1992

Parallel Iterative Methods

N. P. Christochoides

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

S. B. Kim

M. K. Samartzis

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
92-035

PARALLEL ITERATIVE METHODS

N. P. Chrisochoides
Elias N. Houstis
S. B. Kim
M. K. Samartzis
J. R. Rice

CSD-TR 92-035
June 1992

Parallel Iterative Methods

N.P. Chrisochoides,
E.N. Houstis, S.B. Kim, M.K. Samartzis and J.R. Rice*
Department of Computer Science
Purdue University
West Lafayette, IN 47907

June 12, 1992

Abstract

In this paper we discuss the implementation of the ITPACK library [Kinc 82] in the parallel (//)ELLPACK environment [Hous 92] and report on its performance on the nCUBE II parallel machine. In this study we are concerned with the numerical solution of second order elliptic partial differential equations (PDEs) on rectangular regions with mixed boundary conditions using finite difference approximations. The parallelization methodology applied is based on the domain decomposition of discrete geometric data structures (grids) associated with the numerical solution of the PDE problem[Chri 91]. The implementation of //ITPACK for boundary value problems defined on general 2-D and 3-D domains for both finite element and difference methods is reported in [Kim 93]. The performance results obtained so far indicate almost optimal computational and space efficiency of the //ITPACK modules.

1 Introduction

This paper presents some preliminary results related to the development of parallel iterative methods for solving large linear systems of algebraic equations derived from the discretization of elliptic partial differential equations. Specifically, we discuss the implementation and performance of a set of parallel iterative methods included in the ITPACK library [Kinc 82] for 5-point difference equations obtained by the discretization of second order elliptic PDE on rectangular regions on the nCUBE II. The extension of this library for non-rectangular regions in 2-D and 3-D domains for both finite difference and finite element methods is discussed in [Kim 93]. This library is currently part of the parallel (//)ELLPACK system [Hous 92] and consists of 7 modules listed in Table 1.

*This work was supported in part by AFSOR 92-J-0069, NSF grant CCF-8619817 and ESPRIT project 2702

//ITPACK module	Indexing	Method
SOR	Red/Black	Successive Over-Relaxation
Jacobi-CG	Natural	Jacobi conjugate gradient
Jacobi-SI	Natural	Jacobi with Chebyshev acceleration
RSCG	Red/Black	Reduced system CG
RSSI	Red/Black	Reduced system SI
SSOR-CG	Red/Black	Symmetric SOR CG
SSOR-CG	Red/Black	Symmetric CG

Table 1. Parallel iterative methods in the //ITPACK library.

The parallel implementation of the //ITPACK library reported is based on the $p \times q$ (checkerboard) decomposition of the orthogonal grid of the 5-point finite difference discretization method. Section 2 of this paper contains a discussion about the parallel implementation of the ITPACK library in the //ELLPACK environment and in Section 3, we present the performance of the various //ITPACK modules on the nCUBE II parallel machine.

2 Parallel ITPACK Library

Throughout we assume that the reader is familiar with the theoretical aspects of the methods included in the sequential ITPACK library. A detail description of each ITPACK module can be found in [Kinc 82] and [Rice 85]. In this section, we discuss the parallelization of this library within the //ELLPACK environment based on the domain decomposition approach[Chri 91]. In this paper we consider only the case of parallel iterative solution of 5-point finite difference equations obtained by the discretization of second order elliptic PDEs on rectangular domains.

The ELLPACK environment supports modular programming through predefined data structures and module interfaces. Specifically, the discrete algebraic data structures are stored in a sparse mode using two arrays: `coef` $(\cdot,\cdot)$ containing the non-zero coefficients, `idcoef` $(\cdot,\cdot)$ containing their column indices. The indexing of the unknowns and its inverse are stored in arrays `i1undx` $(\cdot)$ and `i1endx` $(\cdot)$, respectively. Various control parameters are stored in the `ipasm` $(\cdot)$ and `rpasm` $(\cdot)$ arrays.

