A Ring-Oriented Approach for Block Matrix Factorizations on Shared and Distributed Memory Architectures*

Krister Dackland† Erik Elmroth† Bo Kågström†

Abstract

A block (column) wrap-mapping approach for design of parallel block matrix factorization algorithms that are (trans)portable over and between shared memory multiprocessors (SMM) and distributed memory multicomputers (DMM) is presented. By reorganizing the matrix on the SMM architecture, the same ring-oriented algorithms can be used on both SMM and DMM systems with all machine dependencies comprised to a small set of communication routines. The algorithms are described on high level with focus on portability and scalability aspects. Implementation aspects of the LU, *Cholesky*, and QR factorizations and machine specific communication routines for some SMM and DMM systems are discussed. Timing results show that our portable algorithms have similar performance as machine specific implementations.

1 Introduction

With the introduction of advanced parallel computer architectures a demand for efficient and portable algorithms has emerged. Several attempts to design algorithms and implementations that are portable between *shared memory multiprocessors* (SMM) and *distributed memory multicomputers* (DMM) have been made. For example, implementations of virtual shared memory on DMM architectures [15, 21, 23] and the simulation of message passing with portable communication libraries on SMM architectures [3, 4, 13]. A major drawback with these attempts is that the demands for generality often deteriorates the performance for many problems that can be efficiently solved if the portability aspects are neglected. However, it is possible to obtain portability without loss of performance for a restricted class of problems. In this paper we restrict our study to parallel block matrix factorizations, such as the LU, *Cholesky*, and QR factorizations [7, 11, 12]. This class of problems can most likely be enlarged to several block algorithms, such as the ones included in LAPACK [2].

Our goal is to construct algorithms that, without loss of performance compared to implementations tuned for specific machines, are portable over and between different SMM and DMM architectures. This is done by further development of the ring-oriented block algorithms for DMM and SMM architectures, respectively, presented in [5, 7]. The algorithms are described on high level with focus on the portability aspects. Results are presented for implementations on Alliant FX2816, Intel iPSC/2 hypercube and IBM 3090 VF/600J and compared to the machine specific results presented in [5, 7]. We also discuss

*Financial support has been received by the Swedish National Board of Industrial and Technical Development under grant NUTEK 89-02578P.

†Institute of Information Processing, University of Umeå, S-901 87 Umeå, Sweden

the scalability of these parallel algorithms, which is a measure of their capabilities to effectively use an increasing number of processors [18].

The outline is as follows. A generic ring-oriented block algorithm is presented in the perspective of a DMM architecture in Section 2. The algorithm is adapted to an SMM architecture in Section 3. The implementation of the machine specific communication routines are discussed in Section 4. Performance results are presented in Section 5, followed by a short discussion of performance modeling in Section 6. Finally, Section 7 discusses the results and presents some conclusions.

2 A Generic Ring-Oriented Block Algorithm

We assume that the target DMM architecture can embed a unidirected ring topology. The $m \times n$ matrix A to be factorized is block column wrap mapped onto the nodes to achieve good load balancing. For a given processor k $(= 0 : p-1)$, these blocks form a local matrix A_{local}, comprising the block columns $k+1$, $k+1+p$, $k+1+2p$, etc of A, where p is the number of processors.

Let nbl denote the number of block columns in A (for clarity we assume $m \geq n$), nbl_{local} the number of block columns in A_{local}, and *cb_index* the (block) index for the current block column. If the (local) block index i is initialized to 1, then a high level generic node algorithm for computing a matrix factorization can be described in the following way:

For $i\_global = 1 : nbl$
    If (I hold block i_global)
        If $(i\_global > 1)$
            UPDATE $A_{\text{local}}(i)$ w.r.t *CURRENT_BLOCK*(*cb_index*) (1)
        End If
        FACTORIZE $A_{\text{local}}(i)$ & generate *NEXT_CURRENT_BLOCK* (2)
        BROADCAST(*NEXT_CURRENT_BLOCK*, *cb_index*) (3)
        $i = i + 1$
    End If
    If $(i\_global > 1)$
        UPDATE $A_{\text{local}}(i : nbl_{\text{local}})$ w.r.t. *CURRENT_BLOCK*(*cb_index*) (4)
    End If
    RECEIVE(*CURRENT_BLOCK*, *cb_index*) (5)
End For

