SEMI-ITERATIVE METHODS ON DISTRIBUTED MEMORY MULTIPROCESSOR ARCHITECTURES

A. Hadjidimos, E.N. Houstis, J.R. Rice, M.K. Samartzis and E.A. Vavalis

Purdue University
Department of Computer Sciences
West Lafayette, IN 47907

ABSTRACT

In the parallel ELLPACK (//ELLPACK) project we are developing a library of parallel iterative methods for distributed memory multiprocessor systems and software tools for partitioning and allocation of the underlying computations. In this paper we discuss the implementation issues within the //ELLPACK environment and present preliminary performance results of some of the modules on three hypercube based architectures: NCUBE, iPSC/1 and iPSC/2. These results indicate that the iterative methods are capable of delivering close to optimal scaled speed-ups while the combination of concurrent/vector processing can lead to sizable improvements of the overall performance. These experiments have shown that distributed memory systems are capable of solving significantly large problems effectively.

1. INTRODUCTION

In this paper we discuss the implementation of a library of parallel iterative methods (//ITPACK) for solving large linear systems of algebraic equations obtained by discretizing elliptic partial differential equations (PDEs) with various finite element and difference schemes. Preliminary performance results are reported for some of the modules for three existing hypercube machines. This library currently consists of the 12 modules listed in Table 1.1 using well known iterative methods and ordering schemes. For the efficient development of new parallel elliptic solvers and the transformation of existing ones from the sequential ELLPACK system, we have developed and implemented a parallel software environment [Hous 89a] for distributed memory multiprocessor systems. The architecture of this system is described in Section 2, its current functionality is capable of supporting *geometry decomposition* methods at the levels of the user interface and solution process. All the domain splitting solvers are driven by a *geometry decomposition tool* described in [Hous 89b] and [Chri 89].

//ITPACK module	Ordering Scheme	Method
Jacobi–CG	block, arrow–head	Jacobi conjugate gradient
Jacobi–SI	block, arrow–head	Jacobi with Chebyshev acceleration
SOR	block, arrow–head	Successive Over–Relaxation
SSOR–CG	block, arrow–head	Symmetric SOR conjugate gradient
SSOR–SI	block, arrow–head	Symmetric SOR with Chebyshev acceleration
Jacobi Schwarz	block	Schwarz splitting with Jacobi iteration
GS Schwarz	block	Schwarz splitting with Gauss–Seidel iteration

Table 1.1 Parallel iterative methods in the //ITPACK library.

This research was supported in part by AFOSR grant 88-0243, ARO grant DAAG29-83-K-0026 and NFS grant CCR-8619817.

The parallel block iterative schemes are based on the sequential modules in the ELLPACK system which are also driven by the geometry decomposition tool. The iterative Schwarz splitting schemes are formulated on subdomains which are implicitly defined through the non-overlapping domain splitting produced by the decomposition tool and an overlapping parameter. The paper is organized as follows. Section 2 contains a brief description of the proposed architecture for the //ELLPACK system.

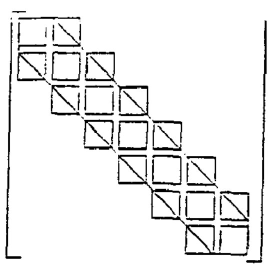
In Section 3 we present the parallel implementation strategies used and discuss their communication complexity. Performance results obtained on Ncube, iPSC/1 and iPSC/2 hypercube machines are presented and discussed in Section 4. A summary of the experience gained so far is included in Section 5.

