Conf-910263--4

UCRL- JC-105426
PREPRINT

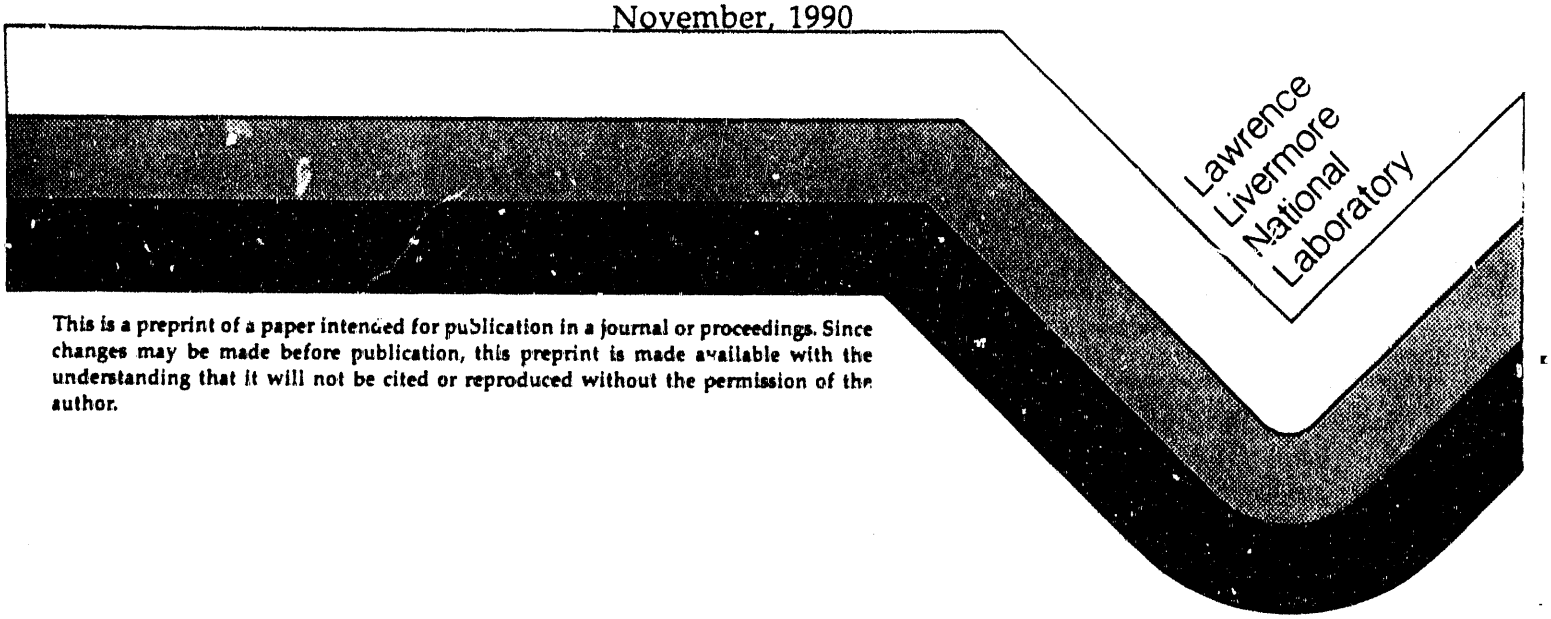
JAN 16 1991

GAUSS ELIMINATION: A CASE STUDY ON PARALLEL MACHINES

Karen H. Warren and Eugene D. Brooks III
Massively Parallel Computing Initiative
Lawrence Livermore National Laboratory
Livermore, CA 94550

This paper was prepared for submittal to
CompCon Spring '91, San Francisco, California
February 25-March 1, 1991

November, 1990



DISCLAIMER

Gauss Elimination: A Case Study on Parallel Machines*

Karen H. Warren and Eugene D. Brooks III

Massively Parallel Computing Initiative
Lawrence Livermore National Laboratory
Livermore, CA 94550

UCRL-JC--105426

DE91 006259

Abstract: We report our experiences with the Gauss elimination algorithm on several parallel machines. Several different software designs are demonstrated, ranging from a simple shared memory implementation to use of a message passing programming model. In this work we find that the efficient use of local memory is critical to obtaining good performance on scalable machines. Machines with large coherent caches appear to require the least software effort in order to obtain effective performance.

Keywords: Gauss elimination, shared memory, message passing, coherent caches, local memory

1 Introduction

Regardless of the portability of the parallel programming language used, developing parallel software applications that are fast and efficient is a machine dependent process. PCP [1], a portable Parallel C Preprocessor, was used for the examples discussed herein. PCP is an implementation of the split-join parallel programming model, similar to Harry Jordan's Force [2] and the IBM SPMD model [3], in which the user explicitly controls the execution of a large number of processors. PCP has been successfully ported to the Sequent Balance, Sequent Symmetry, Alliant FX/8, SGI, Stellar, Cray, BBN TC2000 and uniprocessor hardware.

