Generalized Cannon's Algorithm for Parallel Matrix Multiplication*

Hyuk-Jae Lee

Department of Computer Science
Louisiana Tech University
Ruston, LA 71272, USA
hlee@engr.latech.edu

James P. Robertson and José A.B. Fortes

School of Electrical and Computer Engineering
Purdue University
W. Lafayette, IN 47907, USA
{robertso,fortes}@ecn.purdue.edu

Abstract

Cannon's algorithm is a memory-efficient matrix multiplication technique for parallel computers with toroidal mesh interconnections. This algorithm assumes that input matrices are block distributed, but it is not clear how it can deal with block-cyclic distributed matrices. This paper generalizes Cannon's algorithm for the case when input matrices are block-cyclic distributed across a two-dimensional processor array with an arbitrary number of processors and toroidal mesh interconnections. An efficient scheduling technique is proposed so that the number of communication steps is reduced to the least common multiple of P and Q for a given $P \times Q$ processor array. In addition, a partitioning and communication scheme is proposed to reduce the number of page faults for the case when matrices are too large to fit into main memory. Performance analysis shows that the proposed generalized Cannon's algorithm (GCA) requires fewer page faults than a previously proposed algorithm (SUMMA). Experimental results on Intel Paragon show that GCA performs better than SUMMA when blocks of size larger than about (65×65) are used. However, GCA performance degrades if the block size is relatively small while SUMMA maintains the same performance. It is also shown that GCA maintains higher performance for large matrices than SUMMA does.

1 Introduction

Matrix multiplication is one of the most important basic operations in scientific computations and many algorithms for parallel matrix multiplication have been proposed. An early development is Cannon's algorithm [6] which is memory-efficient and suitable for toroidal mesh interconnections. The original algorithm assumes that the input matrices are block-distributed, but it is not clear how to efficiently implement this algorithm for block-cyclic (block-scattered) distribution. This paper revisits Cannon's algorithm and generalizes it to the case when input matrices are block-cyclic distributed. An efficient schedule for reducing communication overhead is also proposed. The efficiency of the proposed algorithm is analyzed and comparatively evaluated thru experiments conducted on an Intel Paragon.

Extensive research has focused on development of parallel matrix multiplication subroutines [4]-[11]. Communication and memory overheads are two major obstacles for these parallel subroutines. To reduce communication overheads, various ideas have been explored. One important approach is to overlap communications with computations. Another technique tries to increase the granularity of computations between data exchanges so that the number of communications can be minimized. Both solutions benefit if the communication is done by using communication primitives that can be efficiently implemented by the target architecture. For example, if an architecture requires more overhead for broadcasts than for shifts, then it is better to use the latter instead of the former if possible (and vice versa). Memory overhead increases traffic between different levels of memory hierarchies and therefore is another important source of degradation.

Previously proposed parallel multiplication algorithms fall into two broadly defined classes. One class relies on using broadcast communication primitives (possibly using shift primitives as well). Algorithms included in this class are BMR (Broadcast-Multiply-Roll) [1, 2], PUMMA (Parallel Universal Matrix Multiplication Algorithm) [4], and SUMMA (Scalable Universal Matrix Multiplication Algorithm) [5]. The other class does not use broadcasts but relies exclusively on shifts. This class includes Cannon's algorithm and systolic matrix multiplication [2],[3],[6].

* This research was partially funded by the National Science Foundation under grants MIP-9500673 and CDA-9015696.

ICS 97 Vienna Austria

The original Cannon's algorithm [6] assumes a skewed distribution of the input matrices and consists of repeated shift-and-multiply steps. The following example gives a clear picture of the original Cannon's algorithm for the case when all matrices are square.

Example 1 Consider the matrix multiplication algorithm which computes $C = A \times B$. Suppose that the size of a given processor array is (6×6). In Cannon's algorithm, A and B are decomposed into 36 blocks in a (6×6) arrangement, which are initially aligned and multiplied by each other as shown in Fig. 1. The next step is to shift A to the left and B up to neighbor processors where block-wise multiplication can take place again and its result can be added to the current value of $C_{i,j}$. This shift and multiply step is repeated until all blocks in a row of A are multiplied by all blocks in a column of B. The data distributions at $t = 0, 1,$ and 3 are shown in Fig. 1. ■

