Universitat Autònoma de Barcelona

PARALLEL PROGRAMMING

SERGIO OYAGA & CARLOS MOUGAN

MPI

Submitted To:
Ana Sikora
Parallel Programming
Computer Science
Department

Submitted By :
Sergio Oyaga
Carlos Mougan
MSc in Modelling
CRM

1 Motivation & Description

For this assignment the goal is to improve the execution time of Laplace equation code by using MPI. Message Passing Interface (MPI) is a standardized and portable message-passing standard. The standard defines the syntax and semantics of a core of library routines useful to a wide range of users writing portable message-passing programs.

For this problem, let's suppose that we are going to use N processes to solve the problem in a distributed way. Accordingly with the characteristics of the problem, each of these processes will have to carry on the computation over a portion of the matrix A, taking care for interchanging the necessary data and synchronizing with other processes. The specific processes that will have to communicate will depend on how the data is partitioned among them. The simplest partition of A is shown in

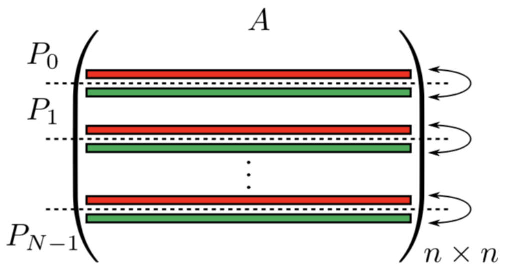

Figure 1: Diagram of the partition of the matrix A

Figure 1. In this case, each process P_m takes care of the computation of m/N rows of A and, in the general case ($0 < m < N - 1$), it needs the last row of P_{m-1} and the first of P_{m+1}.

We are going to use bash in order to execute in parallel our code using different configurations. Multiple sockets in the same computer o multiple sockets in multiple computers. This is teh reason why we are going to use the Wilma Cluster tool.

1.1 Wilma Cluster

The Wilma Cluster is composed by 12 execution resources/node. Each node offer 2 sockets and 6 core/socket.

In order to use the cluster we have to access with a different user (aoclspv). So we are executing two different user in the same computer "aolin" & "aoclspv".

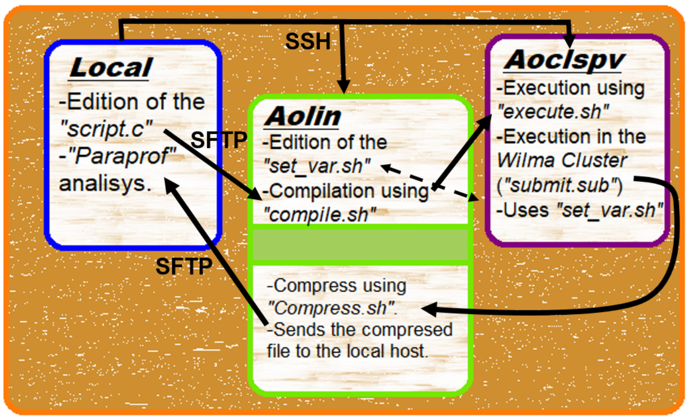
SSH
Local
-Edition of the "script.c"
-"Paraprof" analisys.
SFTP
SFTP
Aolin
-Edition of the "set_var.sh"
-Compilation using "compile.sh"
-Compress using "Compress.sh".
-Sends the compresed file to the local host.
Aoclspv
-Execution using "execute.sh"
-Execution in the Wilma Cluster ("submit.sub")
-Uses "set_var.sh"

Figure 2: Wilma Cluster Methodology

In figure 2 we see the data transition flux between the computers and the two users in the same computer. The local host is connected via SSH with the two users in the remote server. Also via SFTP with the AOLIN user in order to send and receive the needed files.

The local host allows us to edit the ".c" files and visualize the performance. We use TAU in the local host because the SSH connection doesn't allow us to charge graphical interfaces.

To agile the execution we have used bash scripts "*set_var.sh*" , "*execute.sh*" & "*compress.sh*".

