Experience with the Yeremin Algorithm on IBM-SPx*

Elise deDoncker and Ajay Gupta

Department of Computer Science, Western Michigan University, Kalamazoo, MI 49008-5021, U. S. A.
e-mail: {elise,gupta}@cs.wmich.edu

Abstract

In this paper we discuss our experiments with a parallel preconditioned conjugate gradient solver for large, sparse, symmetric systems on IBM-SPx. The algorithm uses Factorized Sparse Approximate Inverse (FSAI) preconditioning [8] and is layered over MPI.

1 Preconditioned CG Algorithm

This work was motivated in part by need for solving large, sparse linear systems arising in Finite Element computations [2, 1]. We implemented a CG algorithm with FSAI preconditioning [8] on a 128 processor nCUBE-2 [4] and on a network of SUN workstations. In the present paper we focus on the performance characteristics of the algorithm on an IBM-SP system running MPI [7].

Let $Ax = b$ be a positive definite, symmetric system of size $n \times n$. The FSAI preconditioned CG method [8] solves a system of the form $G^\tau GAx = G^\tau Gb$, with $G = diag(G_L)^{-1/2}G_L = diag(G_L A G_L^\tau)^{-1/2}G_L$, and G_L satisfies $\tilde{G_L} = diag(L_A)G_L$, where $\tilde{G_L}$ is a least Frobenius norm approximation to the inverse of the Cholesky factor L_A of A, with respect to a given sparsity pattern.

For the sparsity pattern of G we choose that of the lower triangle of A. If g_i denotes the subrow of the (n_i) non-zero elements on row i of G_L, then these elements are obtained by solving the $n_i \times n_i$ linear system $S_i {g_i}^\tau = [0, \ldots, 0, 1]^\tau$, where S_i is the principal submatrix of A corresponding to the element locations of g_i within row i of A.

For discussions of the efficiency of parallel preconditioned CG methods and stability considerations see, e.g., [3, 9, 5]. Issues regarding the convergence behavior of finite precision Lanczos and CG computations are furthermore studied in [6]. We have tested the algorithm for arbitrary sparsity patterns of A. The sparse matrices A and G have their non-zero elements stored by means of the element value and column number. We use a block row distribution of A and G, where processor k has block row A_k of A, the corresponding part of the right hand side and block row G_k of G. The number of rows in A_k and G_k is determined in such a way that each processor has about the same share of the non-zero elements of A. The local vectors of the CG iteration are distributed accordingly.

Matrix-vector multiplications with A and G assume the block row representation of the matrix in a straightforward way, whereas the pre-multiplication with G^τ uses the block column representation of G^τ. The use of MPI is illustrated for the latter in the code listed

*Supported in part by NSF CCR-9405377.

below. The *MPI_Barrier* calls are used to separate the communications involved in consecutive global operations, in view of the fact that the *MPI_Allreduce* is tag-less.

```
#include "mpi.h"
#include <stdio.h>

struct t {
        float value;
        int  column;
};
typedef struct t data;

void cmvec(node,mat,no_of_col,x,prd,n)

int node; data **mat; int no_of_col; float *x, *prd; int n;

/*  Matrix vector product mat^t * x where the calling process has
    no_of_col columns of mat^t, and the corresponding els. of the vector x;
    the contributions from all processors to x are gathered. The product
    vector is returned in prd. */
{
    float  *part, *vector();
    int    i, j, source;

    part = vector(1,n);
    for ( i=1; i<=n; i++ ) part[i] = 0.0;

    for ( j=1; j<=no_of_col; j++)
       for(i=1;i<=mat[j][0].column;i++)
             part[mat[j][i].column] += mat[j][i].value*x[j+1];

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Allreduce(&part[1], &prd[1], n, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    free_vector(part,1,n);
}
```

Taking into account different network characteristics, the algorithm analysis is similar to that given on a hypercube topology in [4]. Each CG iteration is dominated by three matrix-vector products (involving A, G^τ and G) and two inner products. In summary, in view of the distribution of A, the computation time for the matrix-vector multiplication Av is $\mathcal{O}(NZ(A)/p)$, where $NZ(A)$ is the number of non-zero entries in A and p is the number of processors.