In the case of parallel ELLPACK the above data structures are local to each subdomain. Furthermore, a new set of data structures has been introduced that hold the communication information related to where to find values that are non-local and where to send values that needed elsewhere. This information is organized into two groups. The first group, named *communication workspace buffers*, contains data indicating where the incoming values are stored. The second group, called *decomposition data structures*, provides all the information needed in order for each processor (subdomain) to know which values to expect from other processors, where these values are coming from, where are they going to be stored locally, how many values are coming, how many values are to be sent, and where to send these values. In the case of the Red/Black (RB) ordering, we need all the above data separately for the red and black points.

Next we describe one of the //ITPACK modules in terms of various subtasks and discuss their parallelization. The same observations apply to the other modules.

2.1 Parallel SOR-CG method with RB ordering

This module, called `g5ibm1` (we use the ITPACK names for routines), is currently implemented to solve the finite difference equations obtained by the 5-point approximation of second order elliptic PDEs defined on rectangular domains with mixed boundary conditions on message-passing machines. Its parallel implementation is based on checkerboard decomposition of the rectangular grid. All modules are capable of computing adaptively the optimal iteration parameters involved.

Basic Subtasks of the g5i6m1 Module:

1. Initialization
2. Check dimension
3. Compute some of the information needed for the communication to be performed
4. Scale the system
5. Remove rows and cells when the off diagonal elements are very "small"
6. Initialize workspace pointers
7. Select indexing scheme and determine appropriate information (this module must use the Red/Black ordering)
8. Permute the system according to the Red/Black information
9. Rearrange system according to the ASIS ordering
10. Check for sufficient workspace and initialize it

—— **start iterative process** ——

11. Initial setups for the iterative process and some more setups for the communication
12. Iterations

—— **post processing of solution** ——

13. Check convergence
14. Put solution in place (a trick is used here)
15. Reverse the permutation of the system
16. Perform error analysis and accuracy estimates
17. Unscale the system
18. Setup return parameters

The modifications of the sequential codes made for parallelization are minimal. We have added two new routines `q5i9cv`, `q5i9cr` that set up the information needed for the communication and initialize the communication work space, `q5i9cv` is for natural ordering and q5i9cr for RB ordering. In subtask 11 where the iteration starts, we need to send the black values of the current approximation, as well as to compute the total number of equations in our problem. This is done by the routine `rbdex` for RB ordered systems and `bdex` for natural (ASIS) ordered systems. Both routines use bidirectional exchange to send the data. Apart of these new routines the rest of the code is the same. The values that need to be communicated can be found by looking at the arguments of the `mesg`, `mesg1` subroutines (these routines are defined later). These are mainly, unknowns and pseudo-residuals.

The code for the iteration routine **q5i8i6** is in general the same. We had to insert calls to `mesg` or `mesg1` that handle the communication when data from other nodes are needed. Also, we had to insert calls to routines like `r1bdod` that performs the dot product for a vector distributed over the processors. The new

routines needed to support the parallelization of this part of the computation are: rbdex, rbcast, clpsadd, r5brd, r1bdod. Specifically, in the sequential q5i8i6 we have changed the calls to r1bldo to calls to r1bdod and added two calls to mesg1. The first mesg1 call is just before a call to r5i9pb, where we need to have the black backward pseudo residual values and the second one is just before a call to r5ibrd. There we need to have all the backward pseudo residual values, but since the black ones are already in place, we send only the red ones. These values are also used by q5i9p5.

Communication also takes place when matrix-vector operations appear in the method. The routines q5i9pf and q5i9bs also contain calls to mesg1. Other routines like q5i9st and r5i9pb contain calls to r1bdod. It is worth pointing out that calls to mesg, mesg1 are used when the method is about to perform some kind of matrix-vector multiplication.

2.2 Implementation of message-passing for //ITPACK modules

The communication requirements of the various //ITPACK modules are independent of the particular method considered but depend on the ordering assumed. Their implementation requires that correct data are sent to neighbors and stored in predetermined locations from where they can be retrieved using q5i9gr (we discuss this routine separately since it is an essential part of the parallel implementation).

