CS546 HOMEWORK2

SERGIO PENAVADES SUAREZ
CWID: A20387929

1. Introduction

The purpose of this document is to show the benefits of using parallel computation instead of serial computation.

Gauss algorithm is used to solve mathematical equation problems with several unknown variables.

The Gauss formula is the following:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Where A represents the coefficients matrix, X de unknown variables vector, and B is the constants vector.

The procedure of the algorithm is to normalize row by row the A matrix to obtain the final solution as fast as possible. In our case, it will calculate the same number of operations for the same matrix dimension N*N. As a result, thanks to this property, the algorithm can be parallelized.

In addition, due to the vast amount of mathematical operations needed to solve big matrixes, Gauss algorithm is a great benefit as it can compute multiple operations at the same time which are independent from the others.

2. Analysis using pthreads

In this first section I will analyze parallel computation using pthreads. Also, I identified two ways of paralleling the Gauss algorithm.

On the one hand, I will use one thread to compute one row of the matrix A. So, if I the matrix has 20 rows, I will use 20 threads.

On the other hand, I will use one thread to compute n rows of the matrix A. So, if I the matrix has 20 rows, I will use 20/n threads (rounding 20/n).

2.1. Code

Parallelize using one thread to compute one row of A

In the piece code below, I show the functions that calculate the Gauss algorithm using one thread per row. The way in which I programmed the code is:

- First of all, the Gauss function is executed and one thread is assigned to compute each row of the matrix.
- Each thread executes the method addingThread, which obtains the value of the vector B assigned to each row.
- Lastly, the program returns to the function gauss to obtain the final substitution.

```
void *addingThread ( void *arg){
 float multiplier;
 int col;
 struct normAndRow *estructureAux = arg;
 int j;
  for (j=0; j <1; j++){
    multiplier = A[estructureAux->row+j][estructureAux->norm] / A[estructureAux->norm][estructureAux->norm];
    for (col = estructureAux->norm; col < N; col++) {
      A[estructureAux->row+j][col] -= A[estructureAux->norm][col] * multiplier;
    }
    B[estructureAux->row+j] -= B[estructureAux->norm] * multiplier;
  }
  return 0;
}

void gauss() {
  int norm, row, col;  /* Normalization row, and zeroing
                        * element row and col */
  pthread_t thread_CS546;
  struct normAndRow *threadStructure;

/* Gaussian elimination */
  for (norm = 0; norm < N - 1; norm++) {
    for (row = norm + 1; row < N; row++) {
      threadStructure = malloc(sizeof(struct normAndRow));
      threadStructure->norm = norm;
      threadStructure->row = row;
      pthread_create(&thread_CS546, NULL, addingThread, (void*)threadStructure);
    }
    pthread_join(thread_CS546, NULL);
  }

  /* Back substitution */
  for (row = N - 1; row >= 0; row--) {
    X[row] = B[row];
    for (col = N-1; col > row; col--) {
      X[row] -= A[row][col] * X[col];
    }
    X[row] /= A[row][row];
  }
}
```

Parallelize using one thread to compute n rows of A

In the piece code below, I show the functions that calculate the Gauss algorithm using n threads per row. The way in which I programmed the code is:

- First of all, the Gauss function is executed, a number (nRowsPerThread) is assigned that determines the number of threads that will be executed in parallel.
- Each thread is assigned the number of rows and executed the function addingThread, which calculates the value of B for its range of rows.
- Lastly, the program returns to the function gauss to obtain the final substitution.

```
void *addingThread ( void *arg){
 float multiplier;
 int col;
 struct normAndRow *estructureAux = arg;
 int j;
  for (j=0; j <nRowsPerThread; j++){
    multiplier = A[estructureAux->row+j][estructureAux->norm] / A[estructureAux->norm][estructureAux->norm];
    for (col = estructureAux->norm; col < N; col++) {
      A[estructureAux->row+j][col] -= A[estructureAux->norm][col] * multiplier;
    }
    B[estructureAux->row+j] -= B[estructureAux->norm] * multiplier;
  }
  return 0;
}

 void gauss() {
   int norm, row, col;  /* Normalization row, and zeroing
                         * element row and col */
   pthread_t thread_CS546;
   struct normAndRow *threadStructure;

   nRowsPerThread = (N-1)/2;
   if (nRowsPerThread <1 ){
     nRowsPerThread=1;
   }
   printf("%d\n",nRowsPerThread );
 /* Gaussian elimination */
   for (norm = 0; norm < N - 1; norm++) {
     for (row = norm + 1; row < N; row+=nRowsPerThread) {
       threadStructure = malloc(sizeof(struct normAndRow));
       threadStructure->norm = norm;
       threadStructure->row = row;
       pthread_create(&thread_CS546, NULL, addingThread, (void*)threadStructure);
     }
     pthread_join(thread_CS546, NULL);
   }

   /* Back substitution */
   for (row = N - 1; row >= 0; row--) {
     X[row] = B[row];
     for (col = N-1; col > row; col--) {
       X[row] -= A[row][col] * X[col];
     }
     X[row] /= A[row][row];
   }
 }
```

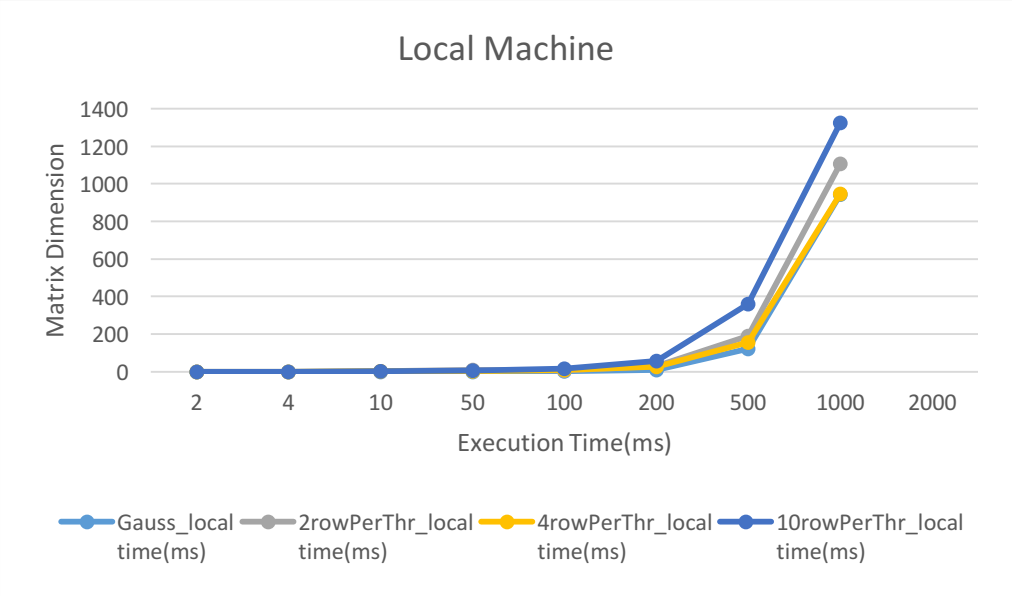
2.2. Comparison

Local machine

This table contains the values obtained from the local machine for the sequential Gauss algorithm and for the 4 different rows per thread in the parallel Gauss algorithm.

