Intern. J. Computer Math., Vol. 44, pp. 269–290
Reprints available directly from the publisher
Photocopying permitted by license only

PARALLELIZABLE RESTARTED ITERATIVE METHODS FOR NONSYMMETRIC LINEAR SYSTEMS. II: PARALLEL IMPLEMENTATION

WAYNE D. JOUBERT and GRAHAM F. CAREY

The University of Texas at Austin, U.S.A.

(Received 9 June 1991; in final form 7 July 1991)

In Part I we defined several new algorithms which gave the same iterates as the standard GMRES algorithm but require less computational expense when implemented on scalar and vector computers. In this paper, we demonstrate that the new algorithms are even more economical than the standard algorithms when applied to several other computer architectures: machines with distributed memory, machine which use local cache memory, and machines for which it is desired to use disk memory or solid state disk memory to solve the given problem. The increased efficiency is due to a reduction in the number of memory transfers required by the algorithms. In this paper we describe how these algorithms are implemented in order to obtain the improved performance. Numerical comparisons of the algorithms are given on several representative computers.

KEY WORDS: GMRES, iterative method, linear equations, nonsymmetric, parallel, cache, Orthomin.

C.R. CATEGORIES: G.1.3 Numerical Linear Algebra, D.1.3 Concurrent Programming, J.2 Physical Sciences and Engineering.

1 INTRODUCTION

In Part I we examined the class of *restarted minimal residual algorithms* for solving the linear system

$$Au = b, \tag{1}$$

where $A \in \mathbb{C}^{N \times N}$ is nonsingular and non-Hermitian. This class includes such algorithms as the popular GCR and GMRES algorithms. We also defined several new algorithms to compute the same iterates as these algorithms, including the following:

- A Chebyshev basis algorithm which uses Chebyshev polynomials to represent the Krylov space basis for each restart cycle;

This work was supported in part by the National Science Foundation under Grant DCR-8518722, by the Department of Energy under Grant DE-FG05-87ER25048, and by Cray Research Inc. under Grant LTRDTD 1/18/90, with the University of Texas at Austin.

- A GMRES/Chebyshev basis algorithm which uses GMRES for the first cycle to compute eigenvalue estimates in order to apply the Chebyshev basis algorithm for subsequent cycles; and
- A GMRES/repeated basis algorithm for which GMRES is used for the first cycle and the resulting vector recursion coefficients are used to generate the basis for following cycles.

The two algorithms which use Chebyshev bases require as little as half the work of the standard GMRES algorithm. These algorithms also offer added benefits for alternate computer architectures. This is because the standard GCR and GMRES algorithms utilize Gram Schmidt orthogonalization procedures which combine global and local operations within each iteration. Specifically, global inner products are done at each iteration, along with the local matrix-vector product operation and SAXPY operations (i.e., operations of the form $y \leftarrow y + \alpha x$, where x and y are vectors and α is a scalar). In comparison, the new algorithms do not require inner products at each iteration, but instead the inner products are performed together at the end of the restart cycle.

This feature of the new algorithms impacts several computer architectures. First, for shared memory multiprocessor machines, the number of synchronization points per iteration is reduced. Secondly, and more importantly, for machines with distributed memory, the number of global communications is decreased, which is beneficial if the startup time for a global communication is significant. Finally, for machines with cache memory (and analogously machines with solid state disk or for which out-of-core solution of the linear equations is desired), the number of times per iteration for which it is necessary to bring the vectors and matrix into cache is reduced, i.e. more work can be done without accessing main memory.

The problem of synchronization points for the conjugate gradient method is examined in [Seager 1986] and [Saad/Schultz 1985b]. The possibility of obtaining greater parallel efficiency by performing several matrix-vector products in sequence is suggested in [Saad/Schultz 1985a], and an algorithm for cache memory machines is presented in [Chronopoulos 1986] and [Chronopoulos/Gear 1989a, 1989b] which is based on unrolling the conjugate gradient loop to reduce the frequency of inner product calculations to every s iterations, for some $s \geq 1$. These ideas are extended to other conjugate gradient-type methods in, for example, [Chronopoulos 1989], [Chronopoulos/Kim 1990]. Loop unrolling for iterative methods is considered also in [Joubert 1990b] in relation to hybrid truncated/restarted methods for nonsymmetric systems. The algorithms set forth in the current paper rely on a similar approach but differ in important ways. First, the algorithms of [Chronopoulos/Kim 1990] rely on the power basis, which is often ill-conditioned; on the other hand, the more well-conditioned Chebyshev basis is used here, giving the opportunity to iterate for more steps before orthogonalization is needed, thus increasing efficiency. Secondly, the use of a pseudo-inverse at strategic parts of the new algorithms in this paper guarantees that the algorithms cannot break down.

The use of alternate choices of the basis allows a further economizing modification to be made, which applies to many sparse matrix problems. For sparse matrices which arise from the discretization of partial differential equations over a physical domain, it is typical to subdivide the physical grid into subdomains in order to map the problem to a distributed memory machine. Similarly, for cache memory machines it is possible to subdivide the problem into sections which each fit into cache memory. In this paper we set forth a technique which allows several local matrix-vector products to be performed without the need for interprocessor communication (for distributed memory machines) or main memory access (for cache memory machines). This technique is based on increasing the amount of data communicated locally between subdomains at each instance, so that fewer communications are needed.

The remainder of this paper is arranged as follows. In Section 2 we set forth the implementation of the algorithms on cache memory machines. Then in Section 3 we give the implementation for distributed memory machines. Finally, in Section 4 we give the results of numerical experiments on several different computers.

2 IMPLEMENTATION ON CACHE MEMORY MACHINES

The basic assumption for cache memory machines is that each processor has a relatively small amount of memory which can be accessed by the CPU much faster than the larger main memory of the processor. Some examples of machines which fit this model include the Alliant FX/8, the Intel iPSC/860 processor, the IBM 3090, and the IBM RS/6000 series machine. Machines such as the Cray Y/MP with solid state disk may also be considered to fit this model, and furthermore for nearly any machine the secondary disk memory may be considered the "main memory" for solving a given problem, and the main semiconductor memory may be considered the "cache memory". We will not consider here architectures with several levels of cache memory, although the present work can be applied here also to advantage. We also point out that the trend towards computers with cache memory may be expected to increase, as CPU clock speeds continue to increase while memory address spaces also increase.

The basic goal in programming cache memory machines is to reduce the access to the slower main memory. Means for doing more work in cache without going to main memory are discussed below, for several cases based on the size of the cache memory in comparison to the problem size.

2.1 *Implementations Categorized in Terms of Cache Size*

Many different considerations influence the choice of the means of solution of a particular problem, including the problem size, matrix storage format, preconditioner, and restart frequency, for example. Based on such details, each particular case can be custom optimized. We will not cover every possible situation here, but instead will discuss several of the more important cases for the algorithms of interest.

We recall from Part I that the vector work for a cycle of GMRES is composed of the following: the Arnoldi basis generation phase (matrix-vector product, preconditioning, inner products, and SAXPY operations per step), and then the update of the approximate solution and residual (SAXPY operations). On the other hand, the new algorithms involve: the basis generation phase (matrix-vector product, preconditioning, and SAXPY operations per step), the inner product computations, the update of the approximate solution and residual (SAXPY operations), and then the computation of the residual norm for the stopping test.

We now discuss the implementation for several cases:

Case 1: All data in cache. This is the trivial case. If the program, matrix, preconditioner, solution, residual, temporary vectors, coefficients, and all basis vectors can be stored in cache, then the entire problem can be solved in cache memory without any access to main memory.

Case 2: Matrix and preconditioner in cache. In this case, the cache can hold the matrix, preconditioner and also several (two to four) vectors. For this case we examine each of the several phases of the algorithms.

- In the basis generation, for GMRES at each step it is necessary to retrieve from main memory all previous basis vectors once, to perform the needed inner product and SAXPY for each vector. Then it is necessary to store the new basis vector. For the GMRES/repeated basis algorithm, the same basic scheme may be used, omitting the inner products. However, for the Chebyshev basis algorithms, only three vectors are used in the recursion, which can be kept in cache, so at each basis generation step it is necessary only to store the new basis vector.

