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*

TR 91-54, November 1991 (Revised April 1994)

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. We show that special hardware permitting simultaneous communication on all the ports of the processors does not improve the overall scalability of the matrix multiplication algorithms on a hypercube. We also discuss the dependence of scalability on technology dependent factors such as communication and computation speeds and show that under certain conditions, it may be better to have a parallel computer with k-fold as many processors rather than one with the same number of processors, each k-fold as fast.

*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.

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 the 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, 3, 5, 6, 7, 9, 11, 17, 18, 37, 8].

Some of the early parallel formulations of matrix multiplication were developed by Cannon [5], Dekel, Nassimi and Sahni [9], and Fox *et al.* [11]. Variants and improvements of these algorithms have been presented in [3, 18]. In particular, Berntsen [3] presents an algorithm which has a strictly smaller communication overhead than Cannon's algorithm, but has a smaller degree of concurrency. Ho *et al.* [18] present another variant of Cannon's algorithm for a hypercube which permits communication on all channels simultaneously. This algorithm too, while reducing communication, also reduces the degree of concurrency.

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 [23]. The *isoefficiency function* [24, 26] is one such metric of scalability which is a measure of an algorithm's capability to effectively utilize an increasing number of processors on a parallel architecture. Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel systems [19, 15, 24, 25, 28, 35, 36, 39, 38, 14, 26, 13, 22]. An important feature of the isoefficiency function is that it succinctly captures the impact of communication overheads, concurrency, serial bottlenecks, load imbalance, etc. in a single expression.

In this paper, we use the isoefficiency metric [24] 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 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 CM-5 parallel computer for up to 512 processors. We show that special hardware permitting simultaneous communication on all the ports of the processors does not improve the overall scalability of the matrix multiplication algorithms on a hypercube. We also discuss the dependence of scalability of parallel matrix multiplication algorithms on technology dependent factors such as communication and computation speeds and show that under certain conditions, it may be better to have a parallel computer with

k-fold as many processors rather than one with the same number of processors, each k-fold as fast.

The organization of the paper is as follows. In Section 2, we define the terms that are frequently used in the rest of the paper. Section 3 gives an overview of the isoefficiency metric of scalability. In Section 4, we give an overview of several parallel algorithms for matrix multiplication and give expressions for their parallel execution times. In Section 5, we analyze the scalability of all the parallel formulations discussed in Section 4. In Section 6, we provide a detailed comparison of all the algorithms described in this paper and derive the conditions under which each one is better than the rest. In Section 7, we analyze the impact of permitting simultaneous communication on all ports of the processors of a hypercube on the performance and scalability of the various matrix multiplication algorithms. In Section 8, the impact of technology dependent factors on the scalability of the algorithm is discussed. Section 9 contains some experimental results comparing the performance of our parallel formulation with that of Cannon's algorithm on the CM-5. Section 10 contains concluding remarks.

2 Terminology

In this section, we introduce the terminology that shall be followed in the rest of the paper.

Parallel System : We define a parallel system as the combination of a parallel algorithm and the parallel architecture on which it is implemented.

Number of Processors, p: The number of homogeneous processing units in the parallel computer that cooperate to solve a problem.

Problem Size, W: The time taken by the serial algorithm to solve the given problem on a single processor. This is also equal to the sum total of all the useful work done by all the processors while solving the same problem in parallel using p processors. For instance, for the multiplication of two $n \times n$ matrices[1], we consider $W = O(n^3)$.

Parallel Execution Time, T_p: The time taken by p processors to solve a problem. For a given parallel system, T_p is a function of the problem size and the number of processors.

Parallel Speedup, S: The ratio of W to T_p.

Total Parallel Overhead, T_o: The sum total of all the overheads incurred by all the processors during the parallel execution of the algorithm. It includes communication costs, non-essential work and idle time due to synchronization and serial components of the

[1] In this paper we consider the conventional $O(n^3)$ serial matrix multiplication algorithm only. Serial matrix multiplication algorithms with better complexity have higher constants and are not used much in practice.

algorithm. For a given parallel system, T_o is usually a function of the problem size and the number of processors and is often written as $T_o(W, p)$. Thus $T_o(W, p) = pT_p - W$.

Efficiency, E**:** The ratio of S to p. Hence, $E = W/pT_p = 1/(1 + \frac{T_o}{W})$.

Data Communication Costs, t_s **and** t_w**:** 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, in this paper 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 with respect to the unit computation time.

3 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 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 (*e.g.*, [21, 12, 29, 34, 16, 10, 31, 33, 32, 40]). 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.