The communication time for acquiring v in all processors can be given as $\sigma + \tau(p+n)$, where τ^{-1} is the data transfer rate and σ is the latency of the network. For the multiplication of the form $G^\tau v$, processor k has block column G_k^τ. The vector v need not be broadcast but the contribution of all processors to the elements of the result vector are accumulated. If n is large, the network protocol (such as TCP/IP) will partition v in a number of packets. Hence, the communication time will then be $\sigma w + \tau(p + \frac{n}{w})$ where $\frac{n}{w}$ is the packet size.

If, for the distribution of G, the same as that of A is used, the computational effort of each processor for solving the subsystems needed for the block rows of G is $\mathcal{O}((\frac{NZ(A)}{p})^3)$. Note that, in the uniform case where each processor has n/p rows of approximately $NZ(A)/n$ elements of A each, the work for solving the subsystems at each processor is $\mathcal{O}(\frac{(NZ(A))^3}{pn^2})$, while the total work is $\mathcal{O}(\frac{(NZ(A))^3}{n^2})$; which is optimal.

2 Numerical Experiments

For our experiments we used the Argonne National Laboratory IBM-SPx system, which is essentially an IBM-SP2 with SP1 nodes. The ANL SPx consists of 128 nodes and two compile servers. According to the ANL IBM-SPx User's Guide (http://www.mcs.anl.gov/Projects/sp), each node is an RS/6000 model 370; the model has a 62.5 MHz clock, a 32-KB data cache, and a 32-KB instruction cache. Key features of the system are: 128 MBytes of memory per node; 1 GByte local disk on each node (400 MBytes available to users, rest for paging and the operating system); a full Unix on each node (IBM AIX 3.2.5); each node accessible by Ethernet from the Internet; high-performance Omega switch (63 μ-sec latency, 35 MBytes/sec bandwidth when using "user space" option). In addition, the ANL SPx has a large high-performance file system (220 GBytes of RAID disk and a 6-TByte automated tape library). The peak performance of each node is 125 MFlops (1 64-bit floating-point add and 1 floating-point multiply in each clock cycle).

We collected results for large, sparse matrices (e.g., size 20000×20000). We examined the times for the preconditioning phase including/ not including the times for receiving the local systems. Analogously, for the CG phase we considered the times including/ not including receiving of the block rows.

Below we will refer to the test examples as x-y where x is the order of the system and y equals twice the maximum number of elements before (and including) the diagonal, on any row of the matrix. The non-zero matrix elements and their column numbers were generated at random, as well as the right hand side of each system.

In Figures 1-4 we give timing results obtained for test cases 3000-100, 5000-50, 10000-100 and 20000-50. The times plotted for preconditioning (—) and CG (...) are those not including the data distribution at the beginning of the two stages. The total time (plotted in full line) does include the times for receiving the local systems and the block rows of A in each processor for the preconditioning and the CG stage, respectively. The time for sending the data, which is only incurred in one processor, is not included. The maximum time reported by any one processor is given.

Apparently, the preconditioning times dominate in our test examples, especially for large systems. This is due to the fact that very few CG iterations are needed after preconditioning. Table 1 displays the values of $\mathbf{r}^2$ (= the square of the residual) for successive iterations.

On the ANL SPx, the CG times reveal speedups for $1, \ldots, 4$ (sometimes to about 8) processors for the examples under consideration. For higher numbers of processors, the communication times involved in the global operations degrade the parallel performance. The behavior is similar, e.g., on ethernet connected workstations. However, on systems where the network speed is relatively fast - as compared to the processor speed, the CG iterations continue to speed up for higher numbers of processors. For example, for 3000-100 on nCUBE-2, the CG time continued decreasing through 64 processors on a 128p machine [4] (that time included the receiving of the block rows). This warrants running the CG phase in parallel on such machines.