Processor 0 starts to factorize (2) the first block column and broadcasts (3) the factors, denoted *NEXT_CURRENT_BLOCK* to all other processors in the ring. Then, all processors update their remaining blocks (4), i.e. the blocks to the right of the last factorized block column in A, with respect to the received *CURRENT_BLOCK*. Notice that the next current processor, i.e. the right neighbour in the ring, updates (1) and factorizes (2) the next block column and broadcasts it (*NEXT_CURRENT_BLOCK*) before it completes the update corresponding to the previous factorization (*CURRENT_BLOCK*). This enables an overlapping called *pipelining between iterations* [5, 7].

In the implementations of the algorithms, the factorization of a block column is performed by a call to the corresponding level 2 routine, and the update is performed by one or more calls to level 3 BLAS [8, 19, 20]: `DGEMM` and `DTRSM` in LU, `DGEMM`, `DSYRK`, and `DTRSM` in $Cholesky$, `DGEMM` and `DTRMM` in QR.

The only extra storage needed is a matrix to hold the current (factorized) block. All synchronizations are implicit through the message passing of the current blocks. Except for

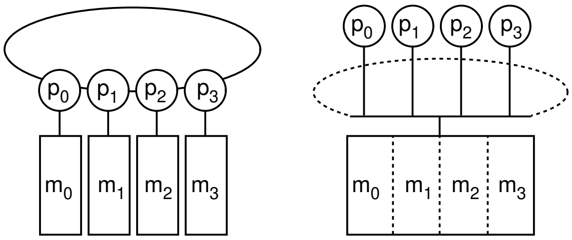

FIG. 1. *DMM and SMM models viewed as a ring of processors*

the current blocks, a processor only refers to A_{local}. This gives an inherited data locality from the initial matrix distribution in the DMM environment, such that each processor performs the work on separate block columns. We would like to have the same data locality on an SMM environment.

3 Shared Memory and Ring-Oriented Block Algorithms

Our approach is to use the DMM paradigm as a model and run the same factorization routines on an SMM architecture viewed as a ring of processors, see Figure 1.

To accomplish this we reorganize A into A_{reorg} such that consecutive block columns of A_{reorg} are the block columns of A interchanged in a block column wrap mapped manner. By partitioning the work evenly over the p processors, each processor only makes access to a logical local matrix A_{local} that refers to a submatrix of the restructured global matrix A_{reorg}. In Figure 1 the memory units m_i of the DMM and SMM models store the same block columns of A. Moreover, it follows that the references to A_{local} can be made identical to the ones in the generic DMM algorithm. The main difference is how we deal with the current (factorized) block of A. In the DMM algorithm this block is broadcasted to all processors that keep it as a local copy. In the SMM algorithm the current block is a block column of A_{reorg} in the global shared memory, which can be accessed by all processors. The message passing of the current block is implemented as a change of reference for the current block to different locations in A_{reorg}.

4 Implementation of Communication Routines

To make the factorization routines (trans)portable over and between different SMM and DMM environments, all machine and compiler dependencies have been captured in communication routines with the same interfaces for both SMM and DMM systems. These implement the broadcasting and the reception of double precision and integer matrices on three MIMD machines using their extensions to FORTRAN 77 for communication and synchronization. The machines and compilers used are: Alliant FX2816 and FX/FORTRAN-2800 [1], IBM 3090 VF/600J and VS FORTRAN 2.5 [16], the Intel iPSC/2 hypercube and f77 [17].