Two advantages of Cannon's algorithm for square matrices are readily apparent. First, all processors are busy in all iterations. Second, the only communication necessary is the initial skew of the input matrices and the shifting of blocks in each iteration. The initial skew can be eliminated by allocating matrices initially in a skewed manner. A data distribution independent computation and active library approach have been studied for development of a source-to-source translator that automatically allocates data arrays as required by given subroutines [13]. In machines where a toroidal mesh-interconnection (virtual or real) is assumed for processors, shifts are highly efficient and therefore Cannon's algorithm performs very well. The first question addressed by this paper is whether Cannon's algorithm can be generalized to work with matrices of arbitrary shape and size initially stored according to arbitrary block-cyclic distributions and arbitrary mesh ratio (size, shape, and blocks must still obey certain restrictions that guarantee matrix-multiplication to be well-defined). The second question is whether and when the generalized Cannon's algorithm offers performance advantages over broadcast-based algorithms, namely, SUMMA. Both questions are given positive answers supported by analysis and experimental results.

The rest of this paper is organized as follows. Section 2 develops the generalized Cannon's algorithm (GCA). An efficient method to reduce communication frequency is also proposed. Section 3 develops a partitioning scheme to reduce memory faults. Section 4 evaluates the proposed technique against previous work. Experimental results obtained with an Intel Paragon are provided in Section 5. Section 6 presents conclusions of the paper.

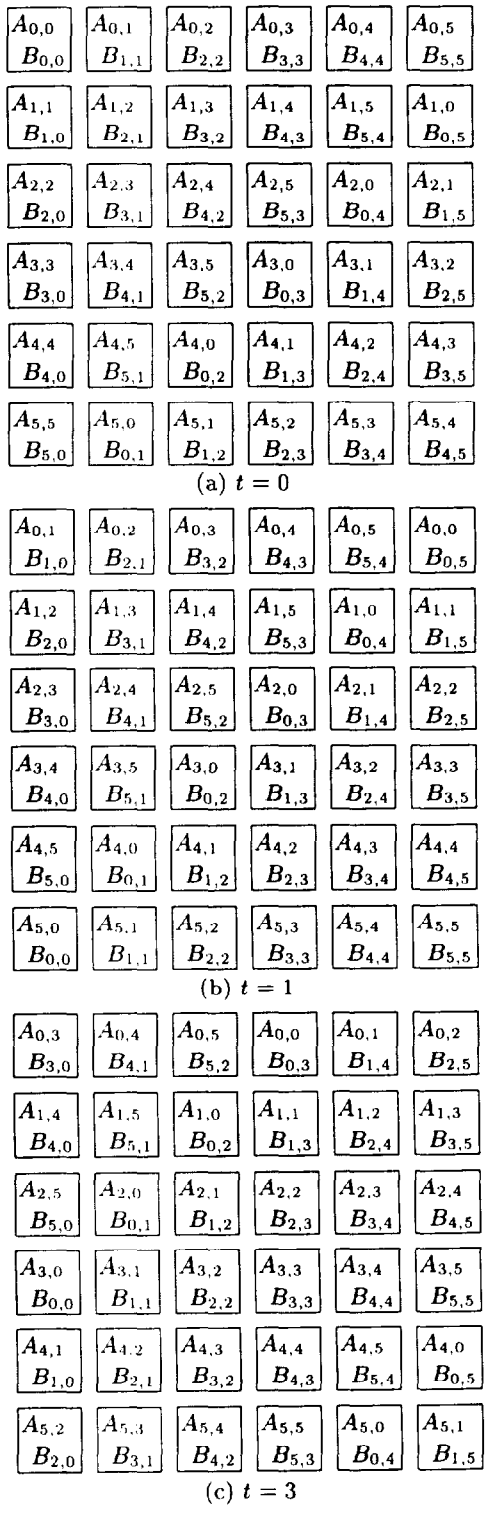


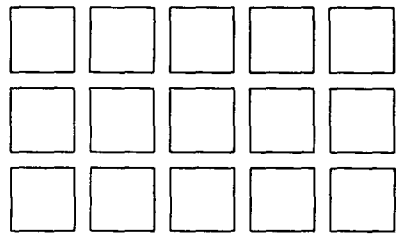
Figure 1: Data distribution for Cannon's algorithm

