization, Bishof et al.'s two-step algorithm and our improvement on it, and evaluated their performance on the Power 4 processor. The results we have obtained can be summarized as follows:

1. Dongarra's algorithm can attain at most 20% of the peak performance on the Power 4, because half of the computation has to be done with BLAS 2.

2. Bishof's algorithm, which consists entirely of BLAS 3 routines, can attain 1.6 to 1.9 times the performance of Dongarra's algorithm.

3. Our improvements on Bishof's algorithm can increase the performance by 10 to 15% and achieves 40% of the peak performance.

4. Numerical experiments on the Frank matrices show that Bishof's algorithm can attain the same level of accuracy as Dongarra's algorithm.

From these results, we can say that Bishof's algorithm and our improvements on it are a good alternative to the widely used Dongarra's algorithm when computing eigenvalues on processors with hierarchical memory.

When it comes to computing the eigenvectors, however, it is known that Bishof's algorithm requires twice the computational work of Dongarra's algorithm for each eigenvector. This is because in the former case, the back-transformation of the eigenvectors of the tridiagonal matrix also consists of two steps and each step requires the same amount of work as that of the back-transformation in Dongarra's algorithm. As a result, Bishof's algorithm loses competitiveness when a large number (e.g., more than $N/2$) eigenvectors are necessary. It remains our future work to develop a more efficient way for computing the eigenvectors in this case. Also, a parallel version of Bishof's algorithm for distributed-memory parallel machines is under development.

Chapter 7

Computation of Eigenvectors of Real Symmetric Tridiagonal Matrices on Shared-Memory Machines

7.1 Introduction

In this chapter, we study algorithms for computing the eigenvectors of a real symmetric tridiagonal matrix when the corresponding eigenvalues are given. Combined with the other three steps shown in Fig. 6.1, namely, tri-diagonalization by Householder transformations, computation of the eigenvalues of the tridiagonal matrix by the bisection method and back-transformation, this algorithm can be used to compute the eigenvalues and eigenvectors of a general real symmetric matrix.

Of these four steps, the tri-diagonalization step can easily be parallelized both on shared-memory and distributed-memory parallel machines. The reader is referred to the literature cited in section 6.1 for details. It is also easy to find the eigenvalues of the tri-diagonal matrix in parallel by, for example, using the bisection or multi-section methods. Back-transformation also poses no difficulty in parallelization, because each eigenvector can be back-transformed independently.

Calculation of the eigenvectors of the tri-diagonal matrix is more difficult to parallelize, however, because one has to ensure orthogonality of the calculated eigenvectors. Many new algorithms have been developed to address this problem, including the divide and conquer method [28][47], Dhillon's algorithm [31], and the multicolor inverse iteration method [71]. Among them, the divide and conquer method is very efficient and outperforms conventional methods such as the QL method and the inverse iteration method even on a sequential computer. But it is suitable only for the case where all the eigenvalues and eigenvectors are needed. Dhillon's method, which is an improvement over the conventional inverse iteration, obviates the need for explicit orthogonalization and still can produce orthogonal eigenvectors. This algorithm is implemented in the latest version of LAPACK (version 3.0) as a subroutine *dstegr*. But it does not always work well when the relative gaps of the eigenvalues are very small. In such cases, one has to use the subroutine *dstein*, which uses the conventional inverse iteration. The multicolor inverse iteration method reduces the number of orthogonalization to a minimum and thereby extracts parallelism in the computation of the eigenvectors. But it has the limitation that the level of available parallelism becomes quite low when the eigenvalues are clustered.

In this chapter, we propose another approach for computing the orthogonal eigenvectors of a real symmetric tri-diagonal matrix based on the idea given in [108]. Like Dhillon's method and the multicolor inverse iteration, our method is based on the conventional inverse iteration. But instead of eliminating or reducing the orthogonalization, we choose to parallelize the orthogonalization process itself. To this end, we abandon using the modified Gram-Schmidt orthogonalization procedure, which is the bottleneck in parallelizing the conventional method, and instead, choose to hold the basis of the orthogonal complementary subspace of the calculated eigenvectors explicitly and successively modify it by the Householder transformations. When implemented on shared-memory multiprocessors, our method needs only $O(N)$ interprocessor synchronization to compute all eigenvectors of an N by N matrix. Moreover, in our method, two thirds of the total arithmetic operation can be performed with the BLAS-3 (matrix-matrix multiplication) routines. It is therefore especially suited for modern SMP machines with hierarchical memory.

The paper is organized as follows: In section 2, we briefly review the conventional inverse iteration method along with the difficulty in parallelizing it. We also give some assumptions and notations. The basic idea of our new algorithm, the Householder inverse iteration method, is given in section 3. The blocked version of this algorithm, which allows the use of the BLAS-3 routines, is discussed in section 4. Results of performance evaluation on the Hitachi SR8000, a shared-memory multiprocessor system, can be found in section 5. Concluding remarks are given in the final section.

7.2 Review of the conventional inverse iteration method

7.2.1 The conventional inverse iteration method

Given an N by N real symmetric tri-diagonal matrix $\mathbf{T}$ along with approximations to its eigenvalues $\{e_i\}_{i=1}^{N}$ $(e_1 \leq e_2 \leq \ldots \leq e_N)$, we consider the problem of computing the eigenvectors $\{\mathbf{v}_i\}$ corresponding to the eigenvalues $\{e_i\}$. In the conventional inverse iteration method (IIM), we perform the iteration

$$\mathbf{v}_i^{(m)} := (\mathbf{T} - e_i'\mathbf{I})^{-1}\mathbf{v}_i^{(m-1)} \tag{7.1}$$

for each i starting from the approximate eigenvalue e_i' and some initial vector $\mathbf{v}_i^{(0)}$. It is expected that if e_i' is sufficiently close to e_i, the component of $\mathbf{v}_i^{(0)}$ which is parallel to $\mathbf{v}_i$ is amplified during the iteration and $\mathbf{v}_i^{(m)}$ converges to $\mathbf{v}_i$.

But in finite precision arithmetic, the component parallel to other eigenvectors, say $\mathbf{v}_k$, remains in the calculated vector due to numerical errors. This causes the problem that orthogonality of the eigenvectors, one of the basic properties that the exact eigenvectors of a real symmetric matrix should have, is not guaranteed sufficiently. To remedy this problem, in the conventional inverse iteration method, $\mathbf{v}_i^{(m)}$ is orthogonalized against previously calculated eigenvectors after each iteration. This is usually done with the modified Gram-Schmidt (MGS) method [101][100]. Because the magnitude of $\mathbf{v}_k$ component remaining in the calculated vector $\mathbf{v}_i$ is shown to be proportional to $(e_k - e_i)^{-1}$ according to error analysis [101][100], orthogonalization is usually done only against those eigenvectors which belong to eigenvalues close to e_i.

The algorithm of the conventional inverse iteration with orthogonalization by the MGS method is shown as Algorithm 7.1. Here, the dot denotes the inner product of two vectors, and $|| * ||_2$ denotes the L_2-norm of a vector. In the practical algorithm, additional processes are necessary to deal with degenerate or tightly clustered eigenvalues, such as changing the initial vector or displacing some of the eigenvalues slightly. But these are omitted in the shown algorithm.

In Algorithm 7.1, the innermost loop over k corresponds to the MGS orthogonalization, in which the newly calculated vector $\mathbf{v}_i^{(m)}$ is orthogonalized against the previously calculated eigenvectors $\mathbf{v}_k$ within the same group $\mathrm{G}(i)$. An example of grouping of the eigenvalues is shown in Fig. 7.1.

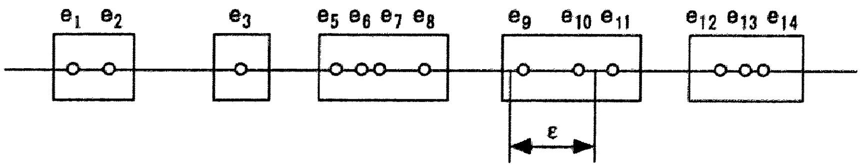


Figure 7.1: Grouping of the eigenvalues in the conventional inverse iteration method.

```
[Algorithm 7.1 Conventional inverse iteration method]
[Grouping of the eigenvalues]
Define two eigenvalues as belonging to the same group when their
distance is smaller than or equal to some criterion $\epsilon$. Let the group
to which the $i$-th eigenvalue belongs be denoted by $\mathrm{G}(i)$.
for $i$=1: $N$
  Set some initial vector $\mathbf{v}_i^{(0)}$.
  $m := 1$
  until $\mathbf{v}_i^{(m)}$ converges
    $\mathbf{v}_i^{(m)} := (\mathbf{T} - e_i'\mathbf{I})^{-1}\mathbf{v}_i^{(m-1)}$
    for all $k \in \mathrm{G}(i)$ $(k < i)$ do
      $\mathbf{v}_i^{(m)} := \mathbf{v}_i^{(m)} - (\mathbf{v}_i^{(m)t}\mathbf{v}_k)\mathbf{v}_k$
    end
    $\mathbf{v}_i^{(m)} := \mathbf{v}_i^{(m)} / \| \mathbf{v}_i^{(m)} \|_2$
  end
  $\mathbf{v}_i := \mathbf{v}_i^{(m)}$
end
```

7.2.2 Difficulty with the conventional algorithm

In the conventional IIM, the eigenvectors belonging to different groups can be calculated independently, for the orthogonalization of the calculated vectors is done only within each group. It is therefore natural in parallelizing this algorithm to exploit the group-level parallelism by allocating each group to one processor. In fact, the ScaLAPACK routine *pdstein* adopts this strategy.

But as the size of the matrix grows, the distance between adjacent eigenvalues becomes smaller, and the number of eigenvalues belonging to a group becomes large. In particular, it has been observed in many problems that if the criterion for grouping is set at $\epsilon = 10^{-3} \| \mathbf{T} \|_1$, which is a widely accepted value [101], most of the eigenvalues belong to one group when N is greater than 1000. If this is the case, most of the calculation has to be performed by one processor, and there is virtually no effect of parallelization.

When the group-level parallelism is not available, the modified Gram-Schmidt method itself has to be parallelized. Because the method is sequential about index k, the only possibility is to parallelize the BLAS-1 (vector-vector operation) routines that appear in the innermost loop, such as the inner product $c = \mathbf{v}_i^{(m)t}\mathbf{v}_k$ and the AXPY operation $\mathbf{v}_i^{(m)} := \mathbf{v}_i^{(m)} - c\mathbf{v}_k$. But this would cause as many as $O(N^2)$ interprocessor synchronization to compute all the eigenvectors, when most of the eigenvalues belong to the same group. Considering the fact that other parts of the eigenvalue solver such as the tri-diagonalization and back-transformation need only $O(N)$ synchronization, this is prohibitively expensive.

From the above discussion, we can say that there is no effective scheme for parallelizing the conventional inverse iteration method, when most of the eigenvalues belong to the same group.

7.3 The Householder inverse iteration method

In this section, we give the basic idea and the algorithm of the Householder inverse iteration method [108], which is a new eigenvector solver suited for a shared-memory concurrent computer. We also compare the arithmetic operation count of the new algorithm with that of the conventional method.

7.3.1 The basic idea

In the conventional inverse iteration method, the components of the newly computed vector that are parallel to the previously computed eigenvectors are removed by the modified Gram-Schmidt orthogonalization. However, because the MGS method is sequential about index k, the BLAS-1 operations such as $c = \mathbf{v}_i^{(m)t}\mathbf{v}_k$ and $\mathbf{v}_i^{(m)} := \mathbf{v}_i^{(m)} - c\mathbf{v}_k$ have to be parallelized when the group-level parallelism is not available. This brings about small granularity of $O(N)$ and extremely large amount of synchronization of $O(N^2)$.

To avoid this problem, we abandon using the MGS method for orthogonalization. Instead, we choose to hold the basis of the orthogonal complementary subspace of the previously calculated eigenvectors explicitly. Then we can make the newly calculated vector orthogonal to the previous eigenvectors by projecting it onto this subspace. After that, the orthogonal complementary subspace is updated so that it is orthogonal also to the newly calculated eigenvector.

