JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **38,** 28–36 (1996)
ARTICLE NO. 0126

Parallelizing a Level 3 BLAS Library for LAN-Connected Workstations

KUO-CHAN HUANG, FENG-JIAN WANG, AND PEI-CHI WU[1]

Department of Computer Science and Information Engineering, National Chiao Tung University, 1001 Ta-Hsueh Road, Hsinchu, Taiwan, Republic of China

LAN-connected workstations are a heterogeneous environment, where each workstation provides time-varying computing power, and thus dynamic load balancing mechanisms are necessary for parallel applications to run efficiently. Parallel basic linear algebra subprograms (BLAS) have recently shown promise as a means of taking advantage of parallel computing in solving scientific problems. Most existing parallel algorithms of BLAS are designed for conventional parallel computers; they do not take the particular characteristics of LAN-connected workstations into consideration. This paper presents a parallelizing method of Level 3 BLAS for LAN-connected workstations. The parallelizing method makes dynamic load balancing through *column-blocking* data distribution. The experiment results indicate that this dynamic load balancing mechanism really leads to a more efficient parallel level 3 BLAS for LAN-connected workstations. © 1996 Academic Press, Inc.

1. INTRODUCTION

Since 1979 researchers have developed three levels of basic linear algebra subprograms (BLAS) [15]. One major purpose of these subprograms is to facilitate the development of scientific applications. BLAS are used as basic building blocks for developing more complex and higher level linear algebraic subprograms. For instance, Level 1 BLAS deal with vector-to-vector operations, Level 2 [7, 8] handle vector-to-matrix operations, and Level 3 [9, 10] do various matrix-to-matrix operations. Many hardware vendors have provided their own implementation of BLAS optimized according to specific hardware features. With the help of these BLAS, users can now spend less efforts in developing application software, even with better performance. Recently, parallel BLAS have been increasingly employed to speed up scientific computations. Several parallel libraries have been developed on different platforms [2, 3, 11, 14, 17].

LAN-connected workstations have been getting widespread use and are considered a cost-effective parallel computing environment for many applications. LAN-connected workstations could be a heterogeneous environment composed of workstations with different configurations or even different architectures. The situation may be even more complicated since each workstation may be shared by more than one user at a time and the user number for a workstation might change as time goes on. Therefore, each workstation would provide unequal and time-varying computing power to a parallel application. Computing models are different in LAN-connected workstations and conventional parallel computers, where a user may allocate a group of processors for dedicated use in a time period. Dynamic load balancing mechanisms are necessary for parallel applications to run efficiently on LAN-connected workstations. However, most existing parallel implementations of BLAS were designed based on conventional parallel computers, without concern for the need for dynamic load balancing capabilities on LAN-connected workstations.

This paper explores the parallelization of Level 3 BLAS for LAN-connected workstations taking dynamic load balancing into consideration. Dynamic load balancing has received a great deal of attention in the literature [5, 6, 18, 19]. The basic idea of dynamic load balancing is to balance the workloads on all processors to increase the system throughput or reduce the wall clock execution time of an application. One approach is to allow the load to be migrated from heavy nodes to light ones. Our approach divides a task into subtasks whose number is larger than the processors' and then assigns each processor one subtask at a time. Only when a processor completes its subtask is it assigned a new one. The approach achieves dynamic load balancing through proper data partition and task assignment. We have conducted several experiments to investigate various data partition methods. According to the results of experiments which will be described in detail in later sections, we propose the *dynamic column-blocking* method as the best data partition method to run parallel Level 3 BLAS efficiently on LAN-connected workstations. We have implemented a parallel LU factorization routine using our parallel Level 3 BLAS to show the effectiveness of our parallelizing method and discuss the issue of parallel library interface.

In the next section we give high level descriptions of parallel implementations of Level 3 BLAS. Section 3 discusses the importance, advantages, and issues of LAN-connected workstations. Section 4 presents various data distribution methods, our experiments and comparative analysis. Section 5 discusses the implementation of LU

[1] E-mail: {kchuang, pcwu, fjwang}@csie.nctu.edu.tw.

factorization using our parallel Level 3 BLAS. Conclusions and future work are presented in Section 6.

