CS458: Project 3

Shule Li

---

1. Problem Decription

In this project we revisit parallel Gauss elimination method to triangularize a given matrix and solve the related linear system. The input matrix can be a given or random matrix. The parallelization method considered here are MPI, OpenMP and Pthreads. We run the algorithm for two distinct input matrix dimensions: 512 and 2048, on two different platforms: Cycle1, a x86 Linux machine with 8 processors, and niagara1, a SunBox with 32 hardware threads. We verify the correctness of each parallelization method, and compare the performance of different methods and platforms.

2. Build and Run Instruction

The source code is *gauss_mpi.c* for mpi, *gauss_omp.c* for OpenMP. To compile on the x86 machine, simply link $Makefile\_mpi.x86$ or $Makefile\_omp.x86$ to $Makefile$. To compile on the SunBox, use $Makefile\_mpi.sunos$ or $Makefile\_omp.sunos$ instead. The MPI implementation uses MPI 3.0.2 for x86, MPI 1.2.7 for SunBox. The OpenMP implementation uses GCC 4.1.2 for x86, GCC 4.2.0 for SunBox. To run the program with MPI, execute:

```
mpirun -np P ./run -nX
```

where P is the number of processors, X is the matrix dimension (default to 512). To run the program with OpenMP, execute:

```
./run -pP -nX
```

where P and X carry the same meaning as before. The program outputs computation time as well as the verification result. To turn on randomized matrix, uncomment the commented lines in the initMatrix() function. To see the output solution vector, uncomment the commented line in the verification section in main().

3. Algorithm

The algorithm of Gauss elimination is as follows. For a given square matrix A, start the iteration with the first row (we index the iteration by i). The row of consideration of each each iteration is called the pivot row. For a given

pivot row i at iteration i, we must ensure that the diagonal element ($A[i][i]$) is nonzero. If not, we need to swap it with row j below row i that has nonzero element $A[j][i]$. When doing the swap, we also swap the corresponding vector element. After this is done, we do a linear elimination of the column i for all the subsequent rows below row i with respect to the pivot row, including the right hand side (hereon RHS) vector. Carry out this iteration till the final row, the resulting matrix will be upper-triangular. To solve the linear system, we simply backtrack from the last row to the first row. This algorithm is $O(n^3)$, and can be easily parallelized. We now describe the three parallel strategies used in this project.

MPI: In the message passing case, we first initialize the matrix A, RHS vector B on the master processor. We then assign pieces of A and B to each workers. Notice that the master already holds the entire matrix, so no assignment is needed. Each worker will get a matrix of size $n/p \times n$ (lines 309 through 354). The assignment is done in a cyclic fashion (row 0 assigned to processor 0, row 1 assigned to processor 1, etc...) so that even when we are near the end of the iterations, we can still get decent parallelism. The actual message passing is done from line 360 through line 375. Because we want to send a section of the matrix as long as a section of the RHS vector and the swap information, we create a message block (*messageblock in the code) to hold all the data a particular worker needs for its calculation. We use the following functions to pack/unpack data into/from the messageblock:

```
PackMessage()
UnpackMessage()
```

whose details are defined at lines 170 and 197. After each worker gets its data, we start the iteraction (line 386). The first thing to do is to determine the pivot row, which is the same as the iteration number. Since the pivot row can be held by any processor, we need to first find out which processor has the up to date data of the pivot row. Notice that we cannot let the master to determine it since rows stored on master will be outdated when the update starts unless the master is its owner (responsible to update that row). We find the owner of the pivot row by the following line:

```
pivotloc = row%nproc
```

where row is the pivot row index, nproc is the number of processors, pivotloc is the MPI id of the pivot owner. The pivot owner then try to ensure the pivot has a nonzero diagonal element (lines 390 through 396) by calling function getPivot(). If the diagonal value is zero, the function will look at the locally stored matrix and attemp to swap the pivot row with a row under it. This attemp may fail as the local matrix is not the entire thing. When it fails, it sets the *exchange* flag to 1, signaling that the pivot row needs to be swapped with a row that is owned by another processor. If this is the case, each other processors then look at its locally stored matrix and prepare a potential swapping row by