Let $\mathbf{v}_i'$ be the newly calculated (i-th) eigenvector (before orthogonalization), V_{i-1} be the subspace spanned by the 1st to (i–1)-th eigenvectors, namely, $V_{i-1} = span\{\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_{i-1}\}$, $V_{i-1}^{\perp}$ be the orthogonal complementary subspace of V_{i-1} in $\mathbf{R}^N$, $\mathbf{Q}_{i-1}$, an N by $N-i+1$ matrix, be the orthonormal basis of $V_{i-1}^{\perp}$, and $\mathbf{e}_j$ be an $N-i+1$ dimensional vector whose j-th component is one and all the other components are zero. Then the orthogonalization process for $\mathbf{v}_i'$ can be described as follows:

(1) Calculate $\mathbf{p}_i = \mathbf{Q}_{i-1}^t\mathbf{v}_i'$.

(2) Find a Householder transformation $\mathbf{H}_i = \mathbf{I}_{N-i+1} - \alpha_i\mathbf{w}_i\mathbf{w}_i^t$ which clears the second and the following components of $\mathbf{p}_i$.

(3) Calculate $\mathbf{Q}_{i-1}\mathbf{H}_i$.

(4) Adopt the first column of $\mathbf{Q}_{i-1}\mathbf{H}_i$ as the orthogonalized new eigenvector $\mathbf{v}_i$, and adopt the matrix that consists of the second and the following columns of $\mathbf{Q}_{i-1}\mathbf{H}_i$ as $\mathbf{Q}_i$.

In the step (1) above, $\mathbf{v}'_i$ is projected onto $V_{i-1}^{\perp}$ and the resulting vector is expanded using the orthonormal basis $\mathbf{Q}_{i-1}$. The vector of coefficients in this expansion is given by $\mathbf{p}_i$. In step (3), The Householder transformation $\mathbf{H}_i$ is applied to $\mathbf{Q}_{i-1}$ from the right. Then, the first column of $\mathbf{Q}_{i-1}\mathbf{H}_i$ is parallel to the projection of $\mathbf{v}'_i$ onto $V_{i-1}^{\perp}$, because

$$\begin{aligned}(\mathbf{Q}_{i-1}\mathbf{H}_i)\mathbf{e}_1 &= (1/\beta_i)\mathbf{Q}_{i-1}\mathbf{H}_i\mathbf{H}_i\mathbf{p}_i \\ &= (1/\beta_i)\mathbf{Q}_{i-1}\mathbf{p}_i = (1/\beta_i)\mathbf{Q}_{i-1}\mathbf{Q}_{i-1}^t\mathbf{v}'_i. \qquad (7.2)\end{aligned}$$

Here we used the fact that $\mathbf{H}_i\mathbf{p}_i = \beta_i\mathbf{e}_1$ for some β_i, and assumed that β_i is not zero because $\beta_i = 0$ would imply that $\mathbf{v}'_i$ consists only of the components which are parallel to the previously calculated eigenvectors. We can also show that all the other columns of $\mathbf{Q}_{i-1}\mathbf{H}_i$ are orthogonal to $\mathbf{v}'_i$ because

$$\mathbf{v}_i'^t(\mathbf{Q}_{i-1}\mathbf{H}_i)\mathbf{e}_j = \mathbf{p}_i^t\mathbf{H}_i\mathbf{e}_j = \beta_i\mathbf{e}_1^t\mathbf{e}_j = 0 \qquad \text{for} \quad j > 1. \qquad (7.3)$$

So we can adopt the former as the orthogonalized eigenvector $\mathbf{v}_i$ and the latter as $\mathbf{Q}_i$, an orthonormal basis of the new orthogonal complementary subspace $V_i^{\perp}$, in step (4).

As an initial orthonormal basis, we use the unit matrix of order N. In our algorithm, this initial matrix $\mathbf{Q}_0 = \mathbf{I}_N$ is successively updated by the Householder transformations, and is finally transformed to a matrix whose column vectors are the eigenvectors of $\mathbf{T}$. Considering that the Householder transformations keep the orthogonality of a matrix to high accuracy [46], it can be expected that the eigenvectors obtained by this algorithm are highly orthogonal. Moreover, the main operations of this algorithm are projection of $\mathbf{v}_i$ to $V_{i-1}^{\perp}$ in step (1) and the Householder transformation of $\mathbf{Q}_{i-1}$ in step (2), both of which are the BLAS-2 (matrix-vector) operations. The number of interprocessor synchronization needed to parallelize the algorithm on SMP machines is therefore $O(1)$ for orthogonalization of one eigenvector and $O(N)$ for all eigenvectors.

7.3.2 The algorithm

Details of the Householder inverse iteration method are shown as Algorithm 7.2. The additional procedures for degenerate or tightly clustered eigenvalues are not shown in the figure, but are the same as for the conventional method.

```
[Algorithm 7.2 Householder inverse iteration method]
$\mathbf{Q}_0 := \mathbf{I}_N$
$\mathbf{V}_0 := \phi$ (an $N$ by 0 matrix)
for $i$=1: $N$
  Set some initial vector $\mathbf{v}_i^{(0)}$.
  $m := 1$
  until $\mathbf{v}_i^{(m)}$ converges
    $\mathbf{v}_i' := (\mathbf{T} - e_i'\mathbf{I})^{-1}\mathbf{v}_i^{(m-1)}$
    $\mathbf{p}_i := \mathbf{Q}_{i-1}^t \mathbf{v}_i'$ ($\mathbf{p}_i$ is a vector of length $N - i + 1$.)    (A.1)
    Find a Householder transformation $\mathbf{H}_i = \mathbf{I}_{N-i+1} - \alpha_i \mathbf{w}_i \mathbf{w}_i^t$
    which clears the second and the following components of $\mathbf{p}_i$.
    ($\mathbf{w}_i$ is a vector of length $N - i + 1$.)
    $\mathbf{q}_i := \alpha_i \mathbf{Q}_{i-1} \mathbf{w}_i$ ($\mathbf{q}_i$ is a vector of length $N$.)    (A.2)
    $\mathbf{v}_i^{(m)} := (\mathbf{Q}_{i-1})_1 - \mathbf{q}_i(\mathbf{w}_i^t)_1$
    (($\mathbf{A}$)$_i$ denotes the $i$-th column of matrix $\mathbf{A}$.)
    $m := m+1$
  end
  $\mathbf{Q}_{i-1}' := \mathbf{Q}_{i-1} - \mathbf{q}_i \mathbf{w}_i^t$
  (update $\mathbf{Q}_{i-1}$ by the Householder transformation)    (A.3)
  $\mathbf{V}_i := [\mathbf{V}_{i-1} | (\mathbf{Q}_{i-1}')_1]$
  Set $\mathbf{Q}_i$ to be the matrix obtained by eliminating the first column
  of $\mathbf{Q}_{i-1}'$.
end
```

7.3.3 Arithmetic operation count

The main operations of the Householder inverse iteration are equations (A.1), (A.2) and (A.3) in Algorithm 7.2. The equation (A.1) projects the computed vector $\mathbf{v}_i$ to the orthogonal subspace $V_{i-1}^{\perp}$, while (A.2) and (A.3) performs the Householder transformation. Assuming that the inverse iteration converges with single iteration, each of (A.1), (A, 2) and (A.3) needs $2N(N-i+1)$ operations for the i-th eigenvector, and about N^3 operations for all eigenvectors. The total operation count is therefore $3N^3$. On the other hand, the conventional IIM needs $2N^3$ arithmetic operations when all the eigenvalues belong to the same group. This means that our method requires 1.5 times the operation count of the conventional method.

However, in contrast to the conventional algorithm, where almost all the operations are done in BLAS-1 routines such as inner-product and AXPY, our algorithm is based on

BLAS-2 routines such as matrix-vector multiplication and rank-1 update of a matrix. Our method therefore leaves room for code optimization such as loop unrolling. By combining such techniques with reduced interprocessor synchronization, our new method has the potential to outperform the conventional method on shared-memory machines.

When the number of wanted eigenvectors is smaller than N, say N', the number of operations needed to perform each of (A, 1), (A, 2) and (A, 3) is

$$\sum_{i=1}^{N'} 2N(N-i+1) = N^2N' - \frac{1}{2}NN'(N'+1) + NN'. \tag{7.4}$$

So we need about $3N^2N' - (3/2)N'^2N$ total operations. Because the conventional IIM needs about $2NN'^2$ operations, our current algorithm is not competitive when N' is considerably smaller than N. However, by using the WY-representation, it is in principle possible to construct a modified algorithm which requires only $O(NN'^2)$ operations. We are now developing such an algorithm.

7.4 The blocked algorithm

To attain high performance on a modern computer with hierarchical memory, it is important to increase the locality of data reference and use the data as many times as possible while it is in the cache. Such consideration becomes more important in shared-memory multiprocessor environment, because it helps preventing performance degradation due to bus contention between the processors, by enabling most of the data accesses to be done in the local cache associated with each processor.

In numerical linear algebra algorithms, locality of data reference can usually be increased by blocking, that is, by reconstructing the algorithm so that most of the computation is performed in BLAS-3 routines. The BLAS-3 routines can perform $O(L^3)$ operations on $O(L^2)$ data when the size of blocking is L, and thereby reduce the memory access by a factor of L when L is chosen so that all the necessary blocks can be stored in the cache.

In our algorithm described in the previous section, blocking is possible by deferring application of the Householder transformation on $\mathbf{Q}$ until several transformations are available, and then applying these successive transformations at once using the *WY representation* [46]. Let L be the size of blocking and i be an integer such that $1 \leq i \leq N$ and $\mathrm{mod}(i, L) = 1$. Then, in the i-th step of the blocked algorithm, after generating the Householder transformation $\mathbf{H}_i = \mathbf{I}_{N-i+1} - \alpha_i \mathbf{w}_i \mathbf{w}_i^t$, we skip its application on $\mathbf{Q}_{i-1}$ and instead accumulate it as WY representation for block Householder transformation as follows:

$$\mathbf{Y}^{(0)} = \mathbf{w}_i \tag{7.5}$$

$$\mathbf{W}^{(0)} = -\alpha_i \mathbf{w}_i \tag{7.6}$$

The following $L-1$ steps are executed in a similar way. At the $i+j$-th step $(1 \le j \le L-1)$, the matrices $\mathbf{Y}$ and $\mathbf{W}$ are updated as follows:

$$\begin{aligned} \mathbf{z} &= -\alpha_{i+j}(\mathbf{I} + \mathbf{W}^{(j-1)}\mathbf{Y}^{(j-1)})\mathbf{w}_{i+j} & (7.7) \\ \mathbf{W}^{(j)} &= [\mathbf{W}^{(j-1)}|\mathbf{z}] & (7.8) \\ \mathbf{Y}^{(j)} &= [\mathbf{Y}^{(j-1)}|\mathbf{w}_{i+j}], & (7.9) \end{aligned}$$

where $[\mathbf{A}|\mathbf{B}]$ denotes concatenation of two matrices. At the end of the $i+L-1$-th step, the block Householder transformation is applied to $\mathbf{Q}_{i-1}$, generating $\mathbf{Q}_{i+L-1}$ directly:

$$\mathbf{Q}_{i+L-1} = \mathbf{Q}_{i-1}(\mathbf{I} + \mathbf{W}^{(L-1)}\mathbf{Y}^{(L-1)})^t \tag{7.10}$$

As is clearly seen from eq. (7.10), application of the block Householder transformation can be done using only matrix-matrix multiplications, or BLAS-3 routines.

Of course, we also have to change eq. (A.1) in the non-blocked algorithm, because the matrix $\mathbf{Q}_{i+j-1}$ has not received necessary transformation at intermediate stages $i+j$ $(1 \le j \le L-1)$. The correct formula to calculate $\mathbf{p}_{i+j}$ is

$$\begin{aligned} \mathbf{p}_{i+j} &= (\mathbf{I} + \mathbf{W}^{(j-1)}\mathbf{Y}^{(j-1)})\mathbf{Q}_{i-1}^t \mathbf{v}'_{i+j} \\ &= \mathbf{Q}_{i-1}^t \mathbf{v}_{i+j} - \mathbf{W}^{(j-1)}\mathbf{Y}^{(j-1)}\mathbf{Q}_{i-1}^t \mathbf{v}'_{i+j} & (7.11) \end{aligned}$$

Though the additional terms in eq. (7.11) increase the number of arithmetic operations slightly, the performance improvement due to the use of BLAS-3 will more than compensate for it.

We summarize the blocked version of our Householder Inverse Iteration method as Algorithm 7.3. Here, we assume for simplicity that N is divisible by L. In this algorithm, two thirds of the total operation can be done in BLAS-3, and the locality of data reference is greatly improved compared with the original algorithm given in the previous section.