2. PARALLEL LEVEL 3 BLAS

The following is a list of the Level 3 BLAS, where α and β are scalars; A, B, and C are matrices; and op means transpose or conjugate transpose:

- GEMM forms the matrix–matrix product

$$C \leftarrow \alpha \times \mathrm{op}(A)\mathrm{op}(B) + \beta \times C.$$

- SYMM forms the matrix–matrix product

$$C \leftarrow \alpha \times AB + \beta \times C$$
$$\text{or } C \leftarrow \alpha \times BA + \beta \times C.$$

- SYRK forms the symmetric rank-k update

$$C \leftarrow \alpha \times AA^{\mathrm{T}} + \beta \times C$$
$$\text{or } C \leftarrow \alpha \times A^{\mathrm{T}}A + \beta \times C.$$

- SYR2K forms the symmetric rank-$2k$ update

$$C \leftarrow \alpha \times AB^{\mathrm{T}} + \alpha \times BA^{\mathrm{T}} + \beta \times C$$
$$\text{or } C \leftarrow \alpha \times A^{\mathrm{T}}B + \alpha \times B^{\mathrm{T}}A + \beta \times C.$$

- TRMM forms the matrix–matrix product

$$B \leftarrow \alpha \times \mathrm{op}(A)B$$
$$\text{or } B \leftarrow \alpha \times B\mathrm{op}(A).$$

(where A is a triangular matrix).
- TRSM forms the matrix–matrix product

$$B \leftarrow \alpha \times \mathrm{op}(A^{-1})B$$

(solves a system of equations with the same coefficient matrix but several right-hand sides, where A is a triangular matrix)

$$\text{or } B \leftarrow \alpha \times B\mathrm{op}(A^{-1}).$$

Each Level 3 BLAS routine, in one form or another, represents a matrix–matrix product. To parallelize the Level 3 BLAS efficiently, our approach constructed a building block of matrix–matrix products, an efficient parallel implementation, for every Level 3 BLAS routine.

3. LAN-CONNECTED WORKSTATIONS

3.1. Computing Scenarios in LAN-Connected Workstations

Scientific and engineering programs are getting more and more complex and involve larger and larger amounts of data. The result is an ever increasing need for computing power. However, few small-to-medium-size organizations and enterprises can afford an expensive supercomputer. In these organizations, PCs and workstations are usually chosen for personal computing or data processing. As the downsizing trend continues, more and more powerful PCs and workstations are appearing in ordinary organizations and enterprises. These PCs and workstations are usually connected by LANs for sharing resources, such as printers and file servers. Such working environments show new opportunities for parallel computing. The following scenarios show the practicality of LAN-connected environments:

1. An engineer uses a workstation for product design and document preparation. When a design is drafted, a simulation is performed to test or verify the design. Although such simulations usually take a long time, e.g., 1–2 days, it can be speeded up considerably by utilizing idle workstations connected by the organization's LAN.
2. Managers often use PCs for writing notes or proposals. If, however, they want to employ artificial intelligence for data analysis or decision making, the AI application may take up to 30 min. Calculation must be speeded up to a reasonable response time of, say, 30 s. A speedup of 60 is needed.

Although LAN-connected workstations are not as efficient as supercomputers, the above scenarios nonetheless point out how effective they could be: daily work, personal computing, and parallel computing can be integrated in a heterogeneous LAN-connected system of PCs or workstations. This results in an environment that saves money and simplifies system management and maintenance.

3.2. Issues in Developing a Parallel Library for LAN-Connected Workstations

Several parallel libraries of linear algebra, e.g., *ScaLAPACK* [13], already exist. Most of them are designed for conventional parallel computers, such as hypercube machines. They can be ported to LAN-connected workstations if they use a portable communication library, such as PVM. However, the ported libraries in general do not perform efficiently in such a divergent environment. A new design is thus needed for these libraries.

There are at least two differences between LAN-connected workstations and traditional parallel computers that must be considered when designing an efficient parallel library:

1. *Communication resources.* Only one message can be transmitted on an Ethernet (bus) at any given time. In a hypercube system, however, since each node contains multiple physical communication links, more than one message can be transmitted in parallel. Also, the communication links in LAN-connected workstations usually contain less communication bandwidth but higher latency, compared with those in traditional parallel computers. The associated high communication cost in LAN-connected workstations might limit the advantages of parallelism for applications.

2. *Shared computing environment.* When someone runs a parallel program on a hypercube machine, he or she can allocate a number of dedicated processors of equal computing power. In most organizations and enterprises, however, workstations are not used solely for parallel computing. Workstations are usually used for several jobs concurrently, and the workload on each machine changes dynamically. A parallel program on such a system cannot allocate a group of workstations that have equal workloads. Hence, dynamic load balancing is needed to help achieve higher performance and increase system utilization.

