UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

The Preconditioned Conjugate Gradient Method on the Connection Machine

Charles Tong

October 1988

(Revised February 1989)

CAM Report 88-33

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90024-1555

THE PRECONDITIONED CONJUGATE GRADIENT METHOD ON THE CONNECTION MACHINE [+]

CHARLES TONG
Computer Science Department, University of California, Los Angeles, Los Angeles, California 90024, United States of America

ABSTRACT

This paper presents the results of the Connection Machine implementation of a number of preconditioners for the preconditioned conjugate gradient method. The preconditioners implemented include those based on the incomplete LU factorization, the modified incomplete LU factorization, the symmetric successive overrelaxation, and others such as several polynomial preconditioners and the hierarchical basis preconditioner. Results based on numerical experiments show that both the degree of parallelism inherent in a preconditioner and its convergence rate improvement play important roles on the overall execution time performance on parallel computers. Factors that affect the performance of the preconditioners will also be discussed. We conclude that to search for the best preconditioner on a parallel machine, we have to consider the tradeoffs between fast convergence rate and high degree of parallelism as well as the architecture of the target parallel computer.

Key Words : Connection Machine (CM), preconditioned conjugate gradient (PCG) method, diagonal natural ordering, parallel red/black ordering, parallel computations.

1. Introduction

The conjugate gradient method, coupled with "good" preconditioning, has been known as an efficient technique for solving large sparse symmetric positive definite linear systems of equations such as those generated by the discretization of elliptic partial differential equations in two or three dimensions. In the past, many preconditioners have been proposed which helped to make the PCG method very competitive for computer implementation. Some of these preconditioners offer condition number improvement of an order of magnitude, so that the overall operation count to achieve convergence is greatly reduced. For example, the modified incomplete LU factorization with natural ordering was widely used on sequential computers. However, such preconditioners often

[+] This work was supported in part by the Department of Energy under contract DE-FG03-87ER25037, the National Science Foundation under contracts NSF-DMS87-14612 and BBS 87 14206, and the Army Research Office under contract DAAL03-88-K-0085.

have the property that they are very much sequential. For example, the maximum degree of parallelism of the MILU preconditioner is $O(\sqrt{N})$ where N is the number of unknowns. Thus, these sequential preconditioners are unable to exploit efficiently the computational resources offered by massively parallel computers such as the CM, as it will be shown later from the experimental results. Other preconditioners, such as the red/black ordering version of the MILU preconditioner, have a high degree of inherent parallelism. Nevertheless, experiments and analyses have shown that these parallel preconditioners often improve the condition number by a small constant, resulting in only small improvement in overall execution time performance. The conflict between fast convergence rate and high degree of parallelism in a preconditioner should prompt researchers to search for better preconditioners based not only on the criterion of best improvement on condition number or least operation counts, but also high degree of parallelism. (Ref. 1) has an excellent survey of the performance of many preconditioners on vector and parallel computers. The purpose of this paper is to address the implementation of preconditioners on one particular parallel single-instruction-multiple-data (SIMD) computer, namely, the Connection Machine.

This paper presents the results of implementation of a number of preconditioners on the CM. Among these preconditioners are : incomplete LU (ILU) factorization with natural ordering and red/black ordering, Modified incomplete LU (MILU) factorization with natural ordering and red/black ordering, symmetric successive overrelaxation (SSOR) with natural ordering and red/black ordering, several polynomial preconditioners and the hierarchical basis preconditioner. In section 2, the basic conjugate gradient method, different types of orderings, as well as the detailed formulation of the preconditioners are covered. In section 3, details of implementation are presented. In section 4, the iteration counts as well as CM execution times and MFLOPS counts achieved on the CM for different preconditioners will be presented and observations will be discussed.

2. The Preconditioned Conjugate Gradient Method

2.1 The Preconditioned Conjugate Gradient (PCG) Algorithm

The PCG algorithm for the solution of a large sparse symmetric positive definite linear system of equations

$$A\ u = b \qquad (2.1)$$

where A is an $N \times N$ symmetric positive definite matrix and u and b are $N \times 1$ vectors, is given as follow : [2,3]

```
$r = b - A\,u$                          ; initial residual
$p = 0$
Repeat
    $z = M^{-1}\,r$                     ; preconditioning
    $\beta = new <r, z> / old <r, z>$
    $p = z + \beta p$                   ; updating direction
    $\alpha = new <r, z> / <p, Ap>$
    $u = u + \alpha p$                  ; updating solution
    $r = r - \alpha Ap$                 ; updating the residual
until $<r, r> \le tolerance$
```

where $<\cdot, \cdot>$ denotes the usual Euclidean inner product, and r and p are $N \times 1$ residual and search direction vectors respectively.

The matrix M is called the preconditioning matrix and the speed with which the algorithm converges depends strongly on the choice of M. It is desirable to have M approximate A so that the condition number $\kappa(M^{-1}A)$ is smaller than that of A alone, that M retains the sparsity feature, and that the computational overhead to solve the system of equations

$$M\,z = r \tag{2.2}$$

is relatively small.

2.2 Model problems and orderings

Consider the Dirichlet boundary value problem

$$-(pu_x)_x - (qu_y)_y = r(x,y) \qquad on\ \Omega = (0,1)^2 \tag{2.3a}$$

$$u(x,y) = g(x,y) \qquad on\ \partial\Omega \tag{2.3b}$$

where $p(x,y)$, $q(x,y)$ and $r(x,y)$ are smooth functions and $p(x,y)$ and $q(x,y)$ are positive on Ω. The formulation in the rest of this section assumes $p(x,y) = q(x,y) = 1$ and $r(x,y) = g(x,y) = 0$ (Poisson equation with zero boundary condition). The 5-point finite difference approximation of the Poisson problem is

$$u_{j+1,k}+u_{j-1,k}+u_{j,k+1}+u_{j,k-1}-4u_{j,k} = b_{j,k}h^2 \quad j,k = 1,\cdot\cdot, n-1 \ \ where\ n = \frac{1}{h}, \tag{2.4}$$

where h is the grid spacing in both the x and y directions and $u_{j,k}$ is used to approximate the value of $u(jh,kh)$. In terms of the shift operators, the problem can be rewritten as

$$A_{j,k}\,u_{j,k} = -\frac{h^2 f_{j,k}}{4}, \quad A_{j,k} = 1 - \frac{1}{4}\,(E_x + E_x^{-1} + E_y + E_y^{-1}), \tag{2.5}$$

where E_x and E_y are shift operators along the x and y directions, such that

$$E_x \, u_{j,k} = u_{j+1,k} \;,\; E_x^{-1} \, u_{j,k} = u_{j-1,k} \;,\; E_y \, u_{j,k} = u_{j,k+1} \;,\; E_y^{-1} \, u_{j,k} = u_{j,k-1} \;.$$

Therefore, $A_{j,k}$ is a local operator at the grid point (jh,kh). A stencil representation of operator $A_{j,k}$ can be found in (Ref. 3,4). A collection of local operators $A_{j,k}$ at all grid points together with the way that the grid points are ordered form the coefficient matrix A.