calling function $FindExchange()$, which returns a local index (*eline* of lines 402 to 406) of the prepared row for a potential swapping. These indices are gathered by the pivot owner using MPI_Gather to determine an exchange partner (lines 412 to 426). Two buffers (named *inbuffer and *outbuffer) are used for the actual exchange. The matrix, RHS vector and swap vector information are packed and unpacked using functions $PackLine()$ and $UnpackLine()$. We revert the order of MPI_Send and MPI_Recv for the pivot owner and the exchanger so that deadlock is avoided. Once the swapping is done, the owner of the pivot row broadcast it to all processors for use. The rest of the calculation is very similar to the sequential version, where each processor work on those rows they own, update the rows as well as the RHS vector elements under the pivot row. On finishing, we send the matrix and vector pieces back to the master for reassembling (lines 506 through 521). Here again we use $PackMessage()$ and $UnpackMessage$ to minimize the number of messages we have to send/receive. The verification is done only on master, and is identical to the sequential version. There is another method where we can dynamically assign the submatrix for each interation to all the workers. The workers carry out the computation, and send back the pieces to the master. Since communication between the master and each worker has to be done per iteration, there is a huge communication overhead $2p \times n$ involved where in the previous method we described each iteration only has one broadcast of the pivot row. To use this implementation, use the source file *gauss_mpi_dynamic.c* instead. (verified to be correct but very slow)

OpenMP: The OpenMP version of the algorithm is quite simple. The only thing needed modification is in the actual calculation, where we add directives for parallel computation. We first get the size of matrix chunk for each worker as:

```
my_size = nsize/nproc
```

We start the parallel section using (line 216):

```
#pragma omp parallel shared(matrix, B, row, nsize, my_size, pivotrow)
private(my_id, pivotVal)
```

where matrix, RHS vector(B), the row index of the pivot row, the size of the matrix, the size of each chunk and the data stored on the pivot row are declared as shared. Each processor has its own version of id and the pivot value (matrix elements that has the same column number as the pivot row number). In the parallel section, we define local indices ii and jj for iteration, and start the parallel for loop using the following directive:

```
#pragma omp for schedule(dynamic, my_size) nowait
```

Since each processor is on its own for a certain iteration, we use dynamic nowait scheduling. The processes are joined by the closure of the parallel section. The solving and verification section is identical to the sequential version.

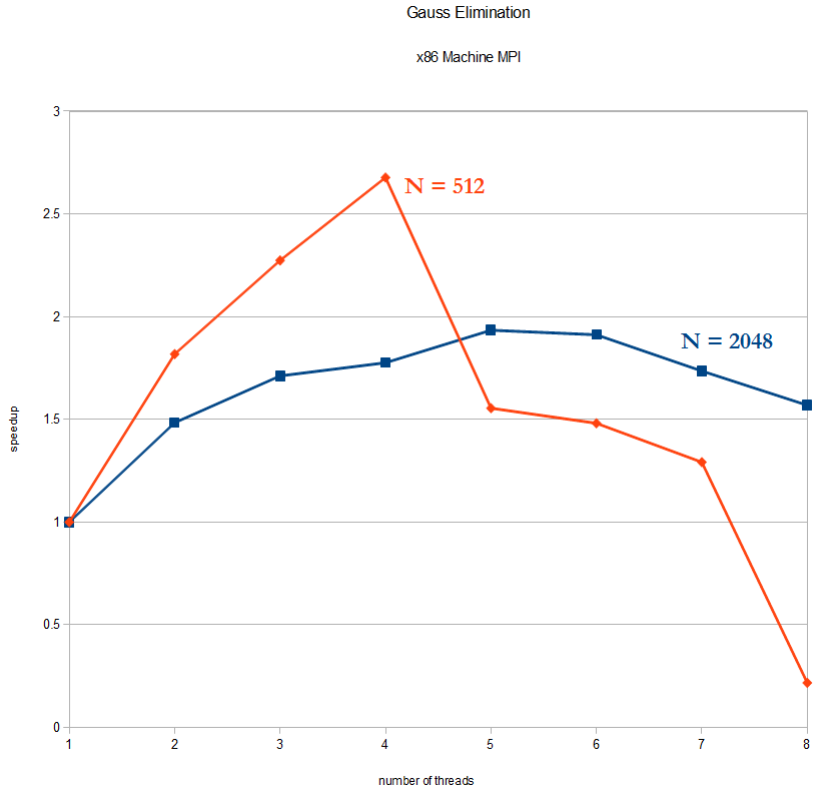

Figure 1:

Pthread: The pthread implementation is done in Project 1, where we dynamically assign the submatrix below the pivot row to the worker threads. Since we are using a shared memory model, workers do not have to send back the updated matrix piece for the next iteration. In the MPI version of dynamic assignment, huge overhead is induced by the communication between the master and workers to update the matrix on each iteration.