The implementations on the iPSC/2 consist of interfaces to iPSC/2 communication library routines. For the SMM systems Alliant and IBM the `BROADCAST` and `RECEIVE` routines are implemented as an update of a current block pointer. To ensure that only one processor can get access to the pointer at a time it is synchronized by semaphores. In Figure 2 the body of the Alliant `BROADCAST` and `RECEIVE` subroutines are displayed. The routines look the same for the IBM system except for the syntax of the semaphores.

```
C     BROADCAST
C
      IF (MSGID .EQ. CB) THEN
99       CALL ENTER()
         IF (SHDATA(1) .NE. 0) THEN
            CALL LEAVE()
            GOTO 99
         ELSE
            SHDATA(0) = LOCAL_CBI
            LOCAL_CBI = LOCAL_CBI + N
            SHDATA(1) = P
         ENDIF
         CALL LEAVE()
      ENDIF
```

```
C     RECEIVE
C
      IF (MSGID .EQ. CB) THEN
99       CALL ENTER()
         IF (SHDATA(0) .EQ. CBI) THEN
            CALL LEAVE()
            GOTO 99
         ELSE
            CBI = SHDATA(0)
            SHDATA(1) = SHDATA(1) - 1
         ENDIF
         CALL LEAVE()
       ENDIF
```

FIG. 2. *Alliant double precision* BROADCAST *and* RECEIVE *subroutine bodies*

The BROADCAST and RECEIVE implementations are not general, but sufficient for the *LU*, *Cholesky*, and *QR* factorizations. For example, there is no need for explicit synchronization of the pivot vector in the *LU* factorization, since it is implicitly synchronized by the communication for the current block. Therefore, the communication routines for integer matrices are implemented as quick returns. Before calling a factorization routine, the "message passing" on the SMM system has to be initialized by a call to an initialization routine. Further details can be found in [6].

5 Performance results

We present performance results for the *LU*, *Cholesky*, and *QR* factorizations implemented as node subprograms in FORTRAN 77 and compiled for Alliant FX2816, Intel iPSC/2, and IBM 3090 VF/600J.

When we use these node subprograms the processors are assumed to be allocated and the matrix A is properly distributed or reordered. The timing of the routines measures the execution time for the factorizations, which all involve $O(n^3)$ arithmetic operations (for $m = n$) and communications (excluding the initial matrix distribution or reordering). The initial reordering of A on the SMM platforms is an $O(n^2)$ operation which is perfectly parallelizable.

The results in Tables 1 - 3 show the performance measured in *Mflops*, computed as the maximum time over p nodes divided by the number of floating point operations counted as in [2]. The optimal block size nb and the *parallel efficiency*, $E(p)$, are displayed for the three algorithms. The parallel efficiency is computed as

$$E(p) = \frac{Mflops(p)/Mflops(p_{\min})}{p/p_{\min}}.$$

where *Mflops*(x) is the *Mflops* number for x processors, and $p_{\min}$ is the minimum number of nodes that could solve the problem. The value of $p_{\min}$ is one for the Alliant and IBM systems while it is restricted by the available storage on each node for the iPSC/2 (at most 4 *Mbytes* per node including the code).

Alliant FX2816. The SMM system consists of 16 Intel i860 processors each with a theoretical peak performance of 40 *Mflops* in double precision real arithmetic. We use

TABLE 1
Alliant FX2816: LU, Cholesky, and QR factorizations of a 1024×1024 *matrix.*

	LU			**Cholesky**			**QR**		
p	nb	*Mflops*	$E(p)$	nb	*Mflops*	$E(p)$	nb	*Mflops*	$E(p)$
1	64	25.1	1.00	64	25.2	1.00	32	25.8	1.00
2	48	47.8	0.95	64	47.7	0.95	32	49.4	0.96
3	48	67.2	0.89	48	66.2	0.88	32	70.9	0.92
4	48	86.4	0.86	32	79.8	0.79	32	90.8	0.88
5	48	102.0	0.81	32	90.0	0.71	16	109.0	0.85
6	32	113.3	0.75	32	96.0	0.63	16	125.5	0.81
7	32	125.7	0.72	16	103.6	0.59	16	141.7	0.79
8	32	137.0	0.68	16	109.5	0.54	16	153.2	0.74

