Memory Access Optimizations for High-Performance Computing

TR 93-02
Jeffrey S. Clary and S.C. Kothari

January 13, 1993

Iowa State University of Science and Technology
Department of Computer Science
226 Atanasoff
Ames, IA 50011

Memory Access Optimizations for High–Performance Computing

Jeffrey S. Clary
S. C. Kothari

Iowa State University
Department of Computer Science
Ames, IA 50010

clary@iastate.edu
kothari@cs.iastate.edu

Abstract

This paper discusses the importance of memory access optimizations which are shown to be highly effective on the MasPar architecture. The study is based on two MasPar machines, a 16K–processor MP–1 and a 4K–processor MP–2. A *software pipelining* technique overlaps memory accesses with computation and/or communication. Another optimization, called the *register window* technique reduces the number of loads in a loop. These techniques are evaluated using three parallel matrix multiplication algorithms on both the MasPar machines. The matrix multiplication study shows that for a highly computation intensive problem, reducing the interprocessor communication can become a secondary issue compared to memory access optimization. Also, it is shown that memory access optimizations can play a more important role than the choice of a superior parallel algorithm.

Keywords: load/store architecture, memory accesses, matrix multiplication, parallel programming.

1 Introduction

This paper discusses the importance of memory access optimizations to achieve high performance on the MasPar architecture. The study is based on two MasPar machines, a 16K–processor MP–1 and a 4K–processor MP–2.

On MasPar computers, the local memory at a processing element (PE) is slow, so memory access time can cause a significant performance loss. We describe memory access optimization techniques that minimize such performance loss. The main optimization technique is called *software pipelining*. It reduces the average cost of a memory access. The technique can be effectively applied to computation loops containing floating point operations along with memory accesses. It is also applicable to communication loops that move a block of data from the local memory of one PE to another PE. In software pipelining, the loops are programmed to pipeline the computation (or communication) operations of one iteration with the memory accesses to load the data for the computation (or communication) of the next iteration of the loop. Being overlapped with other useful operations, memory accesses are partly hidden, thus reducing the overall time for the execution of a program.

The *register window* technique is another important optimization. It is used to reduce the number of loads in a loop. The technique involves reprogramming a loop so that a set of W registers, called a register window, covers a block of W elements of an array. A new inner block is added to the loop to perform computations of W elements in the register window. In an example of matrix multiplication C = A*B, this technique is used to modify the loop for multiplying submatrices at a PE. By using a register window to cover W elements of C from a row, the number of memory accesses for loading the matrix A is reduced by a factor of W.

As a part of this study, three parallel matrix multiplication algorithms were implemented on two MasPar machines. The first algorithm uses a parallel prefix addition. The second algorithm is a systolic algorithm using nearest neighbor communication. The third is an algorithm requiring

broadcasting, which is slower than nearest–neighbor communication on MasPar. For the multiplication of two NxN matrices on an NxN processor array, the execution times of the systolic and the broadcast algorithms are both of order N and the execution time of the parallel prefix algorithm is of order N log N. We show that on MasPar machines memory access optimizations can play a more important role than the choice of a superior parallel algorithm. An interesting result is that a software pipelined implementation of the parallel prefix algorithm outperforms a plain implementation of the superior systolic algorithm. However, after memory access optimizations are applied the systolic algorithm is the best, as expected.

A detailed performance analysis provides insights into the impact of memory access optimization techniques, performance of parallel matrix multiplication algorithms on the MasPar architecture, and the relative performance of MP–1 and MP–2 computers on a highly computation intensive problem such as matrix multiplication. Interprocessor communication turns out to be a secondary issue compared to memory access optimization. Optimization techniques described in this paper are shown to provide substantial performance improvements beyond improvements possible through compiler optimization. Section 2 is a brief description of MasPar computers, section 3 describes three parallel matrix multiplication algorithms used in this study, section 4 describes optimization techniques, section 5 provides the performance analysis, and conclusions are presented in section 6.

2 MasPar MP–1 and MP–2

The MasPar MP–1 and MP–2 computers are based on a single–instruction stream, multiple data stream (SIMD) architecture with processing elements (PEs) arranged in a 2D toroidal grid. A parallel program runs on the array control unit (ACU) which broadcasts instructions to the PEs. Each PE can communicate with any of its 8 nearest neighbors using fast *xnet* communication, and arbitrary communication patterns can be implemented using the slower *router* communication.