[Algorithm 7.3 Blocked version of the Householder inverse iteration method]

```
$\mathbf{Q}_0 := \mathbf{I}_N$
$\mathbf{V}_0 := \phi$ (an $N$ by 0 matrix)
for $ib$=1: $N/L$
  $i := (ib - 1) * L + 1$
  for $j$=0: $L - 1$
    Set some initial vector $\mathbf{v}_{i+j}^{(0)}$.
    $m := 1$
    until $\mathbf{v}_{i+j}^{(m)}$ converges
      $\mathbf{v}'_{i+j} := (\mathbf{T} - e'_{i+j}\mathbf{I})^{-1}\mathbf{v}_{i+j}^{(m-1)}$
      if $j = 0$
        $\mathbf{p}_i := \mathbf{Q}_{i-1}^t \mathbf{v}'_i$
      else
        $\mathbf{p}_{i+j} := \mathbf{Q}_{i-1}^t \mathbf{v}'_{i+j} - \mathbf{W}^{(j-1)}\mathbf{Y}^{(j-1)}\mathbf{Q}_{i-1}^t \mathbf{v}'_{i+j}$
        ($\mathbf{p}_{i+j}$ is a vector of length $N - i + 1$.)
      end if
      Find a Householder transformation
      $\mathbf{H}_{i+j} = \mathbf{I}_{N-i+1} - \alpha_{i+j}\mathbf{w}_{i+j}\mathbf{w}_{i+j}^t$
      which clears the second and the following components
      of $\mathbf{p}_{i+j}$.
      if $j = 0$
        $\mathbf{Y}^{(0)} := \mathbf{w}_i$
        $\mathbf{W}^{(0)} := -\alpha_i \mathbf{w}_i$
      else
        $\mathbf{z} := -\alpha_{i+j}(\mathbf{I} + \mathbf{W}^{(j-1)}\mathbf{Y}^{(j-1)})\mathbf{w}_{i+j}$
        $\mathbf{W}^{(j)} := [\mathbf{W}^{(j-1)}|\mathbf{z}]$
        $\mathbf{Y}^{(j)} := [\mathbf{Y}^{(j-1)}|\mathbf{w}_{i+j}]$
      end if
      $m := m+1$
    end
    $\mathbf{Q}'_{i-1} := \mathbf{Q}_{i-1}(\mathbf{I} + \mathbf{W}^{(L-1)}\mathbf{Y}^{(L-1)})^t$
    Partition $\mathbf{Q}'_{i-1}$ as $\mathbf{Q}'_{i-1} = [\mathbf{Q}_{i-1}^L|\mathbf{Q}_{i-1}^R]$, where $\mathbf{Q}_{i-1}^L$ consists
    of the first $L$ columns of $\mathbf{Q}'_{i-1}$.
    $\mathbf{V}_{i+L-1} := [\mathbf{V}_{i-1}|\mathbf{Q}_{i-1}^L]$
    $\mathbf{Q}_{i+L-1} := \mathbf{Q}_{i-1}^R$.
  end
end
```

7.5 Numerical results

7.5.1 Computing environments

We evaluated the performance and numerical accuracy of our Householder Inverse Iteration method on one node of the Hitachi SR8000, an SMP (shared-memory processors) machine with 8 processors per node [95]. Each processor has a peak performance of 1 GFLOPS and the total performance per node is 8 GFLOPS. We also used SR8000/G1, which has 14.4GFLOPS of total peak performance. For parallelization of the program, we used an automatically parallelizing FORTRAN compiler and specified the loops to be parallelized using compiler directives. As test matrices, we used the following two kinds of matrices:

(a) The Frank matrix: $\mathbf{A}_{ij} = \min(i, j)$.

(b) Matrices obtained from a generalized eigenvalue problem $\mathbf{Av} = e\mathbf{Bv}$. Here $\mathbf{A}$ and $\mathbf{B}$ are random matrices whose elements were extracted from uniform random numbers in [0,1]. The diagonal elements of $\mathbf{B}$ were then replaced with 10^4 to ensure positive definiteness.

Both types of matrices were first tri-diagonalized by orthogonal transformations and then used as an input matrix for our algorithm.

7.5.2 Performance

First we show in Table 7.1 the execution times of the conventional inverse iteration method and the non-blocked version of the Householder inverse iteration method on the SR8000. The input matrices we used here are of type (a), but the execution times for matrices of type (b) were almost the same. The numbers in the parentheses show the execution time for computing the eigenvectors only, while those outside also include the time to compute the eigenvalues by the bisection method. We also show the execution time of the conventional IIM on the Hitachi S3800, a vector supercomputer that has the same peak performance of 8GFLOPS. Here, the time is for computing both the eigenvalues and eigenvectors, because the numerical library we used for this measurement did not have the function to compute only the eigenvectors.

The figures show that the Householder inverse iteration method is more efficient than the conventional ones, especially when N is small, and achieves 2.4 times the performance when computing the eigenvectors of a matrix of order 1000. When comparing the execution time on the SR8000 and the S3800, one can see that while the conventional method

Table 7.1: Performance comparison of the Householder and the conventional IIM

Problem size	Conventional IIM (SR8000)	Householder IIM (SR8000)	Conventional IIM (S3800)
N=1000	4.21s (3.92s)	2.06s (1.64s)	2.15s
N=2000	18.84s (17.61s)	12.05s (10.68s)	12.40s
N=4000	98.46s (94.11s)	83.37s (78.68s)	80.65s

fails to exploit the performance of the SMP machine due to a large number of interprocessor synchronization, our new method solves this problem and succeeds in attaining the same level of performance as that of the vector supercomputer even on the SMP machine.

Table 7.2 shows the execution times of the conventional and the Householder IIM on the SR8000/G1. In this case, the execution times of the blocked algorithm described in section 4 are also shown. It is apparent from the table that the blocking works well and increases the performance by about 50%. For the case of $N = 1000$, the blocked version of the Householder IIM achieves more than 3.1 times the performance of the conventional method.

As can be seen from tables 7.1 and 7.2, the superiority of our algorithm over the conventional IIM is large when N is small and decreases as N grows. This is natural considering that our algorithm reduces interprocessor synchronization at the cost of increased operation count. Note, however, that the cost of interprocessor synchronization is relatively low on the SR8000 [95]. For other SMP machines that have higher interprocessor synchronization cost, the effect of reducing the synchronization is larger and the effectiveness of our algorithm will remain for much larger value of N.

Table 7.2: Performance comparison of the Householder and the conventional IIM (SR8000/G1, execution time for the inverse iteration part.)

Problem size	Conventional IIM	Householder IIM (non-blocked)	Householder IIM (blocked)
N=1000	2.20s	0.98s	0.70s
N=2000	9.93s	6.81s	4.36s
N=4000	49.84s	49.11s	30.76s

7.5.3 Numerical accuracy

To check the numerical accuracy of the new method, we evaluated the residual and orthogonality of the computed eigenvectors for the new and the conventional method. Here, the residual is defined as the maximum of the L_2-norm of $\mathbf{T}\mathbf{v}_i - e_i\mathbf{v}_i$ over all i, where e_i

is the computed i th eigenvalue and $\mathbf{v}_i$ is the computed corresponding eigenvector. The orthogonality is defined as the modulus of the element of $\mathbf{V}^t\mathbf{V} - \mathbf{I}_N$ with the maximum modulus, where $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_N]$.

The results for the Frank matrices and the matrices from generalized eigenvalue problems are shown in Tables 7.3, 7.4 and Tables 7.5, 7.6, respectively. As can be seen from the tables, the residual for the non-blocked version of the Householder IIM is as good as that for the conventional one. As for the orthogonality of the computed eigenvectors, the method gives results that are better than or at least as good as those for the conventional method. It is also clear that blocking does not deteriorate the numerical accuracy either in terms of residual or orthogonality.

Table 7.3: Comparison of the accuracy of the Householder and the conventional IIM (Residual, Frank matrices)

Problem size	Conventional IIM	Householder IIM	Householder IIM
		(non-blocked)	(blocked)
N=1000	0.164×10^{-7}	0.164×10^{-7}	0.164×10^{-7}
N=2000	0.111×10^{-6}	0.111×10^{-6}	0.111×10^{-6}
N=4000	0.528×10^{-6}	0.528×10^{-6}	0.528×10^{-6}

Table 7.4: Comparison of the accuracy of the Householder and the conventional IIM (Orthogonality, Frank matrices)

Problem size	Conventional IIM	Householder IIM	Householder IIM
		(non-blocked)	(blocked)
N=1000	0.138×10^{-12}	0.400×10^{-14}	0.433×10^{-14}
N=2000	0.945×10^{-13}	0.622×10^{-14}	0.644×10^{-14}
N=4000	0.821×10^{-13}	0.124×10^{-13}	0.127×10^{-13}

Table 7.5: Comparison of the accuracy of the Householder and the conventional IIM (Residual, Matrices from generalized eigenvalue problems)

Problem size	Conventional IIM	Householder IIM	Householder IIM
		(non-blocked)	(blocked)
N=1000	0.881×10^{-12}	0.858×10^{-12}	0.895×10^{-12}
N=2000	0.478×10^{-11}	0.475×10^{-11}	0.478×10^{-11}
N=4000	0.195×10^{-10}	0.197×10^{-10}	0.196×10^{-10}

Table 7.6: Comparison of the accuracy of the Householder and the conventional IIM (Orthogonality, Matrices from generalized eigenvalue problems)

Problem size	Conventional IIM	Householder IIM (non-blocked)	Householder IIM (blocked)
N=1000	0.824×10^{-11}	0.867×10^{-11}	0.837×10^{-11}
N=2000	0.932×10^{-11}	0.976×10^{-11}	0.892×10^{-11}
N=4000	0.119×10^{-10}	0.118×10^{-10}	0.155×10^{-10}

7.6 Conclusion

In this article, we proposed a new algorithm for computing the eigenvectors of a real symmetric matrix on shared-memory concurrent computers. In our algorithm, we chose to hold the basis of the orthogonal complementary subspace of the previously calculated eigenvectors and successively update it by the Householder transformations. This obviates the need for the modified Gram-Schmidt orthogonalization, which is the bottleneck in parallelizing the conventional inverse iteration, and reduces the number of interprocessor synchronization from $O(N^2)$ to $O(N)$. The performance of the algorithm is further enhanced with the blocking technique, which allows the use of BLAS-3 routines. The orthogonality of the computed eigenvectors is expected to be good because the Householder transformations keep the orthogonality to high accuracy.

We evaluated our method on one node of the Hitachi SR8000, an SMP machine with 8 processors, and obtained up to 3.1 times the performance of the conventional method when computing all the eigenvectors of matrices of order 1000 to 4000. The orthogonality of the eigenvectors is better than or at least as good as that of the conventional method.

Our future work will include application of this algorithm to distributed-memory parallel computers.

Chapter 8

Fast Fourier Transform on Distributed-Memory Vector Parallel Machines

8.1 Introduction

The fast Fourier transform (FFT) is one of the most widely used algorithms in the field of scientific computing. It can reduce the computational work needed to compute the Fourier transform of an N-point complex sequence from $O(N^2)$ to $O(N \log N)$ and has played an important role in areas as diverse as signal processing, computational fluid dynamics, solid state physics and financial engineering, etc.

The FFT has a large degree of parallelism in each stage of the computation, and accordingly, its implementations on parallel machines have been well studied. See, for example, [12] [21] [93] for implementations on shared-memory parallel machines and [2] [38] [53] [57] [93] [94] for implementations on distributed-memory parallel machines. Recently, distributed-memory machines with (pseudo-)vector processing nodes have become increasingly popular in high-end applications. The machines classified in this category include NEC SX-7, Fujitsu VPP5000 and Hitachi SR2201 and SR8000. To attain high performance on this type of machines, one has to achieve both high single-processor performance and high parallel efficiency at the same time. The former is realized by maximizing the length of the innermost loops, while the latter is realized when the volume of inter-processor communication is minimized. Implementations based on the transpose algorithm [42] [64] which satisfy both of these requirements are given in [2] [53] [94].

