To appear in IEEE Transactions on Parallel and Distributed Systems, 1995.

Performance and Scalability of Preconditioned Conjugate Gradient Methods on Parallel Computers*

Anshul Gupta, Vipin Kumar, and Ahmed Sameh

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

TR 92-64, November 1992 (Revised May 1994)

Abstract

This paper analyzes the performance and scalability of an iteration of the Preconditioned Conjugate Gradient Algorithm on parallel architectures with a variety of interconnection networks, such as the mesh, the hypercube, and that of the CM-5^{TM†} parallel computer. It is shown that for block-tridiagonal matrices resulting from two dimensional finite difference grids, the communication overhead due to vector inner products dominates the communication overheads of the remainder of the computation on a large number of processors. However, with a suitable mapping, the parallel formulation of a PCG iteration is highly scalable for such matrices on a machine like the CM-5 whose fast control network practically eliminates the overheads due to inner product computation. The use of the truncated Incomplete Cholesky (IC) preconditioner can lead to further improvement in scalability on the CM-5 by a constant factor. As a result, a parallel formulation of the PCG algorithm with IC preconditioner may execute faster than that with a simple diagonal preconditioner even if the latter runs faster in a serial implementation. For the matrices resulting from three dimensional finite difference grids, the scalability is quite good on a hypercube or the CM-5, but not as good on a 2-D mesh architecture. In case of random unstructured sparse matrices with a constant number of non-zero elements in each row, the parallel formulation of the PCG iteration is unscalable on any message passing parallel architecture, unless some ordering is applied on the sparse matrix. The parallel system can be made scalable either if, after re-ordering, the non-zero elements of the $N \times N$ matrix can be confined in a band whose width is $O(N^y)$ for any $y < 1$, or if the number of non-zero elements per row increases as N^x for any $x > 0$. Scalability increases as the number of non-zero elements per row is increased and/or the width of the band containing these elements is reduced. For unstructured sparse matrices, the scalability is asymptotically the same for all architectures. Many of these analytical results are experimentally verified on the CM-5 parallel computer.

*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

Solving large sparse systems of linear equations is an integral part of mathematical and scientific computing and finds application in a variety of fields such as fluid dynamics, structural analysis, circuit theory, power system analysis, surveying, and atmospheric modeling. With the availability of large-scale parallel computers, iterative methods such as the Conjugate Gradient method for solving such systems are becoming increasingly appealing, as they can be parallelized with much greater ease than direct methods. As a result there has been a great deal of interest in implementing the Conjugate Gradient algorithm on parallel computers [1, 2, 12, 14, 16, 21, 25, 29, 30]. In this paper, we study performance and scalability of parallel formulations of an iteration of the Preconditioned Conjugate Gradient (PCG) algorithm [6] for solving large sparse linear systems of equations of the form $\mathbf{A}\ \mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a symmetric positive definite matrix. Although, we specifically deal with the Preconditioned CG algorithm only, the analysis of the diagonal preconditioner case applies to the non-preconditioned method also. In fact the results of this paper can be adapted to a number of iterative methods that use matrix-vector multiplication and vector inner product calculation as the basic operations in each iteration.

The scalability of a parallel algorithm on a parallel architecture is a measure of its capability to effectively utilize an increasing number of processors. It is important to analyze the scalability of a parallel system because one can often reach very misleading conclusions regarding the performance of a large parallel system by simply extrapolating the performance of a similar smaller system. Many different measures have been developed to study the scalability of parallel algorithms and architectures [3, 7, 10, 15, 19, 22, 28, 32, 33]. In this paper, we use the isoefficiency metric [18, 7, 19] to study the scalability of an iteration of the PCG algorithm on some important architectures. The isoefficiency function of a combination of a parallel algorithm and a parallel architecture relates the problem size to the number of processors necessary for an increase in speedup in proportion to the number of processors. Isoefficiency analysis has been found to be very useful in characterizing the scalability of a variety of parallel algorithms [9, 8, 13, 17, 23, 27, 26, 31]. An important feature of isoefficiency analysis is that it succinctly captures the effects of characteristics of the parallel algorithm as well as the parallel architecture on which it is implemented, in a single expression.

The scalability and the overall performance of an iteration of the PCG schemes on a parallel computer depends on - *(i)* hardware parameters such as the interconnection network and communication and computation speeds, *(ii)* the characteristics of the matrix $\mathbf{A}$ such as its degree of sparsity and the arrangement of the non-zero elements within it, and *(iii)* the use and the choice of preconditioners. This paper analyzes the impact of all three types of factors on the scalability of a PCG iteration on parallel architectures with parallel interconnection networks such as mesh, hypercube and that of TMC's CM-5. Two different kinds of matrices are considered. First we analyze the scalability of a PCG iteration on block-tridiagonal matrices resulting from the discretization of a 2-dimensional self-adjoint elliptic partial differential equation via finite differences using natural ordering of grid points. Two commonly used schemes for mapping the data onto the processors are compared, and one is shown to be strictly better than the other. We also consider the use of two different types of preconditioners - a diagonal preconditioner, and one resulting from the truncated Incomplete Cholesky factorization of the matrix of coefficients. The truncated version of Incomplete Cholesky factorization replaces triangular solvers by matrix-vector products, which is henceforth referred to as IC factorization. The results are then extended to the matrices resulting from three dimensional finite difference grids. The second type of matrices that are studied are unstructured sparse symmetric positive definite matrices. The analytical results are verified through extensive experiments on the CM-5 parallel computer. This analysis helps in answering a number of questions, such as -

Given a problem and a parallel machine, what kind of efficiencies are expected?

How should the problem size be increased with the number of processors to maintain a certain efficiency?

Which feature of the hardware should be improved for maximum returns in terms of performance per unit cost?

How does the mapping of the matrix of coefficients onto the processors affect the performance and scalability of a PCG iteration?

How does the truncated Incomplete Cholesky (IC) preconditioner compare with a simple diagonal preconditioner in terms of performance on a parallel computer?

What kind of improvement in scalability can be achieved by re-ordering the sparse matrix?

Which parts of the algorithm dominate in terms of communication overheads and hence determine the overall parallel speedup and efficiency?

The organization of this paper is as follows. Section 2 defines the terms frequently used in the paper. Section 3 describes the data communication model for the parallel architectures considered for analysis in this paper. Section 4 gives an overview of the isoefficiency metric of scalability. The serial PCG algorithm is described in Section 5. The scalability analysis of a PCG iteration on different parallel architectures is presented in Sections 6 and 7 for block-tridiagonal matrices and in Section 8 for unstructured sparse matrices. The experimental results are presented in Section 9, and 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: 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 an optimal (or the best known) sequential algorithm to solve the given problem on a single processor. For example, for multiplying an $N \times N$ matrix with an $N \times 1$ vector, $W = O(N^2)$.

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 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 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, $E = W/pT_P = 1/(1 + \frac{T_o}{W})$.

3 Parallel architectures and message passing costs

In this paper we consider parallel architectures with the following interconnection networks: mesh, hypercube, and that of the CM-5 parallel computer. We assume that in all these architectures, **cut-through** routing is used for message passing. The time required for the complete transfer of a message of size q between two processors that are d connections away; (*i.e.*, there are $d - 1$ processors in between) is given by $t_s + t_h(d - 1) + t_w q$, where t_s is the *message startup time*, t_h (called *per-hop time*) is the time delay for a message fragment to travel from one node to an immediate neighbor, and t_w (called *per-word time*) is equal to $\frac{1}{B}$ where B is the bandwidth of the communication channel between the processors in words/second.

The analytical results presented in the paper are verified through experiments on the CM-5 parallel computer. On CM-5, the fat-tree [20] like communication network provides (almost) simultaneous paths for communication between all pairs of processors. The length of these paths may vary from 2 to $2 \log p$, if the distance between two switches or a processor and a switch is considered to be one unit [20]. Hence, the cost of passing a message of size q between any pair of processors on the CM-5 is approximately $t_s + t_w q + t_h \times O(\log p)$. An additional feature of this machine is that it has a fast control network which can perform certain global operations like *scan* and *reduce* in a small constant time. This feature of the CM-5 makes the overheads of certain operations, like computing the inner product of two vectors, almost negligible compared to the communication overheads involved in the rest of the parallel operations in the CG algorithm.

On most practical parallel computers with cut-through routing, the value of t_h is fairly small. In this paper, we will omit the t_h term from any expression in which the coefficient of t_h is dominated by coefficients of t_w or t_s.

4 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.*, [18, 7, 11, 32]). 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* [18, 7] 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 fixed efficiency, W should be proportional to $T_o(W, p)$ or the following relation must be satisfied:

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

where $e = \frac{E}{1-E}$ is a constant depending on the efficiency to be maintained. Equation (1) is the central relation that is used to determine the isoefficiency function of a parallel system. 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 as the isoefficiency function of the parallel system for efficiency E.

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 those of 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. 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.

5 The serial PCG algorithm

begin
  $i := 0$; $\mathbf{x}_0 := 0$; $\mathbf{r}_0 := \mathbf{b}$; $\rho_0 := ||\mathbf{r}_0||_2^2$;
  while $(\sqrt{\rho_i} > \epsilon ||\mathbf{r}_0||_2)$ and $(i < i_{max})$ do
  begin
    Solve $\mathbf{M}\,\mathbf{z}_i = \mathbf{r}_i$;
    $\gamma_i := \mathbf{r}_i^T\,\mathbf{z}_i$;
    $i := i + 1$;
    if (i = 1) $\mathbf{p}_1 := \mathbf{z}_0$
    else begin
      $\beta_i := \gamma_{i-1}/\gamma_{i-2}$;
      $\mathbf{p}_i := \mathbf{z}_{i-1} + \beta_i \mathbf{p}_{i-1}$;
    end;
    $\mathbf{w}_i := \mathbf{A}\,\mathbf{p}_i$;
    $\alpha_i := \gamma_{i-1}/\mathbf{p}_i^T\,\mathbf{w}_i$;
    $\mathbf{x}_i := \mathbf{x}_{i-1} + \alpha_i \mathbf{p}_i$;
    $\mathbf{r}_i := \mathbf{r}_{i-1} - \alpha_i \mathbf{w}_i$;
    $\rho_i := ||\mathbf{r}_i||_2^2$;
  end; {*while* }
  $\mathbf{x} := \mathbf{x}_i$;
end.

Figure 1: *The Preconditioned Conjugate Gradient algorithm.*

Figure 1 illustrates the PCG algorithm [6] for solving a linear system of equations $\mathbf{A}\,\mathbf{x} = \mathbf{b}$, where $\mathbf{A}$ is a sparse symmetric positive definite matrix. The PCG algorithm performs a few basic operations in each iteration. These are matrix-vector multiplication, vector inner product calculation, scalar-vector multiplication, vector addition and solution of a linear system $\mathbf{M}\,\mathbf{z} = \mathbf{r}$. Here $\mathbf{M}$ is the preconditioner matrix, usually derived from $\mathbf{A}$ using certain techniques. In this paper, we will consider two kinds of preconditioner matrices $\mathbf{M}$ - *(i)* when $\mathbf{M}$ is chosen to be a diagonal matrix, usually derived from the principal diagonal of $\mathbf{A}$, and *(ii)* when $\mathbf{M}$ is obtained through a truncated Incomplete Cholesky (IC) factorization [14, 29] of $\mathbf{A}$. In the following subsections, we determine the serial execution time for each iteration of the PCG algorithm with both preconditioners.

