Scalability of Parallel Algorithms for Matrix Multiplication*

Anshul Gupta and Vipin Kumar

Department of Computer Science,
University of Minnesota
Minneapolis, MN - 55455

agupta@cs.umn.edu and *kumar@cs.umn.edu*

Abstract

A number of parallel formulations of dense matrix multiplication algorithm have been developed. For arbitrarily large number of processors, any of these algorithms or their variants can provide near linear speedup for sufficiently large matrix sizes and none of the algorithms can be clearly claimed to be superior than the others. In this paper we analyze the performance and scalability of a number of parallel formulations of the matrix multiplication algorithm and predict the conditions under which each formulation is better than the others. We present a parallel formulation for hypercube and related architectures that performs better than any of the schemes described in the literature so far for a wide range of matrix sizes and number of processors. The superior performance and the analytical scalability expressions for this algorithm are verified through experiments on the Thinking Machines Corporation's CM-5^{TM}† parallel computer for up to 512 processors.

1 Introduction

Matrix multiplication is widely used in a variety of applications and is often one of the core components of many scientific computations. Since dense matrix multiplication algorithm is highly computation intensive, there has been a great deal of interest in developing parallel formulations of this algorithm and testing its performance on various parallel architectures [1, 2, 4, 5, 6, 8, 9, 12, 13, 18, 7].

Some of the early parallel formulations of matrix multiplication were developed by Cannon [4], Dekel, Nassimi and Sahni [8], and Fox *et. al.* [9]. Variants and improvements of these algorithms have been presented in [2, 13]. In particular, Berntsen [2] presents an algorithm which has a strictly smaller communication overhead than Cannon's algorithm, but has a smaller degree of concurrency [11].

For arbitrarily large number of processors, any of these algorithms or their variants can provide near linear speedup for sufficiently large matrix sizes, and none of the algorithms can be clearly claimed to be superior than the others. Scalability analysis is a an effective tool for predicting the performance of various algorithm-architecture combinations. Hence a great deal of research has been done to develop methods for scalability analysis [15]. In this paper, we use the isoefficiency metric [10, 15] to analyze the scalability of a number of parallel formulations of the matrix multiplication algorithm for hypercube and related architectures. We analyze the performance of various parallel formulations of the matrix multiplication algorithm for different matrix sizes and number of processors, and predict the conditions under which each formulation is better than the others. We present a parallel algorithm for the hypercube and related architectures that performs better than any of the previously described schemes for a wide range of matrix sizes and number of processors. The superior performance and the analytical scalability expressions for this algorithm are verified through experiments on the CM-5 parallel computer for up to 512 processors.

In this paper we assume that on a message passing parallel computer, the time required for the complete transfer of a message containing m words between two adjacent processors is given by $t_s + t_w m$, where t_s is the message startup time, and t_w (per-word communication time) is equal to $\frac{y}{B}$ where B is the bandwidth of the communication channel between the processors in bytes/second and y is the number of bytes per word. For the sake of simplicity, we assume that each basic arithmetic operation (*i.e.*, one floating point multiplication and one floating point addition in case of matrix multiplication) takes unit time. Therefore, t_s and t_w are relative data communication costs normalized w.r.t. the unit computation time.

2 The Isoefficiency Metric of Scalability

It is well known that given a parallel architecture and a problem instance of a fixed size, the speedup of a parallel algorithm does not continue to increase

*This work was supported by IST/SDIO through the Army Research Office grant # 28408-MA-SDI to the University of Minnesota and by the University of Minnesota Army High Performance Computing Research Center under contract # DAAL03-89-C-0038.

†CM-5 is a trademark of the Thinking Machines Corporation.

with increasing number of processors but tends to saturate or peak at a certain value. For a fixed problem size, the speedup saturates either because the overheads grow with increasing number of processors or because the number of processors eventually exceeds the degree of concurrency inherent in the algorithm. For a variety of parallel systems, given any number of processors p, speedup arbitrarily close to p can be obtained by simply executing the parallel algorithm on big enough problem instances [15]. The ease with which a parallel algorithm can achieve speedups proportional to p on a parallel architecture can serve as a measure of the scalability of the parallel system.

Let us define the size W of a problem as the time taken by an optimal (or the best known) sequential algorithm to solve the given problem on a single processor. Let $T_o(W,p)$ be the sum total of all the overheads incurred by all the p processors during the parallel execution of the algorithm. Now the efficiency of a parallel algorithm-architecture combination (henceforth referred to as a parallel system) is given by $E = \frac{W}{W+T_o(W,p)} = \frac{1}{1+\frac{T_o(W,p)}{W}}$. For a class of parallel systems called **scalable** parallel systems, the efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased. In order to maintain a fixed efficiency, W should be proportional to $T_o(W,p)$ or the following relation must be satisfied :

$$W = KT_o(W,p), \tag{1}$$

where $K = \frac{E}{1-E}$ is a constant depending on the efficiency to be maintained. The isoefficiency function [10, 15] of a parallel system is determined by abstracting W as a function of p through algebraic manipulations on Equation (1). If the problem size needs to grow as fast as $f_E(p)$ to maintain an efficiency E, then $f_E(p)$ is defined as the isoefficiency function of the parallel system for efficiency E. The smaller the isoefficiency function, the more scalable the parallel system is considered.