4. Performance Analysis

Fig.1, 2, 3 show the speed up of Gauss MPI, OpenMP and Pthreads on the x86 machine. The ratio of actual computation to communication depends positively on n but negatively on p. For the $p = 512$ case, if the number of processors is greater than 4, we see a sharp decline in performance, indicating the system bottleneck for communication of more than 4 processors. This behavior is also seen in the Pthread implementatoin (Fig.3, $N = 2048$ curve). Interestingly, the $N = 512$ performance of the MPI implementation on the x86 machine

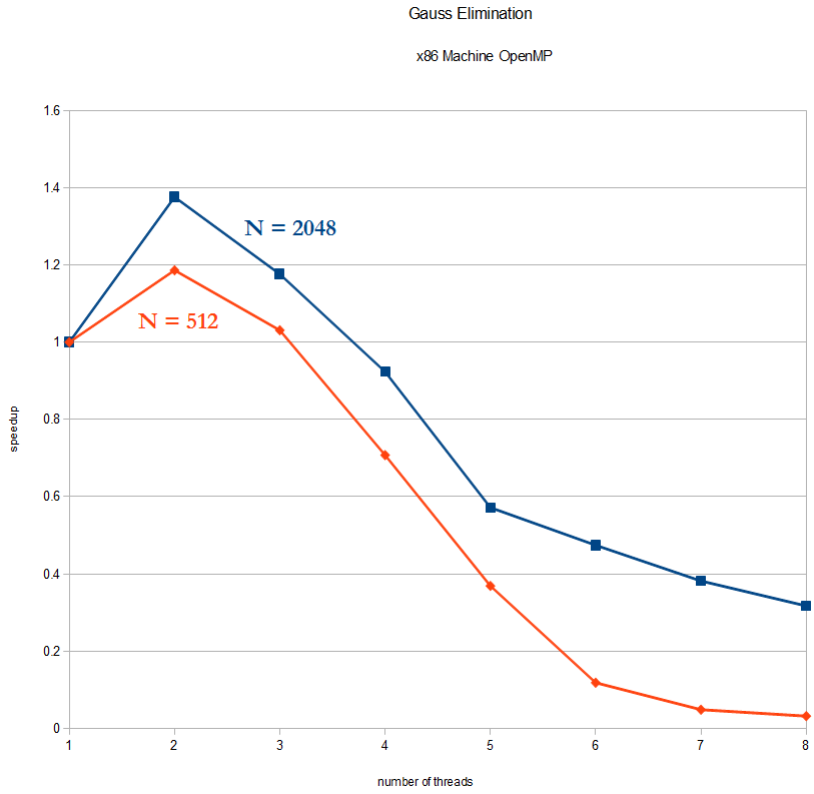
Gauss Elimination
x86 Machine OpenMP
N = 2048
N = 512
speedup
number of threads
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8

Figure 2:

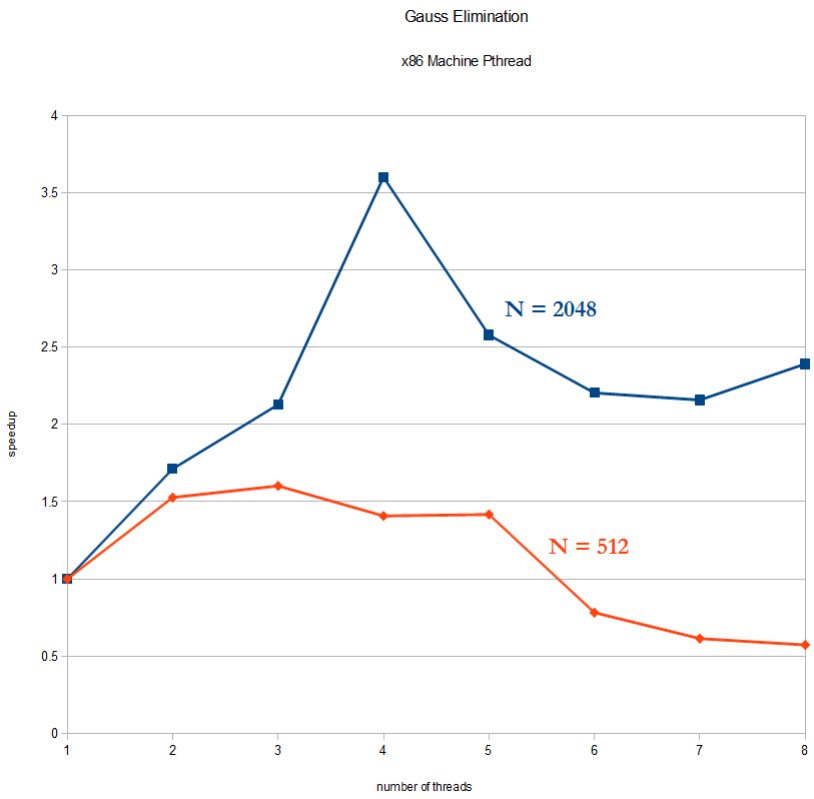
Gauss Elimination
x86 Machine Pthread
speedup
number of threads
N = 2048
N = 512
0
0.5
1
1.5
2
2.5
3
3.5
4
1
2
3
4
5
6
7
8