2 Generalized Cannon's Algorithm (GCA)

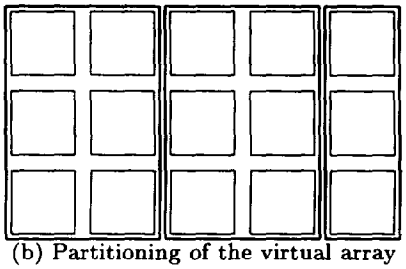
This section proposes a block-cyclic version of Cannon's algorithm and a scheduling scheme that reduces the number of communications. Throughout the section, matrix multiplication $C = AB$ is considered where matrices A, B, and C are partitioned into $(b\_x \times b\_z)$, $(b\_z \times b\_y)$, and $(b\_x \times b\_y)$ blocks, respectively. In its simplest form, Cannon's algorithm requires that the size of the processor array be the same as the partition size (i.e., the number of blocks) of matrix C $(= b\_x \times b\_y)$. Let such an array be called *the virtual processor array*. However, in general, the available physical processor array is smaller than the virtual processor array. Hence, the virtual array needs to be partitioned to be allocated into the available physical array. A single physical processor needs to store many data blocks and perform computations with these data blocks. Hence, the global indices of matrix blocks should be carefully converted into the local block indices in each processor.

Consider the case when the virtual processor array is not square. For example, suppose that matrices A, B, and C are decomposed into 6, 10, and 15 blocks, in a (3×2), (2×5), and (3×5) arrangement, respectively. Then, the virtual processor array is 3×5 (Figure 2 (a)). The size of the virtual processor array is larger than that of the partition of matrix A. In this case, the virtual processor array needs to be decomposed into subarrays of the same size as the partition of matrix A ((3×2) in this example). Figure 2 (b) shows the decomposition of the virtual processor array. Blocks of matrix A are skewed in the same way as Cannon's algorithm, and mapped into each of these subarrays (Figure 2 (c)). Note that the third subarray is smaller than the matrix A partition, and therefore, more than one block is stored in a single processor. With this mapping, three copies of matrix A exist in the virtual processor array. For matrix B, the virtual processor array is decomposed into (2×5) subarrays and blocks of matrix B are mapped into each of these subarrays (Figure 2 (d)). With these distributions of matrices A and B, all blocks are well-aligned and therefore multiplications of A and B can take place for every processor. In the next step, matrix A shifts left and matrix B shifts up (Figure 2 (e)). Then, again matrices A and B are well aligned, and all processors can compute matrix product. In general, the virtual processor array needs to be decomposed into subarrays with the size of $(b\_x \times b\_z)$, for distribution of matrix A. For matrix B, the virtual processor array needs to be decomposed into subarrays with the size of $(b\_z \times b\_y)$. Then, the multiply and shift steps are repeated for `b_z` times.

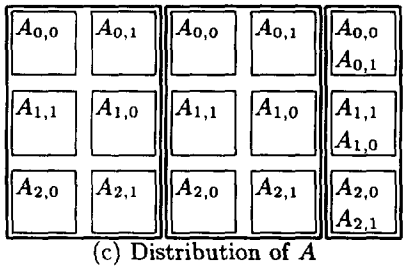
Consider the allocation of the virtual processor array to a physical processor array. Recall that multiple copies of matrices A and B exist in the virtual pro-



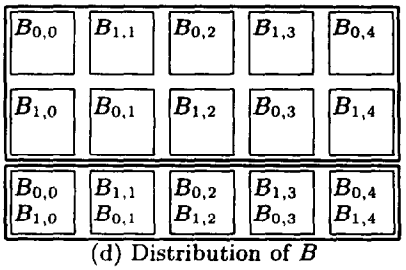
(a) 3×5 virtual processor array



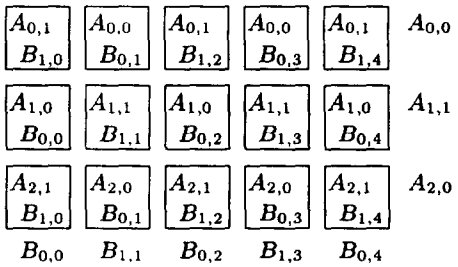
(b) Partitioning of the virtual array



(c) Distribution of A



(d) Distribution of B



(e) Distribution of A and B after data multiplication and shift step

Figure 2: Data distribution for GCA

cessor array. For example, in Figure 2 (c), processors $(0,0)$, $(0,2)$ and $(0,4)$ contain $A_{0,0}$. By allocating these three processors to one physical processor, just one copy of $A_{0,0}$ is maintained in the physical processor array. In general, all virtual processors horizontally separated by `b_z` processors contain the same block of matrix A while all virtual processors vertically separated by `b_z` processors contain the same block of matrix B. Hence, all processors separated by `b_z` processors either horizontally or vertically should be mapped into the same physical processor. In this way, only one copy of input matrix is stored in the physical processor array.