The communication information is implemented by the routines mesg and mesg1. These two routines are similar. The routine mesg1 is the one that can handle the case of a color-ordered (e.g., red/black) system. Specifically, each of these routines gets as input (argument) the data to be sent to neighboring processors (subdomains) and then receives the values these processors send for this subdomain. The incoming data from other processors are stored in the communication buffer rcomb. We have set up 5 pointers into this buffer:

lpwe	→	points to the first value received from	west
lpea	→	points to the first value received from	east
lpso	→	points to the first value received from	south
lpno	→	points to the first value received from	north
lpwo	→	points to the first value of the rest of the buffer which is used as scratch space for copying	

The values are stored in the following order: West, East, South, North, Work. The routines q5i9cv, q5i9cr set the values for the variables nptor(4), nptos(4), nrptor(4), nbptor(4), nbptos(4), nrptos(4), which indicate:

nptor	→	number of values to receive
nptos	→	number of values to send
nrptor	→	number of RED values to receive
nrptos	→	number of RED values to send
nbptos	→	number of BLACK values to send
nbptor	→	number of BLACK values to receive

Each of these variables is a four element vector with each element containing the specific number for one of the four neighbors of the subdomain. For example, the elements of nrptos are:

(1)	# of RED values to send	west
(2)	# of RED values to send	east
(3)	# of RED values to send	south
(4)	# of RED values to send	north

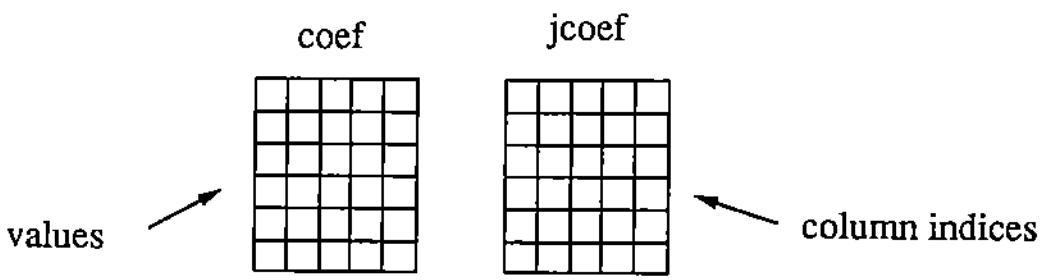
The routines first send all the local data that need to be sent and then receive the appropriate data. To send the local data is not straightforward. We use routines *gather* (for mesg) and *gath* for (mesg1) that collect the appropriate values to be sent in a continuous buffer (rcomb (lpwo)). The routines *gather* and *gath* work similarly, but *gath* also has to use some coloring information. Both get as an argument, the matrix numunk produced by 5pstar, that contains the local numbering of the grid points (equations). The argument idir contains the direction where the data collected by *gather* or *gath* is to be sent.

In the routine *gather*, since the assumed geometry is simple, we only have to traverse the correct side of the grid, get the number of the equations from the numunk matrix, check that these points are active (indicated by numunk(.,.) > 0), and collect the value. For non-active points we get a value of 0 for numunk. In the routine *gath* we also have to use the ordering information provided by vector l1undx. We traverse the side that is indicated by *idir* and first discard non-active points (numunk(.,.) <= 0). Then we check if the point is of the correct color by comparing its number with the number of the red points in the subdomain. The values collected are put in consecutive spaces in the workspace. It is the responsibility of the receiver to unpack the message correctly. This is done by the routine *scat* which uses information from a mask vector ip(·). This vector is set up by the routine q5i9cr and has the color information for the incoming data.

After collecting the data to send, we send them out and start receiving data. In the case of mesg, the incoming data are in the correct format and can be put directly into the correct locations of the communication buffer rcomb. mesg1 has to use routine *scat* that scatters incoming data into the correct positions of the rcomb vector.