The MP–1 and MP–2 can have from 1K to 16K processors. Both machines have a clock rate of

12.5 MHz, and the same instruction set. However, the MP–1 uses 4–bit processors while the MP–2 uses 32–bit processors. The MP–2 can perform floating point operations four to five times faster than the MP–1. Measured cycle times for several instructions are shown in Table 1.

Operation	MP–1 Cycles	MP–2 Cycles
Load/Store	70	35
Xnet	43	50
Floating Point Multiply	225	41
Floating Point Add	127	26

Table 1: MasPar instruction cycle times for 32–bit operations

The PEs have no cache memory, but have forty 32–bit registers each. Memory accesses on each PE's local memory are done only through explicit load and store instructions. Other instructions, including interprocessor communication, are register based. The number of cycles for a memory operation is significant compared to other operations. The MasPar architecture includes a buffering mechanism which allows up to four pending memory accesses. The memory accesses can be overlapped with either computation or communication. Thus, the performance loss due to memory accesses can be minimized because the memory access time is ''hidden'' behind useful computation or communication and not seen as additional overhead.

This study used a 16K–PE MP–1 and a 4K–PE MP–2. The MP–1 has 16K bytes of local memory per processor compared to 64K bytes per processor for the MP–2. Thus, the total PE memory is the same on both the computers. The MP–2 used in this study has a higher raw floating point speed than the MP–1, since it has one–fourth as many PEs, each four to five times faster depending on floating point operations.

3 Matrix Multiplication Algorithms

Three parallel algorithms are described to calculate the product C of matrices A and B, each of size NxN. The processor array is assumed to be of size PxP. For simplicity, the algorithms are illustrated using a hypothetical 4x4 PE array, and for 4x4 A, B, and C matrices. Multiplication of matrices with length a multiple of the length of the PE array is performed using block decomposition. The algorithms remain the same except that on each processor, instead of scalar addition and multiplication, matrix addition and multiplication is performed on submatrices.

3.1 Algorithm 1: Parallel Prefix Sum

This algorithm requires loading the A matrix in normal order and the B matrix transposed. It produces the C matrix in normal order as shown in Figure 1. In each iteration, the algorithm computes N values of C. Notice that the parallel prefix sum requires $\log_2 N$ communication steps and as many addition steps. Communication hops of up to N/2 processors are required.

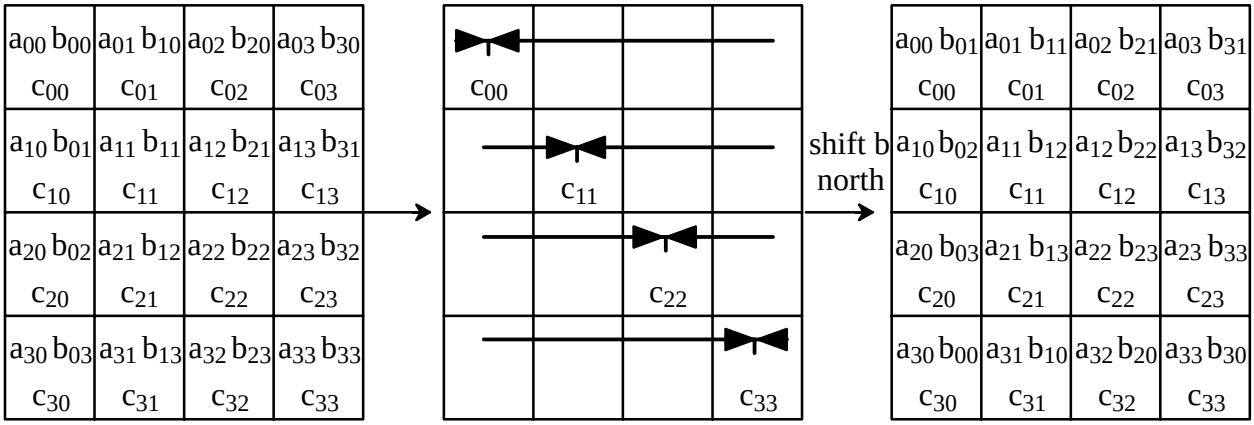


Figure 1: Illustration of matrix multiplication using parallel prefix sum

The algorithm is shown below:

For **i** = 0 to **P** – 1

STEP 1 (Multiplication): **ctemp** = **a** * **b**

STEP 2 (Parallel Prefix Sum): Sum **ctemp** in row **j** into **c** in column (**i**+**j**) mod **P**

