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
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 $O(n \log 1/\epsilon)$ arithmetic operations (executed on sequential computer). Here $n$ is the number of unknowns and $\epsilon > 0$ 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:

\[
\begin{align*}
\begin{cases}
-\nabla \cdot (a(x) \nabla u(x)) &= f(x) \quad \text{in } \Omega, \\
u(x) &= g(x) \quad \text{on } \Gamma_1, \\
\frac{\partial u}{\partial n}(x) &= 0 \quad \text{on } \Gamma_2,
\end{cases}
\end{align*}
\]

(1)

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

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

\[ T_1, T_2, \ldots, T_l \]

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

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

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
$N_k \setminus N_{k-1}$ (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

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

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

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

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

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

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

**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., $\gamma = \max_{r \in T} \gamma_r$, 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 $A^{(k)}$:

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

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 $\gamma$ be a constant in the strengthened CBS inequality. Then
\[
1 - \gamma^2 \leq \frac{v_2^T S^{(k)} v_2}{v_2^T A^{(k-1)} v_2} \leq 1 \quad \text{for all } v_2 \in \mathbb{R}^{n_k-1}.
\]

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

Definition 2.1 (Two-level preconditioner).
\[
M^{(l)} = \begin{bmatrix} A_{11}^{(l)} & 0 \\ A_{21}^{(l)} & A^{(l-1)} \end{bmatrix} \begin{bmatrix} I & A_{12}^{(l-1)} \\ 0 & I \end{bmatrix}.
\]

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., $\text{Cond} (M^{(l-1)} A^{(l)}) = O(\delta^2) = O(\log^2 n)$, cf. e.g., Vassilevski [10], [11].

Definition 2.2 (V-cycle multilevel preconditioner).

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

and for $k = 2, 3, \ldots, l$, let
\[
M^{(k)} = \begin{bmatrix} A_{11}^{(k)} & 0 \\ A_{21}^{(k)} & M^{(k-1)} \end{bmatrix} \begin{bmatrix} I & A_{12}^{(k-1)} \\ 0 & I \end{bmatrix}.
\]

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).

Set \( M^{(1)} = A^{(1)} \)

for \( k = 2, 3, \ldots, l \)

\[
M^{(k)} = \begin{bmatrix}
A^{(k)}_{11} & 0 \\
A^{(k)}_{21} & \tilde{M}^{(k-1)}_{22}
\end{bmatrix} \begin{bmatrix}
I & A^{(k)-1}_{12} \\
0 & I
\end{bmatrix}.
\]

where

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

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., \( \tilde{M}^{(k-1)} \) 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 \( \tilde{M}^{(k-1)} = M^{(k-1)} \), 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 \( \tilde{M}^{(k-1)} \) 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 \( \tilde{M}^{(k-1)} x = y \).