Another important operation needed for the implementation of the communication among subdomains is *getting values out of the communication buffers.* An interesting thing about ITPACK is the way matrix vector multiplications are performed using the coef, idcoef structures. The usual code looks like the following (also see the diagram):

```
      do  20    jj=2, maxn2
          call  q5i9gr(n,u,jcoef(1,jj), work(  ))
          do    10 = 1,n
              v(ll) = v(ll) - coeff(ll,jj)*work(ll)
10        continue
20    continue
```



We perform the multiplication columnwise, with respect to the columns jcoef of coef. (In loop 20 column 1 never takes part, since it is the diagonal which we know that contains 1.0's). The routine q5i9gr collects the unknowns u that are to be multiplied with the coefficients in the jj column of coef into the work vector. Then loop 10 performs the multiplication. For the parallel implementation, we know that some values of the unknown are not local and are sent by neighboring processors. These values are stored somewhere (for our case in the communication buffer rcomb). *If* q5i9gr *has the ability to know which values are local and which are non-local and where to find these non-local values; nothing from the above loop should be changed. The only thing we need is to add, before the beginning, a call to* mesg, *so to ensure that the correct values of u are indeed stored in the communication buffer.* So q5i9gr is the complement of the mesg, mesg1 routines which send and receive non-local data and *place* it in the correct positions in the buffers; the routine g5i9gr collects the values it needs from these buffers to rearrange them in order for the multiplication to be performed. For the case of rectangular domains and the checkerboard decomposition, we have set up a scheme that does just

that. It provides `q5i9gr` the information whether a value it seeks is local or non-local. If the value is non-local, then there is also available the position in the communication buffer where the value should be. Since the call to `mesg` is before the loop, we know that the correct values have to be in place (thanks to the `mesg` and `mesg1` routines). Below we explain how this is done, but notice that this scheme is specific to the rectangular case when we number the equations, the unknowns and generate the equations. In the discretization, we number non-local unknowns with special numbers. These special numbers encode 3 pieces of information:

1. that this unknown is non-local,
2. where this unknown is local, i.e., which neighbor owns this unknown, and
3. where it is placed by `mesg` in the communication buffer when it is received.

Let n be the index of an unknown in the subdomain which has (local) dimension *ngrx by ngry*. Then $1 \leq n \leq ngrx * ngry$ if the unknown is local and $n > ngrx * ngry$ if the unknown is non-local. If n is non-local its value is computed as follows:

$$n = (ngrx * ngry * 10) * \text{direction} + \text{position}$$

direction = 2 if the unknown is local to the subdomain to the west
= 3 if the unknown is local to the subdomain to the east
= 4 if the unknown is local to the subdomain to the south
= 5 if the unknown is local to the subdomain to the north

position = is the position in the buffer, the correct part of the buffer given by the respective pointer (`ipwe`, `ipea`, etc.)

3 Performance Results

In this section we present some performance data of the //ITPACK modules listed in Table 1 for 5-point star finite difference equations on the nCUBE II parallel machine. The parallel 5-point star discretization module used for obtaining the algebraic data computes the "optimal" decomposition for the given machine configuration and grid size or accepts values for the $p \times q$ decomposition from the user. Table 2 presents raw time and error data for all phases of the numerical solution of a given PDE problem with the Jacobi-CG parallel iterative solver.

Time and error measures	Number of processors						
	1	2	4	8	16	32	64
Discretization (% of total)	3.433	3.498	3.563	3.661	3.852	3.912	4.032
Indexing (% of total)	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Solution (% of total)	96.566	96.500	96.437	96.333	96.136	96.041	95.878
Communication (% of total)	0.000	0.114	0.205	0.606	1.039	1.746	2.710
Discretization (sec)	17.227	8.801	4.496	2.352	1.266	0.656	0.352
Indexing (sec)	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Solution (sec)	484.602	242.781	121.695	61.875	31.590	16.109	8.359
Total solve time(sec)	501.836	251.586	126.191	64.230	32.859	16.773	8.719
Communication (sec)	0.008	0.281	0.254	0.379	0.332	.281	0.227
Per iteration (sec)	4.846	2.428	1.217	0.619	0.316	0.161	0.084
max abs error	7.97E-2	8.90E-2	9.16E-2	9.03E-2	8.96E-2	8.94E-02	8.94E-2
# of iterations	100	100	100	100	100	100	100

Table 2. The performance of a single precision parallel implementation of Jacobi-CG for a model elliptic PDE problem defined on the unit square. A fixed 200 × 200 grid is used and the process is terminated after 100 iterations.

