UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Experiments with Algebraic Multilevel Preconditioners on Connection Machine

I.D. Mishev
V. Austel
T.F. Chan
P.S. Vassilevski

July 1993

CAM Report 93-25

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

EXPERIMENTS WITH ALGEBRAIC MULTILEVEL PRECONDITIONERS ON CONNECTION MACHINE
July 24, 1993

I.D. MISHEV, V. AUSTEL, T.F. CHAN, AND P.S. VASSILEVSKI

ABSTRACT. The performance of algebraic two– and multilevel preconditioners are compared with unpreconditioned **CG** method on a massively parallel computer CM–2, using a modification of C–language.

1. INTRODUCTION

In recent years conjugate gradient (**CG**) method became one of the most efficient methods for the solution of linear systems of algebraic equations that arise in the finite element solution of elliptic differential equations. The main advantage of **CG** is that its convergence does not require any choice of parameters, unlike other methods as **SOR**, **SSOR** etc. The convergence of the **CG** can be substantially improved by a proper choice of a preconditioner. In recent time much work has been devoted to the construction of preconditioners of optimal order, i.e., to solve a linear system one needs $O(n \log 1/\varepsilon)$ arithmetic operations (executed on sequential computer). Here n is the number of unknowns and $\varepsilon > 0$ is the tolerance used in the **CG** method.

Scientific and engineering practice poses that more complicated problems to be solved. One possible respond to this demand is to use parallel computers. Massively parallel computers such as the Connection machine (**CM**) possess high degree of parallelism, but most of the commonly used algorithms on sequential and vector computers, like pointwise and block–incomplete factorization preconditioners for the **CG** are not as easy to parallelize. Unpreconditioned **CG** (**CG**(none)) can exploit very successfully the hypercube architecture of **CM**, but its performance can deteriorate for more difficult problems like problems with discontinuous coefficients of the differential equation. The algebraic multilevel iteration (**AMLI**) preconditioners (cf. Axelsson and Vassilevski [2], [3], Vassilevski [11]) which are stable versions of the

---

1991 *Mathematics Subject Classification.* 65N20, 65F10.

Key words and phrases. multilevel preconditioning, CM-2, massively parallel computation, parallel implementation, elliptic problem.

This work of the first, third and the fourth authors has been partially supported by the National Science Foundation grants NSF ASC 92–01266 and NSF INT 92–20287, and also in part by the Department of Energy under contract DE–FG03–87ER25037 and by the Army Research Office under contract ARO DAAL03-91-G-150.

hierarchical basis methods (see, Yserentant [12], Bank, Dupont and Yserentant [6]) both in two and three space dimensions, are very efficient for such class of difficult problems.

This report demonstrates that the **AMLI** methods can be efficiently implemented on massively parallel computers such as **CM–2**; we achieve good timings and our results are competitive with previously performed work in this direction by Tong [9] and Elman and Guo [8]. Our implementation of the **AMLI** methods uses the hybrid multilevel cycle as proposed in Vassilevski [11]. In this way, while preserving the optimality of the method we were able to increase the parallelizable properties of the algorithm since it is very close to a V–cycle one. Our experiments show that indeed the hybrid multilevel cycle becomes close in performance to the pure V–cycle, which is already very efficient for two dimensional domains. In three dimensions the V–cycle is not as good, so we expect that the hybrid multilevel cycle to provide very efficient parallelizable method.

The remainder of this report is organized as follows. In §2 the two– and multilevel hierarchical basis preconditioners are formulated and their properties outlined. §3 contains description of the main feature of an **C**–based language and our parallel implementation of **AMLI**. Section §4 contains the experiments we performed. Finally, in the Appendix, we provide two codes written in **UC** that implement the **CG** (none) and **PCG** with two–level preconditioning.

2. Two and Multilevel Preconditioners

We consider the following model diffusion equation:

$$(1) \qquad \begin{cases} -\nabla.(a(x)\nabla u(x)) &= f(x) & \text{in } \Omega, \\ u(x) &= g(x) & \text{on } \Gamma_1, \\ \frac{\partial u}{\partial n}(x) &= 0 & \text{on } \Gamma_2, \end{cases}$$

where $\Omega = [0,1] \times [0,1]$, $\Gamma = \partial\Omega$, $\Gamma = \Gamma_1 \cup \Gamma_2$, $a(x)$, $f(x)$ are given functions in Ω and $g(x)$ is defined on Γ_1.