While there have been considerable efforts towards a high performance parallel implementation of the FFT, the problem of providing the user with more flexibility of data distribution has attracted relatively little attention. To compute the FFT in a distributed-memory environment, the user need to distribute the input data among processors in a manner specified by the FFT routine, call the routine, and receive the output data again in a manner specified by the routine. In many cases, the data distribution scheme used by the FFT routine is fixed, so if it is different from that used in other parts of the program, the user has to rearrange the data before or after calling the routine. This problem could be mitigated if data redistribution routines are provided along with the FFT routine. However, because the FFT requires only $O(N \log N)$ computation when the number of data points is N, the additional overhead incurred by the redistribution routines is often too costly.

To solve the problem, Dubey et al. [38] propose a general-purpose subroutine for 1-dimensional FFT. Their routine is quite flexible in the sense that it can accept general block cyclic data distributions. Here, block cyclic distribution is a data distribution in which the data is divided into blocks of equal size, say L, and the i-th block is allocated to node $\mathrm{mod}(i, P)$, where P is the number of nodes. Their routine has a marked advantage that the amount of inter-processor communication needed for performing the FFT is independent of the block size L. However, it has several shortcomings. First, it is based on the binary exchange algorithm [42] [64], which requires $O((N/P) \log P)$ inter-processor communication for each node. This is much greater than the communication volume of $O(N/P)$ required by the transpose algorithm. Second, it is not self-sorting, so if one needs a sorted output, additional inter-processor communication is necessary. Finally, no consideration on vectorization has been given.

In this chapter, we propose another general-purpose 1-dimensional FFT routine for distributed-memory vector-parallel machines. Our method is an extension of an FFT algorithm proposed by Takahashi [94], which is based on the transpose algorithm. His algorithm has the advantage that it requires only one global transposition, is self-sorting, and can input/output data scattered with cyclic ($L = 1$) distribution. We extend this algorithm to accept input data scattered with a block cyclic distribution of block size L_1 and to output the result using a block cyclic distribution of another block size, say L_2. L_1 and L_2 are arbitrary as long as N is a multiple of $P^2 * L_1 * L_2$. This flexibility can be realized without increasing the amount of inter-processor communication, in contrast to the approaches that rely on redistribution routines.

If our method is implemented in a straightforward manner, however, the length of the innermost loops tends to become shorter as the block sizes grow, causing degradation of single-processor performance. We solve this problem by adopting the Stockham's FFT [97] suited to vector processors as the FFT kernels and employing loop merging techniques. We implemented our method on the Hitachi SR2201, a distributed-memory parallel machine with pseudo-vector processing nodes, and measured its performance using 1 to 16 nodes.

The rest of this chapter is organized as follows: In section 8.2 we describe the conventional FFT algorithms for vector-parallel machines. Our new implementation is introduced in section 8.3 along with several considerations to attain high performance on vector-parallel machines. Section 8.4 shows the performance of our routine on the Hitachi SR2201. Conclusions are given in the final section.

8.2 Conventional FFT algorithms for vector-parallel machines

8.2.1 1-D FFT algorithms for vector machines

In this section, we will explain conventional algorithms for 1-dimensional FFT on vector and vector-parallel machines following [2] [13] [94]. The discrete Fourier transform of a 1-dimensional complex sequence $\{f_0, f_1, \ldots, f_{N-1}\}$ is defined as follows:

$$c_k = \sum_{j=0}^{N-1} f_j \omega_N^{jk} \qquad (k = 0, 1, \ldots, N-1), \tag{8.1}$$

where $\omega_N = \exp(-2\pi i/N)$ and $i = \sqrt{-1}$.

When N can be factored as $N = N_x N_y$, the indices j and k can be expressed in a two-dimensional form:

$$\begin{aligned} j &= j_x N_y + j_y \qquad (j_x = 0, \ldots, N_x - 1, \quad j_y = 0, \ldots, N_y - 1), && (8.2) \\ k &= k_x + k_y N_x \qquad (k_x = 0, \ldots, N_x - 1, \quad k_y = 0, \ldots, N_y - 1). && (8.3) \end{aligned}$$

Accordingly, $\{f_j\}$ and $\{c_k\}$ can be regarded as two-dimensional arrays:

$$\begin{aligned} f_{j_x, j_y} &= f_{j_x N_y + j_y}, && (8.4) \\ c_{k_x, k_y} &= c_{k_x + k_y N_x}. && (8.5) \end{aligned}$$

Using these notations, we can rewrite eq. (8.1) as follows:

$$\begin{aligned} c_{k_x, k_y} &= \sum_{j_y=0}^{N_y-1} \sum_{j_x=0}^{N_x-1} f_{j_x, j_y} \omega_N^{(j_x N_y + j_y)(k_x + k_y N_x)} \\ &= \sum_{j_y=0}^{N_y-1} \left(\left(\sum_{j_x=0}^{N_x-1} f_{j_x, j_y} \omega_{N_x}^{j_x k_x} \right) \omega_N^{j_y k_x} \right) \omega_{N_y}^{j_y k_y}. \end{aligned} \tag{8.6}$$

This shows that the Fourier transform of $\{f_j\}$ can be computed by the following algorithm proposed by Bailey [13]:

[Algorithm 8.1]

1. Compute $c'_{k_x,j_y} = \sum_{j_x=0}^{N_x-1} f_{j_x,j_y} \omega_{N_x}^{j_x k_x}$ by repeating N_x-point FFT N_y times.
2. Multiply c'_{k_x,j_y} by $\omega_N^{j_y k_x}$.
3. Compute $c_{k_x,k_y} = \sum_{j_y=0}^{N_y-1} c'_{k_x,j_y} \omega_{N_y}^{j_y k_y}$ by repeating N_y-point FFT N_x times.

The factor $\omega_N^{j_y k_x}$ appearing in step 2 is called *twiddle factor* and the step 2 is called *twiddle factor multiplication*. This algorithm requires about the same amount of computational effort as the FFT of N data points. It is especially suited to vector machines if the loops about j_y and k_x are used as the innermost loops in steps 1 and 3, respectively. Then the innermost loops will have a fixed length of N_y and N_x and the factor ω, which is a constant within these loops, can be loaded outside the loops.

8.2.2 1-D FFT algorithms for distributed-memory vector-parallel machines

In the algorithm explained in the previous subsection, we decompose the 1-D FFT into multiple FFTs of smaller size and use this multiplicity for vectorization. In the case of distributed-memory vector-parallel machines, we need another dimension to use for parallelization. To this end, we factor N as $N = N_x N_y N_z$ and introduce a three-dimensional notation for the indices j and k:

$$\begin{aligned} j &= j_x N_y N_z + j_y N_z + j_z & (8.7) \\ & \quad (j_x = 0, \ldots, N_x - 1, \quad j_y = 0, \ldots, N_y - 1, \quad j_z = 0, \ldots, N_z - 1), \\ k &= k_x + k_y N_x + k_z N_x N_y & (8.8) \\ & \quad (k_x = 0, \ldots, N_x - 1, \quad k_y = 0, \ldots, N_y - 1, \quad k_z = 0, \ldots, N_z - 1). \end{aligned}$$

By regarding the input and output sequences as three-dimensional arrays f_{j_x,j_y,j_z} and c_{k_x,k_y,k_z}, we can rewrite eq. (8.1) as follows:

$$\begin{aligned} & c_{k_x,k_y,k_z} \\ = & \sum_{j_z=0}^{N_z-1} \left(\left(\sum_{j_y=0}^{N_y-1} \left(\left(\sum_{j_x=0}^{N_x-1} f_{j_x,j_y,j_z} \omega_{N_x}^{j_x k_x} \right) \omega_{N_x N_y}^{j_y k_x} \right) \omega_{N_y}^{j_y k_y} \right) \omega_N^{j_z (k_x + k_y N_x)} \right) \omega_{N_z}^{j_z k_z}. \end{aligned} \tag{8.9}$$

This suggests the following five-step FFT proposed by Takahashi [94]:

[Algorithm 8.2: Five-step FFT]

1. Compute $c'_{k_x,j_y,j_z} = \sum_{j_x=0}^{N_x-1} f_{j_x,j_y,j_z} \omega_{N_x}^{j_x k_x}$ by repeating N_x-point FFT $N_y N_z$ times.
2. Twiddle factor multiplication (I): multiply c'_{k_x,j_y,j_z} by $\omega_{N_x N_y}^{j_y k_x}$.
3. Compute $c''_{k_x,k_y,j_z} = \sum_{j_y=0}^{N_y-1} c'_{k_x,j_y,j_z} \omega_{N_y}^{j_y k_y}$ by repeating N_y-point FFT $N_x N_z$ times.
4. Twiddle factor multiplication (II): multiply c''_{k_x,k_y,j_z} by $\omega_N^{j_z(k_x+k_y N_x)}$.
5. Compute $c_{k_x,k_y,k_z} = \sum_{j_z=0}^{N_z-1} c''_{k_x,k_y,j_z} \omega_{N_z}^{j_z k_z}$ by repeating N_z-point FFT $N_x N_y$ times.

Because the operation in step 1 consists of $N_y N_z$ independent FFTs, we can, for example, use the index j_y for vectorization and the index j_z for parallelization. Steps 3 and 5 can be executed in a similar way.

8.2.3 Implementations based on the five-step algorithm

There are many possible ways to exploit the parallelism in Algorithm 2 for vectorization and parallelization. For example, Agarwal et al. [2] propose to scatter the three-dimensional array along the z-direction in steps 1 and 2, and along the x-direction in steps 3-5, both using block distribution. In this case, indices j_y, j_z and j_y can be used for vectorization in steps 1, 3 and 5, respectively. Takahashi [94] suggests to scatter the data along the z-direction in steps 1-4, and along the x-direction in step 5, both in a cyclic manner. In this case, vectorization can be done with respect to indices j_y, j_x and j_y in steps 1, 3 and 5, respectively. These methods are classified as the *transpose algorithms*, because all the inter-processor data transfers are done in the form of *global transposition*, i.e., redistribution of a multi-dimensional array scattered along one direction along another direction.

These methods have several advantages: first, they require only one global transposition. The volume of inter-processor communication due to this is $O(N/P)$ per node and is much smaller than $O((N/P)\log P)$, which would be required by the binary exchange algorithms [42] [64]. Second, the innermost loops have a fixed length of N_x, N_y or N_z in steps 1, 3 and 5, respectively. Takahashi also notes that it is possible to extend the length of the innermost loops to $N^{2/3}/P$ by setting $N_x = N_y = N_z = N^{1/3}$ and using loop merging techniques [94]. In addition, his implementation has a natural user interface in the sense that both the input and output data are ordered and distributed in a cyclic fashion [94].

However, some users may need more flexibility of data distribution. For example, block cyclic distribution is frequently used when solving linear simultaneous equations or eigenvalue problems on distributed-memory machines [18]. So if the user wants to connect the FFT routine with these routines, it is more convenient that the FFT routine can input/output data using block cyclic data distribution with user-specified block sizes. Note that the block sizes suitable for input and output data may not be the same, so it is more desirable if they can be specified independently. In the following section, we will propose an algorithm that satisfies these requirements.

8.3 A vector-parallel FFT with flexible data distribution

8.3.1 Conditions on the block sizes

In this section, we propose a 1-D parallel FFT algorithm with the following two properties:

1. The input and output data are scattered with block cyclic distributions with user-specified block sizes L_1 and L_2, respectively.

2. Only one global transposition is needed throughout the algorithm.

And we optimize the algorithm for vector-parallel machines.

Before explaining our algorithm, we will establish a necessary and sufficient condition on L_1 and L_2 for the existence of such an algorithm. For simplicity, here we deal only with the radix-2 FFT and assume that P, L_1 and L_2 are powers of two.

Proposition 8.1 *A necessary and sufficient condition for the existence of a 1-D parallel FFT algorithm that satisfies the above two properties is* $P^2 * L_1 * L_2 \leq N$.

Proof *Here we only show that this is a necessary condition. We prove the sufficiency in the following subsections by actually constructing an algorithm.*

An N-point radix-2 FFT consists of $p = \log_2 N$ stages. By examining its signal flow graph [97], we know that each of the intermediate quantities at the q-th stage ($1 \leq q \leq p$) is computed from every 2^{p-q}-th elements of the input data $\{f_j\}_{j=0}^{N-1}$. These elements reside on the same node if and only if $2^{p-q} \geq L_1 * P$ (assuming $P \geq 2$). This means that we need a global transposition right after the stage

$$q_1 = p - \log_2(L_1 * P) \tag{8.10}$$

or earlier.