The isoefficiency function of a combination of a parallel algorithm and a parallel architecture relates the problem size to the number of processors necessary to maintain a fixed efficiency or to deliver speedups increasing proportionally with increasing number of processors. For a given parallel algorithm, for different parallel architectures, W may have to increase at different rates w.r.t. p in order to maintain a fixed efficiency. A small rate or isoefficiency function indicates a high scalability. Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel systems [15]. An important feature of isoefficiency analysis is that in a single expression, it succinctly captures the effects of characteristics of the parallel algorithm as well as the parallel architecture on which it is implemented.

3 Parallel Matrix Multiplication Algorithms

In this section we briefly describe some well known parallel matrix multiplication algorithms give their parallel execution times.

3.1 A Simple Algorithm

Consider a logical two dimensional mesh of p processors (with $\sqrt{p}$ rows and $\sqrt{p}$ columns) on which two $n \times n$ matrices A and B are to be multiplied to yield the product matrix C. Let $n \geq \sqrt{p}$. The matrices are divided into sub-blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ which are mapped naturally on the processor array. The algorithm can be implemented on a hypercube by embedding this processor mesh into it. In the first step of the algorithm, each processor acquires all those elements of both the matrices that are required to generate the $\frac{n^2}{p}$ elements of the product matrix which are to reside in that processor. This involves an all-to-all broadcast of $\frac{n^2}{p}$ elements of matrix A among the $\sqrt{p}$ processors of each row of processors and that of the same sized blocks of matrix B among $\sqrt{p}$ processors of each column which can be accomplished in $2t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}$ time. After each processor gets all the data it needs, it multiplies the $\sqrt{p}$ pairs of sub-blocks of the two matrices to compute its share of $\frac{n^2}{p}$ elements of the product matrix. Assuming that an addition and multiplication takes a unit time, the multiplication phase can be completed in $\frac{n^3}{p}$ units of time. Hence, the parallel execution time is:

$$T_p = \frac{n^3}{p} + 2t_s \log p + 2t_w \frac{n^2}{\sqrt{p}} \tag{2}$$

This algorithm is memory-inefficient. The memory requirement for each processor is $O(\frac{n^2}{\sqrt{p}})$ and thus the total memory requirement is $O(n^2\sqrt{p})$ words as against $O(n^2)$ for the sequential algorithm.

3.2 Cannon's Algorithm

A parallel algorithm that is memory efficient and is frequently used is due to Cannon [4, 1]. Again the two $n \times n$ matrices A and B are divided into square submatrices of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ among the p processors of a wrap-around mesh (which can be embedded in a hypercube). The sub-blocks of A and B residing with the processor (i,j) are denoted by A^{ij} and B^{ij} respectively, where $0 \leq i < \sqrt{p}$ and $0 \leq j < \sqrt{p}$. In the first phase of the execution of the algorithm, the data in the two input matrices is aligned in such a manner that the corresponding square submatrices at each processor can be multiplied together locally. This is done by sending the block A^{ij} to processor

$(i, (j+i) mod \sqrt{p})$, and the block B^{ij} to processor $((i+j) mod \sqrt{p}, j)$. The copied sub-blocks are then multiplied together. Now the A sub-blocks are rolled one step to the left and the B sub-blocks are rolled one step upward and the newly copied sub-blocks are multiplied and the results added to the partial results in the C sub-blocks. The multiplication of A and B is complete after $\sqrt{p}$ such steps. On a hypercube with cut-through routing, the time spent in the initial alignment step can be ignored w.r.t. to the $\sqrt{p}$ shift operations during the multiplication phase, as the former is a simple one-to-one communication along non-conflicting paths. Since each sub-block movement in the second phase takes $t_s + t_w \frac{n^2}{p}$ time, the total parallel execution time for all the movements of the sub-blocks of both the matrices is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_s\sqrt{p} + 2t_w\frac{n^2}{\sqrt{p}} \qquad (3)$$

3.3 Berntsen's Algorithm

Due to nearest neighbor communications on the $\sqrt{p} \times \sqrt{p}$ wrap-around array of processors, Cannon's algorithm's performance is the same on both mesh and hypercube architectures. In [2], Berntsen describes an algorithm which exploits greater connectivity provided by a hypercube. The algorithm uses $p = 2^{3q}$ processors with the restriction that $p \leq n^{3/2}$ for multiplying two $n \times n$ matrices A and B. Matrix A is split by columns and B by rows into 2^q parts. The hypercube is split into 2^q subcubes, each performing a submatrix multiplication between submatrices of A of size $\frac{n}{2^q} \times \frac{n}{2^{2q}}$ and submatrices of B of size $\frac{n}{2^{2q}} \times \frac{n}{2^q}$ using Cannon's algorithm. It is shown in [2] that the time spent in data communication by this algorithm on a hypercube is $2t_s p^{1/3} + \frac{1}{3}t_s \log p + 3t_w\frac{n^2}{p^{2/3}}$, and hence the total parallel execution time is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_s p^{1/3} + \frac{1}{3}t_s \log p + 3t_w\frac{n^2}{p^{2/3}} \qquad (4)$$

The terms associated with both t_s and t_w are smaller in this algorithm than the algorithms discussed in Sections 3.1 to 3.2. It should also be noted that this algorithm, like the one in Section 3.1 is not memory efficient as it requires storage of $2\frac{n^2}{p} + \frac{n^2}{p^{2/3}}$ matrix elements per processor.