2. PARALLEL ELLPACK ARCHITECTURE

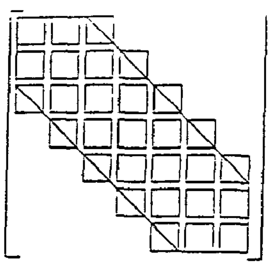
The overall design objective of the //ELLPACK is the creation of a uniform programming environment in which parallel software modules associated with the components of PDE solvers can be implemented and interfaced with minimum effort. The transformation of the existing sequential ELLPACK modules to parallel versions was another important consideration that influenced the design of //ELLPACK system. Following the ELLPACK conventions, a parallel PDE solver either consists of a set of modules realizing the various steps of the PDE solution or a single module. In ELLPACK terminology this solvers are called *multiphase* or *triples*. In the current implementation of //ELLPACK architecture, the partition and the allocation of the underlying computation takes place at the discrete geometrical data structures (mesh data). This phase is implemented by the *geometry decomposition* tool [Hous 89b] which is supported by a number of automatic mapping algorithms [Chri 89]. In the //ELLPACK library we implement and study linear solvers which assume **block** or **substructuring** ordering of algebraic data defined with respect to a predefined mesh decomposition. In contrast with ELLPACK, the indexing of the algebraic data in //ELLPACK takes place prior to the generation of these data. This sequencing is necessary for reducing the communication overhead among processors. For the so called multi-phase PDE solvers, we have generalized the ELLPACK interfaces among the various modules [Hous 89a] in order to accommodate the geometry decomposition, indexing communication interfaces and I/O parameters and data structures. Specifically, a parallel multi-phase elliptic PDE solver consists of eight primitive modules: **domain discretization, mesh generation, indexing, discretization, discrete system solution, post processing of the solution and computation back play.** Currently the first three modules are executed at the host or fast remote servers. The indexing takes place at the host while the interfaces of these modules are broadcast to all processing elements of the targeted architecture. Each individual processor is responsible for the generation of the equations associated with the assigned subdomains and setting up the communication interfaces among them, all this is guided by the decomposition tool. The algebraic data are stored locally in sparse mode together with global indexing information. The communication interfaces depend on the ordering scheme used and are defined in terms of the local and global information generated by the domain processor and the indexing module. In order to implement the back play of the parallel execution, we use the SEECUBE tool [Couc 87].

3. A LIBRARY OF PARALLEL ITERATIVE METHODS

One of the objectives of the //ELLPACK project is the design and the implementation of an efficient portable general purpose library of iterative methods for the solution of large PDE discretization systems of arbitrary PDE domain geometry. This library is designed to operate efficiently on distributed memory multiprocessor machines. The modules in this library are driven by two tools the **geometry decomposition** tool described in [Hous 89b] and [Chri 89], and an **expert system** that supports the selection of the appropriate solution modules and their parameters. The proposed library currently consists of the ITPACK methods implemented under two different orderings and two iterative schemes based on Schwarz splitting. These are four semi-iterative schemes with conjugate gradient and Chebyshev acceleration and the basic SOR method implemented using **block** and **substructuring** orderings of the equations and the corresponding unknowns. In the **block** ordering we assume a *geometry decomposition* of contiguous elements which results in a rowwise partition of the associated algebraic data structures of $Au = f$. Each processor i is assigned the computation associated with r_i consecutive rows of the matrix A and the corresponding slice of the unknown u and the right hand side f. Depending on the discretization method (or the block structure of the coefficient matrix A) and the hardware configuration (number of processors) selected (see Figure 3.1) we map the partitioned data structure onto an interconnection scheme (ring, grid, etc) which can be embedded into a hypercube ensemble of processors. In this way we have communication only between nearest neighbor nodes.



(a) ring



(b) 2-d grid

Figure 3.1 Data structure obtained from the block ordering scheme.

The **substructuring** ordering is defined with respect to some decomposition or splitting of the associated discrete geometric data structures into a number of non–overlapping subdomains. In [Chri 89] and [Hous 89b] this partition problem is studied and an appropriate algorithmic infrastructure is developed for its automatic solution. In this ordering scheme the algebraic data associated with subdomain interfaces are ordered last while the rest are ordered first. This leads to the *Arrow–Head* structure indicating in Figure 3.2.

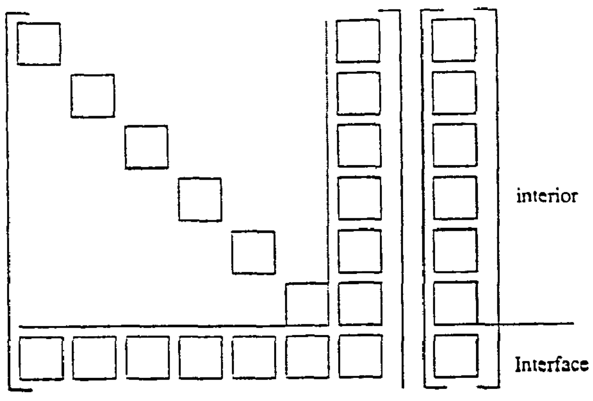


Figure 3.2 Arrow-Head structure obtained by the substructuring ordering scheme.

For the implementation of the Schwarz splitting modules we derive them from non-overlapping domain decompositions and the overlapping region is defined implicitly by expanding these substructures or superelements. The amount of overlapping is a fixed parameter for all subdomains and it is considered as an acceleration parameter.