We use piece-wise linear finite elements to discretize the problem on a sequence of successively refined triangulations

$$T_1, T_2, \cdots, T_l$$

and the corresponding nested (by construction) finite element spaces are

$$V_1 \subset V_2 \subset \cdots \subset V_l.$$

Then we compute the stiffness matrices

$$A^{(k)} = \{a(\phi_j^{(k)}, \phi_i^{(k)})\}_{i,j=1}^{n_k} \qquad k = 1, 2, \ldots, l$$

where N_k is the set of nodes at level k and $\{\phi_i^{(k)}\}_{i=1}^{n_k}$ is the standard nodal basis of V_k. By construction we have $N_{k-1} \subset N_k$. We use at the k th level the partition

$N_k \backslash N_{k-1}$ (new nodes) and N_{k-1} (old nodes) of the nodal set N_k . Corresponding to this ordering, $A^{(k)}$ admits the following block two-by-two form:

$$A^{(k)} = \begin{bmatrix} A_{11}^{(k)} & A_{12}^{(k)} \\ A_{21}^{(k)} & A_{22}^{(k)} \end{bmatrix},$$

where

$$\begin{aligned} A_{11}^{(k)} &= \{a(\phi_j^{(k)}, \phi_i^{(k)}\}_{i,j} \ : \ x_i, x_j \in N_k \backslash N_{k-1} \\ A_{12}^{(k)} &= \{a(\phi_j^{(k)}, \phi_i^{(k)}\}_{i,j} \ : \ x_i \in N_{k-1}, x_j \in N_k \backslash N_{k-1} \\ A_{22}^{(k)} &= \{a(\phi_j^{(k)}, \phi_i^{(k)}\}_{i,j} \ : \ x_i, x_j \in N_{k-1}. \end{aligned}$$

Lemma 2.1. *(Bank and Dupont* [5]*)* $A_{11}^{(k)}$ *is a well–conditioned matrix, i.e.,*

$$\|A_{11}^{(k)}\| = \sup_{v_1 \neq 0} \frac{\|A_{11}^{(k)} v_1\|}{\|v_1\|} = O(1), \qquad \|A_{11}^{(k)^{-1}}\| = O(1),$$

and there exist two positive constants d_1 *and* d_2*, independent of* k *and the possible jumps in the coefficient* $a(x)$*, such that*

$$d_1 \leq \frac{v_1^T A_{11}^{(k)} v_1}{v_1^T D_{11}^{(k)} v_1} \leq d_2 \qquad \text{for all } v_1 \in R^{n_k - n_{k-1}},$$

where $D_{11}^{(k)} = diag(A_{11}^{(k)})$.

Corollary 2.1. *We can solve problems* $A_{11}^{(k)} v_1 = w_1$ *by the* ***CG*** *method, using* $D_{11}^{(k)}$ *as a preconditioner, independent of possible jumps in the coefficients in* (1)*, for* $O(\log 1/\varepsilon)$ *iterations with an accuracy* $\varepsilon > 0$.

Lemma 2.2. *(Bank and Dupont* [5]*, Axelsson and Gustafsson* [1]*) There is a constant* $\gamma \in (0,1)$ *that can be estimated locally, i.e.,* $\gamma = \max_{\tau \in T_k} \gamma_\tau$*, independent of the possible jumps in the coefficient of* (1)*, such that the following strengthened Cauchy inequality holds:*

$$a(v_1, v_2) \leq \gamma (a(v_1, v_1)^{1/2} (a(v_2, v_2)^{1/2}$$

for all $v_2 \in V_{k-1}$*, and* $v_1 \in V_k$ *such that* $v_1(x) = 0$*, all* $x \in N_{k-1}$.

Consider now the Choleski factorization of the matrix $A^{(k)}$:

$$A^{(k)} = \begin{bmatrix} A_{11}^{(k)} & 0 \\ A_{21}^{(k)} & S^{(k)} \end{bmatrix} \begin{bmatrix} I & A_{11}^{(k)^{-1}} A_{12}^{(k)} \\ 0 & I \end{bmatrix}, \tag{2}$$

where $S^{(k)}$ is the Schur complement

$$S^{(k)} = A_{22}^{(k)} - A_{21}^{(k)} A_{11}^{(k)^{-1}} A_{12}^{(k)}.$$

The following equivalence result holds (cf. Axelsson and Gustafsson [1]).

Lemma 2.3. *Let γ be a constant in the strengthened CBS inequality. Then*

$$1-\gamma^2 \le \frac{v_2^T S^{(k)} v_2}{v_2^T A^{(k-1)} v_2} \le 1 \qquad \text{for all } v_2 \in R^{n_{k-1}}.$$

This motivates the following two–level preconditioner (Bank and Dupont [5] and Axelsson and Gustafsson [1]).

Definition 2.1 (Two-level preconditioner).

$$M^{(l)} = \begin{bmatrix} A_{11}^{(l)} & 0 \\ A_{21}^{(l)} & A^{(l-1)} \end{bmatrix} \begin{bmatrix} I & A_{11}^{(l)^{-1}} A_{12}^{(l)} \\ 0 & I \end{bmatrix}. \tag{3}$$

To solve a given problem $M^{(l)}x = y$, where x and y are n_l dimensional vectors we perform:

Forward:
- solve $A_{11}^{(l)} v_1 = y_1$,
- compute $z = A_{21}^{(l)} v_1$,
- solve $A^{(l-1)} v_2 = y_2 - z$,

Backward:
- set $x_2 = v_2$,
- compute $w = A_{12}^{(l)} x_2$,
- solve $A_{11}^{(l)} z = w$,
- compute $x_1 = v_1 - z$.