3.4 The DNS Algorithm

3.4.1 One Element Per Processor Version

An algorithm that uses a hypercube with $p = n^3 = 2^{3q}$ processors to multiply two $n \times n$ matrices was proposed by Dekel, Nassimi and Sahni in [8, 17]. The p processors can be visualized as being arranged in an $2^q \times 2^q \times 2^q$ array. In this array, processor p_r occupies position (i, j, k) where $r = i2^{2q} + j2^q + k$ and $0 \leq i, j, k < 2^q$. Thus if the binary representation of r is $r_{3q-1}r_{3q-2}...r_0$, then the binary representations of i, j and k are $r_{3q-1}r_{3q-2}...r_{2q}$, $r_{2q-1}r_{2q-2}...r_q$ and $r_{q-1}r_{q-2}...r_0$ respectively. Each processor p_r has three data registers a_r, b_r and c_r, respectively. Initially, processor p_s in position (0,j,k) contains the element $a(j,k)$ and $b(j,k)$ in a_s and b_s respectively. The computation is accomplished in three stages. In the first stage, the elements of the matrices A and B are distributed over the p processors. As a result, a_r gets $a(j,i)$ and b_r gets $b(i,k)$. In the second stage, product elements $c(j,k)$ are computed and stored in each c_r. In the final stage, the sums $\Sigma_{i=0}^{n-1}c_{i,j,k}$ are computed and stored in $c_{0,j,k}$.

The above algorithm accomplishes the $O(n^3)$ task of matrix multiplication in $O(\log n)$ time using n^3 processors. Since the processor-time product of this parallel algorithm exceeds the sequential time complexity of the algorithm, it is not cost-optimal. In the following sub-sections we present two cost-optimal variations of this algorithm which use fewer than n^3 processors.

3.4.2 Variant With More Than One Element Per Processor

This variant proposed in [8, 17] can work with n^2r processors, where $1 < r < n$, thus using one processor for more than one element of each of the two $n \times n$ matrices. The algorithm is similar to the one above except that a logical processor array of r^3 (instead of n^3) superprocessors is used, each superprocessor comprising of $(n/r)^2$ hypercube processors. In the second step, multiplication of blocks of $(n/r) \times (n/r)$ elements instead of individual elements is performed. This multiplication of $(n/r) \times (n/r)$ blocks is performed according to the algorithm in Section 3.2 on $\frac{n}{r} \times \frac{n}{r}$ subarrays (each such subarray is actually a subcube) of processors using Cannon's algorithm for one element per processor. This step will require a communication time of $2(t_s + t_w)\frac{n}{r}$.

In the first stage of the algorithm, each data element is broadcast over r processors. In order to place the elements of matrix A in their respective positions, first the buffer $a_{(0,j,k)}$ is sent to $a_{(k,j,k)}$ in $\log r$ steps and then $a_{(k,j,k)}$ is broadcast to $a_{(k,j,l)}, 0 \leq l < r$, again in $\log r$ steps. By following a similar procedure, the elements of matrix B can be transmitted to their respective processors. In the second stage, groups of $(n/r)^2$ processors multiply blocks of $(n/r) \times (n/r)$ elements each processor performing n/r computations and $2n/r$ communications. In the final step, the elements of matrix C are restored to their designated processors in $\log r$ steps. The communication time can thus be shown to be equal to $(t_s + t_w)(5\log r + 2\frac{n}{r})$ resulting in the parallel run time given by the following equation:

$$T_p = \frac{n^3}{p} + (t_s + t_w)(5\log(\frac{p}{n^2}) + 2\frac{n^3}{p}) \qquad (5)$$

If $p = \frac{n^3}{\log n}$ processors are used, then the parallel execution time of the DNS algorithm is $O(\log n)$. The processor-time product is now $O(n^3)$, which is same as the sequential time complexity of the algorithm.

3.5 Our Variant of the DNS Algorithm

Here we present another scheme to adapt the single element per processor version of the DNS algorithm to be able to use fewer than n^3 processors on a hypercube. In the rest of the paper we shall refer to this algorithm as the GK variant of the DNS algorithm. As shown later in Section 5, this algorithm performs better than the DNS algorithm for a wide range of n and p. Also, unlike the DNS algorithm which works only for $n^2 \leq p \leq n^3$, this algorithm can use any number of processors from 1 to n^3. In this variant, we use $p = 2^{3q}$ processors where $q < \frac{1}{3}\log n$. The matrices are divided into sub-blocks of $\frac{n}{2^q} \times \frac{n}{2^q}$ elements and the sub-blocks are numbered just the way the single elements were numbered in the algorithm of Section 3.4.1. Now, all the single element operations of the algorithm of Section 3.4.1 are replaced by sub-block operations; *i.e.*, matrix sub-blocks are multiplied, communicated and added.

Let t_{mult} and t_{add} is the time to perform a single floating point multiplication and addition respectively, and $t_{mult} + t_{add} = 1$. In the first stage of this algorithm, $\frac{n^2}{p^{2/3}}$ data elements are broadcast over $p^{1/3}$ processors for each matrix. In order to place the elements of matrix A in their respective positions, first the buffer $a_{(0,j,k)}$ is sent to $a_{(k,j,k)}$ in $\log p^{1/3}$ steps and then $a_{(k,j,k)}$ is broadcast to $a_{(k,j,l)}, 0 \leq l < p^{1/3}$, again in $\log p^{1/3}$ steps. By following a similar procedure, the elements of matrix B can be sent to the processors where they are to be utilized in $2\log p^{1/3}$ steps. In the second stage of the algorithm, each processor performs $(\frac{n}{p^{1/3}})^3 = \frac{n^3}{p}$ multiplications. In the third step, the corresponding elements of $p^{1/3}$ groups of $\frac{n^2}{p^{2/3}}$ elements each are added in a tree fashion. The first stage takes $4t_s \log p^{1/3} + 4t_w \frac{n^2}{p^{2/3}} \log p^{1/3}$ time. The second stage contributes $t_{mult}\frac{n^3}{p}$ to the parallel execution time and the third stage involves $t_s \log p^{1/3} + t_w \frac{n^2}{p^{2/3}} \log p^{1/3}$ communication time and $t_{add}\frac{n^3}{p}$ computation time for calculating the sums. The total parallel execution time is therefore given by the following equation:

$$T_p = \frac{n^3}{p} + \frac{5}{3}t_s \log p + \frac{5}{3}t_w \frac{n^2}{p^{2/3}} \log p \qquad (6)$$