For the modules of the //ITPACK the initial estimation of the solution and various acceleration parameters are implicitly calculated inside each module or are provided by the expert system front-end. Finally, stopping criteria similar to the ones used in sequential ITPACK [Rice 85] are employed. For the description of the iterative schemes included in //ITPACK we assume thought the scaled splitting $(I-L-U)u = c$ of the discrete equations, with L being a strictly lower triangular matrix and U being a strictly upper triangular one. In the next subsection we indicate some important implementation issues related to the parallel formulation of the methods in Table 1.1.

3.1 Parallel SOR

Assuming the above mentioned $(I-L-U)$ splitting of the matrix A, the n^{th} iteration of the SOR method is given by

$$u^{(n+1)} = \omega(Lu^{(n+1)}+Uu^{(n)}+c)+(1-\omega)u^{(n)} \qquad (3.1)$$

where ω is the overrelaxation parameter. The first SOR iteration uses ω=1 and a heuristic procedure is used to estimate the optimum relaxation factor ω.

First, we consider the SOR implementation over the block structures of Figure 3.1. Since SOR reuses updated values as soon as they are available, each processor must wait for the previous processors' updates before it starts working on its equations. In this implementation we parallelize the SOR iterations in a pipelined fashion. It is known that pipelining combined with global broadcasts leads to a serial execution. Thus, we use a pipeline technique to calculate the vector/vector operations for avoiding global broadcasts. A number of specific details about the implementation of the algorithm are worth pointing out. Before the scaling of the matrix A and the right hand side vector f we communicate the diagonal entries of A in a nearest neighbor fashion. More precisely each node receives the associated diagonal entries from the nearest neighbors (right and left in the ring case and north, south, west and east in the grid case) and sends its own entries. Within each iteration we first receive the updated values of the unknown vector u from the "previous" (right in ring case and south and west in the grid case) processors. Next the updating of the local unknowns takes place and these values are sent to the "next" processors (left in the ring case and north and east in the grid case). The calculation of the inner products take place locally and a similar communication scheme is used to form the various norms globally. Finally only the last processor applies the stopping test and sends a "halt" flag to the others when convergence has been reached. It is clear from the above discussion that only nearest neighbor data transfer is used while we overlap communication with computation as much as possible. Furthermore, it can be seen that for this implementation we need more iterations than the sequential SOR method to reach convergence.

Second, we present the formulation of the domain decomposition SOR. In this case, it is assumed that the algebraic data are stored in an arrow-head structure (Figure 3.2) where each diagonal block and its corresponding coupling interface matrix are stored in different processors. For the interface equations there are three mapping alternatives. They can be allocated to

1) a single processor, or
2) be partitioned rowwise for symmetric matrices and columnwise for non-symmetric ones and allocated to the previous used processors, or
3) recursively be partitioned using some substructuring of the interfaces.

For two dimensional problems we are exploring the first two options. The third alternative has been formulated in [Farh 87] in connection with SOR iteration which we plan to implement in the //ITPACK library.

3.2. Parallel Semi-iterative Methods.

It is well known that the basic iterative methods (such as Jacobi, SSOR) can be accelerated by using appropriate linear combinations of consecutive iterants of the basic methods. These acceleration techniques lead to the so-called semi-iterative methods. We consider four such methods based on the Jacobi and SSOR basic iteration schemes and the accelerations Chebyshev (SI) and conjugate gradient (CG). A detailed description of these methods can be found in [Kinc 82] while some of the implementation issues are discussed in [Rice 85]. If G is the Jacobi iteration matrix then the methods are defined by the iteration equation

$$u^{(n+1)} = \rho_{n+1}\,[\gamma_{n+1}(Gu^{(n)} + k) + (1-\gamma_{n+1})u^{(n)}] + (1-\rho_{n+1})u^{(n-1)} \qquad (3.2)$$

where ρ_{n+1} and γ_{n+1} are appropriate acceleration parameters. In the case of Chebyshev acceleration these parameters are given in terms of the M(G), largest, and m(G), smallest eigenvalues of G. In the absence of their values, adaptive procedures can be used to estimate them [Rice 85]. In order to visualize the parallel implementation of the sequential ITPACK routines we present the Jacobi–SI module in a high level form.

```
d := f;

start_iterations;
d := d-Au;   compute pseudo residual
dnrm = $d^T \times d$; unrm = $u^T \times u$; compute d and u norms
if adapt_test(dnrm) then; adaptivity tests
  if cme_test(dnrm,unrm) then;
    compute_cme; recompute M(G)
  else if sme_test(dnrm,unrm) then;
    compute_sme; recompute m(G)
  endif;
endif;
if conv_tests(dnrm,unrm,sme,cme) goto unscale(u);
u := compute(sme,cme,u); compute new estimation of u
goto start_iterations;

unscale(u); unscaling of u
```

Algorithm 3.1 The sequential adaptive Jacobi–SI algorithm.