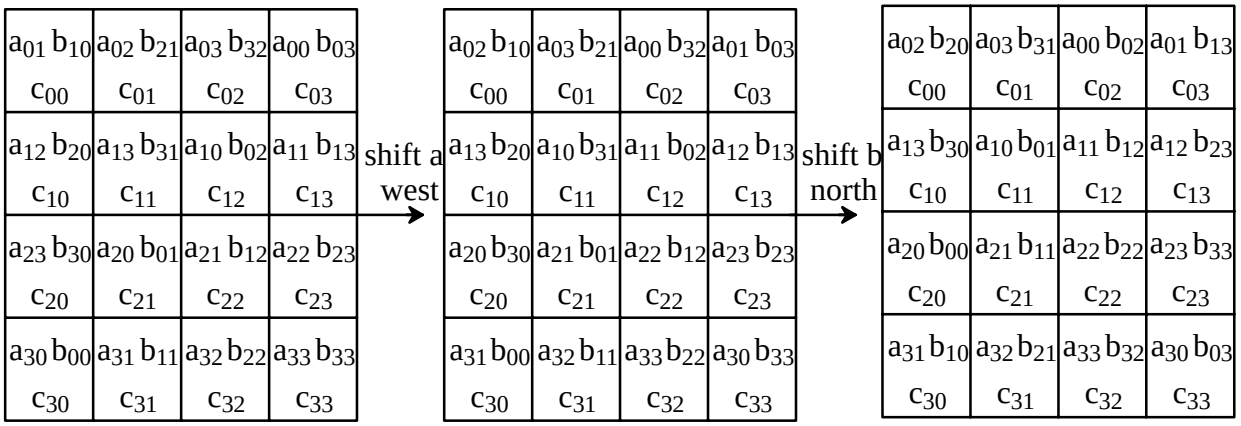
STEP 3 (Communication): Shift each **b** one processor north

In each iteration, the algorithm calculates nxproc values of C. On the 0th iteration the diagonal elements of C are calculated, on first iteration the elements one to the right of the diagonal are calculated, and so on.

3.2 Algorithm 2: Systolic Processing

This algorithm requires that the matrices A and B be initially loaded in a shifted order. Each row of A is shifted east until each diagonal element a_{ii} is on the eastmost edge of the processor array. Similarly, each column of B is shifted south until each diagonal element is on the southmost edge, as shown in Figure 2.

Starting from the initial layout, successive layouts are shown after each communication step in the first iteration of the loop. The value c_{ij} is computed on processor P_{ij}. For example, c_{00} is computed by calculating the products $a_{01}b_{10}$, $a_{02}b_{20}$, $a_{03}b_{30}$, and $a_{00}b_{00}$ and accumulating the sum in four successive iterations of the loop on processor P_{00}.



(a) Initial data layout (b) A matrix shifted west (c) B matrix shifted north

Figure 2: Illustration of matrix multiplication using the systolic method

The algorithm is shown below:

For **i** = 0 to **P**–1

STEP 1 (Multiplication): **ctemp** = **a** * **b**

STEP 2 (Addition): **c** = **c** + **ctemp**

STEP 3 (Communication): Shift each **a** one processor west

STEP 4 (Communication): Shift each **b** one processor north

This and other such systolic algorithms can be designed using the method described in [2].

3.3 Algorithm 3: Broadcast

This algorithm, reported by Fox [1], begins with matrices A, B, and C all stored in normal order. As in algorithms 1 and 2, c_{ij} is computed on processor P_{ij}. For example, c_{00} is computed by calculating the products $a_{00}b_{00}$, $a_{01}b_{10}$, and $a_{02}b_{20}$, and $a_{03}b_{30}$ and accumulating the sum in 4 successive iterations, as described in the algorithm below and illustrated in figure 3.