This execution time can be further reduced by using a more sophisticated scheme for one-to-all broadcast on a hypercube [14]. This is discussed in detail in [11].

4 Scalability Analysis

Recall from Section 2 that the isoefficiency function for a certain efficiency E can be obtained by equating W with $\frac{E}{1-E}T_o$ (Equation (1)) and then solving this equation to determine W as a function of p. In most of the parallel algorithms described in Section 3, the communication overhead has two different terms due to t_s and t_w. When there are multiple terms in T_o of different order, it is often not possible to obtain the isoefficiency function as a closed form function of p. As p and W increase in a parallel system, efficiency is guaranteed not to drop if none of the terms of T_o grows faster than W. Therefore, if T_o has multiple terms, we balance W against each individual term of T_o to compute the respective isoefficiency function. The component of T_o that requires the problem size to grow at the fastest rate w.r.t. p determines the overall isoefficiency function of the entire computation. Sometimes, the isoefficiency function for a parallel algorithm is due to the limit on the concurrency of the algorithm. For instance, if for a problem size W, an algorithm can not use more than $h(W)$ processors, then as the number of processors is increased, eventually W has to be increased as $h^{-1}(p)$ in order to keep all the processors busy and to avoid the efficiency from falling due to idle processors. If $h^{-1}(p)$ is greater than any of the isoefficiency terms due to communication overheads, then $h^{-1}(p)$ is the overall isoefficiency function and determines the scalability of the parallel algorithm. Thus it is possible for an algorithm to have little communication overhead, but still a bad scalability due to limited concurrency.

We now determine the isoefficiency functions for all the algorithms discussed in Section 3. The problem size W is taken as n^3 for all the algorithms.

4.1 Isoefficiency Analysis of Cannon's Algorithm

From Equation (3), it follows that the total overhead over all the processors for this algorithm is $2t_s p\sqrt{p} + 2t_w n^2\sqrt{p}$. In order to determine the isoefficiency term due to t_s, W has to be proportional to $2Kt_s p\sqrt{p}$ (see Equation (1)), where $K = \frac{1}{1-E}$ and E is the desired efficiency that has to be maintained. Hence the following isoefficiency relation results:

$$n^3 = W \propto 2Kt_s p\sqrt{p} \qquad (7)$$

Similarly, to determine the isoefficiency term due to t_w, n^3 has to proportional to $2Kt_w n^2\sqrt{p}$. Therefore,

$$n^3 \propto 2Kt_w n^2\sqrt{p}$$
$$=> \quad n \propto 2Kt_w\sqrt{p}$$
$$=> \quad n^3 = W \propto 8K^3t_w^3p^{1.5} \qquad (8)$$

According to both Equations (7) and (8), the asymptotic isoefficiency function of Cannon's algorithm is $O(p^{1.5})$. Also, since the maximum number of processors that can be used by this algorithm is n^2, the isoefficiency due to concurrency[1] is also $O(p^{1.5})$. Thus Cannon's algorithm is as scalable on a hypercube as any matrix multiplication algorithm using $O(n^2)$ processors can be on any architecture.

The above analysis also applies to the simple algorithm because both the degree of concurrency and the communication overheads (due to the t_w term which determines the overall isoefficiency function) are the same for these two algorithms.

4.2 Isoefficiency Analysis of Berntsen's Algorithm

The overall overhead function for this algorithm can be determined from the expression of the parallel execution time in Equation (4) to be $2t_sp^{4/3} + \frac{1}{3}t_sp\log p + 3t_wn^2p^{1/3}$. By an analysis similar to that in Section 4.1, it can be shown that the isoefficiency terms due to t_s and t_w for this algorithm are given by the following equations:

$$n^3 = W \propto 2Kt_sp^{4/3} \qquad (9)$$

$$n^3 = W \propto 27K^3t_w^3p \qquad (10)$$

Recall from Section 3.3 that for this algorithm, $p \le n^{3/2}$. This means that $n^3 = W \propto p^2$ as the number of processors is increased. Thus the isoefficiency function due to concurrency is $O(p^2)$, which is worse than any of the isoefficiency terms due to the communication overhead. Thus this algorithm has a poor scalability despite little communication cost due to its limited concurrency.