The use of the conjugate gradient acceleration yields a convergence rate which is nearly always faster than Chebyshev acceleration and can be described also by Algorithms 3.1 and 3.2. In the conjugate gradient the estimation of the spectral radius ($M(G)$ and $m(G)$) is not needed for calculating the acceleration parameters but used in performing the stopping and adaptivity tests. These parameters are calculated using the product $d^T \times A \times d$.

The Symmetric SOR (SSOR) based semi-iterative methods are characterized by the matrix

$$Q = [\omega(2-\omega)]^{-1}(I-\omega L)(I-\omega U)$$

involved in the iteration matrix $G = I - Q^{-1}A$. Again the semiiteration equations are defined by (3.2) where ρ_n and γ_n can be determined by either Chebyshev or conjugate gradient accelerations. For the implementation we use the fact that, each step of the SSOR method consists of two SOR (forward and backward) sweeps. For the visualization of the inherent parallelism in the semi-iterative SSOR computation we include a high level description of the SSOR–SI ELLPACK module. The CG case is almost identical with the differences indicated above.

```
d := f;

start_iterations;
sor(forward); forward SOR sweep
sor(backward); backward SOR sweep
residual(forward,backward);
if (adapt) then; adaptivity test
 if (acceleration) then;
  estimate(accel); estimate acceleration parameters
 endif;
 if (relaxation) then;
  estimate(omega); estimate optimum omega
 endif;
endif;
if (convergence) goto unscale;
goto start_iterations;

unscale(u); unscaling of u
```

Algorithm 3.2 The sequential adaptive SSOR–SI algorithm.

3.2.1 Parallel Jacobi Semi-iterative Methods

Starting with the **block** partition given above (Figure 3.1), we denote by $A^{(i)}$, $u^{(i)}$ and $f^{(i)}$ the linear system blocks assigned to processor i and we assume they reside in its local memory. For the parallel Jacobi-SI method we see from the description of Algorithm 3.1 that each processor needs to broadcast its own diagonal entries of A and receive appropriate diagonal entries of other processors for scaling and unscaling the matrix A. This can be done by means of fanout/fanin algorithms [Ayka 88], [Gust 88]. The computation of the inner products required for the calculation of the norms of **u** and **d** is done by first computing on each processor i, the local part of the inner product and then using additive bidirectional exchanges [Gust 88] to calculate the inner product as the sum of all partial sums. The calculation of the new estimation of **u** within each iteration is done locally after receiving appropriate values of the old estimation from neighbor processors. From the above discussion one concludes that the only additional computation required to parallelize the original algorithm is due to indexing. In addition, interprocessor communication involves mostly nearest neighbor exchanges of one dimensional arrays and global exchanges of scalars. This parallel implementation gives a good aspect ratio ($r = \frac{time\ for\ computation}{time\ for\ communication}$) which leads to efficient performance.

If we use the *substructuring* ordering and consider the corresponding arrow-head data structures in Figure 3.2 then we assign to each processor the equations associated with a subdomain. Then the interface equations are equally distributed among all processors. In this implementation, within each iteration the computation of the required inner products is performed in two phases. First, the part local to each processor is computed and then bidirectional exchanges among all processors are used to compute the part associated with the interface data. For the updating of the unknown vector within each iteration only the values of the interface unknowns are broadcast.

From Algorithms 3.1 and 3.2 and the preceding discussion it can be seen that the parallel implementation of the conjugate gradient acceleration can be treated similarly. Moreover, the calculation of the inner product $d^T \times A \times d$ that is required for estimating its acceleration parameters can be treated similarly using the fanin/fanout algorithms.