The *isoefficiency function* [24, 26] is one such metric of scalability which is a measure of an algorithm's capability to effectively utilize an increasing number of processors on a parallel architecture. 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. The efficiency of a parallel system is given by $E = \frac{W}{W + T_o(W,p)}$. If a parallel system is used to solve a problem instance of a fixed size W, then the efficiency decreases as p increases. The reason is that the total overhead $T_o(W, p)$ increases with p. For many parallel systems, for a fixed p, if the problem size W is increased, then the efficiency increases because for a given p, $T_o(W, p)$ grows slower than $O(W)$. For these parallel systems, the efficiency can be maintained at a desired value (between 0 and 1) for increasing p, provided W is also increased. We call such systems **scalable** parallel systems. Note that for a given parallel algorithm, for different parallel architectures, W may have to increase at different rates with respect to p in order to maintain a fixed efficiency. For example, in some cases, W might need to grow exponentially with respect to p to keep the efficiency from dropping as p is increased. Such a parallel system is poorly scalable because it would be difficult to obtain good speedups for a large number of processors, unless the size of the problem being solved is enormously large. On the other hand, if W needs to grow only linearly with respect to p, then the parallel system is highly scalable and can easily deliver speedups increasing linearly with respect to the number of processors for reasonably increasing problem sizes. The isoefficiency functions of several common parallel systems are polynomial functions of p; *i.e.*, they are $O(p^x)$, where $x \geq 1$. A small power of p in the isoefficiency function indicates a high scalability.

If a parallel system incurs a total overhead of $T_o(W,p)$, where p is the number of processors in the parallel ensemble and W is the problem size, then the efficiency of the system is given by $E = \frac{1}{1+\frac{T_o(W,p)}{W}}$. In order to maintain a constant efficiency, W should be proportional to $T_o(W,p)$ or the following relation must be satisfied:

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

Here $K = \frac{E}{1-E}$ is a constant depending on the efficiency to be maintained. Equation (1) is the central relation that is used to determine the isoefficiency function. This is accomplished 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 to be the isoefficiency function of the parallel algorithm-architecture combination for efficiency E.

Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel systems [24, 15, 25, 27, 35, 36, 39, 38, 14, 26, 13, 22]. 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. By performing isoefficiency analysis, one can test the performance of a parallel program on a few processors, and then predict its performance on a larger number of processors. But the utility of the isoefficiency analysis is not limited to predicting the impact on performance of an increasing number of processors. It can also be used to study the behavior of a parallel system with respect to changes in other hardware related parameters such as the speed of the processors and the data communication channels.

4 Parallel Matrix Multiplication Algorithms

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

4.1 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 $\frac{n}{\times}\sqrt{p} \times \frac{n}{\times}\sqrt{p}$ 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 (Section 2), the multiplication phase can be completed in $\frac{n^3}{p}$ units of time. Thus the total parallel execution time of the algorithm is given by the following equation:

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

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.

4.2 Cannon's Algorithm

A parallel algorithm that is memory efficient and is frequently used is due to Cannon [5]. 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 if the algorithm was to be implemented on it). 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 $\sqrt{p}$ steps of rolling the sub-blocks of A and B leftwards and upwards, respectively, and multiplying the in coming sub-blocks in each processor. In a hypercube with cut-through routing, the time spent in the initial alignment step can be ignored with respect 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:

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

4.3 Fox's Algorithm

This algorithm is due to Fox *et al* and is described in detail in [11] and [10]. The input matrices are initially distributed among the processors in the same manner as in the algorithm in Section 4.1. The algorithm works in $\sqrt{p}$ iterations, where p is the number of processors being used. The data communication in the algorithm involves successive broadcast of the the sub-blocks of A in a horizontal direction so that all processors in the ith row receive the sub-block $A^{i(i+j)}$ in the jth iteration (iterations are numbered from 0 to j - 1). After each broadcast the sub-blocks of A are multiplied by the sub-blocks of B currently residing in each processor and are accumulated in the sub-blocks of S. The last step of each iteration is the shifting of the sub-blocks of B in all the processors to their respective North neighbors in the wrap-around mesh, the sub-blocks of the topmost row being rolled into the bottommost row. Thus, for the mesh architecture, the algorithm takes $(t_s + t_w\frac{n^2}{p})\sqrt{p}$ time in communication in each of the $\sqrt{p}$ iterations, resulting in a total parallel execution time of $\frac{n^3}{p} + t_w n^2 + t_s p$. By sending the sub-blocks in small packets in a pipelined fashion, Fox *et al.* show the run time of this algorithm to be:

$$T_p = \frac{n^3}{p} + 2t_w\frac{n^2}{\sqrt{p}} + t_s p \tag{4}$$

Clearly the parallel execution time of this algorithm is worse than that of the simple algorithm or Cannon's algorithm. On a hypercube, it is possible to employ a more sophisticated scheme for one-to-all broadcast [20] of sub-blocks of matrix A among the rows. Using this scheme, the parallel execution time can be improved to $\frac{n^3}{p} + 2t_w\frac{n^2}{\sqrt{p}} + t_s\sqrt{p}\log p + 2n\sqrt{t_s t_w \log p}$, which is still worse than Cannon's algorithm. However, if the procedure is performed in an asynchronous manner (*i.e.*, in every iteration, a processor starts performing its computation as soon as it has all the required data, and does not wait for the entire broadcast to finish) the computation and communication of sub-blocks can be interleaved. It can be shown that if each step of Fox's algorithm is not synchronized and the processors work independently, then its parallel execution time can be reduced to almost a factor of two of that Cannon's algorithm.

4.4 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 [3], 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 [3] 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:

$$T_p = \frac{n^3}{p} + 2t_s p^{1/3} + \frac{1}{3}t_s \log p + 3t_w \frac{n^2}{p^{2/3}} \tag{5}$$

The terms associated with both t_s and t_w are smaller in this algorithm than the algorithms discussed in Sections 4.1 to 4.2. It should also be noted that this algorithm, like the one in Section 4.1 is not memory efficient as it requires storage of $2\frac{n^2}{p} + \frac{n^2}{p^{2/3}}$ matrix elements per processor.