4.3 Isoefficiency Analysis of the DNS Algorithm

It can be shown that the overhead function T_o for this algorithm is $(t_s + t_w)(\frac{5}{3}p\log p + 2n^3)$. Since W is $O(n^3)$, the terms $2(t_s + t_w)n^3$ will always be balanced w.r.t. W. This term is independent of p and does not contribute to the isoefficiency function. It does however impose an upper limit on the efficiency that this algorithm can achieve. Since, for this algorithm, $E = \frac{1}{1+\frac{5/3p\log p}{n^3}+2(t_s+t_w)}$, an efficiency higher than $\frac{1}{1+2(t_s+t_w)}$ can not be attained, no matter how big the problem size is. Since t_s is usually a large constant for most practical MIMD computers, the achievable efficiency of this algorithm is quite limited on such machines. The other term in T_o yields the following isoefficiency function for the algorithm:

$$n^3 = W \propto \frac{5}{3}Kt_sp\log p \qquad (11)$$

The above equation shows that the asymptotic isoefficiency function of the DNS algorithm on a hypercube is $O(p\log p)$. It can easily be shown that an $O(p\log p)$ scalability is the best any parallel formulation of the conventional $O(n^3)$ algorithm can achieve on any parallel architecture [3] and the DNS algorithm achieves this lower bound on a hypercube.

4.4 Isoefficiency Analysis of the GK Algorithm

The total overhead T_o for this algorithm is equal to $\frac{5}{3}t_sp\log p + \frac{5}{3}t_wn^2p^{1/3}\log p$ and the following equations give the isoefficiency terms due to t_s and t_w respectively for this algorithm:

$$n^3 = W \propto \frac{5}{3}Kt_sp\log p \qquad (12)$$

$$n^3 = W \propto \frac{125}{27}K^3t_w^3p(\log p)^3 \qquad (13)$$

5 Relative Performance of the Four Algorithms on a Hypercube

Subsections 4.1 through 4.4 give the overall isoefficiency functions of the four algorithms on a hypercube architecture. The asymptotic scalabilities and the range of applicability of these algorithms is summarized in Table 1.

Note that Table 1 gives only the asymptotic scalabilities of the four algorithms. In practice, none of the algorithms is strictly better than the others for all possible problem sizes and number of processors. Further analysis is required to determine the best algorithm for a given problem size and a certain parallel machine depending on the number of processors being used and the hardware parameters of the machine. A detailed comparison of these algorithms based on their respective total overhead functions is presented in the next section.

We compare a pair of algorithms by comparing their total overhead functions (T_o) as given in Table 1. For instance, while comparing the GK algorithm with Cannon's algorithm, it is clear that the t_s term for the GK algorithm will always be less than that for Cannon's algorithm. Even if $t_s = 0$, the t_w term of the GK algorithm becomes smaller than that of Cannon's algorithm for $p >$ 130 million. Thus, $p =$ 130 million is the cut-off point beyond which the GK algorithm will perform better than Cannon's algorithm irrespective of the values of n. For $p <$ 130 million, the performance of the GK algorithm

[1] $n^2 \propto p \;=>\; n^3 = W \propto p^{1.5}$.

Algorithm	Total Overhead Function, T_o	Asymptotic Isoeff. Function	Range of Applicability
Berntsen's	$2t_sp^{4/3} + \frac{1}{3}t_sp\log p + 3t_wn^2p^{1/3}$	$O(p^2)$	$1 \le p \le n^{3/2}$
Cannon's	$2t_sp^{3/2} + 2t_wn^2\sqrt{p}$	$O(p^{1.5})$	$1 \le p \le n^2$
GK	$\frac{5}{3}t_sp\log p + \frac{5}{3}t_wn^2p^{1/3}\log p$	$O(p(\log p)^3)$	$1 \le p \le n^3$
DNS	$(t_s + t_w)(\frac{5}{3}p\log p + 2n^3)$	$O(p\log p)$	$n^2 \le p \le n^3$

Table 1: *Communication overhead, scalability and range of application of the four algorithms on a hypercube.*

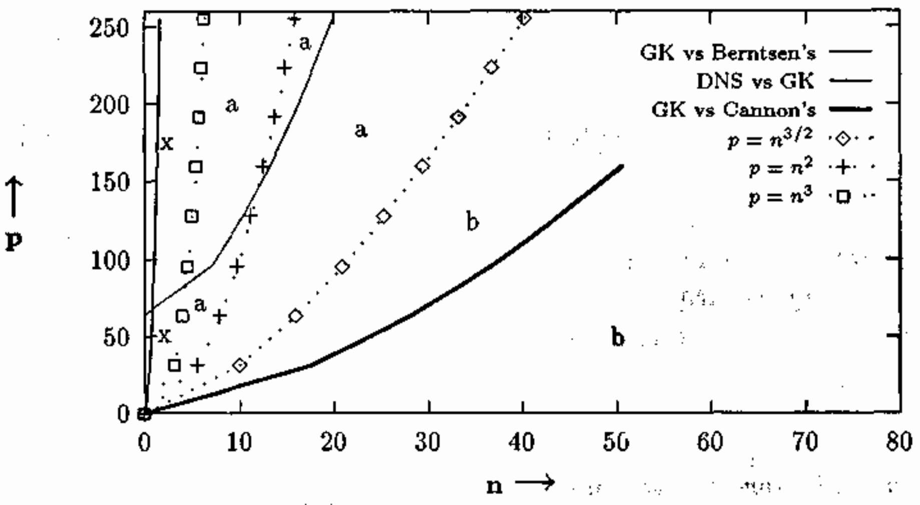

Figure 1: *A comparison of the four algorithms for* $t_w = 3$ *and* $t_s = 150$.

will be better than that of Cannon's algorithm for values of n less than a certain threshold value which is a function of p and the ration of t_s and t_w. A hundred and thirty million processors is clearly too large, but we show that for reasonable values of t_s, the GK algorithm performs better than Cannon's algorithm for very practical values of p and n.