3.2.2 Parallel SSOR Semi-iterative Methods

From the discussion in Sections 3.1 and 3.2 it is clear that the parallel realization of accelerated SSOR methods follows easily from the parallel implementation of the SOR method and the parallel accelerated techniques used for the Jacobi semi-iterative methods. Specifically, in the block structured case, we perform both the backward and the forward SOR sweeps in a pipelined fashion. In the same way we calculate the associated pseudo-residuals and the inner products needed. It can be seen that the substructuring ordering transforms the two SOR sweeps to a Jacobi-like scheme. Within each SSOR iteration the updates of the interior unknowns local to each subdomain are performed in parallel while a broadcast of the interface unknown vector is needed before each processor starts updating its interface unknowns. Note that, in addition to the inner products needed for the estimation of the acceleration parameters, additional vector/vector operations are performed to estimate the optimum relaxation parameter ω even in the conjugate gradient case.

3.3 Parallel Schwarz Splitting Methods

It has been recognized that the Schwarz splitting technique is a powerful alternative for solving elliptic PDEs in parallel architectures (see [Rodr 84], [Tang 87] and [Hous 88b]). For the //ELLPACK implementation of the Schwarz splitting the domain decomposition tool is used first to split the PDE domain into a set of non-overlapping subdomains. Then, the overlapping parameter is used to expand the subdomains to overlapping ones and on each one of them appropriate boundary conditions are imposed where needed. In this way the original PDE problem is replaced by a number of PDE subproblems. The interaction among them is achieved by means of an iteration scheme with the solution of each of them being assigned to a different processor. It is important to notice that each subproblem needs to receive updates on its "pseudo-boundaries" from processors associated with the solution of the *"neighbor"* subproblems and to send its local updated unknowns on the "pseudo-boundaries" of the *"neighbor"* subdomains to corresponding processors. The concept of "neighbor" subproblem is determined from the connectivity information of the geometric decomposition, the overlapping parameter and the ordering of the subdomains. The order of receiving and sending the data depends on the iteration scheme used. Most of the basic convergence properties, developed essentially in [Rodr 84] and [Tang 87], have been extended to cover cases where more than two overlapping subdomains share a common part of the original region. A detailed discussion of a //ELLPACK implementation of the Schwarz splitting associated with Jacobi and Gauss-Seidel iterative schemes, together with numerical experiments on hypercube architectures, is given in [Hous 88b].

4. PERFORMANCE RESULTS

In this section we present the performance data of some //ITPACK modules solving the linear equation system arising from discretizing Elliptic PDEs on the NCUBE, iPSC/1 and iPSC/2 hypercube multiprocessors. Table 4.1 indicates the specific hardware configurations of these machines used in our experiments. Extensive performance comparison of these machines together with a detailed description of their hardware and software characteristics can be found in [Duni 88].

	NCUBE	iPSC/2	iPSC/1
CPU	custom 32–bit	80386/387	80286/287
Memory	512K	4M	512K
Clock Rate	6 MHZ	16 MHZ	8 MHZ
Nodes	128	32	32

Table 4.1 Hypercube configurations used for the implementation of semi–iterative methods.

The linear systems considered are obtained by discretizing linear second order elliptic partial differential equations on rectangular and non–rectangular domains. The timings reported here do not include the cost of the discretization phase, this time being at most 10% of the total solution cost. The solution time includes the time per iteration and the cost of adaptive and convergence test procedures. All computations were carried out in single precision. The FORTRAN implementation of the algorithms are almost identical for the three machines and portability was achieved by conditional compilation of the lower level communication and timing routines.

Figure 4.1 and Table 4.2 show the scaled performance of the parallel accelerated Jacobi semi–iterative methods. In this experiment we consider the solution of the elliptic equation $u_{xx} + u_{yy} - 100u = f$ on the domain $\Omega = [0,10]x[0,10]$ with solution $u = 1$ on the boundary of Ω and using the 5-point star finite difference method.

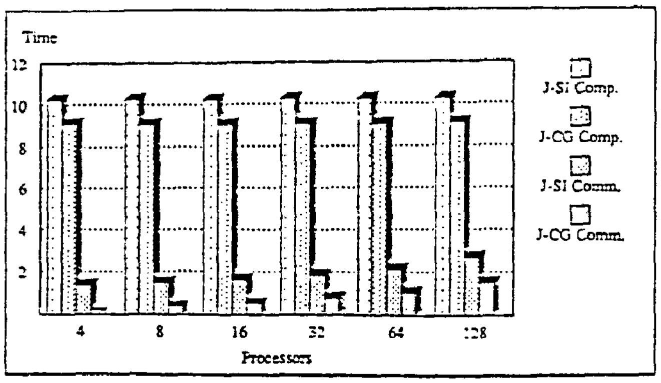


Figure 4.1 Scaled speed up of parallel Jacobi–SI (J–SI) and Jacobi–CG (J–CG) on the NCUBE multiprocessor. Each processor is assigned 400 equations and the assumed iteration tolerance is 10^{-6}.

