Project 1 Report

Chai Yi 9008308232
chyi13@student.bth.se

1.Blocked Matrix-Matrix Multiplication

1.1 Processors partition

I partition the matrix according to different number of processors. If the program gets N processors, it will partition the result matrix C as m*n blocks (m*n = N). The m and n calculation process is as follows:

```
calcu_rows_cols{
m = sqrt(N);          // initialize m as square root of N
while (N mod m != 0)
    m--;              // find the integer solution
n = N/m;
}
```

1.2 Matrix transposition

The second matrix in matrix-matrix multiplication is used in column-wise. But matrix array is stored as row-wise, and matrix data transmission between master and slave nodes costs a lot of time. So I transpose the second matrix before the real multiplication which I could use the matrix in row-wise.

```
transpose()
```

1.3 MPI implementation

Master node is the node to assign task to slave nodes and it needs to do its own part of the calculation. The process is as follows:

1）Matrix initialization and transposition

Firstly, master initializes two SIZE*SIZE matrices A B and transposes the second matrix B.

2) Task assignment

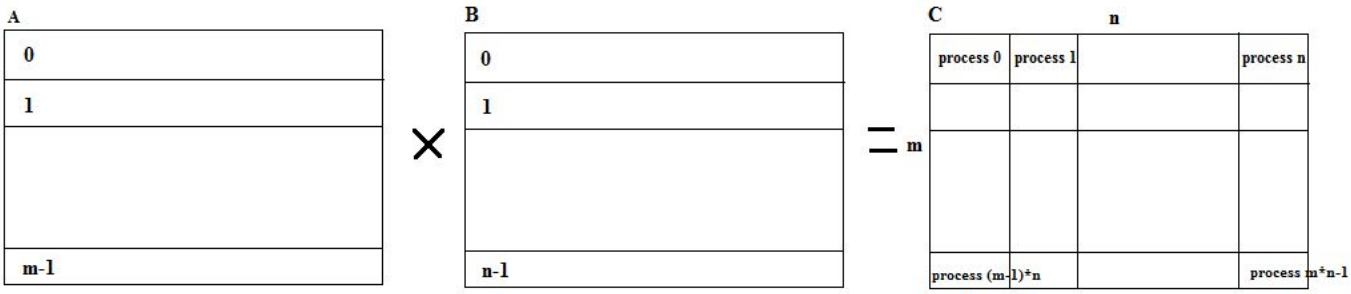

Figure 1

The matrix A is divided into m row blocks and transposed matrix B is into n row blocks which m and n is determined in section 1.1. So a (SIZE/m*SIZE) lines of matrix A and (SIZE/n*SIZE) lines of matrix B is sent to each processor as in figure. Process [row_id,col_id]

deals with row_id row block in matrix A and col_id row block in matrix B as shown in figure 1. For example process 1 deals with multiplication between block 0 in A and block 1 in B.

The assignment of each block(excl master processor) calculated as follows:

```
for each processor_id{
    sub_row = processor_id/n;     // partition in processor level
    sub_col = processor_id - sub_row*n;
    start_row_id = sub_row*(SIZE/m);       // partition in matrix level
    start_col_id = sub_col*(SIZE/n);

    send (start_row_id, processor_id);
    send (start_col_id, processor_id);
    send (sub_matrix_A[] ,processor_id);   // sub_matrix_A starts from [start_row_id][0] in matrix A and it has SIZE*SIZE/m elements;
    send (sub_matrix_B[],processor_id);    // sub_matrix_B starts from [start_col_id][0] in matrix B and it has SIZE*SIZE/n elements;
}
```

3) Master node work

After sending the tasks to the slave nodes, master node will do its own part of the multiplication. The multiplication result is stored in matrix C:

```
for each row_id < SIZE/m{
  for each col_id < SIZE/n{
      for each k<SIZE{
        matrix_C[row_id][col_id] = matrix_A[row_id][k]*matrix_B[col_id][k];
      }
  }
}
```

4) Slave node work

When slave node receives the start_row_id, start_col_id and sub matrix of matrix A, B, it will do its multiplication as in 3). And the result(matrix C) will send back to master node with specific start_row_id and start_col_id. The matrix C only has SIZE*SIZE/m elements which will save a lot of transmission time.

