An Experimental Study of Methods for Parallel Preconditioned Krylov Methods

Doug Baxter, Joel Saltz, Martin
Schultz, Stan Eisenstat and Kay Crowley

Research Report YALEU/DCS/RR- 629
June 1988

The authors were supported in part by the Office of Naval Research under Grant N00014-86-0564, Distribution Unlimited.

Abstract

High performance multiprocessor architectures differ both in the number of processors, and in the delay costs for synchronization and communication. In order to obtain good performance on a given architecture for a given problem, adequate parallelization, good balance of load and an appropriate choice of granularity are essential.

We discuss the implementation of parallel version of PCGPAK for both shared memory architectures and hypercubes. Our parallel implementation is sufficiently efficient to allow us to complete the solution of our test problems on 16 processors of the Encore Multimax/320 in an amount of time that is a small multiple of that required by a single head of a Cray X/MP, despite the fact that the peak perfomance of the Multimax processors is not even close to the supercomputer range. We illustarate the effectiveness of our approach on a number of model problems from reservoir engineering and mathematics.

Table of Contents

1. Introduction

The analysis of the multiprocessor performance achievable by a robust and efficient iterative sparse linear system solver is of great interest for the following reasons:

1. general linear solvers are of great interest in their own right as they can form the key computational kernal for a very wide variety of problems and
2. such a solver present varied workloads to multiprocessor machines and require those constructing programs to parallelize, cluster or aggregate work and assign work to processors without the benefit of *a priori* problem specific information.

In this paper we utilize a Krylov method solver, PCGPAK, a package produced and marketed by Scientific Computing Associates Inc. of New Haven, Connecticut. These types of iterative methods frequently out-perform direct methods [9] and preconditioning with approximate LU factorizations is often crucial in obtaining fast efficient convergence of Krylov iterative methods. These preconditioners require a lower and upper sparse triangular solve at each iteration.

With increases in both the numbers and the performance of processors in multiprocessor machines the organization of multiprocessor memory in a hierarchical manner becomes increasingly attractive from an architectural point of view. In order to achieve good performance on machines with hierarchical memory structures, for example hypercubes, work must be organized so that data accesses are reasonably local. Because the data dependencies in sparse matrix iterative algorithms are controlled by the matrix given to the program, an appropriate partition of the problem is not obtainable before the matrix is presented. Iterative sparse matrix algorithms consequently present a particularly interesting challenge from the standpoint of the developing methods for automated runtime problem partitioning.

Experimental results are presented using the Intel iPSC hypercube. The iPSC is a message passing or fragmented memory machine in which interprocessor interactions are handled via sending messages. The messages sent have relatively high communication latencies. We regard message passing machines such as the iPSC as having very pronounced memory hierarchies; there is a very large performance penalty to be paid when programs must move data between processors frequently and/or in large quantities.

Our results make clear the crucial importance of appropriate runtime partitioning. The efficiencies obtained on this machine overall, were substantially inferior to those obtained on the Encore Multimax. However with our methods, we are able to demonstrate that we can obtain good efficiencies for problems of a moderate size.

In Section 2 we briefly present the preconditioned Krylov methods and describe the test problems used in this paper. In Section 3 we describe the shared memory parallel implementations of the major tasks in the basic preconditioned Krylov method in PCGPAK. In Section 4 we present the performance results gained from these implementations and demonstrate that as long as the triangular solve is dealt with properly, excellent results can be obtained for a rather wide variety of problems on a closely coupled shared memory machine such as the Encore Multimax. We present experimental data that reflects on the usefulness of parallel preconditioners in Section 5, and in Section 6 we discuss the role played by memory hierarchies in determining performance of sparse matrix computations and present performance data obtained on the Intel iPSC. Finally we present our conclusions in Section 7.

2. Preconditioned Krylov Methods Background

We briefly present the basics of Krylov methods such as are found in PCGPAK. The interested reader may find the details of GMRES(k) in [11] and other methods are described in [2, 5]

Consider a large, sparse, system of linear equations of the form

$$Mx = b \tag{2.1}$$

where M is a real matrix of order N, b is a given vector of length N and x is unknown vector to be computed.

Given an initial guess x_0, Krylov methods generate an approximate solution x_i from the translated Krylov space $x_0 + K_i$ where

$$K_i \subset span\{r_0, Mr_0, ..., M^{i-1}r_0\}.$$

x_i is usually chosen to minimize some norm of its residual $b - Mx_i$ [10].

The basic tasks involved in Krylov methods are sparse matrix-vector multiplies with matrix M, additions of scalar multiples of vectors to other vectors (SAXPYs), and vector inner-products. The latter are used in determining the linear combination of Krylov vectors to add to the initial guess so as to minimize the norm of the residual.

Preconditioned Krylov methods consist of using an auxiliary matrix $Q = Q_l Q_r$ to first generate the preconditioned system

$$(Q_l^{-1} M Q_r^{-1}) Q_r x = Q_l^{-1} b$$

Where Q is some approximation to M and $Q_l^{-1}v$ and $Q_r^{-1}v$ for any vector v are easy to compute. The approximate LU factorization preconditioners [5, 4, 8] take Q to be LU where L is lower triangular and U is upper triangular. Hence the preconditioned matrix-vector multiply in the resulting Krylov method consists of doing a forward and backward sparse triangular solves as well as the sparse matrix multiply by M. It is in this context that we are interested in solving linear systems for sparse triangular (lower or upper) matrices.

2.1. The Test Problems

We now present the eight test problems used in our experiments.

Problem 1 (SPE1) This problem models the pressure equation in a sequential black oil simulation. The grid is $10 \times 10 \times 10$ with one unknown per gridpoint for a total of 1000 unknowns.

Problem 2 (SPE2) This problem arises from the thermal simulation of a steam injection process. The grid is $6 \times 6 \times 5$ with 6 unknowns per grid point giving 1080 unknowns. The matrix is a block seven point operator with 6×6 blocks.