Upon mapping the virtual processors onto physical ones it may be possible to eliminate some virtual communication by mapping communicating virtual processors into the same physical one. Since the size of the physical processor is small, some data movement in the virtual processor array does not require actual data movement in the physical processor array. For example, suppose that the virtual processor array is 6×6 while the physical processor array is 3×3. Then, three left-shift steps in the virtual processor array correspond to no data movement in the physical processor array. By taking advantage of this basic idea, one can reduce the number of communications.

Example 1 (continued) Figure 1 (c) shows the data storage of matrices A and B after three steps of data movement. Note that the data blocks stored in each processor are the same as the initial data distribution (in Figure 1 (a)). Hence, no communication is necessary if the computations with these blocks are performed right after the computations with initial data storage. For matrix multiplication, all operations are commutative, and therefore it is possible to change the order of computations. Hence, by performing the computations as shown in Figure 1 (c) immediately after those in Figure 1 (a), one can eliminate one global communication step. Any pair of computation stages separated by three shifts can be computed without communication. Hence, significant communication overhead can be reduced by reordering computations in such a way that two computation stages separated by three shifts must always be executed consecutively. ■

The above example gives an idea of how to reduce the number of communications by changing the execution order. For the 3×3 processor array, three shifts return data to the original processor source. In general, given a $P \times Q$ processor array, $lcm(P,Q)$ shifts return the data to the original processor, where $lcm(P,Q)$ denotes the least common multiple of P and Q. Hence, computations on the initial data distribution and computations on the data distribution that result from $lcm(P,Q)$ data movements can be done without communications. Therefore, to reduce communications, the execution order should be changed in such a way that all computations that need $lcm(P,Q)$ data movements in the virtual processor array are performed consecutively.

The pseudocode of GCA for matrix multiplication is shown in the following program. For simplicity, this program assumes that both P and Q divide `M,N,` and `K`, respectively. In this code, `C_{loc}[i][j]`, `A_{loc}[i][j']`, and `B_{loc}[i'][j]` denote the matrices of A, B, and C, stored in local memory. Variables `i'` and `j'` need to be modified when data shifts cause the change of storage location in a local memory.

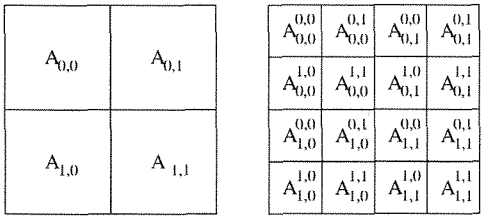
```
{
 lcm = least_common_multiplier(P,Q);
 for(t=0; t<lcm; t++) {
  for(i=0; i< (M/P); i++) {
   for(j=0; j < (N/Q); j++) {
    for(l=0; l < (K/lcm); l++) {
     j'=(j%k+l*lcm/Q)%(k/Q);
     i'=(i%k+l*lcm/P)%(k/P);
     C_{loc}[i][j]+=A_{loc}[i][j']
                   *B_{loc}[i'][j];
  }}}
 shift_west(A_{loc}[i][j'],0<=i<M/P,0<=j'<k/lcm);
 shift_north(B_{loc}[i'][j],0<=i'<k/lcm,0<=j<N/Q);
 }
}
```

The number of communications required by Cannon's algorithm is `b_z` while that by the proposed scheme is $lcm(P,Q)$. In general, `b_z` is much larger than P or Q. Hence, the proposed scheme can greatly reduce communication overhead.

3 Partitioning

In this section, a new partitioning scheme is proposed to reduce the number of page faults. The basic idea is explained with Figure 3. Suppose that there are 2×2 processors. Matrix A is decomposed into 2×2 submatrices as shown in Figure 3 (a) and each block is stored in one processor. In the Cannon's algorithm, each processor completes all computations with its local matrix, and then shifts the local matrix to the left processor. This computation and shift step repeats until the local matrix moves back to the original processor. If a matrix is too large to fit into main memory, page faults occur at each computation and shift step.

In the new scheme, the submatrix in each processor is decomposed again into submatrices in such a way that each submatrix fits into main memory. In Figure 3 (b), each submatrix is partitioned into 2×2 submatrices, and assume that each of the submatrices



(a) Initial partitioning of matrix A (b) Partitioning of submatrices

Figure 3: Partitioning of matrix

fits into main memory. Initially, all processors have its own local submatrix of A ($A^{0,0}_{i,j}$) and performs computations on the submatrix. Once these computations are finished, each submatrix $A^{0,0}_{i,j}$ is sent to the left processor so that all processors perform other computations with the new submatrix received from the right processor without any disk read. The computation and shift step repeats until the submatrices move back to the original processors. Therefore, submatrices $A^{0,0}_{i,j}$ are used by all necessary processors before they are paged out of main memory. Once all computations with $A^{0,0}_{i,j}$ are completed, submatrices $A^{0,1}_{i,j}$ are loaded into main memory and then used for computation and shift. This procedure continues until all submatrices are loaded and used for computations. In this way, many disk reads can be eliminated.