5) Receiving result

Master node will gather block results from all slave nodes, then it will put the blocks back to matrix C.

1.4 Experiment result

1.4.1 Compile

```
[dv2407@mpi_master ~/examples]$ mpicc -o matmul_block matmul_block.c -lm
```

1.4.2 Blocked version and row-wise version performance (1024*1024)

1) 1 node

Blocked: 12.784278s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 1 matmul_block
SIZE = 1024, number of nodes = 1
Execution time on  1 nodes: 12.784278
```

Row-wise: 71.809059s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 1 matmul_mpi_rowwise
SIZE = 1024, number of nodes = 1
Execution time on  1 nodes: 71.809059
```

2) 2 nodes

Blocked: 6.757494s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 2 matmul_block
SIZE = 1024, number of nodes = 2
Execution time on  2 nodes: 6.757494
```

Row-wise: 36.141042s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 2 matmul_mpi_rowwise
SIZE = 1024, number of nodes = 2
Execution time on  2 nodes: 36.141042
```

1) 4 nodes

Blocked: 3.823804s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 4 matmul_block
SIZE = 1024, number of nodes = 4
Execution time on  4 nodes: 3.823804
```

Row-wise: 18.440824s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 4 matmul_mpi_rowwise
SIZE = 1024, number of nodes = 4
Execution time on  4 nodes: 18.440824
```

1.4.3 Conclusion

Transposition blocked version of matrix-matrix multiplication is much faster than row-wise version in different processor situations. Blocked version is 14.61702s faster than row-wise version with 4 nodes environment, 29.383548s with 2 nodes and 59.024781s with 1 node.

2. MPI-based Laplace Approximation

2.1 Work partition

My approach is based on row-wise blocked partition. The matrix is partitioned into N large rows that N is the number of processors current used by the program. Every process has row_size rows and row_size = SIZE/N.

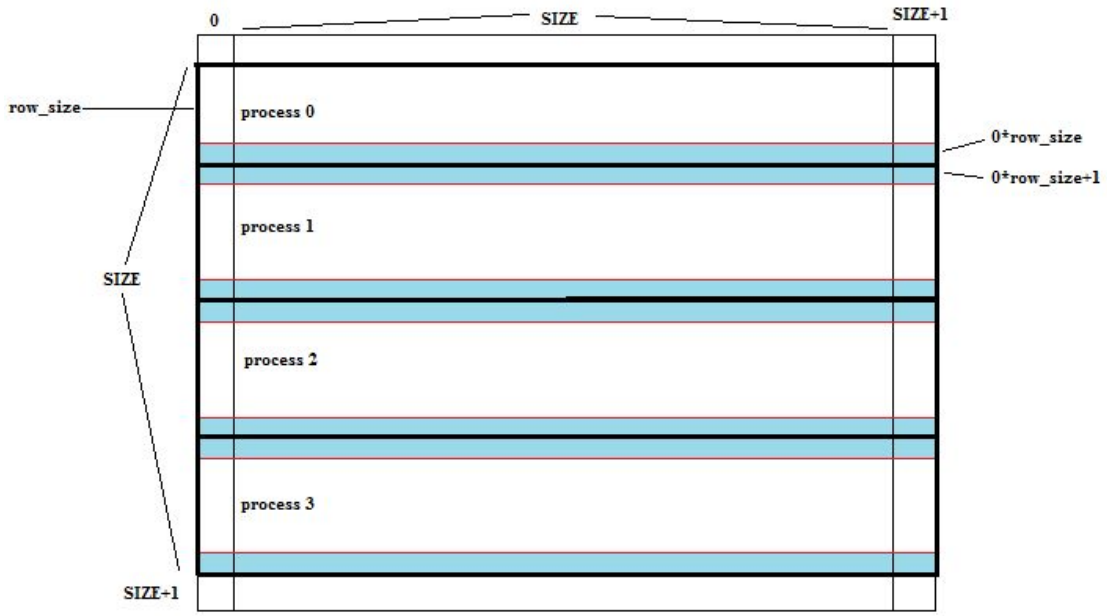

Figure 2

For example, process 0's block is from 1 to row_size in matrix, process 1 is from row_size+1 to row_size*2 and so on.