5.1 Diagonal preconditioner

During a PCG iteration for solving a system of N equations, the serial complexity of computing the vector inner product, scalar-vector multiplication and vector addition is $O(N)$. If $\mathbf{M}$ is a diagonal matrix, the complexity of solving **M z = r** is also $O(N)$. If there are m non-zero elements in each row of the sparse matrix **A**, then the matrix-vector multiplication of a CG iteration can be performed in $O(mN)$ time by using a suitable scheme for storing **A**. Thus, with the diagonal preconditioner, each CG iteration involves a few steps of $O(N)$ complexity and one step of $O(mN)$ complexity. As a result, the serial execution time for one iteration of the PCG algorithm with a diagonal preconditioner can be expressed as follows:

$$W = c_1 N + c_2 m N \tag{2}$$

Here c_1 and c_2 are constants depending upon the floating point speed of the computer and m is the number non-zero elements in each row of the sparse matrix.

5.2 The IC preconditioner

In this section, we only consider the case when **A** is a block-tridiagonal matrix of dimension N resulting from the discretization of a 2-dimensional self-adjoint elliptic partial differential equation via finite differences using natural ordering of grid points. Besides the principal diagonal, the matrix **A** has two diagonals on each side of the principal diagonal at distances of 1 and $\sqrt{N}$ from it. Clearly, all the vector operations can be performed in $O(N)$ time. The matrix-vector multiplication operation takes time proportional to $5N$. When **M** is an IC preconditioner, the structure of **M** is identical to that of **A**.

A method for solving **M z = r**, originally proposed for vector machines [29], is briefly described below. A detailed description of the same can be found in [18]. As shown in Section 6, this method is perfectly parallelizable on CM-5 and other architectures ranging from mesh to hypercube. In fact, this method leads to a parallel formulation of the PCG algorithm that is somewhat more scalable and efficient (in terms of processor utilization) than a formulation using a simple diagonal preconditioner.

The matrix **M** can be written as $\mathbf{M} = (\mathbf{I} - \mathbf{L})\mathbf{D}(\mathbf{I} - \mathbf{L}^T)$, where **D** is a diagonal matrix and **L** is a strictly lower triangular matrix with two diagonals corresponding to the two lower diagonals of **M**. Thus, the system **Mz = r** may be solved by the following steps:

solve $(\mathbf{I} - \mathbf{L})\mathbf{u} = \mathbf{r}$
solve $\mathbf{D}\mathbf{v} = \mathbf{u}$
solve $(\mathbf{I} - \mathbf{L}^T)\mathbf{z} = \mathbf{v}$

Since **L** is strictly lower triangular (*i.e.*, $\mathbf{L}^N = 0$), **u** and **z** may be written as $\mathbf{u} = \Sigma_{i=0}^{N-1}\mathbf{L}^i\mathbf{r}$ and $\mathbf{z} = \Sigma_{i=0}^{N-1}(\mathbf{L}^i)^T\mathbf{v}$. These series may be truncated to $(k+1)$ terms where $k \ll N$ because **M** is diagonally dominant [14, 29]. In our formulation, we form the matrix $\tilde{\mathbf{L}} = (\mathbf{I} + \mathbf{L} + \mathbf{L}^2 + ... + \mathbf{L}^k)$ explicitly. Thus, solving **Mz = r** is equivalent to

(i) $\mathbf{u} \approx \tilde{\mathbf{L}}\mathbf{r}$
(ii) $\mathbf{v} \approx \mathbf{D}^{-1}\mathbf{u}$
(iii) $\mathbf{z} \approx \tilde{\mathbf{L}}^T\mathbf{v}$

The number of diagonals in the matrix $\tilde{\mathbf{L}}$ is equal to $\frac{(k+1)(k+2)}{2}$. These diagonals are distributed in $k+1$ clusters at distances of $\sqrt{N}$ from each other. The first cluster, which includes the principal diagonal, has $k+1$ diagonals, and then the number of diagonals in each cluster decreases by one. The last cluster has only one diagonal which is at a distance of $k\sqrt{N}$ from the principal diagonal. Thus solving

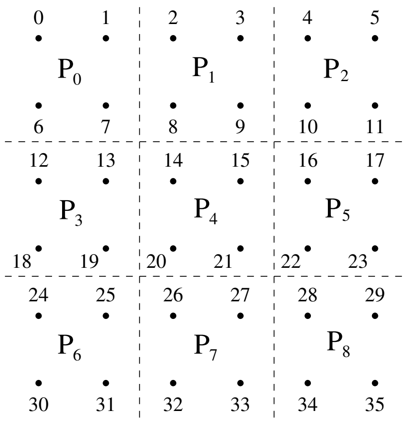


Figure 2: *Partitioning a finite difference grid on a processor mesh.*

the system **M z = r**, in case of the IC preconditioner, is equivalent to performing one vector division (step (ii)) and two matrix-vector multiplications (steps (i) and (iii)), where each matrix has $\frac{(k+1)(k+2)}{2}$ diagonals. Hence the complexity of solving **M z = r** for this case is proportional to $(k+1)(k+2)N$ and the serial execution time of one complete iteration is given by the following equation:

$$W = (c_1 + (5 + (k+1)(k+2))c_2) \times N$$

Here c_1 and c_2 are constants depending upon the floating point speed of the computer. The above equation can be written compactly as follows by putting $\eta(k) = c_1 + c_2(5 + (k+1)(k+2))$.

$$W = \eta(k)N \tag{3}$$

6 Scalability analysis: block-tridiagonal matrices

In this section we consider the parallel formulation of the PCG algorithm with block-tridiagonal matrix of coefficients resulting from a square two dimensional finite difference grid with natural ordering of grid points. Each point on the grid contributes one equation to the system **A x = b**; *i.e.*, one row of matrix **A** and one element of the vector **b**.

6.1 The parallel formulation

The points on the finite difference grid can be partitioned among the processors of a mesh connected parallel computer as shown in Figure 2. Since a mesh can be mapped onto a hypercube or a fully connected network, a mapping similar to the one shown in Figure 2 will work for these architectures as well. Let the p processors be numbered from $P_{(0,0)}$ to $P_{(\sqrt{p}-1,\sqrt{p}-1)}$. If the N points of the grid are indexed from $[0,0]$ to $[\sqrt{N}-1, \sqrt{N}-1]$, then processor $P_{(i,j)}$ stores a square subsection of the grid with row indices varying from $[i\sqrt{N/p}]$ to $[(i+1)\sqrt{N/p}-1]$ and column indices varying from $[j\sqrt{N/p}]$ to $[(j+1)\sqrt{N/p}-1]$. If the grid points are ordered naturally, point (q,r) is responsible for row number $[q\sqrt{N}+r]$ of matrix **A** and the element with index $[q\sqrt{N}+r]$ of the vector **b**. Hence, once the grid points are assigned to the processors, it is easy to determine the rows of matrix **A** and the elements of vector **b** that belong to each processor.

In the PCG algorithm, the scalar-vector multiplication and vector addition operations do not involve any communication overhead, as all the required data is locally available on each processor and the results are stored locally as well. If the diagonal preconditioner is used, then even solving the system **M z = r** does not require any data communication because the resultant vector **z** can be obtained by simply dividing each element of **r** by the corresponding diagonal element of **M**, both of which are locally available on each processor. Thus, the operations that involve any communication overheads are computation of inner products, matrix-vector multiplication, and, in case of the IC preconditioner, solving the system **M z = r**.

In the computation of the inner product of two vectors, the corresponding elements of each vector are multiplied locally and these products are summed up. The value of the inner product is the sum of these partial sums located at each processor. The data communication required to perform this step involves adding all the partial sums and distributing the resulting value to each processor.

In order to perform parallel matrix-vector multiplication involving the block-tridiagonal matrix, each processor has to acquire the vector elements corresponding to the column numbers of all the matrix elements it stores. It can be seen from Figure 2 that each processor needs to exchange information corresponding to its $\sqrt{\frac{N}{p}}$ boundary points with each of its four neighboring processors. After this communication step, each processor gets all the elements of the vector it needs to multiply with all the matrix elements it stores. Now the multiplications are performed and the resultant products are summed and stored locally on each processor.

A method for solving **M z = r** for the IC preconditioner **M** has been described in Section 5.2. This computation involves multiplication of a vector with a lower triangular matrix $\tilde{\mathbf{L}}$ and an upper triangular matrix $\tilde{\mathbf{L}}^T$, where each triangular matrix has $\frac{(k+1)(k+2)}{2}$ diagonals arranged in the fashion described in Section 5.2. If the scheme of partitioning **A** among the processors (every processor stores $\sqrt{\frac{N}{p}}$ clusters of $\sqrt{\frac{N}{p}}$ matrix rows each) is extended to $\tilde{\mathbf{L}}$ and $\tilde{\mathbf{L}}^T$, then it can be shown that for $\sqrt{\frac{N}{p}} > k$ the data communication for performing these matrix-vector multiplications requires each processor in the mesh to send $k\sqrt{\frac{N}{p}}$ vector elements to its immediate north, east, south and west neighbors.

6.2 Communication overheads

In this section we determine the overall overhead due to parallel processing in a PCG iteration. As discussed in the previous subsection, the operations that incur communication overheads are computation of inner products, matrix-vector multiplication and solving the system **M z = r**. Let these three components of T_o be $T_o^{Inner-Prod}$, $T_o^{Matrix-Vector}$, and $T_o^{PC-solve}$, respectively. In order to compute each component of T_o, first we compute the time spent by each processor in performing data communication for the operation in question. The product of this time with p gives the total time spent by all the processors in performing this operation and T_o is the sum of each of these individual components.

6.2.1 Overhead in computing the inner products

The summation of p partial sums (one located at each processor) can be performed through recursive doubling in $(t_s + t_w)\log p$ time on a hypercube, and in $(t_s + t_w)\log p + t_h\sqrt{p}$ time on a two dimensional mesh. It takes the same amount of time to broadcast the final result to each processor. On a machine like the CM-5, such operations (known as *reduction* operations) are performed by the control network in a small constant time which can be ignored in comparison with the overhead incurred in other parts of the algorithm, such as, matrix-vector multiplication. The following equations give the expressions for the total overhead for each iteration over all the processors for computing the three inner products (lines 6, 14 and 17 in Figure 1) on different architectures. In these equations t_w is ignored in comparison with t_s as the latter is much larger in most practical machines.

$$T_o^{Inner-Prod} \approx 6(t_s \log p + t_h\sqrt{p}) \times p \qquad (2-D\ mesh) \tag{4}$$

$$T_o^{Inner-Prod} \approx 6t_s \log p \times p \qquad (Hypercube) \tag{5}$$

$$T_o^{Inner-Prod} \approx 0 \qquad (CM-5) \tag{6}$$

6.2.2 Overhead due to matrix-vector multiplication

During matrix-vector multiplication, each processor needs to exchange vector elements corresponding to its boundary grid points with each of its four neighboring processors. This can be done in $4t_s + 4t_w\sqrt{\frac{N}{p}}$ time on a mesh, hypercube or a virtually fully connected network like that of the CM-5. The total overhead for this step is given by the following equation:

$$T_o^{Matrix-Vector} = 4(t_s + t_w\sqrt{\frac{N}{p}}) \times p \tag{7}$$

6.2.3 Overhead in solving M z = r

If a simple diagonal preconditioner is used, then this step does not require any communication. For the case of the IC preconditioner, as discussed in Section 6.1, the communication pattern for this step is identical to that for matrix-vector multiplication, except that $k\sqrt{\frac{N}{p}}$ vector elements are now exchanged at each processor boundary. The required expression for the overall overhead for this step is as follows.

$$T_o^{PC-solve} = 0 \qquad (diagonal\ preconditioner) \tag{8}$$

$$T_o^{PC-solve} = 4(t_s + t_w k\sqrt{\frac{N}{p}}) \times p \qquad (IC\ preconditioner) \tag{9}$$

6.2.4 Total overhead

Now, having computed each of its components, we can write the expressions for the overall T_o using equations (4) to (9). The following equations give the required approximate expressions (after dropping the insignificant terms, if any) for T_o.