4. PARALLEL MATRIX MULTIPLICATION IN LAN-CONNECTED WORKSTATIONS

This section presents the parallel implementation of the matrix–matrix product, the building block of our parallel Level 3 BLAS library. We explored two data partition methods, the *square-blocking* and the *column-blocking* methods. To find the best parallel implementation with dynamic load balancing, experiments for these two methods with both static and dynamic data allocation were conducted. Our implementation was done using C language and *PVM* [12, 16]. The scheduling of matrix computations was controlled by one host process and the multiplications were done with multiple slave processes running on different machines.

4.1. Sequential Implementation

In this subsection, we investigate a set of sequential implementations of matrix multiplication and use their best as the basis of comparisons for parallel implementations in the following subsections. Matrix multiplication is a simple operation: a straightforward implementation of about 10 lines of C code has a time complexity of $O(n^3)$ for n by n matrices. For large matrices, the performance of matrix multiplication is greatly affected by the locality of data references. The way a very large array (e.g., a matrix of 2M bytes) is accessed improves the hit ratio of caches a lot in computers with memory hierarchies (cache memory). Block algorithms have been proposed to take advantage of modern hierarchical memory systems [9, 10, 15]. Most researchers in this field now agree that block algorithms can lead to better performance in modern computers that have hierarchical memory systems [7, 8, 15]. Below is an example of matrix multiplication, $C_{m\times p} = A_{m\times n} \times B_{n\times p}$:

```
double A[m][n], B[n][p], C[m][p];
int i, j, k;
double f;
```

There are five different implementations of matrix multiplication:

1. *Loop-i-j-k.* The method is used in many elementary linear algebra textbooks. It is straightforward but performs poorly in modern computers.

```
for (i=0; i<m; i++)
 for (j=0; j<p; j++)
 {
  f=0;
  for (k=0; k<n; k++)
   f += A[i][k] * B[k][j];
  C[i][j] = f;
}
```

2. *Loop-reordering.* This method reorders the nested loops of k and j in the previous one to improve the locality of references.

```
for (i=0; i<m; i++)
 for (k=0; k<n; k++)
 {
  f = A[i][k];
  for (j=0; j<p; j++)
   C[i][j] += f * B[k][j];
}
```

3. *Matrix–column.* This method divides the matrix–matrix multiplication into a set of matrix–vector multiplication, where each multiplies matrix A with a column of matrix B to produce a column of matrix C.

4. *Block algorithm without submatrices copy.* Both this method and the following one are block algorithms. They divide a large matrix into smaller submatrices and then perform matrix multiplication on these submatrices. This algorithm operates in place on the submatrices of A and B. When two submatrices are multiplied, the matrix–column method is adopted.

5. *Block algorithm with submatrices copy.* This method copies the submatrices of A and B into a temporary area.

Table I shows the results of our test on these five implementations. The matrices A, B, and C are 500×500 in size and consist of double-precision floating-point numbers. They are initialized as 1.1. We used a SPARC station 2 as our platform, and the code was compiled with the GNU C compiler. The time was measured by `time()`. Obviously, the upper algorithms are less efficient. The block algorithm with submatrices copy performs best.

The above results may be explained through analysis of the cache locality of the memory references. The SPARC station 2 is equipped with a direct mapping cache of 64K bytes, where each cache line contains 16 bytes, i.e., two

TABLE I
Performance Test on Five Versions of Matrix Multiplication

Algorithm	Exec. time (s)	Speed rating	Cache misses
loop-i-j-k	108	1.00	125,250,000
loop-reordering	87	1.24	125,125,000
matrix-column	76	1.42	62,750,000
block w/o copy	70	1.54	4,000,000
block with copy	63	1.71	2,875,000

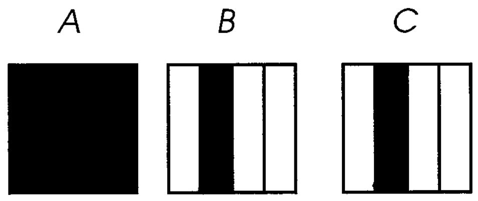
double-precision floating-point numbers in SPARC. Consider the cache misses when accessing the matrices. The block algorithms here are of block size 50 × 50. A cache system usually moves a cache line of data from the main memory to the cache when a cache miss occurs. If the data needed next are in the same block, the CPU can obtain the data from the cache instead of from the main memory, thereby reducing the data access time. Different algorithms use different memory access patterns, and thus need different execution time. The fourth column in Table I shows the predicted number of cache misses for the five implementations of matrix multiplication.

Another factor that affects the performance is the use of registers for scalar variables. For example, in the matrix–column method, a register holds the temporary result from a sequence of add operations. The matrix–column method accesses memory n times when producing a column of length n. In contrast, the loop-reordering method needs n^2 memory accesses to produce a column of length n. The matrix–column method thus outperforms the loop-reordering method.