We show in the next section that the above operations can be implemented very successfully on the **CM-2**. If we solve a system with the matrix $A^{(l-1)}$ by the **CG** (none), the two-level preconditioner is not optimal in the sense that it takes too many iterations to converge.

Now we define multilevel preconditioner which is almost optimal, i.e., $Cond\,(M^{(l)^{-1}} A^{(l)}) = O(l^2) = O(\log^2 n)$, cf. e.g., Vassilevski [10], [11].

Definition 2.2 (V–cycle multilevel preconditioner).

$$Set \quad M^{(1)} = A^{(1)},$$

and for $k = 2, 3, \ldots, l$, *let*

$$M^{(k)} = \begin{bmatrix} A_{11}^{(k)} & 0 \\ A_{21}^{(k)} & M^{(k-1)} \end{bmatrix} \begin{bmatrix} I & A_{11}^{(k)^{-1}} A_{12}^{(k)} \\ 0 & I \end{bmatrix}. \tag{4}$$

If we replace the Schur complement $S^{(k)}$ in (2) with a more accurate approximation than $M^{(k-1)}$ as in (4) we can achieve optimality of the resulting multilevel preconditioner [2], [3].

Definition 2.3 (Algebraic multilevel iteration (AMLI) preconditioner).

$$\textit{Set} \quad M^{(1)} = A^{(1)}$$

for $k = 2, 3, \ldots, l$

$$M^{(k)} = \begin{bmatrix} A_{11}^{(k)} & 0 \\ A_{21}^{(k)} & \tilde{M}_\nu^{(k-1)} \end{bmatrix} \begin{bmatrix} I & A_{11}^{(k)^{-1}} A_{12}^{(k)} \\ 0 & I \end{bmatrix}. \tag{5}$$

where

$$\tilde{M}_\nu^{(k)^{-1}} = \left[I - p_\nu \left(M^{(k)^{-1}} A^{(k)} \right) \right] A^{(k)^{-1}}. \tag{6}$$

Here $p_\nu(t)$ is a polynomial of degree $\nu \geq 1$ such that

(1) $p_\nu(0) = 1$,

(2) $0 \leq p_\nu(t) < 1$, $t \in (0, \Delta)$ for some $\Delta \geq 1$.

Then $1 - p_\nu(t) = Q_{\nu-1}(t)t$ for some polynomial $Q_{\nu-1}(t)$ of degree $\nu - 1$ and

$$\left[I - p_\nu \left(M^{(k)^{-1}} A^{(k)} \right) \right] A^{(k)^{-1}} = Q_{\nu-1} \left(M^{(k)^{-1}} A^{(k)} \right) M^{(k)^{-1}},$$

i.e., $\tilde{M}_\nu^{(k)^{-1}}$ does not contain $A^{(k)^{-1}}$. In the case of first degree polynomial, namely for $p_\nu(t) = 1 - t$ we get the pure V-cycle multilevel preconditioner (4). In Vassilevski [11] it has been shown that one can allow a hybrid V-cycle, i.e., to let $\nu = 1$ and $p_\nu(t) = 1 - t$, hence $\tilde{M}_\nu^{(k-1)} = M^{(k-1)}$, at most of the levels and only for $k = sk_0 + 1$, $s = 1, 2, \ldots, l/k_0 - 1$, for a given parameter $k_0 > 1$ to use $\tilde{M}_\nu^{(k-1)}$ for a sufficiently large $\nu = \nu_k$, and still preserve the optimality of the method.

The following Horner scheme implements one preconditioning step for solving $\tilde{M}_\nu^{(k-1)} x = y$.

Algorithm 1. *Let* $Q_\nu(t) = q_0 + q_1 t + \ldots + q_{\nu-1} t^{\nu-1}$.

$x^{(0)} = 0$

for $r = 1$ *until* ν *solve*

$M^{(k-1)} x^{(r-1)} = q_{\nu-r} y + A^{(k-1)} x^{(r-)}$

Then $x = x^{(r)}$

3. Parallel implementation of the algorithms

Here we show how once suitable data structure is chosen, all considered algorithms can be very easily prepared for execution on a parallel computer.

We have used the parallel language **UC** [4] to implement the method. **UC** is a minor modification of **C**. It adds the following features to **C**: a data type – *index-set*, a meta-operator – *reduction*, and four constructs *par*, *solve*, *oneof* and *seq* to express dependencies among statements.

An index-set is a constant data item that represents an ordered set of integers. An index-set is typically declared as follows:

```
index_set I:i = {0..N-1};
```

The first identifier (I in the example) refers to the index-set as a whole; the second identifier refers to an arbitrary *element* of the set, and is used to reference index elements within **UC** constructs.