In order to determine ranges of p and n where the GK algorithm performs better than Cannon's algorithm, we equate their respective overhead functions and compute n as a function of p. We call this $n_{Equal-T_o}(p)$ because this value of n is the threshold at which the overheads of the two algorithms will be identical for a given p. If $n > n_{Equal-T_o}(p)$, Cannon's algorithm will perform better and if $n < n_{Equal-T_o}(p)$, the GK algorithm will perform better. If we equate T_o for the two algorithms, then $T_o^{(Cannon)} = 2t_sp^{3/2} + 2t_wn^2\sqrt{p} = T_o^{(GK)} = \frac{5}{3}t_sp\log p + \frac{5}{3}t_wn^2p^{1/3}\log p$. Therefore,

$$n_{Equal-T_o}(p) = \sqrt{\frac{(5/3p\log p - 2p^{3/2})t_s}{(2\sqrt{p} - 5/3p^{1/3}\log p)t_w}} \quad (14)$$

Similarly, equal overhead conditions can be determined for other pairs of algorithms too and the values of t_w and t_s can be plugged in depending upon the machine in question to determine the best algorithm for a give problem size and number of processors. We have performed this analysis for three practical sets of values of t_w and t_s. In the rest of the section we demonstrate the practical importance of this analysis by showing how any of the four algorithms can be useful depending on the problem size and the parallel machine available.

Figures 1, 2 and 3 show the regions of applicability and superiority of different algorithms.

The plain lines represent equal overhead conditions for pairs of algorithms. For a curve marked "X vs Y" in a figure, algorithm X has a smaller value of communication overhead to the left of the curve, algorithm Y has smaller communication overhead to the right side of the curve, while the two algorithms have the same value of T_o along the curve. The lines with symbols $\diamond$, $+$ and $\square$ plot the functions $p = n^{3/2}$, $p = n^2$ and $p = n^3$, respectively. These lines demarcate the regions of applicabilities of the four algorithms (see Table 1) and are important because an algorithm might not be applicable in the region where its overhead function T_o is mathematically superior than others. In all the figures in this section, the region marked with an **x** is the one

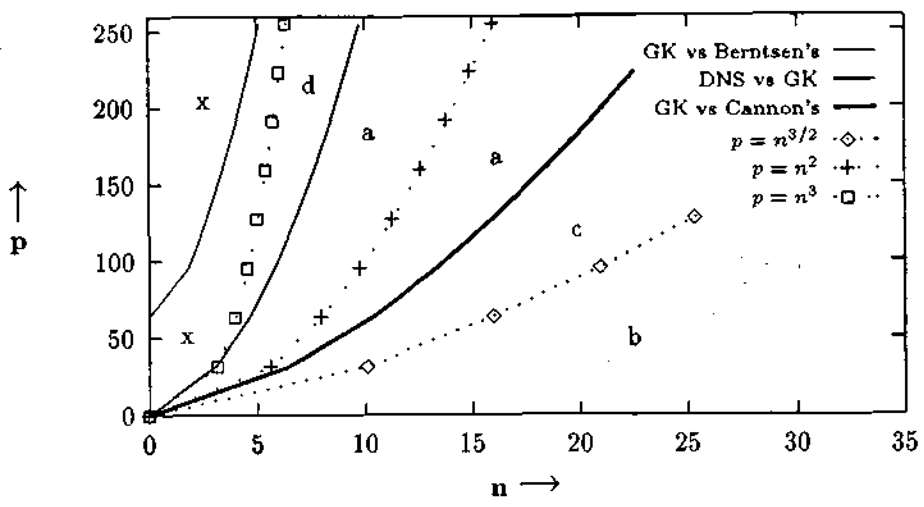

Figure 2: *A comparison of the four algorithms for $t_w = 3$ and $t_s = 10$.*

where $p > n^3$ and none of the algorithms is applicable, the region marked with an **a** is the one where the GK algorithm is the best choice, the symbol **b** represents the region where Berntsen's algorithm is superior to the others, the region marked with a **c** is the one where Cannon's algorithm should be used and the region marked with a **d** is the one where the DNS algorithm is the best.

Figure 1 compares the four algorithms for $t_w = 3$ and $t_s = 150$. These parameters are very close to that of a currently available parallel computer like the nCUBE2^{TM*}. In this figure, since the $n_{Equal-T_o}$ curve for the DNS algorithm and the GK algorithm lies in the **x** region, and the DNS algorithm is better than the GK algorithm only for values of n smaller than $n_{Equal-T_o}(p)$. Hence the DNS algorithm will always[2] perform worse than the GK algorithm for this set of values of t_s and t_w and the latter is the best overall choice for $p > n^2$ as Berntsen's algorithm and Cannon's algorithm are not applicable in this range of p. Since the $n_{Equal-T_o}$ curve for GK and Cannon's algorithm lies below the $p = n^{3/2}$ curve, the GK algorithm is the best choice even for $n^{3/2} \leq p \leq n^2$. For $p < n^{3/2}$, Berntsen's algorithm is always better than Cannon's algorithm, and for this set of t_s and t_w, also than the GK algorithm. Hence it is the best choice in that region in Figure 1.

In Figure 2, we compare the four algorithms for a hypercube with $t_w = 3$ and $t_s = 10$. Such a machine could easily be developed in the near future by using faster CPU's (t_w and t_s represent relative communication costs w.r.t. the unit computation time) and reducing the message startup time. By observing the $n_{Equal-T_o}$ curves and the regions of applicability of these algorithms, the regions of superiority of each of the algorithms can be determined just as in case of Figure 1. It is noteworthy that in Figure 2 each of the four algorithms performs better than the rest in some region and all the four regions **a**, **b**, **c** and **d** contain practical values of p and n.