- **The CM-5**

$$T_o = 4(t_s p + t_w \sqrt{pN}) \qquad (diagonal\ preconditioner) \tag{10}$$

$$T_o = 4(2t_s p + t_w(k+1)\sqrt{pN}) \qquad (IC\ preconditioner) \tag{11}$$

- **Hypercube**

$$T_o = 6t_s p \log p + 4t_w \sqrt{pN} \qquad (diagonal\ preconditioner) \tag{12}$$

$$T_o = 6t_s p \log p + 4(k+1)t_w \sqrt{pN} \qquad (IC\ preconditioner) \tag{13}$$

- **Mesh**

$$T_o = 6t_s p \log p + 4t_w \sqrt{pN} + 6t_h p \sqrt{p} \qquad (diagonal\ preconditioner) \tag{14}$$

$$T_o = 6t_s p \log p + 4(k+1)t_w \sqrt{pN} + 6t_h p \sqrt{p} \qquad (IC\ preconditioner) \tag{15}$$

6.3 Isoefficiency analysis

Since we perform the scalability analysis with respect to one PCG iteration, the problem size W will be considered to be $O(N)$ and we will study the rate at which N needs to grow with p for a fixed efficiency as a measure of scalability. If $T_o(W, P)$ is the total overhead, the efficiency E is given by $\frac{W}{W+T_o(W,p)}$. Clearly, for a given N, if p increases, then E will decrease because $T_o(W, p)$ increases with p. On the other hand, if N 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 N as a function of p. In our parallel formulation, T_o has a number of different terms due to t_s, t_w, t_h, etc. When there are multiple terms of different orders of magnitude with respect to p and N in W or T_o, it is often impossible to obtain the isoefficiency function as a closed form function of p. For a parallel algorithm-architecture combination, as p and W increase, 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 asymptotic isoefficiency function of the entire computation.

6.3.1 Diagonal preconditioner

Since the number of elements per row (m) in the matrix of coefficients is five, from Equation (2), we obtain the following expression for W:

$$W = N(c_1 + 5c_2) \tag{16}$$

Now we will use Equations (10), (12), (14) and (16) to compute the isoefficiency functions for the case of diagonal preconditioner on different architectures.

- **The CM-5**

  According to Equation (1), in order to determine the isoefficiency term due to t_s, W has to be proportional to $4et_s p$ (see Equation (10)), where $e = \frac{E}{1-E}$ and E is the desired efficiency that has to be maintained. Therefore,

$$N(c_1 + 5c_2) \propto 4et_s p$$

$$=> \quad N \propto p\frac{4et_s}{c_1 + 5c_2} \tag{17}$$

  The term due to t_w in T_o is $4t_w\sqrt{pN}$ (see Equation (10)) and the associated isoefficiency condition is determined as follows:

$$N(c_1 + 5c_2) \propto 4et_w\sqrt{pN}$$

$$=> \quad \sqrt{N} \propto \sqrt{p}\frac{4et_w}{c_1 + 5c_2}$$

$$=> \quad N \propto p(\frac{4et_w}{c_1 + 5c_2})^2 \tag{18}$$

  According to both Equations (17) and (18), the overall isoefficiency function for the PCG algorithm with a simple diagonal preconditioner is $O(p)$; *i.e*, it is a highly scalable parallel system which requires only a linear growth of problem size with respect to p to maintain a certain efficiency.

- **Hypercube**

  Since the terms due to t_w are identical in the overhead functions for both the hypercube and the CM-5 architectures, the isoefficiency condition resulting from t_w is still determined by Equation (18). The term associated with t_s yields the following isoefficiency function:

$$=> \quad N \propto \frac{6et_s}{c_1 + 5c_2} p \log p \tag{19}$$

  Since Equation (19) suggests a higher growth rate for the problem size with respect to p to maintain a fixed E, it determines the overall isoefficiency function which is $O(p \log p)$. Also, t_s has a higher impact on the efficiency on a hypercube than on the CM-5.

- **Mesh**

  The isoefficiency term due to t_s will be the same as in Equation (19) because the terms due to t_s in the overhead functions for the hypercube and the mesh are identical. Similarly, the isoefficiency term due to t_w will be the same as in Equation (18). Balancing W against the term due to t_h in Equation (14), we get

$$N(c_1 + 5c_2) \propto 6et_h p\sqrt{p}$$

$$=> \quad N \propto \frac{6et_h}{c_1 + 5c_2} p^{1.5} \tag{20}$$

  Now N has to grow as $O(p)$ to balance the overheads due to t_w (Equation (18)), as $O(p \log p)$ to balance the overhead due to t_s (Equation (19)), and as $O(p^{1.5})$ to balance the overhead due to t_h (Equation (20)). Clearly, Equation (20) determines the asymptotic isoefficiency function for the mesh.

6.3.2 IC preconditioner

The following overall isoefficiency functions can be derived for the case of the IC preconditioner using the analysis similar to that in Section 6.3.1, taking the expression for W from Equation (3) and expressions for T_o from Equations (11), (13) and (15) for various architectures:

$$N \propto p(\frac{4et_w(k+1)}{\eta(k)})^2 \qquad (CM-5) \tag{21}$$

$$N \propto \frac{6et_s}{\eta(k)} p \log p \qquad (hypercube) \tag{22}$$

$$N \propto \frac{6et_h}{\eta(k)} p\sqrt{p} \qquad (mesh) \tag{23}$$

The isoefficiency functions given by the above equations are asymptotically the same as those for the diagonal preconditioner (Equations (17) to (20)), but with different constants. The impact of these constants on the overall efficiency and scalability of the PCG iteration is discussed in Section 6.4.

6.4 Discussion

A number of interesting inferences can be drawn from the scalability analysis performed in Section 6.3. For a typical MIMD mesh or hypercube with $t_s \gg t_w$, matrix-vector multiplication and solution of **M z = r** with the preconditioner **M** incur relatively small communication overheads compared to the computation of inner-product of vectors. For these architectures, the inner-products calculation contributes the overhead term that determines the overall isoefficiency function and the total communication cost is dominated by the message startup time t_s. In contrast, on the CM-5, the communication overhead in the inner product calculation is minimal due to the control network. As a result, the CM-5 is ideally scalable for an iteration of this algorithm; *i.e.*, speedups proportional to the number of processors can be obtained by simply increasing N linearly with p. Equivalently, bigger instances

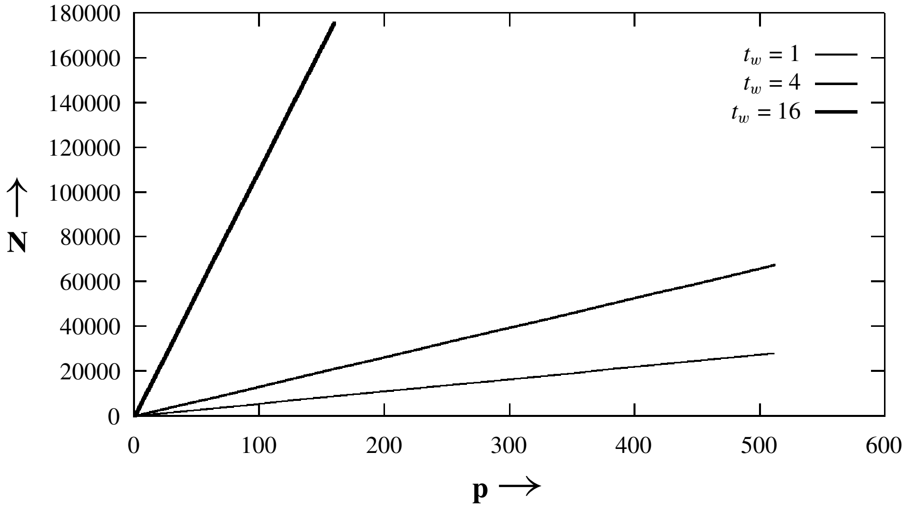


Figure 3: *Isoefficiency curves for* $E = 0.5$ *with a fixed processor speed and different values of channel bandwidth.*

of problems can be solved in a fixed given time by using linearly increasing number of processors. In the absence of the control network, even on the CM-5 the overhead due to message startup time in the inner product computation would have dominated and the isoefficiency function of the PCG algorithm would have been greater than $O(p)$. Thus, for this application, the control network is highly useful.

There are certain iterative schemes, like the Jacobi method [6], that require inner product calculation only for the purpose of performing a convergence check. In such schemes, the parallel formulation can be made almost linearly scalable even on mesh and hypercube architectures by performing the convergence check once in a few iterations. For instance, the isoefficiency function for the Jacobi method on a hypercube is $O(p \log p)$. If the convergence check is performed once every $\log p$ iterations, the amortized overhead due to inner product calculation will be $O(p)$ in each iteration, instead of $O(p \log p)$ and the isoefficiency function of the modified scheme will be $O(p)$. Similarly, performing the convergence check once in every $\sqrt{p}$ iterations on a mesh architecture will result in linear scalability.

