FACULTY OF COMPUTING
& INFORMATION TECHNOLOGY
KING ABDULAZIZ UNIVERSITY

كلية الحاسبات
وتقنية المعلومات
جامعة الملك عبدالعزيز

CPCS 414 High Performance Computing
Course Project

Gaussian Elimination Parallelism using OpenMP and MPI

Name: Amal Meer

Date: 21/04/2021

Contents

Changes on the Code Structure 3
Description of the used MPI collective communication calls 3
Description of the used OpenMP directives 5
The used test environment 5

Changes on the Code Structure

The code will be changed by separating the normalization from the elimination. Currently, the normalization is applied when computing the next rows using the ratio, but the row itself is not updated, meaning that the resulted upper triangular matrix has leading coefficients not necessary 1, which results in the necessity of dividing by the leading coefficient in each step in the back substitution. Although separating the normalization will add an extra for loop that has $\sum_{i=0}^{n-1}\sum_{j=i+1}^{n} 1 = \frac{1}{2}n(n+1)$ divisions, it will reduce ratio computation which has $\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} 1 = \frac{1}{2}n(n-1)$ divisions, and reduce the division on the back substitution which are n division, so in total $\frac{1}{2}n(n-1) + n = \frac{1}{2}n(n+1)$ divisions will be avoided, which is the same as the additional overhead. This indicates that this separation will not increase the computations. However, it will help on parallelizing the code by broadcasting the row values after normalization to the other rows. The code after update.

```
/* Applying Gauss Elimination */
for (i = 0; i < n ; i++) {                    // error and exit if a[i][i] zero
      for (j = i+1; j < n+1; j++) {
            a[i][j] = a[i][j] / a[i][i]; // normalize row i for each column j
      }
      a[i][i] = 1;                           // trivial case, no need to compute
      for (j = i + 1; j < n; j++) {
            for (k = i+1; k < n + 1; k++) {//start from i+1, <i are 0, i trivial
                  a[j][k] = a[j][k] - a[j][i]*a[i][k];
            }
            a[j][i] = 0;                    // trivial case, no need to compute
      }
}

/* Obtaining Solution by Back Subsitution */
x[n - 1] = a[n - 1][n];
for (i = n - 2; i >= 0; i--) {
      x[i] = a[i][n];
      for (j = i + 1; j < n; j++)
            x[i] = x[i] - a[i][j] * x[j];
}
```

Description of the used MPI collective communication calls

1- **Partition:** Gauss Elimination will create an upper triangular matrix by doing normalization and multiple elimination for each row. For that, each processor can process one row (domain decomposition). Back Substitution computes the target values using the resulted upper triangular matrix. Since it uses the resulted matrix, it needs to be done in one process, where this process will scatter the rows between processes, gather the resulted rows into one matrix, and do back substitution on this one matrix (Functional decomposition).

2- **Communication:** In Gauss elimination, each row will be eliminated i times (i = row index) using the upper i rows after normalization. For that, Gauss will work by normalizing the row, broadcasting this normalized loop to the succeeding rows to be eliminated, and doing the elimination after receiving the normalized loop. This will divide Gauss into 2 blocks, receiver block, which receives the broadcasted message and do the elimination, and sender block, which normalize the loop and broadcast the normalized loop to the next rows.

3- **Agglomerate:** To decrease the communication and task creation overhead, less processors will be used and the rows will be divided between them. That is, each processor will take $\lceil n/p \rceil$ rows (n = # rows, p = #processors). This will change the sender block as it will also do the elimination for the next rows in the processor after doing the normalization.

4- **Map:** Create one task per processor. Each task agglomerates $\lceil n/p \rceil$ rows.

Below is the pseudocode, followed by Table1 which describes the MPI collective communication calls. Point-to-point communication is not used as the normalized row is needed by all the succeeding rows. Using point-to-point to send to the succeeding rows only will complicates communication management without improving the performance.