4.5 The DNS Algorithm

4.5.1 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 [9, 35]. 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 processor-efficient. This algorithm can be made processor-efficient by using fewer that n^3 processors, *i.e.*; by putting more than one element of the matrices on each processor. There are more than one ways to adapt this algorithm to use fewer than n^3 processors. The method proposed by Dekel, Nassimi and Sahni in [9, 35] is as follows.

4.5.2 Variant With More Than One Element Per Processor

This variant of the DNS algorithm can work with $n^2 r$ 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 4.3 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:

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

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.

4.6 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 6, 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

4.5.1. Now, all the single element operations of the algorithm of Section 4.5.1 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. Also, according to the assumption of Section 2, $t_{mult} + t_{add} = 1$. 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:

$$T_p = \frac{n^3}{p} + \frac{5}{3} t_s \log p + \frac{5}{3} t_w \frac{n^2}{p^{2/3}} \log p \tag{7}$$

This execution time can be further reduced by using a more sophisticated scheme for one-to-all broadcast on a hypercube [20]. This is discussed in detail in Section 5.4.

5 Scalability Analysis

If W is the size of the problem to be solved and $T_o(W, P)$ is the total overhead, then the efficiency E is given by $\frac{W}{W+T_o(W,p)}$. Clearly, for a given W, if p increases, then E will decrease because $T_o(W, p)$ increases with p. On the other hand, if W increases, then E increases because the rate of increase of T_o is slower than that of W for a scalable algorithm. 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 4, 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 with respect to 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.

In the following subsections, we determine the isoefficiency functions for all the algorithms discussed in Section 4. The problem size W is taken as n^3 for all the algorithms.

5.1 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:

$$n^3 = W \propto 2Kt_s p\sqrt{p} \tag{8}$$

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

$$n^3 \propto 2Kt_w n^2\sqrt{p}$$

$$=> \quad n \propto 2Kt_w\sqrt{p}$$

$$=> \quad n^3 = W \propto 8K^3 t_w^3 p^{1.5} \tag{9}$$

According to both Equations (8) and (9), 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[2] 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.

All of the above analysis also applies to the simple algorithm and the asynchronous version of Fox's algorithm also because the degree of concurrency of Cannon's algorithm as well as its communication overheads (asymptotically, or within a constant factors) are identical to these algorithms.

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

5.2 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 (5) to be $2t_s p^{4/3} + \frac{1}{3}t_s p \log p + 3t_w n^2 p^{1/3}$. By an analysis similar to that in Section 5.1, it can be shown that the isoefficiency terms due to t_s and t_w for this algorithm are given by the following equations:

$$n^3 = W \propto 2Kt_s p^{4/3} \tag{10}$$

$$n^3 = W \propto 27K^3 t_w^3 p \tag{11}$$

Recall from Section 4.4 that for this algorithm, $p \leq n^{3/2}$. 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.

5.3 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 with respect to 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:

$$n^3 = W \propto \frac{5}{3}Kt_s p \log p \tag{12}$$

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 [4] and the DNS algorithm achieves this lower bound on a hypercube.

5.4 Isoefficiency Analysis of the GK Algorithm

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

$$n^3 = W \propto \frac{5}{3} K t_s p \log p \tag{13}$$

$$n^3 = W \propto \frac{125}{27} K^3 t_w^3 p (\log p)^3 \tag{14}$$

The communication overheads and the isoefficiency function of the GK algorithm can be improved by using a more sophisticated scheme for one-to-all broadcast on a hypercube [20]. Due to the complexity of this scheme, we have used the simple one-to-all broadcast scheme in our implementation of this algorithm on the CM-5. We therefore use Equation (7) in Sections 6 for comparing the GK algorithm with the other algorithms discussed in this paper. In the next subsection we give the expressions for the run time and the isoefficiency function of the GK algorithm with the improved one-to-all broadcast.

5.4.1 GK Algorithm With Improved Communication

In the description of the GK algorithm in Section 4.6, the communication model on a hypercube assumes that the one-to-all broadcast of a message of size m on p hypercube processors takes $t_s \log p + t_w m \log p$ time. In [20], Johnsson and Ho present a more sophisticated one-to-all broadcast scheme that will reduce this time to $t_s \log p + t_w m + 2t_w \log p \lceil \sqrt{\frac{t_s m}{t_w \log p}} \rceil$. Using this scheme, the sub-blocks of matrices A and B can be transported to their destination processors in time $4t_w \frac{n^2}{p^{2/3}} + \frac{4}{3} t_s \log p + 8 \frac{n}{p^{1/3}} \sqrt{\frac{1}{3} t_s t_w \log p}$ with the condition that $\sqrt{\frac{3 t_s n^2}{t_w p^{1/3} \log p}}$ is considered equal to 1 if $3 t_s n^2 < t_w p^{1/3} \log p$. A communication pattern similar to that used for this one-to-all broadcast can be used to gather and sum up the sub-blocks of the result matrix C with a communication time of $t_w \frac{n^2}{p^{2/3}} + \frac{1}{3} t_s \log p + 2 \frac{n}{p^{1/3}} \sqrt{\frac{1}{3} t_s t_w \log p}$.