The ordering of grid points on the 2-D grid determines the form of the coefficient matrix A and also that of the preconditioners. Using the natural ordering, grid points are ordered in row-wise (or column-wise) manner. And using the red/black ordering, grid points are first partitioned into red and black groups such that a grid point (j,k) is red if $j+k$ is even and black if it is odd. Then the grid points within red group are ordered, followed by the ordering of the grid points within the black group. In the context of parallel computation, the natural and red/black orderings evolve into their parallel versions, which are called diagonal and parallel red/black orderings respectively. These two (partial) orderings are defined as follow

Diagonal ordering

$$(j,k) < (m,n) \qquad \text{if } j+k < m+n,$$

Parallel Red/black ordering

$$(j,k) < (m,n) \qquad \text{if } (j,k) \text{ \it is red} \text{ and } (m,n) \text{ \it is black},$$

where the order of updates during preconditioning is determined by the inequality condition (e.g. in ascending or descending order). The stencil representations of these two orderings for the grid points on a uniform 6×6 square grid are illustrated in (Ref. 4). For the diagonal ordering, the grid points that have the same sum $j+k$ can be performed in parallel, and the same is true for the red/black ordering where the grid points are of the same color. This means that if this problem is solved on a parallel computer, then a sweep (or one iteration) using red/black ordering can at best be computed in constant time (independent of the number of grid points) while a sweep using diagonal ordering can at best be computed in $O(\sqrt{N})$ time, where N is the number of grid points. Here we can see that the parallel red/black ordering offers higher degree of parallelism ($O(N)$ operations can be performed in parallel) than the diagonal ordering (at most $O(\sqrt{N})$ operations can be performed in parallel). Nevertheless, the convergence rate improvement of the preconditioners using diagonal ordering are usually better than those using parallel red/black ordering.

The red/black ordering can be generalized to the multi-color ordering. The grid points can be colored using, for example, four different colors using the same idea as the red/black ordering with no grid points of the same color adjacent to each other. A large number of different types of ordering and their convergence rate performance can be found in (Ref. 5,6). Only red/black ordering will be implemented in this experiment.

2.3 Preconditioners

The preconditioners implemented in this experiment are described in the following sub-sections.

2.3.1 Incomplete LU (ILU) preconditioners[7,8,9]

2.3.1.1 ILU with diagonal ordering

The family of ILU preconditioners has many variations. One such ILU preconditioner can be defined as

$$M_I = L\,U \quad \textit{such that } M_I(i,j) = A\,(i,j) \text{ for } \textit{all } A\,(i,j) \neq 0 \tag{2.6}$$

where L and U are lower and upper triangular matrices respectively and L has unit diagonal. The coefficients of the L and U matrix can be calculated by performing an LU factorization on M_I. On some computers when minimizing the use of memory space is an important issue, a different formulation can be used which is called the incomplete Cholesky factorization and the corresponding conjugate gradient method is called the ICCG method. The preconditioning step of incomplete Cholesky factorization, however, requires more arithmetic operations than the ILU, and since memory space is not a major problem in this experiment, the ILU preconditioner is to be preferred. The coefficients of the L and U matrices for problems with periodic boundary condition can be calculated in a much simpler way as described in the following (Experimental results shows that using this formulation for the Poisson problem with Dirichlet boundary condition gives more or less the same iteration counts.)

$$L_{j,k} = \frac{1}{4}\,(\,a - E_x^{-1} - E_y^{-1}\,)\,, \quad U_{j,k} = 1 - \frac{1}{a}\,E_x - \frac{1}{a}\,E_y, \tag{2.7a}$$

where a is a constant to be determined. The product of $L_{j,k}$ and $U_{j,k}$ is thus

$$M_{I\,(j,k)} = \frac{1}{4}\,[a + \frac{2}{a} - (E_x + E_y + E_x^{-1} + E_y^{-1}) + \frac{1}{a}(E_xE_y^{-1} + E_x^{-1}E_y)] \tag{2.7b}$$

By matching the entries of M and A according to the requirement formulated above, the following condition is obtained

$$a + \frac{2}{a} = 4\,. \tag{2.7c}$$

The solutions of this equation are $a = 2 \pm \sqrt{2}$. We use $a = 2 + \sqrt{2}$ since the coefficients of the corresponding difference operator $R_I = M_I - A$ have smaller absolute values. Thus, the diagonally-ordered ILU preconditioning consists of a forward solve (L^{-1}) followed by a backward solve (U^{-1}). Due to the sequential nature of the diagonally-ordered ILU preconditioning, these forward and backward solves can at best be performed in $O(\sqrt{N}\,)$

time on parallel computers.

2.3.1.2 ILU with parallel red/black ordering

The local operators $L_{j,k}$ and $U_{j,k}$ for the parallel red/black ordering are

$$L_{j,k} = \begin{cases} 1\,, & (j,k)\ red \\ 1 - \frac{1}{4}\,(E_x + E_x^{-1} + E_y + E_y^{-1}\,) & ,\ (j,k)\ black \end{cases} \tag{2.8a}$$

$$U_{j,k} = \begin{cases} 1 - \frac{1}{4}\,(E_x + E_x^{-1} + E_y + E_y^{-1}\,)\,, & (j,k)\ red \\ \frac{3}{4}\,, & (j,k)\ black \end{cases} \tag{2.8b}$$

The parallel red/black ILU local operator can be formed as follows

$$M_{Irb\,(j,k)} =$$

$$\begin{cases} 1 - \frac{1}{4}\,(\,E_x + E_x^{-1} + E_y + E_y^{-1}\,)\,, & (j,k)\ \ red \\ \frac{3}{4} - \frac{1}{4}(E_x{+}E_x^{-1}{+}E_y{+}E_y^{-1}) + \frac{1}{16}(E_x + E_x^{-1} + E_y + E_y^{-1})^2\,, & (j,k)\ \ black \end{cases} \tag{2.8c}$$

Again, this preconditioning consists of a forward solve (L^{-1}) followed by a backward solve (U^{-1}). In this case, however, since all the red points can be updated in parallel and so can the all the black points, the solves can at best be done in constant time. The stencil representations of the local operators for the diagonally-ordered and parallel red/black-ordered ILU preconditioners can be found in (Ref. 4).

2.3.2 MILU preconditioners[7,8,9,10]

2.3.2.1 MILU with diagonal ordering

Again there are many variations within the class of MILU preconditioners.[10] One such MILU preconditioner with diagonal ordering is defined as follows

$$M_M = L\ U \tag{2.9a}$$

with the requirements that $M_M(i,j) = A\,(i,j)$ for all $A\,(i,j) \neq 0\ i \neq j$ and the sum of the entries of each row of the error matrix $R_M = M_M - A$ equals $\delta = ch^2$, where h is the grid spacing and c is a nonnegative constant that is independent of h. For the model Poisson problem with Dirichlet boundary conditions, the L and U matrices can be calculated by performing an LU decomposition of the M_M matrix. For the model Poisson problem with periodic boundary conditions, the local operator can be obtained as

$$M_{M\,(j,k)} = \frac{1}{4}[a + \frac{2}{a} - (E_x + E_y + E_x^{-1} + E_y^{-1}) + \frac{1}{a}(E_xE_y^{-1} + E_x^{-1}E_y)] \quad (2.9b)$$

Again, equating the entries of M_M and A according to the requirements, we have

$$a + \frac{4}{a} - 4 = \delta\,. \quad (2.9c)$$

By solving the above equation, we obtain

$$a = 2 + \frac{\delta}{2} + \frac{1}{2}\sqrt{8\,\delta + \delta^2} \quad (2.9d)$$

The steps involved in this preconditioning are the same as those of the ILU with diagonal ordering. Hence, this preconditioning can at best be completed in $O(\sqrt{N})$ time on parallel computers.