TABLE 2
Intel iPSC/2: LU, Cholesky, and QR factorizations of a 1000×1000 *matrix.*

	LU				**Cholesky**				**QR**			
p	nb	*Mflops*	$E(p)$	$Prev$	nb	*Mflops*	$E(p)$	$Prev$	nb	*Mflops*	$E(p)$	$Prev$
4	4	1.29	1.00	1.29	8	1.29	1.00	1.29	4	1.55	1.00	1.55
8	2	2.53	0.98	2.53	8	2.54	0.99	2.53	2	3.06	0.99	3.06
16	2	4.93	0.96	4.93	4	4.88	0.95	4.83	2	5.99	0.97	6.00
32	2	9.40	0.91	9.39	2	8.73	0.85	8.39	1	11.58	0.93	11.65
64	1	17.10	0.83	17.02	2	13.40	0.65	12.79	2	21.54	0.87	21.71

the DGEMM routine from the Alliant library libalgebra 2.0 [1]. Its best uniprocessor performance is about 35 *Mflops*. All other level 3 BLAS used are from the GEMM-based library [19, 20]. In Table 1, performance results on one to eight processors for the three routines are shown. By restricting the size of the cluster to eight, only the two processors (of four) that have distinct cache controllers on each processor module are used. This minimizes the memory conflicts in the global cache.

Intel iPSC/2 Hypercube. This DMM system has 64 scalar SX nodes. Each node is equipped with an Intel 80386, 4 *Mbyte* memory, and a Weitec 1167, which has a theoretical peak performance just under 0.6 *Mflops* in double precision real arithmetic. The BLAS used are the standard FORTRAN 77 implementations accompanying [2]. The best performance obtained from the level 3 BLA routine DGEMM is around 0.5 *Mflops* [5]. Results for iPSC/2 are presented in Table 2 together with previously presented results, *prev*, for machine specific implementations [5].

IBM 3090 VF/600J. The SMM system has 6 processors, each equipped with a vector facility (VF). The theoretical peak performance is 138 *Mflops* per processor in double precision real arithmetic. The practical peak performance obtained from the level 3 BLA routine DGEMM is around 108 *Mflops* [7]. The level 3 BLAS used are from ESSL [16], except for a tuned FORTRAN implementation of DTRMM [22]. Results for IBM 3090 are presented in Table 3 together with previously presented results, *prev*, for machine specific implementations [7].

TABLE 3
IBM 3090 VF/600J: LU, Cholesky, and QR factorizations of a 1200×1200 *matrix.*

	LU				Cholesky				QR			
p	nb	*Mflops*	$E(p)$	*Prev*	nb	*Mflops*	$E(p)$	*Prev*	nb	*Mflops*	$E(p)$	*Prev*
1	48	97.3	1.00	98.1	256	99.6	1.00	91.9	32	92.8	1.00	94.2
2	32	188.6	0.97	192.9	96	178.5	0.90	173.1	32	183.2	0.99	182.9
3	32	275.7	0.94	277.6	64	255.0	0.85	247.2	32	267.2	0.96	266.0
4	32	356.6	0.92	355.2	32	315.8	0.79	307.8	32	345.8	0.93	345.2
5	32	427.9	0.88	431.0	48	353.8	0.71	354.4	32	402.7	0.87	413.2

6 Performance Modeling

A model that identifies different components of the total execution time $T_{\text{pred}}(p)$ using p processors can be expressed as

$$T_{\text{pred}}(p) = T_a(p) + T_c(p) + T_w(p), \tag{1}$$

where $T_a(p)$ is the arithmetic time, $T_c(p)$ is the communication time, and $T_w(p)$ is the waiting time, including both idle and busy waiting. From $T_{\text{pred}}(p)$ it is, for example, possible to estimate optimal block sizes for given matrix sizes (m, n) and p. A reasonable simplification is to consider average times for the different time components. The motivation is that for sufficiently large matrices and almost optimal block sizes, the initial "matrix distribution" gives the processors approximately the same amount of work.