Figure 3:

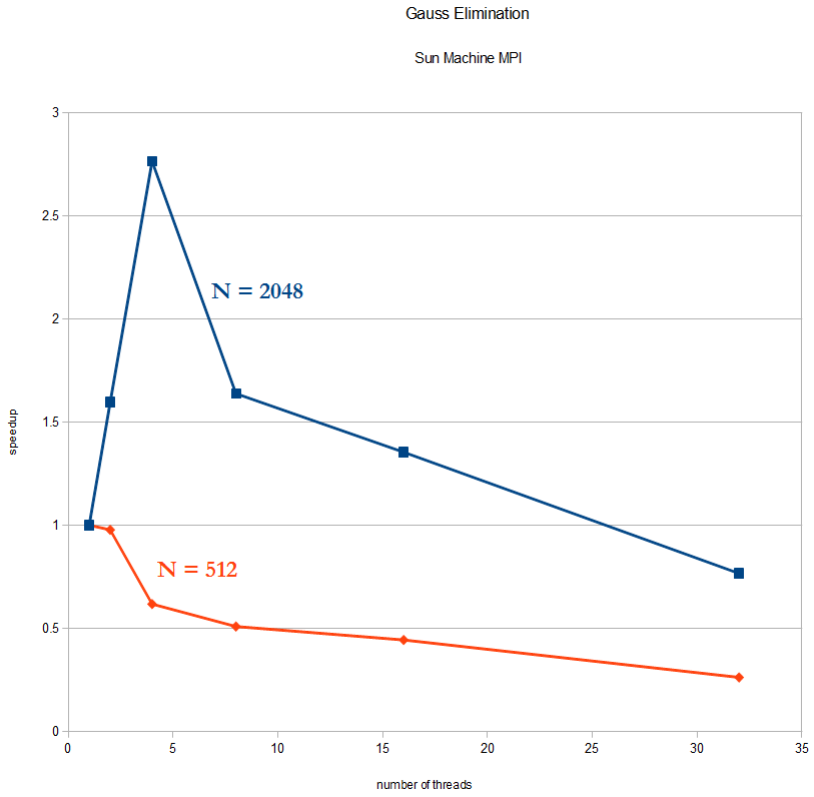

Figure 4:

is better than that of $N = 2048$. This is not the case for the shared memory implementations. In message passing, the communication overhead depends on the size of the data structure being sent/received. Therefore although there will be more computations per communication with a greater n, the communication also takes longer. While in shared memory there is no this kind of countering effect since data structures are not being sent around. The result of the said tradeoff is determined by the implementation detail as well as the underlying architecture. For the OpenMP implementation, we see that the parallel performance is poor for $p > 2$. Although OpenMP offers , it requires fine tuning on each particular architecture for optimal performance. In our case, it is probably better to use a MPI-OpenMP hybrid to simplfy the code as well as getting good performance on the x86 machines. The performance of the Pthread implementation is in agreement with the MPI implementation, except that the $N = 2048$ case performs better because of the lack of communication in this model.

Fig. 4, 5, 6 show the speed up of Gauss MPI, OpenMP and Pthreads on the SunBox. Here we have 8 cores 32 hardware threads. For the MPI implemen-

