INF-3201 Parallel Programming Shared Memory

Enrico Tedeschi
ete011@post.uit.no

I. Introduction

The goal of this assignment is to parallelize a piece of code using shared memory techniques preferably using **OpenMP**.

A. Requirements

- Choose a piece of software to parallelize
- Parallelize it with shared-memory techniques
- Evaluate speedup

II. Technical Background

– Concurrency and parallelism concepts
– Parallel programming concepts
– Basic programming approach
– Knowledge of C language
– Notion of design pattern principles
– Theory about software engineering
– Knowledge of git to manage the software versions

III. Analysis

[2]Programming a message-passing multicomputer can be achieved by:

1) Designing a special parallel programming language
2) Extending the syntax/reserved words of an existing sequential high-level language to handle message-passing
3) Using an existing sequential high-level language and providing a library of external procedures for message-passing

In this assignment the third implementation will be applied by using MPI (Message Passing Interface).
The Mandelbrot set is a problem which has a computation time of $O(z^2)$ and its equation is:

$$z_0 = 0 \quad (1)$$

$$z_{k+1} = z_k^2 + c \quad (2)$$

To compute and display the Mandelbrot set, a significant amount of time is required. Displaying the Mandelbrod set is an example of processing a bit-mapped image and as with all these kinds of problems it could be speedup by using parallel computation.
The given sequential code, **RoadMap.c**, displays a map of 800x800 pixels with a zoom level equal to 10.
The most computationally expensive call is the **solve** function in the CreateMap procedure which is called for each pixel of the display:

```
for(y=0;y<HEIGHT;y++){
    for(x=0;x<WIDTH;x++){
      colorMap[y][x]=palette[solve(translate_x
    (x),translate_y(y))];
    }
  }
```

the function computes each pixel of the Mandelbrot set so it could be the most propriate part of the code to parallelize. The function solve is executde 800x800 times per each zoom level:

```
inline int solve(double x, double y)
{
  double r=0.0,s=0.0;

  double next_r,next_s;
  int itt=0;

  while((r*r+s*s)<=4.0) {
    next_r=r*r-s*s+x;
    next_s=2*r*s+y;
    r=next_r; s=next_s;
    if(++itt==ITERATIONS)break;
  }

  return itt;
}
```

It became apparent that the *solve* function was the most expensive function in matter of time when analysing the *C* program *RoadMap* with *GProf*:

```
 %      self
time    seconds    name
80.29   12.53    solve (RoadMap.c:62)
 6.48   1.01     solve (RoadMap.c:63)
 6.10   0.95     solve (RoadMap.c:63)
 3.43   0.54     solve (RoadMap.c:62)
```

```
  3.43   0.54     translate_x (RoadMap.c:36)
```

To run on the cluster the program with GProf:

```
sh run.sh gprof
```

Another fraction of the given RoadMap code which is relevant in the computation is the visualization of the set. However this part won't be taken into consideration while improving the code with MPI.

IV. Implementation

A. Design and Main Schema (??)

To solve the problem, the SPMD (Single Process Multiple Data) computational model has been chosen. Each process will execute the same code, in that way a statement is necessary. To provide the statement the **Master/Slave Design Pattern** has been implemented. It could be useful to use it while developing a multi-task application and it gives you more control of your application's time management.[1]

To achieve the communication, blocking MPI send and receive are being used in either static or dynamic version.

```
MPI_Send(&d, s, ty, pr, msg, comm);
MPI_Recv(&d, s, ty, pr, msg, comm, &status);
```

- d: data to send
- s: size of the data
- ty: type of the data
- pr: process to send or receive the data
- comm: communicator
- status: status of the receiver (always setted on MPI_STATUS_IGNORE in this project)

The total amount of work has been divided in rows, each row has 800px to solve and the static and dynamic version have a different approach of how the work is divided and how the processes are assigned.

B. Static Implementation

In the static implementation each process already knows how many rows to solve, in that way no time is spent in sending initial messages between master and slaves processes. Each slave has to calculate at the beginning how many rows to compute.

```
int rows = (HEIGHT / (size-1));
```

Where the HEIGHT is the total number of rows and size is the amount of processors involved in the computation; -1 because the slaves are $size - master$. The master receives the result back from each slave ensuring that a correct crc is received and that the solution is displayed on the screen.

As said before this is an SPMD programme as such is good to have state which control it. To distinguish between the slave and the master MPI libraries are provided. Every time the code starts from one process there is a check on the *rank*:

```
if (rank == 0)
  //MASTER code
else
  //SLAVE code
```

where the rank is the calling process in the communicator.

```
int rank;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

*1) **Master**:* The master waits for each row to receive a message from all the slaves with the solution of the problem. In the previous version the master waited for each pixel, so the message size was smaller but the messages were definitely too many and the computation did not even look parallel, taking almost the same time as the sequential one.
In the optimized static version the buffer size is $WIDTH + 1$ instead of 3 elements which are the colour in the [x][y] position and x and y coordinates. Sending pairs of coordinates of individual pixels will result in excessive communication. Indeed it contains all the colours of the row, and the number of the row in the last position of the array.

