Parallel Gaussian Elimination Using OpenMP and MPI

S.F.McGinn and R.E.Shaw
Department of Applied Statistics and Computer Science
University of New Brunswick
Saint John, N.B. Canada E2L 4L5
shawn@unb.ca and reshaw@unbsj.ca

Abstract

In this paper, we have presented a parallel algorithm for Gaussian Elimination. Elimination in both a shared memory environment, using OpenMP, and in a distributed memory environment, using MPI. Parallel LU and Gaussian algorithms for linear systems have been studied extensively and the point of this paper is to present the results of examining various load balancing schemes on both platforms. The results show an improvement in many cases over the default implementation.

1. Introduction

Given a system **Ax = b**, we can utilize several different methods to obtain a solution. If a unique solution is known to exist, and the coefficient matrix is full, a direct method such as Gaussian Elimination is usually selected. There are several papers that emphasize various parallel approaches to solving a system with Gaussian Elimination [1,2,6,8]. In this paper, we are concerned with examining the effect of different load balancing schemes available with OpenMP in a shared memory environment and on a distributed platform where MPI was used as the message passing interface.

Some work has been done on load balancing for Gaussian Elimination such as the article by Howe and Bratcher [7] which compares cyclic and block mapping schemes. A good parallel algorithm for Gaussian Elimination is difficult, however, because of the inherent dependencies in the algorithm, plus the corresponding load balancing issues.

Both versions of the algorithm were run on an IBM RS/6000 SP. This machine has 4 distributed nodes, where each node consists of 4 processors contained within a shared memory environment [3]. With this machine, you have the ability to run programs exclusively within the shared environment, or within the distributed environment, or you can run programs that take advantage of both. Other platforms were used for testing, but the SP results have been kept based on the fact that we could test all of our programs on the same architecture.

2. OpenMP Parallel Version

The first parallel program uses OpenMP to distribute the work among the processors in a shared memory environment. (see Figure 1) The results show a substantial increase in performance over the sequential version. Various load balancing schedules affect the performance of the resulting code and are specified at runtime with a schedule clause.

```
    do pivot = 1, (n-1)
!$omp parallel do private(xmult) schedule(runtime)
        do i = (pivot+1), n
            xmult = a(i,pivot) / a(pivot,pivot)
            do j = (pivot+1), n
                a(i,j) = a(i,j) - (xmult * a(pivot,j))
            end do
            b(i) = b(i) - (xmult * b(pivot))
        end do
!$omp end parallel do
    end do
```

Figure 1.
OpenMP parallel version of the forward elimination algorithm

With a static scheme and a specified chunk size, each processor is statically allocated *chunk* iterations. The allocation of iterations is done at the beginning of the loop, and each thread will only execute those iterations assigned to it. Using static without a specified chunk size implies the system default chunk size of *n/p*. Using a dynamic scheme, each thread is allocated a chunk of iterations at the beginning of the loop, but the exact set of iterations that are allocated to each thread is not known.

Table 1.
CPU time (seconds) with n = 400 and p = 4

Chunk	default	1	2	4	8	16	32	64	128
Static	0.74	1.46	1.81	1.77	1.15	0.82	0.77	0.66	0.57
Dynamic	2.27	2.53	2.38	2.11	1.41	0.97	0.76	0.61	0.56
Guided	0.78	0.80	0.78	0.81	0.74	0.69	0.68	0.68	0.59

Table 2.
CPU times (seconds) with n = 800 and p = 4

Chunk	default	1	2	4	8	16	32	64	128
Static	8.35	20.89	21.66	21.41	17.50	11.48	10.27	9.47	10.27
Dynamic	22.63	22.54	22.10	28.59	19.21	11.66	9.59	9.74	10.39
Guided	9.33	9.53	9.28	9.47	9.49	9.10	8.95	9.84	11.10

Table 3.
CPU times (seconds) with n = 1200 and p = 4

Chunk	default	1	2	4	8	16	32	64	128
Static	51.01	65.69	66.54	65.57	63.01	56.26	54.88	53.61	53.06
Dynamic	85.38	85.54	85.46	82.27	69.88	51.45	42.54	42.09	43.65
Guided	46.10	46.55	46.24	45.71	45.25	44.58	43.61	43.50	43.24

Table 4.
Load Balancing Speedup results using varying values of n

	400	800	1200
Static	3.95	4.59	2.84
Dynamic	4.02	4.00	3.44
Guided	3.81	4.28	3.35

The guided scheme allocates a system dependent chunk of iterations between threads at the beginning of the loop. It is similar to dynamic scheduling such that once a thread has completed its work it is allocated a new chunk of iterations. The difference is that the new chunk size of iterations decreases exponentially as the iterations available decreases to a specified minimum chunk size. If no chunk size is specified, the minimum is 1 [4].

In our results, (see Tables 1, 2, 3 and 4) we have compared the time spent on the forward elimination phase of the problem. The backward substitution phase runs extremely fast, and is a very minimal factor in the time it takes to run this program [7]. Thread communication time has not been extracted from the times listed. When using a *parallel do* directive in OpenMP, a *do loop* must immediately follow the directive. Because of this restriction, a timer would have to be inserted within the *do loop* and would effect performance.