Once we have the edited files we send it to the "aolin" user. In this user we edit the "*set_var.sh*" file. This file edits the execution parameters, the file names & and exports the TAU metrics. Once this file is edited we can execute the "*compile.sh*" and this uses the "*set_var.sh*" and compiles the C file.

Once compiled we use the other user "AOCLSPV" to execute the C code using the WILMA CLUSTER via the "*submit.sub*" file, appending it to the execution queue.

Once the execution is finalised, using the "AOLIN" user we compress and send the information via SFTP to the local host. Now we can execute the performance.

2 No optimized program

Our initial code is a sequencial solution of the already common Laplace 2D problem. Were the main process consists in modifying step by step a bidimensional matrix.

The MPI optimization consists in a subdivision of the sequential process of the matrix in M parallel processes.

The size of the matrix will be $n \cdot n$ subdivided in m blocks. The size of M obviously depends in the number of process.

Each process is run in parallel and independent way. Each process has a size of:

$$\frac{n}{M} + 2$$

Due to the system communication between process, each process its only aware of the before and after process (Listing 1).

As consequence of the Laplace problem nature we need know what are the surrounding points. That's the reason of the "+2" in the dimension, it stores the boundary of the neighbours matrix.

To summarize we have m process that subdivide the original matrix and send and receive the first and last n values to the closest neighbors.

```
int iter = 0;
 while ( global_error > tol*tol && iter < iter_max )
 {
   iter++;
   if (rank > 0) {
                 MPI_Isend(&A[n],n,MPI_FLOAT,rank-1,1, MPI_COMM_WORLD, &request);
                 MPI_Recv(&A[0],n,MPI_FLOAT,rank-1,2, MPI_COMM_WORLD,&status);
                 }
   if (rank < size - 1 ) {
                 MPI_Isend(&A[(n*(m-1))+1],n,MPI_FLOAT,rank+1,2, MPI_COMM_WORLD,&request);
                 MPI_Recv(&A[(n*m)+1],n,MPI_FLOAT,rank+1,1, MPI_COMM_WORLD,&status);
                 }
   error= laplace_step (A, temp, n, m);
   MPI_Reduce(&error,&global_error,1,MPI_FLOAT,MPI_MAX,0,MPI_COMM_WORLD);
   float *swap= A; A=temp; temp= swap;
 }
```

Listing 1: No Optimized MPI Code

3 Optimized Code

In order to improve the execution time we will try different optimization strategies. After taking into consideration different methods as:

- Overlapping communication and computation
- Block partitioning
- Block communications
- Hybrid solution

We decided to implement the "Overlapping communication and computation" optimization because it seemed to help us understand deeper the communication process & and how the non sequential movement of information works.

```
int iter = 0;
int i = 0;
 while ( global_error > tol*tol && iter < iter_max )
 {
   iter++;
        memcpy(Aprim,&A[n],sizeof(n));
        memcpy(Aprimlast,&A[(n*(m-1))+1],sizeof(n));
   if (rank > 0) {
                  MPI_Isend(&Aprim[0],n,MPI_FLOAT,rank-1,1, MPI_COMM_WORLD, &request);
                  }
   if (rank < size - 1 ) {
                  MPI_Isend(&Aprimlast[0],n,MPI_FLOAT,rank+1,2, MPI_COMM_WORLD, &request);
                  }
        error= laplace_step (A, temp, n, m,1);
  if (rank > 0) {
                  MPI_Recv(&A[0],n,MPI_FLOAT,rank-1,2, MPI_COMM_WORLD,&status);
                  }
   if (rank < size - 1 ) {
                  MPI_Recv(&A[(n*m)+1],n,MPI_FLOAT,rank+1,1, MPI_COMM_WORLD,&status);
                  }
   error_step = laplace_step (A, temp, n, m,0);
   error>error_step? error_step:error;
   MPI_Reduce(&error,&global_error,1,MPI_FLOAT,MPI_MAX,0,MPI_COMM_WORLD);
   float *swap= A; A=temp; temp= swap;
 }
```

Listing 2: Optimized MPI Code

In contrast with the no optimized code each process sends first the data to his neighbors, then computes the Laplace step without computing the border number point. Doing this, each process allows to the rest to send the information. So when we read them, they are already sent. Once read we can compute the border points.