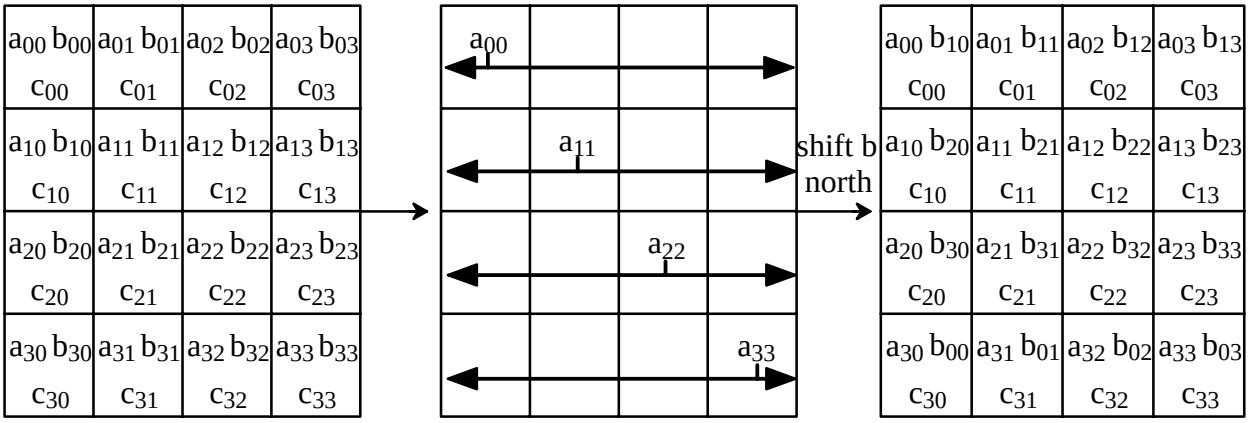
For **i** = 0 to **P**–1

STEP 1: (Broadcast) Broadcast **a** from column (**i** + **j**) mod **P** into **atemp** across each row **j** of processors

STEP 2: (Multiplication) **ctemp** = **atemp** * **b**

STEP 3: (Addition) **c** = **c** + **ctemp**

STEP 4: (Communication) Shift each **b** north one row



(a) Initial data layout (b) Broadcast of A matrix (c) B matrix shifted north

Figure 3: Illustration of matrix multiplication using broadcast

4 Memory Access Optimizations

As shown in Table 1, a PE memory access is an especially expensive operation on MasPar machines. Memory accesses are incurred in accessing data items for computation as well as for memory–to–memory communication between PEs. The total cost of memory accesses in a parallel program can be minimized by reducing the number of memory accesses and/or the average cost of each access. In this section we describe memory access optimization techniques which go beyond compiler optimization and require special programming. All of the psuedocode in this section represents operations performed simultaneously on all PEs, for submatrix multiplication.

4.1 Software Pipelining –– Reducing the Cost of Each Access

The cost of memory accesses can be reduced by ‘‘hiding’’ it behind other computation or communication operation; that is, by overlapping memory operations with other computation or communication using *software pipelining*. On the MasPar, if a PE memory load or store is followed by another operation that does not use the registers involved in the load or store, then the second operation can begin before the memory operation finishes. Consider the code fragment in Figure 4:

```
register a, b, c;

for i = 0 to M–1
    for j = 0 to M–1
        begin
        c = C(i,j)
        for k = 0 to M–1
            begin
            a = A(i,k)
            b = B(k,j)
            c += a * b
            end
        C(i, j) = c
        end
```

Figure 4: Basic submatrix multiply

Elements of the A and B arrays are used in floating point operations immediately after they are accessed, so the floating point operations cannot start until the memory accesses are complete.

But by suitable reprogramming of the loop, each fetch can be started some time before the data is actually needed, as shown in Figure 5:

```
register a0, a1, b0, b1, c;

for i = 0 to M–1
    begin
    for j = 0 to M–1
        begin
        c = C(i, j)
        a0 = A(i,0)
        b0 = B(0,j)
        for k = 0 to M–2 by 2
            begin
            a1 = A(i,k+1)
            b1 = B(k+1,j)
            c += a0 * b0
            a0 = A(i,k+2)
            b0 = B(k+2,j)
            c += a1 * b1
            end
        a1 = A(i,M–1)
        b1 = B(M–1,j)
        c += a0 * b0
        c += a1 * b1
        C(i, j) = c
        end
    end
```

Figure 5: Submatrix multiply with software pipelining

The second loop executes faster on the MasPar for two reasons. The main reason is that memory accesses are overlapped with floating point computations. The computations for c += a*b can be started while the immediately preceding accesses of arrays A and B are still in progress. Another relatively small gain is made because the inner loop is unrolled to a depth of 2, thus saving some loop overhead. In practice, code should be written so that four memory accesses are started at a time, since the MasPar architecture allows that many pending memory operations.