From the results, we can see that each load balancing scheme produces some speedup, depending on the amount of data we are working with. For smaller values of *n*, all three schemes work very well. (see Table 1) As *n* increases from 400 to 800, and then to 1200, we can see that there is a point at which the amount of performance gain decreases for each distinct load balancing scheme. (see Table 4) With the dynamic scheme, the decrease in performance is much slower than it is with the others.

Since the amount of work to be distributed is constantly changing throughout the algorithm, the dynamic scheme proves to work best because of its ability to distribute new iterations while other threads remain occupied.

3. MPI Parallel Version

The algorithm was also ported to a distributed environment where we use MPI to allocate the work across multiple processors. (see Figure 2)

```
root = 0
chunk = n**2/p
```

```
! main loop
do pivot = 1, n-1

    ! root maintains communication
    if (my_rank.eq.0) then

    ! adjust the chunk size
        if (MOD(pivot, p).eq.0) then
            chunk = chunk - n
        endif

        ! calculate chunk vectors
        rem = MOD((n**2-(n*pivot)),chunk)
        tmp = 0
        do i = 1, p
            tmp = tmp + chunk
            if (tmp.le.(n**2-(n*pivot))) then
                a_chnk_vec(i) = chunk
                b_chnk_vec(i) = chunk / n
            else
                a_chnk_vec(i) = rem
                b_chnk_vec(i) = rem / n
                rem = 0
            endif
        continue

        ! calculate displacement vectors
        a_disp_vec(1) = (pivot*n)
        b_disp_vec(1) = pivot
        do i = 2, p
            a_disp_vec(i) = a_disp_vec(i-1)
                          + a_chnk_vec(i-1)
            b_disp_vec(i) = b_disp_vec(i-1)
                          + b_chnk_vec(i-1)
        continue

        ! fetch the pivot equation
        do i = 1, n
            pivot_eqn(i) = a(n-(i-1),pivot)
        continue
        pivot_b = b(pivot)
    endif   ! my_rank.eq.0

! distribute the pivot equation
call MPI_BCAST(pivot_eqn, n,
                    MPI_DOUBLE_PRECISION,
                     root, MPI_COMM_WORLD, ierr)


call MPI_BCAST(pivot_b, 1,
                    MPI_DOUBLE_PRECISION,
                     root, MPI_COMM_WORLD, ierr)

! distribute the chunk vector
call MPI_SCATTER(a_chnk_vec, 1, MPI_INTEGER,
                        chunk, 1, MPI_INTEGER,
                       root, MPI_COMM_WORLD, ierr)

! distribute the data
call MPI_SCATTERV(a, a_chnk_vec, a_disp_vec,
                        MPI_DOUBLE_PRECISION,
                         local_a, chunk,
                        MPI_DOUBLE_PRECISION,
                         root, MPI_COMM_WORLD,ierr)

call MPI_SCATTERV(b, b_chnk_vec, b_disp_vec,
                        MPI_DOUBLE_PRECISION,
                         local_b, chunk/n,
                        MPI_DOUBLE_PRECISION,
                         root, MPI_COMM_WORLD,ierr)

! forward elimination
do j = 1, (chunk/n)
    xmult = local_a((n-(pivot-1)),j) / pivot_eqn(pivot)
    do i = (n-pivot), 1, -1
        local_a(i,j) = local_a(i,j)
                     - (xmult * pivot_eqn(n-(i-1)))
    continue
    local_b(j) = local_b(j) - (xmult * pivot_b)
continue

! restore the data to root
call MPI_GATHERV(local_a, chunk,
                       MPI_DOUBLE_PRECISION,
                        a, a_chnk_vec, a_disp_vec,
                       MPI_DOUBLE_PRECISION,
                        root, MPI_COMM_WORLD, ierr)

call MPI_GATHERV(local_b, chunk/n,
                       MPI_DOUBLE_PRECISION,
                         b, b_chnk_vec, b_disp_vec,
                       MPI_DOUBLE_PRECISION,
                        root, MPI_COMM_WORLD, ierr)

continue ! end of main loop
```

Figure 2.
MPI parallel version of the forward elimination algorithm

One of the major aspects of implementing the Gaussian Elimination algorithm on a distributed memory system is the communication time. This has a significant effect on the resulting performance of the algorithm but, with appropriate modification, we achieved modest speedups.

Using MPI, you almost have to rewrite your code in order to test different load balancing techniques. Through every iteration of the main outer loop, the amount of work changes. Therefore, in order to help balance the

Table 5.
MPI parallel results using 2 processors

.

n	Communication Time	Workload Time	Total Time
400	2.15	0.72	2.88
800	18.28	5.76	24.03
1200	70.98	19.68	90.66

Table 6.
MPI parallel results using 4 processors

n	Communication Time	Workload Time	Total Time
400	3.46	0.37	3.83
800	23.43	2.90	26.32
1200	82.73	9.85	92.58

Table 7.
MPI parallel Speedup results using varying values of n and p processors

N	2	4
400	none	none
800	1.59	1.46
1200	1.60	1.56

load, the chunk size is re-calculated and the data re-distributed among the available processors. Instead of re-writing the algorithm to distribute the data in a different manner, we chose to test the program several times using a different number of processors and varying values of *n*. The technique used for load balancing remains the same, but the chunk sizes to be distributed differ.