Consider how to partition matrices and reduce page faults in a single processor. Suppose that matrices A,B, and C are initially allocated to main memory as shown in Figure 4 (a) where a shaded region represents the part of a matrix that is allocated to main memory. In this example, $N' \times N$ submatrices are allocated for all A, B, and C. Figure 4 (b) illustrates the partitioning of the matrices. The upperleftmost block ($C_{0,0}, A_{0,0}$, and $B_{0,0}$) is $N' \times N'$ square submatrix. Other blocks are also shown in Figure 4 (b). Figure 4 (c) shows the parts of matrices which are involved in computation $C_{0,0} = C_{0,0} + A_{0,0} \times B_{0,0}$ (dark shaded region). As in (a), the light shaded regions still represent the part allocated to main memory. Once this part of computation is completed, $A_{0,0}$ shifts left and $B_{0,0}$ moves up to the next processors, respectively. This computation and shift step repeats until $A_{0,0}$ and $B_{0,0}$ move back to the original processor. Then, the next computation $C_{0,1} = C_{0,1} + A_{0,1} \times B_{1,0}$ is performed. The dark shaded regions in Figure 4 (d) represent the submatrices involved in this computation. Figure 4 (e), (f), $\cdots$, (i), and (j), show the order of remaining computations.

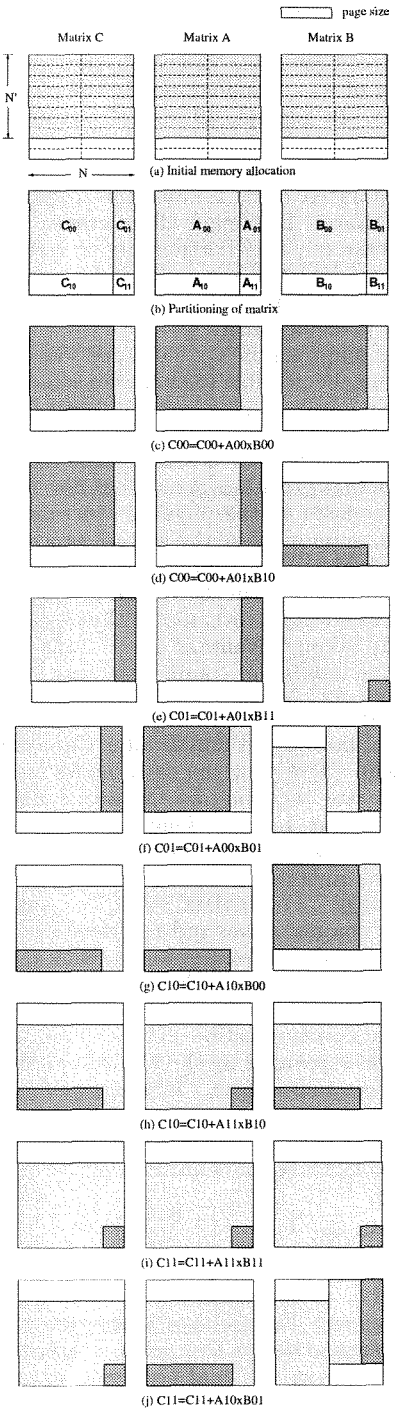


Figure 4: Partitioning and scheduling for page fault reduction

4 Page Fault Analysis of Matrix Multiplication Algorithms

In this subsection, the number of page faults is estimated for both GCA and SUMMA. Figure 4 is used to estimate the number of page faults made by GCA. For simplicity, assume that A, B, and C are $N \times N$ matrices. Let P_{size} be the size (in bytes) of a single page of main memory and M_{size} be the size (in bytes) of main memory. Let $N' \times N'$ be the size of $C_{0,0}$ in Figure 4 (b). For simplicity, assume that each element of a matrix takes one byte of memory space. Initially, $C_{0,0}$ and $C_{0,1}$ are allocated to main memory. At the computation in Figure 4 (g) and (i), $C_{1,0}$ and $C_{1,1}$ are loaded into main memory, respectively. Therefore, a number of page faults occur in these steps. The total number of elements to be loaded in these steps is $(N - N') \times N$. The number of page faults is $\lceil((N - N') \times N)/P_{size}\rceil$. For simplicity, the 'ceiling' operation is omitted in the remaining analysis. Thus, the number of page faults is $((N - N') \times N)/P_{size}$. Matrix A has exactly the same number of page faults as C does.