**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 \( \nu \) solve

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

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, onecof 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:

```c
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,

\[
\text{par (I, J)}
\quad b[i][j] = b[j][i];
\]

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 \text{true} \) and the statement is executed in parallel for all enabled elements. The following statement assigns zero to the diagonal of array \( b \).

\[
\text{par (I, J)}
\quad \text{st (i==j)}
\quad b[i][j] = 0;
\]

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 \( b \).

\[
\text{sum} = \$+(I, J; b[i][j] \cdot b[i][j]);
\]

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 \( 2^n \)-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

\[
A_h^{(k)} y(x_{1,i}, x_{2,j}) = a c^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i}, x_{2,j}) + a w^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i-1}, x_{2,j})
\]

\[
+ a e^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i+1}, x_{2,j}) + a n^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i}, x_{2,j+1})
\]

\[
+ a s^{(k)}(x_{1,i}, x_{2,j}) y(x_{1,i}, x_{2,j-1}).
\]

For every mesh point \( (x_{1,i}, x_{2,j}) \), which is not on the boundary with Dirichlet boundary condition \( \Gamma_1 \), we store in the processor \((i, j)\) the stiffness matrix entries, \( a c[k][i][j], a w[k][i][j], a e[k][i][j], a n[k][i][j], a s[k][i][j] \). If the point \( (x_{1,i}, x_{2,j}) \) has neighbor(s) on \( \Gamma_1 \), which are south and/or west from it, then \( a s[k][i][j] \) and/or \( a w[k][i][j] \) are equal to zero. The same is valid for points on the boundary with Newman boundary
Figure 1. A finite element stencil

\[
\begin{align*}
\text{par (I,J)} \\
\text{z[i][j] = ac[i][j] * y[i][j] + aw[i][j] * y[i-1][j] + as[i][j] * y[i][j-1]} \\
\text{+ ae[i][j] * y[i+1][j] + an[i][j] * y[i][j+1]} \\
\end{align*}
\]

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

\[
\begin{align*}
\text{par (I,J)} \\
\text{st (i%2 == 0 || j%2 == 0)} \\
\text{y[i][j] = ac[i][j] * v[i][j] + aw[i][j] * v[i-1][j] + as[i][j] * v[i][j-1]} \\
\text{+ ae[i][j] * v[i+1][j] + an[i][j] * v[i][j+1] } \\
\end{align*}
\]

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 $\Gamma_1$ 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.** $u(x) = 1$, $a(x) = 1$.

**Problem 2.**

$$u(x) = e^{-x^2(x_1^2 + x_2^2)}$$

$$a(x) = 1/(1 + x_1^2 + x_2^2)$$

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

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

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

and we have chosen

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

**Problem 4 (a singular solution).**

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

The right hand side here is a delta function

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

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 $\nu$ of the polynomials $p_k(t)$ 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 multilevel 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, $h = 1/128$

<table>
<thead>
<tr>
<th>levels</th>
<th>prob.#</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG(0)</td>
<td>iter</td>
<td>401</td>
<td>570</td>
<td>11236</td>
<td>407</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>3.56s</td>
<td>5.06s</td>
<td>99.74s</td>
<td>5.35s</td>
</tr>
<tr>
<td>2 levels</td>
<td>iter</td>
<td>7</td>
<td>7</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>21.84s</td>
<td>19.84s</td>
<td>488.43s</td>
<td>19.34s</td>
</tr>
<tr>
<td>3 levels</td>
<td>iter</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>19.47s</td>
<td>21.93s</td>
<td>241.60s</td>
<td>23.36s</td>
</tr>
<tr>
<td>4 levels</td>
<td>iter</td>
<td>11</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>20.61s</td>
<td>21.59s</td>
<td>110.46s</td>
<td>24.26s</td>
</tr>
<tr>
<td>5 levels</td>
<td>iter</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>24.34s</td>
<td>24.85s</td>
<td>56.10s</td>
<td>27.53s</td>
</tr>
<tr>
<td>6 levels</td>
<td>iter</td>
<td>14</td>