As is demonstrated in this paper, the efficient software design of the Gauss elimination algorithm for the solution of dense linear systems is largely dependent upon exploiting features of the underlying target machine architecture. If the machine presents shared memory only, a straightforward parallel implementation obtains good performance. If the machine has high performance local memory in the form of coherent shared memory caches, we must design an algorithm which explicitly makes use of that aspect of the machine architecture, resulting in a doubling of the required coding effort. If the machine does not provide coherent caches, but possesses large local memories, we find that we must simulate in software the behavior of caches thereby greatly increasing the required lines of code to obtain good performance. This penalty is as high as a factor of five in line count over the original serial version. Finally, if the machine does not provide shared memory, we are forced to use explicit message passing calls having a high software overhead in order to communicate between processors. Performance suffers as a result.

The sections of this paper are as follows. We present the Gauss elimination algorithm in Section 2, describing the data dependencies and the possibilities for different designs depending on the types of memory and cacheing available. In Section 3, we present the simple, straightforward implementation of the algorithm designed for a shared memory machine. We next explain our use of coherent cache shared memory in Section 4 showing the results of its implementation and that of the earlier simpler version on the Sequent Symmetry. We then describe our explicit use of local memory in Section 5, and our use of a message passing model in Section 6, showing results for both using up to 48 processors on the BBN TC2000. We conclude with a discussion in Section 7.

2 Gauss Elimination

For the purposes of this paper, we consider Gauss elimination without pivoting. Including pivoting can be done and does not qualitatively change the results but instead complicates the code examples given and reduces clarity of the discussion.

The Gauss elimination algorithm is composed of

*Work performed under the auspices of the U. S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

two parts. The first is referred to as the reduction wherein the matrix is reduced to an upper triangular form which exposes the last element of the unknown vector to direct solution. Reduction is done in n steps where n is the number of equations. In the *i'th* step suitable multiples of the *i'th* row, known as the pivot row, are subtracted from the rows below in order to zero out matrix coefficients. All of the operations on the rows below can be done in parallel, but each operation depends upon the pivot row.

In the second part, the backsolve, which is started as soon as the matrix is brought to upper triangular form, elements, from n to *1*, of the unknown solution are successively solved for and substituted into the remaining equations. These substitutions can also be done in parallel, as soon as a processor has provided the next element of the solution vector.

3 Simple Parallel Version

Assuming a shared memory machine, the easiest and most obvious way to design the Gauss elimination algorithm is to use parallel do loops in the reduction part, letting the processors work in parallel on the rows in shared memory. Similarly, for the backsolve part the processors work in parallel on the rows in shared memory. A barrier is used at the end of each reduction step and backsolve step to ensure that all of the calculations for that step have been completed before proceeding on to the next step. Implementing this version in PCP added only seven lines of code to the original serial version. The PCP modifications are shown below in bold face.

```
void dgauss(double **a, double *b, int dim)
{
    /* reduction outer loop */
    for(int k = 0; k < dim; k += 1) {
        forall (int i = k+1; i < dim; i +=1) {
            double temp = a[i][k];
            if(temp == 0.0) continue;
            a[i][k] = 0.0;
            temp /= a[k][k];
            for(int j = k + 1; j < dim; j +=1) {
                    a[i][j] - = a[k][j] * temp;
            }
            b[i] - = b[k] * temp;
        }
        barrier;
    }
    /* backsolve outer loop */
    for(int i = dim - 1; i >= 0; i - = 1) {
        master {
            b[i] /= a[i][i];
        }
        barrier;
        forall (int k = i-1; k >= 0; k - = 1) {
            b[k] - = a[k][i] * b[i];
        }
    }
    barrier;
}
```

This version of the code does not constrain the processors to perform repeated subtractions on a given row and uses excessively strong barrier synchronization on each step of the reduction and backsolve parts. If there is sufficient shared memory bandwidth to support the floating point operations, and the problem is large enough to amortize the cost of the barriers, good parallel performance is obtained.