4.2 Register Window Technique – Reducing the Number of Accesses

The *register window* programming technique is used to reduce the number of memory accesses. We will describe the technique using the same example of submatrix multiplication C = A*B. As shown in the basic loop (Figure 4), ordinarily a register is assigned to an element of C which is accessed in all iterations of the inner loop. In this technique, a register window slides over the C matrix, keeping the first W elements from the row in registers, then the next set of W elements in registers, and so on.

The advantage of the register window technique comes from the fact that it allows the use of one load in place of W loads. In the basic matrix multiplication loop, one element of the C matrix is completely calculated at a time requiring accesses to elememts of the A matrix across a row and elements of the B matrix down a column. Using the register window technique, once an element of the A matrix is loaded into a register, it is used W times in computations of the W elements of the C matrix stored in the register window. Thus, having W elements of the C matrix in a register window reduces the number of loads for elements of the A matrix by a factor of W. A rearranged loop using a register window of size W is shown in Figure 6. To simplify the illustration, a set of registers for the register window is treated as an array c().

```
register a, b, c(W);

for i = 0 to M–1
    begin
    for j = 0 to M/W–1
        begin
        for p = 0 to W–1
            c(p) = C(i, j*W+p)
        for k = 0 to M–1
            begin
            a = A(i, k)
            for p = 0 to W–1
                begin
                b = B(j*w+l, k)
                c(p) += a * b
                end
            end
        for p = 0 to W–1
            C(i, j*W+p) = c(p)
        end
    end
```

Figure 6: Submatrix multiplication with register window technique

4.3 Reducing Memory Access Overhead for Communication

There is another way to reduce the number of memory operations in all of the parallel algorithms described above. Each algorithm performs some computations with the elements of A, B, and C on each processor, and then communicates A, B, or both in some direction. For example, the systolic algorithm computes C += A*B on each processor, and them moves the A submatrix each one processor to the west, and the B submatrix each one processor to the north.

For each element of a submatrix, there is a last computation involving the element. Immediately after that computation, the element is still in a register, and may be sent to the appropriate neighboring processor. Following this strategy makes it unnecessary to again fetch the element from memory later when it must be communicated. The receiving processor still must perform a store operation to store the element in memory.

5 Performance Analysis

The following formulas for the execution time of the three algorithms are useful in analyzing the performance of the algorithms and the effectiveness of optimizations.

$$T_1 = P\,[M^3(T_m + T_a + 2T_s) + M^2((\log P)(T_a + T_c) + T_x + 3T_s)\,]$$

$$T_2 = P\,[M^3(T_m + T_a + 2T_s) + M^2(2T_x + 6T_s)\,]$$

$$T_3 = P\,[M^3(T_m + T_a + 2T_s) + M^2(T_c + T_x + 6T_s)]$$

where

- N : length of the matrices A, B, and C
- P : length of the PE array
- M : length of a submatrix on a PE (N/P)
- T_m : time for floating point multiply
- T_a : time for floating point add
- T_s : time for memory load or store
- T_x : time for nearest–neighbor communication
- T_c : time for non–nearest–neighbor communication

These approximate formulas follow from the algorithm descriptions from section 3, but a few comments will make them clearer. The M^3 terms in each formula comes from the cubic operation of submatrix multiplication. The $2T_s$ associated with M^3 in each formula comes from loading elements of the A and B submatrix. The T_s associated with M^2 comes from loading and storing elements of the C submatrix, and the fact that communicating a matrix element from one processor to another requires a load at the source and a store at the destination.

Notice that the parallel prefix sum in algorithm 1 takes logarithmic time in the length of the PE array, not the length of the matrix multiplied. For any real machine, the log term becomes a constant (6 or 7 in this study). On a fixed–size machine, the behavior of the three algorithms is asymtotically identical as problem size grows. Indeed, any reasonable parallel algorithm based on normal serial matrix multiplication (i.e. not based on methods such as those proposed by Strassen, Winograd, etc. [3]) will have an execution time of order N^3. Reducing execution time becomes a problem of reducing constants.