Parallel assignments are the most common **UC** constructs. The simplest form of parallel assignment simultaneously modifies all elements in an array. For instance, the following statement transposes the array b in parallel,

```
par (I,J)
   b[i][j] = b[j][i];
```

The par operator introduces the index-sets I and J into scope, so it is possible to refer to elements of I and J within the body of the par statement.

A par statement may use a predicate to select a subset of the elements from the index-set. The predicate is used to determine the subset of enabled index elements (an index element is enabled if on substituting its value in the predicate, the predicate evaluates to *true*) and the statement is executed in parallel for all enabled elements. The following statement assigns zero to the diagonal of array b.

```
par (I,J)
   st (i==j)
      b[i][j] = 0;
```

A reduction operator performs an associative binary operation on a set of operands, and returns the resulting value. The following statement sums the squares of all the elements of the array b.

```
sum = $+(I,J; b[i][j] * b[i][j]);
```

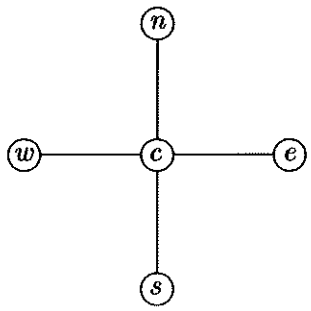
UC programs can be compiled for execution on the Connection Machine 2, a massively parallel computer with a hypercube interconnection network. The hypercube architecture allows very efficient embedding of two dimensional meshes into a 2^m-processor hypercube, when number of points n in every direction is power of two. Moreover, this embedding has properties of perfect shuffle. We use this fact to eliminate difficulties with handling boundary points [7], [9]. Depending on the ratio of the numbers of grid points to the number of available physical processors, each physical processor simulated one or more grid points.

We rewrite finite element approximation as a finite difference one with a stencil shown on fig. 1 and difference operator $A_h^{(k)}$, which encloses on function y in mesh point with coordinates $(x_{1,i}, x_{2,j})$ as

$$
\begin{aligned}
A_h^{(k)} y(x_{1,i}, x_{2,j}) \;=\; & ac^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i}, x_{2,j}) + aw^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i-1}, x_{2,j}) \\
& + ae^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i+1}, x_{2,j}) + an^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i}, x_{2,j+1}) \\
& + as^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i}, x_{2,j-1}).
\end{aligned} \tag{7}
$$

For every mesh point $(x_{1,i}, x_{2,j})$, which is not on the boundary with Dirichlet boundary condition Γ_1, we store in the processor (i, j) the stiffness matrix entries, $ac[k][i][j]$, $aw[k][i][j]$, $ae[k][i][j]$, $an[k][i][j]$, $as[k][i][j]$. If the point $(x_{1,i}, x_{2,j})$ has neighbor(s) on Γ_1, which are south and/or west from it, then $as[k][i][j]$ and/or $aw[k][i][j]$ are equal to zero. The same is valid for points on the boundary with Newman boundary

FIGURE 1. A finite element stencil



condition. Hence we can multiply matrix $A^{(k)}$ times vector y with five multiplications and four additions in every processor. Communications are realized via NEWS network. The operator that computes $z = A^{(k)}y$ is

```
par (I,J)
    z[i][j] = ac[i][j] * y[i][j] + aw[i][j] * y[i-1][j]
              + ae[i][j] * y[i+1][j]+ an[i][j] * y[i][j+1]
              + as[i][j] * y[i][j-1]
```

Multiplication $A_{11}^{(k)}v_1$ (if we solve problem $A_{11}^{(k)}v_1 = y_1$ by **CG** like method), $A_{12}^{(k)}x_2$ and $A_{12}^{(k)}v_1$ are performed in the same way, but not in every processor. For example $y_1 = A_{11}^{(k)}v_1$ is executed as

```
par (I,J)
    st (i%2 == 0 !! j%2 == 0)
       y[i][j] = ac[i][j] * v[i][j] + aw[i][j] v[i-1][j]
                 + ae[i][j] * v[i+1][j] + an[i][j] * v[i][j+1]
                 + as[i][j] * v[i][j-1] ,
```

The above statement "st" means that at least one of the indices i or j is even.

Another very important point for the efficiency of the **CG** operations is the inner product. Here is an operator that computes the standard inner product of two given vectors x and y, $q = (y, v) = y^T v$

```
q = $+(I,J; y[i][j] * v[i][j]);
```

4. NUMERICAL EXPERIMENTS

In this section we present our numerical tests for solving problem (1) discretized on an uniform square mesh of size $h = 1/n$, n is a given integer, using piecewise

linear functions on right triangles. The Dirichlet boundary Γ_1 is on the boundaries $x = 0$ and $y = 0$. On the boundaries $x = 1$ and $y = 1$ Neumann boundary conditions are imposed.