<td>15</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>28.16s</td>
<td>30.17s</td>
<td>36.04s</td>
<td>34.08s</td>
</tr>
<tr>
<td>7 levels</td>
<td>iter</td>
<td>15</td>
<td>16</td>
<td>14</td>
<td>17</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>35.17s</td>
<td>35.48s</td>
<td>33.88s</td>
<td>38.71s</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>polyn. / levels</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>(11211)</td>
<td>iter</td>
<td>9</td>
<td>11</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>21.93s</td>
<td>21.59s</td>
<td>35.50s</td>
<td>41.00s</td>
</tr>
<tr>
<td>(12112)</td>
<td>iter</td>
<td>9</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>21.93s</td>
<td>33.67s</td>
<td>35.11s</td>
<td>39.20s</td>
</tr>
<tr>
<td>(21212)</td>
<td>iter</td>
<td>8</td>
<td>10</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>38.15s</td>
<td>33.07s</td>
<td>49.06s</td>
<td>54.97s</td>
</tr>
<tr>
<td>(22222)</td>
<td>iter</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>38.15s</td>
<td>53.92s</td>
<td>89.33s</td>
<td>143.35s</td>
</tr>
<tr>
<td>(11311)</td>
<td>iter</td>
<td>9</td>
<td>11</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>21.93s</td>
<td>21.59s</td>
<td>39.16s</td>
<td>48.10s</td>
</tr>
<tr>
<td>(13113)</td>
<td>iter</td>
<td>9</td>
<td>7</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>21.93s</td>
<td>40.70s</td>
<td>45.20s</td>
<td>44.26s</td>
</tr>
<tr>
<td>(31313)</td>
<td>iter</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>53.02s</td>
<td>43.01s</td>
<td>95.11s</td>
<td>89.67s</td>
</tr>
<tr>
<td>(33333)</td>
<td>iter</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>no</td>
</tr>
<tr>
<td></td>
<td>time</td>
<td>53.02s</td>
<td>112.27s</td>
<td>260.01s</td>
<td>exper.</td>
</tr>
</tbody>
</table>
REFERENCES


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:*

\[ \text{while} \ (\delta_0 > \varepsilon_1) \ { \}
\]

\[ h = Ad_i, \]
\[ \tau = \delta_0 / d_i^T h, \]
\[ x_{i+1} = x_i + \tau d_i, \]
\[ g_{i+1} = g_i + \tau h, \]
\[ \delta_i = g_i^T g_{i+1}, \]
\[ \beta = \delta_i / \delta_0, \]
\[ \delta_0 = \delta_i, \]
\[ \delta_{i+1} = -g_{i+1} + \beta d_{i+1}, \]
\[ i = i + 1, \]

#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];
    deltai = +(I,J; g[i][j]*g[i][j]);
    beta = deltai / delta0;
    delta0 = deltai;
    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 iterai1;

    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
    iterai1 = 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;
\[ \text{eps1 = eps11 * delta0; } \]

\[
\text{while(delta0 > eps1){} }
\]
\[
\text{itera11 += 1; } \]
\[
\text{par (I2,J2) } \]
\[
/*
st ( i,j ) \in \text{ fine } \) */
\[
\text{st ( i2 % 2 == 0 || j2 % 2 == 0) } \]
\[
h[i2][j2] = ac2[i2][j2]*d[i2][j2] +
aw2[i2][j2]*d[i2-1][j2] +
ae2[i2][j2]*d[i2+1][j2] +
an2[i2][j2]*d[i2][j2+1] +
as2[i2][j2]*d[i2][j2-1];
\]
\[
dth = $(I2,J2; d[i2][j2]*h[i2][j2]);
\]
\[
\text{tau = delta0 / dth; } \]
\[
\text{par (I2,J2) } \]
\[
\text{x[i2][j2] += tau * d[i2][j2]; } \]
\[
\text{par (I2,J2) } \]
\[
\text{g[i2][j2] += tau * h[i2][j2]; } \]
\[
\text{par (I2,J2) } \]
\[
/*
st ( i,j ) \in \text{ fine } \) */
\[
\text{st ( i2 % 2 == 0 || j2 % 2 == 0) } \]
\[
h[i2][j2] = g[i2][j2] / ac2[i2][j2];
\]
\[
delta1 = $(I2,J2; g[i2][j2]*h[i2][j2]);
\]
\[
\text{beta = delta1 / delta0; } \]
\[
delta0 = delta1;
\]
\[
\text{par (I2,J2) } \]
\[
d[i2][j2] = - h[i2][j2] + beta*d[i2][j2];
\]
}\]

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 \(A(\text{coarse})\) times a vector implemented without any transfer of data in the way we executed similar operation with the matrix \(A_{11}\).

\[
/* \text{ solve } Mx = y \rightarrow x = M(-1)y */\]
\[
\text{void minvy(float x[N2][N2], float y[N2][N2]) } \]
\[
\text{\{ } \]
\[
\text{float v[N2][N2], w[N2][N2], z[N2][N2], } \]
\[
x2[N1][N1], z2[N1][N1];
\]
\[
\text{par (I2,J2) } \}
\[
\text{x[i2][j2] = 0.0; } \]
\begin{verbatim}
ALGEBRAIC MULTILEVEL PRECONDITIONERS

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);
\end{verbatim}
/* 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] +
au2[i2][j2] * d[i2+1][j2] +
an2[i2][j2] * d[i2][j2+1] +
au2[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];