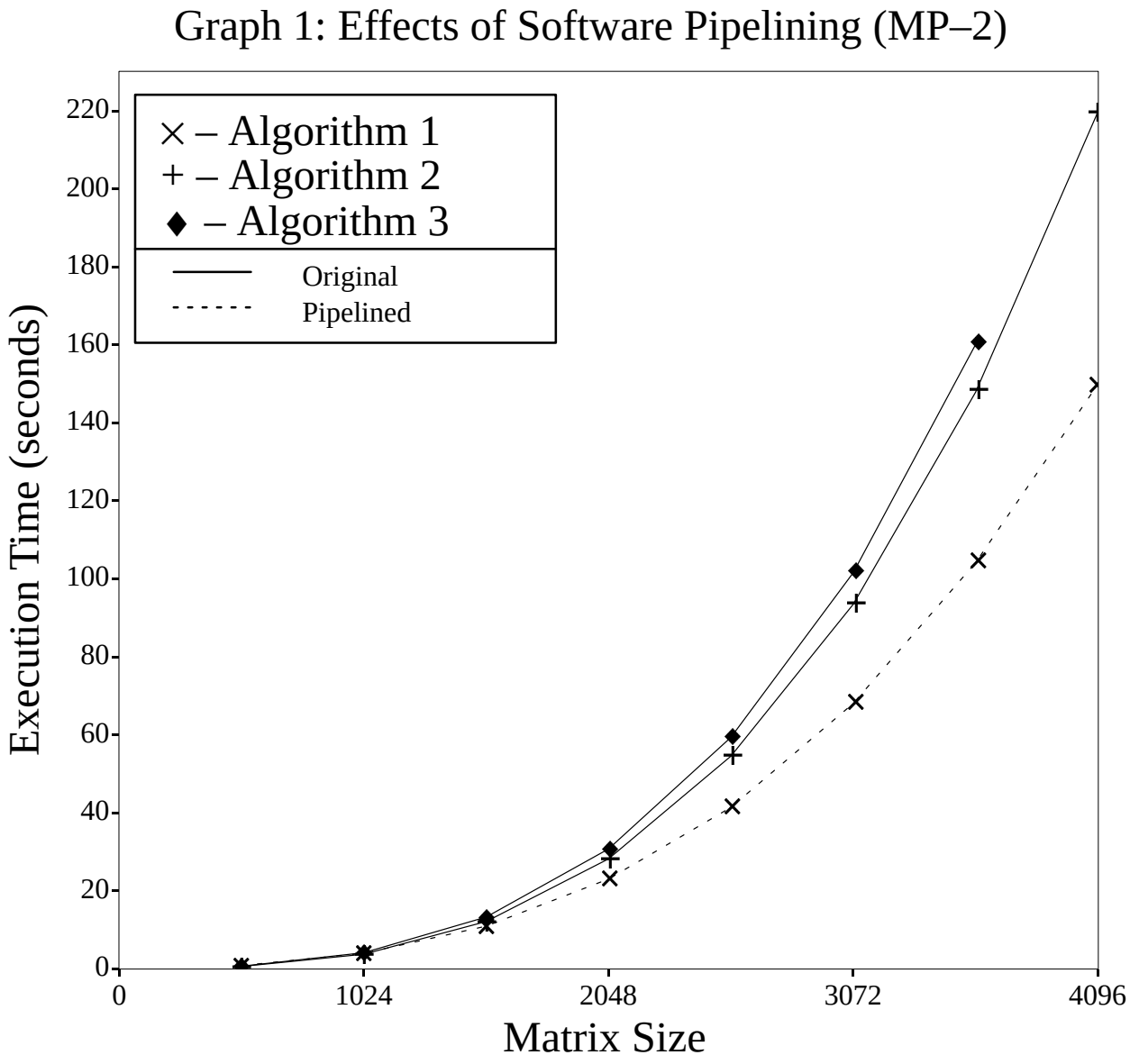
It turns out that reducing the execution time spent on memory accesses is crucial. A memory access takes a significant fraction of the time required for a floating point operation, and the formulas above show that memory access times appear with the cubed terms. Table 2 shows the percentage of execution time saved by applying software pipelining and also the cumulative savings by applying all three optimizations to algorithm 2 for a variety of problem sizes on the MP–1 and MP–2.

Optimizations \ Matrix Size	512	1024	1536	2048	2560	3072	3584	4096
Pipelining (MP–1)	15.1	17.9	18.9	19.7	20.1	20.3	20.6	20.7
Pipelining (MP–2)	34.6	40.8	42.4	43.2	43.8	44.0	44.3	44.5
All (MP–1)	—	28.3	—	26.1	—	25.2	—	24.8
All (MP–2)	44.2	48.6	49.1	49.4	49.6	49.7	49.8	49.8

Table 2: Percentage Reduction of Execution Time from Optimizations (Alg 2)

Another interesting observation, as shown in Graph 1, is that minimizing memory access time on the MasPar can be even more important than the choice of a superior parallel algorithm. The formulas suggest that algorithms 2 and 3 should outperform algorithm 1, the parallel prefix algorithm, and they do if the same level of optimization is used. However, the software pipelined version of algorithm 1 has a significantly lower execution time than the normal versions of the

other two algorithms for large problems.