On the other hand, we also know from the signal flow graph that each of the intermediate quantities at the q-th stage contributes to every 2^q-th elements of the output data

$\{c_k\}_{k=0}^{N-1}$. *These elements reside on the same node if and only if* $2^q \geq L_2 * P$. *This means that we need a global transposition right before the stage*

$$q_2 = \log_2(L_2 * P) \tag{8.11}$$

or later.

To do with only one global transposition, we need

$$q_1 \geq q_2, \tag{8.12}$$

which implies $P^2 * L_1 * L_2 \leq N$. □

A similar result holds when N is not a power of two and we can construct an FFT algorithm with only one global transposition when N is a multiple of $P^2 * L_1 * L_2$.

8.3.2 The basic idea of the algorithm

To realize an FFT algorithm which has the two properties mentioned in the previous subsection, we use Algorithm 2 as a basis. Assume that N is a multiple of $P^2 * L_1 * L_2$, and choose N_x, N_y and N_z so that N_z and N_x are divisible by $L_1 * P$ and $L_2 * P$, respectively. Now we scatter the three-dimensional array along the z-direction in steps 1 and 2 using block cyclic distribution of block size L_1, and along the x-direction in steps 3-5 using block cyclic distribution of block size L_2. Then, from eq. (8.7), we know that the whole input sequence of length N is scattered with a block cyclic distribution of block size L_1. Likewise, the whole output sequence is scattered with a block cyclic distribution of block size L_2. This method requires only one global transposition like the implementations discussed in the previous subsection, and leaves the room for vectorization using indices j_y, j_z and j_y in steps 1, 3 and 5, respectively.

However, a straightforward implementation of this idea may not guarantee sufficient innermost loop length to achieve high single-processor performance. This is because N_x and N_z need to be large enough to be multiples of $L_1 * P$ and $L_2 * P$, respectively, and therefore N_y, which is the length of the innermost loops in steps 1 and 5, tends to become smaller. For example, when $N = 2^{20}$, $P = 16$ and $L_1 = L_2 = 16$, N_y must be less than or equal to 4. We solve this problem by adopting Stockham's algorithm [97] suited to vector processors in the FFTs in steps 1, 3 and 5, and merging as many loops as possible. We will explain the algorithm and storage scheme for our implementation in the next subsection and discuss loop merging techniques in subsection 8.3.4.

8.3.3 The detailed algorithm and the storage scheme

To describe our implementation, we first introduce some notations. Let $X_p^{(i)}$ denote the partial array allocated to node p at step i. The dimension of $X_p^{(i)}$ varies depending on i.

We also define the indices and their ranges as follows:

$$j_x = 0, \ldots, N_x, \quad j_y = 0, \ldots, N_y - 1, \quad j_z = 0, \ldots, N_z - 1, \tag{8.13}$$

$$k_x = 0, \ldots, N_x, \quad k_y = 0, \ldots, N_y - 1, \quad k_z = 0, \ldots, N_z - 1, \tag{8.14}$$

$$p = 0, \ldots, P - 1, \quad q = 0, \ldots, P - 1, \tag{8.15}$$

$$j'_z = 0, \ldots, N_z/(L_1 P) - 1, \quad j''_z = 0, \ldots, L_1 - 1, \tag{8.16}$$

$$k'_x = 0, \ldots, N_x/(L_2 P) - 1, \quad k''_x = 0, \ldots, L_2 - 1. \tag{8.17}$$

Here, j'_z and k'_x are local block numbers within a node and j''_z and k''_x are indices within a block. They are related to j_z and k_x in the following way:

$$j_z = j'_z L_1 P + pL_1 + j''_z, \tag{8.18}$$

$$k_x = k'_x L_2 P + pL_2 + k''_x, \tag{8.19}$$

where p is the node number.

Using these notations, our FFT can be described as follows:

[Algorithm 8.3]

1. Data input: $X_p^{(1)}(j_y, j''_z, j'_z, j_x) = f_{j_x N_y N_z + j_y N_z + j'_z L_1 P + pL_1 + j''_z}$.

2. FFT in the x-direction:

   $X_p^{(2)}(j_y, j''_z, j'_z, k_x) = \sum_{j_x=0}^{N_x-1} X_p^{(1)}(j_y, j''_z, j'_z, j_x)\omega_{N_x}^{j_x k_x}$.

3. Twiddle factor multiplication (I):

   $X_p^{(3)}(j_y, j''_z, j'_z, k_x) = X_p^{(2)}(j_y, j''_z, j'_z, k_x)\omega_{N_x N_y}^{j_y k_x}$.

4. Data packing for global transposition:

   $X_p^{(4)}(j_y, j''_z, j'_z, k''_x, k'_x, q) = X_p^{(3)}(j_y, j''_z, j'_z, k'_x L_2 P + qL_2 + k''_x)$.

5. Global transposition: $X_p^{(5)}(j_y, j''_z, j'_z, k''_x, k'_x, q) = X_q^{(4)}(j_y, j''_z, j'_z, k''_x, k'_x, p)$.

6. Data unpacking:

   $X_p^{(6)}(j'_z L_1 P + qL_1 + j''_z, k''_x, k'_x, j_y) = X_p^{(5)}(j_y, j''_z, j'_z, k''_x, k'_x, q)$.

7. FFT in the y-direction:

   $X_p^{(7)}(j_z, k''_x, k'_x, k_y) = \sum_{j_y=0}^{N_y-1} X_p^{(6)}(j_z, k''_x, k'_x, j_y)\omega_{N_y}^{j_y k_y}$.

8. Twiddle factor multiplication (II):

   $X_p^{(8)}(k''_x, k'_x, k_y, j_z) = X_p^{(7)}(j_z, k''_x, k'_x, k_y)\omega_N^{j_z(k'_x L_2 P + pL_2 + k''_x + k_y N_x)}$.

9. FFT in the z-direction:

   $X_p^{(9)}(k_x'', k_x', k_y, k_z) = \sum_{j_z=0}^{N_z-1} X_p^{(8)}(k_x'', k_x', k_y, k_z)\omega_{N_z}^{j_z k_z}$.

10. Data output: $c_{k_x' L_2 P + p L_2 + k_x'' + k_y N_x + k_z N_x N_y} = X_p^{(9)}(k_x'', k_x', k_y, k_z)$.

In this algorithm, the most computationally intensive parts are the FFTs in steps 2, 7 and 9. The indexing scheme for array $X_p^{(i)}$ is designed so that the index with respect to which the Fourier transform is performed comes last and the loop merging techniques to be described in the next subsection can be applied easily.

8.3.4 Loop merging techniques for achieving high single-processor performance

From algorithm 8.3, it is apparent that we can merge the loops about the first three indices in the FFTs in steps 2, 7 and 9, and use the resulting loop as the innermost loop. Thus the length of the innermost loops can be extended to $N_y N_z / P$, $N_z N_x / P$ and $N_x N_y / P$ in steps 2, 7 and 9, respectively.

To further extend the innermost loop length, we use Stockham's algorithm [97] in performing these FFTs. Let $n = 2^p$ and assume we want to compute the FFT of an n-point sequence $Y_0(0,0), Y_0(1,0), \ldots, Y_0(n-1,0)$. This can be done with the following algorithm.

```
[Algorithm 8.4 Stockham FFT]
do $L = 0, p-1$
  $\alpha_L = 2^L$
  $\beta_L = 2^{p-L-1}$
  do $k = 0,\ \alpha_L - 1$
    do $j = 0,\ \beta_L - 1$
      $Y_{L+1}(l, m) = Y_L(l, m) + Y_L(l + \beta_L, m)\,\omega^{m\beta_L}$
      $Y_{L+1}(l, m + \alpha_L) = Y_L(l, m) - Y_L(l + \beta_L, m)\,\omega^{m\beta_L}$
    end do
  end do
end do
```

The result is stored in $Y_p(0,0), Y_p(1,0), \ldots, Y_p(n-1,0)$.

Notice that the ω in the innermost loop does not depend on l. This means that if we use this algorithm to compute the N_x-point FFT in step 2, we can merge the loops about j_y, j_z'', j_z' and β_L, thereby extending the length of the innermost loop to $N_y N_z \beta_L / P$.

Because the loop of length β_L appears α_L times in Stockham's algorithm, the average length of the innermost loops in step 2 is

$$\begin{aligned} \frac{N_y N_z}{P} \times \frac{\sum_{L=0}^{\log_2 N_x - 1} \alpha_L \beta_L}{\sum_{L=0}^{\log_2 N_x - 1} \alpha_L} &= \frac{N_y N_z}{P} \times \frac{\frac{N_x}{2} \log_2 N_x}{N_x - 1} \\ &\sim N_y N_z \log_2 N_x / 2P. \end{aligned} \tag{8.20}$$

Hence, the loop length can be increased by a factor of $\log_2 N_x/2$. Similarly, the innermost loop length in steps 7 and 9 can be extended to $N_x N_z \log_2 N_y/2P$ and $N_x N_y \log_2 N_z/2P$, respectively.

As an example, consider the case of $N = 2^{20}$, $P = 16$ and $L_1 = L_2 = 16$ which we mentioned in subsection 8.3.2. We can choose $N_x = N_z = 256$ and $N_y = 4$, and then the length of the innermost loops is 256, 4096 and 256 in steps 2, 7 and 9, respectively. This is enough for many vector machines to attain near-peak performance. Thus we can expect our FFT routine to attain high single-processor performance even when L_1 and L_2 are large and N_y is small.

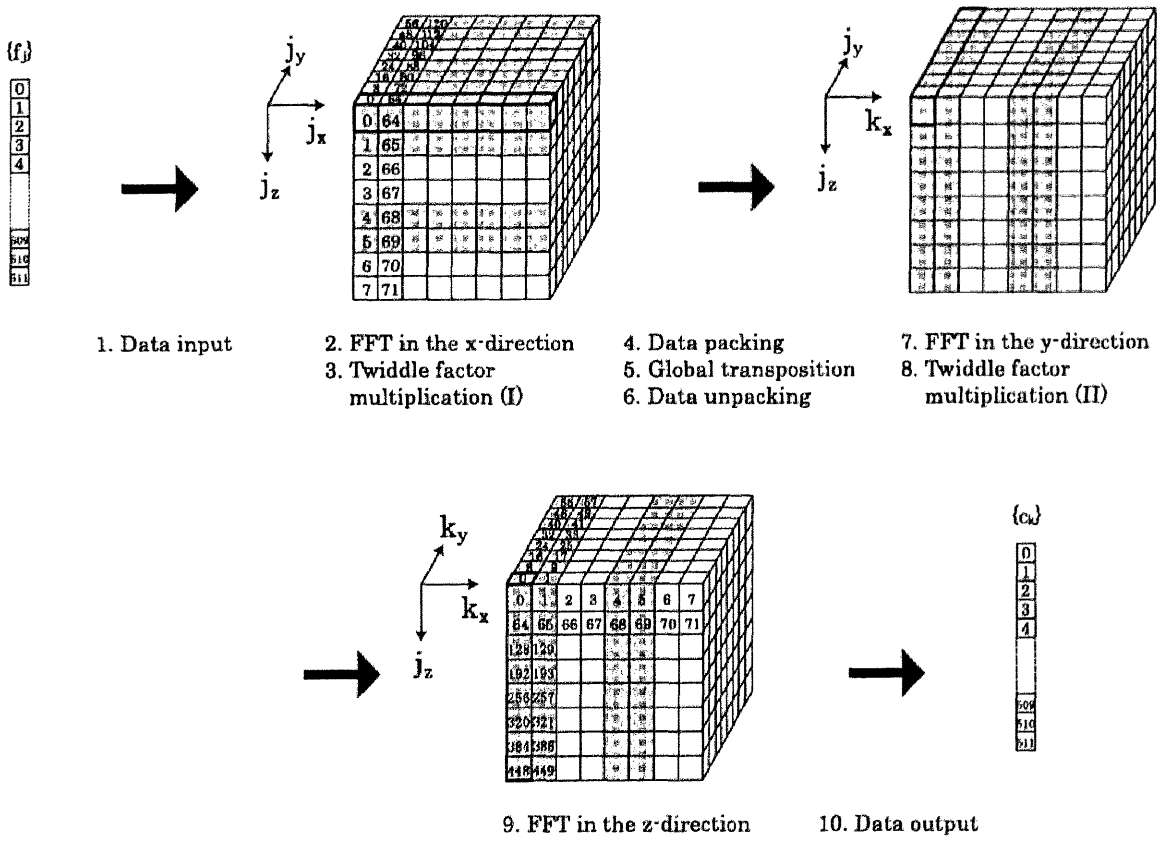
Because the FFT involves only $O(N \log N)$ operations on N-point data, it is also essential for higher performance to minimize memory access. This can be achieved by putting together some of the steps in Algorithm 8.3. For example, data packing for global transposition in step 4 can be combined with step 3. We adopt this kind of optimization techniques in the implementation described in the next section.