The test problems are:

Problem 1. $u(x) = 1\,, \qquad a(x) = 1\cdot$

Problem 2.

$$u(x) = e^{-x_1}(x_1^2 + x_2^2)\,, \qquad a(x) = 1/(1 + x_1^2 + x_2^2)\cdot$$

Problem 3 (discontinuous coefficient $a(x)$).

$$u(x) = (x_1 - x_1^{(0)})(x_2 - x_2^{(0)})\Phi(x)/a(x)\,,$$

$$a(x) = \begin{cases} 1, & x_1 < x_1^{(0)} \quad or \quad x_2 < x_2^{(0)} \\ 10^3, & x_1 > x_1^{(0)} \quad and \quad x_2 > x_2^{(0)} \end{cases}$$

and we have chosen

$$x_1^{(0)} = x_2^{(0)} = 0.5\,, \qquad \Phi(x) = \sin(1/2\pi x_1)/(1 + x_1^2 + x_2^2);$$

Problem 4 (a singular solution).

$$u(x) = \log 1/r\,, \qquad r^2 = (1 - x_1)^2 + (1 - x_2)^2\,, \qquad a(x) = 1.$$

The right hand side here is a delta function

$$f(x) = 1/2\pi\delta((1,1) - x) = \begin{cases} 0, & x \neq (1,1) \\ 1/2x & x = (1,1) \end{cases}$$

In Table 1 we present the number of iterations and the CPU time for the V–cycle multilevel preconditioned **CG**. At the coarsest level we use the **CG**(none). This is not very good choice for problem 3 when the coarse level used is not coarse enough in the sense that the convergence of the coarse–grid solver deteriorates because of the discontinuous coefficient. However, for that difficult Problem 3 an obvious improvement in timing is seen when the number of levels is increased (or equivalently when the coarse–grid problem is small enough)..

Results for the hybrid V-cycle [11] are reported in Table 2. In the first column we show the degrees ν of the polynomials $p_\nu^k(t)$ used at the discretization level $k\,, k = 2, \ldots, 6$. The degrees of the polynomials used at the first (coarse) and at the finest level are assumed one and are not shown.

"*no exper.*" stands for a run that takes more than five minutes.

It is seen that the hybrid mulitlevel preconditioner becomes close to the performance of the V–cycle method for certain inexpensive choices of the polynomial degrees, e.g., for the set of polynomial degrees (1,1,3,1,1) the implementation of the hybrid multilevel iteration method requires 53.16 seconds versus 35.48 seconds for the pure V–cycle when seven levels are used.

TABLE 1. Iteration and CPU time for the V-cycle, $h = 1/128$

levels\prob.#		1	2	3	4
CG(none)	*iter*	401	570	11236	407
	time	3.56 *s*	5.06 *s*	99.74 *s*	5.35 *s*
2 levels	*iter*	7	7	9	8
	time	21.84 *s*	19.84 *s*	488.43 *s*	19.34 *s*
3 levels	*iter*	9	9	10	11
	time	19.47 *s*	21.93 *s*	241.60 *s*	23.36 *s*
4 levels	*iter*	11	11	12	13
	time	20.61 *s*	21.59 *s*	110.46 *s*	24.26 *s*
5 levels	*iter*	13	13	13	15
	time	24.34 *s*	24.85 *s*	56.10 *s*	27.53 *s*
6 levels	*iter*	14	15	14	17
	time	28.16 *s*	30.17 *s*	36.04 *s*	34.08 *s*
7 levels	*iter*	15	16	14	17
	time	35.17 *s*	35.48 *s*	33.88 *s*	38.71 *s*

TABLE 2. Iteration and CPU time for the hybrid V-cycle, Problem 2, $h = 1/128$

polyn./levels		3	4	5	6	7
(1 1 2 1 1)	*iter*	9	11	10	12	14
	time	21.93 *s*	21.59 *s*	35.50 *s*	41.00 *s*	62.62 *s*
(1 2 1 1 2)	*iter*	9	9	11	13	11
	time	21.93 *s*	33.67 *s*	35.11 *s*	39.20 *s*	61.75 *s*
(2 1 2 1 2)	*iter*	8	10	9	12	10
	time	38.15 *s*	33.07 *s*	49.06 *s*	54.97 *s*	80.32 *s*
(2 2 2 2 2)	*iter*	8	8	9	9	9
	time	38.15 *s*	53.92 *s*	89.33 *s*	143.35 *s*	215.57 *s*
(1 1 3 1 1)	*iter*	9	11	7	10	12
	time	21.93 *s*	21.59 *s*	39.16 *s*	48.10 *s*	53.16 *s*
(1 3 1 1 3)	*iter*	9	7	10	11	10
	time	21.93 *s*	40.70 *s*	45.20 *s*	44.26 *s*	104.30 *s*
(3 1 3 1 3)	*iter*	7	9	9	11	12
	time	53.02 *s*	43.01 *s*	95.11 *s*	89.67 *s*	224.18 *s*
(3 3 3 3 3)	*iter*	7	8	9	*no*	*no*
	time	53.02 *s*	112.27 *s*	260.01 *s*	*exper.*	*exper.*