2.3.2.2 MILU with parallel red/black ordering

The local operators $L_{j,k}$ and $U_{j,k}$ for the parallel red/black ordering are

$$L_{j,k} = \begin{cases} 1+\delta\,, & (j,k)\ red \\ 1+\delta - \frac{1}{1+\delta} - \frac{1}{4}(E_x + E_x^{-1} + E_y + E_y^{-1})\,, & (j,k)\ black \end{cases} \quad (2.10a)$$

$$U_{j,k} = \begin{cases} 1 - \frac{1}{4(1+\delta)}(E_x + E_x^{-1} + E_y + E_y^{-1})\,, & (j,k)\ red \\ 1\,, & (j,k)\ black \end{cases} \quad (2.10b)$$

where δ has been defined in the previous sub-section.

Again, we obtain the parallel red/black-ordered MILU preconditioner as

$$M_{Mrb\,(j,k)} =$$

$$\begin{cases} 1+\delta - \frac{1}{4}(E_x + E_x^{-1} + E_y + E_y^{-1})\,, & (j,k)\ red \\ 1+\delta - \frac{1}{1+\delta} - \frac{1}{4}(E_x+E_x^{-1}+E_y+E_y^{-1}) + \frac{1}{16(1+\delta)}(E_x+E_x^{-1}+E_y+E_y^{-1})^2, & (j,k)\ black. \end{cases} \quad (2.10c)$$

This preconditioner, as in the ILU preconditioner with parallel red/black ordering, takes constant time independent of the number of grid points, N, on parallel computers.

2.3.3 Symmetric Successive Overrelaxation (SSOR) precondtioners[4,7,10,11]

2.3.3.1 SSOR with diagonal ordering

The SSOR preconditioner is defined to be

$$M_S = (D - \omega L) D^{-1} (D - \omega L^T) , \tag{2.11a}$$

where D and L are diagonal and strictly lower triangular matrices respectively and and ω is the relaxation parameter such that

$$A = D - L - L^T . \tag{2.11b}$$

For the model Poisson problem with diagonal ordering and periodic boundary conditions, the partitioning leads to the following local operators :

$$D_{j,k} = 1 , \quad L_{j,k} = \frac{1}{4} (E_x^{-1} + E_y^{-1}) , \quad L_{j,k}^{-T} = \frac{1}{4} (E_x + E_y) . \tag{2.11c}$$

The product $M_{S\,(j,k)}$ is thus

$$M_{S\,(j,k)} = 1 - \frac{\omega}{4}(E_x + E_y + E_x^{-1} + E_y^{-1}) + \frac{\omega^2}{16}(2 + E_x^{-1} E_y + E_x E_y^{-1}) . \tag{2.11d}$$

Again, as in ILU and MILU preconditioners with diagonal ordering, this preconditioning also takes $O(\sqrt{N})$ time on parallel computers.

2.3.3.2 SSOR with parallel red/black ordering

For the model Poisson problem with parallel red/black ordering, the partitioning leads to the following local operators :

$$D_{j,k} = 1 , \tag{2.12a}$$

$$L_{j,k} = \begin{cases} 0 , & (j,k)\ red \\ \frac{1}{4} (E_x + E_x^{-1} + E_y + E_y^{-1}) , & (j,k)\ black \end{cases} \tag{2.12b}$$

$$U_{j,k} = \begin{cases} \frac{1}{4} (E_x + E_x^{-1} + E_y + E_y^{-1}) , & (j,k)\ red \\ 0 , & (j,k)\ black \end{cases} \tag{2.12c}$$

Therefore, we obtain the parallel red/black-ordered SSOR preconditioner as

$M_{Srb\,(j,k)} =$

$$\begin{cases} 1 - \frac{\omega}{4}\,(E_x + E_x^{-1} + E_y + E_y^{-1})\,, & (j,k)\ red \\ 1 - \frac{\omega}{4}(E_x + E_x^{-1} + E_y + E_y^{-1}) + \frac{\omega^2}{16}(E_x + E_x^{-1} + E_y + E_y^{-1})^2\,, & (j,k)\ black \end{cases} \tag{2.12d}$$

2.3.4 Polynomial preconditioners [12–17]

2.3.4.1 m-step Jacobi preconditioner

The *m*-step Jacobi preconditioner approximates the inverse of the matrix $A = (I - B)\,P^{-1}$ by using the truncated Neumann series expansion,

$$A^{-1} = P\,(I - B)^{-1} \approx P\,(I + B + B^2 + \cdots + B^{m-1}) = M_{P\,(m)}^{-1}\,, \tag{2.13}$$

where, in operator form, $B = E_x + E_x^{-1} + E_y + E_y^{-1}$.

2.3.4.2 Parametrized polynomial preconditioner

In general, we can consider the polynomial preconditioner as

$$M_{GP\,(m)}^{-1} = \sum_{l=0}^{m-1} \gamma_l\, B^l\,, \tag{2.14}$$

so that the coefficients γ_l, $0 \le l \le m$, can be chosen to minimize a certain objective function. Examples of such preconditioners are the least-squares polynomial preconditioner and the min-max polynomial preconditioner.[15] Only the least-squares preconditioners with m=2,3 and 4 will be implemented here.

2.3.4.4 Other polynomial preconditioners [14]

The same idea for parametrized polynomial preconditioners can be applied to a different splitting - for example, the SSOR splitting. This gives rise to the *m*-step SSOR preconditioners. A variation of this is the multi-color *m*-step SSOR preconditioner. No implementation for this preconditioner is included here.

The polynomial preconditioners formulated above are very good candidates for parallel computation. If the *P* matrix is the identity matrix, then the $m - 1$ steps for the *m*-step Jacobi preconditioning amount to $m - 1$ iterations of the basic Jacobi method followed by accumulating the results of the iterations. As the basic Jacobi method offers a very high degree of parallelism (all grid points can be updated at the same time), so does this m-step Jacobi preconditioner. Thus, on massively parallel computer systems such as the CM, the m-step Jacobi preconditioning takes $O(m)$ time.

2.3.5 Hierarchical basis preconditioner[18,19]

Another class of preconditioners which is effective in reducing the number of iterations required for convergence is based on a hierarchical basis formulation of the finite element discretization of the problem domain. Define the preconditioner as

$$M_{hb} = S\,S^T \tag{2.15}$$

where S operating on a vector is equivalent to sweeping through the levels in ascending order (fine to coarse grid) and accumulating the operations of the grid points on the fine grid to those on the coarse grid; and S^T operating on a vector is equivalent to sweeping through the levels in descending order (coarse to fine grid) and accumulating the operations of the grid points on the coarse grid to those on the fine grid. Since the total number of levels is $O(\log\ n)$ where n is the number of grid points in each of the x or y direction and the operations performed on each level take $O(1)$ time, this preconditioning process can at best be done in $O(\log\ n)$ time.