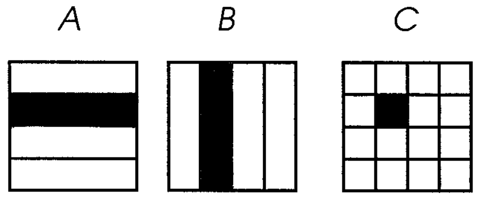
4.2. Static Data Allocation

The column-blocking method and the square-blocking method are two simple and widely used data partition methods for matrix multiplication [2–4, 11, 14, 17].

1. The *column-blocking* method: Matrix B and C are partitioned into the same amount of column blocks. Each column block of C is computed from matrix A and the corresponding column block of B. The partition is shown below:



2. The *square-blocking* method: Matrix A is partitioned into row blocks and matrix B column blocks. Matrix C is partitioned into blocks with the number equal to the product of the numbers of blocks in A and B. Each block of C is computed from corresponding row and column blocks in A and B. The partition is depicted below:



As regards static data allocation, matrix C in both methods is partitioned into the same amount of blocks as the number of slave processes. In the static column-blocking method, the host process first sends all slave processes matrix A and then each of them a column block of B. Each slave process computes a corresponding column block of C from matrix A and the column block of B received, and then sends the column block of C back to the host process. In the static square-blocking method, the host process sends each slave process a row block of A and a column block of B. After computation, a slave process sends the corresponding block of C back to the host process.

Tables II and III show the execution times of both methods in our experiments performed on SPARC stations (Sun4) connected by a 10 Mbps Ethernet. The np in the tables denotes the number of slave processes participating in the computation. Each slave process runs on a distinct processor. The matrices, A, B, and C, are 500 × 500 in size and each consists of double-precision floating-point numbers. The program was compiled with the GNU C compiler and the execution time was measured by `time()`. In the static column-blocking method, matrix B (500 × 500) is evenly divided into np column blocks. In the static square-blocking method, matrix C is divided into 2 × 1 (np = 2), 2 × 2 (np = 4), and 4 × 2 (np = 8) blocks. The execution time for the static column-blocking method is the summation of time for receiving matrix A, time for computation, and time for sending/receiving the column blocks in the slave process. The execution time for the static square-blocking method is the summation of time for receiving the row and column blocks of A and B, time for computation, and time for sending the block of C in the slave process.

The execution time for both methods is shorter than that for the sequential ones in Table I. Tables II and III also show that the static square-blocking method is somewhat more efficient than the static column-blocking method. The result may be explained by the communication costs. Table IV shows the communication costs of both meth-

TABLE II
Static Allocation by the Column-Blocking Method

np	Recv. A (s.)	Computation (s.)	Send/recv col. (s.)	Total (s.)
2	11	32	4	47
4	18	18	2	38
5	21	16	3	40
10	38	6	3	47

TABLE III
Static Allocation by the Square-Blocking Method

np	Recv. (s.)	Computation (s.)	Send back (s.)	Total (s.)
2	13	35	1	49
4	17	18	1	36
8	32	8	1	42

TABLE IV
Communication Costs in Static Allocation Methods

Method (static)	Communication cost
Square-blocking method	$\sqrt{np} \times (\text{size}(A) + \text{size}(B)) + \text{size}(C)$
Column-blocking method	$np \times \text{size}(A) + \text{size}(B) + \text{size}(C)$

TABLE V
Communication Costs in Dynamic Allocation Methods

Methods (dynamic)	Communication cost
Square-blocking	$q \times \text{size}(A) + p \times \text{size}(B) + \text{size}(C)$
Column-blocking	$np \times \text{size}(A) + \text{size}(B) + \text{size}(C)$

ods represented by the amount of data transferred. For the square-blocking method, matrix A and matrix B are partitioned into $\sqrt{np}$ row blocks and column blocks respectively.[2] Since each row block or column block is involved in the computation of $\sqrt{np}$ blocks of matrix C, there are in total $\sqrt{np}$ copies of matrices A and B sent to the slave processes. The matrix C needs to be transferred only once, obviously. Thus the communication cost is $\sqrt{np} \times (\text{size}(A) + \text{size}(B)) + \text{size}(C)$. For the column-blocking method, that matrix A is sent to each slave process once causes transfer of np copies of A. Matrices B and C need to be transferred only once. The communication cost is $np \times \text{size}(A) + \text{size}(B) + \text{size}(C)$.