8.4 Experimental results

We implemented our method on the Hitachi SR2201 [43] and evaluated its performance. The SR2201 is a distributed-memory parallel machine with pseudo-vector processing nodes. Each node consists of a RISC processor with a pseudo-vector mechanism [69], which preloads the data from pipelined main memory to on-chip special register bank at a rate of 1 word per cycle. One node has peak performance of 300MFLOPS and 256MB of main memory. The nodes are connected via a multi-dimensional crossbar network, which enables all-to-all communication among P nodes to be done in $P-1$ steps without contention [114].

Our FFT routine is written in FORTRAN and inter-processor communication is done using remote DMA, which enables data stored in the main memory of one node to be transferred directly to the main memory of another node without buffering. The FFT in the x, y and z direction in steps 2, 7 and 9 is performed using Stockham's radix 4 FFT [97], a variant of Algorithm 8.4 which saves both computational work and memory access by computing Y_{L+2} directly from Y_L.

The computational steps of our implementation are illustrated in Fig. 8.1 for the case of $N = 512$, $P = 2$ and $L_1 = L_2 = 2$. Here, the multi-dimensional arrays in Algorithm 8.3 are expressed as three dimensional arrays using the relationship (8.18) and (8.19). The numbers in the first and third three-dimensional arrays correspond to the indices of input sequence f_j and output sequence c_k, respectively. The shaded area represents elements which are allocated to node 0, and the area enclosed by a thick line represents a set of elements used to perform a single FFT in the x, y or z-direction. It is apparent from the figure that (i) the FFTs in each direction can be computed within each node, (ii) there is only one global transposition, and (iii) the input and output data are scattered with a block cyclic distribution of block size 2, as required.



$\{f_j\}$
j_y
j_x
j_z
k_x
k_y
$\{c_k\}$
1. Data input
2. FFT in the x-direction
3. Twiddle factor multiplication (I)
4. Data packing
5. Global transposition
6. Data unpacking
7. FFT in the y-direction
8. Twiddle factor multiplication (II)
9. FFT in the z-direction
10. Data output

Figure 8.1: Computational steps of our FFT routine.

To measure the performance of our FFT routine, we varied the problem size N from 2^{18} to 2^{24} and the number of nodes P from 1 to 16. We set the output block size L_2 equal to the input block size L_1 to reduce the number of experiments and varied $L_1 = L_2$ from 1 to 16. N_x and N_z are determined so that $N_x \geq L_2P$ and $N_z \geq L_1P$ hold and N_y is set to $N/(N_xN_z)$. The ω's used in the FFT and twiddle factor multiplication are pre-computed, so the time for computing them is not included in the execution time to be reported below.

Table 8.1 shows the execution time and the performance obtained when $N = 2^{20}$. We performed three experiments for each set of L_1 and P and took the best value. From these results, we can see that (i) the performance on a single node is 130 MFLOPS, which is more than 40% of the peak performance (300 MFLOPS for one node), (ii) parallel efficiency is extremely high and is more than 94% when $P = 16$, and (iii) the performance does not change significantly with the block sizes. The last point is due to the optimization techniques we have stated in the previous subsection.

Table 8.1: Performance results for the problem of $N = 2^{20}$

$L_1 = L_2$	1 node	2 nodes	4 nodes	8 nodes	16 nodes
1	0.809 s	0.414 s	0.205 s	0.103 s	0.054 s
	129.5 MF	253.3 MF	510.7 MF	1016.7 MF	1920.8 MF
2	0.809 s	0.413 s	0.205 s	0.102 s	0.054 s
	129.6 MF	253.8 MF	512.1 MF	1027.9 MF	1942.7 MF
4	0.809 s	0.413 s	0.205 s	0.102 s	0.053 s
	129.6 MF	253.9 MF	512.2 MF	1027.9 MF	1971.2 MF
8	0.809 s	0.413 s	0.206 s	0.102 s	0.053 s
	129.6 MF	253.9 MF	509.9 MF	1028.2 MF	1964.7 MF
16	0.809 s	0.413 s	0.205 s	0.102 s	0.053 s
	129.6 MF	253.8 MF	510.7 MF	1027.9 MF	1965.0 MF

The performance results when N is varied is shown in Table 8.2 and Fig. 8.2. Note that we were able to perform the FFT of $N = 2^{22}$ points only when $P \geq 2$ and that of $N = 2^{24}$ points only when $P \geq 8$ because of memory limitation. The performance did not depend on the block sizes, so we showed only the results for $L_1 = L_2 = 16$. It is apparent that the performance increases as the problem size grows and reaches 2152 MFLOPS when $N = 2^{24}$ and $P = 16$, which is 45% of the peak performance (4800 MFLOPS for 16 nodes).

From these results, we can conclude that our FFT routine attains high performance on a (pseudo-)vector-parallel machine and flexibility in data distribution at the same time.

8.5 Conclusion

In this paper, we have proposed a 1-dimensional FFT routine for distributed-memory vector-parallel machines which provides the user with both high performance and flexibility in data distribution. Our routine inputs/outputs data using block cyclic data

Table 8.2: Performance results for $L_1 = L_2 = 16$

N	1 node	2 nodes	4 nodes	8 nodes	16 nodes
2^{18}	0.194 s	0.099 s	0.051 s	0.026 s	0.014 s
	121.7 MF	237.5 MF	466.2 MF	899.9 MF	1681.9 MF
2^{20}	0.809 s	0.413 s	0.205 s	0.102 s	0.053 s
	129.6 MF	253.8 MF	510.7 MF	1027.9 MF	1965.0 MF
2^{22}		1.761 s	0.873 s	0.441 s	0.223 s
		262.0 MF	528.3 MF	1046.7 MF	2070.9 MF
2^{24}				1.894 s	0.935 s
				1062.8 MF	2152.5 MF

distribution, and the block sizes for input and output can be specified independently by the user. It is based on the transpose algorithm, which requires only one global data transposition, and no additional inter-processor communication is necessary to realize this flexibility. A straightforward implementation of our method can cause a problem of short innermost loops when the block sizes are large, but we have shown how to solve this by employing loop merging techniques.

We implemented our method on the Hitachi SR2201, a distributed-memory parallel machine with pseudo-vector processing nodes, and obtained the performance of 2152 MFLOPS, or 45% of the peak performance, when transforming 2^{24} points data on 16 nodes. This result was unchanged for a wide range of block sizes from 1 to 16. It should be easy to adapt our method to other similar vector-parallel machines.

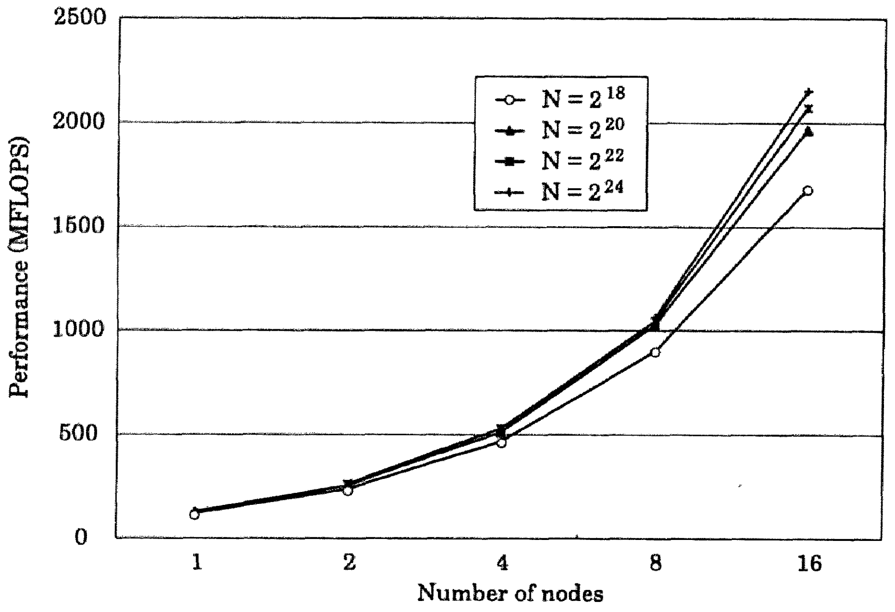


Figure 8.2: Performance results for $L_1 = L_2 = 16$

Chapter 9

Conclusion

9.1 Summary of our study

In this thesis, we studied efficient algorithms for numerical linear algebra on modern high performance architectures such as the shared-memory parallel machine, distributed-memory parallel machine and processors with hierarchical memory. In particular, we focused on three basic linear problems, namely, solution of linear simultaneous equations, eigenvalue problems and the fast Fourier transform, and aimed at developing algorithms that can attain high parallel efficiency, high single-processor performance and high numerical accuracy and stability. The achievements of our study are as follows:

1. In Chapter 3, we proposed the *double-blocked Gaussian elimination method* for distributed-memory parallel machines whose computational nodes consists of processors with hierarchical memory. Compared with conventional parallel blocked Gaussian elimination method, our method has the advantage that it can achieve both high single-processor performance and high parallel efficiency a the same time. We verified the effectiveness of our method on the nCUBE2 and the Hitachi SR2001. The idea of double blocking is useful for other linear algebra problems such as Householder tri-diagonalization and the QR factorization, and in fact, it has been successfully used for Householder tri-diagonalization.

2. In Chapter 4, We designed a sparse direct solver for distributed-memory parallel machines. This solver is designed to deal with sparse symmetric positive definite matrices with 3×3 block nonzero structure, and adopts a locally optimized loop unrolling technique to attain high single-processor performance in the Cholesky factorization part. We implemented our solver on the Hitachi SR2201 and found that our technique can actually improve the performance for a real structural analysis problem.

3. In Chapter 5, we proposed a new parallel direct solver for unsymmetric tridiagonal matrices. In contrast to most of the parallel tridiagonal solver proposed so far, our solver can incorporate partial pivoting for numerical accuracy and stability without sacrificing parallelism. Numerical experiments on the Hitachi SR8000/F1 show that it can achieve speedup of 5.5 times compared with the sequential tridiagonal solver with partial pivoting when the matrix size is 8000 and the number of processor is 8. The experiments also show that accuracy of our method is almost the same as that of the sequential solver and is up to two orders of magnitude better than that of the parallel solver without pivoting.

4. In Chapter 6, we investigated algorithms for tri-diagonalization of a real symmetric matrix on a processor with hierarchical memory. We proposed two improvements for the two-step algorithm of Bishof et al., which can perform tri-diagonalization using only BLAS 3 operations. Numerical experiments on the IBM Power 4 processor show that the resulting algorithm can attain single-processor performance 10 to 15% higher than that of Bishof's algorithm, or about 40% of the peak performance. This is twice the performance of widely used Dongarra's algorithm. We also showed that the accuracy of the two-step algorithm is comparable to that of Dongarra's algorithm.

5. In Chapter 7, we proposed the *Householder inverse iteration method*, a new algorithm for computing the eigenvectors of a real symmetric matrix on a shared-memory parallel machine. Compared with the parallel inverse iteration methods proposed so far, it can reduce the number of interprocessor synchronization in the re-orthogonalization step from $O(N^2)$ to $O(N)$ even when full re-orthogonalization is performed. Numerical experiments on the SR8000 show that our method is up to 3.1 times faster than the conventional inverse iteration when computing all the eigenvectors of matrices of order 1000 to 4000, and still attains the same level of orthogonality of computed eigenvectors.

6. Finall, in Chapter 8, we designed a 1-dimensional FFT program for distributed-memory vector-parallel machines. Our program provides the user with both high performance and flexibility in data distribution, in the sense that the distribution block sizes for input and output can be specified independently by the user. It is based on the transpose algorithm, which requires only one global data transposition, and no additional inter-processor communication is necessary to realize this flexibility. Numerical experiments on the SR2201 shows that it can attain 45% of the peak performance when transforming 2^{24} points data on 16 nodes. This result was unchanged for a wide range of block sizes from 1 to 16.