- In the inner product calculation $Q_{k+1}^{*}Q_{k+1}$ for the new algorithms (see Part I), the following scheme may be used. To compute the inner products, the first n_{ptsub} entries of each of the $q^{(i)}$'s may be brought into cache and the local mutual inner product contributions may be accumulated, and then the same for the next n_{ptsub} elements of each vector, and so forth. Here we define n_{sub} and n_{ptsub} to be positive integers such that $N = n_{\text{sub}}n_{\text{ptsub}}$, where each vector is split into n_{sub} subvectors or subdomains, of n_{ptsub} elements each. For sufficiently small n_{ptsub} this procedure is effective and requires that each vector be brought into cache only once, though possibly in pieces.

- The solution and residual updates may be performed for all algorithms by bringing the solution iterate and residual into cache, bringing the basis vectors into cache one at a time for the updates, and then storing the solution and residual to main memory.

Case 3: Several vectors in cache. If only two to four vectors can be stored in cache, then it is necessary to bring the matrix and preconditioner into cache once for every basis generation step. Other than this, the different algorithms compare to each other in the same way as in Case 2 above, leading again to an advantage for the Chebyshev basis algorithm.

Case 4: No vectors in cache, general case. If the problem size is so large that a single vector cannot be stored in cache and if the matrix has an arbitrary sparsity pattern, then all algorithms may require a rather large number and volume of accesses to main memory. For this case we let all vectors be divided into n_{sub} equal pieces of size n_{ptsub}, as above. The different phases of the algorithms may then be arranged as follows:

- For the matrix-vector product, each subvector of the result is accumulated in sequence. If the matrix has an arbitrary sparsity pattern, then this may require the source vector to be read into cache as many as n_{sub} times, in subvector pieces. Similarly, the result vector must be stored in main memory once, in subvector pieces. If a preconditioner is present, then the work may be further increased.

- For the remainder of the calculation of each basis vector, we assume here that it is possible to store in cache one subvector from the basis vectors, solution and residual, as well as local matrix and preconditioner information for this subvector, all at the same time. Then, for GMRES, it is necessary to bring all previous basis vectors into cache twice per step: once to form the local matrix-vector product and accumulate the local inner products, and once to perform the SAXPYs. On the other hand, the new algorithms require the previous vectors (two for Chebyshev and all previous vectors for the repeated basis algorithm) to be brought into cache only once. In either case, the result vector must be stored in main memory once, in pieces.

- For the remainder of the algorithm, the principles of Case 2 given above apply directly, though the transfer of vectors to and from cache must necessarily occur by subvector pieces.

Case 5: No vectors in cache, banded matrix. We will consider this case in greater detail than the others. The case is the same in all respects to Case 4, except that the matrix is assumed to be banded, with half-bandwidth n_b satisfying $n_b \leq n_{\text{ptsub}}$ where $N = n_{\text{sub}} n_{\text{ptsub}}$ and $2n_b + 1$ is the full bandwidth. We also assume here that the preconditioner is such that in order to obtain a preconditioned vector on a subdomain, it is necessary to have the source vector *only* on that subdomain. This is true, for example, of appropriate cases of line or block Jacobi preconditioning or incomplete factorizations on subdomains (see e.g. [Hageman/Young 1981], [di Brozolo/Robert 1987]). If this condition is not satisfied, e.g. when a global incomplete factorization is used, then one or two additional passes of a vector into cache for each basis vector generation step may be necessary.

Importantly, to obtain a matrix-vector product defined on a subdomain in this case requires only the source vector on that subdomain and also on the two nearest neighbor subdomains, since the matrix is block tridiagonal, with blocks of size n_{ptsub}. If the matrix-vector product pieces are computed in the natural sequential order, then the neighbor subvectors can be arranged to be in cache at the right time without added expense, if extra space is available in cache for them. Besides this, the details of the implementation are the same as in Case 4 above. In particular, for a basis generation step of GMRES, it is necessary to read all previous vectors into cache twice (once for the local matrix-vector products and local inner products, and once for the SAXPYs), but for the new algorithms it is necessary to read the needed previous vectors only once.

This case commonly arises in physical applications. For many problems, e.g. 2- and 3-dimensional problems with an imposed natural lexicographical ordering of the grid points, the matrix dimension grows much faster than the bandwidth, and it is practical to use the scheme described here, even for large problem sizes. For such two-dimensional problems, for example, the subdomains correspond to "strips" or groups of adjacent lines of the domain, and the bandwidth is bounded by a small integer multiple of the largest number of grid points in a line of the grid. Furthermore, the speed improvement from using the new algorithms will continue to be a fixed factor of improvement, even for very large problems (as long as the criteria as specified above for this case are satisfied), based in large part on the relative speeds of the main and cache memory.

We now show how the new algorithms implemented in this scheme allow for the possibility of computing several local matrix-vector products on a subdomain, totally in cache and without any access to main memory.

2.2 Subdomain Overlapping Techniques

For the case of a banded matrix described above, in order to obtain a matrix-vector product Av on a subdomain, it is necessary to have the value of the source vector v on that subdomain as well as the two adjacent neighbors. More generally, we may ask: to complete *several* matrix-vector products, $\{A^i v\}$ on a subdomain, what elements of the original source vector v are necessary? The answer to this question depends on the sparsity pattern of the given matrix A. For matrices derived from finite difference or finite element discretization, the answer is based directly on the type of stencil used for the discretization. In what follows we set up the formal notation for dealing with this, and then illustrate the approach by using a simple model problem.

We let the set of points $\{i\}_{i=1}^N$ be partitioned into subdomains denoted by $\{I_m\}$, $I_m \subseteq \{i\}_{i=1}^N$. Now, for I any subset of $\{i\}_{i=1}^N$, let $S_A(I) = \{j : \exists\, i \in I : e_i^* A e_j \neq 0\}$. This function extracts sparsity information from A: $S_A(\mathrm{I})$ denotes the set of elements of v which are necessary in order to compute Av on I. It is clear that in order to compute $A^i v$ on subdomain m, i.e. the elements of $A^i v$ denoted by the index set I_m, it is necessary to have the entries of v enumerated by the set $S_{A^i}(I_m)$.

We show now that $S_{A^i}(I) \subseteq S_A^i(I)$, where S_A^i denotes the composition of i copies of the function S_A. We have

$$\begin{aligned} S_{A^i}(I) &= \{j : \exists\, l \in I : e_l^* A^{i-1} \cdot A e_j \neq 0\} \\ &= \{j : \exists\, l \in I : \sum_{k \in S_{A^{i-1}}(I)} (e_l^* A^{i-1} e_k) e_k^* A e_j \neq 0\}. \end{aligned} \tag{2}$$

But $\sum_{k \in S_{A^{i-1}}(I)} (e_l^* A^{i-1} e_k) e_k^* A e_j \neq 0$ for $l \in I$ implies that for some $k \in S_{A^{i-1}}(I)$, $e_k^* A e_j \neq 0$. Thus

$$S_{A^i}(I) \subseteq \{j : \exists\, k \in S_{A^{i-1}}(I) : e_k^* A e_j \neq 0\} = S_A(S_{A^{i-1}}(I)). \tag{3}$$

The final result is shown by induction.

Thus, given a domain partitioning, it is possible to use the sparsity pattern information of A in order to determine the values of v needed to form $A^i v$ for a sequence of values of i. In particular, we have the following procedure to compute $\{A^j v\}_{j=1}^{n_{lap}}$ on subdomain m, given $n_{\mathrm{lap}} \geq 1$, the number of matrix-vector products desired:.

1) Determine $S_A^i(I_m)$, $1 \leq i \leq n_{\mathrm{lap}}$.

2) Obtain v on $\bigcup_{i=1}^{n_{\mathrm{lap}}} S_A^i(I_m)$.

3) For $1 \le j \le n_{\mathrm{lap}}$: compute the components $(A^j v)_i$, $i \in \bigcup_{l=0}^{n_{\mathrm{lap}}-j} S_A^l(I_m)$.