The estimation of page faults for B is more complex. The computations in Figure 4 (d) and (h) cause $((N - N') \times N)/P_{size}$ page faults, respectively. The computations in Figure 4 (f) and (g), cause $((N-N') \times N)/P_{size}$ page faults. The computation in Figure 4 (j) also causes $(N - N')(N - N')/P_{size}$ page faults. Hence, the total number of page faults with B is $3((N - N') \times N)/P_{size} + (N - N')(N - N')/P_{size}$. Therefore, the total number of page faults is

$$\begin{aligned} Pf_{GCA}(N) &= 5((N - N') \times N)/P_{size} \\ &+ (N - N')(N - N')/P_{size}. \quad (1) \end{aligned}$$

For SUMMA, each processor has different memory allocation at a given computation step. Figure 5 shows the memory allocation of Processor (0,0). This figure is used to estimate the number of page faults in Processor (0,0). Other processors have almost the same number of page faults. As shown in Figure 5 (a), SUMMA requires two work spaces for A and B, respectively. The size of the work space is $n_b \times N$ for both matrices where n_b is to be chosen by a subroutine developer. To start computation, matrices are allocated to main memory as shown in Figure 5 (b). For C, all elements are loaded. Hence, $((N - N') \times N)/P_{size}$ page faults occur. For A, the first n_b columns are loaded to the work space. Hence, $(N - N')(N - N')/P_{size}$ page faults occur to access A and $(n_b \times N)/P_{size}$ page faults occur to load data into the work space. For B, the first n_b rows are loaded to the work space. Since these rows are initially stored in main memory, no page fault occurs for B. For the access of the work space for B, another $(n_b \times N)/P_{size}$ page faults occur. For the remaining computation,

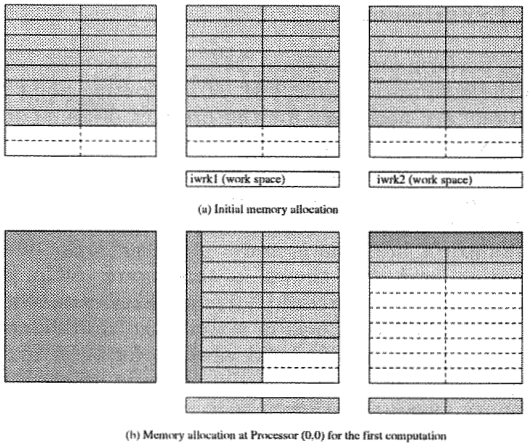


Figure 5: Memory allocation for SUMMA

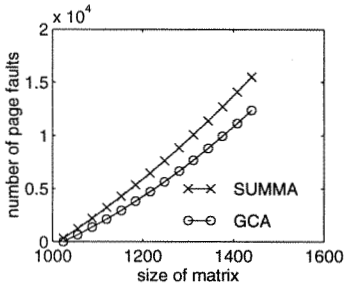


Figure 6: The estimated number of page faults for GCA and SUMMA

all elements of A and C are visited once. The total number of elements in A, B, C and two work spaces is $(3N^2 + 2n_bN)$. The number of elements in outside of main memory is $(3N^2+2n_bN)-M_{size}$. Therefore, approximately $((3N^2+2n_bN)-M_{size})/P_{size}$ page faults occur for the remaining of the computation. Hence, the total number of page faults is

$$\begin{aligned} Pf_{SUMMA}(N) &= ((N - N') \times N)/P_{size} + \\ &(N - N')(N - N')/P_{size} + 2(n_b \times N)/P_{size} \\ &+((3N^2 + 2n_bN) - M_{size})/P_{size}. \quad (2) \end{aligned}$$

Figure 6 compares the estimated page faults for GCA and SUMMA. In this estimation, it is assumed that the memory size is 3 mega bytes, and the page size is 256 bytes. For SUMMA, n_b is 20 as suggested in [5]. As shown in this analysis, GCA causes fewer page faults than SUMMA.

5 Experiments

As a performance measure, MFLOPS are used based on the assumption that $2\ M\ N\ K$ floating point operations are necessary for the product of an $M \times K$ matrix by a $K \times N$ matrix.