Problem 3 (SPE3) This problem comes from an IMPES simulation of a black oil model. The matrix is a seven point operator on a $35 \times 11 \times 13$ grid yielding 5005 equations.

Problem 4 (SPE4) This problem also comes from an IMPES simulation of a black oil model. The matrix is a seven point operator on a $16 \times 23 \times 3$ grid giving 1104 equations.

Problem 5 (SPE5) This problem arises from a fully-implicit, simultaneous solution simulation of a black oil model. It is a block seven point operator on a $16 \times 23 \times 3$ grid with 3×3 blocks yielding 3312 equations.

Problem 6 **(5-Pt)** This problem is a five point central difference discretization of the following equation on the unit square:

$$-\frac{\partial}{\partial x}(e^{-xy}\frac{\partial}{\partial x}u) - \frac{\partial}{\partial y}(e^{xy}\frac{\partial}{\partial y}u) + 2(x+y)(\frac{\partial}{\partial x}u + \frac{\partial}{\partial y}u) + (2 + \frac{1}{1+x+y})u \;=\; f$$

with Dirichlet boundary conditions and f chosen so that the exact solution is

$$u \;=\; x\,e^{xy}\sin(\pi x)\sin(\pi y).$$

The discretization grid is 63 × 63 giving 3969 unknowns. The L5-pt problem is the same problem with a 200 × 200 grid.

Problem 7 **(9-pt)** This problem is a nine point box scheme discretization for the following equation on the unit square:

$$-(\frac{\partial^2}{\partial x^2}u + \frac{\partial^2}{\partial y^2}u) + 2\frac{\partial}{\partial x}u + 2\frac{\partial}{\partial y}u \;=\; f$$

with Dirichlet boundary conditions and f chosen so that the exact solution is

$$u \;=\; x\,e^{xy}\sin(\pi x)\sin(\pi y).$$

The discretization grid is 63 × 63 giving 3969 equations. The L9-pt problem is the same problem with a 127 × 127 grid.

Problem 8 **(7-pt)** This problem is a seven point central difference discretization of the following equation on the unit cube:

$$-\frac{\partial}{\partial x}(e^{xy}\frac{\partial}{\partial x}u) - \frac{\partial}{\partial y}(e^{xy}\frac{\partial}{\partial y}u) - \frac{\partial}{\partial z}(e^{xy}\frac{\partial}{\partial z}u) + 80(x+y+z)\frac{\partial}{\partial x}u + (40 + \frac{1}{1+x+y+z})u \;=\; f$$

with Dirichlet boundary conditions and f chosen so that the exact solution is

$$u = (1-x)(1-y)(1-z)(1-e^{-x})(1-e^{-y})(1-e^{-z}).$$

The discretization grid is 20 × 20 × 20 yielding 8000 equations. The L7-pt problem is the same problem with a 30 × 30 × 30 grid.

3. Parallel Implementations of the Basic Krylov Method

3.1. SAXPY operations, Vector inner-products, and Sparse matrix-vector multiplies

The iterative loop of PCGPAK performs three basic computational tasks:

1. sparse matrix-vector products;
2. scalar-vector products and vector inner-products;
3. sparse triangular forward- and back-solves;

We shall discuss the straightforward parallelization of scalar vector-products, vector inner products and the sparse matrix-vector multiply first, we shall then discuss the parallelization of the sparse triangular solves.

The parallel implementations of the SAXPY, vector inner-products and sparse matrix-vector products are similar. For p processors and a linear system of order n the indices from 1 to n are divided into p contiguous groups of roughly equal size. The i^{th} group is assigned to the i^{th} processor. The local variables IMIN and IMAX contain the limits of the indices associated with the processor.

3.1.1. SAXPY operations

The SAXPY routine

```
do i=1,n
  y(i) = y(i) + a*x(i)
enddo
```

is implemented in parallel as

```
do i=imin,imax
  y(i) = y(i) + a*x(i)
enddo
call waitbar
```

where WAITBAR is a barrier (i.e. WAITBAR does not return until each of the p processes has called WAITBAR).

3.1.2. Vector inner-products

The vector inner-product routine (SDOT)

```
sum = 0
do i=1,n
```

```
  sum = sum + x(i)*y(i)
enddo
```

is implemented in parallel as

```
sum = 0
do i=imin,imax
  sum = sum + x(i)*y(i)
enddo
sum = sumbar(sum)
```

where SUMBAR is a summation barrier (i.e., a value is passed to SUMBAR by each of the p processes and, when all of these values have been summed, the total is returned to each process).

3.1.3. Sparse Matrix-vector multiplies

The sparse matrix-vector product routine

```
do i=1,n
  ax(i) = a(i)*x(i)
  do ij=ija(i),ija(i+1)-1
    ax(i) = ax(i) + a(ij)*x(ija(ij))
  enddo
enddo
```

is implemented in parallel as

```
do i=imin,imax
  ax(i) = a(i)*x(i)
  do ij=ija(i),ija(i+1)-1
    ax(i) = ax(i) + a(ij)*x(ija(ij))
  enddo
enddo
call waitbar
```

where WAITBAR is again a barrier.

3.2. Parallel Triangular Solves

The sparse triangular systems obtained through incomplete factorizations of sparse linear systems allow for the concurrent substitution of a substantial number of rows [13, 1, 7]. It is not difficult to partition the rows of a triangular matrix into a sequence of sets where all rows within a given set can be solved for concurrently. We will discuss only the algorithm and results for solution of a lower triangular matrix L, the situation applying in the upper triangular case is essentially identical.

A directed acyclic graph G can be generated that depicts the order in which variables, described by rows in L, can be solved. The evaluation of rows in L are represented by the vertices of G, and the data dependencies between the rows by G's edges. The dependence of matrix row a on matrix row b is represented by an edge going from vertex b to vertex a. A topological sort may be performed which partitions the directed acyclic graph (DAG) into wavefronts. A stage of this sort is performed by alternately removing all vertices that are not pointed to by edges, and then removing all edges that emanated from the removed vertices. All vertices removed during a given stage constitute a wavefront; the wavefronts are numbered by consecutive integers. An adaptation of a common topological sort algorithm [6] allows the wavefronts of a DAG to be calculated efficiently.