Step 2 requires an access to main memory for cache memory machines, or an interprocessor communication for distributed memory machines. Thus, by increasing the value of n_{lap}, the number of these operations is reduced. The cost of this is a slight increase in the amount of computation per subdomain, since for fixed j the regions $\{S_A^j(I_m)\}_m$ overlap each other, whereas the subdomains $\{I_m\}_m$ do not.

For Case 5 described above, we have $I_m = \{i\}_{i=n_{\mathrm{ptsub}}(m-1)+1}^{n_{\mathrm{ptsub}}m}$, and the half bandwidth n_b of A satisfies $n_b \le n_{\mathrm{ptsub}}$. Since $S_A^j(I_m) \subseteq \{i\}_{i=n_{\mathrm{ptsub}}(m-1)+1-j\cdot n_b}^{n_{\mathrm{ptsub}}m+j\cdot n_b}$, to limit data exchange to be only with immediate subdomain nearest neighbors, it is sufficient to limit $n_{\mathrm{lap}} \cdot n_b \le n_{\mathrm{ptsub}}$, which implies $A^{n_{\mathrm{lap}}}$ has half bandwidth bounded by n_{ptsub}.

The actual value of $S_A^i(I_m)$ depends on the sparsity pattern of A. In practice it may require some effort to compute this set. For the model problem described in Part I, the evaluation of this set is straightforward, as we now show. The model problem arises from applying central finite differencing to

$$-u_{xx}(x, y) - u_{yy}(x, y) + Cu_x(x, y) = g(x, y) \quad \text{on } \Omega = [0, 1]^2, \tag{4}$$
$$u(x, y) = 1 + xy \quad \text{on } \partial\Omega.$$

Here we use uniform mesh spacing h_x and h_y in the x and y directions. This yields a sparse matrix of size $N = (n_x - 1)(n_y - 1)$ (where $h_x = 1/n_x$, $h_y = 1/n_y$), after boundary points have been eliminated.

Suppose $I \subseteq \{i\}_{i=1}^N$ represents a rectangular subdomain within the grid. Then the shape of the 5-point stencil determines the sets of nodes contained in $S_A^i(I)$ as illustrated in Figure 1.

To map this problem to the cache memory machine according to Case 5 as described above, we let n_{ptsub} be a multiple of $(n_x - 1)$, so that each subdomain has a certain number of full x-lines.

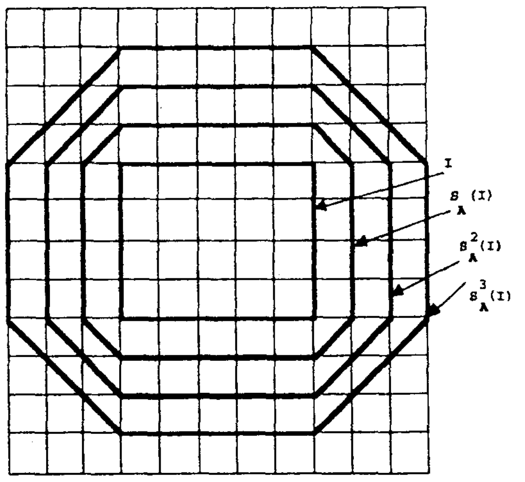

Figure 1. Vector Element Dependence for Powers of A on a Subdomain for the Model Problem.

The subdomain overlap technique can be applied to irregular grids and to other stencils and can be applied as well to arbitrary sparse matrices, with an efficiency depending on the sparsity pattern of A. This technique may also be applied in conjunction with Jacobi preconditioning, line Jacobi, or with incomplete factorizations or other preconditioners applied to appropriate interiors of the subdomains. It is not clear that this overlapping approach could be applied to incomplete factorizations over the entire domain.

2.3. Summary of the Algorithms

To sum up what we have discussed in this section, we now briefly describe in outline form the arrangement of the algorithms for Case 5 described above, utilizing the subdomain overlapping technique.

First, for the Chebyshev basis and repeated basis algorithms (see Part I), we have the following:

```
For $i = 1$ to $k$ by $n_{\text{lap}}$
  For $j = 1$ to $n_{\text{sub}}$
    Get needed block-rows of $A$ and also block-row $j$ of needed previous $\{q^{(l)}\}_l$
    Form block-row $j$ of each of $\{q^{(l)}\}_{l=i}^{i+n_{\text{lap}}-1}$, plus overlaps in block-row
    $j+1$
    If $k = n_{\text{lap}}$ then get mutual inner products of block-row $j$ of all $\{q^{(l)}\}_l$
  Next $j$
Next $i$
If $k \neq n_{\text{lap}}$ then
  For $j = 1$ to $n_{\text{sub}}$
    Get block-row $j$ of all $\{q^{(l)}\}_l$
    Accumulate mutual inner products of block-row $j$ of all $\{q^{(l)}\}_l$
  Next $j$
For $j = 1$ to $n_{\text{sub}}$
  Get block-row $j$ of $u^{(0)}$, $r^{(0)}$, all $\{q^{(l)}\}_l$
  Form block-row $j$ of $u^{(k)}$, $r^{(k)}$
Next $j$
```

The basic structure of the GMRES implementation is as follows:

```
For $i = 1$ to $k$
  For $j = 1$ to $n_{\text{sub}}$
    Get needed block-rows of $A$ and also block-row $j$ of previous $\{q(l)\}_l$
    Form block-row $j$ of $Aq^{(i-1)}$
    Accumulate inner products of block-row $j$ of $Aq^{(i-1)}$ with previous $\{q^{(l)}\}_l$
  Next $j$
  For $j = 1$ to $n_{\text{sub}}$
    Get block-row $j$ of $Aq^{(i-1)}$ and also block-row $j$ of previous $\{q^{(l)}\}_l$
    Form block-row $j$ of $h_{i+1,i}q^{(i)}$
    Form norm of block-row $j$ of $h_{i+1,i}q^{(i)}$
  Next $j$
Next $i$
```

```
For $j = 1$ to $n_{\text{sub}}$
  Form norm of block-row $j$ of $h_{k+1,k}q^{(k)}$
Next $j$
For $j = 1$ to $n_{\text{sub}}$
  Get block-row $j$ of $u^{(0)}$, $r^{(0)}$, all $\{q^{(l)}\}_l$
  Form block-row $j$ of $u^{(k)}$, $r^{(k)}$
Next $j$
```

3 IMPLEMENTATION ON DISTRIBUTED MEMORY MACHINES

The basic assumption for distributed memory machines is that many processors exist, each having its own memory, and that interprocessor communication is possible which is significantly slower than local memory access. Typical examples of the architecture are the Ametek Symult, the Ncube 2, the Intel Hypercube, and the Connection Machine.

A very desirable goal for these machines is to reduce the number and amount of interprocessor communication. To decrease the average amount of communication made per matrix-vector product, a common and effective technique is to use polynomial preconditioning techniques or Chebyshev acceleration methods (see e.g. [Ashby/Manteuffel/Saylor 1989], [Manteuffel 1978]. However, these methods typically require a greater number of matrix-vector products in order to converge to a given stopping criterion. On the other hand, for the current study we only consider algorithms which produce the identical iterates to those of GMRES, at the least possible expense.

The new algorithms reduce the *number* or frequency of global communications required to form the global inner products. This is because all inner products for a restart cycle (except for the residual norm calculation) are done together at the end of the basis generation phase, rather than concurrently with the basis generation. Furthermore, the identical subdomain overlapping ideas described in Section 2 may be used to decrease the frequency of local communications as well.

For machines with distributed memory, the cost of a communication startup is typically a significant factor, though not always. For example, the software of the Connection Machine (i.e. the Paris library) does not have a special primitive operation for forming several global inner products simultaneously. Thus, forming all inner products together has no added advantage, though nonetheless the Chebyshev basis algorithms run faster than standard GMRES, for the same reason as they do on a scalar machine.

The improvement gained by the new algorithms depend on the relative overhead cost of a communication startup. The improvement for large problem sizes and large numbers of processors can be viewed in several ways, depending on the relative rate of growth of the problem size and the number of processors. First, if the number of processors is fixed and the problem size in increased, the startup time for communications will eventually be dominated by the actual amount of computation and communication. On the other hand, if a problem of fixed size is distributed to an increasingly large number of processors, eventually the communication startup time will be significant and the new algorithms will be of greater benefit.