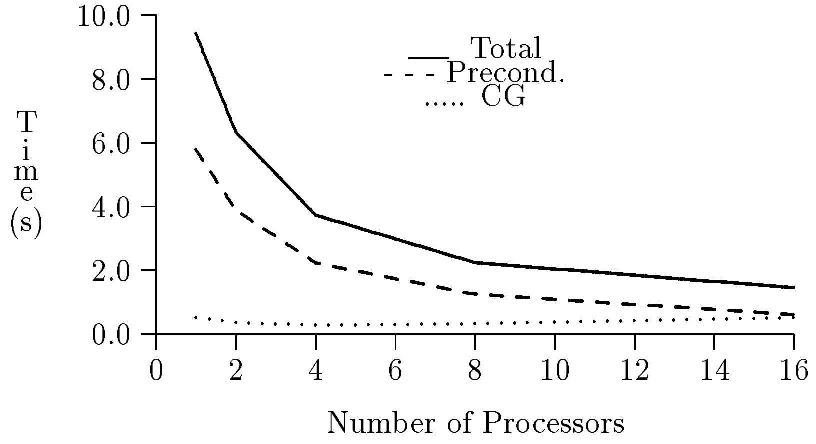

Figure 1: Times for 3000-100

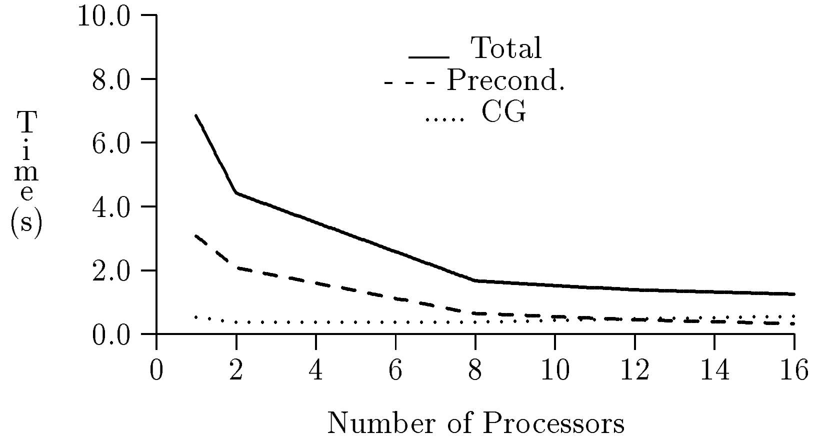

Figure 2: Times for 5000-50

	Iteration Number						
Test case	0	1	2	3	4	5	6
3000-100	186933.57	367.48	0.703951	0.000915	0.000002	0.000000	—
5000-50	84236.03	272.17	0.866025	0.002283	0.000006	0.000000	—
5000-100	342045.75	1160.49	6.216941	0.417507	0.004005	0.000031	0.000002
10000-100	722312.75	3932.91	54.184944	3.212626	0.039358	0.000509	0.000027
20000-50	351731.68	2789.99	33.050797	2.672791	0.039405	0.000534	0.000026

