ME-489 HOMEWORK 4 REPORT

Alperen Çalışkan
Ali Taha Akpınar

December 17, 2023

Abstract

This report explains the MPI communication part of the given homework 4. The code is in C. It already contains the necessary parts regarding the implementation of runSolver function and initial, boundary conditions and the source and necessary file writing commands. It is asked for us to complete the field value creating(q) and the parallelization of the runSolver function using MPI communication inside int main.

1 Introduction

The report starts with explaining the MPI and the functions used ; the pre built functions and their use. It continues with the implementation of the uncompleted part of the code which includes the completion of x coordinates, new field values, rhs array and the messaging part of the ranks. After completing the MPI part the OpenMP part is explained. Pragma for statements are opened for couple of the for loops and it is implemented within the MPI code.

2 MPI Communication

The very first step is to complete the creation of the q field variable by using the x coordinates of every node. So the coordinates are determined as follows:

```
for ( int i = 0; i <= n + 1; i++ ){
     x[i] = x_min + (rank*n + i -1)*dx;
```

After this part the initial conditions are set for every node. The instructor of the course has already implemented necessary parts including file writing. For creating the new field values qn, the following for loop is used.

```
for ( int i = 1; i <= n; i++ ){
      qn[i]=q[i]+dt*rhsq[i];
    }
```

This part helps creating the n number of places inside every q. 2 more values are needed in order to complete boundary connections. Meaning 0 and n+1'th nodes are needed. That is solved by using MPI communications. Below is how it is done:

```
if(rank<size-1){
      double *bye=(double*) calloc(1,sizeof(double));
        bye[0] = q[n];
        //bye_2 = q[1];
        MPI_Send(bye,1,MPI_DOUBLE,rank+1,tag,MPI_COMM_WORLD);
```

```
        //MPI_Send(bye_2, MPI_DOUBLE,rank-1,tag,MPI_COMM_WORLD);
      }

    if(rank>0){
      double *hello=(double*) calloc(1,sizeof(double));
      //hello_2;
        MPI_Recv(hello,1,MPI_DOUBLE,rank-1,tag,MPI_COMM_WORLD,&status);
        //MPI_Recv(hello_2,1,MPI_DOUBLE,rank+1,tag,MPI_COMM_WORLD,&status);
        q[0]=hello[0];
        //q[n+1]=hello_2;
      }
```

In this part using the rank value the n'th node value of every rank except the last one is send to the 0'th node of the following rank and the 0'th node value of every rank except the first rank is send to the n+1'th node of the rank before that one. The right hand side Q values are attained after this communication. Using the following equation:

$$rhs(q,t) = k\frac{d^2}{dx^2}(q[i-1] - 2q[i] + q[i+1]) + f(x[i],t) \tag{1}$$

The following code part is used for this equation's implementation.

```
for (int i = 1; i <= n; i++ ){
      rhsq[i]=(k/pow(dx, 2))*(q[i-1]-2*q[i]+q[i+1])+source(x[i],time_new);
      }
```

The code is now completed as far as the MPI is concerned.

3 Outcome of the MPI

The code is run following these commands:

```
$ mpicc -g -Wall -o HW_4_out HW_4.c -lm
$ mpirun -n 1 HW_4_out 10
```

3.1 Execution with 10 nodes

```
dt :  2.500000  Wall clock elapsed seconds = 0.000041
```

Upon different trials the following times are attained:

```
Wall clock elapsed seconds = 0.000102
Wall clock elapsed seconds = 0.000072
Wall clock elapsed seconds = 0.000055
Wall clock elapsed seconds = 0.000054
```

An average of 0.000065 seconds is obtained for one processor. For two processors the following results are obtained:

```
dt :  0.666667  Wall clock elapsed seconds = 0.000139
Wall clock elapsed seconds = 0.000141
Wall clock elapsed seconds = 0.000159
Wall clock elapsed seconds = 0.000206
Wall clock elapsed seconds = 0.000166
```

The average time is 0.000162 seconds in this case. Because the dt depends on both the number of nodes and the size(rank size), it changes at every different processor and node amount. For four processors the following results are obtained:

```
dt :  0.163934  Wall clock elapsed seconds = 0.000340
Wall clock elapsed seconds = 0.019283
Wall clock elapsed seconds = 0.000290
Wall clock elapsed seconds = 0.000839
Wall clock elapsed seconds = 0.016873
```

The average time is 0.007525 seconds in this case.