We now discuss the mapping of the algorithms to the distributed memory architectures. Each vector is subdivided into a fixed set of pieces which are distributed to the processors. Simple operations like SAXPYs of vectors require no communication. An inner product of vectors requires the accumulation of local inner products then the global summing of local contributions to form the global inner product.

For a matrix with an arbitrary sparsity pattern, performing a matrix-vector product may require a large amount of global communication, for any conceivable distribution scheme, since each target subvector may depend on all source subvectors. On the other hand, if the matrix is banded and if a ring can be embedded into the interconnection topology of the processors, then a straightforward distribution of the vector elements exists for which the matrix-vector product requires only local near-neighbor communication.

We now consider specifically the model problem defined earlier. We subdivide the grid into subdomains, $n_{\text{sub}x}$ in the x direction and $n_{\text{sub}y}$ in the y direction, each of size $n_{\text{ptsub}x}$ by $n_{\text{ptsub}y}$. Thus $(n_x - 1) = n_{\text{sub}x} n_{\text{ptsub}x}$ and $(n_y - 1) = n_{\text{sub}y} n_{\text{ptsub}y}$. This embeds naturally into a 2-dimensional processor grid or torus or a hypercube architecture in such a way that nearest neighbor relations are preserved.

The algorithms map naturally to these architectures with this embedding. Matrix-vector products require communication from the local nearest neighbors. The identical subdomain overlapping ideas described in Section 2 can also be applied directly to this case, to decrease the frequency of nearest-neighbor communications. As before, local per-subdomain preconditioning is possible but global incomplete factorizations require more communication.

We now give rough sketches of the algorithms. The following summarize the action performed by each processor of the computer for the respective algorithms.

With the Chebyshev basis and repeated basis algorithms, we have:

```
For $i = 1$ to $k$ by $n_{lap}$
  Get needed nearest-neighbor values of needed previous $\{q^{(l)}\}_l$
  Form $\{q^{(l)}\}_{l=i}^{i+n_{lap}-1}$ on this subdomain
Next $i$
Form local mutual inner products of all $\{q^{(l)}\}$
Compute global mutual inner products of all $\{q^{(l)}\}$
Form $u^{(k)}$, $r^{(k)}$ on this subdomain
```

The basic structure of the GMRES implementation is as follows:

```
For $i = 1$ to $k$
  Form $Aq^{(i-1)}$ on this subdomain
  Form local and global inner products of $Aq^{(i-1)}$ with previous $\{q^{(l)}\}_l$
  Form $h_{i+1,i}q^{(i)}$ on this subdomain
  Form local and global inner product of $h_{i+1,i}q^{(i)}$
Next $i$
Form $u^{(k)}$, $r^{(k)}$ on this subdomain
```

4 EXPERIMENTAL RESULTS

We now present the results of numerical experiments with these algorithms on several computers. We use the model problem described above and also described in detail in Part I.

The following assumptions are made for these runs: we utilize the initial guess vector of $u^{(0)} = 0$; for the sake of simplicity, we use the simple stopping test

$$\frac{\| r^{(n)} \|}{\| r^{(0)} \|} < \zeta$$

where appropriate, with the value of ζ specified for the particular problem; no preconditioning is used for these problems. We use the variable ϵ_M to denote the machine precision, the smallest positive floating point number such that $1 + \epsilon_M > 1$.

The matrix A, which is derived from the 5-point finite difference stencil for this model problem with natural ordering of the nodes, is zero in all entries except for 5 nonzero diagonals, which are stored compactly as such. The usual Hadamard product technique is used to perform the matrix-vector product: an elementwise multiplication of the source vector and each diagonal is performed and accumulated in the result vector, with appropriate shifts of the vectors applied, based on the position of the diagonal in the matrix. Also, for the application of a cycle of the Chebyshev basis algorithm, it will be noted that one step of the computation is the multiplication of $(A - cI)$ by a vector. By explicitly forming the matrix $(A - cI)$, it is possible to eliminate k SAXPY operations from each restart cycle. This simplification is made for all runs except for the out-of-core solution case.

We now discuss the numerical performance on a Sun workstation, Cray Y-MP, Alliant FX/8, Ametek Symult and Ncube 2 in sequence.

4.1 Sun Workstation Experiments

We first consider out-of-core and in-core solutions on a Sun workstation. Specifically, the workstation is a Sun 4/330 Sparcstation with 25 Mhz clock speed and 24 Mbytes of main memory. The values of ϵ_M are $0.59605E - 7$ (single precision) and $0.11102E - 15$ (double precision). In all cases, the standard Sun f77 compiler is used, with the optimization flag set.

The implementation of the algorithms for in-core solution is straightforward. We now describe the details of the out-of-core implementation. As described in Section 2, the matrix is seen as a block matrix with blocks corresponding to groups of x-lines of the grid. Specifically, we let n_{ptsuby} denote the number of x-lines per subdomain, so that there are $n_{\text{ptsuby}}(n_x - 1)$ elements per block-row. Disk files are used for $u^{(i)}$, $r^{(i)}$, the nonzero diagonals of A, and $\{q^{(i)}\}$. For $u^{(i)}$, $r^{(i)}$ and A, each block-row is stored as a separate record. For $\{q^{(i)}\}$, the vectors are similarly divided by block-rows, and are also grouped in groups of n_{lap} vectors; e.g. each block-row of $\{q^{(i)}\}_{i=1}^{n_{\text{lap}}}$ is stored as a single disk record, and similarly for $\{q^{(i)}\}_{i=n_{\text{lap}}+1}^{2n_{\text{lap}}}$, and so forth. In addition, for the Chebyshev basis and power basis algorithms, the last two vectors of each packet of n_{lap} vector pieces are stored on disk separately, so that to generate each n_{lap} cycle

Table 1 Sun 4/330 workstation performance

Algorithm\Problem	*CPU secs*	*Mflops/CPU sec*	*Real time secs*	*Mflops/RT sec*
GMRES, $n_{\text{lap}} = 1, n_{\text{sub}} = 10$	786.49	0.299	2916.60	0.081
Cheb, $n_{\text{lap}} = 1, n_{\text{sub}} = 10$	264.35	0.584	532.25	0.290
Cheb, $n_{\text{lap}} = 10, n_{\text{sub}} = 10$	178.53	0.883	481.92	0.327
Cheb, $n_{\text{lap}} = 30, n_{\text{sub}} = 10$	161.48	1.023	471.03	0.351
GMRES, $n_{\text{lap}} = 1, n_{\text{sub}} = 1$	17.95	1.310	17.96	1.309
Cheb, $n_{\text{lap}} = 1, n_{\text{sub}} = 1$	10.91	1.358	10.91	1.358

of basis vectors, only these two vectors need to be read in, rather than the entire previous packet of n_{lap} vectors. For the GMRES algorithm, we assume $n_{\text{lap}} = 1$. The standard Fortran unformatted record direct access read/write operations are used, with record sizes being multiples of 8K bytes.

In Table 1 we give results for one cycle of length $k = 30$. Double precision arithmetic is used, and we let $(n_x - 1) = n_{\text{ptsuby}} = 100$ and $Ch_x = 4$ The variable n_{sub} which controls the number of subdomains is set either to 1 (in-core) or 10 (out-of-core). The variable n_{lap} described above defines the frequency at which the matrix is read off disk. It should be noted that for the out-of-core case, the problem is too large to solve in-core on this machine by any restarted method using this restart frequency.

The runs are performed in a single-user environment. Timings and associated Mflop rates are given based on both the CPU time and real time.

Both the CPU time and real time results from these runs are significant. In a multi-user timesharing environment, the CPU timings are of special importance since the user's job is mostly inactive when disk reads and writes are being performed. On the other hand, in a single user environment the real time is more significant. In either case, the above runs show that the Chebyshev basis algorithm gives dramatically reduced timings compared to the standard GMRES algorithm—in either case up to approximately 6 times faster than GMRES. These results indicate that by use of the new algorithms, the cost of out-of-core solution begins to look much more like the cost of in-core solution. Thus, larger problem sizes become more feasible for solution on workstations.