An analysis similar to that in Section 5.4 can be performed to obtain the isoefficiency function for this scheme. It can be shown that the asymptotically highest isoefficiency term now is $\frac{5}{3} t_s p \log p$ which is an improvement over $O(p (\log p)^3)$ isoefficiency function for the naive broadcasting scheme.

The broadcasting scheme of [20] requires that the message be broken up into packets and an optimal packet size to obtain the broadcast time given above is $\sqrt{\frac{t_s m}{t_w \log p}}$ for a message of size m. This means that in the GK algorithm, $\frac{n^2}{p^{2/3}} > \frac{t_s}{t_w} \log p$, or $n^3 = W > (\frac{t_s}{t_w})^{1.5} p (\log p)^{1.5}$. In other words, Johnsson's scheme of reducing the communication time is effective only when there is enough data to be sent on all the channels. This imposes a lower limit on the granularity of the problem being solved. In case of the matrix multiplication algorithm under study in this section, the scalability implication of this is that the problem size has to grow at least as fast as $O(p (\log p)^{1.5})$ with respect to p. Thus the effective isoefficiency function of the GK algorithm with Johnsson's one-to-all broadcast scheme on the hypercube is only $O(p (\log p)^{1.5})$ and not $O(p \log p)$ as might appear from the reduced communication terms.

However, if the message startup time t_s is close to zero (as might be the case for an SIMD machine), the packet size can be as small as one word and an isoefficiency function of $O(p \log p)$ is realizable.

5.5 Summary of Scalability Analysis

Subsections 5.1 through 5.4 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. In this section and the rest of this paper, we skip the discussion of the simple algorithm and Fox's algorithm because the expressions for their iso-efficiency functions differ with that for Cannon's algorithm by small constant factors only.

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

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

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.

6 Relative Performance of the Four Algorithms on a Hypercube

The isoefficiency functions of the four matrix multiplication algorithms predict their relative performance for a large number of processors and large problem sizes. But for moderate values of n and p, a seemingly less scalable parallel formulation can outperform the one that has an asymptotically smaller isoefficiency function. In this subsection, we derive the exact conditions under which each of these four algorithms yields the best performance.

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 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.

$$T_o^{(Cannon)} = 2t_s p^{3/2} + 2t_w n^2\sqrt{p} = T_o^{(GK)} = \frac{5}{3}t_s p \log p + \frac{5}{3}t_w n^2 p^{1/3}\log p$$

$$n_{Equal-T_o}(p) = \sqrt{\frac{(5/3p\log p - 2p^{3/2})t_s}{(2\sqrt{p} - 5/3p^{1/3}\log p)t_w}} \tag{15}$$

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

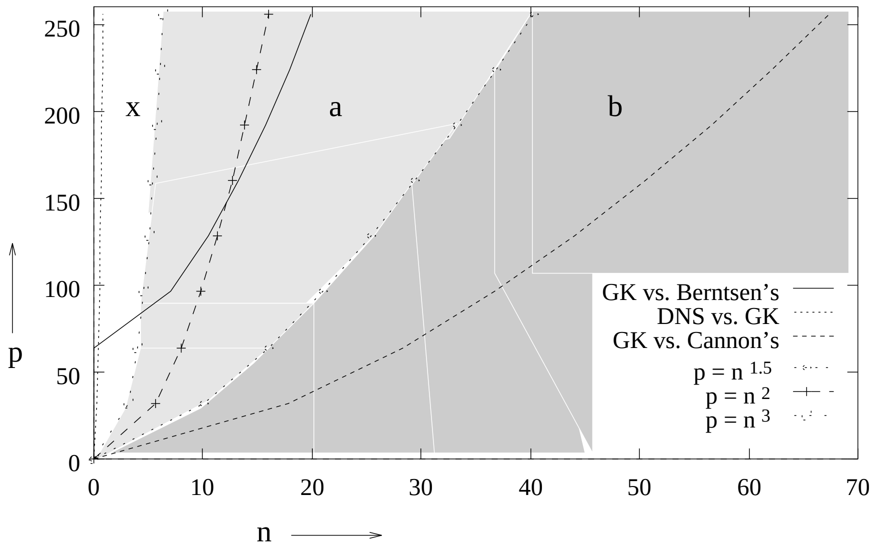

Figure 1: *A comparison of the four algorithms for* $t_w = 3$ *and* $t_s = 150$.

an **x** is the one 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[3] 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 with respect to the unit computation time) and reducing the message startup time. By observing the $n_{Equal-T_o}$ curves and the regions of

†nCUBE2 is a trademark of the Ncube corporation.

[3]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.