REFERENCES

1. O. Axelsson and I. Gustafsson. Preconditioning and two-level multigrid methods of arbitrary degree of approximations. *Math. Comput.*, 40:219–242, 1983.
2. O. Axelsson and P. S. Vassilevski. Algebraic multilevel preconditioning methods,I. *Numer. Math.*, 56:157–177, 1989.
3. O. Axelsson and P. S. Vassilevski. Algebraic multilevel preconditioning methods,II. *SIAM J. Numer. Anal.*, 27:1569–1590, 1990.
4. R. Bagrodia, K. M. Chandy, and E. Kwan. UC: A language for the connection machine. *IEEE*, 53:525–534, 1990.
5. R. Bank and T. Dupont. Analysis of a two-level scheme for solving finite element equations. *Report CNA-159, center for Numerical Analysis, The University of Texas at Austin*, 1980.
6. R. Bank, T. Dupont, and H. Yserentant. The hierarchical basis multigrid method. *Numer. Math*, 52:427–458, 1988.
7. H. C. Elman and E. Agron. Ordering techniques for the preconditioned conjugate gradient method on parallel computers. *Computer Physics Communications*, 53:253–269, 1989.
8. H. C. Elman and X. Guo. Performance enhancement and parallel algorithms for two multilevel preconditioners. *UMIACS TR-91-143, The University of Maryland, College Park, MD 20742*, 1991.
9. C. H. Tong. *Parallel preconditioned conjugate gradient methods for elliptic partial differential equations.* PhD thesis, University of California at Los Angeles, 1989.
10. P. S. Vassilevski. Nearly optimal iterative methods for solving finite element elliptic equations based on the multilevel splitting of the matrix. *Report of Institute for Scientific Computation, University of Wyoming, Laramie, USA*, 9, 1989.
11. P. S. Vassilevski. Hybrid V-cycle algebraic multilevel preconditioning methods. *Math. Comput.*, 58:489–512, 1992.
12. H. Yserentant. On the multilevel splitting of finite element spaces. *Numer. Math*, 49:379–412, 1986.

5. Appendix

Here we provide two computer codes written in **UC**. The first one implements CG(none) for solving the linear system $Ax = b$. First we show the algorithm and then incorporate it in the **UC** code.

Algorithm 2 (CG(none)). *Initialization*

(1) Choose initial solution x_0, $x_0 = 0$,
(2) Compute the defect $g_0 = Ax - b = -b$,
(3) Compute the search direction $d_0 = -g_0$,
$\delta_0 = b^T b$,
Compute the relative tolerance $\varepsilon_1 = \delta_0 \varepsilon$,
Main loop:
while $(\delta_0 > \varepsilon_1)$ {
$h = Ad_i$,
$\tau = \delta_0 / d_i^T h$,
$x_{i+1} = x_i + \tau d_i$,
$g_{i+1} = g_i + \tau h$,
$\delta_1 = g_{i+1}^T g_{i+1}$,
$\beta = \delta_1 / \delta_0$,
$\delta_0 = \delta_1$,
$\delta_{i+1} = -g_{i+1} + \beta d_{i+1}$,
$i = i + 1$,
}

```
#define  N   128

index_set I:i = {0..N-1}, J:j = {0..N-1};

float   ac[N][N], an[N][N], aw[N][N], ae[N][N], as[N][N];

void  cg(float x[N][N], float b{N}{N}, int *iter)
{
     float  d[N][N], g[N][N], h[N][N];
     float  eps, eps1, tau, beta, delta0, delta1, dth;

     eps = 1e-12;
     *iter = 0;

/* initialization          */
     par (I,J) {
        x[i][j] = 0.0;
        g[i][j] = - b[i][j];
        d[i][j] = - g[i][j];
     }
```

```
        delta0 = $+(I,J; b[i][j]*b[i][j]);

        eps1 = eps * delta0;

        while(delta0 > eps1){
             *iter += 1;
    /* multiplication matrix times vector  - h = A * d      */
             par (I,J)
                h[i][j] = ac[i][j]*d[i][j] +
                          aw[i][j]*d[i-1][j] +
                          ae[i][j]*d[i+1][j] +
                          an[i][j]*d[i][j+1] +
                          as[i][j]*d[i][j-1];
             dth = $+(I,J; d[i][j]*h[i][j]);
             tau = delta0 / dth;
             par (I,J)
                x[i][j] += tau * d[i][j];
             par (I,J)
                g[i][j] += tau * h[i][j];
             delta1 = $+(I,J; g[i][j]*g[i][j]);
             beta = delta1 / delta0;
             delta0 = delta1;
             par (I,J)
                d[i][j] = - g[i][j] + beta*d[i][j];
        }
    }
```