We believe that the algorithms we have developed in this thesis are applicable to a wide range of problems arising in science and engineering and will contribute to the progress of these fields by providing them with a means to use modern high performance computers more efficiently.

9.2 Future work

We are planning to extend our study along the following three directions:

- Application to real problems

  In this thesis, we have verified the effectiveness of our algorithms on a number of test problems. However, it remains our future work to integrate our solvers into real applications and evaluate their performance there. In particular, we are interested in solving large eigenvalue/eigenvector problems arising in electronic structure calculations and information retrieval by collaborating with researchers in these fields.

- Development of an automatically-tuned library

  As architectures of high performance computers become more complex and diverse, the cost of developing an optimized linear algebra library for each architecture is growing prohibitively expensive. To solve this problem, several automatically-tuned libraries have been proposed [11][61][62][65]. We are interested in combining the ideas presented in these studies with our algorithm to develop an automatically-tuned library that can attain high performance on a wide range of machines.

- Guaranteeing the accuracy of solution

  As the size of the problems grows, it becomes more and more important to guarantee the accuracy of the computed solution. Recently, simple and efficient methods for finding rigorous error bounds have been proposed for linear simultaneous equations [72][73] and eigenvalue problems [74]. It will be rewarding to investigate the applicability of these ideas to our algorithms.

By combining the results obtained by following these paths, we hope to provide a more efficient, easy-to-use and reliable library for numerical linear algebra.

Bibliography

[1] R. C. Agarwal and J. W. Cooley: Vectorized Mixed Radix Discrete Fourier Transform Algorithms, *Proc. of IEEE*, Vol. 75, No. 9, pp. 1283–1292 (1987).

[2] R. C. Agarwal, F. G. Gustavson and M. Zubair: A High Prformance Parallel Algorithm for 1-D FFT, *Proc. of Supercomputing '94*, pp. 34–40 (1994).

[3] N. Akita: Solution of Three-Term Equations on Vector Processors, *Proceedings of the Annual Meeting of Information Processing Society of Japan*, pp. 1305–1306 (1984).

[4] P. Amestoy, I. S. Duff and J. -Y. L'Excellent: MUMPS MUltifrontal Massively Parallel Solver Version 2.0, Technical Report TR/PA/98/02, CERFACS (1998).

[5] P. Amestoy and I. S. Duff: The PARASOL Project and the Multifrontal Parallel Solver for Sparse Systems, *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, SIAM, Philadelphia, 1999.

[6] P. Amestoy, I. Duff and J. -Y. L'Excellent: Multifromtal Parallel Distributed Symmetric and Unsymmetric Solvers, *Computational Methods in Applied Mechanics and Engineering*, Vol. 184, pp. 501–520 (2000).

[7] P. Amodio and L. Brugnano: The Parallel QR Factorization Algorithm for Tridiagonal Linear Systems, *Parallel Computing*, Vol. 21, pp. 1097–1110 (1995).

[8] E. Anderson, Z. Bai, C. Bishof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen: *LAPACK Users' Guide*, second edition, SIAM, Philadelphia, 1995.

[9] C. Ashcraft, S. C. Eisenstat and J. W. -H. Liu: A Fan-in Algorithm for Distributed Sparse Numerical Factorization, *SIAM Journal on Scientific and Statistical Computing*, Vol. 11, No. 3, pp. 593–599 (1990).

[10] C. Ashcraft, C. Eisenstat, J. W. -H. Liu and A. Sherman: A Comparison of Three Column Based Distributed Sparse Factorization Schemes, in *Proceedings of the Fifth*

SIAM Conference on Parallel Processing for Scientific Computing, SIAM, Philadelphia, 1991.

[11] ATLAS homepage: http://math-atlas.sourceforge.net/

[12] A. Averbuch, E. Gabber, B. Gordissky and Y. Medan: A Parallel FFT on a MIMD Machine, *Parallel Computing*, Vol. 15, pp. 61–74 (1990).

[13] D. H. Bailey: FFTs in External or Hierarchical Memory, *The Journal of Supercomputing*, Vol. 4, pp. 23–35 (1990).

[14] M. W. Berry and M. Browne: *Understanding Search Engines*, SIAM, Philadelphia, 1999.

[15] C. Bishof and C. F. van Loan: The WY Representation for Products of Householder Matrices, *SIAM Journal on Scientific and Statistical Computing*, Vol. 8, No. 1, pp. s2–s13 (1987).

[16] C. Bishof, M. Marques and X. Sun: Parallel Bandreduction and Tridiagonalization, Technical Report 8, *PRISM Working Note*, 1993. http://www-unix.mcs.anl.gov/prism/lib/tech.html

[17] C. Bishof, B. Lang and X. Sun: Parallel Tridiagonalization through Two-step Band Reduction, Technical Report 17, *PRISM Working Note*, 1994. http://www-unix.mcs.anl.gov/prism/lib/tech.html.

[18] 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 Users' Guide*, SIAM, Philadelphia, 1997.

[19] T. Boku, Y. Iwasaki, H. Nakamura and K. Nakazawa: The Architecture of Massively Parallel Processor CP-PACS, *Proceedings of the 2nd Aizu International Symposium on Parallel Algorithms/Architectures Synthesis (pAs '97)*, pp. 31–41, IEEE Computer Society, 1997.

[20] E. G. Boman and B. Hendrickson: A Multilevel Algorithm for Reducing the Envelope of Sparse Matrices, Technical Report SCCM-96-14, Stanford University (1996).

[21] D. A. Carlson: Ultrahigh-Performance FFTs for the Cray-2 and Cray Y-MP Supercomputers, *Journal of Supercomputing*, Vol. 6, pp. 107–116 (1992).

[22] H. Chang, S. Utku, M. Sakama and D. Rapp: A Parallel Householder Tridiagonalization Stratagem Using Scattered Row Decomposition, *International Journal on Numerical Methods in Engineering*, Vol. 26, p. 857–874 (1988).

[23] H. Y. Chang, S. Utku, M. Salama and D. Rapp, D: A Parallel Householder Tri-diagonalization Strategem using Scattered Square Decomposition, *Parallel Computing*, Vol. 6, No. 3, pp. 297–311 (1988).

[24] J. Choi et. al.: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance, *LAPACK Working Notes* 95, 1995.

[25] J. W. Cooley and J. W. Tukey: An Algorithm for the Machine Calculation of Complex Fourier Series, *Mathematics of Computation*, Vol. 19, pp. 297–301 (1965).

[26] I. Crawford and K. Wadleigh: *Software Optimization for High Performance Computing: Creating Faster Applications*, Prentice-Hall, 2000.

[27] D. E. Culler, J. P. Singh and A. Gupta: *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, 1998.

[28] J. J. M. Cuppen: A Divide and Conquer Method for the Symmetric Tri-diagonal Eigenproblem', *Numerische Mathematik*, Vol. 36, pp. 177–195 (1981).

[29] T. A. Davis, P. Amestoy and I. S. Duff: An Approximate Minimum Degree Ordering Algorithm, *SIAM Journal on Matrix Analysis and Applications*, Vol. 17, No. 4, pp. 886–905 (1996).

[30] J. W. Demmel: *Applied Numerical Linear Algebra*, SIAM, Philadelphia, 1997.

[31] I. Dhillon: *A New $O(n^2)$ Algorithm for the Symmetric Tri-diagonal Eigenvalue/Eigenvector Problem*, Ph. D. Thesis, Computer Science Division, University of California, Berkeley, 1997.

[32] S. Domas, F. Desprez and B.Tourancheau: Optimization of the ScaLAPACK LU Factorization Routine Using Communication/Computation Overlap, in *Proceedings of the Euro-Par '96 Parallel Processing*, Lecture Notes in Computer Science, Vol. 1124, pp. 3–10, Springer-Verlag, August, 1996.

[33] J. J. Dongarra and D. C. Sorensen: A Fully Parallel Algorithm for the Symmetric Eigenvalue Problem, *SIAM Journal on Scientific and Statistical Computing*, Vol. 8, No. 2, pp. s139–s154 (1987).

[34] J. J. Dongarra, J. D. Croz, S. Hammarling and R. J. Hanson: An Extended Set of Fortran Basic Linear Algebra Subprograms, *ACM Transactions on mathematical Software*, Vol. 14, No. 1, pp. 1–17 (1988).

[35] J. J. Dongarra, J. D. Croz, S. Hammerling and I. Duff: A Set of Level 3 Basic Linear Algebra Subprograms, *ACM Transactions on Mathematical Software*, Vol. 16, No. 1, pp. 1–17 (1990).

[36] J. J. Dongarra and R. A. van de Geijn: Reduction to Condensed Form for the Eigenvalue Problem on Distributed Architectures, *Parallel Computing*, Vol. 18, No. 9, pp. 973–982 (1992).

[37] J. J. Dongarra, I. S. Duff, D. C. Sorensen and H. A. van der Vorst: *Numerical Linear Algebra on High-Performance Computers*, SIAM, Philadelphia, 1998.

[38] A. Dubey, M. Zubair and C. E. Grosch: A General Purpose Subroutine for Fast Fourier Transform on a Distributed Memory Parallel Machine, *Parallel Computing*, Vol. 20, pp. 1697–1710 (1994).

[39] P. Dubois and G. Rodrigue: An Analysis of the Recursive Doubling Algorithm, in D. J. Kuck and A. H. Sameh, eds., *High Speed Computer and Algorithm Organization*, Academic Press, New York (1977).

[40] I. S. Duff and J. K. Reid: The Multifrontal Solution of indefinite Sparse Symmetric Linear Equations, *ACM Transactions on Mathematical Software*, Vol. 9, pp. 302–325 (1983).

[41] I. S. Duff, A. M. Erisman and J. K. Reid: *Direct Methods for Sparse Matrices*, Oxford University Press, Oxford, 1986.

[42] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker: *Solving Problems on Concurrent Processors*, Vol. I, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[43] H. Fujii, Y. Yasuda, H. Akashi, Y. Inagami, M. Koga, O. Ishihara, M. Kashiyama, H. Wada and T. Sumimoto: Architecture and Performance of the Hitachi SR2201 Massively Parallel Processor System, *Proceedings of IPPS '97*, pp. 233–241, 1997.

[44] K. A. Gallivan et. al.: *Parallel algorithms for Matrix Computations*, SIAM, Philadelphia, 1990.

[45] A. George and J. W. H. Liu: *Computer Solution of Large Sparse Positive Definite Systems*, Prentice Hall, 1981.

[46] G. H. Golub and C. F. van Loan: *Matrix Computations*, 3rd edition, The Johns Hopkins University Press, Baltimore, 1996.

[47] M. Gu and S. Eisenstat: A Divide-and-Conquer Algorithm for the Symmetric Tri-diagonal Eigenproblem, *SIAM Journal on Matrix Analysis and Applications*, Vol. 16, pp. 172–191 (1995).

[48] A. Gupta and V. Kumar: Parallel Algorithms for Forward and Backward Substitution in Direct Solution of Sparse Linear Systems, *Proceedings of the Supercomputing '95*, Dec. 1995.

[49] A. Gupta, G. Karypis and V. Kumar: Highly Scalable Parallel Algorithms for Sparse Matrix Factorization, *IEEE Transactions for Parallel and Distributed Systems*, Vol. 8, No. 5, pp. 502–520 (1997).

[50] M. T. Heath, E. Ng and B. W. Payton: Parallel Algorithms for Sparse Linear Systems, *SIAM Review*, Vol. 33, pp. 420–460 (1990).

[51] M. T. Heath and P. Raghavan: Performance of a Fully Parallel Sparse Solver, *The International Journal of Supercomputer Applications and High Performance Computing*, Vol. 11, No. 1, pp. 49–64 (1997).

[52] M. Hegland: On the Parallel Solution of Tridiagonal Systems by Wrap-around Partitioning and Incomplete LU Factorization, *Numerische Mathematik*, Vol. 59, No. 5, pp. 453–472 (1991).

[53] M. Hegland: Real and Complex Fast Fourier Transforms on the Fujitsu VPP500, *Parallel Computing*, Vol. 22, pp. 539–553 (1996).

[54] D. Heller: Some Aspects of the Cyclic Reduction Algorithm for Block Tridiagonal Linear Systems, *SIAM J. Numer. Anal.*, Vol. 13, No. 4, pp. 484–496 (1976).