2.2 MPI implementation

Every process has its part of the matrix and it runs Laplace approximation in its own part. After the Laplace approximation ends, each slave process will send the top line and the bottom line of its blocked matrix to the master process. They also send their greatest sum of the blocked rows. Then master gathers all the result. It sends the new top and bottom lines to corresponding processes that the processes' next runs could continue. The detailed steps is as follows:

1) Assignment initialization

Firstly, the master node initializes a (SIZE+2)*(SIZE+2) matrix. The elements can be either fixed or random generated. Then it will assign the matrix as in 2.1 but the assignment will add two more lines which are the boundary lines. After the assignment initialization, master node starts a while loop to execute the Laplace approximation.

2) Nodes work

Then Master node and slave nodes will do their part of approximation. The work is the same as SOR which partitions the matrix into odd and even. They will also record the greatest sum of the rows which is the same way as SOR_seq.

```
maxi = -INT_MAX;
for each row_id in row_size{
  sum = 0;
  for each col_id from 0 to SIZE+1{
    if ((row_id+col_id)%2 == (count%2)){          //odd or even;
      matrix[row_id][col_id] Laplace approximation;
      sum = sum + matrix[row_id][col_id];
    }
```

```
 }
 if (maxi<sum)
    maxi = sum;
}
```

3) Gathering result

After the calculation, slave nodes will send their top row and bottom row in their blocked matrix to master node. At the same time, every slave node will send maxi to master node.

Master node will find the greatest maxi and compare it with user predefined difference limit. If the maxi is smaller than the limit, the while loop ends. If not, master node will send boundary lines to each slave node for the next loop.

4) Exit

The last step is to notice every slave node to finalize its work and gather up every blocked matrix.

```
EXIT;
for each slave node{
      send_exit_message();
}
Gather;
for each slave node{
      gather_blocked_matrix();
}
```

2.3 Experiment result

2.3.1 SOR sequence result (2048*2048 random number)

1) in master node. Execution time = 179s

```
[dv2407@mpi_master ~/examples]$ ./sor_seq

size      = 2048x2048
maxnum    = 15
difflimit = 0.0204800
Init      = rand
w         = 0.500000

Initializing matrix...done

Iteration: 100, maxi = 16708.319134, prevmax_odd = 16707.358802
Iteration: 200, maxi = 16733.591802, prevmax_odd = 16733.332110
Iteration: 300, maxi = 16742.265243, prevmax_odd = 16742.143935
Iteration: 400, maxi = 16747.104927, prevmax_odd = 16747.025336
Iteration: 500, maxi = 16750.543496, prevmax_odd = 16750.483281
Iteration: 600, maxi = 16753.230185, prevmax_odd = 16753.181913
Iteration: 700, maxi = 16755.417882, prevmax_odd = 16755.378045
Iteration: 800, maxi = 16757.241197, prevmax_odd = 16757.207693
Iteration: 900, maxi = 16758.786345, prevmax_odd = 16758.757746
Iteration: 1000, maxi = 16760.113802, prevmax_odd = 16760.089079
Iteration: 1100, maxi = 16761.267886, prevmax_odd = 16761.246273
Execution time: 179.000000
Execution time: 179.000000

Number of iterations = 1143
```

2.3.2 Parallel row-wise (2048 * 2048 random number)

1) 1 node execution time = 247.48335s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 1 laplace
Execution time on  1 nodes: 247.488335
difflimit: 0.0204800
count: 1616
```

2) 2 nodes execution time = 126.259383s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 2 laplace
Execution time on  2 nodes: 126.259383
difflimit: 0.0204800
count: 1616
```

3) 4 nodes execution time = 66.079157s

```
[dv2407@mpi_master ~/examples]$ mpirun -np 4 laplace
Execution time on  4 nodes: 66.079157
difflimit: 0.0204800
count: 1616
```

2.3.3 Conclusion

4 nodes MPI version is much faster than SOR sequential version. At the same time, running with 4 nodes is 60.180226s faster than 2 nodes and 2 nodes is 121.223967s faster than 1 node.