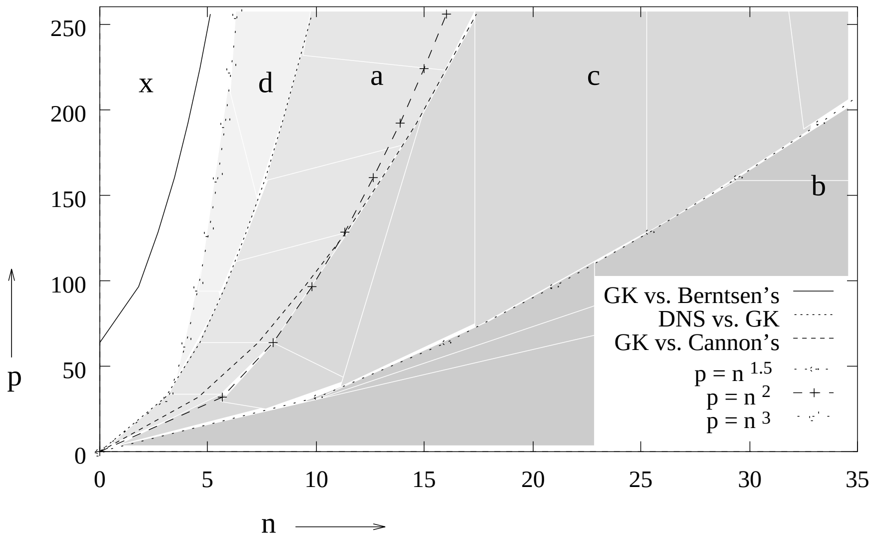

Figure 2: *A comparison of the four algorithms for* $t_w = 3$ *and* $t_s = 10$*.*

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[4]. 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 $p < n^{3/2}$.

7 Scalabilities of Different Algorithms With Simultaneous Communication on All Hypercube Channels

On certain parallel machines like the nCUBE2, the hardware supports simultaneous communication on all the channels. This feature of the hardware can be utilized to significantly reduce the communication cost of certain operations involving broadcasting and personalized communication [20]. In this section we investigate as to what extent can the performance of the algorithms described in Section 4 can be improved by utilizing simultaneous communication on all the $\log p$ ports of the hypercube processors.

Cannon's algorithm (Section 4.2), Berntsen's algorithm (Section 4.4) and the pipelined

[4] 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.

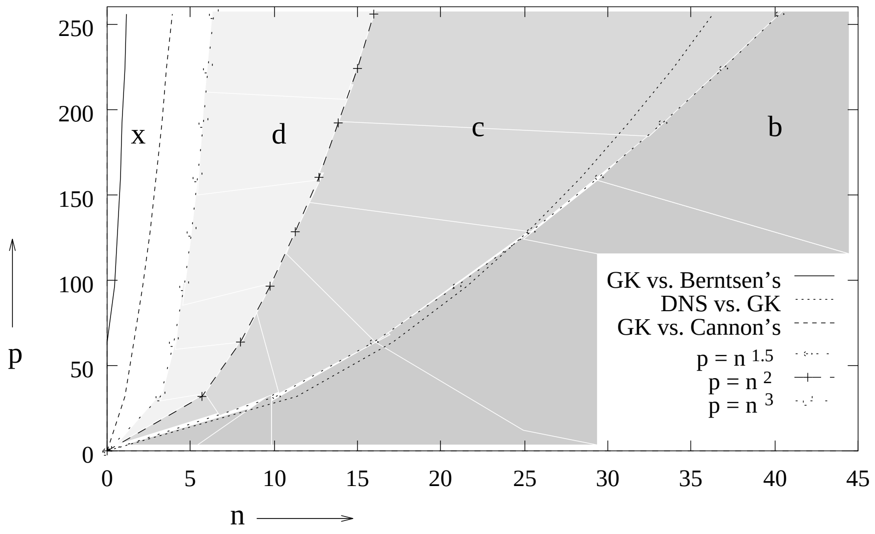

Figure 3: *A comparison of the four algorithms for* $t_w = 3$ *and* $t_s = 0.5$.

version of Fox's algorithm employ only nearest neighbor communication and hence can benefit from simultaneous communication by a constant factor only as the subbocks of matrices A and B can now be transferred simultaneously. The DNS algorithm can also gain only a constant factor in its communication terms as all data messages are only one word long. Hence, among the algorithms discussed in this paper, the ones that can potentially benefit from simultaneous communications on all the ports are the simple algorithm (or its variations [18]) and the GK algorithm.

7.1 The Simple Algorithm With All Port Communication

This algorithm requires an all-to-all broadcast of the sub-blocks of the matrices A and B among groups of $\sqrt{p}$ processors. The best possible scheme utilizing all the channels of a hypercube simultaneously can accomplish an all-to-all broadcast of blocks of size $\frac{n^2}{p}$ among $\sqrt{p}$ processors in time $2t_w \frac{n^2\sqrt{p}}{p \log p} + \frac{1}{2} t_s \log p$. Moreover, the communication of the sub-blocks of both A and B can proceed simultaneously. Thus the parallel execution time of this algorithm on a hypercube with simultaneous communication is given by the following equation:

$$T_p = \frac{n^3}{p} + 2t_w \frac{n^2}{\sqrt{p}\log p} + \frac{1}{2} t_s \log p \tag{16}$$

Recall from Section 4.1 that the simple algorithm is not memory efficient. Ho *et al.* [18] give a memory efficient version of this algorithm which has somewhat higher execution time than that given by Equation (16). It can be shown that the isoefficiency function due to communication overheads is only $O(p \log p)$ now, which is a significant improvement over the

$O(p^{1.5})$ isoefficiency function of this algorithm when communication on only one of the $\log p$ ports of a processor was allowed at a time.