The communication cost for both methods is affected by two factors, np and matrix size. When the matrix size is fixed, for small np (e.g., $np \leq 4$), the static square-blocking method is a bit better than the static column-blocking method. For large np, the static square-blocking method easily beats the static column-blocking method because its communication cost growth is much slower. Now, consider the other factor, matrix size. Let np be 4 and let matrix A and matrix B be of equal size, for example. The communication cost in the static square-blocking method is $2(\text{size}(A) + \text{size}(B))$, whereas that in the static column-blocking method is $4 \cdot \text{size}(A) + \text{size}(B)$. The ratio of the communication costs is 4 : 5. As the matrix size grows, the static square-blocking method performs better because its communication cost increases more slowly.

4.3. Dynamic Data Allocation

Dynamic data allocation methods are like static ones, except that the matrices are partitioned into finer blocks so that the partitioned blocks in matrix C are more than the slave processes. Each block dispatching is decided dynamically, according to the time-varying workload and computing power of processors. The dispatching introduces dynamic load balancing, balancing the workload of processors, to reduce the wall clock execution time of an application and promote system utilization.

We used two dynamic data allocation methods based on the data partition methods mentioned in the previous section:

1. The *dynamic column-blocking method.* In the beginning, the host process sends matrix A and a column block in B to each slave process. Then it sends a new undispatched column block of matrix B to a slave process, once it receives a result from the slave process. The host process stops after receiving all column blocks of C. On the other hand, a slave process first receives matrix A and then repeats the following sequence: receives a column block of B, computes the multiplication for the corresponding column block of C, and then sends the result back.

2. The *dynamic square-blocking method.* The host process first sends a row block of A and a column block of B to each slave process, which computes and returns the result. Then it sends one undispatched row block of A and one undispatched column block of B to a slave process, once it receives the result from the slave process. The host process stops after receiving all blocks of C.

Table V shows the communication costs of these two methods, represented by the amount of data transferred. In the dynamic square-blocking method, matrix A contains p row blocks and B contains q column blocks. As in the discussion for Table IV, there will be q copies of matrix A and p copies of matrix B transferred from the host process to the slave processes. Only one copy of matrix C needs to be sent back to the host process. Thus, the communication cost is $q \times \text{size}(A) + p \times \text{size}(B) + \text{size}(C)$. The communication cost for the dynamic column-blocking method is the same as that for the static method in Table IV. In summary, the communication cost for the square-blocking method depends on the number of partitioned blocks in both A and B, while the cost for the column-blocking method is independent of the number of partitioned blocks in B but influenced by the number of slave processes.

Dynamic load balancing might be improved by fine grain size. For the dynamic square-blocking method, fine grain size means more partitioned blocks in matrices A and B (large q and p). That large p and q lead to large communication cost might make dynamic load balancing contrarily worsen the performance of the dynamic square-blocking method. The experiment in Table VI does reflect this situation. This experiment used four slave processes on Sun 4 workstations of equal computing power and load, and matrices A, B, and C contain 500×500 elements each.

The communication cost for the dynamic column-blocking method, as computed in Table V is unrelated to the grain size. Therefore, the dynamic column-blocking method is a better choice when the grain size is fine and the number of slave processes is fixed. Table VII shows the result of the experiment for the dynamic column-blocking method with four slave processes.

[2] Without loss of generality, here np is selected as a square number.

TABLE VI
The Dynamic Square-Blocking Method with Different Grain Sizes

p (Number of row blocks in A)	q (Number of column blocks in B)	$p \times q$ Size of block in C (granularity)	s
4	4	125 × 125	46
5	5	100 × 100	57
10	10	50 × 50	83
20	20	25 × 25	129

TABLE VIII
A Comparison of Performance in an Unequally Loaded Environment

Methods (grain size)	s
Static square-blocking (250 × 250)	185
Dynamic square-blocking (50 × 50)	92
Static column-blocking (500 × 125)	208
Dynamic column-blocking (500 × 1)	50

Table VIII shows the execution time of matrix multiplication on unequally loaded processors using static square-blocking, dynamic square-blocking, static column-blocking, and dynamic column-blocking methods respectively. They were run with four slave processes on distinct Sun 4 workstations connected by a LAN and one of them has a particularly high load (4.1 in UNIX `rup` command, others 0.1). The results indicate that methods with dynamic load balancing do have better performance in unequally loaded environments.