Pseudocode

```
if master processor then                                       // Scatter
       set num_row = ceil(n/num_processors)
       send matrix of $num_row rows to each processor
else
       receive matrix of $num_row rows from the master

/* Receiver block */
for each row above the starting row of the matrix assigned to this processor
       Wait to receive the row after normalization from other processors
       Eliminate all the successive rows of this matrix using the received normalized row

/* Sender block */                                             // Broadcast
For each row in the matrix assigned to this processor
       Normalize the loop by dividing on the leading coefficient
       Broadcast the normalized loop to all other processors
       Eliminate all the successive rows of this matrix using the current normalized row

Synchronization point                                          // Barrier
If master processor then                                       // Gather
       Gather the matrices from each processor into one matrix
       Do back substitution
```

Table 1: The Used MPI Collective Communication Calls

MPI Com. Call	Usage justification
`MPI_Scatter`	Partition the matrix equally among processors by sending part of the rows
`MPI_Bcast`	Send the normalized row to all other processors to be used in elimination
`MPI_Barrier`	Synchronize all the processor before gathering in order to get the right values of the upper triangular matrix when gathering matrices.
`MPI_Gather`	Gather the matrices from all processes into one matrix in the master to form the upper triangular matrix, to be used in back substitution.

Description of the used OpenMP directives

For the master work (scatter and gather), there is no need for threads. For sender and receiver block, the outer loop can not be parallelized, but the inner loops (normalization and elimination for sender, and elimination for receiver) can be parallelized. For that, we will use `#pragma omp parallel for` three times, which will create k threads, where k is the default number of cores in the local device CPU, and it is the specified value of `ompthreads` in the PBS script when submitting the job on Aziz.

The used test environment

Aziz Username: jalahwal

Name of the queue: thin

Compiler used: impi/5.0.3.048

Input size: 800

Resources: 1 node, 24 CPU, 32 MPI processes, 16 OpenMP threads

Compile commands (run it in Aziz terminal):

```
module load impi/5.0.3.048

mpicc -fopenmp -o Gaussian GaussianElimination.cpp -lm -lstdc++
```

The job script **job.pbs** (execute it in Aziz terminal using the command `qsub job.pbs`)

```
#!/bin/bash                         // Tells PBS that this is bash script
#PBS -N Gaussian                    // Defines the job’s name. Does not affect execution

//Resource requirements. Here it’s 1 node, 24 cpu/node,32 MPI processes,16 OpenMP threads
#PBS -l select=1:ncpus=24:mpiprocs=32:ompthreads=16

#PBS -e error.txt              // Name of the error file
#PBS -o run3_output.txt        // Name of the output file
#PBS -l walltime=00:20:00      //Estimated runtime. Help schedular on allocating resources
#PBS -q thin                   // Submit the job to the thin queue in Aziz
cd $PBS_O_WORKDIR              // Move to the directory from where the script is submitted
module load impi/5.0.3.048     // Load the required modules

// save the list of the compute nodes
cat $PBS_NODEFILE > $PBS_O_WORKDIR/${PBS_JOBNAME}.${PBS_JOBID}.hosts

// count the number of mpi processes that will start based on the mpiprocs value.
NP=`cat $PBS_NODEFILE | wc -l`

echo Job started: `date`            // print start date and time
mpirun -np $NP ./Gaussian           // Run the executable file that was compiled in Aziz
echo Job finished: `date`           // print end date and time
```

To calculate speedup, the program run 3 times, the sequential and parallel code runtime is reported, then the average is used to calculate speedup.

Best speedup obtained = $\frac{ts_1+ts_2+ts_3}{tp_1+tp_2+tp_3} = \frac{average\ sequential\ time}{average\ parallel\ time} = \frac{(1.11+1.11+1.11)/3}{(0.05+0.05+0.04)/3} = \frac{3.33}{0.14} = 23.79$