The effects of the interleaved communication and register window technique (in addition to software pipelining) are not as big, but still significant. The percentage effects of each diminish as problem size increases. For interleaved communication, this is because it affects only the squared terms of the execution time. For the register window technique, this is because it reduces only the loads of elements of A by a constant factor. The loads of B still increase as a cubic function.

Interprocessor communication plays a secondary role and affects performance only slightly in any of the three algorithms. Tables 3 and 4 show the percentage of execution time spent on interprocessor communication for the three algorithms on the MP–1 and MP–2, respectively. The fully optimized version of each was used for algorithms 2 and 3; only software pipelining was applied to algorithm 1. Where memory accesses were necessary for communication, they were counted as communication overhead. The systolic communication pattern of algorithm 2 is clearly the least costly for all problem sizes. Incidentally, the broadcast algorithm uses more memory to store a copy of the A submatrix.

Algorithm \ Matrix Size	1024	2048	3072	4096
Algorithm 1	31.7	21.2	15.9	12.7
Algorithm 2	5.4	2.7	1.8	1.4
Algorithm 3	10.0	5.3	3.6	*

Table 3: Communication as Percentage of Execution Time (MP–1)

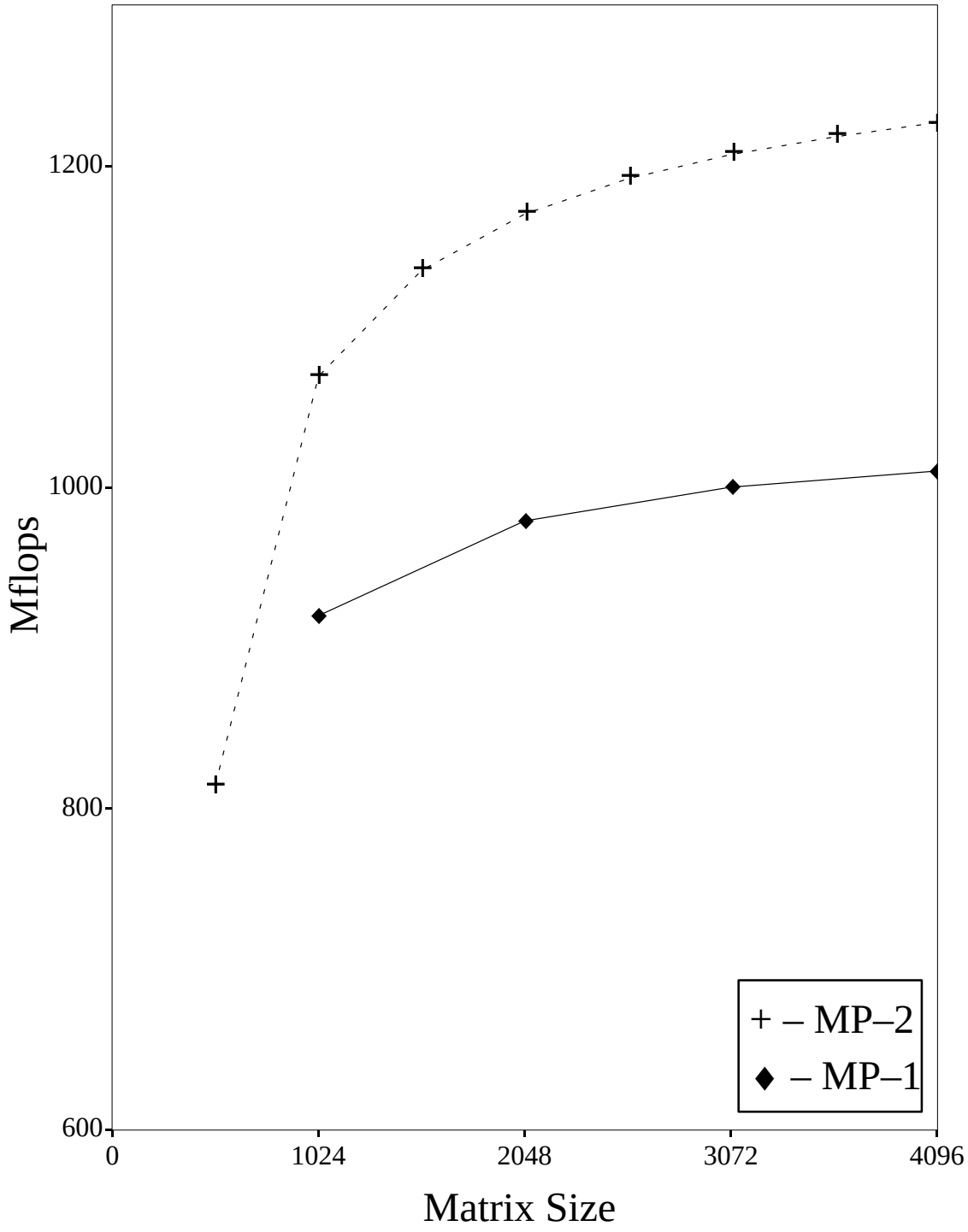
Algorithm \ Matrix Size	512	1024	1536	2048	2560	3072	3584	4096
Alg 1	39.7	33.0	26.6	22.6	19.0	17.1	15.5	13.4
Alg 2	13.1	10.0	6.9	5.2	4.3	3.6	3.1	2.7
Alg 3	23.6	16.0	11.8	9.4	7.9	7.0	6.1	*

