1989

Semi Iterative Methods on Distributed Memory Multiprocessor Architectures

A. Hadjidimos

Elias N. Houstis

Purdue University, enh@cs.purdue.edu

John R. Rice

Purdue University, jrr@cs.purdue.edu

M. K. Samartzis

E.A. Vavalis

Report Number:

89-856

Hadjidimos, A.; Houstis, Elias N.; Rice, John R.; Samartzis, M. K.; and Vavalis, E.A., "Semi Iterative Methods on Distributed Memory Multiprocessor Architectures" (1989). *Department of Computer Science Technical Reports*. Paper 730.

<https://docs.lib.purdue.edu/cstech/730>

SEMI-ITERATIVE METHODS ON DISTRIBUTED
MEMORY MULTIPROCESSOR ARCHITECTURES

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

CSD-TR-856
February 1989

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
Technical Report CSD-TR-856
CAPO Report CER-89-6

ABSTRACT

In the parallel ELLPACK (/ELLPACK) project [Hous 89a] 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 report the formulation, implementation and performance of eight adaptive semi-iterative methods on three hypercube based architectures: NCUBE, iPSC/1 and iPSC/2. Preliminary experimental results indicate that these methods are capable of delivering close to optimal scaled speed ups while the combination of concurrent/vector processing can lead to sizeable improvements of the overall performance. These experiments have shown that distributed memory systems are capable of solving significantly larger problems than cost equivalent shared memory multiprocessor systems.

AMS (MOS) subject classification. 65N35, 65N05.

Key words. *iterative methods, parallel computation, hypercubes, elliptic partial differential equations.*

Abbreviation of the title. *Iterative Methods on Hypercubes.*

1. INTRODUCTION

The main objective of this research is the development of an efficient general purpose library of parallel iterative methods (//ITPACK) for the solution of large algebraic linear systems on distributed memory multiprocessor systems. Currently, the //ITPACK library consists of 12 modules described as follows

//ITPACK module	Ordering Scheme	Method
Jacobi-CG	block, arrow-head	Jacobi conjugate gradient
Jacob-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 iterations
GS Schwarz	block	Schwarz splitting with Gauss-Seidel iterations

We plan to add several multicolor iterative schemes in the future.

The block semi-iterative modules in //ITPACK are based on the ITPACK [Kinc 82] implementations of the corresponding modules in the ELLPACK environment [Rice 85]. In this first version of the paper we describe the parallel Jacobi CG and SI modules on three distributed memory multiprocessor systems with hypercube interconnection. The final paper will include the discussion of the rest. Specifically we measure the performance of the //ITPACK modules on the NCUBE (128 processors), PSC/1 (32 processors) and PSC/2 (32 processors). The //ITPACK modules have been implemented in the parallel ELLPACK (//ELLPACK) system and are driven by a domain decomposition tool [Hous 89b].

The rest of the paper is organized as follows. Section 2 contains a brief description of the corresponding sequential methods. In Section 3 we present the parallel implementation strategies used and analyze their communication complexity. Performance results obtained on Ncube, iPSC/1 and iPSC/2 hypercube machines are presented and discussed in Section 4. The conclusions and discussion of our future plans are in Section 5.

2. SEQUENTIAL SEMI-ITERATIVE ALGORITHMS.

It is well known that the basic iterative methods (such as Jacobi, SOR) 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 two such methods based on the Jacobi iteration and Chebyshev (Jacobi-SI) or conjugate gradient (Jacobi-CG) acceleration. A detailed description of these methods can be found in [Rice 85] and [Kinc 82]. If G is the Jacobi iteration matrix then both methods are defined by the iteration equation

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

where ρ_n and γ_n are appropriate acceleration parameters. 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. For the description of their parallel implementation we present the Jacobi-SI/ELLPACK version.