4 Exploiting Coherent Caches

To exploit the use of local caches for shared memory and to avoid the use of time consuming barriers, the code was altered to use a *spin-wait* flag in shared memory to signal when a given pivot row is ready. We exploit the separate caches for each processor by having each processor work on only specific rows. This is done by explicitly coding the PCP forall loops using the team state variables, `_tindex` and `_tsize`. The code is shown below:

```
void dgauss(double **a, double *b, int dim)
{
    /* Flags are initialized to zero: */
    static int flags[1024];
    master {
        flags[0] = 1;
    }
    /* reduction outer loop */
    for(int k = 0; k < dim; k += 1) {
        /* Wait for the pivot row to be stable: */
        while (flags[k] == 0);
        for (int i = k + 1 + (_tindex + _tsize -
                (k%_tsize))%_tsize;
                        i< dim; i+= _tsize) {
            double temp = a[i][k];
            if(temp == 0.0) continue;
            a[i][k] = 0.0;
            temp /= a[k][k];
            for( int j= k + 1; j< dim; j+= 1) {
                a[i][j] - = a[k][j] * temp;
```

```
            }
            b[i] - = b[k] * temp;
            if (i == k + 1) flags[i] = 1;
        }
    }
    barrier;
    /* Now we perform dim back substitutions.
        Note that the meaning of flag == 0
        now means that the data is ready
        whereas before it meant not ready.
    First solve for the last x: */
    master {
        b[dim-1] /= a[dim-1][dim-1];
        /* Indicate x[dim-1] is solved. */
        flags[dim-1] = 0;
    }
    /* backsolve outer loop */
    for(int i = dim - 1; i >= 1; i - = 1) {
        if (_tindex == ((i-1) % _tsize)) {
            while (flags[i] == 1]);
            b[i-1] - = a[i-1][i] * b[i];
            b[i-1] / = a[i-1][i-1];
            /* Indicate x[i-1] is solved. */
            flags[i-1] = 0;
        }
        else {
            /* Wait for x[i] */
            while (flags[i] == 1);
        }
        for (int k= _tindex; k< i-1; k+= _tsize) {
            b[k] - = a[k][i] * b[i];
        }
    }
    barrier;
}
```

This version of the code produced impressive results on the Sequent Symmetry, a bus based coherent cache machine which provides twice bus limited performance on problems which are small enough to be distributed in the caches without spilling. In Figure 1, a plot of the millions of floating point operations per second, MFLOPS, versus the number of equations, we show the speed of the two codes using 30 processors on a Sequent Symmetry machine having copy-back caches. The points labeled **std** are the results of the simple version described in Section 2 and the points labeled **cust** denotes the version, discussed in this section, which exploits the coherent shared memory caches. The performance which both codes asymptotically approach for large problem sizes, a little greater than 6 MFLOPS, is limited by the bus bandwidth in the system. The problem size for which the coherent cache version of the code reaches a peak and starts to roll off can be calculated from the size of the caches, 64 K bytes. The performance of these two versions of the code on the BBN TC2000, however, was disappointing. The BBN TC2000, a scalable microprocessor based machine, has processors which possess local cached memory and access via a scalable switch to interleaved shared memory. The use of 10 processors was required to reclaim the original serial program performance of one computational node. This arises from the lack of shared memory coherent cache support on the TC2000. On this machine we must exploit local memories explicitly to reduce the adverse impact of slow memory accesses through the switch connecting the processors.

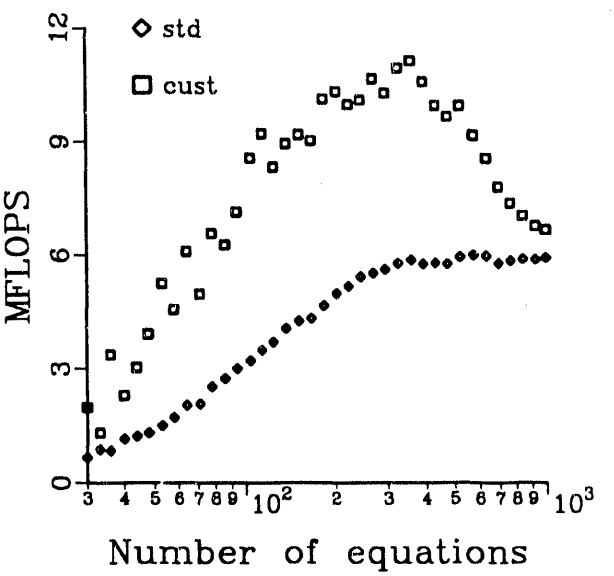


Figure 1: Performance of the two Gauss elimination algorithms as a function of the number of equations, on a 30 processor Sequent Symmetry system equipped with Weitek floating point hardware.

5 Explicit Local Memory Use

To exploit local memory on the BBN TC2000, the rows of the matrix which resides in shared memory are divided up and pulled in to local memory by the processors which will then perform the reduction and backsolve operations on those specific rows. Shared memory is now used only to communicate intermediate results such as pivot rows and newly solved ele-

ments of the solution vector between processors. On a scalable machine lacking coherent caches, read only hot spots, .i.e., locations for which many processors are contending, become a serious problem, and such data needs also to reside in local memory. Implementing this version required 106 lines of code, approximately 5 times more than the original serial version.