Arithmetic Time. We express the average arithmetic time $T_a(p)$ of an algorithm, implemented on a DMM or an SMM system, as

$$T_a(p) = \frac{\#\mathit{flops} \cdot t_a}{S_a(p, nb)} \mathcal{V}(m, n, nb), \tag{2}$$

where *#flops* is the total number of floating point operations of the matrix factorization, t_a is the time required for one floating point operation and $S_a(p, nb)$ is the arithmetic speedup on p processors with block size nb (excluding the costs for communication and waiting). Due to the memory hierarchy of the processor node and software overhead, the arithmetic time on one processor varies for different matrix and block sizes. The function $\mathcal{V}(m, n, nb)$ models this variation in performance. It can, for example, be determined from a least squares fit of uniprocessor timing results for different m, n and block sizes nb. As a consequence also the arithmetic speedup varies and we express $S_a(p, nb)$ as the linear speedup times a factor that models the variation of the arithmetic speedup $(p(1+v)$, where $|v(m, n, nb)| < 1)$. Accurate iPSC/2 models for $\mathcal{V}(m, n, nb)$ and $S_a(p, nb)$ are presented in [5].

Communication Time. Since all communications are expressed by message passing primitives $T_c(p)$ can be modeled in terms of the number of messages each processor communicates (broadcasts or receives) and the cost for sending and receiving a message of a given size. To make the discussion clearer, we assume that $m = n$, all block columns have the same number of columns, i.e. n is a multiple of nb, and that the number of block columns $nbl = n/nb$ is a multiple of p. In Table 4 the total message traffic of the three ring-oriented block algorithms is displayed.

In each iteration of the generic algorithm one message is broadcasted and $p-1$ messages are received. For the LU factorization, the factorized block and a part of the pivot vector

TABLE 4

Message traffic of the ring-oriented block algorithms for $m = n$ and $nbl = n/nb$.

	BROADCAST		RECEIVE	
Factorization	# Blocks	Size (in bytes)	# Blocks	Size (in bytes)
LU	nbl	$4(n^2 + n \cdot nb)$	$(p-1)nbl$	$4(p-1)(n^2 + n \cdot nb)$
	nbl	$8n$	$(p-1)nbl$	$8(p-1) \cdot n$
Cholesky	$nbl - 1$	$4(n^2 - n \cdot nb)$	$(p-1)(nbl-1)$	$4(p-1)(n^2 - n \cdot nb)$
QR	nbl	$4(n^2 + n \cdot nb)$	$(p-1)nbl$	$4(p-1)(n^2 + n \cdot nb)$
	nbl	$8n \cdot nb$	$(p-1)nbl$	$8(p-1) \cdot n \cdot nb$

are broadcasted in each iteration. In the *Cholesky* factorization, only the factorized block is sent in each iteration (except for the last one). The size of the current block in iteration i is $(n - i \cdot nb)nb$ for the Cholesky factorization while it is $(n - (i-1)nb)nb$ for the LU and QR factorizations. In the QR factorization we use a compact WY representation of the block transformation in each step (see, e.g., [12]), so besides the factorized block we have to broadcast a triangular matrix S in each block iteration. The costs for communicating these messages are entirely determined by our target architecture. For a DMM environment the usual linear model $\alpha + M\beta$ can be used, where α is the startup cost and β is the cost per unit for communicating a message of M bytes. Average time models of $T_c(p)$ for iPSC/2 are presented in [5]. In an SMM environment a single address space is accessible to every processor in the system. The "message passing" is accomplished through synchronized access of shared variables in a shared memory system. The communication costs are determined by the memory hierarchy of the system and the costs for synchronization (e.g. cache and latency effects). A modeling of the Alliant FX2800 is under investigation.