Equations (17) and (18) suggest that the PCG algorithm is highly scalable on a CM-5 type architecture and a linear increase in problem size with respect to the number of processors is sufficient to maintain a fixed efficiency. However, we would like to point out as to how hardware related parameters other than the number of processors affect the isoefficiency function. According to Equation (18), N needs to grow in proportion to the square of the ratio of t_w to the unit computation time on a processor. According to Equation (17), N needs to grow in proportion to the ratio of t_s to the unit computation time. Thus isoefficiency is also a function of the ratio of communication and computation speeds. Figure 3 shows theoretical isoefficiency curves for different values of t_w for a

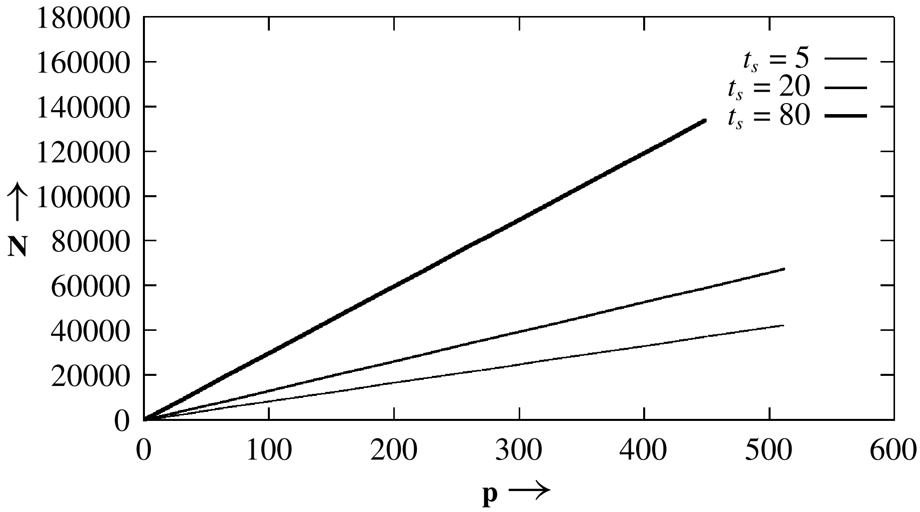


Figure 4: *Isoefficiency curves for $E = 0.5$ with a fixed processor speed and different values of message startup time.*

hypothetical machine with fixed processor speed ($(c_1 + 5c_2) = 2$ microseconds[1]) and message startup time ($t_s = 20$ microseconds). Figure 4 shows isoefficiency curves for different values of t_s for the same processor speed with $t_w = 4$ microseconds. These curves show that the isoefficiency function is much more sensitive to changes in t_w than t_s. Note that t_s and t_w are normalized with respect to CPU speed. Hence, effectively t_w could go up if either the CPU speed increases or inter-processor communication bandwidth decreases. Figure 3 suggests that it is very important to have a balance between the speed of the processors and the bandwidth of the communication channels, otherwise good efficiencies will be hard to obtain; *i.e.*, very large problems will be required.

Apart from the computation and communication related constants, isoefficiency is also a function of the efficiency that is desired to be maintained. N needs to grow in proportion to $(\frac{E}{1-E})^2$ in order to balance the useful computation W with the overhead due to t_w (Equation (18)) and in proportion to $\frac{E}{1-E}$ to balance the overhead due to t_s (Equation (17)). Figure 5 graphically illustrate the impact of desired efficiency on the scalability of a PCG iteration. The figure shows that as higher and higher efficiencies are desired, it becomes increasingly difficult to obtain them. An improvement in the efficiency from 0.3 to 0.5 takes little effort, but it takes substantially larger problem sizes for a similar increase in efficiency from 0.7 to 0.9. The constant $(\frac{et_w}{c_1+5c_2})^2$ in Equation (18) indicates that a better balance between communication channel bandwidth and the processor speed will reduce the impact of increasing the efficiency on the rate of growth of problem size and higher efficiencies will be obtained more readily.

The isoefficiency functions for the case of the IC preconditioner for different architectures given

[1]This corresponds to a throughput of roughly 10 MFLOPS. On a fully configured CM-5 with vector units, a throughput of this order can be achieved very easily.

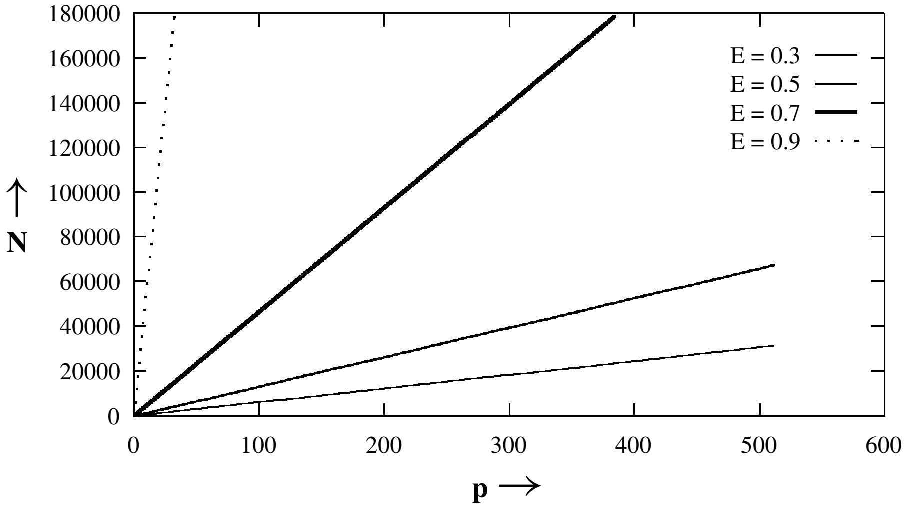
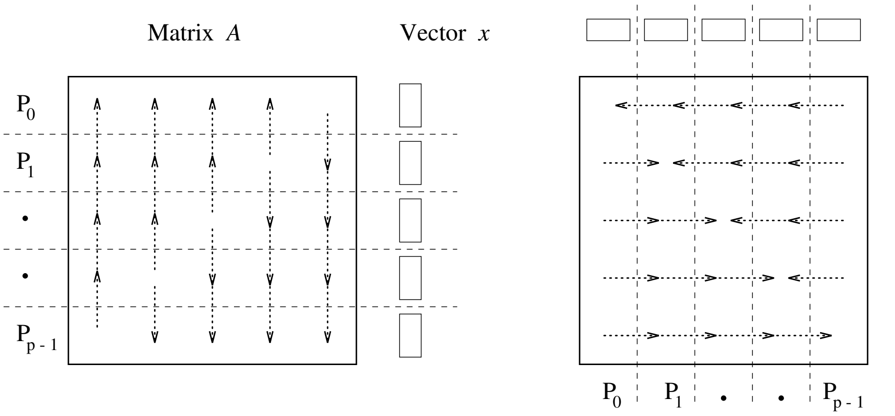


Figure 5: *Isoefficiency curves for different efficiencies with* $t_s = 20$ *and* $t_w = 4$.

by Equations (21) to (22) are of the same order of magnitude as those for the case of the diagonal preconditioner given by Equations (17) to (20). However, the constants associated with the isoefficiency functions for the IC preconditioner are smaller due to the presence of $\eta(k)$ in the denominator which is $c_1 + 5c_2 + (k+1)(k+2)c_2$. As a result, the rate at which the problem size should increase to maintain a particular efficiency will be asymptotically the same for both preconditioners, but for the IC preconditioner the same efficiency can be realized for a smaller problem size than in the case of the diagonal preconditioner. Also, for given p and N, the parallel implementation with the IC preconditioner will yield a higher efficiency and speedup than one with the diagonal preconditioner. Thus, if enough processors are used, a parallel implementation with the IC preconditioner may execute faster than one with a simple diagonal preconditioner even if the latter was faster in a serial implementation. The reason is that the number of iterations required to obtain a residual whose norm satisfies a given constraint does not increase with the number of processors used. However, the efficiency of execution of each iteration will drop more rapidly in case of the diagonal preconditioner that in case of the IC preconditioner as the number of processors are increased.

It can be shown that the scope of the results of this section is not limited to the type of block-tridiagonal matrices described in Section 5.2 only. The results hold for all symmetric block-tridiagonal matrices where the distance of the two outer diagonals from the principal diagonal is N^r $(0 < r < 1)$. Such a matrix will result from a rectangular $N^r \times N^{1-r}$ finite difference grid with natural ordering of grid points. Similar scalability results can also be derived for matrices resulting from three dimensional finite difference grids. These matrices have seven diagonals and the scalability of the parallel formulations of an iteration of the PCG algorithm on a hypercube or the CM-5 is the same as in case of block-tridiagonal matrices. However, for the mesh architecture, the results will be different. On a two dimensional mesh of processors, the isoefficiency due to matrix-vector multiplication will be



(a) All-to-all broadcast with rowwise striping

(b) Multinode accumulation with columnwise striping

Figure 6: Data communication during matrix-vector multiplication in a simple mapping scheme for the block-tridiagonal matrix.

$O(p^{3/2})$ for the matrices resulting from 3-D finite difference grids. Thus, unlike the block-tridiagonal case, here the overheads due to both matrix vector multiplication and inner-product computation are equally dominant on a mesh.

7 An alternate mapping for the block-tridiagonal case and its impact on scalability

A scheme for mapping the matrix of coefficients and the vectors onto the processors has been presented in Section 6.1. In this section, we describe a different mapping and analyze its scalability. Given the matrix of coefficients **A** and the vector **b**, this mapping is fairly straightforward and has often been used in parallel implementations of the PCG algorithm due to its simplicity [16, 2, 21, 14]. Here we will compare this mapping with the one discussed in Section 6.1 in the context of a CM-5 type architecture.

According to this scheme, the matrix **A** and the vector **b** are partitioned among p processors as shown in Figure 6. The rows and the columns of the matrix and the elements of the vector are numbered from 0 to $N-1$. The *diagonal storage* scheme is the natural choice for storing the sparse matrix in this case. There are N elements in the principal diagonal and they are indexed from [0] to $[N-1]$. The two diagonals in the upper triangular part have $N-1$ and $N-\sqrt{N}$ elements indexed from [0] to $[N-2]$ and from [0] to $[N-\sqrt{N}-1]$ respectively. The two diagonals in the lower triangular part of **A** have the same lengths as their upper triangular counterparts. These are indexed from [1] to $[N-1]$ and from $[\sqrt{N}]$ to $[N-1]$ respectively. Processor P_i stores $\frac{N}{p}i$ to $\frac{N}{p}(i+1)-1$ rows of matrix **A** and elements with indices from $[\frac{N}{p}i]$ to $[\frac{N}{p}(i+1)-1]$ of vector **b**. The preconditioner and all the other vectors used in the computation are partitioned similarly.

7.1 Communication overheads

As for the previous mapping, the communication overheads in the inner-product computation on the CM-5 are insignificant. The only significant overheads are incurred in matrix-vector multiplication and solving the system **M z = r** in case of the IC preconditioner.

The structure of matrix **A** is such that while multiplying it with a vector, the ith element of the vector has to be multiplied with elements of the diagonals of the matrix with indices $[i]$, $[i+1]$, $[i+\sqrt{N}]$, $[i-1]$ and $[i-\sqrt{N}]$. Thus each element of the vector is required at locations which are at a distance of 1 or $\sqrt{N}$ on either side. The data communication required for the multiplication of the two inner diagonals with appropriate elements of the vector can be accomplished by each pair of neighboring processors exchanging the vector elements located at the partition boundaries. This can be accomplished in $2(t_s + t_w)$ time. The amount of data communication required for the multiplication of the two outer diagonals with appropriate elements of the vector depends upon the number of elements of the vector stored in each processor.