Table 4: Communication as Percentage of Execution Time (MP–2)

* Not enough memory was available to run a problem of this size.

Finally, Graph 2 shows the performance in Mflops for the fastest algorithm (systolic) on the MP–1 and MP–2. Mflops were calculated from the execution time and the number of floating point operations required to calculate the matrix product.

Graph 2: Performance of Fully Optimized Algorithm 2



6 Conclusions

This paper describes memory access optimization techniques that are highly effective on MasPar machines. A software pipelining technique makes use of a buffering mechanism which allows pending memory accesses. The software pipelining technique overlaps memory access with computation or communication and thus reduces the overall execution time of the program. Another optimization called the register window technique can be applied to reduce the number loads in a loop. A buffering mechanism for pending memory accesses and a large number of registers are becoming commonplace in modern parallel computers based on load/store processors and we expect that techniques such as described here will have increasing importance for high performance computing. It is shown that for a highly computation intensive problem like matrix multiplication, the interprocessor communication can become a secondary issue compared to memory access optimization.

In a parallel computer the problem size can be expected to grow far beyond the number of processors. As this happens, the parallel complexity of an algorithm, where the number of processors is assumed to grow with the problem size, can lose its significance. The paper shows an interesting result where a software pipelined implementation of an order N log N algorithm outperforms a plain implementation of a superior algorithm of order N.

A detailed performance analysis of matrix multiplication is provided on MasPar MP–1 and MP–2 machines. Using the example of matrix multiplication, it is shown that the memory access optimization techniques provide substantial improvements beyond what is possible through compiler optimization.

7 Bibliography

[1] G.C. Fox and S.W. Otto, ‘‘Matrix Algorithms on a Hypercube I: Matrix Multiplication,’’ *Parallel Computing 4*, Elsevier Science Publishers B.V. (North Holland), 1987.

[2] Kothari, S.C., Gannett, E., and Oh, H., ‘‘Optimal Designs of Linear Flow Systolic Architectures,’’ *Proc. Int. Conf. on Parallel Processing*, 1989, pp. I–247—I–256.

[3] Knuth, Donald E., *The Art of Computer Programming*, Addison–Wesley Publishing Company, 1981, vol. 2.

[4] *MasPar Assembly Language (MPAS) Reference Manual*, MasPar Computer Corporation, Sunnyvale, CA, 1990.

[5] *MasPar Commands Reference Manual*, MasPar Computer Corporation, Sunnyvale, CA, 1990.

[6] *MasPar Parallel Application Language (MPL) Reference Manual*, MasPar Computer Corporation, Sunnyvale, CA, 1990.

[7] *MasPar Parallel Application Language (MPL) User Guide*, MasPar Computer Corporation, Sunnyvale, CA, 1990.

[8] *MasPar Standard Programming Manual*, MasPar Computer Corporation, Sunnyvale, CA, 1990.

8 Acknowledgments

We would like to thank the Scalable Computing Laboratory at Iowa State University for providing the MP–1 and MP–2 machines and the support staff. We also thank Michael Carter, Mark Fienup, and Youngtae Kim for useful discussions on the MasPar architecture.

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Tech Report: TR 93-02
Submission Date: January 13, 1993