Waiting Time. The average waiting time $T_w(p)$ can be divided into two parts; *busy waiting* time, $T_{bw}(p)$, and *idle waiting* time, $T_{iw}(p)$. $T_{bw}(p)$ comprises the time needed for the first processor to factorize and "distribute" the first current block, and the time when different processors are "waiting for" messages to arrive. The last part exists if the current processor uses more time to produce the next factorized block (including its broadcast) than the other processors need for the update corresponding to the previous factorization. Essentially, $T_{iw}(p)$ comprises the time some processors are idle at the end of the computations (mainly due to imbalance of the load). In [5] an iPSC/2 model for $T_w(p)$ is discussed.

7 Discussion

The performance results in section 5 show that the generic ring-oriented algorithms are (trans)portable and give good parallel performance on both DMM and SMM environments. The results compete favourably with previously published results for machine specific implementations of similar algorithms on IBM 3090 VF/600J [7] and iPSC/2 [5]. To our knowledge the Alliant FX2816 results are the best available for these three matrix factorizations programmed entirely in FX/FORTRAN-2800. The similarity in performance on the iPSC/2 is explained by the small differences between the generic implementations and the machine specific versions. It follows that the modeling and performance evaluation in [5] also are valid for the ring-oriented algorithms discussed here. The modeled execution time, T_{pred}, predicts the best block size to be the same as the best one in real execution, except occasionally, when it predicts a nearby block size which in real execution shows a performance close to the optimal one. For the SMM environments the good performance of the ring-oriented algorithms are explained by the reorganization of the matrix, giving each processor block columns that are stored consecutively, and the proportionately low synchronization costs. The consecutive storage enables updating of several block columns in one operation, instead of one operation per block column. This implies less amount of software overhead and larger matrices in the level 3 BLAS operations. In [7] software overhead and level 3 fractions for parallel block matrix factorizations and their impact on the performance of an SMM system are discussed.

Due to space limits, we have only presented results for a fixed problem size and a varying number of processors. In this case the parallel speedup is an appropriate measure of the scalability of the algorithm. Both the parallel speedup and the parallel efficiency are determined by the serial fraction of the algorithm, as well as, communication costs and possible overhead due to redundant work. For the number of processors we have used, the algorithm-architecture combinations show speedups with a linear behaviour and we can argue that the algorithms have good scaling properties. In general, the solution time may finish to decrease (or even increase) due to the parallel overheads if we continue to increase the number of processors while maintaining a fixed problem size. It would then be more appropriate to consider the *scaled-speedup* which is defined as the speedup obtained when the problem size is increased linearly with p [14]. For example, results for iPSC/2 in [5] show that the maximal performance per processor is almost constant if the matrix is large enough (see also [9]).

The GEMM-based level 3 BLAS [19, 20] and a highly optimized uni-processor DGEMM routine make the ring-oriented approach efficient on both DMM and SMM platforms. Here demonstrated on the Alliant FX2816 system. Now, all machine dependencies are restricted to the communication primitives, the DGEMM routine and lower level BLAS. The implementation of the communication primitives on an SMM environment is cost effective and easy, and will be further simplified in FORTRAN 90, which offers dynamic data structures.

We believe our approach is applicable to block algorithms in general, making it possible to construct a complete library portable over and between DMM and SMM platforms. This enables use of routines in sequence, hopefully without redistribution or reorganization of the matrix between each operation. If a reorganization is needed on a SMM system, it is an easily parallelized $O(m \cdot n)$ operation. The cost for redistributing a matrix on a DMM environment is determined by the connectivity properties of the physical interconnection network, and is normally more expensive than on an SMM model. Moreover, it is possible to extend our approach to a 2-dimensional mesh topology and use a square block scattered decomposition of the matrices (e.g., see [10, 9]) for both DMM and SMM environments.

References

[1] Alliant Computer Systems Corporation, "FX/FORTRAN-2800, Programmer's Handbook", "FX/SERIES Linear Algebra Library", 1990.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[3] L. Bomans, D. Roose, and R. Hempel, "The Argonne/GMD macros in Fortran for portable programming and their implementation on the Intel iPSC/2", *Parallel Computing*, 15, (1990), pp 119-132.