The data indicate almost perfect scaled speed up for both methods. It is apparent that Jacobi–CG is faster than Jacobi–SI since it requires fewer iterations (42) compared to the Jacobi–SI (55). On the other hand, the need of computing the inner product $d^T \times A \times d$ leads to an increase of the time needed for communication. It should be pointed out that for comparison purposes in the data in Figure 4.1 and Table 4.2 we keep the spectral radius of the iteration matrix the same as we change the mesh size of the discretization.

	NCUBE		iPSC/2	
Nodes	T_{Comp}	T_{Comm}	T_{Comp}	T_{Comm}
4	764.2	84.7	199.5	20.9
8	764.6	85.1	201.9	21.1
16	765.1	86.2	203.0	22.3
32	766.9	88.3	204.2	24.0

Table 4.2 Communication (T_{Comm}) and computation (T_{Comp}) time requirements of Jacobi–SI on NCUBE and iPSC/2 for 4000 equations per processor.

Table 4.2 lists the numerical values of the scaled performance of the Jacobi–SI (similar measurements have been obtained for the Jacobi–CG) for the NCUBE and iPSC/2 machines. These data suggest that iPSC/2 is a faster machine while the speed up of the method is identical. In Table 4.3 we give the fixed size speed ups achieved by the parallel Jacobi–SI on the NCUBE after 400 iterations for the PDE problem considered in Figure 4.1. The fixed size speed up has been determined with respect to the sequential implementation.

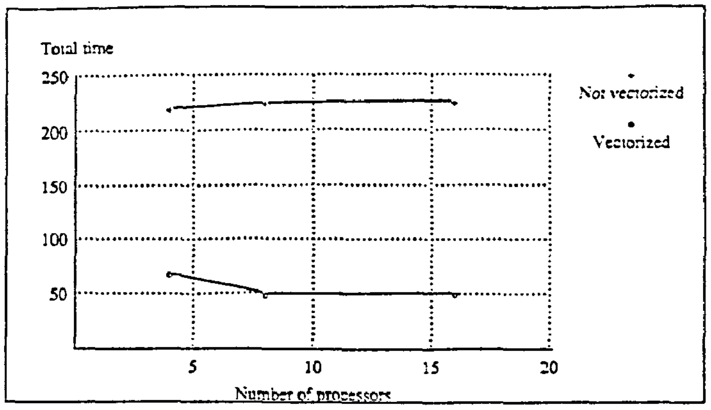


Figure 4.2 The effect of iPSC/2 vector processing for Jacobi–SI. Each processor is assigned a fixed number of equations as the hypercube size increases from 4 to 8 to 16, thus the linear system size increases by a factor of 4 and the memory used by a factor of 16.

Nodes	Grid	Speed up	Grid	Speed up	Grid	Speed up
4	3200 × 4	1.12		1.23		—
8	1600 × 8	2.08		2.17		—
16	800 × 16	4.19		4.52		4.38
32	400 × 32	7.93	128 × 128	8.52	256 × 256	9.10
64	200 × 64	14.93		15.90		17.45
128	100 × 128	13.60		26.74		33.15

Table 4.2 Fixed speed up of Jacobi–SI on NCUBE after 400 iterations.

Figure 4.2 illustrates the effect of vector processors in the iPSC/2. The results indicate that a speed up by a factor of four is obtained from vectorization. The vectorization of the Jacobi–SI subcomputations is achieved by using the vector BLAS, fill in, gather and scatter routines.

The performance of the parallel SOR module with block ordering is given in Figure 4.3. In this experiment we have considered the Poisson problem on the unit square with Dirichlet boundary conditions and true solution $u = 3e^{x+y}(x-x^2)(y-y^2)$. The 5-point star discretization module has been used and the fixed point speed up was calculated with respect to the sequential SOR method.