[55] B. Hendrickson, E. Jessup and C. Smith: Toward an Efficient Parallel Eigensolver for Dense Symmetric Matrices, *SIAM Jounal on Scientific Computing*, Vol. 20, No. 3, pp. 1132–1154 (1999).

[56] N. J. Higham: *Accuracy and Stability of Numerical Algorithms*, SIAM, Philadelphia, 1996.

[57] S. L. Johnson and R. L. Krawitz: Cooley-Tukey FFT on the Connection Machine, *Parallel Computing*, Vol. 18, pp. 1201–1221 (1992).

[58] G. Karypis and V. Kumar: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs, *SIAM Journal on Scientific Computing*, Vol. 20, No. 1, pp. 359–392 (1998).

[59] T. Katagiri and Y. Kanada: Performance Evaluation of Blocked Householder Algorithm on Distributed Memory Parallel Machine, *Transactions of the Information Processing Society of Japan*, Vol. 39, No. 7, pp. 2391–2394 (1998), in Japanese.

[60] T. Katagiri and Y. Kanada: A Parallel Implementation of Eigensolver and Its performance, presented at *Ninth SIAM Conference on Parallel Processing for Scientific Computing* (1999).

[61] T. Katagiri, H. Kuroda and Y. Kanada: A Methodology for Automatically Tuned Parallel Tridiagonalization on Distributed Memory Vector Parallel Machines, *Proceedings of Vector and Parallel Processing 2000*, pp. 265–277, Portugal, June 2000.

[62] T. Katagiri, H. Kuroda K. Ohsawa and Y. Kanada: I-LIB: An Automatically Tuned Parallel Numerical Library and Its Performance Evaluation, *Proceedings of JSPP (Joint Symposium on Parallel Processing) 2000*, pp.27–34, 2000, in Japanese.

[63] T. Katagiri: *A Study on Large Scale Eigensolvers for Distributed Memory Parallel Machines*, Ph. D. Thesis, Information Science Division, University of Tokyo, 2001.

[64] V. Kumar, A. Grama, A. Gupta and G. Karypis: *Introduction to Parallel Computing*, The Benjamin/Cummings Publishing Company, CA, 1994.

[65] H. Kuroda T. Katagiri, and Y. Kanada: Performance of Automatically Tuned Parallel GMRES(m) method on Distributed Memory Machines, *Proceedings of Vector and Parallel Processing 2000*, pp. 251–264, Portugal, June 2000.

[66] LINPACK Benchmark homepage: http://www.top500.org/lists/linpack.php

[67] J. W. H. Liu: The Multifrontal Method for Sparse Matrix Solution: Theory and Practice, *SIAM Review*, Vol. 34, No. 1, pp. 82–109 (1992).

[68] K. Murata, R. Oguni and Y. Karaki: *Supercomputers: Applications to Scientific and Engineering Computing*, Maruzen, Tokyo, 1985.

[69] K. Nakazawa, H. Nakamura, H. Imori and S. Kawabe: Pseudo Vector Processor Based on Register-Windowed Superscalar Pipeline, *Proceedings of Supercomputing '92*, pp. 642–651 (1992).

[70] K. Naono, Y. Yamamoto, M. Igai and H. Hirayama: High Performance Imlementation of Tri-diagonalization on the SR8000, *Proceedings of HPC-ASIA2000*, Vol. I, pp. 206–219, IEEE Computer Society, 2000.

[71] K. Naono, Y. Yamamoto, M. Igai, H. Hirayama and N. Ioki: A Multi-color Inverse Iteration for a High Performance Real Symmetric Eigensolver, in Ludwig, B. and Wismuller, K. (eds.), *Proc. of Euro-Par 2000*, Lecture Notes in Computer Science 1900, pp. 527–531, Springer-Verlag, 2000.

[72] T. Ogita, S. Oishi and Y. Ushiro: Fast Verification of Solutions for Sparse Monotone Matrix Equations, *Computing*, Supplement 15, pp. 175–187 (2001).

[73] T. Ogita, S. Oishi and Y. Ushiro: Computation of Sharp Rigorous Componentwise Error Bounds for the Approximate Solutions of Systems of Linear Equations, *Reliable Computing*, Vol. 9, No. 3, pp. 229–239 (2003).

[74] S. Oishi: Fast Enclosure of Matrix Eigenvalues and Singular Values via Rounding Mode Controlled Computation, *Linear Algebra and its Applications*, Vol. 324, pp. 133–146 (2001).

[75] PARASOL homepage: http://www.parallab.uib.no/parasol

[76] B. N. Parlett: *The symmetric Eigenvalue Problem*, SIAM, Philadelphia, 1998.

[77] A. Pothen, H. Simon and K. Liou: Partitioning Sparse Matrices with Eigenvectors of Graphs, *SIAM Journal on Matrix Analysis and Applications*, Vol. 11, pp. 430–452.

[78] PSPACES homepage: http://www-users.cs.umn.edu/ mjoshi/pspaces

[79] C. H. Reinsch: Smoothing by Spline Functions, *Numerische Mathematik*, Vol. 10, pp. 177–183 (1967).

[80] S. H. Roosta: *Parallel Processing and Parallel Algorithms: Theory and Computation*, Springer-Verlag, 2000.

[81] E. Rothberg and A. Gupta: An Evaluation of Left-looking, Right-looking and Multifrontal Approaches to Sparse Cholesky Factorization on Hierarchical Memory Machines, *International Journal of High Speed Computing*, Vol. 5, pp. 537–593 (1993).

[82] E. Rothberg and A. Gupta: An Efficient Block-oriented Approach to Parallel Sparse Cholesky Factorization, *SIAM Journal on Scientific Computing*, Vol. 15, No. 6, pp. 1413–1439 (1993).

[83] E. Rothberg: Performance of panel and Block Approaches to Parallel Sparse Cholesky Factorization on the iPSC/860 and paragon Multicomputers, *SIAM Journal on Scientific Computing*, Vol. 17, No. 3, pp. 699–713 (1996).

[84] S. A. Salvini and L. S. Mulholland: The NAG FORTRAN Library, *Proceedings of the Ninth SIAM Conference on Parallel Processing and Scientific Computing*, SIAM, Philadelphia, 1999.

[85] H. Samukawa: A Parallel Tridiagonal Solver Based on ETC Ordering, *Proceedings of JSPP (Joint Symposium on Parallel Processing) 2000*, pp. 83–90 (2000), in Japanese.

[86] R. S. Schreiber and B. N. Parlett: Block Reflectors: Theory and Compuation, *SIAM Journal on Numerical Analysis*, Vol. 25, pp. 189–205 (1987).

[87] R. S. Schreiber and C. F. van Loan: A Storage-efficient WY Representation for Products of Householder Transformations, *SIAM Journal on Scientific and Statistical Computing*, Vol. 10, pp. 52–57 (1989).

[88] K. S. Stanley: *Execution Time of Symmetric Eigensolvers*, Ph. D Thesis, Computer Science Division, University of California at Berkeley, 1997.

[89] G. W. Stewart: *Matrix Algorithms, Vol. I: Basic Decompositions*, SIAM, Philadelphia, 1998.

[90] G. W. Stewart: *Matrix Algorithms, Vol. II: Eigensystems*, SIAM Philadelphia, 2001.

[91] H. S. Stone: An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations, *J. Assoc. Comput. Mach.*, Vol. 20, pp. 27–38 (1973).

[92] K. Sumiyoshi and T. Ebisuzaki: Performance of Parallel Solution of a Block Tridiagonal Linear System on Fujitsu VPP 500, *Parallel Computing*, Vol. 24, pp. 287–304 (1998).

[93] P. N. Swarztrauber: Multiprocessor FFTs, *Parallel Computing*, Vol. 5, pp. 197–210 (1987).

[94] D. Takahashi: Parallel FFT Algorithms for the Distributed-Memory Parallel Computer Hitachi SR8000, *Proc. of JSPP2000*, pp. 91–98, 2000 (in Japanese).

[95] Y. Tamaki, N. Sukegawa, M. Ito, Y. Tanaka, M. Fukagawa, T. Sumimoto, and N. Ioki: Node Architecture and Performance Evaluation of the Hitachi Super Technical Server SR8000, *Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems*, pp. 487–493 (1999).

[96] F. Tisseur and J. J. Dongarra: A Parallel Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem on Distributed Memory Architectures, *SIAM Journal on Scientific Computing*, Vol. 20, No. 6, pp. 2223–2236 (1999).

[97] C. Van Loan: *Computational Frameworks for the Fast Fourier Transform*, SIAM, Philadelphia, 1992.

[98] R. S. Varga: *Matrix Iterative Analysis*, Prentice-Hall, 1962.

[99] J. H. Wilkinson: *Rounding Errors in Algebraic Processes*, Her Majesty's Stationary Office, London, 1963.

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

[101] J. H. Wilkinson. and C. Reinsch(eds.): *Linear Algebra*, Springer Verlag, 1971.

[102] B. Wilkinson, C. M. Allen and M. Allen: *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice-Hall, 1998.

[103] Y. Yamamoto and T. Fujiwara: Numerical Stabilization of the First-Principles Molecular-Dynamics Method for Metals, *Physical Review B*, Vol. 46, No. 20, pp. 13596–13598 (1992).

[104] Y. Yamamoto and T. Okochi: Optimization of the Gaussian Elimination Method for Massively Parallel Processors, *Proceedings of JSPP (Joint Symposium on Parallel Processing) '95*, pp. 217–224 (1995), in Japanese.

[105] Y. Yamamoto and T. Okochi: A New Variant of the Gaussian Elimination Method Optimized for RISC-based Distributed-Memory Parallel Processors, presented at *HPC-Asia '95*, Taipei, Oct. 1995.

[106] Y. Yamamoto, M. Igai and K. Naono: Development and Evaluation of a Sparse Direct Solver for Distributed-Memory Parallel Processors, *Proceedings of the Riken Symposium on Linear Algebra and its Applications*, The Institute of Physical and Chemical Research (Riken), Nov. 1999.

[107] Y. Yamamoto, M. Igai and K. Naono: Development and Evaluation of a Direct Solver for Sparse Symmetric Systems on Distributed-Memory Parallel Processors, *Transactions of the Information Processing Society of Japan*, Vol. 41, No. 5, pp. 1567–1576 (2000), in Japanese.

[108] Y. Yamamoto, M. Igai and K. Naono: A New Algorithm for Accurate Computation of Eigenvectors on Shared-Memory Parallel Processors and its Evaluation on the SR8000, *Journal of the Information Processing Society of Japan*, Vol. 42, No. 4, pp. 771–778 (2001), in Japanese.

[109] Y. Yamamoto, M. Igai and K. Naono: A Parallel Linear Equation Solver for Nonsymmetric Tridiagonal Matrices, *Journal of the Information Processing Society of Japan*, Vol. 42, No. SIG9 (HPS), pp. 19–27 (2001), in Japanese.

[110] Y. Yamamoto, M. Igai and K. Naono: A Vector-Parallel FFT with a User-Specifiable Data Distribution Scheme, in M. Guo and L. T. Yang, eds., *Parallel and Distributed Processing and Applications*, Lecture Notes in Computer Science 2745, Springer-Verlag, pp. 362–374, 2003.

[111] Y. Yamamoto, M. Igai and K. Naono: A Parallel Direct Linear Equation Solver for Nonsymmetric Tridiagonal Matrices, *Proceedings of the SIAM Conference on Applied Linear Algebra*, Williamsburg, VA, July 2003.

[112] Y. Yamamoto, M. Igai and K. Naono: A New BLAS-3 Based Parallel Algorithm for Computing the Eigenvectors of Real Symmetric Matrices, in L. T. Yang and Y. Pan, eds., *High Performance Scientific and Engineering Computing - Hardware/Software Support*, Kluwer Academic Publishers, to appear in Oct. 2003.

[113] Y. Yamamoto, M. Igai and K. Naono: Vector-Parallel Algorithms for 1-Dimensional Fast Fourier Transform, in M. Guo and L. T. Yang, eds., *Parallel/Distributed processing with applications*, Kluwer Academic Publishers, to appear in Dec. 2003.

[114] Y. Yasuda, H. Fujii, H. Akashi, Y. Inagami, T. Tanaka, J. Nakagoshi, H. Wada and T. Sumimoto: Deadlock-Free Fault-Tolerant Routing in the Multi-Dimensional Crossbar Network and its Implementation for the Hitachi SR2201, *Proceedings of IPPS '97*, pp. 346–352, 1997.