Table 1: r^2 values during CG iterations

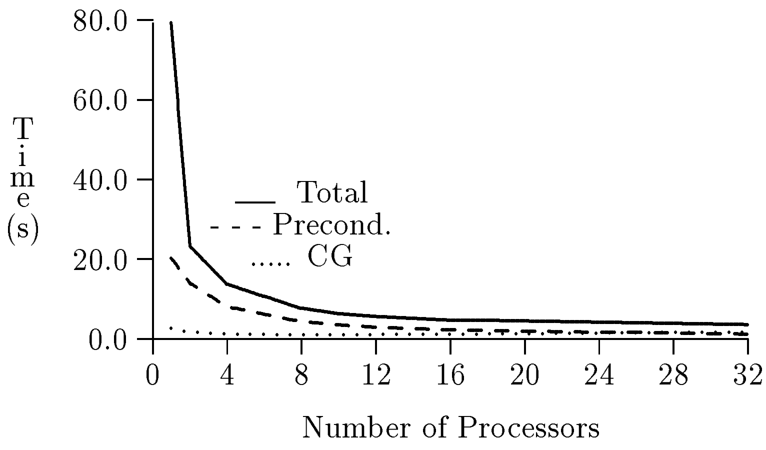

Figure 3: Times for 10000-100

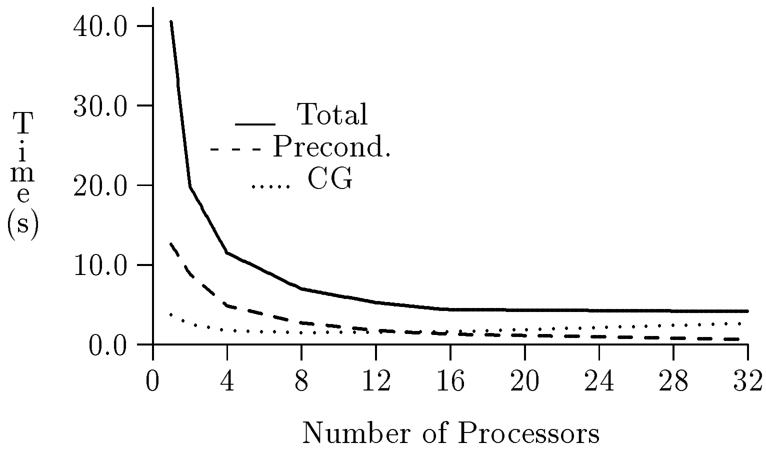

Figure 4: Times for 20000-50

In view of the memory capacity of each node on the ANL SPx, one approach might be to have the entire matrix representation available in each node, run the preconditioner computation in parallel, and the few necessary CG iterations locally. However, the latter would require to also have the whole G available in each processor - where, currently, we generate each block row locally within its processor; this results in a trade-off in view of the communication which would be involved in distributing G. Since the preconditioner computations are completely local, they allow for a good parallel performance. Thus, on the ANL SPx it is indeed desirable to have the whole A available in each processor, thus avoiding the communications incurred in distributing the block rows/ local systems. Note that the times corresponding in this approach are those of the Precond. and CG times in Figures 1-4.

Acknowledgement.
The authors gratefully acknowledge use of the Argonne High-Performance Computing Research Facility. The HPCRF is funded principally by the U. S. Department of Energy, Mathematical, Information and Computational Sciences Division (ER-31).

References

[1] J. Ball, E. de Doncker, P. Ealy, and A. Gupta. Adaptive integration for finite element type problems. In A. Tentner, editor, *Proceedings of High Performance Computing 1996*, pages 238–243. The Society for Computer Simulation, 1996.

[2] R. L. Burden and J. D. Faires. *Numerical Analysis.* Prindle, Weber & Schmidt, 3rd edition, 1985.

[3] A. T. Chronopoulos and C. W. Gear. s-step iterative methods for symmetric linear systems. *J. on Comp. and Appl. Math.*, 25:153–168, 1989.

[4] E. de Doncker and A. K. Gupta. Coarse grain preconditioned conjugate gradient solver for large sparse systems. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 472–477, 1995.

[5] J. W. Demmel, M. T. Heath, and H. A. van der Vorst. Parallel numerical linear algebra. Technical report, U. C. Berkeley Computer Science Division, 1993. UCB//CSD-92-703.

[6] A. Greenbaum and Z. Strakos. Predicting the behavior of finite precision lanczos and conjugate gradient computations. *SIAM J. Matrix Anal. Appl.*, 13(1):121–137, 1992.

[7] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* The MIT Press, 1994.

[8] L. Yu. Kolotilina and A. Yu. Yeremin. Factorized sparse approximate inverse preconditioning ii: Solution of 3d fe systems on massively parallel computers. Technical report, Elegant Mathematics, Inc., 1992.

[9] H. Van der Vorst. The convergence behavior of preconditioned CG and CG-S in the presence of rounding errors. In O. Axelsson and L. Yu Kolotilina, editors, *Preconditioned Conjugate Gradient Methods, Lecture Notes in Mathematics*, volume 1457. Springer-Verlag, Berlin, 1990.