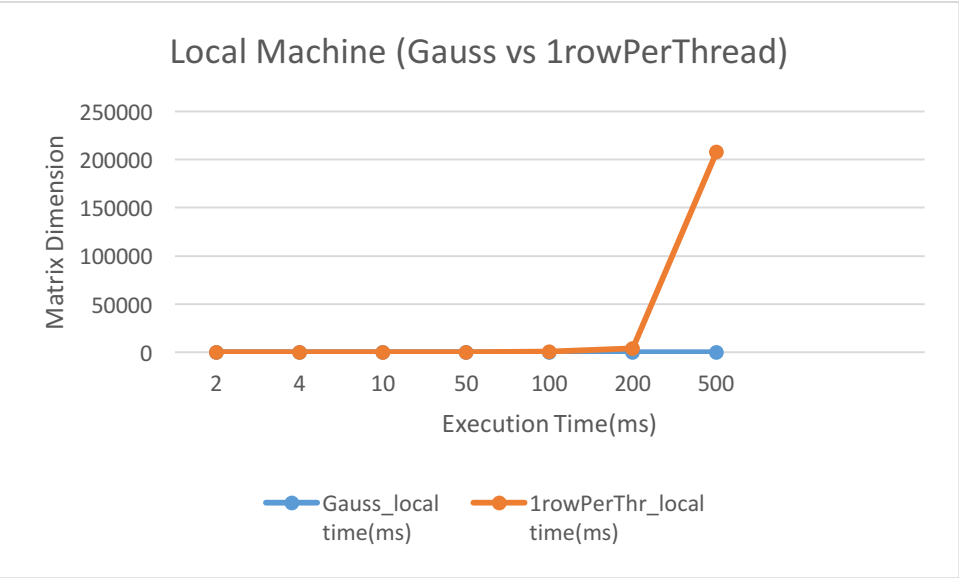
Matrix dimension	Gauss_local time(ms)	1rowPerThr_ local time(ms)	2rowPerThr_lo cal time(ms)	4rowPerThr_lo cal time(ms)	10rowPerThr_l ocal time(ms)
2	0,018	0,419	0,15	0,115	0,119
4	0,02	0,472	0,297	0,266	0,275
10	0,019	1,033	0,584	1,223	1,47
50	0,227	57,309	9,745	3,69	6,922
100	2,464	438,501	5,242	7,396	17,251
200	7,376	3931,35	27,944	27,483	58,181
500	119,774	208043	190,1	157,048	358,786
1000	944		1106,76	946,624	1326,1

This figure shows the comparison between the execution times obtained from the sequential Gauss algorithm and the parallel Gauss algorithm obtained from the values above in the local machine.



From this figure I can conclude that there are not many significant differences in the execution times. This is due to the fact that this execution is done in a local system that needs to create many threads at the same that we increase the matrix's dimensions, therefore the local system is not able to handle the huge amount of computational operations and takes a long time to resolve the algorithm.

The following figure shows that for the case of 1 row being executed per thread, time increases more abruptly as the matrix dimension increases due to the difficulty of handling huge amounts of threads at the same time.