The second algorithm implements the two-level preconditioned **CG**. To solve linear systems with matrix A_{11}, i.e. to solve $A_{11}x = b$ we use the **CG** with diagonal preconditioner (one Jacobi iteration). This preconditioner makes the algorithm more robust in the case when the coefficient of the differential equation has large jumps. The subroutine "cga11" realizes this algorithm. We do not use vectors with the size of the matrix A_{11}. Multiplications with the matrix A_{11} have been commented on in §3.

```
#define  N2   128
#define  N1    64

index_set I2 : i2 = {0..N2-1}, J2 : j2 = {0..N2-1},
          I1 : i1 = {0..N1-1}, J1 : j_1 = {0..N1-1};

float  ac2[N2][N2], an2[N2][N2], aw2[N2][N2],
       ae2[N2][N2], as2[N2][N2],
       ac1[N1][N1], an1[N1][N1], aw1[N1][N1],
       ae1[N1][N1], as1[N1][N1];
```

```
void  cga11(float x[N2][N2], float b[N2][N2])
{
     float  d[N2][N2], g[N2][N2], h[N2][N2];
     float  eps, eps1, tau, beta;
     float  delta, delta0, dth;
     int    itera11;

     par (I2,J2) {
         d[i2][j2] = 0.0;
         g[i2][j2] = 0.0;
         x[i2][j2] = 0.0;
         h[i2][j2] = 0.0;
     }
     eps11 = 1e-6
     itera11 = 0;
/*  initial guess   x =  b / ac                          */
     par (I2,J2)
/*       st ( (i,j) \in fine )                           */
         st (i2 % 2 == 0 || j2 % 2 == 0)
            x[i2][j2] = b[i2][j2] / ac2[i2][j2];
/*  compute defect    d = Ax, g = d - rhs                */
     par (I2,J2)
/*       st ( (i,j) \in fine )                           */
         st (i2 % 2 == 0 || j2 % 2 == 0)
            d[i2][j2] = ac2[i2][j2]*x[i2][j2] +
                        aw2[i2][j2]*x[i2-1][j2] +
                        ae2[i2][j2]*x[i2+1][j2] +
                        an2[i2][j2]*x[i2][j2+1] +
                        as2[i2][j2]*x[i2][j2-1];
     par (I2,J2)
/*       st ( (i,j) \in fine )                           */
         st (i2 % 2 == 0 || j2 % 2 == 0)
            g[i2][j2] = d[i2][j2] - rhs[i2][j2];
/*  compute pseudoresidual   h = g / ac                  */
     par (I2,J2)
/*       st ( (i,j) \in fine )                           */
         st (i2 % 2 == 0 || j2 % 2 == 0)
            h[i2][j2] = g[i2][j2] / ac2[i2][j2];
     par (I2,J2)
         d[i2][j2] = - h[i2][j2];

     delta0 = $+(I2,J2; g[i2][j2]*h[i2][j2]);
     delta = delta0;
```

```
    eps1 = eps11 * delta0;

    while(delta0 > eps1){
        itera11 += 1;
  par (I2,J2)
/*          st ( (i,j) \in fine )                          */
            st (i2 % 2 == 0 || j2 % 2 == 0)
               h[i2][j2] = ac2[i2][j2]*d[i2][j2] +
                           aw2[i2][j2]*d[i2-1][j2] +
                           ae2[i2][j2]*d[i2+1][j2] +
                           an2[i2][j2]*d[i2][j2+1] +
                           as2[i2][j2]*d[i2][j2-1];

          dth = $+(I2,J2; d[i2][j2]*h[i2][j2]);
          tau = delta0 / dth;
          par (I2,J2)
              x[i2][j2] += tau * d[i2][j2];
          par (I2,J2)
              g[i2][j2] += tau * h[i2][j2];
        par (I2,J2)
/*          st ( (i,j) \in fine )                          */
            st (i2 % 2 == 0 || j2 % 2 == 0)
              h[i2][j2] = g[i2][j2] / ac2[i2][j2];
         delta1 = $+(I2,J2; g[i2][j2]*h[i2][j2]);
         beta = delta1 / delta0;
         delta0 = delta1;
         par (I2,J2)
             d[i2][j2] = - h[i2][j2] + beta*d[i2][j2];
    }
}
```