In Figure 3, we present a comparison of the four algorithms for $t_w = 3$ and $t_s = 0.5$. These parameters are close to what one can expect to observe on a typical SIMD machine like the CM-2. For the range of processors shown in the figure, the GK algorithm is inferior to the others[3]. Hence it is best to use the DNS algorithm for $n^2 \leq p \leq n^3$, Cannon's algorithm for $n^{3/2} \leq p \leq n^2$ and Berntsen's algorithm for $p < n^{3/2}$.

6 Experimental Results

We verified a part of the analysis of this paper through experiments of the CM-5 parallel computer. On this machine, the fat-tree [16] like communication network on the CM-5 provides simultaneous paths for communication between all pairs of processors. Hence the CM-5 can be viewed as a fully connected architecture which can simulate a hypercube connected network. We implemented Cannon's algorithm described in Section 3.2 and the algorithm described in Section 3.5.

Since the CM-5 can be considered as a fully connected network of processors, the expression for the parallel execution time for the algorithm of Section 3.5 will have to be modified slightly. The first part

*nCUBE2 is a trademark of the Ncube corporation.

[2] Actually, the $n_{Equal-T_o}$ curve for DNS vs GK algorithms will cross the $p = n^3$ curve for $p = 2.6 \times 10^{18}$, but clearly this region has no practical importance.

[3] The GK algorithm does begin to perform better than the other algorithms for $p > 1.3 \times 10^8$, but again we consider this range of p to be impractical.

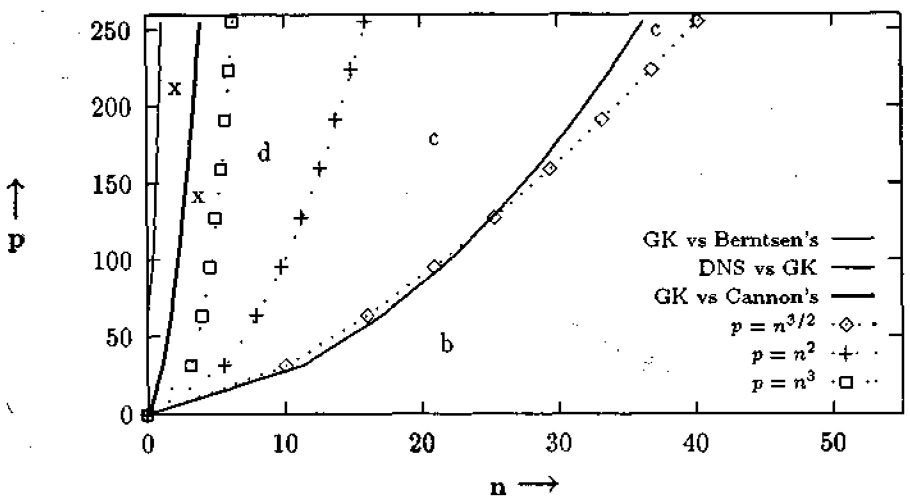

Figure 3: *A comparison of the four algorithms for $t_w = 3$ and $t_s = 0.5$.*

of the procedure to place the elements of matrix A in their respective positions, requires sending the buffer $a_{(0,j,k)}$ to $a_{(k,j,k)}$. This can be done in one step on the CM-5 instead of $\log(p^{1/3})$ steps on a conventional hypercube. The same is true for matrix B as well. It can be shown that the following modified expression gives the parallel execution time for this algorithm on the CM-5:

$$T_p = \frac{n^3}{p} + t_s(\log p + 2) + t_w \frac{n^2}{p^{2/3}}(\log p + 2) \quad (15)$$

Computing the condition for equal T_o for this and Cannon's algorithm by deriving the respective values of T_o from Equations (15) and (3), it can be shown that for 512 processors, Cannon's algorithm should perform better that our algorithm for $n > 295$. Since the number of processors has to be a perfect square for Cannon's algorithm on square matrices, in Figure 4, we draw the efficiency vs n curve for $p = 484$ for Cannon's algorithm and for $p = 512$ for the GK algorithm[4]. The cross-over point closely matches the predicted value. These experiments suggest that the algorithm of Section 3.5 can outperform the classical algorithms like Cannon's for a wide range of problem sizes and number of processors. Moreover, as the number of processors is increased, the cross-over point of the efficiency curves of the GK algorithm and Cannon's algorithm corresponds to a very high efficiency. As seen in Figure 4, the cross-over happens at $E \approx 0.93$ and Cannon's algorithm can not outperform the GK algorithm by a wide margin at such high efficiencies. On the other hand, the GK algorithm achieves an efficiency of 0.5 for a matrix size of 112×112, whereas Cannon's algorithm operates at an efficiency of only 0.28 on 484 processors on 110×110 matrices. In other words, in the region where the GK algorithm is better than Cannon's algorithm, the difference in the efficiencies is quite sig..ificant.

[4]This is not an unfair comparison because the efficiency can only be better for smaller number of processors.