2.4 Convergence rates[4,7,20]

The convergence rates of the conjugate gradient method with different preconditioners for the model Poisson problem, which depend on both the corresponding condition number as well as the distribution of the eigenvalues of the preconditioned system, can be studied either by matrix iterative analysis or by Fourier analysis.[4] A summary of the results of the Fourier analysis of the preconditioners can be found in (Ref. 4).

3. Implementation

3.1 The Connection Machine

The detailed description of the Connection Machine can be found in (Ref. 21-23). The CM used in these experiments is a 16k-node CM-2 running at a clock frequency of about 6.7 MHz. The language used for program development was *LISP. During the experimental phase, it was found that if single-precision floating point numbers were used, there was a discrepancy between the recursively computed residual and the actual residual. Hence, it was decided to use double-precision floating point numbers throughout the experiments.

3.2 Processor mapping

The 2-D model problem can be nicely mapped onto the 2-D NEWS grid of the Connection Machine. Depending on the ratio of the number of grid points to the number of available physical processors, each physical processor simulates one or more grid points. To run a 128×128 grid problem, the following configuration command in *LISP can be used :

*(*cold–boot :initial–dimensions* '(128 128))

In this case, since there are altogether $128 \times 128 = 16384$ grid points and we have 16384 (16k) physical processors, each physical processor can perform the computations for a single grid point. Suppose 65336 (256×256) grid points are to be simulated but the same number of physical processors (16k) are available, then each physical processor has to take the computation load of 4 grid points. By using the virtual processing capability of the CM, this mapping (the mapping of 4 grid points to one physical processor) is transparent to the users and is performed automatically when the

*(*cold–boot :initial–dimensions* '(256 256))

statement is executed.

An advantage of mapping the problem on the 2D NEWS grid is that the neighboring grid points are mapped to neighboring processors; and since the communication overhead between neighboring processors using the NEWS communication is extremely fast, and that most of the communications are between local grid points in most cases, good total execution time performance can be expected.

One major concern is that if each processor simulates one grid point and red/black ordering is used, then only half of the processors will be active during updating the black or the red grid points, resulting in low processor utilization. It happens that the virtual processing capability on the CM can handle this problem elegantly : if the number of grid points is less than or equal to the number of physical processors available, then some processors will be idle in any case, and there is no way to improve the utilization of the processors. Suppose the number of grid points is 4 times as many as the number of physical processors, then by using the **cold–boot* as described above, two black and two red points will be mapped to each physical processor. Here we can see that the physical processors will be kept busy all of the time, computing either black or red points.

Another concern is how the boundary grid points are handled. Since no computation is needed for the boundary grid points other than providing data to their neighbors, one way is not to map them to any processors. During computation, these processors that simulate the grid points which are located next to the boundary grid points will be performing slightly different tasks from the other interior processors. An example is the local operation $A_{j,k}$ which requires fetching data from four neighbors (north, south, east, and west). Since these next-to-boundary grid points have one or two neighbors missing (e.g. the grid points at the upper right hand corner will not have east and north neighbors), they have to execute this operator a little differently. Because the CM is a SIMD machine and cannot execute two different active operations simultaneously, the updating of interior and next-to-boundary grid points have to be done in two separate steps, resulting in longer execution time. Another way is to map boundary grid points also to actual processors. The drawback to this scheme is that these boundary processors will be idle most of the time. However, since the operations to be performed on all interior processors will be identical, the updating takes only one step, resulting in shorter execution time compared to the first scheme. This latter scheme is chosen for our implementation for reasons that it is simple to implement and it will probably give better performance.

To distinguish between the boundary grid points and the interior ones, a parallel boolean variable, *grid−interior−flag*, is declared which is set to true for the interior grid points, and false for the boundary grid points. Every time computation is to be performed only on the interior processors (or grid points), the following *LISP statement can be used to achieve the desired results.

```
(*when grid-interior-flag (do something to the grid points))
```

3.3 The preconditioned conjugate gradient method on the CM

3.3.1 Implementation of basic conjugate gradient iteration

One iteration of the PCG method requires 3 inner products (including the residual calculation), 3 multiply-and-add operations, 1 matrix-vector product calculation, 2 scalar divisions, one comparison, plus the computations required for preconditioning. Let's look at how each of these operations is performed on the CM.

3.3.1.1 Multiply-and-add operation (SAXPY)

This operation is in the form of $y=ax+b$ where x and b are vectors and a is a scalar. If each processor takes care of one element in the vector and the scalar a is supplied by the host system, then the tasks performed by each processor are first to receive the a, then multiply a by the x which is stored within the processor, and lastly add to this product the variable b which is also stored within the processor. The *LISP version of this operation is

```
(*when grid-interior-flag (*set pvar-y (+!! (*!! pvar-x (!! a)) pvar-b)))
```

The time to perform this operation depends only on the virtual processor ratio (number of grid points per physical processor) and this ratio depends on the total number of grid points and the total number of available processors. For a particular virtual processor ratio (VP ratio), this operation takes constant time.

3.3.1.2 Matrix-vector product

The matrix in this case is A, which, when operating on a vector, is equivalent to the parallel execution of the local operator $A_{j,k}$ (defined previously) on each element of the vector. For the Poisson problem, each processor adds the data fetched from the processors to its north, south, east, and west, divides the sum by 4, and subtracts the quotient from its own data. In *LISP code, it can be represented as

```
(*when grid-interior-flag
   (*set pvar-ap (-!! pvar-p
                 (/!! (+!!
                       (news!! pvar-p 1 0)
                       (news!! pvar-p 0 1)
                       (news!! pvar-p -1 0)
                       (news!! pvar-p 0 -1)
                     )
                     (!! 4.0)
                 )
              )
   )
)
```

where the 'news!!' function has 3 arguments - the variable to be fetched from the destination processor, the relative distance of the destination processor in the x-direction, and the relative distance of the destination processor in the y-direction respectively. (The piece of code described above reflects the change in the software release 5.0. The old version used 'pref-relative-grid!!' instead of 'news!!'.) Here the parallel variables 'pvar-p' and 'pvar-ap' are the inputs and outputs respectively.

Again, this operation can be done in constant time for a particular VP ratio.

3.3.1.3 Inner Product

The inner product has been known as a bottleneck to the performance of the PCG method. It is important that this inner product operation can be done efficiently. It can be observed that the hypercube configuration of the CM helps to speed up this computation. It allows multiplication to be done in parallel in all processors, and the the partial sums are accumulated in the form of a binary tree. The *LISP code for inner product calculation is

```
(*when grid-interior-flag (setq inner-product (*sum (*!! pvar-r pvar-r))))
```

3.3.1.4 Others

Other computational needs include 2 scalar divides and 1 comparison for convergence. These computations are performed on the front end computer and require constant time.

In summary, without considering the preconditioning, the overall computation time for each iteration on the CM is dominated by the inner product operation. Even though the parallel computational complexity of the inner product computation is $O(\log N)$ where N is the number of grid points, it can be seen later that the performance of the inner product calculation on the CM is comparable to the other operations such as the SAXPY operation.