7.1.1 Diagonal preconditioner

If the number of elements per processor ($\frac{N}{p}$) is greater than or equal to $\sqrt{N}$ (*i.e.*, $p \leq \sqrt{N}$), then the required communication can be accomplished by each pair of neighboring processors exchanging $\sqrt{N}$ vector elements at the partition boundaries in time $2(t_s + t_w\sqrt{N})$. Thus the total time spent in communication by each processor is equal to $2(t_s + t_w\sqrt{N}) + 2(t_s + t_w) \approx 2(2t_s + t_w\sqrt{N})$. The following equation gives the expression for $T_o^{Matrix-Vector}$ in this case.

$$T_o^{Matrix-Vector} = 2(2t_s + t_w\sqrt{N}) \times p \qquad (p \leq \sqrt{N}) \tag{24}$$

If the number of elements per processor is less than $\sqrt{N}$ (*i.e.*, $p > \sqrt{N}$), then each processor has to exchange all its $\frac{N}{p}$ elements with processors located at a distance of $\frac{p}{\sqrt{N}}$ from it on either side; *i.e.*, processor P_i has to communicate with processors $P_{i+\frac{p}{\sqrt{N}}}$ and $P_{i-\frac{p}{\sqrt{N}}}$, as long as $0 \leq i \pm \frac{p}{\sqrt{N}} < p$. Under the assumption that communication between any two processors on the CM-5 has the same cost, the total communication time per processor will be $2(t_s + t_w) + 2(t_s + t_w\frac{N}{p}) \approx 2(2t_s + t_w\frac{N}{p})$. The following equation, therefore, gives the expression for the total overhead due to matrix-vector multiplication:

$$T_o^{Matrix-Vector} = 2(2t_s + t_w\frac{N}{p}) \times p \qquad (p > \sqrt{N}) \tag{25}$$

7.1.2 IC preconditioner

Recall from Section 6.1, solving **M z = r** for the case of IC preconditioner can be viewed as two matrix-vector multiplication operations. Each of these matrices have $\frac{(k+1)(k+2)}{2}$ diagonals distributed in $k+1$ clusters with a distance of $\sqrt{N}$ between the starting point of each cluster. The first cluster has $k+1$ diagonals, and then the number of diagonals in each cluster decreases by one. It can be shown that the following equations[2] give the total overheads involved in this operation on the CM-5.

[2]For the sake of simplicity, we are skipping the expressions for the cases when $\frac{\sqrt{N}}{k} < p \leq \sqrt{N}$.

$$T_o^{PC-solve} = 2(2t_s + t_w k\sqrt{N}) \times p \qquad (p \leq \frac{\sqrt{N}}{k}) \tag{26}$$

$$T_o^{PC-solve} = 2(2t_s + t_w \frac{N}{p})k \times p \qquad (p > \sqrt{N}) \tag{27}$$

7.2 Isoefficiency Analysis

7.2.1 Diagonal preconditioner

An analysis similar to that in Section 6.3 yields the following isoefficiency term due to t_s (from Equations (16) and (25)):

$$N \propto p \frac{4et_s}{c_1 + 5c_2} \tag{28}$$

When $p > \sqrt{N}$, the term due to t_w in T_o is $2t_w N$ (see equation (10)) which is balanced with $N(c_1 + 5c_2)$ for any number of processors as both the terms are $O(N)$. However, when $p \leq \sqrt{N}$, t_w does give rise to an isoefficiency term that is determined as follows -

$$N(c_1 + 5c_2) \propto 2et_w p\sqrt{N}$$

$$=> \quad \sqrt{N} \propto p \frac{2et_w}{c_1 + 5c_2}$$

$$=> \quad N \propto p^2 (\frac{2et_w}{c_1 + 5c_2})^2 \tag{29}$$

Thus, when $p > \sqrt{N}$, Equation (28) gives the overall isoefficiency function for the CG algorithm with a diagonal preconditioner on the CM-5. However, if $p \leq \sqrt{N}$, Equation (29) determines the overall isoefficiency function as it indicates a faster growth of N with respect to p.

Equation (28) indicates that on the CM-5, N has to increase only linearly with p to maintain a fixed efficiency. This suggests that the parallel system is highly scalable. For a given N and $p > \sqrt{N}$, the efficiency obtained will be equal to $\frac{1}{1 + \frac{2t_s p}{(c_1+5c_2)N} + \frac{t_w}{c_1+5c_2}}$. Note that there is an upper-bound on the achievable efficiency which can not exceed $\frac{1}{1+\frac{t_w}{c_1+5c_2}}$. Thus only efficiencies lower than this upper limit can be maintained by increasing N in proportion to p. If the computation speed of the processors of the parallel computer is much higher compared to the communication bandwidth of the interconnection network (*i.e.*, t_w is much higher compared to (c_1+5c_2)), this upper threshold on the achievable efficiency could be quite low. Efficiencies higher than this threshold can be attained only if $p \leq \sqrt{N}$, in which case, as indicated by Equation (29), N has to increase in proportion to p^2 in order to maintain a fixed efficiency with increasing p. In contrast, in case of the mapping of Section 6.1, there is no upper-bound on the achievable efficiency and the isoefficiency function is always linear for a CM-5 type parallel machine.

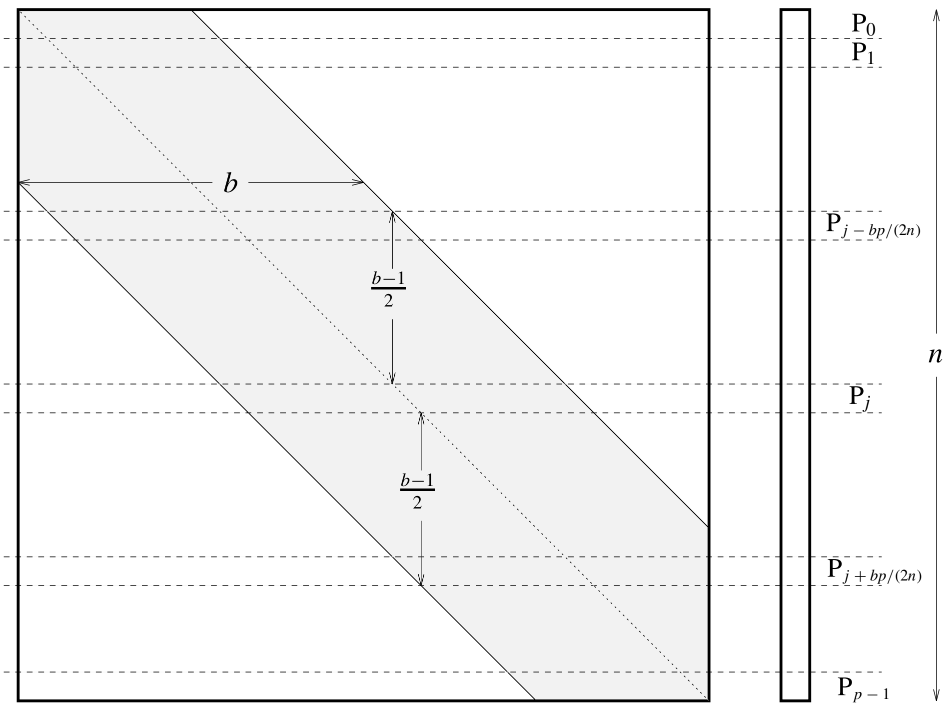


Figure 7: *Partition of a banded sparse matrix and a vector among the processors.*

7.2.2 IC preconditioner

For this case, W is given by Equation (3). This equation is similar to Equation (16) except that the constant is different. From the expressions for T_o for this case, it can be seen that even the terms in T_o for the IC preconditioner differ, if at all, from those in the T_o for the diagonal preconditioner by a constant factor of $(k+1)$. Therefore the isoefficiency functions for the IC preconditioner will be similar to those for the diagonal preconditioner, but the constants associated with the terms will be different and will depend on chosen value of k.

The upper-bound on the achievable efficiency on a fully connected network when $p > \sqrt{N}$ is now $\frac{1}{1+\frac{(k+1)t_w}{\eta(k)}}$ instead of $\frac{1}{1+\frac{t_w}{c_1+5c_2}}$, as in the case of the diagonal preconditioner. The typical values of k vary from 2 to 4. It is observed that for typical values of c_1 and c_2 for a machine like the CM-5, $\frac{1}{c_1+5c_2} < \frac{k+1}{\eta(k)}$ for $k = 2$ and $k = 3$, and $\frac{1}{c_1+5c_2} \approx \frac{k+1}{\eta(k)}$ for $k = 4$. As a result, it is possible to obtain higher efficiencies using a diagonal preconditioner than the IC preconditioner, and in case of the IC preconditioner, efficiencies decrease as k increases. This trend is totally opposite to that for the data mapping scheme of Section 6.1 in which the IC preconditioner yields better efficiencies than the diagonal preconditioner and the efficiencies increase with k.

8 Scalability analysis: unstructured sparse matrices

In this section, we consider a more general form of sparse matrices, in which the non-zeros are distributed randomly and do not form a regular pattern that can be utilized effectively. Such matrices occur in some applications, notably in linear programming problems. Often, such systems are encountered where the non-zero elements of matrix **A** occur only within a band around the principal diagonal. Even if the non-zero elements are scattered throughout the matrix, it is often possible to restrict them to a band through certain re-ordering techniques [4, 5]. Such a system is shown in Figure 7 in which the non-zero elements of the sparse matrix are randomly distributed within a band along the principal diagonal. Let the width of the band of the $N \times N$ matrix be given by b, and $b = \beta N^y$, and $0 \leq y \leq 1$. Suitable values of the constants β and y can be selected to represent the kind of systems being solved. If $\beta = 1$ and $y = 1$, we have the case of a totally unstructured sparse matrix.

The matrix **A** is stored in the *Ellpack-Itpack* format [24]. In this storage scheme, the non-zero elements of the matrix are stored in an $N \times m$ array while another $N \times m$ integer array stores the column numbers of the matrix elements. It can be shown that *co-ordinate* and the *compressed sparse column* storage formats incur much higher communication overheads, thereby leading to unscalable parallel formulations. Two other storage schemes, namely *jagged diagonals* [24] and *compressed sparse rows* involve communication overheads similar to the *Ellpack-Itpack* scheme, but the latter is the easiest to work with when the number of non-zero elements is almost the same in each row of the sparse matrix. The matrix **A** and the vector **b** are partitioned among p processors as shown in Figure 7. The rows and the columns of the matrix and the elements of the vector are numbered from 0 to $N-1$. Processor P_i stores $\frac{N}{p}i$ to $\frac{N}{p}(i+1)-1$ rows of matrix **A** and elements with indices from $\frac{N}{p}i$ to $\frac{N}{p}(i+1)-1$ of vector **b**. The preconditioner and all the other vectors used in the computation are partitioned similarly.

We will study the application of only the diagonal preconditioner in this case; hence the serial execution time is given by Equation (2). Often the number of non-zero elements per row of the matrix **A** is not constant but increases with N. Let $m = \alpha N^x$, where the constants α and x ($0 \leq x \leq 1$) can be chosen to describe the kind of systems being solved. A more general expression for W, therefore, would be as follows:

$$W = c_1 N + c_2 \alpha N^{1+x} \tag{30}$$

8.1 Communication overheads

For the diagonal preconditioner, $T_o^{PC-solve} = 0$ as discussed in Section 6. It can be shown that $T_o^{Matrix-Vector}$ dominates $T_o^{Inner-Prod}$ for most practical cases. Therefore, $T_o^{Matrix-Vector}$ can be considered to represent the overall overhead T_o for the case of unstructured sparse matrices.