However, as mentioned in [18], the lower limit on the message size imposes the condition that $n \geq \frac{1}{2}\sqrt{p}\log p$. This requires that $n^3 = W \geq \frac{1}{8}p^{1.5}(\log p)^3$. Thus the rate at which the the problem size is required to grow with respect to the number of processors in order to utilize all the communication channels of the hypercube is higher than the isoefficiency function of the algorithm implemented on a simple hypercube with one port communication at a time.

7.2 The GK Algorithm With All Port Communication

Using the one-to-all broadcast scheme of [20] for a hypercube with simultaneous all-port communication, the parallel execution time of the GK algorithm can be reduced to the following:

$$T_p = \frac{n^3}{p} + t_s \log p + 9t_w \frac{n^2}{p^{2/3}\log p} + 6\frac{n}{p^{1/3}}\sqrt{t_s t_w} \tag{17}$$

The communication terms now yield an isoefficiency function of $O(p\log p)$, but it can be shown that lower limit on the message size entails the problem size to grow as $O(p(\log p)^3)$ with respect to p which is not any better that the isoefficiency function of this algorithm on a simple hypercube with one port communication at a time.

7.3 Discussion

The gist of the analysis in this section is that allowing simultaneous on all the ports of a processor on a hypercube does not improve the overall scalability of matrix multiplication algorithms. The reason is that simultaneous communication on all channels requires that each processor has large enough chunks of data to transfer to other processors. This imposes a lower bound on the size of the problem that will generate such large messages. In case of matrix multiplication algorithms, the problem size (as a function of p) that can generate large enough messages for simultaneous communication to be useful, turns out to be larger than what is required to maintain a fixed efficiency with only one port communication at a time. However, there will be certain values of n and p for which the modified algorithm will perform better.

8 Isoefficiency as a Function of Technology Dependent Factors

The isoefficiency function can be used not only to determine the rate at which the problem size should grow with respect to the number of processors, but also with respect to a variation in other hardware dependent constants such as the communication speed and processing power of the processors used etc. In many algorithms, these constants contribute a multiplicative term to the isoefficiency function, but in some others they effect the asymptotic isoefficiency of a parallel system (*e.g.*, parallel FFT [14]). For instance, a multiplicative term of $(t_w)^3$ appears in most isoefficiency functions of matrix multiplication algorithms described in this paper. As discussed earlier, t_w depends on the ratio of the data communication speed of the channels to the computation speed of the processors used in the parallel architecture. This means that if the processors of the multicomputer are replaced by k times faster processors, then the problem size will have to be increased by a factor of k^3 in order to obtain the same efficiency. Thus the isoefficiency function for matrix multiplication is very sensitive to the hardware dependent constants of the architecture. For example, in case of Cannon's algorithm, if the number of processors is increased 10 times, one would have to solve a problem 31.6 times bigger in order to get the same efficiency. On the other hand, for small values of t_s (as may be the case with most SIMD machines), if p is kept the same and 10 times faster processors are used, then one would need to solve a 1000 times larger problem to be able to obtain the same efficiency. Hence for certain problem sizes, it may be better to have a parallel computer with k-fold as many processors rather than one with the same number of processors, each k-fold as fast (assuming that the communication network and the bandwidth etc. remain the same). This should be contrasted with the conventional wisdom that suggests that better performance is always obtained using fewer faster processors [2].

9 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 [30] 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 4.2 and the algorithm described in Section 4.6.

On the CM-5, the time taken for one floating point multiplication and addition was measured to be 1.53 microseconds on our implementation. The message startup time for our program was observed to be about 380 microseconds and the per-word transfer time for 4 byte words was observed to be about 1.8 microseconds[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 4.6 will have to be modified slightly. The first part of the procedure to place the elements of matrix A in their respective positions, requires sending the buffer

[5]These values do not necessarily reflect the communication speed of the hardware but the overheads observed for our implementation. For instance, a function call in the program associated with sending or receiving a message could contribute to the message startup overhead.

$a_{(0,j,k)}$ 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:

$$T_p = \frac{n^3}{p} + t_s(\log p + 2) + t_w \frac{n^2}{p^{2/3}}(\log p + 2) \tag{18}$$

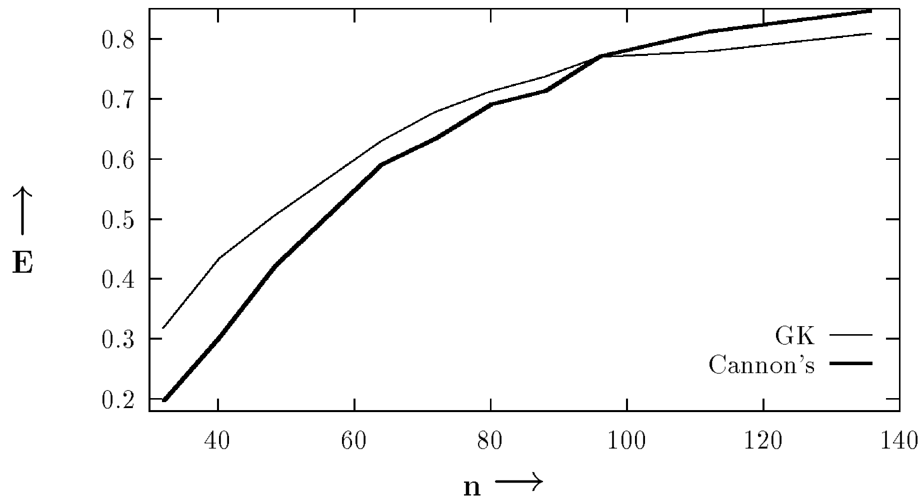

Figure 4: *Efficiency as a function of matrix size for Cannon's algorithm and GK the algorithm for 64 processors.*