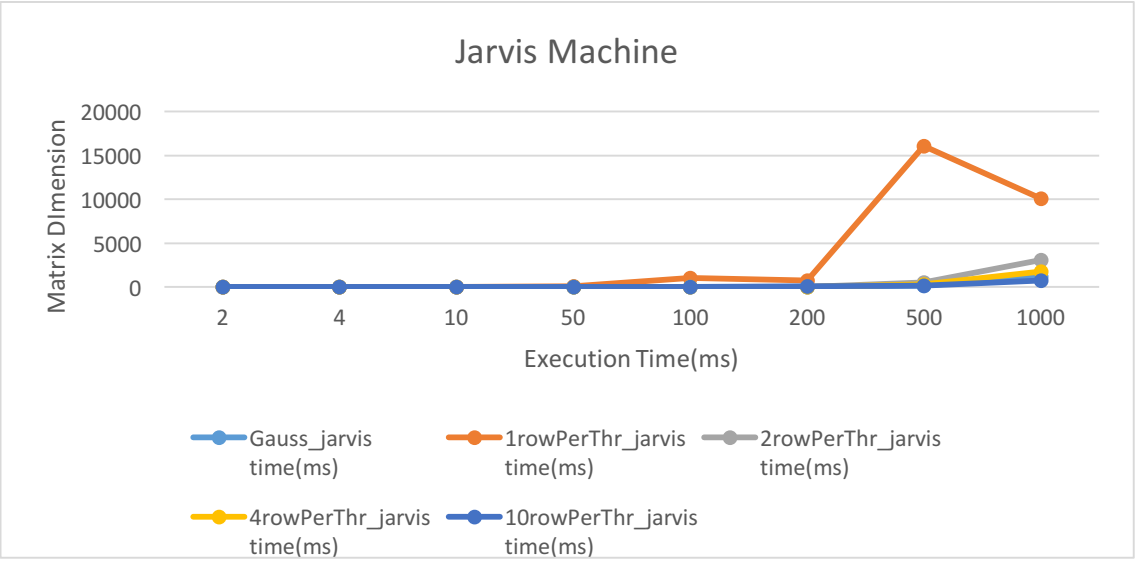


Jarvis machine

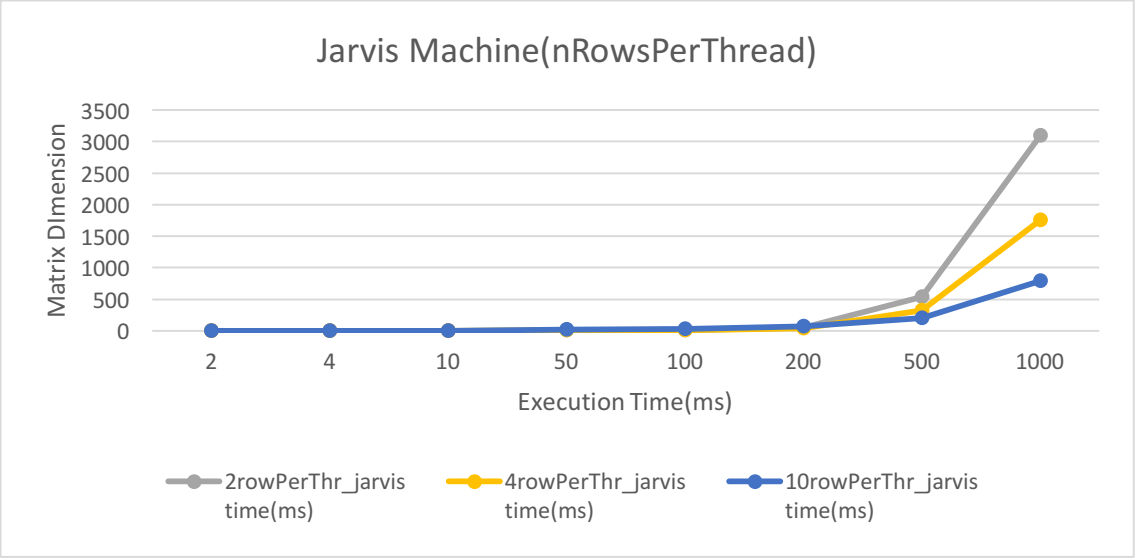
This table contains the values obtained from the Jarvis machine for the sequential Gauss algorithm and for the 4 different rows per thread in the parallel Gauss algorithm.

Matrix dimension	Gauss_jarvis time(ms)	1rowPerThr_ jarvis time(ms)	2rowPerThr_ jarvis time(ms)	4rowPerThr_ jarvis time(ms)	10rowPerThr_ jarvis time(ms)
2	0,01	0,167	0,137	0,187	0,225
4	0,01	0,397	0,366	0,444	0,321
10	0,048	1,873	0,843	3,294	2,019
50	0,562	63,016	3,993	5,469	15,931
100	4,471	1037,48	12,432	13,11	27,015
200	13,524	740,011	46,826	41,168	66,549
500	154,49	16057,7	540,698	323,67	205,676
1000	1181,89	10084,4	3095,45	1750,81	789,886

This figure shows the comparison between the execution times obtained from the sequential Gauss algorithm and the parallel Gauss algorithm obtained from the values above in the Jarvis machine.