[4] J. Boyle, R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens, *Portable Programs for Parallel Processing*, Holt, Rinehart and Winston, Inc, 1987.

[5] K. Dackland and E. Elmroth, "Parallel Block Matrix Factorizations for Distributed Memory Multi-

computers", Report UMINF-92.03, Inst. of Information Processing, Univ. of Umeå, S-901 87 Umeå, May 1992.

[6] K. Dackland and E. Elmroth, "Ring-oriented Block Matrix Factorization Algorithms for Shared and Distributed Memory Architectures", Report UMINF-92.04, Inst. of Information Processing, Univ. of Umeå, S-901 87 Umeå, June 1992.

[7] K. Dackland, E. Elmroth, B. Kågström, and C. Van Loan, "Parallel Block Matrix Factorizations on the Shared Memory Multiprocessor IBM 3090 VF/600J", *International Journal of Supercomputer Applications*, Vol 6.1, MIT Press (1992), pp 69-97

[8] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms", *ACM Trans. on Mathematical Software*, Vol. 16 (1990) pp 1-17, 18-28.

[9] J. Dongarra, R. van de Geijn, and D. Walker, "A Look at Scalable Dense Linear Algebra Libraries", ORNL/TM-12126, Oak Ridge National Lab., Oak Ridge, Tennessee 37831, July 1992.

[10] G. Fox, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors*, Vol. 1, Prentice Hall, Englewood Cliffs, N.J., 1988.

[11] K. Gallivan, R. Plemmons and A. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations", *SIAM Review*, Vol. 32 (1990), pp 54-135.

[12] G. Golub and C. Van Loan, *Matrix Computations*, 2nd edition, John Hopkins Press, 1989.

[13] A. Geist, M. Heath, B. Peyton, and P. Worley, "PICL a Portable Instrumented Communication Library", Tech. Memorandum ORNL/TM-6150, Oak Ridge National Laboratory, 1990, pp 1213-1222.

[14] J. L. Gustafson, G. R. Montry, and R. E. Benner, "Development of Parallel Methods for a 1024-Processor Hypercube", *SIAM J. Sci. Statist. Comput.*, Vol. 9(4) (1988), pp 609-638.

[15] H. Hellwagner, "A Survey of Virtually Shared Memory Schemes", TUM-I9056, Institut für Informatik, Technische Universität München, 1990.

[16] IBM, "Engineering and Scientific Subroutine Library Guide and Reference", SC23-0184-3, Nov.1988. "VS FORTRAN Version 2 Release 5 Language and Library Reference", SC26-4221-6, Dec.1990.

[17] Intel Corporation, "iPSC/2 and iPSC/860, Programmer's Reference Manual", June 1990.

[18] V. Kumar and A. Gupta, "Analyzing Scalability of Parallel Algorithms and Architectures", TR 91-18, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, MN - 55455, Revised Nov. 1992.

[19] B. Kågström P. Ling, and C. Van Loan, "High Performance GEMM-Based Level-3 BLAS: Sample Routines for Double Precision Real Data", in M. Durand and F. El Dabaghi (eds.), *High Performance Computing II*, Elsevier Science Publishers B.V. (North-Holland), (1991), pp 269-281.

[20] B. Kågström and C. Van Loan, "GEMM-Based Level-3 BLAS", Tech. Report CTC91TR47, Cornell University, December 1989.

[21] K. Li, "IVY: A Shared Virtual Memory on Loosely Coupled Multiprocessors", Ph.D. Thesis, Yale University, 1986.

[22] P. Ling, "A Set of High Performance Level-3 BLAS Structured and Tuned for the IBM 3090 VF and Implemented in Fortran 77", Report UMINF-179.90, Inst. of Information Processing, Univ. of Umeå, S-901 87 Umeå, May 1990.

[23] M. Stumm and S. Zhou, "Algorithms Implementing Distributed Shared Memory", *IEEE Computer*, Vol. 23, No. 5, (1989), pp 54-64.