3.3.2 Implementation of preconditioners

For preconditioning, depending on whether diagonal ordering or red/black ordering is used, the way to map the preconditioning algorithms on the CM and the order of computation times can be quite different.

3.3.2.1 Implementation of preconditioners with diagonal ordering

Assuming that grid points (1,1) and (n,n) are located at the lower left and upper right corners of the domain respectively, then the L-solve starts at the grid point (1,1) and updates one diagonal at a time until grid point (n,n) is reached, where n is the number of interior grid points in each dimensions. This constitutes a wavefront moving from the lower left corner to the upper right corner and is called a forward sweep. During the forward sweep, each active processor (those processors located on the wavefront) updates its grid point value by averaging with the corresponding variables fetched from its south and west neighbors. The *LISP code is

```
(*when processor-is-active
   (*set pvar-destination
      (-!! pvar-source
          (*!! pvar-a (news!! pvar-destination -1 0))
          (*!! pvar-b (news!! pvar-destination 0 -1))
      )
   )
)
```

where *pvar-a* and *pvar-b* are weighting factors corresponding to the entries in the unit lower triangular L matrix (recall $M = LU$).

The backward sweep (or the U-solve) can be performed similarly except that now the sweep starts at grid point (n,n) and proceeds to grid point (1,1). Also, since U does not have unit diagonal, an additional division is needed.

The way to select a diagonal of grid points to be updated and leave the other processors idle can be achieved by the use of a parallel boolean variable. This variable is initially set to false on all processors except the one corresponding to grid point (1,1). The 'true' value of this variable can be programmed to propagate diagonally. This technique can be applied to the ILU, MILU and SSOR preconditioners with natural ordering. For diagonal ordering, since there are $O(\sqrt{N})$ diagonals in a 2-D grid, the corresponding preconditioning takes $O(\sqrt{N})$ time.

3.3.2.2 Implementation of preconditioners with red/black ordering

The preconditioners with red/black ordering can be implemented using two parallel boolean flags, namely the red and the black flags, which indicate whether the corresponding grid point is red or black point. When selecting the red points for updating, this can be done by

```
(*when red-flag (do something to the red points))
```

and the black points can be updated in the same way. For red/black ordering, since all the red points can be updated in parallel, and so are the black points, the corresponding preconditioning only takes constant time. This technique is used on the implementation of ILU, MILU and SSOR preconditioners with red/black ordering.

3.3.2.3 Implementation of polynomial preconditioners

During each step of the *m*-step Jacobi preconditioning (for the Poisson problem), all processors corresponding to the interior grid points are active and they all fetch data from their north, south, east and west neighbors by performing

```
(*when grid-interior-flag
  (*set pvar-destination
    (/!! (+!! (news!! pvar-destination 0 1)
              (news!! pvar-destination 1 0)
              (news!! pvar-destination 0 -1)
              (news!! pvar-destination -1 0)
         )
         (!! 4.0)
    )
  )
)
```

and this is to be performed m-1 times and the *pvar-destination* is is to be accumulated. With a fixed VP ratio, the polynomial preconditioning takes $O(m)$ time.

3.3.2.4 Implementation of the hierarchical basis preconditioner

This preconditioning step involves sweeping upward or downward through all the levels. At each level, the grid points belonging to the corresponding level need to communicate with processors which are at a relative distance of a power of 2 away in the x and/or y direction. This can be achieved by using either the NEWS!! or the *SETF functions depending on whether the active processors are receiving or sending data respectively. To identify the level to which a grid point belongs, a parallel integer variable is used to store the level number. Moreover, since three types of communication patterns are required depending on where the grid points are located, some mechanism is needed to distinguish them. (Refer to Ref. 19 for details.) There are several ways to achieve this, and the way used here is to create another parallel variable to do the job. Once all these criteria have been established, the details of implementation is straightforward; thus they are not to be narrated.

3.4 Experiments

The PCG method was applied to the model problem described in section 2.2 with different values of $q(x,y)$. In particular, the following sections describe the numerical experiments that were performed.

3.4.1 Experiment 1

The model problem was the Poisson equation with $p(x,y) = q(x,y) = 1$, $r(x,y) = g(x,y) = 0$ and initial guess $u(0) = 1$. Various preconditioners were used as listed below using $n = 33$, 65, 129, 257 and 513. The stopping criterion used was $||r_i|| / ||r_0|| \leq 10^{-6}$.

- Conjugate gradient method without preconditioning (CG)
- PCG - ILU with diagonal ordering (ILU diagonal)
- PCG - MILU with diagonal ordering (MILU diagonal)
- PCG - SSOR with diagonal ordering and $\omega = 1$ (SSOR diagonal)
- PCG - ILU with red/black ordering (ILU (R/B))
- PCG - MILU with red/black ordering (MILU (R/B))
- PCG - SSOR with red/black ordering and $\omega = 1$ (SSOR (R/B))
- PCG - *m*-step Jacobi preconditioner (different m were used)

- PCG - least-squares polynomial preconditioner with m = 2 where $\gamma_0 = 7/6$ and $\gamma_1 = 5/6$ (LS2)
- PCG - least-squares polynomial preconditioner with m = 3 where $\gamma_0 = 35/32$, $\gamma_1 = 50/32$ and $\gamma_2 = 35/32$ (LS3)
- PCG - least-squares polynomial preconditioner with m = 4 where $\gamma_0 = 37/40$, $\gamma_1 = 49/40$, $\gamma_2 = 91/40$ and $\gamma_3 = 63/40$ (LS4)
- PCG - hierarchical basis preconditioner (PCGHB)

3.4.2 Experiment 2

The same model problem was used with $p(x,y) = 1$, $q(x,y) = 10$ and the rest are the same as in experiment 1. Only the preconditioners that give relatively good performance in experiment 1 are used in this experiment.

3.4.3 Experiment 3

The same model problem was used with $p(x,y) = 1$, $q(x,y) = 100$ and the rest are the same as in experiment 2.

For each of the experiments, the following types of data are to be obtained :

- the iteration counts to achieve convergence, and
- the execution time on the CM.

4. Results and discussion

4.1 Results

The iteration counts for experiment 1 are shown in Table 1. These iteration counts can be used to verify the results of theoretical analyses based on Fourier analysis.[4] The corresponding CM cpu times are shown in Table 2.