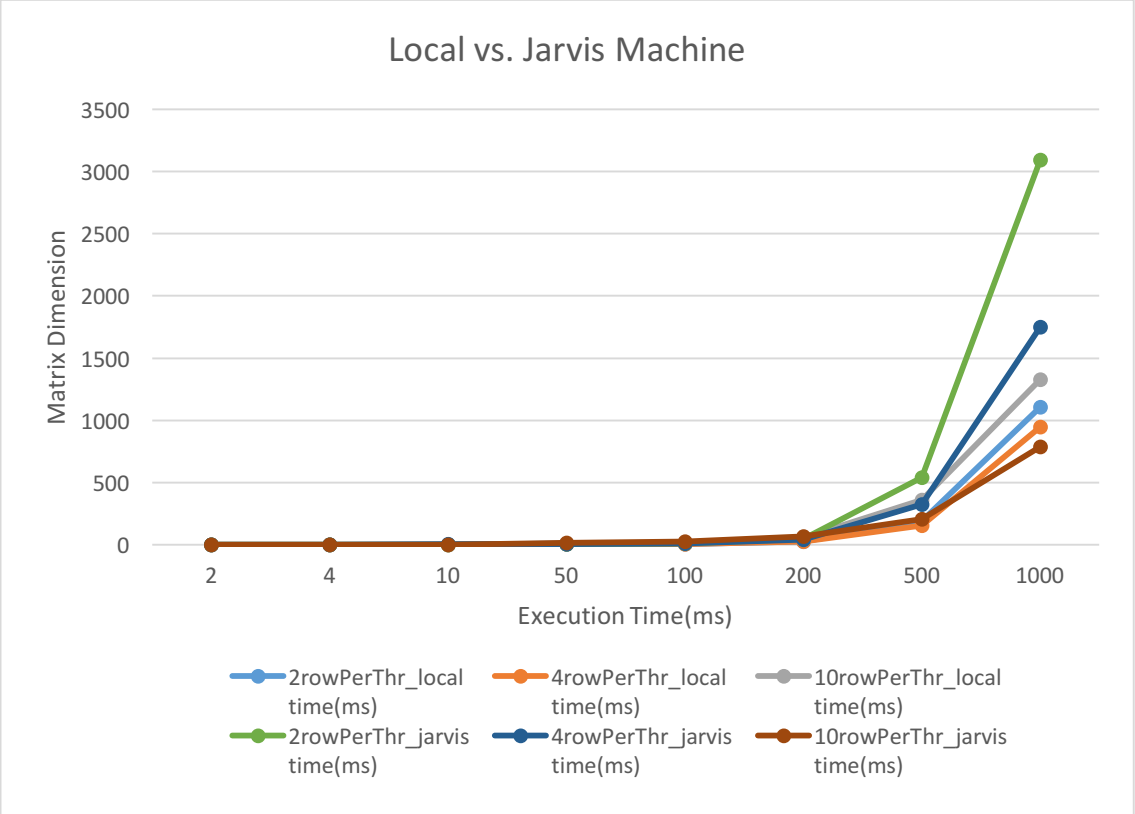


From this figure, compared to the local machine, Jarvis machine executed the Gauss algorithm much faster as it is a supercomputer and it has the necessary resources to execute multiple threads at the same time, having a very low delay.



This figure shows the execution in the Jarvis machine of 2,4 and 10 rows per thread. It can be seen that as the number of rows increases, the execution time is higher as it needs to handle less threads at the same time.

Both machines



This graph shows that Jarvis machine is more efficient that the local system in the execution times. It is expected that for the same rows per thread, Jarvis machine would be more efficient than my local system. In the case of 10 rows per thread, it can be seen that due to the great capacity of the Jarvin machine, for big matrices it is 45% faster that the local machine. On the other hand, in the case of the 4 rows per thread, the local system appears to be 40% faster than the Jarvis machine, but this is not the results expected.

3. Analysis using MPI

In this case an analysis will be made using MPI.