```
int buffer[WIDTH+1];
  /*
  * 0-(WIDTH-1) -> colours
  * WIDTH -> y coordinate
  **/
```

The master doesn't send anything to the slaves, it just receives data from them when they are finished. To organize better the *for* cycles and to guarantee that the master will receive data $WIDTH * slaves$ times there are three of them: the first one iterates for each slave, the second one for each row and the third one for each pixel and the receiver is just outside the third one.

```
int size;
MPI_Comm_size(MPI_COMM_WORLD, &size);
int s; //slave
for(s = 1; s < size; s++){
  for(i = (rows*(s-1)); i < (rows*s); i++){
    MPI_Recv(&buffer, WIDTH+1, int, from_any,
    tag, comm, status);
    for(j = 0; j < WIDTH; j++){
      y = buffer[WIDTH];
      colorMap[y][j] = buffer[j];
      crc += colorMap[y][j];
    }
  }
}
```

once the buffer has been received, the master takes care of filling colorMap with the new colours and updates the *crc* for the final checksum. When the master has received all the rows from the slaves it computes (if there are any) the remaining rows from the $HEIGHT/(size-1)$ operation. They are cycled with this formula:

```
for(i = (rows*(size-1)); i < HEIGHT; i++){
  //cycle each pixel and solve it
  }
```

*2) **Slave**:* Each slave compute a fixed number of rows which is established when it starts the computation and sends the buffer to the master containing the information of one row per time once the solve function is completed. To determinate the start row, a start variable is created following this rule in each slave:

```
int start = rows * (rank-1);
```

in this way the first slave will start from 0 until rows-1, the second one from rows until $rows*2$, and so on until the last one which will be from $rows*(size-1)$ until $HEIGHT-remainder$. The remainder of the rows are computed from the master.
The code below shows how the buffer is sent to the master:

```
for(i = start; i < (start + rows); i++){
  buffer[WIDTH] = i;
  for(x = 0; x < WIDTH; x++){
    buffer[x] = palette[solve(translate_x(x),
    translate_y(i))];
  }
  MPI_Send(&buffer, (WIDTH+1), int, master,
    tag, comm);
}
```

Even if at the end the crc is correct, to avoid imperfection in the visualization due to some delay in any processes, at the end of each call of $CreateMap()$, a **barrier** is implemented; it blocks until all processes in the communicator have reached this routine.

```
MPI_Barrier(MPI_COMM_WORLD);
```

Nonetheless the use of the barrier is for visual purpose only and it could have been avoided if that would have caused relevant further computation.

C. Dynamic Implementation (**??**)

In the dynamic implementation the master calls the first free process available in the work-pull. Using a work-pull ensures a dynamic load balancing which is great to achieve the best speed up possible. Master and slaves communicate each other by sending array of *int* which contains the information of one row of the set.

```
int buffer[WIDTH+2];
  /*
  * 0-(WIDTH-1) -> colors
  * WIDTH -> row
  * WIDTH+1 -> rank
  * */
```

It's important in the dynamic version to keep track of which process has just finished the job by knowing its rank, in that way the master could immediately send more work to it.

*1) **Master**:* The master starts with sending the first n rows to the n slaves ($n = size - 1$), one row per slave. After that a while-true loop begins and the master starts to receive the data. Per each data received the count variable must be incremented and when $rcv\_count = HEIGHT$ then the master can send the poison ($row = -1$) to all the slaves and the while cycle breaks. Likewise, the master sends the rows only if $row < HEIGHT$ and the while cannot be broken if the master didn't finish sending and either receiving.

```
int rcv_count = 0;
do{
  if(rcv_count < HEIGHT){
    MPI_Recv(&buffer, from_any, result_tag);
    rcv_count++;
    /* ... color the map and get crc ...*/
    if (row < HEIGHT) {
      MPI_Send(&row,buffer[WIDTH+1],row_tag);
      row++;
    }
  }
  else{
    int poison = -1;
    for(k = 1; k < size; k++){
      MPI_Send(&poison, k, row_tag);
    }
    break;
  }
}while(1);
```

The variable rcv_count is incremented only when the master receive a message and the variable row is incremented only when it sends a row to a certain slave.

*2) **Slave**:* The execution of the slave is really simple, it receives the first row, solves it, sends the result to the master and then receives data until $row! = -1$.

```
MPI_Recv(&row, master, row_tag);
while(row != -1){
  buffer[WIDTH] = row;
  for(x = 0; x < WIDTH; x++){
    buffer[x] = palette[solve(translate_x(x),
    translate_y(row))];
  }
  MPI_Send(&buffer, master, result_tag);
  MPI_Recv(&row, master, row_tag);
}
```

D. Environment

The code has been developed using JetBrains CLion 1.1 on Windows 10 and due to a more easy way to compile and test the MPI programs with a Linux based console, the compilation and the execution of the code has been made on an Ubuntu Virtual Machine with 4 core assigned, using VirtualBox 4.2 and Ubuntu version 14.04.3
To synchronize the cluster and the local machine and to keep trace of all the changes in the code, a git repository was created and the git command on linux were used to commit and to push/pull data from repositories.
The benchmarking and the test with more than 4 cores has been done in the UiT uvrocks cluster. The connection with the cluster was established by using ssh on a linux machine:

```
ssh –X –A ete011@uvrocks.cs.uit.no
```

It was necessary also to make some changes in the Makefile since the programs to run are three: *RoadMap, RoadMap_static, RoadMap_dynamic*.

V. Result and Benchmarking

For the benchmarking, the sequential version is compared with the final dynamic version and with two static versions on the local machine using 4 cores and on the cluster using "profile". The first static version was not the optimized one and there was an exchange of messages for each pixel of the set, which makes the computation too slow. In the second one the buffer has a bigger size and the slaves compute the entire row before sending the result to the master.
To evaluate the static and dynamic version is necessary to edit the *run.sh* script by changing the name of the file you want to execute and edit the *Makefile* in such a way that it generates the files you want to evaluate, for example by adding the **RoadMapProf_dynamic** file in it:

```
RoadMapProf_dynamic: RoadMap_dynamic.c
  $(CCP) $(CFLAGS) -pg RoadMap_dynamic.c -o
    RoadMapProf_dynamic $(INC) $(LIB)
```

It is necessary now to edit the *run.sh* script, for example, it is possible to edit it in that way if a *profile* evaluation on the dynamic version wanted to be done:

```
if [ "$1" = "profile" ]; then
  mpirun [..commands] RoadMapProf_dynamic
  otfprofile RoadMapProf_dynamic.otf
  pdflatex result.tex
```

and then executing it with *run.sh*:

```
sh run.sh profile
```

The solve function is called from the *CreateMap* function, so it will be necessary to modify it in the static and dynamic version to get the speed up. The main tests are executed on the cluster using the given $run.sh$ script. The *profile* evaluation has been used to test the static and the dynamic version using 5 and 10 cores.
The Fig **??** represents the summary of the execution of the dynamic version on the cluster using 10 cores. The function which takes more time is *solve* but also the MPI_Recv plays a good part in the computation, that means that the dynamic version could be improved more by trying to optimize the MPI_Recv calls, maybe avoiding all the 16100 messages received and implementing something more balanced in matter of *number-of-calls/size-of-buffer*.

The histograms generated, Fig **??**, show how the processes distribute the work from each other. The process 0 is the master. It is the process which works for longer time and makes more invocations (send and receive calls). The second histogram shows that in the slaves most of the time is taken to send the data while in the master most of the computation time is to receive the data.

In the **static** version with 10 cores, Fig **??**, the master (process 0) takes care only about receiving data and the message length is exactly the same for each process. As can be seen in Fig **??** most of the time in the static version is taken from the synchronization, or else the MPI_Barrier. At every cycle, indeed, each process has to wait that all the other processes have finished their work before starting with the next zoom level. This could be optimised by saving the state for each level and letting the master work on it while the slaves continue computing without interruptions.

The graphic in Fig **??** represents the $Speedup(x)$ where *x = number of processes*. The evaluation was made running the static and the dynamic version on the cluster with the number of processes from 2 to 15. The speedup (on the y line) is calculated: $Speedup(x) = sequential\_time - parallel\_time(x, v)$; where x is the number of cores and v is the version which could be static or dynamic. The parallel_time(x,v) is an average of 5 execution on the cluster with the same settings.
The graphic in Fig **??** represent the time that each function (**RoadMap, RoadMap_static, RoadMap_dynamic**) takes to compute the problem. Both graphics show that the parallel version is always faster than the static one, most likely because of the

Barrier in the static version and because in the dynamic implementation the master provides always a job for the inactive slaves. However the graphics also show that if using two cores the computation is slower than the sequential one; that is because with two cores there will be one master with only one slave working, so basically the code works like the sequential one but with all the MPI calls of the parallel implementation.
The scaling with more than 15 number of processes is basically irrelevant, indeed using more than 15 cores the improvement is almost null.

VI. Discussion

It is also possible to consider the scaling in matter of **size** and **zoom levels**. That is, how good the parallel programs scale compared to the sequential version if you consider the size of the map and the zoom levels? In Fig **??** it is possible to see that the sequential version, more the size is increased more the time to solve the function goes up, and for $x \to \inf$ the sequential function looks almost exponential; on the other hand, in the static and the dynamic version the time increase so slowly and the function looks more linear, even with a huge map to compose such as *2000x2000 px*.

The same happens with scaling the zoom level like in Fig **??**. We could say that the sequential version has a linear incrementation with a function comparable to $y = mx$ with $m = 1$ while the parallel versions have a value of m closer to $m = 1/n$ where *n* is bigger larger the zoom level.

VII. Conclusion

As we can see in Fig **??** and Fig **??**, running the static and the dynamic programs on the cluster gave us an improvement on the speed up which is good for the purpose of this project.
However the speed up could also be further improved and also the instance of using only two processes should be taken into account. I'm satisfied about the obtained results and I'd also consider the scaling part as a success.

References

[1] Application design patterns: Master/slave. http://www.ni.com/white-paper/3022/en/, 2006.

[2] Barry Wilkinson and Michael Allen. Parallel programming: Techniques and applications using networked workstations and parallel computers.