The wavefronts calculated through this process can be utilized directly in implementing a very general method for scheduling the row substitutions required for the solution of the triangular equations. The row substitutions in any wavefront may be executed simultaneously. A very straightforward method for solving the problem is consequently to partition the problem's solution into *phases,* each of which is dedicated to a given wavefront. On shared memory machines, the straightforward application of this technique requires a global synchronization between phases. The major sources of inefficiency in this simple method are:

1. imbalance of load due to differing quantities of work assigned to processors during a given phase and
2. the cost of the global synchronization.

Detailed experimental and theoretical analysis of these sources of inefficiency may be found in [7] [3] along with several extensions of the above algorithm.

4. Parallel Results

In this section, we present benchmarks for using various solution techniques offered by PCG-PAK to solve the benchmark problems discussed in Section 2.1 and we identify the computational tasks that require the most CPU time. These results are for a single processor on the Encore Multimax/320.

The results are shown in Table 1, which is divided into five broad categories, showing the method and preconditioner used, the iteration count needed to reach the stopping criteria; the

				Percents of Total Time					
Test Problem	GMRES (K)	Precond-itioner	Number of Iterations	Reduced System	Factor-ization	Matrix Vector Product	Triangular Solves	SAXPY and SDOT	Total Time (Seconds)
SPE1	1	ILU(0)	26	17	6	26	33	17	3.6
SPE2	1	ILU(0)	11	–	23	31	37	9	10.1
SPE3	10	MILU(0)	24	15	11	25	28	20	24.1
SPE4	10	ILU(0)	17	21	12	22	26	19	3.1
SPE5	20	ILU(0)	43	–	2	17	22	59	64.0
5-PT	5	MILU(0)	20	17	6	23	30	23	17.6
9-PT	10	ILU(2)	20	–	20	18	38	24	47.9
7-PT	1	ILU(0)	20	–	6	28	38	27	46.4
L5-PT	4	MILU(0)	54	8	3	27	34	28	410.7
L9-PT	10	ILU(1)	80	–	5	22	37	36	603.3
L7-PT	1	ILU(0)	31	–	4	29	39	28	239.9

Table 1: Breakdown of Times for single processor PCGPAK on the Encore Multimax/320 with APC-02s

percentage of CPU time spent in the reduced system computation when that methodology was used; the percentage of time spent forming the preconditioner; and the percentage of time spent in the various subtasks of the basic preconditioned algorithms. The column labeled "SAXPY and SDOT" corresponds to item 2 of the basic tasks. These computations involve adding a scalar multiple of one vector to another vector (SAXPY operation) and computing vector inner-products (SDOT operation). The stopping criteria used was to reduce the Euclidian norm of the residual by six orders of magnitude, starting with an initial guess of 0.

These results show that the sparse matrix-vector products occupy roughly 20% to 30% of the CPU time, the sparse triangular solves occupy roughly 20% to 35% of the time, and the SAXPY and SDOT operations take from 25% to 40% (with the exceptions of SPE2 (10%) and SPE5 (60%)). When reduced system preprocessing is used, it takes from 7% to 18% of the CPU time. The formation of the preconditioner takes from 1% to 10% of the time (except for SPE2 (24%)). Thus, matrix-vector products, sparse triangular solves, and SAXPY and SDOT operations dominate the total CPU time.

All these experiments used the "right-oriented" preconditioning, *i.e.* the preconditioned problem was

$$MQ^{-1}\hat{x} = b, \qquad x = Q^{-1}\hat{x}$$

where M is the coefficient matrix and Q is the preconditioner. The initial guess was $x_0 \equiv 0$.

The timings were done on an Encore Multimax/320 with 13 megahertz APC/02 boards an version 2.1 of the FORTRAN compiler.

The parallel implementations of the SAXPY, SDOT and Sparse matrix-vector multiply routines all run at efficiencies well above 90% for all of the model problems. We do see some superlinear speedups for the model problems in these operations due to increased cache size when more processors are used. However, when speeds are compared between problems that fit in the cache for a range of number of processors used, we still get above 90% efficiency. To be conservative in our projections in Section 4.2 we use the 90% efficiency mark for doing comparisons.

4.1. Triangular Solve results

In Table 2 we give the measured efficiencies of the parallel sparse triangular solve using the wavefront parallelization methods described above. The efficiencies are computed by taking the time from a sequential sparse triangular solve on a single processor and dividing by the number of processors multiplied by parallel sparse triangular solve times. The sequential code has **none** of the parallel overhead in it.

We note that among the problems depicted, SPE1, SPE3 and SPE4 have the lowest efficiencies for 16 processors. These problems are quite small; we performed symbolic operation count based analysis to estimate the efficiency that would be achieved given our mapping of workload to processors in the absence of any other overhead costs and the number agree closely (within 10%) to the observed efficiencies.

As expected, the efficiencies for all problems tend to decrease with increasing numbers of processors. This decrease is only a trend; changes in the distribution of load can in some cases cause small *improvements* in efficiency with increasing numbers of processors. For the five, nine and seven point templates 16 processor efficiencies ranged from 48% to 77%.

4.2. Projections for total times for the model problems

For the Encore Multimax/320, the speedups attainable for 75% to 95% of the computation performed in PCGPAK have been demonstrated in the preceding sections. In this section we project the expected performance of entire PCGPAK package for the model problems on the Multimax/320.