Figure 7 illustrates the number of MFLOPS vs. the domain size. The Y-axis shows the number of MFLOPS and the X-axis shows the size of A and B. All matrices are square, and a processor array of size 8×8 is used. As shown in this figure, GCA performs better than SUMMA for all sizes of matrices. For small matrices, this is mainly due to differences in communication overhead whereas for large matrices, it is a consequence of difference in efficiency of memory usage. As shown in this figure, the number of MFLOPS decreases drastically when local blocks no longer fit into memory, and must be paged to disk. Since SUMMA uses more memory than GCA, its performance dropoff occurs earlier than that of GCA.

Figure 8 compares scalability of GCA and SUMMA. The X-axis shows the size of processor array and the Y-axis shows MFLOPS. The processor grid is always square, therefore 64 processors are arranged as an 8×8 array. Each processor has one block of size 256×256. Both algorithms maintain quite a good performance as the processor size increases. Again though, GCA is more efficient than SUMMA.

Figure 9 illustrates the number of MFLOPS vs. the number of processors. As the experiment of Figure 8, the processor grid is always square. However, in this experiment, the matrix sizes are fixed at 720×720. As the number of processors increases, the communication time becomes a larger factor in the program execution time, and this causes the number of MFLOPS to drop.

Figure 10 illustrates the number of MFLOPS for varying block sizes. For this figure, the Y-axis shows the number of MFLOPS and the X-axis shows the block size for each processor. The matrix sizes were fixed at 2048×2048. The processor array was fixed at 8×8. This means that for a block size of 256, each processor has one 256×256 block. For a block size of 2, each processor has 128 (2×2) blocks. In order for SUMMA to have the best possible performance, the number of columns of blocks of B in each processor is the same, and the number of rows of blocks of A in each processor is the same. This allows each processor to have the same number of blocks of C (since the number of blocks of C is equal to the number of rows of blocks of A by the number of columns of blocks of B). In this manner, an entire row or column of blocks can be broadcasted at one time. This is why SUMMA's times are constant for varying block sizes. However, MFLOPS of GCA decreases drastically as the block size decreases. If the block size is smaller than about sixty five, SUMMA performs better than GCA.

The result in Figure 10 shows that it is desirable to choose the largest blocks. However, if the block size is too large, the number of blocks is so small that it is impossible to evenly distribute these blocks across the processor array. Therefore, performance is degraded by poor load balancing. Hence, it is best to choose the largest block size among those which guarantee full utilization of processors.

Figures 11 and 12 illustrate the number of MFLOPS when an 8×4 processor array and an 8×7 processor array are used, respectively. For a given matrix $A(M \times K)$ and matrix $B(K \times M)$, the block size is chosen to be $M/8 \times K/8$ and $K/8 \times N/4$ for matrix A and B, respectively, in experiment of Figure 11. On the other hand, the experiment of Figure 12 chooses the block size to be $M/8 \times K/56$ and $K/56 \times N/7$ for matrix A and B, respectively. These block sizes are the largest possible sizes which allow full processor utilization. In Figure 11, GCA achieves higher MFLOPS than SUMMA. However, in Figure 12, GCA achieves less number of MFLOPS than SUMMA. This is because a small block size is used for full processor utilization.