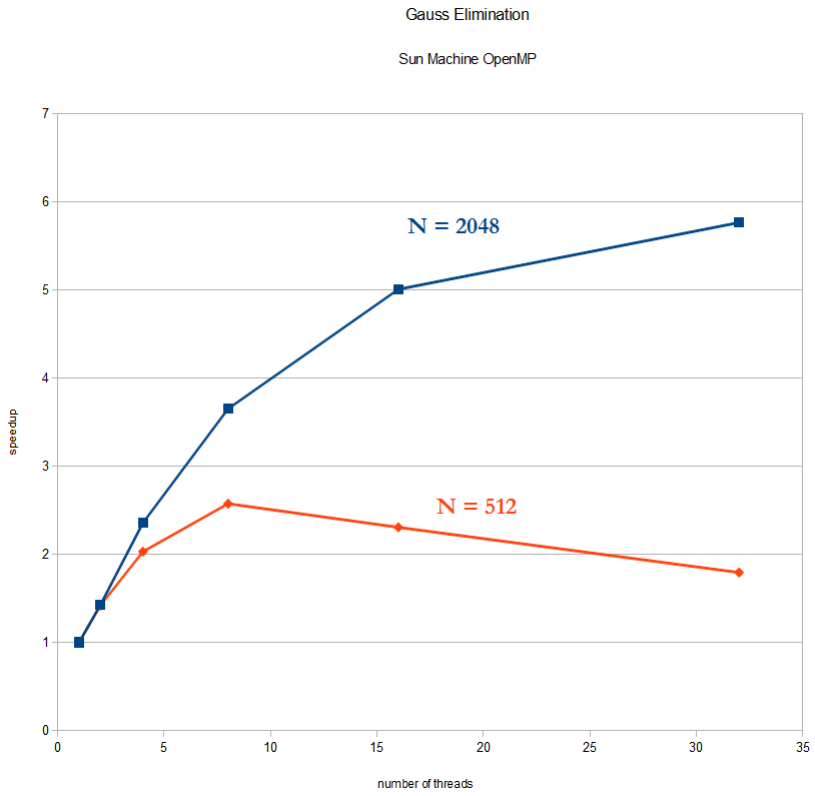
Gauss Elimination
Sun Machine OpenMP
N = 2048
N = 512
speedup
number of threads
0
1
2
3
4
5
6
7
0
5
10
15
20
25
30
35

Figure 5:

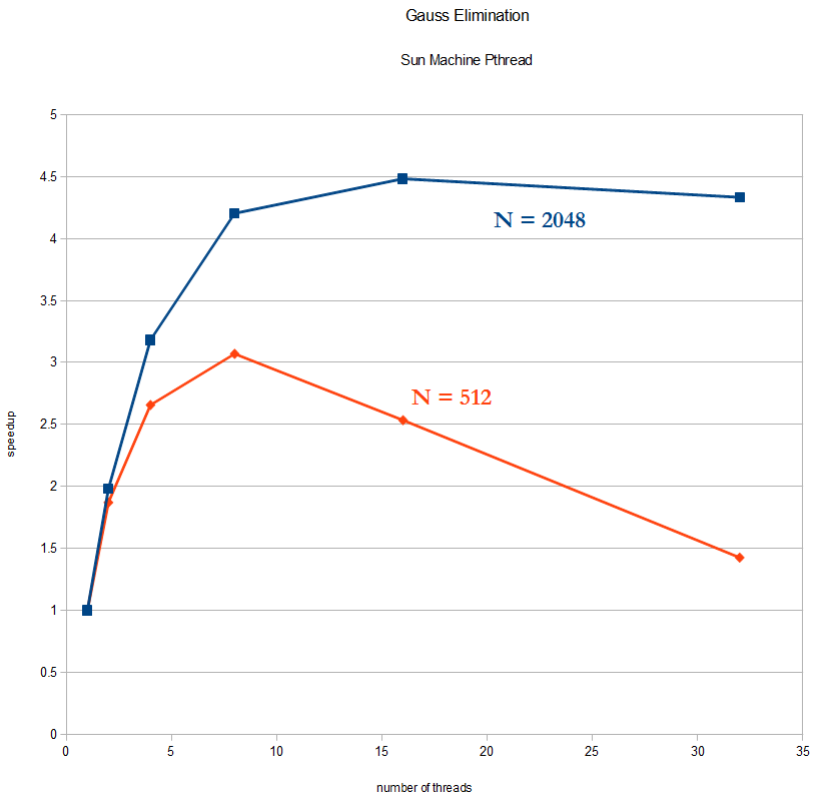
Gauss Elimination
Sun Machine Pthread
N = 2048
N = 512
speedup
number of threads
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
5
0
5
10
15
20
25
30
35

Figure 6:

tation, we see a similar behavior as on the x86 machine, except that now the $N = 2048$ case performs much better than the $N = 512$ case. This shows a dependence of the matrix dimension tradeoff on the architecture of the communication network: in this case the network scales well with the size of the message. The decline of performance on more than 8 threads can be either a result of poor network scalability on the number of nodes, or implementation inherent. For the OpenMP implementation, we see much more consistent performance compared to that on x86 machines. The SunBox performs better in shared memory because of its 4 threads per core architecture as indicated by the Pthread results (Fig.6) for large matrix dimension. The $N = 512$ case does not perform well in either OpenMP or Pthread, because with the same amount of computation but a increased number of processors, not only the ratio overhead from invalidation/update becomes higher, the chance of false sharing can also be much greater.