In the course of solving linear systems arising from time dependent and /or nonlinear partial

Efficiencies for Triangular Solves (Percent)

Test Problem	Number of processors														
	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SPE1	72	59	50	42	40	35	32	29	27	27	25	23	22	20	20
SPE2	85	73	70	62	55	54	52	50	42	42	40	38	37	34	36
SPE3	86	73	66	59	53	47	47	42	41	39	34	32	32	31	31
SPE4	76	67	57	49	43	40	33	26	26	28	27	28	27	25	24
SPE5	82	85	75	77	66	73	61	69	71	58	55	56	58	52	48
5-PT	92	89	83	79	76	78	71	64	63	66	61	58	53	50	48
9-PT	94	89	86	81	74	67	75	67	61	56	52	49	47	51	55
7-PT	92	92	86	83	88	89	83	87	71	85	79	80	81	77	77
L5-PT	94	91	88	85	83	80	78	76	75	75	71	68	69	65	61
L9-PT	95	92	88	86	85	83	76	74	71	73	68	63	68	65	61
L7-PT	93	90	89	85	82	87	85	85	74	84	78	82	80	65	77

Table 2: Multiprocessor Triangular Solves on the Multimax/320 with APC-02 boards.

differential equations, many sparse linear systems need to be solved. Most often the sparsity pattern in these systems does not change from one time step (or Newton-like step) to the next. PCGPAK allows the user to take advantage of this and save the symbolic preprocessing results from the first (or previous) call to be used in subsequent calls. Table 3 shows a further breakdown of the setup phase components into numeric and symbolic parts for the Multimax/320.

In Table 4 we present the expected times for the nonsymbolic part of the PCGPAK computation for multiple processors. The computation of the approximate factorization has the same organization as the lower triangular solve, and hence the wavefront methodology may be used to gain speedups as demonstrated in Section 4.1. The numeric part of the reduced system computation is comparable to the matrix-vector product and we assume speedups of 90% for this computation. We have excluded the symbolic part here as it may be amortized over many solves.

In Table 5 we present the projected efficiencies for the model problems on the Encore Multimax/320 with APC-02 CPU boards. For the larger problems (SPE5 through L7-PT) the efficiencies are above 60% for 16 processors, and above 85% for five or fewer processors. Even for the smaller problems the efficiency is above 60% with fewer than six processors.

It is clear from the data in Table 1 that the efficient parallelization of the triangular solve is essential for obtaining acceptable efficiencies for the problem as a whole. An intersting question is how well a parallelizing FORTRAN compiler would do on this subproblem. Since the data dependencies would not be known *a priori* to such a compiler, the best it can do is to parallelize of

Percents of Total Time for Setup Phase

Test Problem	Precond-itioner	Reduced System		Factorization	
		Symbolic	Numeric	Symbolic	Numeric
SPE1	ILU(0)	4.3	12.5	0.7	5.1
SPE2	ILU(0)	–	–	1.5	23.3
SPE3	MILU(0)	3.6	11.5	3.3	7.3
SPE4	ILU(0)	5.2	15.9	4.1	7.5
SPE5	ILU(0)	–	–	0.3	2.0
5-PT	MILU(0)	4.4	13.0	0.7	5.6
9-PT	ILU(2)	–	–	11.2	9.2
7-PT	ILU(0)	–	–	0.9	5.3
L5-PT	MILU(0)	2.0	5.9	0.3	2.5
L9-PT	ILU(1)	–	–	2.5	2.0
L7-PT	ILU(0)	–	–	0.6	3.6

Table 3: Single processor PCGPAK on the Encore Multimax/320 with 13 MHz APC-02s

Total Times (Excluding Symbolic Setup) (seconds)

Test Problem	Number of processors															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SPE1	3.4	2.1	1.5	1.2	1.1	.9	.9	.8	.8	.7	.7	.6	.6	.6	.6	.6
SPE2	9.9	5.8	4.3	3.3	2.9	2.6	2.3	2.0	1.9	1.9	1.7	1.6	1.6	1.5	1.5	1.3
SPE3	22.4	12.5	8.9	7.0	5.9	5.2	4.7	4.2	3.9	3.6	3.4	3.3	3.2	3.0	2.8	2.7
SPE4	2.8	1.7	1.2	.9	.8	.7	.7	.6	.7	.6	.5	.5	.4	.4	.4	.4
SPE5	63.8	36.4	24.0	18.6	14.8	12.9	10.7	9.9	8.5	7.6	7.3	6.8	6.3	5.8	5.6	5.4
5-PT	16.7	9.2	6.2	4.8	3.9	3.3	2.8	2.5	2.4	2.1	1.9	1.8	1.7	1.7	1.6	1.5
9-PT	42.5	23.2	15.9	12.2	10.1	8.8	8.0	6.6	6.2	5.9	5.7	5.5	5.3	5.0	4.4	4.0
7-PT	46.0	25.1	16.7	12.9	10.1	8.5	7.3	6.6	5.7	5.7	4.7	4.5	4.1	3.8	3.6	3.4
L5-PT	401.3	218.7	147.5	112.1	90.9	76.4	66.5	58.8	52.8	47.8	43.4	40.8	38.3	35.4	33.9	32.7
L9-PT	588.2	318.2	214.8	164.0	132.5	110.9	96.0	87.3	78.5	72.0	64.6	61.2	58.6	52.5	50.1	48.4
L7-PT	238.0	130.9	88.5	66.7	54.4	46.1	38.5	34.0	30.2	29.0	24.9	23.6	21.3	20.0	20.6	17.8

Table 4: Total Projected Times on the Multimax/320 with APC-02 boards

the computations in any one row. In Figures 1 and 2, we depict for varying numbers of processors the following efficiencies:

1. that obtained from the sparse triangular solve parallelized by scheduling rows concurrently;

Percent Efficiency

Test	Number of processors														
Problem	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
SPE1	83	75	69	62	61	56	53	49	47	47	44	42	40	38	38
SPE2	85	78	75	70	64	63	61	60	52	52	50	48	47	44	46
SPE3	90	84	80	76	72	68	68	63	63	61	55	54	54	53	53
SPE4	85	80	74	69	64	62	55	47	47	50	48	50	48	46	45
SPE5	88	88	86	86	83	85	80	84	84	79	78	78	79	76	74
5-PT	91	90	88	86	84	85	82	78	78	79	77	75	72	70	68
9-PT	92	89	87	85	80	76	81	76	72	68	65	62	60	64	67
7-PT	92	92	89	87	90	90	87	90	82	88	85	86	87	84	84
L5-PT	92	91	90	88	88	86	85	84	84	84	82	81	81	79	77
L9-PT	92	91	90	89	88	88	84	83	82	83	80	77	80	78	76
L7-PT	91	90	90	88	86	88	88	88	82	87	84	86	85	77	84

Table 5: Projected Total Efficiencies

2. that obtained from parallelizing only the inner-products of the triangular solves, *i.e.*,parallelizing computations within a row;
3. that obtained for the entire iterative portion of the calculation, when the triangular solve parallelized by scheduling rows concurrently, and
4. that obtained for the entire iterative portion of the calculation that would result when only the inner-products of a triangular solve are parallelized.