7 Concluding Remarks

In this paper we have presented the scalability analysis of a number of matrix multiplication algorithms described in the literature [4, 8, 2, 13]. Besides analyzing these classical algorithms, we show that the GK algorithm that we present in this paper outperforms all the well known algorithms for a significant range of number of processors and matrix sizes. The scalability analysis of all these algorithms provides several important insights regarding their relative superiority under different conditions. None of the algorithms discussed in this paper is clearly superior to the others because there are a number of factors that determine the algorithm that performs the best. In this paper we predict the precise conditions under which each formulation is better than the others. It may be unreasonable to expect a programmer to code different algorithms for different machines, different number of processors and different matrix sizes. But all the algorithms can stored in a library and the best algorithm can be pulled out by a smart preprocessor/compiler depending on the various parameters.

We show that an algorithm with a seemingly small expression for the communication overhead is not necessarily the best one because it may not scale well as the number of processors is increased. For instance, the best algorithm in terms of communication overheads (Berntsen's algorithm described in Section 3.3) turns out to be the least scalable one with an isoefficiency function of $O(p^2)$ due its lim-

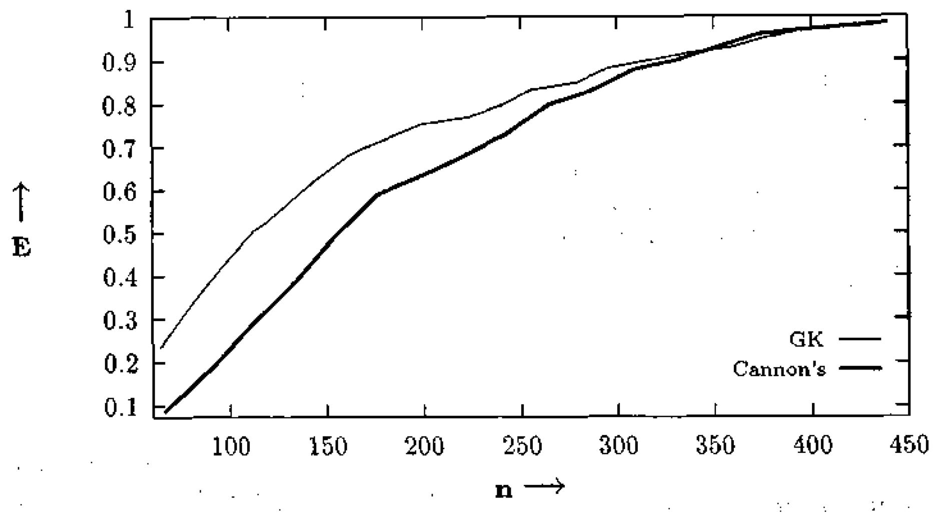

Figure 4: *Efficiency vs matrix size for Cannon's algorithm ($p = 484$) and the GK algorithm ($p = 512$).*

ited degree of concurrency.

References

[1] S. G. Akl. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall, 1989.

[2] Jarle Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12:335 - 342, 1989.

[3] Dimitri P. Bertsekas and John N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice Hall, 1989.

[4] L. E. Cannon. A cellular computer to implement the Kalman Filter Algorithm. Technical report, Ph.D. Thesis, Montana State University, 1969.

[5] V. Cherkassky and R. Smith. Efficient mapping and implementations of matrix algorithms on a hypercube. *The Journal of Supercomputing*, Vol. 2:7 - 27, 1988.

[6] N. P. Chrisopchoides, M. Aboelaze, E. N. Houstis, and C. E. Houstis. The parallelization of some level 2 and 3 BLAS operations on distributed memory machines. In *Proceedings of the 1st International Conference of the Austrian Center of Parallel Computation*. Springer-Verlag Series Lacture Notes in Computer Science, 1991.

[7] Eric F. Van de Velde. Multicomputer matrix computations: Theory and practice. In *Proceedings of the 1989 Conferece on Hypercubes, Concurrent Computers, and Applications*, pages 1303 - 1308, 1989.

[8] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel matrix and graph algorithms. *SIAM Journal of Computing*, 10:657 - 673, 1981.

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

[10] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. Technical report, Computer Science Department, University of Minnesota, April 1993.

[11] Anshul Gupta and Vipin Kumar. The scalability of Matrix Multiplication Algorithms on parallel computers. Technical Report TR 91-54, Computer Science Department, University of Minnesota, Minneapolis, MN 55455, 1991.

[12] Paul G. Hipes. Matrix multiplication on the JPL/Caltech Mark IIIfp hypercube. Technical Report C3P 746, Concurrent Computation Program, California Institute of Technology, Pasadena, CA - 91125, 1989.

[13] Ching-Tien Ho, S. Lennart Johnsson, and Alan Edelman. Matrix multiplication on hypercubes using full bandwidth and constant storage. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 447 - 451, 1991.

[14] S. Lennart Johnsson and Ching-Tien Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249 - 1268, September 1989.

[15] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. Technical report, TR-91-18, Computer Science Department, University of Minnesota, June 1991. A short version of the paper appears in the Proceedings of the 1991 International Conference on Supercomputing, Germany, and as an invited paper in the Proc. of 29th Annual Allerton Conference on Communication, Control and Computing, Urbana,IL, October 1991.

[16] Charles E. Leiserson. Fat-trees : Universal networks for hardware efficient supercomputing. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 393 - 402, August 20 - 23, 1985.

[17] S. Ranka and S. Sahni. *Hypercube Algorithms for Image Processing and Pattern Recognition*. Springer-Verlag, New York, 1990.

[18] Walter F. Tichy. Parallel matrix multiplication on the connection machine. Technical Report RIACS TR 88.41, Research Institute for Advanced Computer Science, NASA Ames Research Center, Ames, Iowa, 1988.