INPUT : f the right hand side vector f

A the coefficient matrix

u the initial estimation of the unknown vector

```

scale(A)                                scaling of A

d := f ; d := d+(I-A)u ; d := d-u      compute pseudo residuals

dnorm = d × dT ; unorm = u × uT      compute norms of d and u

if conv_test(dnorm,unorm) then stop      stopping test based on dnorm and unorm
if adapt_test(dnorm) then               test for recomputation of parameters
    dtnrm = d × A × dT
    if cme_test(dnorm,unorm) then        test for recomputing M(G)
        cme = accel(dtnrm)              compute M(G)
    else if sme_test(dnorm,unorm) then    test for recomputing m(G)
        sme = accel(dtnrm)              compute m(G)
    endif
endif

u := compute(sme,cme,u)                  compute new estimation of u

unscale(A)                               unscaling of A

```

Algorithm 2.1 The sequential adaptive Jacobi-SI algorithm.

3. PARALLEL SEMI-ITERATIVE METHODS.

Now we describe the reorganization of the computation in the above algorithms so they can run efficiently on hypercube architectures while preserving the convergence properties of the original sequential schemes. This reorganization is based on two partition schemes of the algebraic data structures while the reordered computation consist primarily by vector/vector and matrix/vector operations. It has been recognized ([Keye 87], [Ayka 88], [Duni 88], [Gust 88]) that these type of operations can be implemented very efficiently on distributed memory architectures using fanin/fanout algorithms or bidirectional exchanges. Next we present two parallel implementations of the adaptive Jacobi-SI algorithm.

3.1 Block Jacobi-SI method.

In this implementation the algebraic data structures of the system $Au = f$ are partitioned rowwise. Each processor i is assigned the computation associated with r_i consecutive rows of the matrix A and the corresponding slice of the unknown and the right hand side f . Depending on the bandwidth of the matrix A and the hardware configuration (number of processors) selected this can be achieved by mapping the block structure onto a ring (see Figure 3.1a) or grid (see Figure 3.1b) interconnections which can be embedded into a hypercube ensemble of processors.

Figure 3.1 Block partitions used for parallel accelerated Jacobi.

We denote by $A^{(i)}$, $u^{(i)}$ and $f^{(i)}$ the linear system blocks assigned to processor i and assume they reside in its local memory. According to the description of Algorithm 2.1, 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. The computation of the inner products required for the calculation of the norms of u and d are done by first computing on each processor i , the local part of the inner product and then use additive bidirectional exchanges to calculate the inner product as the sum of all partial sums. An inner product like expression $d \times A \times d^T$ can be treated similarly and only some indexing overhead is involved to get the appropriate locations of the matrix elements. 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 has to do only with 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 and efficient performance.

3.2 Domain-Decomposition Jacobi-SI Method.

Many scientific/engineering applications involve computations associated with geometrical objects. Very often these computations require the solution of large linear systems. Domain decomposition or substructuring methods are based on some

decomposition or splitting of the associated discrete geometric data structures in non-overlapping subdomains. They conceptually treat each subdomain as a separate computation. In [Hous 89c] we study this partition problem while in [Chri 89] we are developing software tools that assist the user in obtaining nearly optimal domain decompositions. The algebraic data structures obtained using this partitioning strategy depends very much on the ordering of the unknowns and equations. Here we formulate the semi-iterative methods which assume that equations associated with subdomain interfaces are ordered last while the rest of them are ordered first, this is the *Arrow-Head* ordering. Figure 3.2 indicates the data structures obtained by this scheme.

Figure 3.2 Data structure obtained from the Arrow-Head ordering scheme for linear equations.

4. PERFORMANCE RESULTS AND DISCUSSION

In this section we present the performance data of the Jacobi-SI and CG methods on the Ncube, iPSC/1 and iPSC/2 hypercubes 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]. All computations presented in this paper were carried out in single precision FORTRAN.

	Ncube	iPSC/2	iPSC/1
CPU-Node	custom 32-bit	80386/387	80286/287
Memory-Node	512K	4M	512K
Clock Rate	6 MHZ	16 MHZ	8 MHZ
Number of 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 is at most 10% of the total solution cost. The solution time includes the time per iteration and the cost of adaptive and the convergence test procedures. The FORTRAN implementation of the algorithms is almost identical for the three machines. Specifically, they differ in the timing and communication routines used.