These graphs make it clear that runtime interrow parallelization is crucial if we are to obtain good efficiencies in this situation. Thus the intra-row parallelization potentially available from parallelizing compilers cannot provide the desired performance.

4.3. Comparison of Parallel with Vector Timings

We can compare the projected times obtained for the iterative portion of our calculation with execution times on a single processor Cray X/MP. The Cray timings are for generic PCGPAK in FORTRAN77 [9] compiled with version 1.3 of the CFT77 compiler. The results are shown in Table 6 We note that for large three dimensional problems, the sixteen processor Encore is roughly the speed of a Cray X/MP. This is quite surprising as the Multimax/320 is basically a super minicomputer composed of 16 single chip mircomputers (National Semiconductor 32332S) which costs about on tenth as much as the Cray X/MP. This illustrates the power of parallel computing as distinct from vector computing.

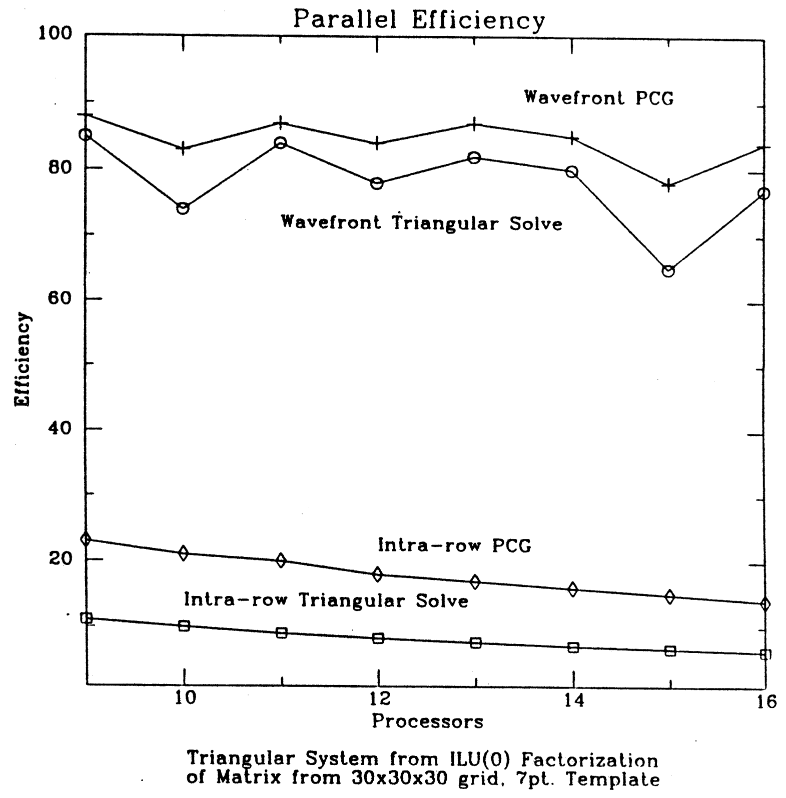

Figure 1:

We are currently in the process of creating an integrated parallel Preconditioned Krylov solver. We have encountered in the integrated package some unanticipated inefficiencies arising from somewhat obscure origins. Despite these difficulties, the current performance measurements still come within approximately 25 percent of our predictions.

In addition, the clock speed of the Multimax is currently 13.5 MHZ; this will be increased to 15MHZ in the near future. We consequently fully expect our performance predictions to be met or exceeded. We also mention that the Cray timings might be improved by as much as a factor of 2 by use of compiler directives and standard library routines. This still does not alter the flavor of our result: the parallel Multimax is a significant fraction of a Cray. In addition the incomplete factorization and the reduced system preprocessing do not vectorize to any significant extent, but these computations can be expected to run at least as fast as the triangular solves on the Encore.

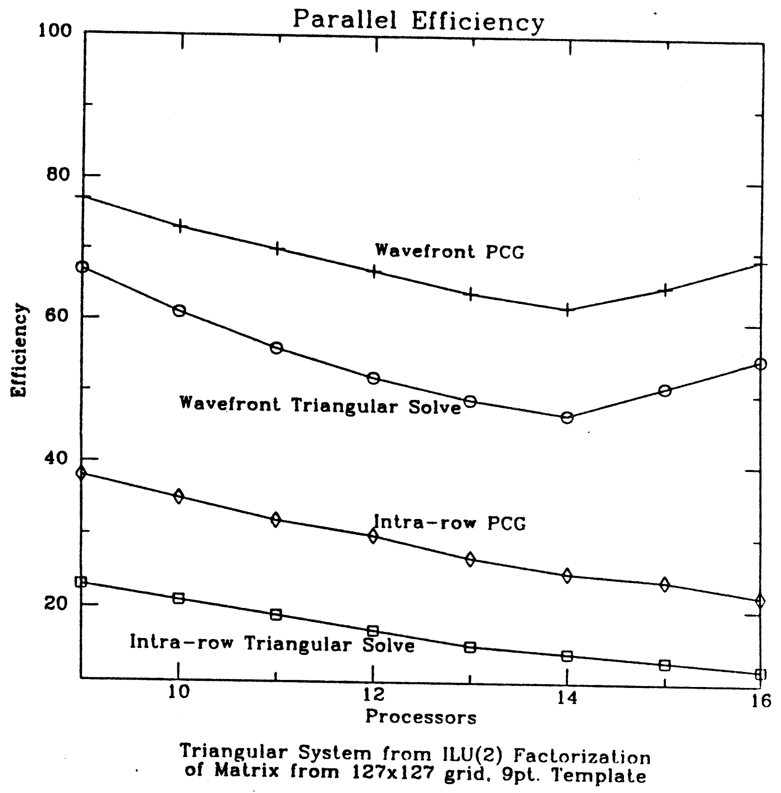

Figure 2:

Multimax/Cray Comparison

Test Problem	Multimax (16)	Cray X/MP	Ratio
SPE1	.48	.11	4.4
SPE2	.93	.15	6.1
SPE3	2.11	.44	4.8
SPE4	.30	.06	5.0
SPE5	5.21	1.04	5.0
5-PT	1.25	.50	2.5
9-PT	3.47	1.02	3.4
7-PT	3.20	2.04	1.6
L5-PT	29.99	12.57	2.4
L9-PT	47.17	15.30	3.1
L7-PT	17.09	10.13	1.7

Table 6: Total Iterative Time (seconds)

5. Parallelized Preconditioners

We shall now consider the tradeoffs involved in using simpler preconditioners that parallelize essentially perfectly or by not preconditioning at all and using many direction vectors in GMRES to ensure convergence (this parallelizes on the Multimax with efficiencies of $\approx$ 90 %). We shall see that there are no clear advantages to the use of such preconditioners, at least in forgiving multiprocessor environments such as that provided by the Multimax. In Table 7 we display the results of using GMRES with a large number of direction vectors and no preconditioning or reduced system preprocessing. The stopping criteria used in Table 1 was also used for results in Table 7

Test Problem	GMRES (K)	Number of Iterations	SAXPY and SDOT	Matrix-Vector Product	Actual Total iterative time with preconditioning
SPE1	100	718	1.24	.15	.049
SPE2	100	>2000	5.37	1.84	.073
SPE3	100	>2000	17.45	1.84	.301
SPE4	100	120	.20	.02	.042
SPE5	100	>2000	11.53	2.83	.294
5-PT	100	265	1.68	.31	.252
9-PT	100	140	.81	.30	.566
7-PT	100	86	1.02	.36	.478
L-9PT	100	403	11.33	4.21	4.206
L-7PT	50	263	6.39	4.08	2.379

Table 7: Projected times for the Cray X/MP with no preconditioning and no reduced system (in seconds)

In the column labeled *iterations* in Table 7 the indication > 2000 indicates that the method failed to converge in 2000 iterations. The last column indicates the total run times from a vectorized version of PCGPAK on the Cray X/MP where preconditioning and reduced system preprocessing were used as indicated in Table 1. The column labeled "SAXPY and SDOT" are the estimated times for the GMRES operations (excluding matrix-vector multiplies) assuming a speed of 120 megaflops. The column labeled "Matrix-Vector Product" are the estimated times using the speeds measured on the Cray for each matrix-vector product. The sums of the SAXPY and Matrix columns give the estimated total time for GMRES(K) without preconditioning and reduced system preprocessing.

The performance of a polynomial preconditioning with a degree d polynomial and GMRES(m) may be estimated from the number of iterations shown in Table 7 by dividing the number of iterations in the table by d (as long as $d * m$ is less than K in column 2 of the table). The

SAXPY/SDOT times for such a method would be reduced by at best a factor of d and the matrix-vector product times at best would remain the same since the number of iterations would at best be reduced by a factor of d.

Hence a comparison of the data in this table clearly shows that polynomial preconditioning is unlikely to outperform the approximate factorization preconditionings with reduced system preprocessing because of the large number of iterations required. Of course, a practical difficulty incurred by polynomial preconditioning is determining the coefficients to use for the polynomial.

6. Distributed Memory Performance

Experimental results are presented using the Intel iPSC hypercube. The iPSC is a message passing or fragmented memory machine in which interprocessor interactions are handled via sending messages. The messages sent have relatively high communication latencies. We regard message passing machines such as the iPSC as having very pronounced memory hierarchies; there is a very large performance penalty to be paid when programs must move data between processors frequently and/or in large quantities. As has been discussed above, the efficient implementation of PCGPAK hinges on obtaining satisfactory performance for the sparse triangular solve. We present only results for the sparse triangular solve, implementation and benchmarking of the other procedures discussed above on the iPSC is currently underway.

High performance multiprocessor architectures differ both in the number of processors, and in the delay costs for synchronization and communication. In order to obtain good performance on a given architecture for a given problem, an appropriate choice of granularity is essential. The partitioning and mapping of the computation should be performed in a way that is able to take advantage of the multiprocessor architecture.

Experimental results measuring the performance of triangular solves on the Intel iPSC hypercube are presented in this section. As has been previously discussed, the triangular solve proves to be a crucial factor in determining the multiprocessor performance achievable by Krylov space methods when preconditioning using incompletely factored matrices is employed. The efficiencies obtained from the triangular solve represent a pessimistic lower bound on the performance that can be obtained by the iterative portion of the algorithm. We present here only the results pertaining to the triangular solve, the work on implementing and benchmarking other portions of the algorithm on the iPSC is still in progress.

Clustering or aggregation methods can be used to reduce the number of communication startups and the volume of information that must be communicated between processors. The effective use of these methods is essential for obtaining satisfactory results on machines such as the iPSC. The method used to aggregate is discussed in detail in [7] and involves two steps. In the first step, a sort of coordinate system is obtained for the DAG.

This coordinate system is obtained through a process of peeling off layers of the DAG. It is then straightforward to map the problem to a multiprocessor in a manner that restricts the fan-in and fan-out of data between processors. To the extent allowed by the data dependencies in the algorithm, it also becomes possible to map work so that only nearby processors have to communicate. Finally, the coordinate system is used to allow the specification of work clusters in a parametric manner.