It is worth observing that the communication cost is a small percentage of the entire computation cost (smaller than 3% for a 200x200 grid on a 64 processor configuration) for all //ITPACK modules. Table 3 indicates a very impressive fixed speedup and Table 4 shows almost 96% scaled speedup for the SOR module. The data is similar for all the //ITPACK modules. Our data indicate that the communication cost peeks at small configurations. This is due to the contention problem, since the connectivity of the decomposition does not match the connectivity of the hypercube for a small number of processors. The speedups computed with respect to the time of the //ITPACK modules on a single processor. It has been observed by Mo Mu that the original sequential modules are a little faster for the PDE model problem considered here. This is due primarily to the selection of the data structures used for storing communication and domain decomposition information. The more general version of //ITPACK performs as well as the sequential ITPACK on a single processor [Kim 93]. We believe that the results presented here prove both the scalability of parallel iterative methods and the almost optimal behavior of the parallel implementation.

Processors	Jacobi-CG	Jacobi-SI	RSCG	RSSI	SSOR CG	SSOR SI
2	1.99	1.99	1.99	1.99	1.99	1.99
4	3.98	4.00	3.93	3.94	3.96	3.97
8	7.81	7.86	7.65	7.67	7.73	7.72
16	15.27	15.36	14.72	14.79	15.01	14.96
32	29.92	30.25	28.32	28.59	29.14	29.08
64	57.56	58.69	52.89	54.06	55.62	55.52

Table 3. The fixed speedup obtained on nCUBE II by each //ITPACK module for a model PDE problem and grid size 200 × 200 after 100 iterations.

# Processors	Grid Size	Per Iteration Time
1	102 × 52	.573
2	102 × 102	.574
4	202 × 102	.578
8	202 × 202	.584
16	402 × 202	.590
32	402 × 402	.592
64	802 × 402	.599

Table 4. The scaled speedup for the SOR module. The table gives the time for one iteration when the number of equations per processor is held constant.

References

[Chri 91] Chrisochoides, N.P., Houstis, E.N., Houstis, C.E., Papachiou, P.N., Kortesis, S.K., and Rice, J.R., "Domain Decomposer: A software tool for mapping PDE computations to parallel architectures", *Fourth International Symposium on Domain Decomposition Methods for Partial Differential Equations*, (Edited by R. Glowinski, Y.A. Kuznetsov, G., Meurant, J. Periaux, O.B. Widlund), SIAM, pp. 341–357, (1991).

[Hadj 89] Hadjidimos, A., Houstis, E.N., Rice, J.R., Samartzis, K.M., and Vavalis, E.A., "Semi-iterative methods on distributed memory multiprocessor architectures", *Proceedings of International Conference on Supercomputing*, (Edited by E.N. Houstis and D. Gannon), ACM press , pp. 82–90, (1989).

[Hous 92] Houstis, E.N., and Rice, J.R., "Parallel ELLPACK: A development and problem solving environment for high performance computing machines", *Programming Environments for High-Level Scientific Problem Solving*, (Edited by P.W. Gaffney and E.N. Houstis), Elsevier Science Publishers B.V. (North-Holland), pp. 229–241, (1992).

[Kim 93] Sang Ban Kim, *Parallel Numerical Methods for Partial Differential Equations*, Ph.D. Thesis, Purdue University, to appear.

[Kinc 82] Kincaid, D.J., Respess, J., Young, D.M., and Grimes, R., "Algorithm 586: ITPACK 2c: A Fortran package for solving large sparse linear systems by adaptive accelerated iterative methods", *ACM Trans. Math. Soft.* 8, pp. 302–322, (1982).

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