The results of this effort using 48 processors on the BBN TC2000 are shown in Table 1. n denotes the number of equations in the linear system, *MFLOPS* is the overall speed of the code measured in millions of floating point operations per second, t_{sec} is the total execution time in seconds, t_r the reduction time, t_b the backsolve time and S the speedup. Speedups are relative to the original serial version of the code running in local memory.

n	MFLOPS	t_{sec}	t_r	t_b	S
100	3.40	0.20	0.16	0.04	2.6
200	10.47	0.52	0.46	0.06	7.7
300	18.85	0.96	0.88	0.08	13.7
400	27.65	1.55	1.44	0.11	19.8
500	31.34	2.68	2.48	0.20	22.2
600	38.32	3.78	3.61	0.17	28.0
700	42.25	5.43	5.25	0.18	33.0
800	44.75	7.66	7.46	0.19	35.0
900	46.96	10.38	10.16	0.23	36.7
1000	48.39	13.82	13.56	0.26	38.1

Table 1. Performance using local memory explicitly.

6 Message Passing

To obtain a fair comparison of the version of the code which uses shared memory for communication with one which uses the a message passing programming model we converted our example to the Livermore Message Passing System, LMPS, available on the BBN TC2000. The message overheads of this package are similar to those found on any of the available message passing machines such as those offered by INTEL or NCUBE.

This version of the code required 160 lines, including those lines required to move the matrix from shared memory to local memory. In general, a similar overhead is incurred on a message passing machine as data is rearranged in configurations suitable for each step of a complex application. In this version of LMPS, direct support for broadcast of the pivot row to waiting processors does not exist and a given processor must send the pivot row to other processors in a loop iteration.

The results for this version of the code using up to 48 processors on the BBN TC2000 are tabulated in Table 2. Again, n denotes the number of equations in the linear system, *MFLOPS* is the overall speed in millions of floating point operations per second, t_{sec}, the total time in seconds, t_r is the reduction time, t_b is the backsolve time and S is the speedup. Due to the high message overhead we achieved a speedup of only 12.37 using 48 processors for a 1000 × 1000 system. This version of the code is clearly inferior to the previous version which uses the high bandwidth and low latency shared memory to move data between processors. In particular, note the disaster which occurs in the backsolve.

Performance can be improved by using a tree broadcast to communicate pivot rows, but this will still be slower than using the interleaved shared memory facility by a factor of log N where N is the number of processors being used in the application.

n	MFLOPS	t_{sec}	t_r	t_b	S
100	0.32	2.18	1.25	0.93	0.25
200	1.21	4.48	3.01	1.46	0.89
300	2.00	9.10	6.91	2.19	1.45
400	3.73	11.53	8.70	2.84	2.66
500	5.71	14.69	11.03	3.66	4.05
600	6.46	22.39	18.04	4.36	4.72
700	8.32	27.61	22.47	5.13	6.50
800	11.01	31.12	25.26	5.87	8.60
900	13.34	36.57	30.05	6.52	10.42
1000	15.71	42.57	35.36	7.21	12.37

Table 2. Performance using explicit message passing.

7 Discussion

We have presented several different software designs for the Gauss elimination algorithm demonstrating the simple use of shared memory, the exploitation of coherent caches for shared memory, the explicit use of local memory, and the use of message passing. These implementations were a result of considering the architecture of the target machine. As one attempts to exploit hardware that provides less service the software costs climb rapidly. It is evident that software costs are the key issue when considering the use of multiprocessors.

As we scale up the number of processors in a shared memory design, hot spots become a significant problem which must be carefully sought out and avoided. It is clear that effective use of local memory, whether in the form of caches or explicit local memories will be a critical issue in exploiting scalable machines in the future.

The actual floating point performances obtained using 30 processors on the Sequent Symmetry (about 11 MFLOPS), and using 48 processors on the BBN TC2000 (about 50 MFLOPS), are not too impressive when compared to the approximately 300 MFLOP performance which a single processor of the Cray YMP can achieve on a linear system solve. What we have demonstrated is how various multiprocessor designs can be exploited for the solution of a linear system and what the software costs of efficiently utilizing different hardware designs are. It is clear that good speedups can be obtained for a linear system solver given appropriate multiprocessor architectures and due attention to coding strategies. Scalable machines utilizing the 200 MFLOP floating point units soon to appear on the market will completely eclipse current day supercomputers, providing that the communication capability between nodes is appropriately increased along with the floating point speed.

References

[1] Eugene D. Brooks III, *PCP: A Parallel Extension of C that is 99% Fat Free*, UCRL-99673, Lawrence Livermore National Laboratory, 1988.

[2] Harry F. Jordan, *The Force: A Highly Portable Parallel Programming Language*, Proceeding of the International Conference on Parallel Processing, August, 1989.

[3] F. Darema, D. A. George, V. A. Norton and G. F. Pfister, *A single-program-multiple data computational model for EPEX/FORTRAN*, Parallel Computing, April, 1988.

-END-

DATE FILMED

02 / 13 / 91