In the applications discussed here, the clustering of work is controlled by two parameters, the *block size* which describes the number of consecutive DAG layers assigned to a processor and the *window size*, or number of wavefronts per block. The reduction in communication overhead is however, achieved at the risk of load imbalance, making this the critical tradeoff. The reader is referred to [7] for further details regarding this issue.

In Table 8 we present results from two moderate sized problems solved on a 32 node Intel iPSC hypercube. The problems used were the L5-pt problem and a larger version of the L5-pt problem on a 300×300 grid. Recall that the triangular system is formed in this case using the reduced system, so that we are using triangular systems with 20000 and 45000 rows in the L5-pt problem and the larger version of the L5-pt problem respectively. The inter-row data dependencies are obviously also changed when the reduced system is formed.

We depict the total parallel efficiency, the total execution time, the number of computational phases and the estimated communication time. Note that the best efficiencies we are able to obtain increase with the size of the problem. In the smaller of the two problems the best efficiency is 31% while in the larger problem the best efficency increases to 53%. These efficiencies in the absence of aggregation were 12% and 16% respectively.

The ability to aggregate work plays a central role in extracting increased efficiency from larger problems. When matrix rows are assigned to processors in an unaggregated manner, each row corresponding to an interior mesh point must communicate its value to another processor. Consequently, the communication volume does not tend to decrease as problems become larger. The above data did reveal a small improvement in efficiency when the larger problem was solved even in the absence of aggregation; the amount of computational work per phase does decrease as problem size grows, even in the absence of aggregation.

200 x 200 point mesh

Window Block Size	Efficiency (%)	Total Time	Phases	Communication Time
1 – No grey code	6%	4.58	398	
1	12%	2.34	398	1.92
2	21%	1.35	200	1.06
4	31%	0.90	100	0.45
6	24%	1.21	67	0.65
8	18%	1.54	50	0.57

300 x 300 point mesh

Window Block Size	Efficiency (%)	Total Time	Phases	Communication Time
1 – No grey code	8%	7.61	598	
1	16%	3.99	598	3.45
2	31%	2.07	299	1.57
4	53%	1.21	149	0.88
6	44%	1.44	99	0.65
8	33%	1.95	75	

Table 8: Larger Triangular Solves (Times in seconds)

The aggregation here is performed using graph techniques on sparse matrices. In the absence of a method that is able to capture the geometrical relationship between matrix rows, it is not possible to optimize the *mapping* of the unaggregated problem. In 8, we also depict the performance figures when the problems were executed without the use of grey coding. This was a conservative attempt to estimate the effect of not taking the problem geometry into account. In this case we still map in a manner that ensures that the assignment of work to processors was such that each processor needed to communicate with only one other processor in each phase. In the larger of the two problems our mapping and aggregation methods made the difference between an 8% efficiency and a 53% efficiency.

Aggregation allows a tradeoff between communication costs and time wasted due to load imbalance. For a given size machine, as problem size grows, one can obtain the same load balance by aggregating work into increasingly large chunks. This leads to increasingly favorable ratios of computation to communication costs.

In Table 9 we summarize experimental results for a variety of triangular solves implemented on the iPSC. We present parallel efficiency values for a 32 processor cube; the sequential code used in calculating the efficiency values was a separate sequential code run on one node of the hypercube.

The choice of aggregation parameters in the problems presented in 9 was made through a process of symbolically estimating the balance of load that would result from differing degrees of aggregation, using a method to be described below. Using this estimated load balance along with an estimate of communication time we calculated an estimate of the optimal degree of aggregation. The development of robust heuristics for choosing the degree of aggregation is underway, we do not address this issue here.

Percent Efficiencies

Template	Mesh size	Preconditioner	% Efficiency
5 pt.	63 x 63	Reduced system MILU(0)	7.0
9 pt.	63 x 63	ILU(2)	11.0
7 pt.	20 x 20 x 20	ILU(0)	10.0
5 pt.	200 x 200	Reduced system MILU(0)	31.0
9 pt.	127 x 127	ILU(1)	22.0
7 pt.	30 x 30 x 30	ILU(0)	30.0
5 pt.	300 x 300	Reduced system MILU(0)	53.0

Table 9: Triangular Solves: 32 node Intel iPSC

The timings presented were obtained by parallelizing and aggregating the triangular solve computations. The efficiencies obtained for many of the problems listed in 9 were quite low.

It is natural to inquire as to the degree to which one might expect to improve those efficiencies though improved programming, mapping and scheduling. There is clearly a tradeoff made between costs of communication and the costs of load imbalance as granularity is increased. While one can reduce communication costs by aggregating work in small triangular solves, load imbalances increase rapidly with aggregation. These effects are documented in detail in [7] and will be examined below in the context of a model problem.

A great deal of care needs to be taken in ensuring that the actual code on message passing machines functions efficiently. Without the appropriate techniques, high overheads can result from managing and coordinating the execution of a single irregular problem on a number of processors with separate address spaces. One way of determining the role of these inefficiencies is to compute an *estimated optimal runtime*, which approximates the runtime that would be observed with the actual work distribution but in the absence of any other multiprocessing overheads. We perform an operation count based analysis at the time the workload is aggregated, this yields an estimated speedup. The execution time of the separate sequential program on one processor divided by this estimated speedup gives us this estimated optimal runtime. One can also estimate the time required for communication, by maintaining the pattern of communication and specifying the correct message sizes but deleting all computation and all actual data movement required to pack messages. In an efficient code, the communication costs and the estimated optimal runtime should approximately add up to the actual runtime.