3.1. Code

In the code below, the principal process splits the matrix in groups of rows and information necessary (row to normalize, norm, actual row) to make the computational algorithm to the secondary processes. This data is sent synchronically as the secondary processes require the whole information in order to execute. These processes will make the associated mathematical calculations and will sent the results the main function the value of the vector B associated to each row. Lastly, the main process updates the data and shows the results.

```
int nRowsPerThread;
void gauss() {

  int norm, row, col;  /* Normalization row, and zeroing
      * element row and col */
  float multiplier;

  printf("Computing Parallely with mpi.\n");

  nRowsPerThread = (N-1)/2;

  float row0Buffer[N];
  float valueBuffer[N*nRowsPerThread];
  int sizeRow0Buffer = sizeof(row0Buffer)/sizeof(float);
  int sizeValueBuffer = sizeof(valueBuffer)/sizeof(float);

 for (int l = 0; l < N ; l++) {
        for ( int m = 0; m < N ; m++) {
        printf("Pintando matriz poscion %i %i, valor %f\n", l,m,A[l][m]);
        }
      }

  /* Gaussian elimination */
  for (norm = 0; norm < N - 1; norm++) {

    int processNumber=0;

    for (int  i = 0; i < N; i++) {
      row0Buffer[i]= A[norm][i];
    //printf("Pintando matriz linea 1 poscion %i, valor %f\n", i,A[0][i]);
    //printf("Pintando buffer linea 1 %f\n", row0Buffer[i]);
    }
    for (row = norm + 1; row < N; row+=nRowsPerThread) {
      printf("******ENTORRRRR\n");
      processNumber++;
     for (int j = 1; j < nRowsPerThread + 1 ; j++) {
       for ( int k = 0; k < N ; k++) {
       valueBuffer[(j-1)*N+k]= A[j+((row - 1))][k];
      //printf("Pintando matriz linea 1 poscion %d %d, valor %f\n", j,k,valueBuffer[j+k-1]);
       }
      }
     MPI_Request request;
     MPI_Isend(&row0Buffer, N , MPI_FLOAT, processNumber , 0, MPI_COMM_WORLD, &request);
     MPI_Isend(&valueBuffer,nRowsPerThread*N , MPI_FLOAT, processNumber, 1, MPI_COMM_WORLD, &request);
     MPI_Isend(&norm, 1 , MPI_INT, processNumber , 2, MPI_COMM_WORLD, &request);
     MPI_Isend(&row, 1 , MPI_INT, processNumber , 3, MPI_COMM_WORLD, &request);
   }

  }
  /* (Diagonal elements are not normalized to 1.  This is treated in back
   * substitution.)
   */

  /* Back substitution */
  for (row = N - 1; row >= 0; row--) {
    X[row] = B[row];
    for (col = N-1; col > row; col--) {
      X[row] -= A[row][col] * X[col];
    }
    X[row] /= A[row][row];
  }
}

void addingProcess(){
  int go = 1;
  int col;
  while( go == 1){

    float row0BufferProcess[N];
    float valueBufferpProcess[N*nRowsPerThread];
    float multiplier;
    int norm;
    int row;
    float bBuffer[N];
```

```
    MPI_Recv(row0BufferProcess, N, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(valueBufferpProcess, 1, MPI_FLOAT, 0 , 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&norm, 1, MPI_INT, 0 , 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&row, 1, MPI_INT, 0 , 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    int j;
    for (j=0; j <nRowsPerThread; j++){
      multiplier = valueBufferpProcess[j+j+norm] / row0BufferProcess[norm];
      for (col = norm; col < N; col++) {
        A[row][col] -= A[norm][col] * multiplier;
      }
      B[row] -= B[norm] * multiplier;
    }

    MPI_Request request;
    MPI_Isend(bBuffer, N, MPI_FLOAT,0,4,MPI_COMM_WORLD, &request);

  }

}
```

3.2. MPI Analysis

For the case of the Gauss algorithm using MPI, the principal process needs to sent a huge amount of information to the secondary processes as they don't share the memory. Therefore, MPI is not a valid solution as it created a great delay exchanging data from one process to another. This delay is greater that the actual computational time.

4. Outputfile

The piece of code presented above represents the write function (writefile() in the code) which I use to generate and write the output file required. Using a+, information is appended, simplifying the complexity.

```
void writefile(char *data)
{
    file = fopen("paralell_nThreads.txt", "a+");
    if(file == NULL)
    {
        printf("Unexpected error opening the file");
    }
    fwrite(data, sizeof(char), strlen(data), file);

    fclose(file);
}
```

5. Conclusion

If I was interested in implementing the parallel Gauss algorithm, I would choose pthreads instead of MPI as pthreads shares the memory and doesn't waste time exchanging information with the secondary processes as MPI does. MPI would be a valid solution in the case of parallelizing a vast amount of computational operation with a low requirement of data exchange.