Computing the condition for equal T_o for this and Cannon's algorithm by deriving the respective values of T_o from Equations (18) and (3), it can be shown that for 64 processors, Cannon's algorithm should perform better that our algorithm for $n > 83$. Figure 4 shows the efficiency vs n curves for the two algorithms for $p = 64$. It can be seen that as predicted, our algorithm performs better for smaller problem sizes. The experimental cross-over point of the two curves is at $n = 96$. A slight deviation from the derived value of 83 can be explained due to the fact that the values of t_s and t_w are not exactly the same for the two programs. For 512 processors, the predicted cross-over point is for $n = 295$. Since the number of processors has to be a perfect square for Cannon's algorithm on square matrices, in Figure 5, we draw the efficiency vs n curve for $p = 484$ for Cannon's algorithm and for $p = 512$ for the GK algorithm[6]. The cross-over point again closely matches the predicted value. These experiments suggest that the algorithm of Section 4.6 can outperform the classical algorithms like Cannon's for a

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

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 5, 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 significant.

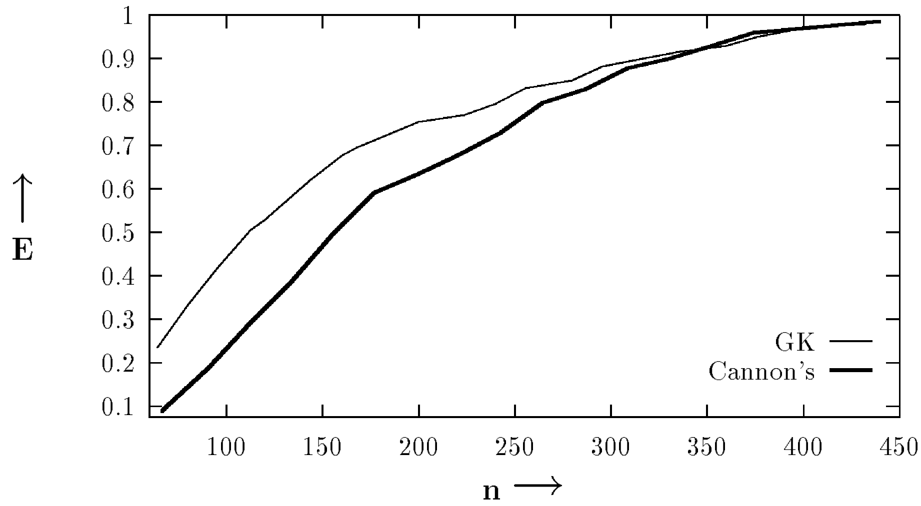

Figure 5: *Efficiency vs matrix size for Cannon's algorithm (p = 484) and the GK algorithm (p = 512).*

10 Concluding Remarks

In this paper we have presented the scalability analysis of a number of matrix multiplication algorithms described in the literature [5, 9, 11, 3, 18]. 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. These factors are the communication related constants of the machine in use such as t_s and t_w, the number of processors employed, and the sizes of the matrices to be multiplied. 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 4.4) turns out to be the least scalable one with an isoefficiency function of $O(p^2)$ due its limited degree of concurrency. The algorithm with the best asymptotic scalability (the DNS algorithm with $O(p \log p)$ isoefficiency function) has a limit on the achievable efficiency, which can be quite low if the message startup time is high. Thus this algorithm too is outperformed by others under a wide range of conditions. For instance, even if t_s is 10 times the values of t_w, the DNS algorithm will perform worse than the GK algorithm for up to almost 10,000 processors for any problem size.

We also show that special hardware permitting simultaneous communication on all the ports of the processors does not improve the overall scalability of the matrix multiplication algorithms on a hypercube. The reason is that simultaneous communication on all ports requires that each processor has large enough messages to transfer so that all the channels can be utilized simultaneously. This imposes a lower bound on the size of the problem that will generate such large messages and hence limits the concurrency of the algorithm. The limited concurrency translates to reduced scalability because for a given problem size more than certain number of processors can not be used.

We discuss the dependence of scalability of parallel matrix multiplication algorithms on technology dependent factors such as communication and computation speeds. Contrary to conventional wisdom, we show that under certain conditions, it may be better to use several slower processors rather than fewer faster processors.

References

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

[2] M. L. Barton and G. R. Withers. Computing performance as a function of the speed, quantity, and the cost of processors. In *Supercomputing '89 Proceedings*, pages 759–764, 1989.