4.1 Performance of Block Jacobi-SI and Jacobi-CG

In this experiment we consider the solution of the elliptic equation $u_{xx} + u_{yy} - 100u = f$ on the domain $\Omega = [0,10] \times [0,10]$ with solution $u = 1$ on the boundary of Ω . We apply the 5 point star finite difference method with fixed and scaled grids. Figure 4.1 shows the scaled performance of the Jacobi-SI and CG semi-iterative methods. 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 less iterations (42) compared to the Jacobi-SI (55).

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} .

In Table 4.2 we give the fixed size speed ups achieved by the parallel Jacobi-SI on the NCUBE after 400 iterations. Table 4.3 lists the numerical values of the scaled performance of the Jacobi-SI 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.

Processors	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.

Processors	NCUBE		iPSC/2	
	Comp. Time	Comm. Time	Comp. Time	Comm. Time
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 4.3 Scaled speed up of Jacobi-SI on NCUBE and iPSC/2 for 4000 equations per processor.

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

Figure 4.2 The effect of iPSC/2 vector processing for Jacobi-SL. 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.

4.2 The performance of domain decomposition Jacobi-SL

In this experiment we consider the partial differential equation $u_{xx} + u_{yy} - (1+x)u = f(x)$ defined on the L-shaped domain depicted in Figure 4.4. The function f 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 L-shaped PDE domain.

The domain decomposition is done heuristically using the *Domain Decomposition Tool* (DecTool) of //ELLPACK and the automatic load partitioning strategies developed in [Hous 89c]. 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 take place. Figure 4.5 shows the per iteration scaled and fixed speed up 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. The final paper will include experimental data indicating the effect of load balancing in the speed up of domain decomposition semi-iterative methods.

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

5. CONCLUSIONS AND FUTURE GOALS

Iterative methods have been shown [Hous 88] to be effective alternatives for solved 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 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 significantly larger problems than cost equivalent shared memory multiprocessor systems. Specifically, we were able to solve close to 1/2 million finite difference equations in a reasonable time on NCUBE with 128 processors.

ACKNOWLEDGMENTS

We are grateful Advanced Computing Research Facility at Argonne Laboratories for allowing us to use their iPSC/1 and to 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. Kortesios and J.R. Rice, *Automatic Load Balanced Partitioning Strategies for PDE Computations*, Purdue University CAPO Technical Report CER-89-7, February 1989.
- [Duni 88] T. H. Dunigan, *Performance of a Second Generation Hypercube*, Tech. Report. ORNL/TM-10881, Oak Ridge National Laboratory, Mathematical Science Section, (1988).
- [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 89a] E.N. Houstis, T.S. Papatheodorou and J.R. Rice, *Parallel ELLPACK: An Expert System for the Parallel Processing of Partial Differential Equations*, (1989), to appear, Math. Comp. Simul., 31.
- [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.
- [Hous 88] 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.
- [Keye 87] D. Keyes, W. Gropp, *A Comparison of Domain Decomposition Techniques for Elliptic Partial Differential Equations and Their Parallel Implementation*, SIAM J. Sci. Stat. Comp. 8, (1987), s166-s202.
- [Kinc 82] D.J. Kincaid, J. Respass, 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.
- [Lich 84] A. Lichnewsky, *Some Vector and Parallel Implementations for Preconditioned Conjugate Gradient Algorithms*. in Proceedings of the NATO Workshop on High Speed Computations (J. Kowalik, ed.), NATO ASI Series, vol. F-7, Springer-Verlag, (1984), 343-359.
- [Rice 85] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, New York, (1985).
- [Seag 86] M. K. Seager, *Parallelizing Conjugate Gradient for the CRAY X-MP*, Parallel Computing 3, (1986), 35-47.