The static square-blocking method has been cited in the literature as the fastest method for parallel matrix multiplication in equally loaded environments [17]. Our results in Table VIII show that in unequally loaded environments, the dynamic column-blocking method is better than the dynamic square-blocking method. In a practical working environment, the load is time-varying; i.e., the environment may sometimes be equally loaded, sometimes not. We conducted another experiment to investigate which method among those discussed in this paper, on the average, performs best in a practical working environment. In the experiment, each method was executed over 140 runs to calculate the average execution time. The 140 runs are distributed evenly over a duration long enough to capture the characteristic of time-varying work load in a practical working environment. For a run we picked five workstations randomly, one as the host process and the others as the slaves. The computing environment for this experiment consisted of 45 Sun workstations of different architectures and configurations and distributed on two different subnets. Each workstation is a time-sharing multiuser system with time-varying work load. In other words, each workstation provided a different and time-varying computer power to applications. Every two pairs of workstations might have different data transfer speed since they may exist in different subnets. The experiment result is shown in Table IX.

Table IX shows that the dynamic column-blocking method needs shortest time in average. The table also indicates that dynamic square-blocking method does not improve its performance with finer granularity, a result that we expected based on Table VI. Our experiment, thus, points out that the dynamic column-blocking method is the best candidate for LAN-connected workstations.

4.4. *High-Speed LAN and Broadcasting*

The fastest sequential algorithm in Section 4.1 takes 63 s. Section 4.2 shows that the fastest method on a 10M bps Ethernet takes 36 s for parallel matrix-multiplication with four slave processes. The speedup is less than 2. The major reason for the unsatisfying speedup should be the communication costs. Here we discuss two promising improvements of communication technology, high-speed LAN and broadcasting facility, to explore which data partition method, dynamic column-blocking or square-blocking, would benefit from such improvements.

Since our environment has no high-speed LAN (faster than 10M bps), our discussion is based on a simulated one. We reduced the amount of data transferred to 1/10 to simulate the reduced communication cost on high-speed

TABLE VII
The Dynamic Column-Blocking Method with Different Grain Sizes

Granularity (size of column block in B)	s
1 column	44
10 columns	44

TABLE IX
The Mean of the Execution Time of Seven Implementations Each over 140 Samples

Methods (granularity)	Average execution time
Dynamic column-blocking (1 column)	42.02
Dynamic square-blocking (125 × 125)	50.77
Dynamic square-blocking (100 × 100)	53.84
Dynamic square-blocking (50 × 50)	80.38
Dynamic square-blocking (25 × 25)	142.21
Static column-blocking (125 columns)	66.82
Static square-blocking (250 × 250)	63.05

TABLE X
The Column-Blocking Method with Dynamic Load Balancing on a 100M bps Ethernet

np	Recv. A (s)	Computation (s)	Recv/send col. (s)	Total (s)
2	1	32	0	33
4	2	17	0	19
5	2	16	0	18
8	3	11	0	14
12	6	7	0	13
16	7	5	0	12
20	9	4	0	13
25	11	4	0	15

LAN. Tables X and XI show the performance, in such an environment, of the dynamic column-blocking and square-blocking methods. When $np = 4$, both dynamic column-blocking and square-blocking methods run 3.3 times faster than the sequential version (the block algorithm in Section 4.1). This indicates that both these methods can take advantage of high-speed LANs to improve their performance and scalability.

Consider a high-speed LAN with the broadcasting (or multicasting) facility. There is only one, the column-blocking method, benefited with the broadcasting facility, because the method involves broadcasting matrix A in the first step. The experiment in Table XII is a modification of that in Table X, where the time for receiving matrix A is kept constant as np is scaled up. When $np = 4$, the speed is 3.5 times higher than the sequential version. When $np = 20$, the speed is 12.6 times higher.

Most parallel BLAS libraries employ the static square-blocking method. This approach is effective because it takes advantage of the interconnection topology of a particular architecture, such as mesh, and the block algorithms of sequential implementation on each processor. As dynamic load balancing is introduced, the square-blocking method may cause a rapid increase of number of data transferred. Hence, the dynamic column-blocking method is superior. In addition, other than the dynamic square-blocking method, the dynamic column-blocking method can take advantage of broadcasting and multicasting, two promising facilities. In summary, the introduction of dynamic load balancing may change our perspective on data

TABLE XI
The Square-Blocking Method with Dynamic Load Balancing on a 100M bps Ethernet

np	Recv. (s)	Computation (s)	Send (s)	Total (s)
2	1	32	2	35
4	2	15	2	19
5	2	13	2	17
10	4	7	0	11
20	28	5	12	45

TABLE XII
The Column-Blocking Method with Broadcasting and Dynamic Load Balancing on a 100M bps Ethernet

np	Recv. A (s)	Computation (s)	Recv/send col. (s)	Total (s)
2	0.5	32	0	33
4	0.5	17	0	18
5	0.5	16	0	17
8	0.5	11	0	12
12	0.5	7	0	8
16	0.5	5	0	6
20	0.5	4	0	5
25	0.5	4	0	5

partition methods. The results of our experiments point out that the dynamic column-blocking method outperforms the dynamic square-blocking method under current environment, even high speed LANs with broadcasting facilities.