4.2 Cray Y-MP Experiments

The University of Texas System Cray Y-MP 8/864 vector computer is used here. These runs are performed in dedicated single-processor mode. The values of ϵ_M are $0.71054E - 14$ (single-precision) and $0.25244E - 28$ (double precision).

The algorithm implementations for in-core and out-of-core solution are basically the same as for the Sun workstation. The cft77 compiler is used, with the do-loops vectorized automatically by the compiler. For the disk input/output for the out-of-core solution, the fast system routines OPENDR, CLOSDR, READDR and WRITDR routines are used; the feature for overlapping disk i/o with computation with these routines is not used, and it is expected that if it were then the timings would be improved.

Table 2 Cray Y-MP performance

Algorithm\Problem	*CPU secs*	*Mflops/CPU sec*	*Real time secs*	*Mflops/RT sec*
GMRES, $n_{\text{lap}} = 1$, $n_{\text{sub}} = 20$	9.51	248.21	2382.02	0.99
Cheb, $n_{\text{lap}} = 1$, $n_{\text{sub}} = 20$	6.45	240.15	581.84	2.66
Cheb, $n_{\text{lap}} = 10$, $n_{\text{sub}} = 20$	6.87	236.16	205.74	7.89
Cheb, $n_{\text{lap}} = 30$, $n_{\text{sub}} = 20$	7.78	229.11	128.60	13.89
GMRES, $n_{\text{lap}} = 1$, $n_{\text{sub}} = 1$	0.455	259.42	0.455	259.17
Cheb, $n_{\text{lap}} = 1$, $n_{\text{sub}} = 1$	0.306	243.33	0.306	243.20

For the runs below, we let $(n_x - 1) = 1000$, $n_{\text{ptsuby}} = 50$, $Ch_x = 4$, $k = 30$ and $n_{\text{sub}} = 1$ or 20. The out-of-core case requires roughly 38 Mwords of total storage. We again give the results for 1 restart cycle. Single precision is used. The Cray run here is in uniprocessor mode, but similar results can be expected in multiprocessor mode.

These results show a dramatic improvement in the real time utilization for the Chebyshev basis algorithm—improvement up to a factor of 14 in the real time Mflop rate. In all cases the number of Mflops per CPU second is good, due to the low overhead of the Cray input/output routines, and since the CPU timer is not active during input/output requests. The CPU-time Mflop rates for the Chebyshev basis algorithm are less than for GMRES because the former has a higher relative proportion of the Hadamard product operations, which are slower since they make use of three vectors instead of two.

It is expected that good results would also be obtained if the solid state disk were used for secondary storage.

4.3 *Alliant FX/8 Experiments*

The Alliant FX/8 computer of the Texas Institute for Computational Mechanics at the University of Texas at Austin is used for experiments to test the use of a local cache. This is an 8-processor shared memory machine with 64 Mbytes main memory and a 512-Kbyte cache memory which is available to the 8 processors. The data transfer rates for cache to main memory and cache to a processor are 188 and 376 Mbytes per second, respectively. Each processor or computational element is compatible with the Motorola MC68000 architecture and has floating point and vector capabilities. IEEE standard 32-bit (single precision) and 64-bit (double precision) arithmetic are employed. The values of ϵ_M are $0.59605E - 7$ (single precision) and $0.11103E - 15$ (double precision).

For these runs we basically assume Case 2 described in Section 2 above. Some modification is necessary since the Alliant is a multiprocessor machine, though the experiments we give below utilize one processor only. For this case, vectors are divided into equal-length pieces for each processor. Local inner products are accumulated separately for each processor and then combined. For the computation of all $(k + 1)(k + 2)/2$ inner products for algorithms such as the Chebyshev basis algorithm, the following scheme is used to reduce the amount of throughput to cache memory. A variable n_{cache} is defined, and for each subvector associated with a processor, the basis vectors are further subdivided into subvectors of length n_{cache}. Then, in sequence all mutual inner products of basis vectors associated with each piece are computed and accumulated. By this scheme, if n_{cache} is sufficiently small, then every element of every vector passes through the cache only once, instead of a separate time for each mutual inner product. At the same time, n_{cache} must be large enough to take advantage of the vector capability of the Alliant processors.

Since the Chebyshev basis algorithm requires only two previous vectors to compute each new vector, unlike GMRES, the Chebyshev basis algorithm should make better use of cache in the vector generation phase, if the matrix and three vectors can be fitted into the cache at the same time. For problems which are even larger, the subdomain overlapping scheme described previously in Section 2 may be used to improve the use of cache while slightly increasing the number of arithmetic operations.

The machine specifications mentioned above indicate that cache access on the Alliant is about twice as fast as main memory access. On the other hand, for the straightforward implementation of the GMRES algorithm, if the cache is large enough to hold two entire vectors at once, then most of the SAXPY, dot product and Hadamard product operations of the GMRES algorithm require only one out of the 2 or 3 required vectors to be brought in from main memory. Thus GMRES itself makes some utilization of cache memory. Furthermore, the memory access operations are not the only time-consuming elements of the SAXPY, dot product and Hadamard product operations. It can be expected from this that the improvement in Mflop rate for the new algorithms over GMRES should be significantly less than a factor of two.

For these runs we let $n_x = n_y = 181$ and $Ch_x = 4$, and we give results for one restart cycle of length k. For simplicity, one processor is used, with double precision

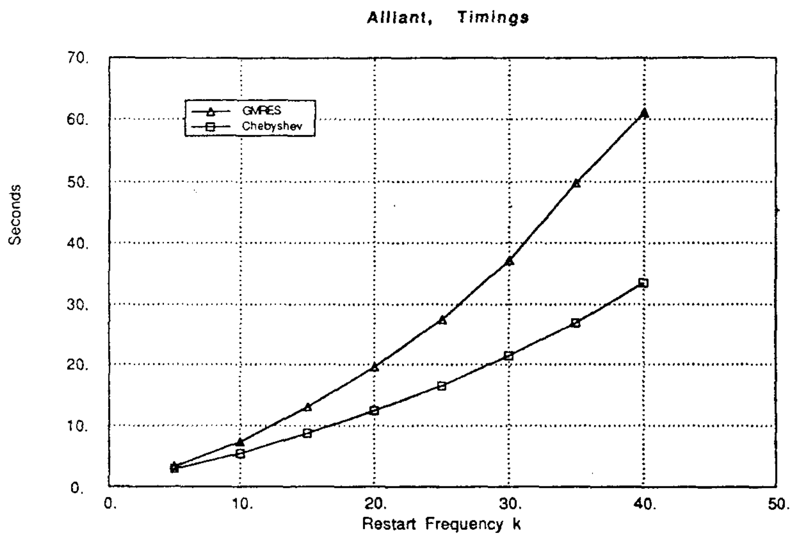

Figure 2.

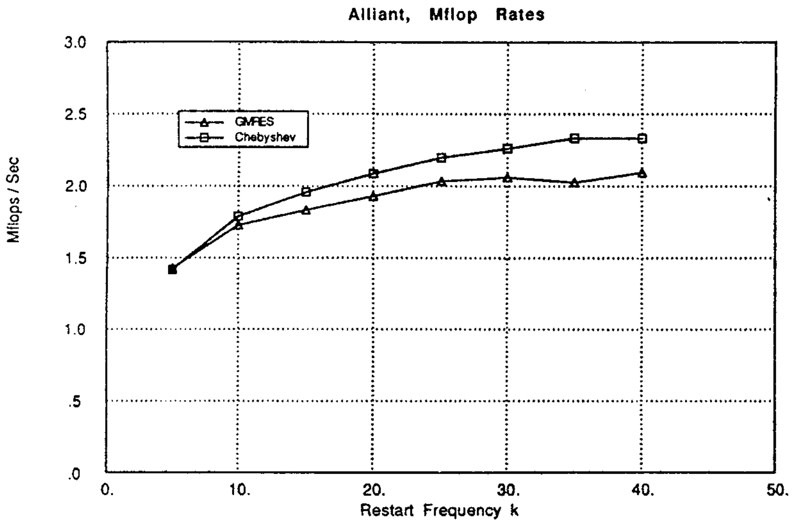

Figure 3.

arithmetic. We use $n_{\text{cache}} = 300$ as described above for the inner product computations.