3.2 Execution with 50 nodes

```
dt :  0.103093  Wall clock elapsed seconds = 0.004199
all clock elapsed seconds = 0.004175
Wall clock elapsed seconds = 0.004038
Wall clock elapsed seconds = 0.004329
Wall clock elapsed seconds = 0.004337
```

An average of 0.004216 seconds is obtained for one processor. For two processors the following results are obtained:

```
dt :  0.025445  Wall clock elapsed seconds = 0.000730
Wall clock elapsed seconds = 0.000617
Wall clock elapsed seconds = 0.002502
Wall clock elapsed seconds = 0.001935
Wall clock elapsed seconds = 0.001871
```

The average time is 0.001531 seconds in this case. For four processors the following results are obtained:

```
dt :  0.006309  Wall clock elapsed seconds = 0.004901
Wall clock elapsed seconds = 0.011216
Wall clock elapsed seconds = 0.028324
Wall clock elapsed seconds = 0.009337
Wall clock elapsed seconds = 0.008929
```

The average time is 0.012541 seconds in this case.

It is clear that the time increases as the number of nodes increases as expected. Since more nodes require more calculations. It comes as a surprise that for almost every case the time increased as the number of processors have increased.

4 An OPENMP Implementation

The sections where the update of heat values occurred can be parallelized with using OPENMP. This new implementation is a combination of both MPI and OPENMP. The following code is portions where the code structure has changed:

```
#pragma omp parallel for
  for (int i = 1; i <= n; i++ ){
    rhsq[i]=(k/pow(dx, 2))*(q[i-1]-2*q[i]+q[i+1])+source(x[i],time_new);
    }
#pragma omp parallel for
  for ( int i = 1; i <= n; i++ ){
    qn[i]=q[i]+dt*rhsq[i];
```

```
#pragma omp parallel for
 for ( int i = 1; i <= n; i++ ){
   q[i] = qn[i];
 }
```

The program can be run as same as the previous version on the terminal.

4.1 Execution with 10 nodes

```
dt :  2.500000  Wall clock elapsed seconds = 0.000071
Wall clock elapsed seconds = 0.000101
Wall clock elapsed seconds = 0.000106
Wall clock elapsed seconds = 0.000105
Wall clock elapsed seconds = 0.000101
```

An average of 0.0000968 seconds is obtained for one processor. For two processors the following results are obtained:

```
dt :  0.666667 Wall clock elapsed seconds = 0.001514
Wall clock elapsed seconds = 0.001320
Wall clock elapsed seconds = 0.001288
Wall clock elapsed seconds = 0.001214
Wall clock elapsed seconds = 0.002297

```

The average time is 0.0015266 seconds in this case. For four processors the following results are obtained:

```
dt :  0.163934  Wall clock elapsed seconds = 8.031095
Wall clock elapsed seconds = 6.077350
Wall clock elapsed seconds = 5.932502
Wall clock elapsed seconds = 5.739295
Wall clock elapsed seconds = 5.967465
```

The average time is 6.3495414 seconds in this case.

4.2 Execution with 50 nodes

```
dt :  0.103093  Wall clock elapsed seconds = 0.003391
Wall clock elapsed seconds = 0.003147
Wall clock elapsed seconds = 0.003075
Wall clock elapsed seconds = 0.003019
Wall clock elapsed seconds = 0.003083
```

An average of 0.003143 seconds is obtained for one processor. For two processors the following results are obtained:

```
dt :  0.025445  Wall clock elapsed seconds = 0.024692
Wall clock elapsed seconds =  0.028371
Wall clock elapsed seconds = 0.025912
Wall clock elapsed seconds = 0.028500
Wall clock elapsed seconds = 0.025576
```

The average time is 0.0266102 seconds in this case. For four processors the following results are obtained:

```
dt :  0.006309  Wall clock elapsed seconds = 111.971888
Wall clock elapsed seconds = 101.454792
Wall clock elapsed seconds = 124.732619
Wall clock elapsed seconds = 92.959652
Wall clock elapsed seconds = 104.674587
```

The average time is 107.1587076 seconds in this case.

By comparing the results with MPI and OPENMP implementation, it is obvious that creating parallel regions in already MPI-paralleled region results in decrease in execution speed. This may be because the parallel regions create a racing condition, or the incompatibility between working principles of OPENMP and MPI.

5 Conclusion

It is observed that the execution time is not decreasing properly in both structures. The speed increases after changing from one process to two processes; however, the speed gets lower than that of one process when the number of processes are four. This is mainly because the number of processors reserved for the virtualbox does not exceed 4, thus the thread count after that does not only reduce efficiency but decreases it. This may be due to racing condition where threads have to wait for each other to finish updating. With OPENMP implementation, the program runs slower than the previous version. It would be better to use OPENMP only, or MPI only to get better reductions in computing time.