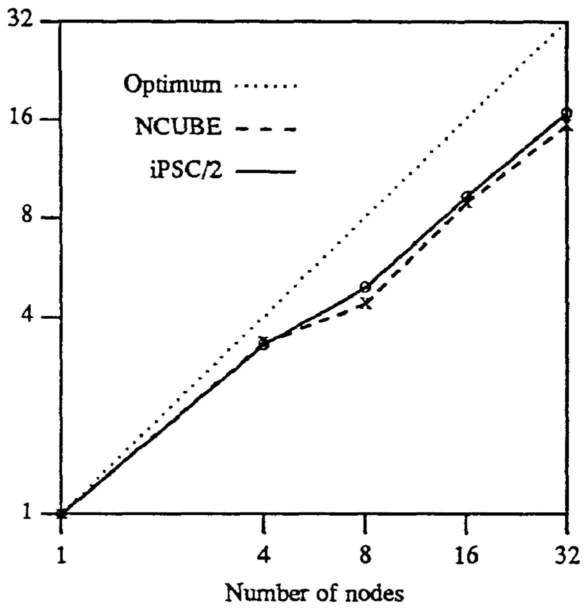


Figure 4.3 Fixed speed up of the block structured SOR on the NCUBE and iPSC/2.

To measure the performance of the //ITPACK modules that assume the substructuring ordering of the algebraic data, we consider the partial differential equation $u_{xx} + u_{yy} - (1+x)u = f(x)$ defined on the rectilinear domain depicted in Figure 4.4. The function f is selected so that $u = e^{(x+y)}$ and the rectangular grid is defined by an overlay grid of 64 × 64 lines.

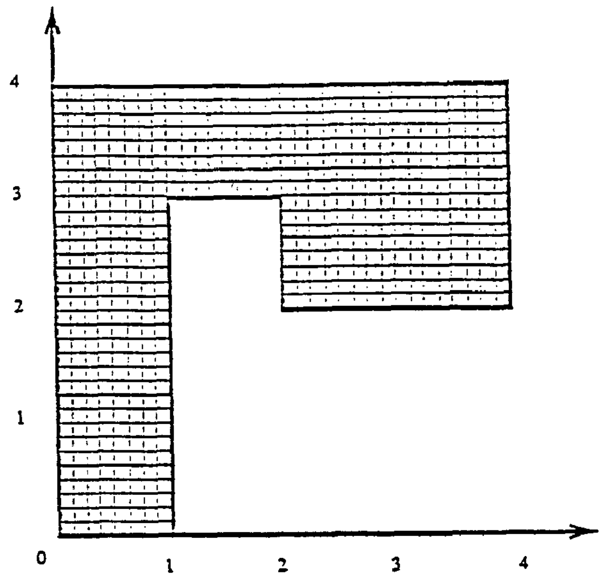


Figure 4.4 The rectilinear PDE domain.

The domain decomposition is done heuristically using the *domain decomposition tool* of //ELLPACK and the automatic load partitioning strategies developed in [Hous 89b]. We have used the //ELLPACK-NCUBE 5-point star discretization module to generate linear systems of equations. In this environment the **host processor** gets the domain's decomposition from DecTool, then applies the specified ordering of the algebraic data structures and distributes this information to **processor nodes** where the generation of the discrete equations takes place. Figure 4.5 shows the per iteration scaled and fixed speed ups achieved in the case of Jacobi–CG on NCUBE. For these measurements we used 100 equations per processor to compute the scaled speed up and 64 × 64 overlay grid for the estimation of the fixed speed up. It is worth pointing out that the performance of this method depends on the decomposition.

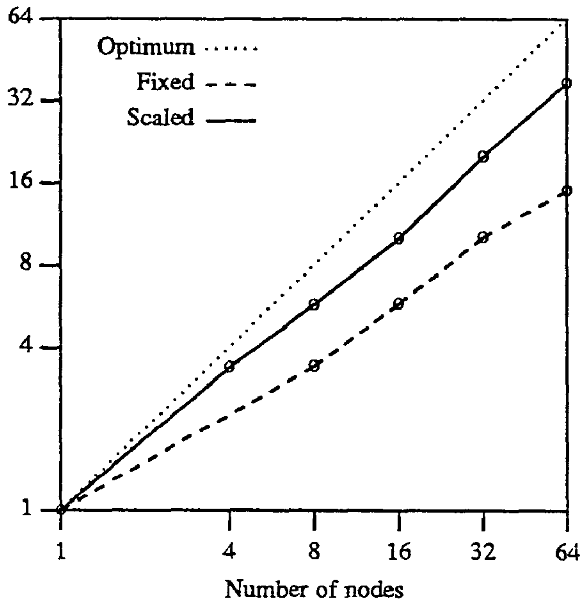


Figure 4.5 Per iteration speed ups of the domain decomposition Jacobi–CG method on the NCUBE.