These runs show an improved utilization of the cache memory by the Chebyshev basis algorithm over the GMRES algorithm. As mentioned earlier, the Chebyshev basis algorithm is richer in the more expensive matrix-vector product operations which require three vectors for each floating point vector operation, so normalizing according to this factor would increase the relative strength of the Chebyshev basis algorithm.

4.4 Ametek Symult Experiments

As an example of a distributed memory machine, the Ametek Symult 2010 computer of the Computer Science Department of the University of Texas at Austin is used. This machine has 24 processors linked in a two-dimensional toroidal array of size 4×6. Each processor is a 25 Mhz Motorola 68020 32-bit processor equipped with a 68881 floating point coprocessor. All processors have 1 Mbyte of memory each, except for the 8 lower processors which have 8 Mbytes each. A Sun 3 front end computer is used. The values of ϵ_M are $0.59605E-7$ (single precision) and $0.11108E-15$ (double precision).

The physical domain for the model problem is mapped to the processors in such a way that when all 24 processors are used, $n_{\text{sub}x} = 4$ and $n_{\text{sub}y} = 6$. The Fortran f77s2010 compiler is used to compile the code with the optimization flag selected. The standard routines FSEND and FRECV are used for interprocessor communication. Global inner products are computed by sending the local accumulations to processor 0, adding them, and then distributing them to all the processors. Similar results are obtained if the global inner products are formed using spread operations, which involve a sequence of send-and-add operations in the x and y directions.

We give experimental results of two types: either the problem size is fixed and the number of processors is allowed to increase, or the number of vector elements per subdomain is fixed and the number of subdomains (processors) is increased with the problem size.

First we fix $(n_x - 1) = 60$ and $(n_y - 1) = 30$. We let $Ch_x = 4$ and $k = 5$ and converge to $\zeta = 10^{-6}$. Double precision is used. The subdomains are mapped to the processors lexicographically with $1 \leq n_{\text{sub}x} \leq 4$ and $1 \leq n_{\text{sub}y} \leq 6$. The $n_{\text{sub}x} \times n_{\text{sub}y}$ combinations considered are $1 \times 1, 2 \times 1, 3 \times 1, 4 \times 1, 4 \times 2, 4 \times 3, 4 \times 5$ and 4×6. We consider the GMRES, GMRES/Chebyshev basis and GMRES/repeated basis algorithms with $n_{\text{lap}} = 1$ or 5. In all cases convergence occurs in 175 iterations (see Figs. 4 and 5).

These results show the efficiency of the new algorithms compared to GMRES on a larger number of processors, due to reduced communication overhead. As expected, the use of domain overlapping increases runtime, but this is to be traded off with reduced communication overhead. As before, the lower Mflop rates for the GMRES/Chebyshev basis algorithm compared to the repeated basis algorithm are because the former is richer in matrix-vector products, which are slightly more expensive than inner products and SAXPY operations, since the Hadamard product operation requires three vectors while the SAXPY and inner product each require two.

In the next set of runs we impose the condition $n_{\text{ptsub}x} = 15$, $n_{\text{ptsub}y} = 10$. We force termination at 50 iterations in each case. As before, $k = 5$, $Ch_x = 4$, and double precision is used (see Figs. 6 and 7).

The results from these runs are similar though less dramatic. The new algorithms

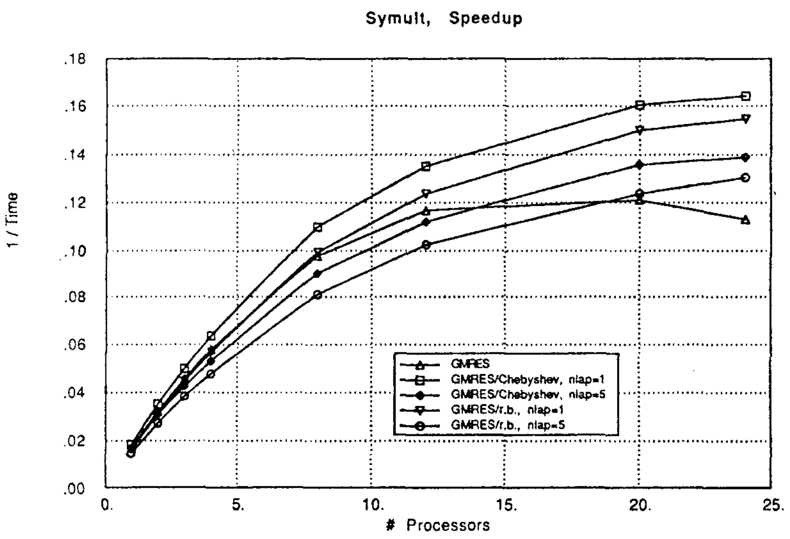

Figure 4.

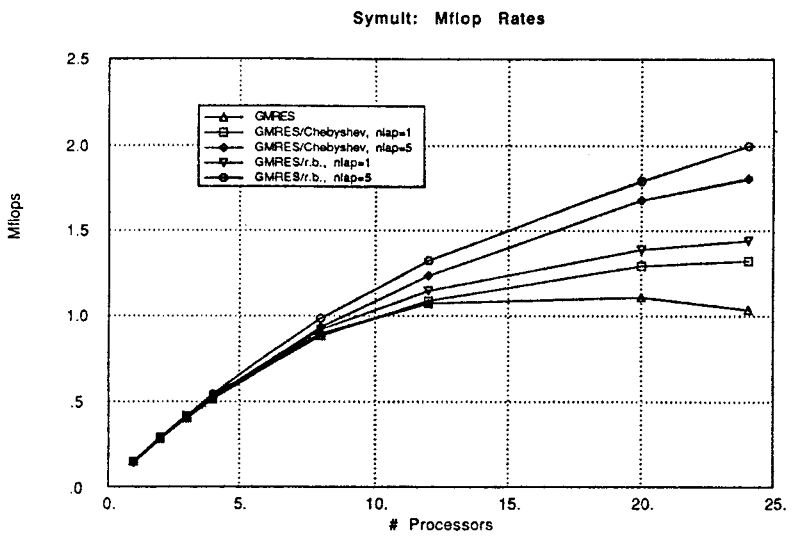

Figure 5.

show less communication overhead than GMRES. It is worth noting that the global communication is significantly more costly than the local communications for the matrix-vector products. Also, the difference in Mflop rates for GMRES and the repeated basis algorithm on 1 processor may be attributed to the added overhead of the small system solve, which is not reckoned in the operation counts. As mentioned previously, this factor may be significant when k is very large.

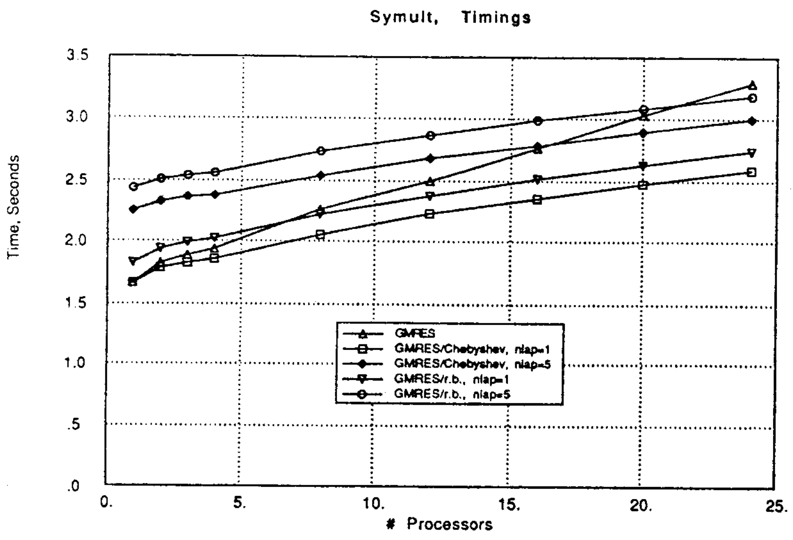

Figure 6.