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

[4] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[5] L. E. Cannon. *A cellular computer to implement the Kalman Filter Algorithm.* PhD thesis, Montana State University, Bozman, MT, 1969.

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

[7] 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 First International Conference of the Austrian Center of Parallel Computation.* Springer-Verlag Series Lecture Notes in Computer Science, 1991.

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

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

[10] G. C. Fox, M. Johnson, G. Lyzenga, S. W. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors: Volume 1.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[11] 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.

[12] Ananth Grama, Anshul Gupta, and Vipin Kumar. Isoefficiency: Measuring the scalability of parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August, 1993. Also available as Technical Report TR 93-24, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[13] Ananth Grama, Vipin Kumar, and V. Nageshwara Rao. Experimental evaluation of load balancing techniques for the hypercube. In *Proceedings of the Parallel Computing '91 Conference*, pages 497–514, 1991.

[14] Anshul Gupta and Vipin Kumar. The scalability of FFT on parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):922–932, August 1993. A detailed version available as Technical Report TR 90-53, Department of Computer Science, University of Minnesota, Minneapolis, MN.

[15] Anshul Gupta, Vipin Kumar, and A. H. Sameh. Performance and scalability of preconditioned conjugate gradient methods on parallel computers. Technical Report TR 92-64, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1992. A short version appears in *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 664–674, 1993.

[16] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of parallel methods for a 1024-processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4):609–638, 1988.

[17] 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, 1989.

[18] C.-T. Ho, S. L. 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.

[19] Kai Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability.* McGraw-Hill, New York, NY, 1993.

[20] S. L. Johnsson and C.-T. Ho. Optimum broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9):1249–1268, September 1989.

[21] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin/Cummings, Redwood City, CA, 1994.

[22] Vipin Kumar, Ananth Grama, and V. Nageshwara Rao. Scalable load balancing techniques for parallel computers. Technical Report 91-55, Computer Science Department, University of Minnesota, 1991. To appear in *Journal of Distributed and Parallel Computing*, 1994.

[23] Vipin Kumar and Anshul Gupta. Analyzing scalability of parallel algorithms and architectures. Technical Report TR 91-18, Department of Computer Science Department, University of Minnesota, Minneapolis, MN, 1991. To appear in *Journal of Parallel and Distributed Computing*, 1994. A shorter version appears in *Proceedings of the 1991 International Conference on Supercomputing*, pages 396-405, 1991.

[24] Vipin Kumar and V. N. Rao. Parallel depth-first search, part II: Analysis. *International Journal of Parallel Programming*, 16(6):501–519, 1987.

[25] Vipin Kumar and V. N. Rao. Load balancing on the hypercube architecture. In *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, pages 603–608, 1989.

[26] Vipin Kumar and V. N. Rao. Scalable parallel formulations of depth-first search. In Vipin Kumar, P. S. Gopalakrishnan, and Laveen N. Kanal, editors, *Parallel Algorithms for Machine Intelligence and Vision*. Springer-Verlag, New York, NY, 1990.

[27] Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem: A Summary of Results. In *Proceedings of the International Conference on Parallel Processing*, 1990. An extended version appears in *Journal of Parallel and Distributed Processing*, 13:124–138, 1991.

[28] Vipin Kumar and Vineet Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem. *Journal of Parallel and Distributed Computing*, 13(2):124–138, October 1991. A short version appears in the *Proceedings of the International Conference on Parallel Processing*, 1990.

[29] J. Lee, E. Shragowitz, and S. Sahni. A hypercube algorithm for the 0/1 knapsack problem. In *Proceedings of 1987 International Conference on Parallel Processing*, pages 699–706, 1987.

[30] C. E. Leiserson. Fat-trees : Universal networks for hardware efficient supercomputing. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 393–402, 1985.

[31] Y. W. E. Ma and Denis G. Shea. Downward scalability of parallel architectures. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 109–120, 1988.

[32] Paul Messina. Emerging supercomputer architectures. Technical Report C3P 746, Concurrent Computation Program, California Institute of Technology, Pasadena, CA, 1987.

[33] Cleve Moler. Another look at Amdahl's law. Technical Report TN-02-0587-0288, Intel Scientific Computers, 1987.

[34] Michael J. Quinn and Year Back Yoo. Data structures for the efficient solution of graph theoretic problems on tightly-coupled MIMD computers. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 431–438, 1984.

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

[36] Vineet Singh, Vipin Kumar, Gul Agha, and Chris Tomlinson. Scalability of parallel sorting on mesh multicomputers. *International Journal of Parallel Programming*, 20(2), 1991.

[37] 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, Moffet Field, CA, 1988.

[38] Jinwoon Woo and Sartaj Sahni. Hypercube computing: Connected components. *Journal of Supercomputing*, 1991. Also available as TR 88-50 from the Department of Computer Science, University of Minnesota, Minneapolis, MN.

[39] Jinwoon Woo and Sartaj Sahni. Computing biconnected components on a hypercube. *Journal of Supercomputing*, June 1991. Also available as Technical Report TR 89-7 from the Department of Computer Science, University of Minnesota, Minneapolis, MN.

[40] Patrick H. Worley. The effect of time constraints on scaled speedup. *SIAM Journal on Scientific and Statistical Computing*, 11(5):838–858, 1990.