5. LU FACTORIZATION

Most problem solving work is done with a sequential program at the beginning. As the problem size and computation need grow, parallel computing shows a promising way speeding up the problem solving. There are at least two approaches to speed up the problem solving with parallel computing. One is to develop a parallel algorithm and work out the corresponding parallel implementation. This approach requires understanding of the problem solving process, and also faces the high costs of designing, coding, debugging, and testing.

Another approach is to replace the associated sequential libraries used by the sequential program with parallel ones, and the program context changes nothing. An example is to replace the sequential BLAS in scientific programs with parallel BLAS. This approach, unlike the former, seems unable to take advantage of all parallelism. However, the performance improvement is good for many cases at a negligible cost.

To apply the second approach smoothly, an important premise for the parallel library is to maintain its interface consistent with the original (sequential). Unfortunately, few existing parallel libraries meet this requirement. For example, most parallel numerical library routines of static square-blocking methods, e.g., *ScaLAPACK,* require additional parameters to describe data block size and/or the processor topology. These parameters do not appear in the sequential version. Our parallel BLAS library meets this premise. Our routines contain an interface which is consistent with the well-defined sequential one [1, 15]. They adopt the dynamic column-blocking method, whose communication cost is not affected by the block size. Therefore, in our library, the block size is set to one column for better performance and release user from specification of block size. Each time our parallel routine is called, it determines the optimal number of processors to be used in parallel according to the input matrix size. This feature releases users from calculating the optimal number of processors needed. In addition, the trivial work in all the data communication and processor arrangement is also encapsulated.

Here is an example applying the second approach with our parallel Level 3 BLAS. The application programmed is LU factorization [15]. In the computation of LU factorization, the matrices to be processed are getting smaller at each next iteration. Our parallel routines which automatically determine the optimal processor number is very helpful for such kind of application to attain good performance at each iteration. Two Level 3 BLAS routines, TRSM and GEMM, performing *triangular solve* and *rank-k update* respectively, contribute most to LU factorization. Applying the second approach, the parallel implementation of LU factorization is easy; we just linked routine DGETRF, the LU factorization in LAPACK [1], to our parallel TRSM and GEMM.

To factorize a matrix with 800×800 elements takes 152 s in the sequential program and 102 s in our parallel version with four processors. The parallel program runs about 1.5 times faster than the sequential one. The speedup of 1.5 with four processors for LU factorization is only a bit smaller than the datum for its building block (GEMM for matrix multiplication) in Section 4 (63 s/36 s = 1.75). This indicates that higher level applications can be speeded up by our parallel Level 3 BLAS routines at a negligible cost. Furthermore, our parallel Level 3 BLAS routines with the dynamic column-blocking method can produce significant effect of dynamic load balancing when applied in higher level applications. Table XIII shows the average execution time of 140 runs for LU factorization whose underlying parallel Level 3 BLAS routines use static square-blocking and dynamic column-blocking methods respectively. The 140 runs were done in the same environment as for Table IX in Section 4.3.

6. CONCLUSION AND FUTURE WORK

Our experiments on parallel Level 3 BLAS, considering dynamic load balancing, indicate that

1. When dynamic load balancing is applied on LAN-connected workstations, the dynamic column-blocking method is the best choice. The static square-blocking method adopted in most existing parallel BLAS libraries is not.
2. High-speed LANs plays a significant role in improving currently unsatisfying performance. To improve the scalability of parallel implementation, the column-blocking method can take more advantage of the broadcasting facilities than the square-blocking method.
3. With a consistent interface between sequential and parallel numerical libraries, a sequential program can be sped up with the parallel libraries at a negligible cost.

This paper focuses on dense matrices. However, sparse matrices or matrices of other special types are used in many scientific applications. To provide efficient parallel libraries for scientific applications, further studies are needed.