If the distribution of non-zero elements in the matrix is unstructured, each row of the matrix could contain elements belonging to any column. Thus, for matrix-vector multiplication, any processor could need a vector element that belongs to any other processor. As a result, each processor has to send its portion of the vector of size $\frac{N}{p}$ to every other processor. If the bandwidth of the matrix **A** is b, then the ith row of the matrix can contain elements belonging to columns $i - \frac{b}{2}$ to $i + \frac{b}{2}$. Since a processor contains $\frac{N}{p}$ rows of the matrix and $\frac{N}{p}$ elements of each vector, it will need the elements of the vector that are distributed among the processors that lie within a distance of $\frac{bp}{2N}$ on its either side; *i.e.*, processor P_i needs to communicate with all the processors P_j such that $i - \frac{bp}{2N} \leq j < i$ and $i < j \leq i + \frac{bp}{2N}$. Thus the total number of communication steps in which each processor can acquire all the data it needs to perform matrix-vector multiplication will be $\frac{bp}{N}$. As a result, the following expression gives the value of T_o for $b = \beta N^y$.

$$T_o = (t_s \beta p N^{y-1} + t_w \beta N^y) \times p \tag{31}$$

It should be noted that this overhead is the same for all architectures under consideration in this paper from a linear array to a fully connected network.

8.2 Isoefficiency analysis

The size W of the problem at hand is given by Equation (30), which may be greater than $O(N)$ for $x > 0$. Strictly speaking, the isoefficiency function is defined as the rate at which the problem size needs to grow as a function of p to maintain a fixed efficiency. However, to simplify the interpretation of the results, in this section we will measure the scalability in terms of the rate at which N (the size of the linear system of equations to be solved) needs to grow with p rather than rate at which the quantity $c_1 N + c_2 \alpha N^{1+x}$ should grow with respect to p.

According to Equation (1), the following condition must be satisfied in order to maintain a fixed efficiency E:

$$c_1 N + c_2 \alpha N^{1+x} = \frac{E}{1-E}(t_s \beta p^2 N^{y-1} + t_w \beta p N^y)$$

$$=> \quad c_2 \alpha N^{x+y} + c_1 N^y - \frac{E}{1-E} t_w \beta p N^{2y-1} - \frac{E}{1-E} t_s \beta p^2 N^{2y-2} = 0 \tag{32}$$

$$=> \quad N = f_E(p, x, y, \alpha, \beta, t_w, t_s, c_1, c_2) \tag{33}$$

From Equation (32), it is not possible to compute the isoefficiency function f_E in a closed form for general x and y. However, Equation (32) can be solved for N for specific values of x and y. We, therefore, compute the isoefficiency functions for a few interesting cases that result from assigning some typical values to the parameters x and y. Table 1 gives a summary of the scalability analysis for these cases. In order to maintain the efficiency at some fixed value E, the size N of the system has to grow according to the expression in the third column of the Table 1, where $e = \frac{E}{1-E}$.

The above analysis shows that a PCG iteration is unscalable for a totally unstructured sparse matrix of coefficients with a constant number of non-zero elements in each row. However, it can be rendered scalable by either increasing the number of non-zero elements per row as a function of the matrix size, or by restricting the non-zero elements into a band of width less than $O(N)$ for an $N \times N$ matrix using some re-ordering scheme. Various techniques for re-ordering sparse systems to yield banded or partially banded sparse matrices are available [4, 5]. These techniques may vary in their complexity and effectiveness. By using Equation (32) the benefit of a certain degree of re-ordering can be quantitatively determined in terms of improvement in scalability.

In this section we have presented the results for the diagonal preconditioner only. However, the use of the IC preconditioner does not have a significant impact on the overall scalability of the algorithm. If a method similar to the one discussed in Section 6.1 is used, solving the system $\mathbf{M}\,\mathbf{z} = \mathbf{r}$ for the preconditioner $\mathbf{M}$ will be equivalent to two matrix-vector multiplications, where the band of the matrix is k times wider than that of matrix $\mathbf{A}$ if k powers of $\mathbf{L}$ and $\tilde{\mathbf{L}}^T$ are computed (see Section 6.1). Thus, the expressions for both W and T_o will be very similar to those in case of the diagonal preconditioner, except that the constants will be different.

	Parameter values	Isoefficiency function	Interpretation in terms of scalability
1.	$x = 0$, $y = 1$	Does not exist	Unscalable
2.	$x = 0$, $y = \frac{1}{2}$	$N \propto p^2 (\frac{e\beta t_w}{c_1 + c_2\alpha})^2$	O(p^2) scalability (moderately scalable)
3.	$x = 1$, $y = 1$	$N \propto p^{\frac{et_w\beta + \sqrt{e^2 t_w^2 \beta^2 + 4c_2\alpha\beta e t_s}}{2c_2\alpha}}$	Linearly scalable with a high constant
4.	$x = 0$, $y = 0$	$N \propto p \frac{et_w\beta}{2(c_1+c_2\alpha)} \left(1 + \sqrt{1 + \frac{4t_s(c_1+c_2\alpha)}{et_w^2\beta}}\right)$	Linear scalability (highly scalable)
5.	$x = \frac{1}{2}$, $y = \frac{1}{2}$	$N \propto p \frac{et_w\beta}{2c_2\alpha} \left(1 + \sqrt{1 + \frac{4t_s c_2\alpha}{et_w^2\beta}}\right)$	Linear scalability (highly scalable)

Table 1: *Scalability of a PCG iteration with unstructured sparse matrices of coefficients. The average number of entries in each row of the $N \times N$ matrix is αN^x and these entries are located within a band of width βN^y along the principal diagonal.*

9 Experimental results and their interpretations

The analytical results derived in the earlier sections of the paper were verified through experimental data obtained by implementing the PCG algorithm on the CM-5. Both block-tridiagonal and unstructured sparse symmetric positive definite matrices were generated randomly and used as test cases. The degree of diagonal dominance of the matrices was controlled such that the algorithm performed enough number of iterations to ensure the accuracy of our timing results. Often, slight variation in the number of iterations was observed as different number of processors were used. In the parallel formulation of the PCG algorithm, both matrix-vector multiplication and the inner product computation involve communication of certain terms among the processors which are added up to yield a certain quantity. For different values of p, the order in which these terms are added could be different. As a result, due to limited precision of the data, the resultant sum may have a slightly different value for different values of p. Therefore, the criterion for convergence might not be satisfied after exactly the same number of iterations for all p. In our experiments, we normalized the parallel execution time with respect to the number of iterations in the serial case in order to compute the speedups and efficiencies accurately. Sparse linear systems with matrix dimension varying from 400 to over 64,000 were solved using up to 512 processors. For block-tridiagonal matrices, we implemented the PCG algorithms using both the diagonal and the IC preconditioners. For unstructured sparse matrices, only the diagonal preconditioner was used. The configuration of the CM-5 used in our experiments had only a SPARC processor on each node which delivered approximately 1 MFLOPS (double-precision) in our implementation. The message startup time for the program was observed to be about 150 microseconds and

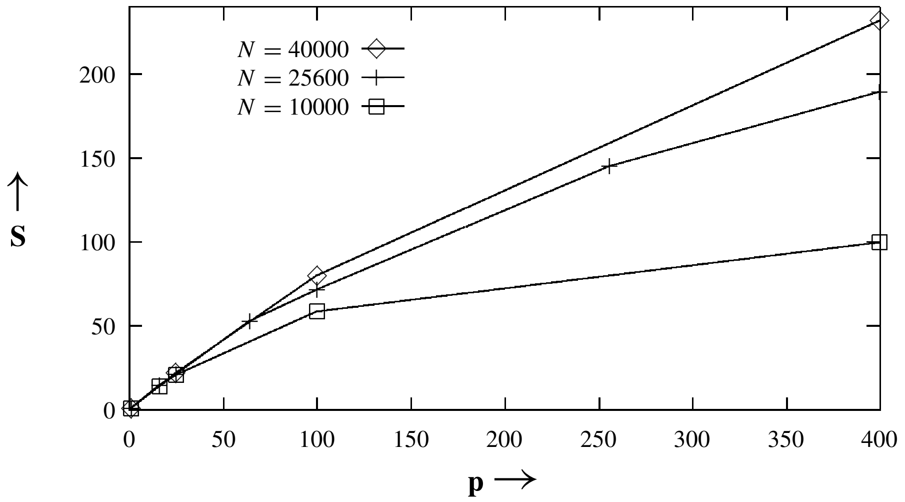


Figure 8: *Speedup curves for block-tridiagonal matrices with diagonal preconditioner.*

the per-word transfer time for 8 byte words was observed to be about 3 microseconds[3].

Figure 8 shows experimental speedup curves for solving problems of different sizes using the diagonal preconditioner on block-tridiagonal matrices. As expected, for a given number of processors, the speedup increases with increasing problem size. Also, for a given problem size, the speedup does not continue to increase with the number of processors, but tends to saturate.

Recall that the use of the IC preconditioner involves substantially more computation per iteration of the PCG algorithm over a simple diagonal preconditioner, but it also reduces the number of iterations significantly. As the approximation of the inverse of the preconditioner matrix is made more accurate (*i.e.*, k is increased, as discussed in Section 5.2), the computation per iteration continues to increase, while the number of iterations decreases. The overall performance is governed by the amount of increase in the computation per iteration and the reduction in the number of iterations. As discussed in Section 6.3.2, for the same number of processors, an implementation with the IC preconditioner will obtain a higher efficiency[4] (and hence speedup) than one with the diagonal preconditioner. Even in case of the IC preconditioner, speedups will be higher for higher values of k. Figure 9 shows the efficiency curves for the diagonal preconditioner and the IC preconditioner for $k = 2$, 3 and 4 for a given problem size. From this figure it is clear that one of the IC preconditioning schemes may yield a better overall execution time in a parallel implementation due to a better speedup than the diagonal preconditioning scheme, even if the latter is faster in a serial implementation. For instance, assume that on a serial machine the PCG algorithm runs 1.2 times faster with a diagonal preconditioner than with

[3]These values do not necessarily reflect the communication speed of the hardware but the overheads observed for our implementation. For instance copying the data in and out of the buffers in the program contributes to the per-word overhead. Moreover, the machine used in the experiments was still in beta testing phase, hence the performance obtained in our experiments may not be indicative of the achievable performance of the machine.

[4]The efficiency of a parallel formulation is computed with respect to an identical algorithm running on a single processor.