The algorithm proceeds by looping through the matrix **A** and vector **b,** setting each equation as the pivot equation. The pivot equation is then broadcast and the remaining *pivot +1* to *n* equations are distributed among the processors. Processor 0 handles the communication maintenance. Because the amount of data to be distributed changes as we progress through the matrix, a test is required to see if the chunk size needs to be adjusted. This will ensure that the amount of work being divided among processors remains somewhat even. The chunk size will decrease with every *p* iterations of the algorithm.

The chunk vectors used in the scatter and gather functions contain *p* elements, where each i^{th} element represents the chunk size of the data that is going to be passed to each *i-1* processor. The MPI_SCATTERV routine allows us to distribute varying amounts of data to each processor. The displacement vectors that will be used with MPI_SCATTERV and MPI_GATHERV contain *p* elements, where each i^{th} element represents the starting point (displacement) at which processor *i-1* will begin to get its *chunk* size of data from p0.

At the end of the communication maintenance, the data is distributed among the available processors. First, the MPI_BCAST library routine was used to pass the pivot equation to each thread [12]. The MPI_SCATTER routine distributes the chunk vector that was calculated from matrix **A**. The reason for scattering this data is because MPI_SCATTERV expects each thread to know the chunk size of data that they are going to receive. Without this information, extra work would have to be done to figure out how much data was received by each processor.

The matrix **A** was distributed so that each processor will get approximately the same amount of work in order to maximize performance. As we loop through the matrix, the amount of rows left to work with decreases. The SCATTERV routine allows you to specify the data you want to send to each processor, which reduces the amount of data transmitted for each row [12].

There was one disadvantage to using the SCATTERV library routine. Since Fortran stores data in column-major order the algorithm was modified to work with this new format. Before performing the next step, the matrix was reverted back to row-major order to take advantage of the original backward substitution algorithm.

Since the amount of time involved in performing the backward substitution phase of this algorithm is minimal as compared to the forward elimination phase this section of the algorithm was run sequentially [8]. Also, the dependence involved in the outer loop of the backward substitution phase limits the amount of parallelization possible.

In our results, (see Tables 5, 6 and 7) comparisons are made based on time spent in the forward elimination phase of the problem. With the MPI version, we were able to extract the time spent on communication and compare it to the time spent on calculations.

The speedups in Table 7 show that as the amount of work increases, there is a point at which the distributed system begins to demonstrate performance. The reason why no speedup was achieved with small *n* is because the amount of overhead involved in distributing the data was overwhelming as compared to the amount of work required. A simple solution to this problem would be to run the application sequentially if the size of *n* is too small.

4. Conclusions

The major disadvantage to using the distributed memory architecture for this application, is that most of the distribution work is data related, not task related. The only way the nodes can access the data is by passing it back and forth. In a shared memory model, this does not cause a problem because the data can be made available to all processors at all times.

One thing to note from the results is the impact on performance that occurs as we change the size of *n*. When we increase the value of *n*, the MPI program displays an improvement in performance as opposed to the OpenMP program where performance increase seems to diminish. It is possible that as *n* increases, we may find a point where the distributed environment will show a greater increase in performance than the shared platform.

5. References

[1] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, X. S. Li, (2000) *Analysis and comparison of two general sparse solvers for distributed memory computers*, ACM Transactions on Mathematical Software, (to appear).

[2] P. R. Amestoy, I. S. Duff, J.-Y. L'Excellent, X. S. Li, (2001) *Impact of the implementation of MPI point-to-point communications on the performance of two general sparse solvers*, Tech report LBNL-48978, Lawrence Berkeley National Laboratory, http://www.nersc.gov/~xiaoye/.

[3] E. Aubanel (2000), *ACRL configuration*, www.cs.unb.ca/acrl/acrl_configuration.html.

[4] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, R. Menon, (2001) Parallel Programming in OpenMP, Morgan Kauffmann Publishers Inc.

[5] W. Cheney, D. Kincaid, (1998) Numerical Mathematics and Computing, third edition, Brooks/Cole Publishing Company.

[6] J. W. Demmel, J. R. Gilbert, X. S. Li, (1997) *An asynchronous parallel supernodal algorithm for sparse Gaussian Elimination*, SIAM Journal on Matrix Analysis and Applications, 20(4):915-952.

[7] J. Howe & S. Bratcher, (1996) *Parallel Gaussian Elimination*, http://www.cse.ucsd.edu/classes/fa98/cse164b/Projects/PastProjects/LU/.

[8] P. S. Pacheco, (1997) Parallel Programming with MPI, Morgan Kaufmann Pub. Inc.

[9] C. Severance, R. Enbody, (1995) *A hybrid approach to load balancing on shared memory parallel processors*, http://www.netfact.com/crs/papers/load_94/.

[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, (1999) MPI – The Complete Reference, Volume 1, The MPI Core, second edition, The MIT Press