We consider below the results of these analysis performed on a small model problem. This problem will will be presented using data from a triangular system generated from a zero fill incomplete factorization of a sparse matrix generated by a 120x120 five point template. The tradeoffs between load imbalance and communication costs in this model problem have been formally analyzed in some detail [13], [7]. In Table 10 we depict parallel efficiency, estimated optimal time, the total time required to solve the problem on a 32 node Intel Hypercube and the estimated communication time. The communication time estimate is obtained by running problems in which computation is deleted but communication patterns are maintained. We note that when we employ a very fine grained parallelism (window and block size equal to one), we pay a very heavy communication penalty relative to the computation time. The completion of this non-computationally intensive problem requires 240 phases, each one of which requires processors to both send and receive data. We can reduce the number of computational phases, and hence reduce the communication time but we do this at the cost of increasing the computation time since the available parallelism is degraded due to load imbalances. While appropriate choice of computational granularity is essential for maximizing computational efficiency, the nature of the triangular solve limits the performance that can be obtained in small to intermediate sized problems. The example here illustrates this well, we obtain a three fold improvement in efficiency through a moderate increase in granularity, i.e. speedup increases to 5.8 from 1.9. The absolute efficiency obtained, even given appropriate choice of granularity is still quite limited.

The communication times added to the estimated optimal times yield quantities that are reasonably close to the total time indicating that the code is not likely to contain gross inefficiencies that exaggerate the difficulties involved in solving this small problem. We note that only rough correspondence is expected as the assumptions made in the operation count analysis are clearly overly simplistic.

Window Block Size	Efficiency (%)	Total Time	Estimated Optimal Time	Communication Time
1	6%	1.25	.09	1.09
2	12%	.60	.11	.49
4	18%	.40	.15	.25
8	15%	.48	.31	.10
10	13%	.56	.36	.08

Table 10: Granularity effects for triangular solves (Times in seconds)

7. Conclusions

We have discussed the implementation of a parallel version of PCGPAK for both shared memory architectures and hypercubes. We implemented and benchmarked portions of the code on the Encore Multimax/320 and have found that our parallel implementation is efficient enough to allow us to complete the solution of our realistic test problems on 16 processors in an amount of time that is a small multiple of that required by a single head of a Cray X/MP, despite the vast disparity in the peak performance of the different types of processors.

The key performance bottleneck identified was the forward and back solution of the the sparse triangular system of equations formed from an incomplete factorization of the matrix. We solved this system using a method capable of detecting and exploiting the data dependencies between row substitutions. To examine the need for runtime detection of parallelism, we performed benchmarks that compared estimated performance that results from exploiting only the parallelism that is available from individual row substitutions. While an effective parallelizing compiler should be able to exploit the parallelism resulting from individual row substitutions, the data dependencies between the matrix rows are clearly determined at runtime and are unavailable to any compiler.

We briefly examined some of the alternatives to using incompletely factored matrices to precondition the conjugate gradient algorithm. Data was presented that suggested that the high operation counts required to solve problems using these other preconditioners posed serious limitations on usefulness of these preconditioners.

The sparse triangular solve was identified as the principal bottleneck to the efficient parallelization of PCGPAK. Data was presented data on the performance of the sparse triangular solve on a message passing machine, the Intel iPSC hypercube. The relatively high communication latencies lead to a striking deterioration in performance. Dramatic improvements in performance could be obtained for moderate sized problems by clustering or aggregating the computational work.

We regard the performance estimates obtained for the triangular solve as pessimistic lower bound estimates on the efficiency that could be achieved by PCGPAK as a whole on the iPSC.

The work clustering methods discussed above are all performed on the basis of data dependencies exhibited at runtime. It follows that this kind of optimization cannot be performed by a compiler. The methods used in parallelizing and clustering the triangular solve are being generalized and are will be incorporated into an automated runtime mapping and scheduling system called PARTY [12]. The system is being designed for use in conjunction with a parallelizing compiler but can also be accessed directly in user programs.

References

[1] A. Greenbaum, *Solving Sparse Triangular Linear Systems Using Fortran with Paralllel Extensions on the NYU Ultracomputer Prototype,* Report 99, NYU Ultracomputer Note, April 1986.

[2] Rati Chandra, *Conjugate Gradient Methods for Partial Differential Equations,* Ph.D. Thesis, Department of Computer Science, Yale University, 1978. Also available as Technical Report 129.

[3] D. M. Nicol and J. H. Saltz, *Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors,* Technical Report 87-39, ICASE, September 1987.

[4] Todd Dupont, Richard P. Kendall and H. H. Rachford Jr., *An approximate factorization procedure for solving self-adjoint elliptic difference equations,* SIAM Journal on Numerical Analysis, 5 (1968), pp. 559–573.

[5] Howard C. Elman, *Iterative Methods for Large, Sparse, Nonsymmetric Systems of Linear Equations,* Ph.D. Thesis, Department of Computer Science, Yale University, 1982. Also available as Technical Report 229.

[6] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Rockville, Maryland, 1978.

[7] J. Saltz, *Automated Problem Scheduling and Reduction of Synchronization Delay Effects,* Report 87-22, ICASE report, July 1987.

[8] J. A. Meijerink and H. A. van der Vorst, *Guidelines for the usage of incomplete decompositions in solving sets of linear equations as occur in practical problems,* Journal of Computational Physics, 44 (1981), pp. 134–155.

[9] Scientific Computing Associates, *PCGPAK: Benchmarks for the FPS and CRAY/XMP,* Technical Report 111, Scientific Computing Associates, New Haven Connecticut, 1987.

[10] ———, *PCGPAK: User's Guide,* Technical Report 106, Scientific Computing Associates, New Haven Connecticut, 1984.

[11] Youcef Saad and Martin H. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems,* SIAM Journal on Scientific and Statistical Computing, 7 (1986), pp. 856–869.

[12] Joel Saltz, Ravi Mirchandaney, Roger M. Smith, David M. Nicol and Kay Crowley, *The Automated Crystal Runtime System: A Framework,* Technical Report 588, Yale University Department of Computer Science, January 1988.

[13] Y. Saad, M. Schultz, *Parallel Implementations of Preconditioned Conjugate Gradient Methods,* Department of Computer Science YALEU/DCS/TR-425, Yale University, October 1985.