5. CONCLUSIONS.

Iterative methods have been shown [Hous 88a] to be effective alternatives for solving well behaved sparse large linear systems of equations with "realistic" accuracy requirements. Their potential is even greater in parallel computation environments. We are developing a library of parallel iterative methods which is an extension of the ITPACK package for distributed memory machines. In this paper we present the performance of some of them on commercially available message passing MIMD (hypercube) machines. The preliminary experimental results indicate that they are capable of exploiting the parallel processing power of these machines and delivering close to optimal scaled speed ups. These data also indicate that the combination of concurrent/vector processing can lead to significant speed ups. Many studies have overlooked the space complexity of parallel algorithms/architecture pairs. Our experiments show that distributed memory systems are capable of solving very large problems. Specifically, we were able to solve close to 1/2 million finite difference equations in a reasonable time on NCUBE with 128 processors. A space and time complexity study for all modules in Table 1.1 is underway and results will be reported elsewhere.

ACKNOWLEDGMENTS

We are grateful to the Advanced Computing Research Facility at Argonne Laboratories for allowing us to use their iPSC/1 and the Advance Computing Facility at Cornell's Theory Center for allowing us to use their iPSC/2.

REFERENCES

[Ayka 88] C. Aykanat, F. Ozguner, F. Ercal and P. Sadayappan, *Iterative Algorithms for Solution of Large Sparse Systems of Linear Equations on Hypercubes,* IEEE Trans. Computers **37**, (1988), 1554–1568.

[Chris 89] N.P. Chrisochoides, E.N. Houstis, C.E. Houstis, S.M. Kortesis and J.R. Rice, *Automatic Load Balanced Partitioning Strategies for PDE Computations,* Purdue University CAPO Technical Report CER-89-7, February 1989.

[Couc 87] A. L. Couch, *Seecube User's Manual,* Tufts University, Department of Computer Science Report, April 1987.

[Duni 88] T. H. Dunigan, *Performance of a Second Generation Hypercube,* Tech. Report. ORNL/TM-10881, Oak Ridge National Laboratory, Mathematical Science Section, (1988).

[Farh 87] C. Farhat and E. Wilson, *Concurrent Iterative Solutions of Large Finite Element Systems,* Communic. in Applied Numerical Methods **8**, (1987), 319-326.

[Gust 88] J. L. Gustafson, G. R. Montry and R. E. Benner, *Development of Parallel Methods for a 1024-Processor Hypercube,* SIAM J. Sci. Stat. Comp. 9, (1988), 609-638.

[Hous 88a]E.N. Houstis, J.R. Rice, C.C. Christara and E.A. Vavalis, *Performance of Scientific Software,* Mathematical Aspects of Scientific Software (Ed. J.R. Rice), Springer-Verlag, (1988), pp 123-155.

[Hous 88b]E.N. Houstis, J.R. Rice, and E.A. Vavalis, *A Schwarz Splitting Variant of Cubic Spline Collocation Methods for Elliptic PDEs.* Hypercube Concurrent Computers and Applications, III (G. Fox, ed.), , ACM Press, (1988), 1746–1754.

[Hous 89a]E.N. Houstis, T.S. Papatheodorou and J.R. Rice, *Parallel ELLPACK: An Expert System for the Parallel Processing of Partial Differential Equations,* Math. Comp. Simul., **31**, (1989), to appear.

[Hous 89b]E.N. Houstis, P.N. Papachiou, J.R. Rice and M.K. Samartzis, *Domain Decomposer: A Software Tool for Partitioning PDE Computations Based on Geometry Decomposition Strategies,* Purdue University Technical Report, in preparation.

[Kinc 82] D.J. Kincaid, J. Respess, D.M. Young and R. Grimes, *Algorithm 586 ITPACK 2C : A Fortran package for Solving Large Sparse Linear Systems by Adaptive Accelerated Iterative Methods,* ACM Trans. Math. Soft. 8, (1982), 302–322.

[Rice 85] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK,* Springer-Verlag, New York, (1985).

[Rodr 84] G. Rodrigue and J. Simon, *Jacobi splittings and the method of overlapping domains for solving elliptic PDEs,* in: *Advances in Computer Methods for Partial Differential Equations*, Vol. V, (R. Vichnevetsky, R. S. Stepleman, editor), (1984), 383–386.

[Tang 87] W.P. Tang, *Schwarz splitting, a model of parallel computations,* Ph.D. Thesis, Stanford University, (1987).