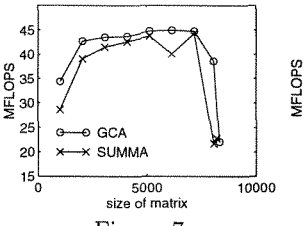


Figure 7

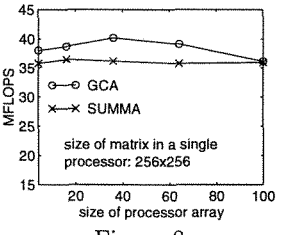


Figure 8

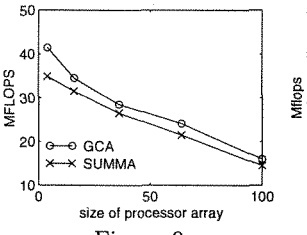


Figure 9

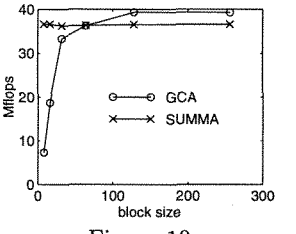


Figure 10

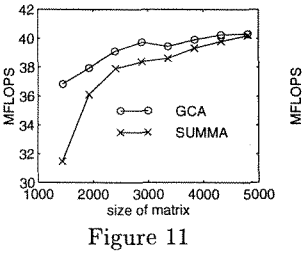


Figure 11

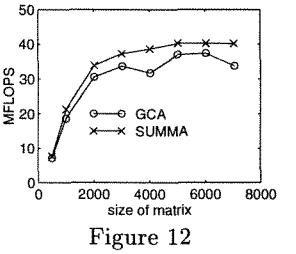


Figure 12

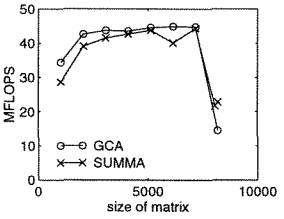


Figure 13

Figure 13 shows the number of MFLOPS with the assumption that input matrices are distributed across processor array as required by SUMMA. In this case, GCA requires initial data movements to redistribute data as required by Cannon's algorithm. Hence, the execution time for GCA is the initial data redistribution time plus the computation time. Figure 13 shows that the MFLOPS of GCA is smaller than that of SUMMA even though the initial data redistribution is required for GCA. This is because the initial redistribution time is much smaller than the computation time.

6 Conclusions

The main contribution of this paper is the generalization of Cannon's algorithm to block-cyclic distributed input matrices. For parallel computers with toroidal mesh interconnections, Cannon's algorithm can be efficiently implemented. The generalized Cannon's algorithm (GCA) has the same advantages as Cannon's algorithm and therefore it is more efficient than other algorithms developed for block-cyclic distributed matrices. Efficiency of GCA degrades when the block size is small. Improvement of GCA for these cases is under investigation. When matrices are too large to fit into main memory, performance degrades due to page faults. A partitioning and scheduling scheme is proposed to reduce page faults. Analysis shows that the performance degradation of GCA with the proposed partitioning is much less than that of SUMMA. Future work includes implementation of the proposed partitioning scheme and validation of the analysis through experiments.

References

[1] G.C. Fox, S.W. Otto, and A.J.G. Hey, "Matrix algorithms on a hypercube I: matrix multiplication," *Parallel Computing,* vol. 4, pp. 17-31, 1987.

[2] S. Huss-Lederman, E.M. Jacobson, A. Tsao, and G. Zhang, "Matrix multiplication on the Intel Touchstone Delta," *Concurrency: Practice and Experience,* vol 6, no. 7, pp. 571-594, Oct. 1994.

[3] K.K. Mathur and S.L. Johnsson, "Multiplication of matrices of arbitrary shape on a data parallel computer," *Parallel Computing,* vol. 20, pp. 919-951, July 1994.

[4] J. Choi, J.J. Dongarra, and D.W. Walker, "Pumma: Parallel universal matrix multiplication algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience,* vol 6, no. 7, pp. 543-570, Oct. 1994.

[5] R. van de Geijn and J. Watts, "SUMMA: Scalable universal matrix multiplication algorithm," University of Texas, Department of Computer Sciences, Tech. Rep. TR-95-13, April 1995. Also: LAPACK Working Note #96, May 1995.

[6] L.E. Cannon, "A cellular computer to implement the Kalman filter algorithm," Ph.D. dissertation, Montana State Univ., Bozeman, MT, 1969.

[7] R.C. Agarwal, F.G. Gustavson, and M. Zubair, "A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication," *IBM Journal of Research and Development,* vol. 38, no. 6, pp. 673-681, 1994.

[8] S.L. Johnsson, "Communication efficient basic linear algebra computations on hypercube architectures," *J. Parallel Distributed Computing,* vol 4, no. 2, pp. 132-172, April 1987.

[9] P. Bjørstad, F. Manne, T. Sørevik, and M. Vajteršic, "Efficient matrix multiplication on SIMD computers," *SIAM J. Matrix Anal. Appl.,* vol. 13, no. 1, pp. 386-401, Jan. 1992.

[10] S. Huss-Lederman, E.M. Jacobson, and A. Tsao, "Comparison of scalable parallel matrix multiplication libraries," in *Proc. of the Scalable Parallel Libraries Conference,* Oct. 1993.

[11] A. Chtchelkanova, J. Gunnels, G. Morrow, J. Overfelt, and R.A. van de Geijn, "Parallel Implementation of BLAS: General Techniques for Level 3 BLAS," Univ. of Texas, Dept. of Computer Sciences, Tech. Rep. TR-95-40, Oct. 1995.

[12] J. Choi, J.J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C. Whaley, "A proposal for a set of parallel basic linear algebra subprograms," LAPACK Working Note #100, May 1995.

[13] H.-J. Lee and J.A.B. Fortes, "Toward data distribution independent parallel matrix multiplication," in *Proc. Int. Parallel Processing Symposium,* pp. 436-440, April 1995.