The next subroutine "minvy" implements one preconditioning step with the matrix M defined by (3). On the coarse grid we solve the problem by **CG**(none). To use the subroutine "cg" we transfer the data between the discretization levels by simple copy operation. It is worth trying other approaches; for example, the product A(coarse) times a vector implemented without any transfer of data in the way we executed similar operation with the matrix A_{11}.

```
/*   solve  Mx = y -> x = M(-1)y                          */
void  minvy(float x[N2][N2], float y[N2][N2])
{
     float  v[N2][N2], w[N2][N2], z[N2][N2],
            x2[N1][N1], z2[N1][N1];

     par (I2,J2) {
         x[i2][j2] = 0.0;
```

```
        w[i2][j2] = 0.0;
        z[i2][j2] = 0.0;
    }

/*  solve  A11*v1 = y1                                          */
    cga11(v, y);

/*  compute  w = A21*v1                                         */
    par (I2,J2)
/*      st ( (i,j) \in coarse)                                  */
        st (i2 % 2 == 1 && j2 % 2 == 1)
           w[i2][j2] = aw2[i2][j2]*v[i2-1][j2] +
                       ae2[i2][j2]*v[i2+1][j2] +
                       an2[i2][j2]*v[i2][j2+1]+
                       as2[i2][j2]*v[i2][j2-1];

/*  compute   z = y2 - w                                        */
    par (I2,J2)
/*      st ( (i,j) \in coarse)                                  */
        st (i2 % 2 == 1 && j2 % 2 == 1)
           z[i2][j2] = y[i2][j2] - w[i2][j2];

/*  transfer  z;  z[N2][N2] -> z2[N1][N1]                       */
    par (I1,J1)
        z2[i1][j_1] = z[2*i1+1][2*j_1+1];

/*  AMLI, for twolevel  solve  A*v2 = y2                        */
    cg(x2, z2);

/*  transfer  x2;  x2[N1][N1] -> x[N2][N2]                      */
    par (I1,J1)
        x[2*i1+1][2*j_1+1] = x2[i1][j_1];

/*  compute  w = A12*v2  , v2 = x2                              */
    par (I2,J2)
/*      st ( (i,j) \in fine )                                   */
        st (i2 % 2 == 0 || j2 % 2 == 0)
           w[i2][j2] = aw2[i2][j2]*x[i2-1][j2] +
                       ae2[i2][j2]*x[i2+1][j2] +
                       an2[i2][j2]*x[i2][j2+1]+
                       as2[i2][j2]*x[i2][j2-1];

/*  solve  A11*z = w                                            */
    cga11(z, w);
```

```
/*  compute  x1 = v1 - z                                    */
     par (I2,J2)
/*       st ( (i,j) \in fine )                              */
         st (i2 % 2 == 0 || j2 % 2 == 0)
             x[i2][j2] = v[i2][j2] - z[i2][j2];
}
```

Finally, the last subroutine "pcg" implements the **PCG** with a preconditioner M.

```
/* pcg  with  preconditioner  M, solve  Ax = b     */
void  pcg(float x[N2][N2], float b[N2][N2], int *iter)
{
     float  d[N2][N2], g[N2][N2], h[N2][N2];
     float  eps, eps1, tau, beta;
     float  delta0, delta1, dth;

     eps = 1e-12;
     *iter = 0;

/*  computing the initial solution  m * x = rhs         */
     minvy(x, rhs);

/*  computing the defect   d = Ax ,  g = d - b                  */
     par (I2,J2)
         d[i2][j2] = ac2[i2][j2]*x[i2][j2] +
                     aw2[i2][j2]*x[i2-1][j2] +
                     ae2[i2][j2]*x[i2+1][j2] +
                     an2[i2][j2]*x[i2][j2+1] +
                     as2[i2][j2]*x[i2][j2-1];
     par (I2,J2)
         g[i2][j2] = d[i2][j2] - b[i2][j2];

     minvy(h, g);
     par (I2,J2)
         d[i2][j2] = - h[i2][j2];
     delta0 = $+(I2,J2; g[i2][j2]*h[i2][j2]);

     eps1 = eps * delta0;

     while(delta0 > eps1){
          *iter += 1;
          par (I2,J2)
              h[i2][j2] = ac2[i2][j2]*d[i2][j2] +
```

```
                        aw2[i2][j2]*d[i2-1][j2] +
                        ae2[i2][j2]*d[i2+1][j2] +
                        an2[i2][j2]*d[i2][j2+1]+
                        as2[i2][j2]*d[i2][j2-1];
            dth = $+(I2,J2; d[i2][j2]*h[i2][j2]);
            tau = delta0 / dth;
            par (I2,J2)
                x[i2][j2] += tau * d[i2][j2];
            par (I2,J2)
                g[i2][j2] += tau * h[i2][j2];
            minvy(h, g);
            delta1 = $+(I2,J2; g[i2][j2]*h[i2][j2]);
            beta = delta1 / delta0;
            delta0 = delta1;
            par (I2,J2)
                d[i2][j2] = - h[i2][j2] + beta*d[i2][j2];
        }
    }
```

Department of Mathematics, Texas A & M University, and Institute for Scientific Computation, College Station, TX 77843-3368
E-mail address: mishev@isc.tamu.edu

Computer Science Department, University of California at Los Angeles, Los Angeles, CA, 90024
E-mail address: austel@cs.ucla.edu

Department of Mathematics, University of California at Los Angeles, Los Angeles, CA, 90024
E-mail address: chan@math.ucla.edu

Department of Mathematics, University of California at Los Angeles, Los Angeles, CA, 90024, and Center for Computer Science and Technology, Bulgarian Academy of Sciences, Acad. G. Bontchev street, block 25 A, 1113 Sofia, Bulgaria
E-mail address: panayot@math.ucla.edu, panayot@bgearn.bitnet