Table 1 : Iteration counts for experiment 1

Preconditioner	Iteration counts				
	n = 33	n = 65	n = 129	n = 257	n = 513
CG	53	103	203	397	772
ILU (diagonal)	8	14	23	42	69
MILU (diagonal)	9	12	18	25	34
SSOR (diagonal)	9	13	17	25	34
ILU (R/B)	26	52	101	199	386
MILU (R/B)	34	60	113	225	421
SSOR (R/B)	26	52	101	199	386
2-step Jacobi	26	51	101	197	384
4-step Jacobi	19	37	71	139	270
6-step Jacobi	15	30	58	113	220
8-step Jacobi	13	26	50	98	191
LS2	29	56	110	216	421
LS3	21	42	82	161	315
LS4	16	32	62	122	238
PCGHB	31	38	44	48	53

Table 2 : CM execution time for experiment 1

Preconditioner	CM cpu time (in sec)				
	n = 33	n = 65	n = 129	n = 257	n = 513
CG	2.9	5.6	11	51	319
ILU (diagonal)	35	115	370	---	---
MILU (diagonal)	24	73	210	---	---
SSOR (diagonal)	27	78	230	---	---
ILU (R/B)	2.6	5.2	10.1	47	292
MILU (R/B)	4.2	7.6	13.9	63	387
SSOR (R/B)	2.4	4.7	9.1	43	276
2-step Jacobi	2.1	4.2	7.9	37	235
4-step Jacobi	2.1	4.0	7.4	36	228
6-step Jacobi	2.1	3.8	7.6	38	236
8-step Jacobi	2.2	4.3	7.8	39	249
LS2	3.2	6.3	12.3	51	306
LS3	3.0	6.0	11.5	49	294
LS4	2.8	5.6	10.9	45	269
PCGHB	10.8	17.7	26	78	337

The MFLOPS for the PCG procedures in experiment 1 were calculated and shown in Table 3. To calculate the MFLOPS, only the standard floating point arithmetic operations such as addition and multiplication were counted. Operations such as data copying, logic evaluations and data communication operations were ignored.

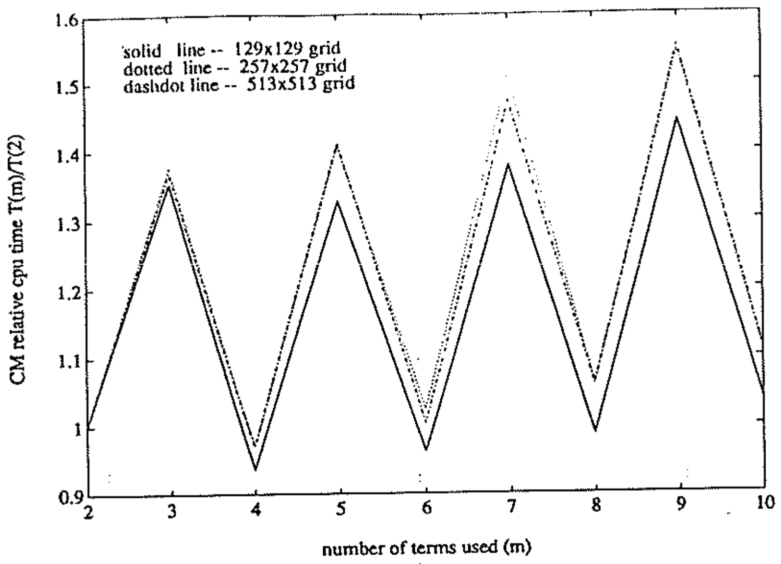

Figure 2 : *m*-step Jacobi Preconditioner - T(m)/T(2) versus *m*

Also the iteration counts and the CM execution times for experiments 2 and 3 are shown in the following tables.

Table 4 : Iteration counts for experiment 2

Preconditioner	Iteration counts				
	n = 33	n = 65	n = 129	n = 257	n = 513
CG	82	159	308	594	1155
ILU (R/B)	41	79	154	297	577
SSOR (R/B)	41	79	154	297	577
2-step Jacobi	41	79	154	296	571
4-step Jacobi	29	56	109	209	404
6-step Jacobi	24	46	89	171	330
LS3	33	64	125	242	472
LS4	25	49	95	183	353

Table 5 : CM cpu time for experiment 2

Preconditioner	CM cpu time (in sec)				
	n = 33	n = 65	n = 129	n = 257	n = 513
CG	4.6	8.9	17.2	82	547
ILU (R/B)	5.0	9.5	18.4	81	507
SSOR (R/B)	4.6	8.8	17.1	76	480
2-step Jacobi	3.7	7.2	13.9	64	393
4-step Jacobi	3.5	6.8	13.2	65	389
6-step Jacobi	3.8	7.3	14.2	69	407
LS3	4.9	9.5	18.5	78	475
LS4	4.6	9.0	17.4	72	431

Table 3 : MFLOPS on CM for the preconditioners for experiment 1

Preconditioner	MFLOPS		
	n = 129	n = 257	n = 513
CG	4.5	7.7	9.5
ILU (diagonal)	0.079	---	---
MILU (diagonal)	0.068	---	---
SSOR (diagonal)	0.065	---	---
ILU (R/B)	3.7	6.3	7.8
MILU (R/B)	3.1	5.3	6.6
SSOR (R/B)	4.0	6.6	8.1
2-step Jacobi	4.6	7.7	9.4
4-step Jacobi	5.0	8.2	9.9
6-step Jacobi	5.3	8.3	10.2
8-step Jacobi	5.5	8.6	10.5
LS2	3.5	6.6	8.7
LS3	3.5	6.5	8.4
LS4	3.4	6.4	8.4
PCGHB	0.65	0.95	0.97

To examine the relationship between the number of terms used, the iteration counts and the CM execution times for the *m*-step Jacobi preconditioner, the iteration counts and execution times were plotted as in Figure 1 and 2.

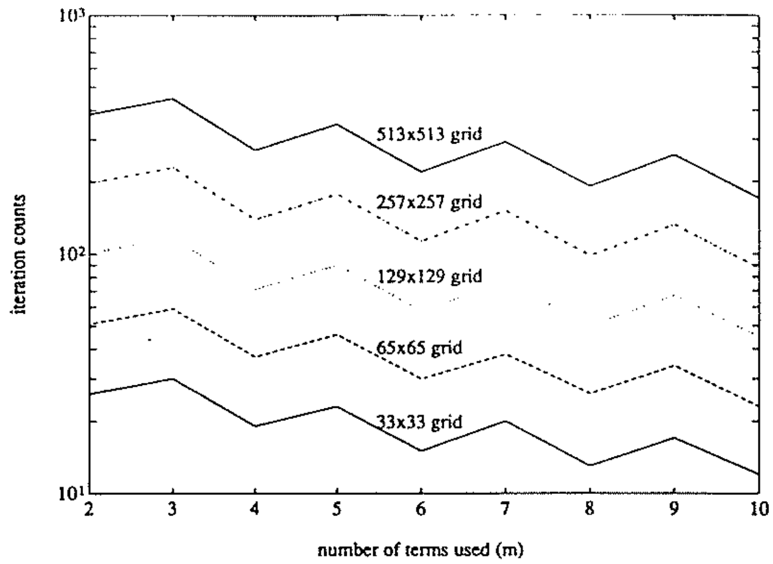

Figure 1 : *m*-step Jacobi Preconditioner - Iteration counts versus m

Table 6 : Iteration counts for experiment 3

Preconditioner	Iteration counts				
	n = 33	n = 65	n = 129	n = 257	n = 513
CG	106	227	441	830	1547
ILU (R/B)	53	113	220	415	773
SSOR (R/B)	53	114	220	415	773
2-step Jacobi	55	115	224	420	788
4-step Jacobi	40	83	158	297	555
6-step Jacobi	33	68	129	242	452
LS3	46	93	180	338	673
LS4	34	71	135	256	478