The rest of the code achieves the same as the non optimized.

4 Performance

In this section we will measure the different performance achieve by tuning hyper parameters of the codes.

The **first temporal comparison** can be seen in figure 3. In this figure we can see the performance of the non optimized code and the optimized code, letting the CPUs and number of process fixed. Be aware that both axis are in logarithmic scale in order to be able have a more clear graphical representation.

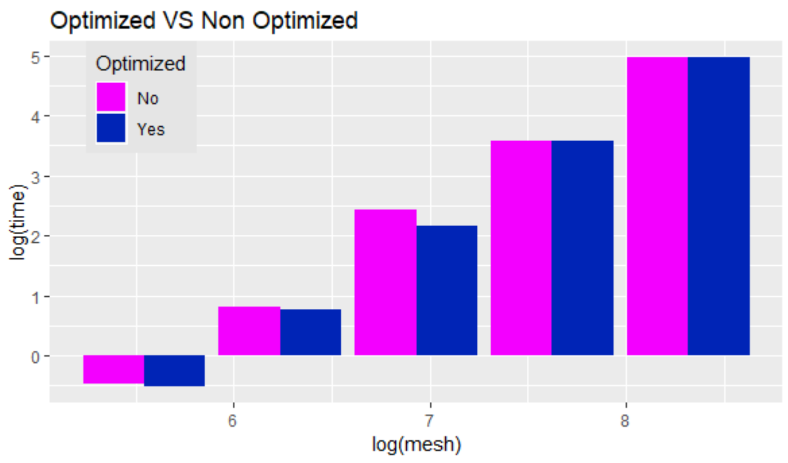

Figure 3: Performance of N process in 1 core vs in N cores

We can observe that the execution time is reduced in every case for the optimized. That's why we decide only to analyze the performance of the optimized code

The **first idea** that we had is to compare what is faster the **communication between process** in the same computer or between several computers, maintaining the total number of process. This can be seen in figure 4.

We can slightly appreciate that the communication between process in the same computer is faster than between process in different computers.

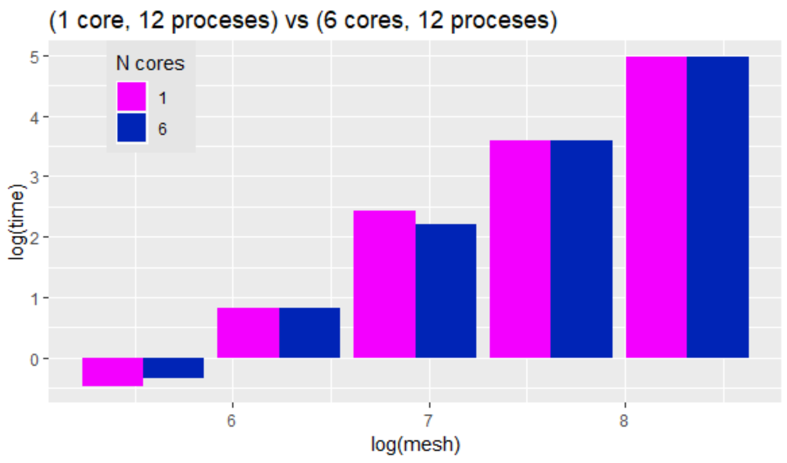

Figure 4: 1 core versus 6 core and 12 process

That's indicates us that if the problem doesn't require a big number of processes it's better to parallelized in the same computer. That won't always be possible if our application requires a lot of memory or a lot of parallel process.

Once seen figure 4, the **second idea** that we thought that it will be useful to visualize the improvement in temporal performance between increasing the number of processes in the same core or the number of cores in Figure 5.