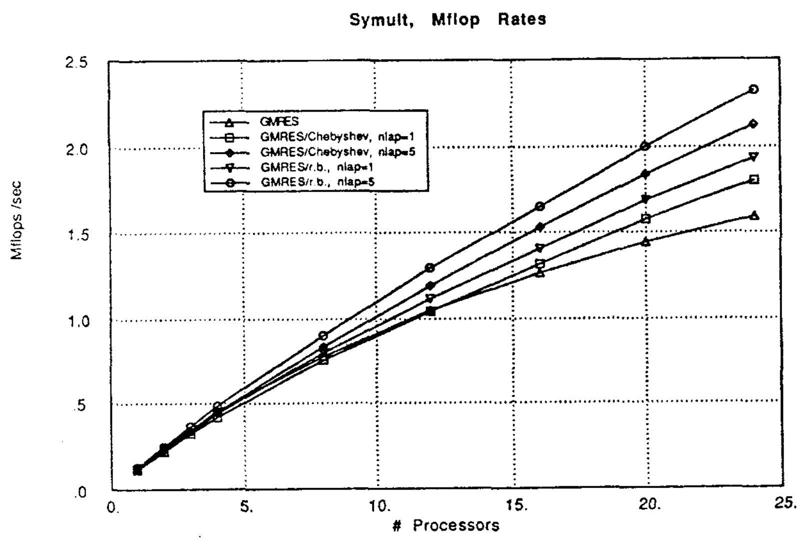

Figure 7.

We conclude from these runs that collecting the inner product computations together in the new algorithm can significantly cut down the communication overhead, resulting in better parallel speedups.

4.5 Ncube 2 Experiments

The Ncube 2 computer of Sandia National Laboratory is also used to test these algorithms. This machine has 1024 processors configured in a hypercube array with a Sun 4 front end computer. Each node contains 4 Mbytes of memory. The CPU units are custom-made chips supporting IEEE arithmetic and running at 20 Mhz clock speed. The values of ϵ_M are $0.59605E-7$ (single precision) and $0.11102E-15$ (double precision).

To run the Fortran code, the Fortran-to-C converter developed by AT&T Bell Labs was used to translate the code to C, and then the Ncube ncc compiler was applied with optimization level 2. The Mathlib assembly language coded routines for the basic linear algebra operations (BLAS) such as SAXPYs and dot products provided by Parallel Solutions Inc. and made available at Sandia National Laboratory were used. In order to perform such operations on subdomain patches which included varying amounts of overlaps, it was necessary to use doubly-nested loops, or more specifically a single loop containing a call to the appropriate BLAS routine to handle the given grid-line in the x direction. Due to compiler overhead, this implementation could be improved upon by using a doubly-nested loop fully coded in assembly language.

Otherwise, the implementation is nearly identical to the Symult implementation, making use of a standard gray code mapping to embed the two-dimensional grid into the hypercube. The standard communication routines NREAD and NWRITE

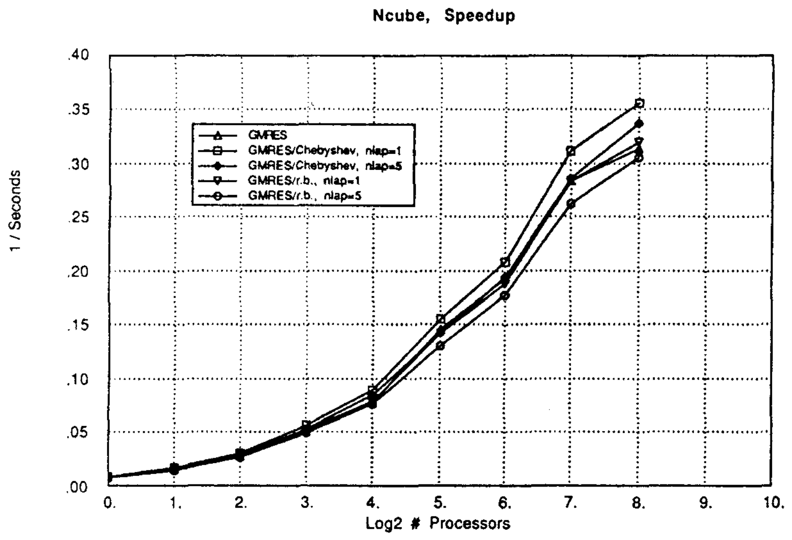

Figure 8.

are used for neighbor communication. Inner products are computed by use of a standard binary exchange technique which requires order log p time, where p is the number of processors.

For these runs we impose $(n_x - 1) = (n_y - 1) = 128$ and allow $n_{\text{sub}x} \times n_{\text{sub}y}$ combinations of the form 1×1, 1×2, 2×2, 2×4, 4×4, 4×8, 8×8, and so forth. As before, we use double precision and let $Ch_x = 4$, $k = 5$ and $\zeta = 10^{-6}$, leading to 370 iterations for convergence in all cases (see Figs. 8 and 9).

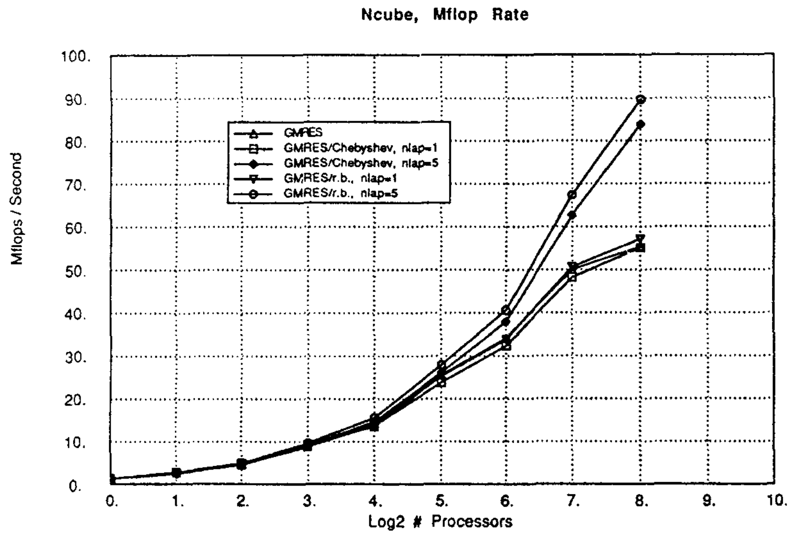

Figure 9.

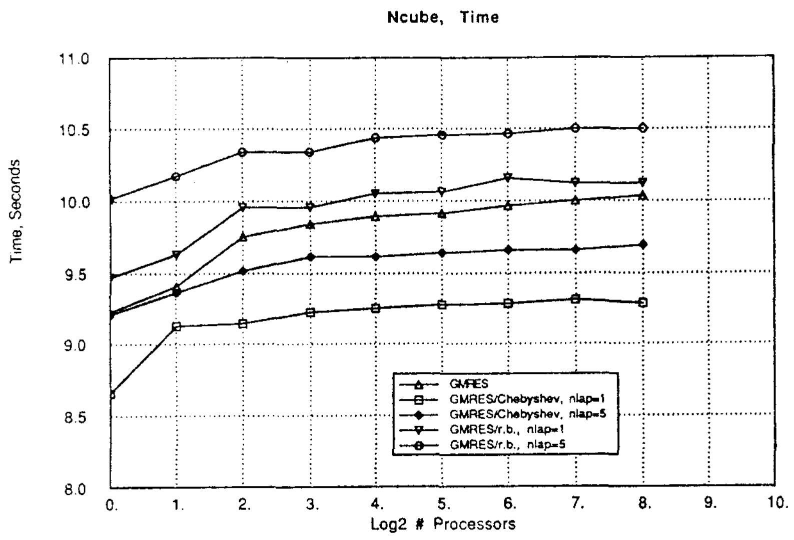

Figure 10.

These results are qualitatively similar to the Symult results but much less dramatic. The comparatively fast communication startup time accounts for the similar behavior of the algorithms. Only a small improvement of the repeated basis algorithm over GMRES is discernible for 256 processors.

In the following runs we fix $n_{\text{ptsub}x} = n_{\text{ptsub}y} = 64$ and vary the number of processors. We perform 100 iterations in all cases and as above let $Ch_x = 4$ and $k = 5$ and use double precision (see Figs. 10 and 11).