Table 7 : CM cpu time for experiment 3

Preconditioner	CM cpu time (in sec)				
	n = 33	n = 65	n = 129	n = 257	n = 513
CG	6.3	13.4	26.2	115	725
ILU (R/B)	6.4	13.7	26.3	113	677
SSOR (R/B)	5.9	12.7	24.3	106	640
2-step Jacobi	5.0	10.5	20.2	88	538
4-step Jacobi	4.9	10.1	19.2	89	532
6-step Jacobi	5.5	11.2	21.4	94	555
LS3	6.9	13.8	26.6	109	639
LS4	6.3	13.0	24.8	101	582

4.2 The Connection Machine Statistics

To evaluate the performance of the PCG method on the CM, the execution time statistics were gathered on some basic operations of the PCG algorithm. These statistics were taken by averaging the CM cpu times from a few sample runs with each run performing the corresponding operations (on parallel variables) 100 times. Table 8 shows that MFLOPS counts of the operations using double-precision floating point numbers while Table 9 shows the MFLOPS count when single-precision floating point numbers are used. The CM used to gather the statistics is the CM-2 with 16k nodes and with single-precision floating point hardware. The operations to be examined are :

- two-operand addition (ADD)
- data copying (COPY)
- two-operand multiplication when both operands are parallel variables (MULT)

- two-operand multiplication when one operand is a scalar variable (MULTA)
- multiply by a scalar and add to a parallel variable (SAXPY)
- two-operand division when both operands are parallel variables (DIV)
- two-operand division when one operand is a scalar variable (DIVA)
- data communication using NEWS grid when distance = 1 (COMM)
- matrix-vector multiplication involving 4 neighbor processors (MVP) which consists of 5 multiplication and a 5-operand addition
- inner product calculation (INNER)

Table 8 : CM execution time for different types of operations (double-precision)

	128x128 grid	256x256 grid	512x512 grid
operation	MFLOPS	MFLOPS	MFLOPS
ADD	13.3	14.4	14.9
COPY	117	234	320
MULT	8.5	9.0	9.1
MULTA	2.5	5.4	8.7
SAXPY	4.4	8.0	11.2
DIV	3.3	3.4	3.4
DIVA	1.7	3.2	3.3
COMM	39.5	64	100
MVP	1.0	1.1	1.2
INNER	1.9	3.3	3.8

Table 9 : CM execution time for different types of operations (single-precision)

	128x128 grid	256x256 grid	512x512 grid
operation	MFLOPS	MFLOPS	MFLOPS
ADD	87	160	180
COPY	328	392	548
MULT	99	152	169
MULTA	3.6	13.7	43
SAXPY	3.5	12.8	36
DIV	78	96.4	107
DIVA	3.5	12.9	37
COMM	67	119	167
MVP	6.4	11.1	14.1
INNER	23.4	74	153

It can be observed that single-precision floating point operations are much faster than the double-precision floating point operations. The reason is that single-precision floating point operations are performed in the floating point hardware which is much faster than the single-bit processors. Another observation from the tables is that NEWS communication and data copying are much faster than the arithmetic operations. Also the inner product time is not much worse than, for example, MULTA even though the former was expected to run in O(P) time where P is the number of processors and the latter was expected to run in O(1) time.

Still another observation is that the increase in VP ratio improves the MFLOPS performance consistently. This behavior demonstrates the positive effect of virtual processing.

4.3 Discussion

- The execution times and MFLOPS obtained in this experiment seems to be far inferior than the performances on the CRAY computers. For example, it was shown in (Ref. 7), for example, that on the CRAY X-MP the PCG method using MILU preconditioner on the 100×100 grid took only a fraction of a second to achieve convergence, while it took about 7 seconds for the best preconditioner on the CM. This poor relative performance is due to a few factors :
    - In this experiment, only one quarter (16k node) of the full machine is used. If we assume a linear scaling factor, the full machine should perform four times as fast.
    - Double-precision floating point numbers are used in these experiments. However, the CM used was only equipped with single-precision floating point hardware. As a result, the double-precision floating point arithmetic operations have to be done on the single-bit processors which are much slower. (Refer to Table 8 and 9.) If the CM used had double-precision floating point hardware, significant improvement in performance would have been possible. (The reason why double-precision floating point numbers are needed is explained in section 3.1.)
    - Since the purpose of this experiment is to compare the relative performances of the preconditioners, no effort was put to optimized the program code. The source program was interpreted rather than compiled. Using optimized code should help improve the performance.
    - It was shown that high VP ratio will improve performance a great deal. When 100×100 grid is used on a 16k node machine, the VP ratio is only 1. If the size of the problem is increased, better performance is expected.

Based on the above factors, it is believed that if everything mentioned is improved, the MFLOPS performance of the CM for the PCG method will be comparable to that of the CRAY.

- The first observation about the running time performances is that the speedup using the best preconditioner (4-step Jacobi, as of Table 2) attempted is not even twice that of the case without any preconditioning. One reason is that the performance of inner product calculation is comparable to that of the other operations (refer to Table 8 and 9) and so the inner product calculation becomes less of a major bottleneck. Another reason is that the basic conjugate gradient method requires only two inner product calculations as opposed to three for preconditioned CG method.

- From Table 1, we can see that although the preconditioners using diagonal ordering give much lower iteration counts than the ones with red/black ordering, their performance on the CM is very poor. The CM execution time of the MILU preconditioner with diagonal ordering, for example, is about 15 times that of the the same preconditioner with red/black ordering (refer to Table 2) for the 129×129 grid, spending most of the time in preconditioning step. Thus, we can conclude that such preconditioners with natural ordering are not good candidates on fine-grain massively parallel computers such as the CM. However, it should be mentioned that if the CM is having mesh-connected network instead of the hypercube network, the preconditioners using natural ordering may turn out to be competitive, since then the inner product time will be quite significant.

- It can be observed from Figure 1 that with increasing number of terms used in the *m*-step Jacobi preconditioning, the number of iterations needed to achieve convergence oscillates but continues to decrease slowly. This agrees with the theoretical analysis, as discussed in Ref. 12. However, from Figure 2, it can be seen that the number of terms which gives optimal running time is 4 for all the experiments attempted. This agrees qualitatively with the results obtained by others.[24]

- Even though the least-squares polynomial preconditioners give close to optimal convergence rate improvement, their performances on the CM are worse than the *m*-step Jacobi preconditioners. Comparing the 6-step Jacobi preconditioner with the 4-step least-squares preconditioner, it can be observed that they give more or less the same iteration count for the Poisson problem. However, the former requires more data communication and less multiplications than the latter. (This is true for at least the isotropic and anisotropic cases). As the NEWS communication time is much faster than the multiplication time as shown in Table 8, it is not difficult to explain this result. It can be predicted that if the ratio of communication time to the arithmetic computation time is increased, there is a point when the least-squares preconditioners will be better. This demonstrates again the impact of architectural characteristics of a parallel computer on the choice of a good preconditioner.

- It was shown from experimental results (Ref. 5,6) that the use of different multi-color orderings does have some effect on convergence rate. For example, 4-color ordering preconditioners are generally better than red/black orderings in terms of convergence rate performance. This behavior should suggest a relationship between VP ratio on the CM (due to the use of finer grids) on the choice of orderings. For example, suppose the VP ratio needed on the CM for a certain problem is four and the red/black ordering is used and 2 red and 2 black grid points are mapped to one physical processor. We can see that even though the 2 red grid points on a processor can potentially be updated at the same time, they can only be updated in two passes. So in this case, if 4-color ordering improves convergence rate performance, we might as well use 4-color ordering instead in this case.