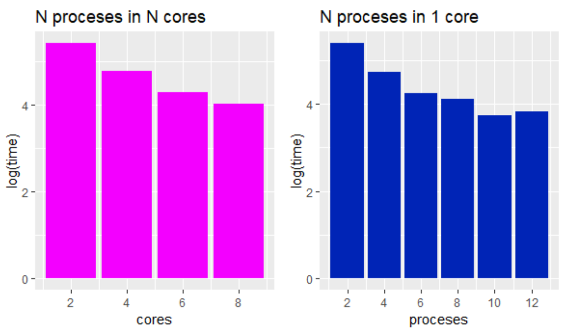

Figure 5: Diagram of the partition of the matrix A

We can observe that the decreasing tendency of the temporal complexity its slightly higher (decreases faster) in the N process N cores than in the N process 1 core.

4.1 Tool Performance Utilities

Another way to see the performance of our code is using specific tools as TAU, that is a is a performance evaluation tool that enables us to:

- To supports parallel profiling and tracing
- Profiling shows you how much (total) time was spent in each routine
- Tracing shows you when the events take place in each process along a timeline

This kind of tools can be really useful to visualize specificity's of the execution. We can use this point of view to understand how much and where do you spend your execution time as well as when the communication occurs. In figure 6 we attach the time performance evaluation of the optmized code.

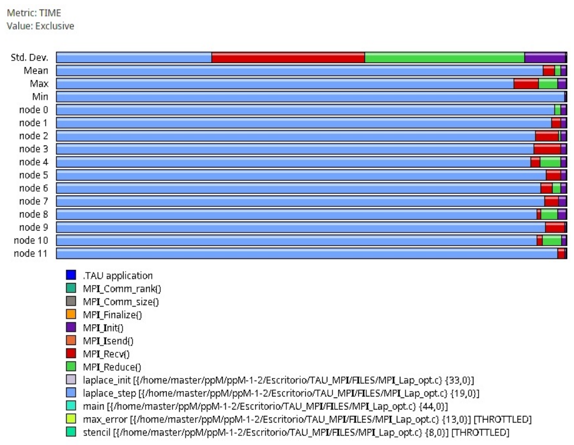

Figure 6: Time Performance with TAU

In Figure 7 we can see where the most relevant events occur. In this graphical representation we can see an example of the Time Line of internal operations of each one of the used process. Essentially, it shows us the information exchanged between different threads in chronological order. This helps us to understand how information/messages are sent and received by the different process to perform the parallelisms.

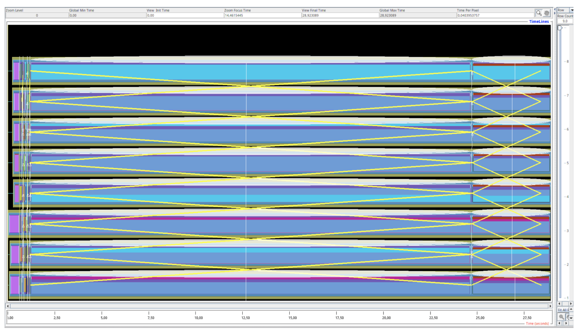

Figure 7: Timeline of 8 process using Jumpshot

5 Conclusions

After implementing the MPI tool, optimizing the code & measuring the performance of the different parameters. We evaluated that the MPI has an advantage over OpenMP. MPI access paralely to memory while OpenMP can't access to the memory in a simultaneously way.

Also MPI process doesn't need a controler that specifies their own information flux, it is not needed to point where each process goes or comes. On the other hand OpenMP has a controler that manages the threads.

As a disadvantage involving the usage of MPI requires the manual paralelization while OpenMp compiler does it by itself.

As suggested in the assignment we thought about combining MPI and OpenMP together, but we have decided not to do it due to that there is a limit in parallelizing a code, and we consider that is not worth it increase the paralelization of our code for this case.

As seeing in the images, if possible, its better to communicate process inside one computer than communicate process between several computers.

Along the development of the assignment we have had the opportunity to get in touch with remote servers, to understand the difference between OpenMP and MPI, we were able to dig deeper in the knowledge in the communication process of a computer and we were able to improve our abilities in evaluating the performance of the algorithm.