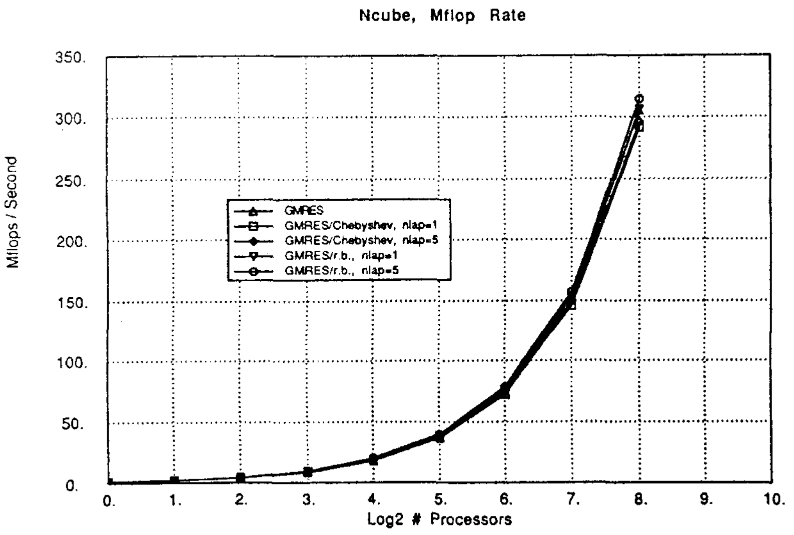

Figure 11.

Again, these results are similar to the Symult runs though less dramatic. There is discernible a slightly greater growth rate in the CPU time for GMRES compared to the other algorithms.

5 CONCLUSIONS

In this paper we have given implementations of the Chebyshev basis restarted algorithms and other algorithms defined in Part I, implementations for distributed memory machines, cache memory machines and machines for which disk memory is used for the solution process. We have also given numerical results which indicate that a careful implementation of the new algorithms can result in a substantial improvement in the runtime.

The most dramatic results are for out-of-core solution. These results indicate that out-of-core solution can have timings more comparable to in-core solution, due to the use of the new algorithms.

For iterative solution on distributed memory machines, the benefits of these algorithms depend on the relative startup cost for interprocessor communication. For sufficiently large problems, this overhead becomes negligible. On the other hand, the algorithms are of greater benefit when a given problem is distributed across a massively parallel machine so that each processor has a relatively small number of vector components.

We remark that all the ideas presented here carry over directly to other generalized conjugate gradient algorithms, such as for example the standard conjugate gradient algorithm. Similar algorithms based on the power basis are set forth, for example, in [Chronopoulos 1986].

Further research is necessary to investigate the important topic of effective parallelizable preconditioning techniques.

Acknowledgements

The use of the Ncube 2 processor of Sandia National Laboratory is gratefully acknowledged.

References

Ashby, S. F., T. A. Manteuffel and P. E. Saylor, Adaptive polynomial preconditioning for Hermitian indefinite linear systems, *BIT*, **29**, (1989), 583–609.

Ashby, S. F., T. A. Manteuffel and P. E. Saylor, A taxonomy of conjugate gradient methods, *SIAM J. Numer. Anal.*, **27**, (6) (1990), 1542–1568.

Barragy, E. and G. F. Carey, A parallel element-by-element solution scheme, *International Journal for Numerical Methods in Engineering*, **26**, (1989), 2367–2382.

Carey, G. F. and E. Barragy, Basis function selection in preconditioning high degree finite element and spectral methods, *BIT*, **29**, (1989), 794–804.

Chronopoulos, A., *A Class of Parallel Iterative Methods Implemeted on Multiprocessors*, Report UIUCDCS-R-86-1267, University of Illinois at Urbana-Champaign, Ph.D. Thesis, November 1986.

Chronopoulos, A. T. *s-step Iterative Methods for (Non)symmetric (In)definite Linear Systems.* Technical Report TR-89-35, Department of Computer Science, University of Minnesota, Minneapolis, Minnesota, 1989.

Chronopoulos, A. T., *Krylov Subspace Iterative Methods for Nonsymmetric Indefinite Linear Systems*, Technical Report TR 90-21, Department of Computer Science, University of Minnesota, Minneapolis, Minnesota, April 1990.

Chronopoulos, A. T. and C. W. Gear, s-step iterative methods for symmetric linear systems, *Journal of Comp. and Applied Math.*, **25,** (1989), 153–168.

Chronopoulos, A. T. and C. W. Gear, Implementation of preconditioned s-step conjugate gradient methods on a multi processor system with memory hierarchy, *Parallel Computing*, **11,** (1989), 37–53.

Chronopoulos, A. T. and S. K. Kim, *s-Step Orthomin and GMRES Implemented on Parallel Computers*, Technical Report TR 90-15, Department of Computer Science, University of Minnesota, Minneapolis, Minnesota, February 1990.

di Brozolo, G. R. and Y. Robert, *Vector and Parallel CG-like Algorithms for Sparse Non-symmetric Systems*, IMAG, Grenoble Cedex France, Report RR 681M, October 1987.

Eisenstat, S. C., H. C. Elman and M. H. Schultz, Variational iterative methods for nonsymmetric systems of linear equations, *SIAM J. Numer. Anal.*, **20,** (2) (1983), 345–357.

Golub, G. H. and C. F. Van Loan, *Matrix Computations*, second edition. Baltimore: John Hopkins University Press, 1989.

Hageman, L. A. and D. M. Young, *Applied Iterative Methods*. New York: Academic Press, 1981.

Hestenes, M. R. and E. L. Stiefel, Methods of conjugate gradients for solving linear systems, *J. Res. Nat. Bur. Standards*, **49,** (1952), 409–436.

Joubert, W. D., *Generalized Conjugate Gradient and Lanczos Methods for the Solution of Nonsymmetric Systems of Linear Equations*, the University of Texas at Austin, Ph.D. Thesis, Report CNA-238, Center for Numerical Analysis, Austin, Texas, January 1990.

Joubert, W., *Iterative Methods for the Solution of Nonsymmetric Systems of Linear Equations*, the University of Texas at Austin, Center for Numerical Analysis, Report CNA-242, February 1990.

Joubert, W. D. and T. A. Manteuffel, *Iterative Methods for Nonsymmetric Linear Systems*, in *Iterative Methods for Large Linear Systems*, David R. Kincaid and Linda J. Hayes, eds. Boston: Academic Press, 1990, pp. 149–171.

Kim, S. K. and A. T. Chronopoulos, *A Class of Lanczos-like Algorithms Implemented on Parallel Computers*, Technical Report TR 89-49, Department of Computer Science, University of Minnesota, Minneapolis, Minnesota, 1989.

Kim, S. K. and A. T. Chronopoulos, *The s-Step Lanczos and Arnoldi Methods on Parallel Computers*, Technical Report TR 89-83, Department of Computer Science, University of Minnesota, Minneapolis, Minnesota, December 1989.

Manteuffel, T. A., Adaptive procedure for estimation of parameters for the nonsymmetric Tchebychev iteration, *Numerische Mathematik*, **31,** (1978), 187–208.

Saad, Y. and M. H. Schultz, Conjugate gradient-like algorithms for solving nonsymmetric linear systems, *Mathematics of Computation*, **44,** (170) (1985), 417–424.

Saad, Y. and M. H. Schultz, *Parallel Implementations of Preconditioned Conjugate Gradient Methods*, Yale University, Department of Computer Science, Report YALEU/DCS/RR-425, October 1985.

Saad, Y. and M. H. Schultz, GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems, *SIAM J. Sci. Stat. Comp.*, **7,** (3) (1986), 856–869.

Seager, M., Parallelizing conjugate gradient for the CRAY X-MP, *Parallel Computation*, **3,** (1986). 35–47.

Vinsome, P. K. W., *ORTHOMIN, an Iterative Method for Solving Sparse Sets of Simultaneous Linear Equations*, in *4th Symposium of Numerical Simulation of Resevoir Performance of the Society of Petroleum Engineers of the AIME*, Los Angeles, Calif., 1976, Paper SPE 5739.