TABLE XIII
The Effect of Dynamic Load Balancing on LU Factorization

Methods of parallel BLAS	Mean (s)
Static square-blocking	206.1
Dynamic column-blocking	180.8

REFERENCES

1. Anderson, E., Dongarra, J. J., and Ostrouchov, S. *Installation Guide for LAPACK.* 1993.
2. Bjorstad, P. E., and Sorevik, T. Data-parallel BLAS as a basis for LAPACK on massively parallel computers. *Linear Algebra for Large Scale and Real-Time Application: Proceedings of the NATO Advanced Study Institute, Leuven, Belgium, August 3–14, 1992.* Pp. 13–20.
3. Choi, J., Dongarra, J. J., Pozo, R., and Walker, D. W. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. *Conference Proceedings of Frontiers of Massively Parallel Computing, 1992.* Pp. 120–127.
4. Choi, J., Dongarra, J. J., and Walker, D. W. PB-BLAS: A set of parallel block basic linear algebra subprograms. Technical Report ORNL/TM-12468, Oak Ridge National Laboratory, 1994.
5. Chou, T. C. K., and Abraham, J. A. Load balancing in distributed systems. *IEEE Trans. Software Eng.* **SE-8,** 4 (July 1982), 401–412.
6. Chow, Y. C., and Kohler, W. H. Models for dynamic load balancing in a heterogeneous multiple processor system. *IEEE Trans. Comput.* **C-28,** 5 (May 1979), 354–361.
7. Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. J. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Software* **14,** 1 (Mar. 1988), 1–17.
8. Dongarra, J. J., Croz, J. D., Hammarling, S., and Hanson, R. J. Algorithm 656, an extended set of basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Software* **14,** 1 (Mar. 1988), 18–32.
9. Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Software* **16,** 1 (Mar. 1990), 1–17.
10. Dongarra, J. J., Croz, J. D., Hammarling, S., and Duff, I. Algorithm 679, a set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Trans. Math. Software* **16,** 1 (Mar. 1990), 18–28.
11. Dongarra, J. J., Geijn, R., and Walker, D. A look at scalable dense linear algebra libraries. *Conference Proceedings of Scalable High Performance Computing, 1992.* Pp. 372–379.
12. Dongarra, J. J., Geijn, R. A., and Whaley, R. C. *A Users' Guide to the BLACS.* 1993.
13. Dongarra, J. J., Moulton, S. A., Ostrouchov, L. S., Petitet, A., and Whaley, R. C. *Installation Guide for ScaLAPACK.* 1993.
14. Falgout, R. D., Skjellum, A., Smith, S. G., and Still, C. H., The multicomputer toolbox approach to concurrent BLAS and LACS.

Conference Proceedings of Scalable High Performance Computing, 1992. P. 121–128.

15. Frpeeman, T. L., and Phillips, C. *Parallel Numerical Algorithms.* Prentice–Hall International, 1992.
16. Geist, A., Beguelin, A., Dongarra, J. J., Jiang, W., Manchek, R., and Sunderam, V. *PVM3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, 1993.
17. Ghosh, B., and Schultz, M. H., Portable parallel level-3 BLAS in Linda. *Conference Proceedings of Scalable High Performance Computing, 1992.* Pp. 416–423.
18. Ni, L. M., and Kwang, K. Optimal load balancing strategies for a multiple processor-system. *Proc. Int. Conf. Parallel Processing.* IEEE Computer Soc. Press, Los Alamitos, CA, 1981, pp. 352–357.
19. Ni, L. M., and Abani, K. Nonpreemptive load balancing in a class of local area networks. *Proc. Computer Networking Sympos.* IEEE Computer Soc. Press, Los Alamitos, CA, 1981, pp. 113–118.

KUO-CHAN HUANG received the B.S. in computer and information engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1993. He has been a student member of ACM and IEEE since 1994. Currently, he is a Ph.D. student in the Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan. His research interests include high-performance computing, parallel programming, software architecture, computational science, and problem-solving environments.

FENG-JIAN WANG is a professor in the Department of Computer Science and Information Engineering, National Chiao Tung University, Taiwan. He received his B.S. in physics from National Taiwan University in 1980, and both the M.S. and Ph.D. in E.E.C.S. from Northwestern University, Illinois, in 1985 and 1988, respectively. His research interests include software engineering, compiler, object-oriented techniques, and distributed systems.

PEI-CHI WU received the B.S., M.S., and Ph.D. from National Chiao Tung University, Hsinchu, Taiwan, in 1989, 1991, and 1995, respectively, all in computer science and information engineering. He has been a member of ACM and IEEE since 1990. He has been a volunteer reviewer of ACM *Computing Reviews* since 1992. He was one of the winners of the 1993 Outstanding Paper Award of the Computer Society of the Republic of China. He is currently in military service. His research interests include compiler construction, object-oriented programming, high-performance distributed computing, and Monte Carlo simulation.

Received November 28, 1994; revised February 7, 1996; accepted July 20, 1996