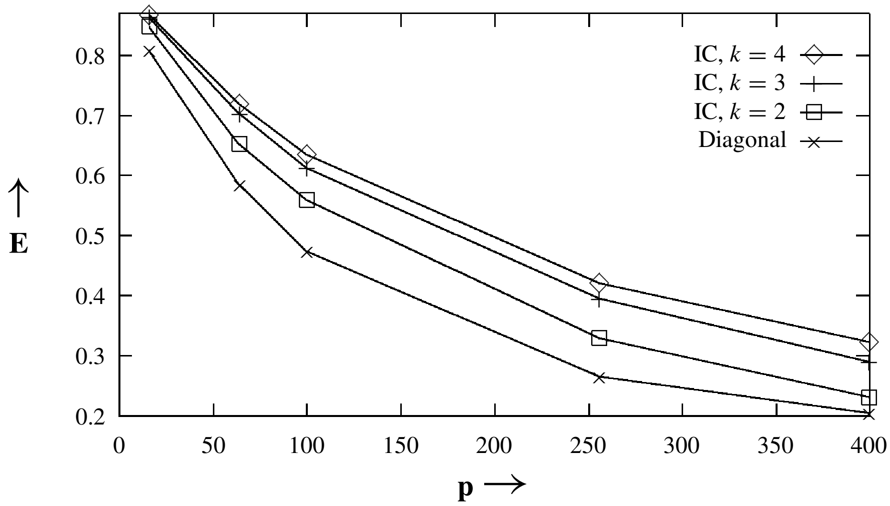


Figure 9: *Efficiency curves for the diagonal and the IC preconditioner with a* 1600×1600 *matrix of coefficients.*

the IC preconditioner for a certain system of equations. As shown in Figure 9, with 256 processors on the CM-5, the IC preconditioner implementation with $k = 3$ executes with an efficiency of about 0.4 for an 80×80 finite difference grid, while the diagonal preconditioner implementation attains an efficiency of only about 0.26 on the same system. Therefore, unlike on a serial computer, on a 256 processor CM-5 the IC preconditioner implementation for this system with $k = 3$ will be faster by a factor of $\frac{0.4}{0.26} \times \frac{1.0}{1.2} \approx 1.3$ than a diagonal preconditioner implementation.

In Figure 10, experimental isoefficiency curves are plotted for the two preconditioners by experimentally determining the efficiencies for different values of N and p and then selecting and plotting the (N, P) pairs with nearly the same efficiencies. As predicted by Equations (17), (18) and (21), the N versus p curves for a fixed efficiency are straight lines. These equations, as well as Figure 10, suggest that this is a highly scalable parallel system and requires only a linear growth of problem size with respect to p to maintain a certain efficiency. This figure also shows that the IC preconditioner needs a smaller problem size than the diagonal preconditioner to achieve the same efficiency. For the same problem size, the IC preconditioner can use more processors at the same efficiency, thereby delivering higher speedups than the diagonal preconditioner.

A comparison of the overhead functions for the data partitioning schemes described in Sections 6 and 7 clearly shows that the overhead due to t_w is higher in the latter scheme. This extra overhead translates into reduced efficiencies which can be seen from Figure 11. This figure shows the efficiency obtained experimentally on the CM-5 as a function of problem size for the two mapping schemes for 256 processors. As expected, the block mapping described in Section 6 performs better than the simple rowwise partitioning described in Section 7.

The difference between the efficiencies obtained for the two schemes will become even more

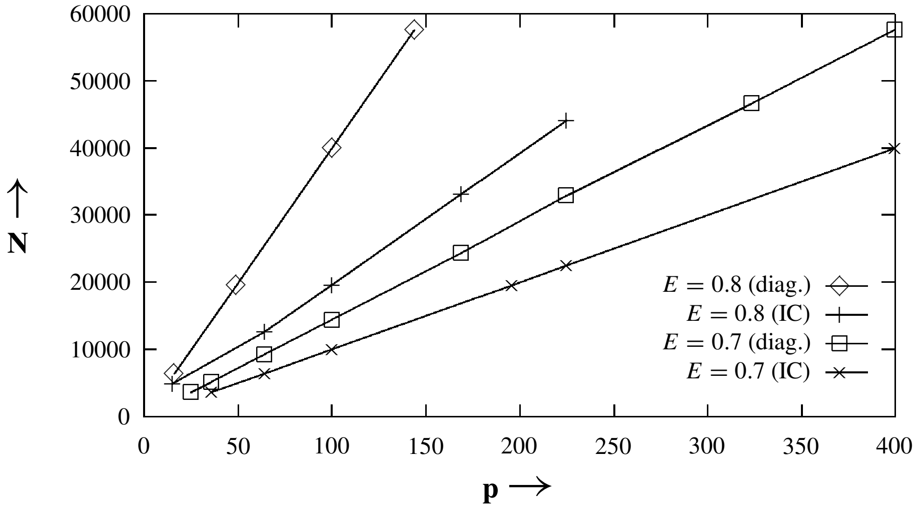


Figure 10: *Isoefficiency curves for the two preconditioning schemes.*

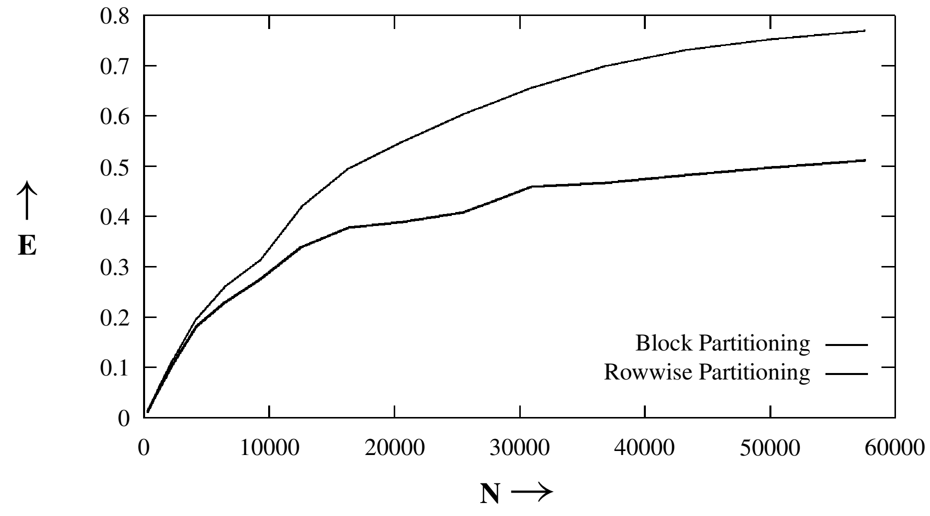


Figure 11: *Efficiency curves for the two partitioning schemes on the CM-5 with 256 processors.*

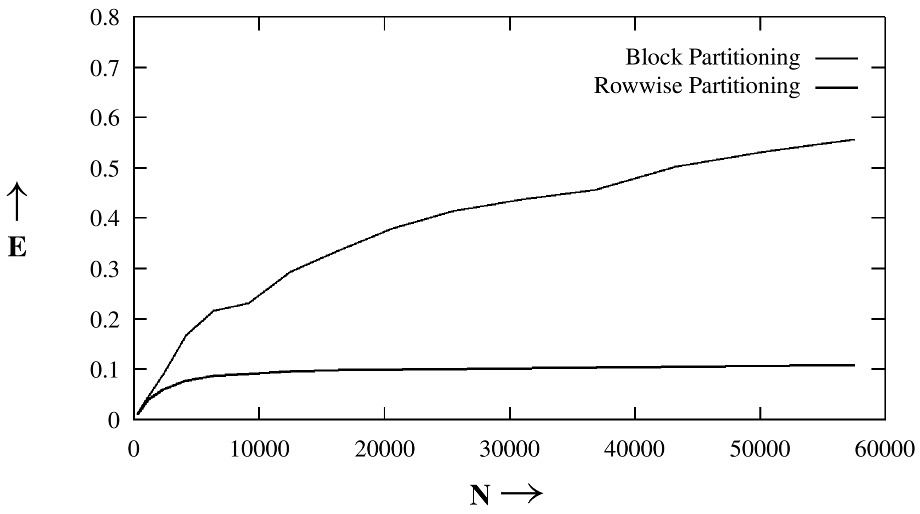


Figure 12: *Efficiency curves for the two mapping schemes on a ten times faster CM-5 with 256 processors.*

apparent if the algorithm is executed on a machine with the same bandwidth of communication channels but much faster CPU's. We simulated the effect of 10 times faster CPU's by artificially lowering the communication channel bandwidth by a factor of 10 (this was done by sending 10 times bigger messages) and dividing the total execution time by 10. Figure 12 shows the experimental efficiency curves for the two schemes for this simulated machine. It can be seen from this figure that the rowwise partitioning of Section 7 is affected much more seriously than block partitioning described in Section 6. Also, as discussed in Section 7.2.1, the efficiency in case of the rowwise scheme suffers from a saturation and increasing the problem size beyond a point does not seem to improve the achieved efficiency.

Figure 13 shows plots of efficiency versus problem size for three different values of p for a totally unstructured sparse matrix with a fixed number of non-zero elements in each row. As discussed in Section 8.2, this kind of a matrix leads to an unscalable parallel formulation of the CG algorithm. This fact is clearly reflected in Figure 13. Not only does the efficiency drop rapidly as the number of processors are increased, but also an increase in problem size does not suffice to counter this drop in efficiency. For instance, using 40 processors, it does not seem possible to attain the efficiency of 16 processors, no matter how big a problem is being solved.

Figures 14 and 15 show how the parallel CG algorithm for unstructured sparse matrices can be made scalable by either confining the non-zero elements within a band of width $< O(N)$, or by increasing the number of non-zero elements in each row as a function of N. Figure 14 shows the experimental isoefficiency curves for a banded unstructured sparse matrix with a bandwidth of $2\sqrt{N}$ and 6 non-zero elements per row. The curves were drawn by picking up (N, p) pairs that yielded nearly the same efficiency on the CM-5 implementation, and then plotting N with respect to p^2. As shown in Table 1, the isoefficiency function is $O(p^2)$ and a linear relation between N and p^2 in Figure

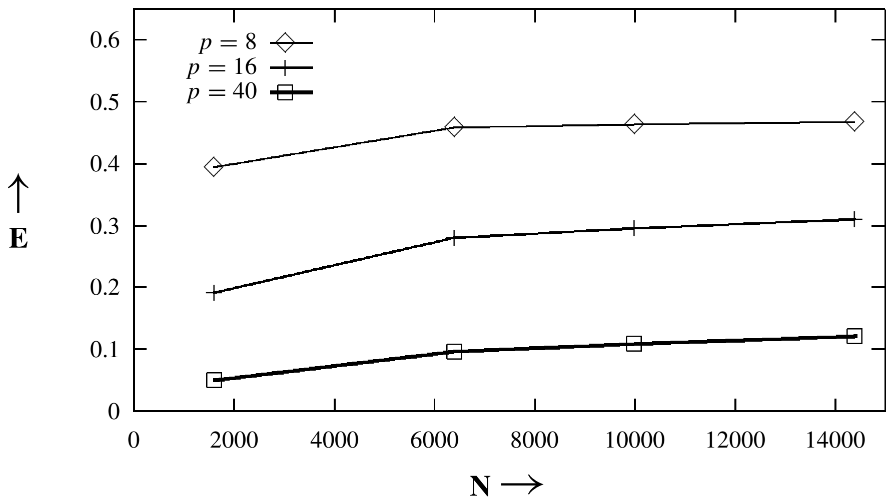


Figure 13: *Efficiency plots for unstructured sparse matrices with fixed number of non-zero elements per row.*