- The hierarchical basis preconditioner performs quite well on the CM even though it is not the optimal one. It can be observed that the growth rate of the CM execution time is less for the hierarchical basis preconditioner than for the others, implying that this preconditioner may turn out to be better for much finer grids when the number of physical processors is kept the same. The requirement for heavy global communication and the relatively slow hypercube network on the CM (compared to the NEWS communication) may be one major reason for the lower performance. A richer and faster interconnection network will make this preconditioner very competitive. It can also be observed that the MFLOPS for this preconditioner is quite low, as is characteristic of multigrid-type algorithms.[25] This demonstrates again that MFLOPS achieved by an algorithm is not the sole determining factor for best performance on parallel computers. (Ref. 25 has demonstrated this well for the multigrid method on the CM.)

5. Conclusion

The rapid growth of parallel computing has revived many numerical algorithms which were deemed as inefficient on sequential computers. While the sequential computational complexity of an algorithm plays a major role in determining its competitiveness, many other factors have to be taken into consideration in the context of parallel computations. The results of the numerical experiments were able to confirm some of these factors for the PCG method. From the algorithmic point of view, both the convergence rate improvement and degree of parallelism are important factors to achieve high efficiency on parallel computers. As these two factors have an adverse effect on each other, tradeoffs have to be made to balance the two to get the best performance. From the architectural point of view, the communication structure of the computers, the ratio of communication speed versus the arithmetic speed also have major impact on the choice of good preconditioners. In conclusion, the experimental results seem to suggest that both the polynomial preconditioners and the hierarchical basis preconditioner are promising candidates on the CM-like computers.

Acknowledgements

The author would especially like to thank Professor Tony Chan for his many suggestions, advices and encouragement leading to the completion of this paper. Also many thanks are to Professor Jay Kuo for his advices and support, and to June Donato for her many comments. The author would also like to thank the Parallel Processing Group at the Information Sciences Institute (ISI) of the University of Southern California (USC) for providing the CM computing facilities as well as instructions of how to use them prior to the installation of CM-2 at UCLA.

References

1. Ortega, J. M., and Voigt, R. G., "Solution of Partial Differential Equations on Vector and Parallel Computers," *SIAM Review*, Vol. 27, No. 2, pp. 149-240, June 1985.
2. Golub, G. H., Van Loan, C. F.,*Matrix Computations*, the Johns Hopkin University Press, 1983, chapter 6.
3. Chandra, R., *Conjugate Gradient Methods for Partial Differential Equations*, Ph. D. Thesis, Computer Science Department, Yale University, 1978.
4. Chan, Tony F., Kuo, C. C. Jay, and Tong, Charles, "Parallel Elliptic Preconditioners : Fourier Analysis and Performance on the Connection Machine", Department of Mathematics, CAM report 88-22, UCLA, August 1988.
5. Agron, E., "Ordering Techniques for the Preconditioned Conjugate Gradient Method on Parallel Computers," Master Thesis, University of Maryland, 1987.
6. Duff, I. S., and Meurant, G. A., "The Effect of Ordering on Preconditioned Conjugate Gradients," September 1988.
7. Chan, Tony F., and Elman, Howard C., "Fourier Analysis of Iterative Methods for Elliptic Problems", CAM Report 87-04, Department of Mathematics, UCLA, 1988. To appear in *SIAM Review*.
8. Dupont, T., Kendall, R. P., and Rachford, H. H. Jr., "An Approximate Factorization Procedure for Solving Self-adjoint Difference Equations," *SIAM J. Numer. Anal.*, vol. 5, No. 3, pp. 559-573.
9. Meijerink, J. A., and Van der Vorst, H. A., "An iterative Solution Method for Linear Systems of which the Coefficient matrix is a symmetric M-Matrix," *Math. Comp.*, Vol. 31, no. 137, pp. 148-162, 1977.
10. Ashcraft, C. C., and Grimes, R.G., "On Vectorizing Incomplete Factorization and SSOR Preconditioners," *SIAM J. Sci. Stat. Comput.*, Vol. 9, No. 1, pp. 122-151, 1985.

11. Axelsson, O., "A Generalized SSOR Method," *BIT*, Vol. 13, pp. 443-467, 1972.

12. Ortega, J. M., *Introduction to Parallel and Vector Solution of Linear Systems*, Frontier of Computer Science, Plenum Press, New York, 1988, Section 3.4.

13. Donato, J., "Fourier Analysis of Polynomial Preconditioners for the 5-point Laplacian," Term Paper, Department of Mathematics, UCLA, 1988.

14. Adams, L., "*m*-step Preconditioned Conjugate Gradient Methods," *SIAM J. Sci. Stat. Comput.*, Vol. 6, No. 2, April 1985.

15. Johnson, O., Micchelli, C. A., Paul, G., "Polynomial Preconditioners for Conjugate Gradient Calculations", *SIAM J. Numer. Anal.*, Vol. 20, No. 2, April 1983, pp. 362-376.

16. Saad, Y., "Practical Use of Polynomial Preconditionings for the Conjugate Gradient Method," *SIAM J. Sci. Stat. Comput.*, Vol. 6, No. 4, Oct. 1985.

17. Ashby, S. F., "Polynomial Preconditioning for Conjugate Gradient Methods," department of Computer Science, U. of Illinois at Urbana-champaign, Report No. UIUCDCS-R-87-1355, 1987.

18. Yserentant, H., "On the Multilevel Splitting of Finite Element Spaces," *Numer. Math.*, Vol. 49. 1986, pp. 349-412.

19. Adam, L. M., and Ong, E. G., "A Comparison of Preconditioners for GMRES on Parallel Computers," in *Parallel Computations and Their Impact on Mechanics*, ed. A. K. Noor, pp. 171-186, The American Society of Mechanical Engineers, New York, N. Y., 1987.

20. Kuo, C.-C. Jay, and Chan, Tony F., "Two-color Fourier Analysis of Iterative Algorithms for Elliptic Problems with Red/Black Ordering", CAM Report 88-15, Department of Mathematics, UCLA, 1988.

21. Hillis, W. D.,*The Connection Machine*, MIT Press, Cambridge, MA, 1985.

22. McBryan, O. A., "State-of-the-art in Highly Parallel Computer Systems, " in *Parallel Computations and Their Impact Mechanics*, ed. A. K. Noor, pp. 31-46, The American Society of Mechanical Engineers, New York, 1987.

23. *Connection Machine Model CM-2 Technical Summary* , by the Thinking Machine Corporation.

24. Jordan, T. L., "Conjugate Gradient Preconditioners for Vector and Parallel Processors," *Elliptic Problem Solvers II*, 1983, pp.127-140.

25. McBryan, O. A., "The Connection Machine : PDE solution on 65536 Processors," *Parallel Computing*, to appear.

26. Hageman, L. A., Young, D. M., *Applied Iterative Methods*, Academic Press, N.Y., 1988.