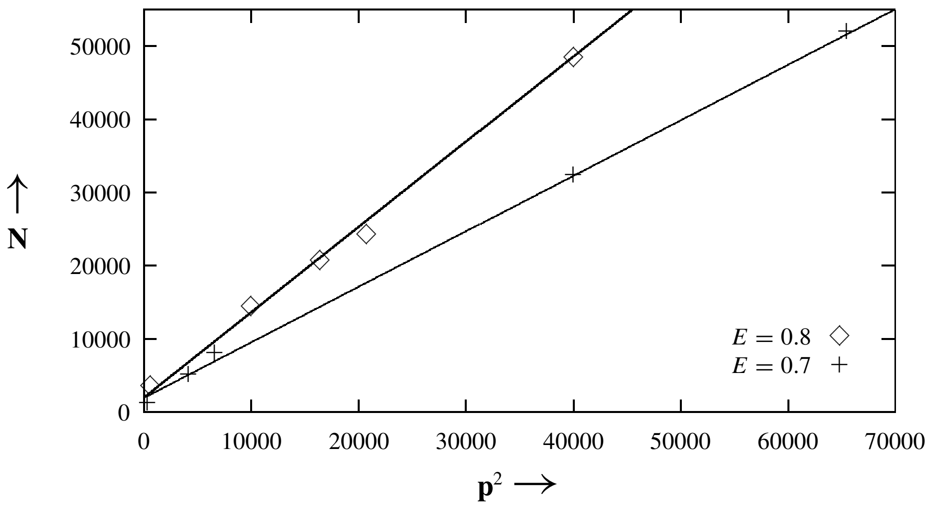


Figure 14: *Isoefficiency curves for banded unstructured sparse matrices with fixed number of non-zero elements per row.*

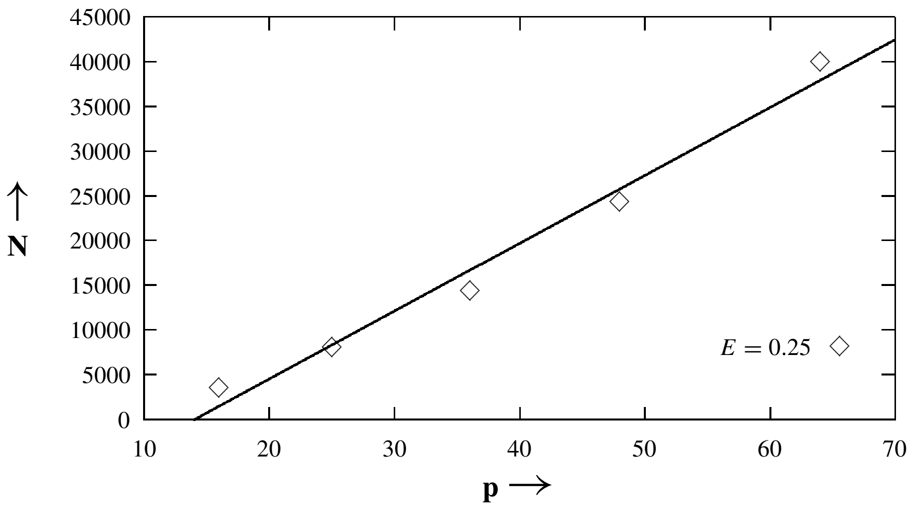


Figure 15: *An isoefficiency curve for unstructured sparse matrices with the number of non-zero elements per row increasing with the matrix size.*

14 confirms this. Figure 15 shows the isoefficiency curve for $E = 0.25$ for a totally unstructured $N \times N$ sparse matrix with $0.0015N$ non-zero elements in each row. As shown in Table 1, the isoefficiency function is linear in this case, although the constant associated with it is quite large because of the terms $2c_2\alpha$ in the denominator, α being 0.0015.

10 Summary of results and concluding remarks

In this paper we have studied the performance and scalability of an iteration of the Preconditioned Conjugate Gradient algorithm on a variety of parallel architectures.

It is shown that block-tridiagonal matrices resulting from the discretization of a 2-dimensional self-adjoint elliptic partial differential equation via finite differences lead to highly scalable parallel formulations of the CG method on a parallel machine like the CM-5 with an appropriate mapping of data onto the processors. On this architecture, speedups proportional to the number of processors can be achieved by increasing the problem size almost linearly with the number of processors. The reason is that on the CM-5, the control network practically eliminates the communication overhead in computing inner-products of vectors, whereas on more conventional parallel machines with significant message startup times, it turns out to be the costliest operation in terms of overheads and affects the overall scalability. The isoefficiency function for a PCG iteration with a block-tridiagonal matrix is $O(p)$ on the CM-5, $O(p \log p)$ on a hypercube, and $O(p\sqrt{p})$ on a mesh. In terms of scalability, if the number of processors is increased from 100 to 1000, the problem size will have to increased 32 times on a mesh, 15 times on a hypercube, and only 10 times on the CM-5 to obtain ten times higher speedups. Also, the effect of message startup time t_s on the speedup is much more significant on a typical hypercube or a mesh than on the CM-5. However, for some iterative algorithms like the Jacobi method, linear scalability can be obtained even on these architectures by performing the convergence check at a suitably reduced frequency.

The current processors used on the CM-5 are SPARC processors that deliver about 1 MFLOPS (double-precision) each in our implementation. These processors are soon expected to be augmented with vector units that may increase the net performance by a factor of 10 to 30. This will reduce the ratio of communication speed to computation speed by a factor a 10 to 30. The message startup time using the current unoptimized communication routines is about 150 microseconds. A 5 to 10 times reduction in t_s in the near future is quite conceivable due to software improvement. After these improvements, the CM-5 will be a massively parallel machine with hardware parameters close to those of the hypothetical machine described in Section 6.4. For these values of the machine parameters, a high inter-processor communication bandwidth will be crucial for obtaining good efficiencies. The reason is that although the isoefficiency function of an iteration of the PCG algorithm is linear with respect to p, it is also proportional to the square of $\frac{E}{1-E}t_w$. Moreover, at such high processor speeds, the simple mapping scheme described in Section 7 will be highly impractical.

We have shown that in case of the block-tridiagonal matrices, the truncated Incomplete Cholesky preconditioner can achieve higher efficiencies than a simple diagonal preconditioner if the data mapping scheme of Section 6.1 is used. The use of the IC preconditioner usually significantly cuts down the number of iterations required for convergence. However, it involves solving a linear system of equations of the form $\mathbf{M}\,\mathbf{z} = \mathbf{r}$ in each iteration. This is a computationally costly operation and may offset the advantage gained by fewer iterations. Even if this is the case for the serial algorithm, in a parallel implementation the IC preconditioner may outperform the diagonal preconditioner as the number of processors are increased. This is because a parallel implementation with IC preconditioner executes at a higher efficiency than one with a diagonal preconditioner.

If the matrix of coefficients of the linear system of equations to be solved is a random unstructured sparse matrix (such matrices often occur in linear programming problems) with a constant number of non-zero elements in each row, a parallel formulation of the PCG method will be unscalable on any practical massage passing architecture unless some ordering is applied to the sparse matrix. The efficiency of parallel PCG with an unordered sparse matrix will always drop as the number of processors is increased and no increase in the problem size is sufficient to counter this drop in efficiency. The system can be rendered scalable if either the non-zero elements of the $N \times N$ matrix of coefficients can be confined in a band whose width is less than $O(N)$, or the number of non-zero elements per row increases with N, where N is the number of simultaneous equations to be solved. The scalability increases as the number of non-zero elements per row in the matrix of coefficients is increased and/or the width of the band containing these elements is reduced. Both the number of non-zero elements per row and the width of the band containing these elements depend on the characteristics of the system of equations to be solved. In particular, the non-zero elements can be organized within a band by using some re-ordering techniques [4, 5]. Such restructuring techniques can improve the efficiency (for a given number of processors, problem size, machine architecture, etc.) as well as the asymptotic scalability of the PCG algorithm.

References

[1] Edward Anderson. Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations. Technical Report 805, Center for Supercomputing Research and Development, University of Illinois, Urbana, IL, 1988.

[2] Cevdet Aykanat, Fusun Ozguner, Fikret Ercal, and Ponnuswamy Sadayappan. Iterative algorithms for solution of large sparse systems of linear equations on hypercubes. *IEEE Transactions on Computers*, 37(12):1554–1567, 1988.

[3] D. L. Eager, J. Zahorjan, and E. D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[4] A. George and J. W.-H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[5] N. E. Gibbs, W. G. Poole, and P. K. Stockmeyer. A comparison of several bandwidth and profile reduction algorithms. *ACM Transactions on Mathematical Software*, 2:322–330, 1976.

[6] Gene H. Golub and Charles Van Loan. *Matrix Computations: Second Edition*. The Johns Hopkins University Press, Baltimore, MD, 1989.

[7] 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.

[8] Anshul Gupta and Vipin Kumar. A scalable parallel algorithm for sparse matrix factorization. Technical Report 94-19, Department of Computer Science, University of Minnesota, Minneapolis, MN, 1994. A short version appears in *Supercomputing '94 Proceedings*. TR available in *users/kumar* at anonymous FTP site *ftp.cs.umn.edu*.

[9] 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.

[10] John L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[11] 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.

[12] S. W. Hammond and Robert Schreiber. Efficient ICCG on a shared-memory multiprocessor. *International Journal of High Speed Computing*, 4(1):1–22, March 1992.

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

[14] C. Kamath and A. H. Sameh. The preconditioned conjugate gradient algorithm on a multiprocessor. In R. Vichnevetsky and R. S. Stepleman, editors, *Advances in Computer Methods for Partial Differential Equations*. IMACS, 1984.

[15] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Communications of the ACM*, 33(5):539–543, 1990.

[16] S. K. Kim and A. T. Chronopoulos. A class of Lanczos-like algorithms implemented on parallel computers. *Parallel Computing*, 17:763–777, 1991.

[17] Kouichi Kimura and Ichiyoshi Nobuyuki. Probabilistic analysis of the efficiency of the dynamic load distribution. In *The Sixth Distributed Memory Computing Conference Proceedings*, 1991.

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

[19] 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.

[20] 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.

[21] Rami Melhem. Toward efficient implementation of preconditioned conjugate gradient methods on vector supercomputers. *International Journal of Supercomputer Applications*, I(1):70–97, 1987.

[22] Daniel Nussbaum and Anant Agarwal. Scalability of parallel machines. *Communications of the ACM*, 34(3):57–61, 1991.

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

[24] Youcef Saad. SPARSKIT: A basic tool kit for sparse matrix computations. Technical Report 90-20, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffet Field, CA, 1990.

[25] Youcef Saad and M. H. Schultz. Parallel implementations of preconditioned conjugate gradient methods. Technical Report YALEU/DCS/RR-425, Yale University, Department of Computer Science, New Haven, CT, 1985.

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

[27] Zhimin Tang and Guo-Jie Li. Optimal granularity of grid iteration problems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I111–I118, 1990.

[28] Fredric A. Van-Catledge. Towards a general model for evaluating the relative performance of computer systems. *International Journal of Supercomputer Applications*, 3(2):100–108, 1989.

[29] Henk A. van der Vorst. A vectorizable variant of some ICCG methods. *SIAM Journal on Scientific and Statistical Computing*, III(3):350–356, 1982.

[30] Henk A. van der Vorst. Large tridiagonal and block tridiagonal linear systems on vector and parallel computers. *Parallel Computing*, 5:45–54, 1987.

[31] 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.

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

[33] J. R. Zorbas, D. J. Reble, and R. E. VanKooten. Measuring the scalability of parallel computer systems. In *Supercomputing '89 Proceedings*, pages 832–841, 1989.