OpenMP, MPI and CUDA Parallelization of Recursive Functions
Project of High Performance Computing
Academic Year: 2022/2023

Group Members:

Mayhar Sadeghi Garjan, 5283082

Edoardo Pastorino, 5169595

December 27, 2022

Contents

Chapter 1

Introduction

This report is the document associated to the implementation of our final project for the High Performance Computing's course of the academic year 2022/2023, at the University of Genova (UNIGE).

The aim of the project is to try to implement and explain a possible approach to parallelize recursive function, based on already existent experiments found online, with the usage of three of the most diffused interfaces for exploiting parallelism inside the code of a programming language. These API are OpenMP, an API to explicitly direct multi-threaded shared-memory parallelism, MPI, a communication protocol for parallel programming, specifically used to allow applications to run in parallel across a number of separate computers and CUDA, a parallel computing platform and programming model developed by NVIDIA for general parallel computing on GPUs. All these three parallelization methods are implemented, in our work, inside program in C language, due to the fact that are well supported by the language and by the relative compilers (there are also other language that allows to use these APIs like C++ and Fortran)

1.1 What is Recursion?

Recursion is a way for solving problems, sometimes also complex one, with the usage of a function that calls itself several times until we reach a predefined termination condition, called base case, and so we obtain the solution. The working criteria is to break down a problem into smaller and simpler sub-problems and then solve those sub-problems recursively (Divide et Impera method).

A general recursion function is composed by two main parts:

```
function(...)
{
    if( Base Case Condition )
    {
        // Base Case
    }
    // Recursive Structure
    function(...)
}
```

- Base case, the smallest version of the problem for which we already know the solution or a terminating condition where a function can return the result immediately.
- Recursive structure, finding the solution to a problem via the solution of its smaller sub-problems.

In general, when the main function of a program is called, the memory is allocated on the stack, a data structure that allows to allocate and deallocate dinamically the data used by functions, following the Last In First Out (LIFO) criteria. When the main function returns some result the previous memory is deallocated. The same happens for a recursive function, in which for each function call a relative memory is allocated on the top of the stack, until the stoppping criteria is met. Now

the memory starts to deallocate from the top of the stack as we get the return value of the function.

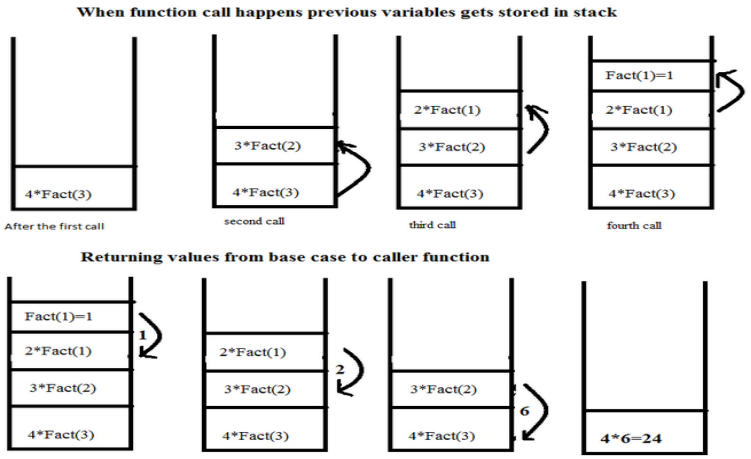

Figure 1.1: Stack Representation of recursive factorial

As the above image demonstrates, for each function call we create a new stack. This behavior isn't good in terms of overhead, because each function call needs its own set of local variables and parameters. It will take extra processing time compared to a loop-based approach. In fact, if you use too many resources with recursive function calls, we can reach the stack overflow situation, where the program can crash just because of too many nested calls. So we can avoid to use recursion when an iterative solution, using a simple loop, will get the job done, due to the fact that a loop could process a billion elements using the same stack.

1.2 What is OpenMP?

OpenMP (Open Multi-Processing) is an application programming interface that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran. The main feature is that allows you to use many threads in parallel, in a single process, for increasing the execution performance of a program, exploting the single program multiple data parallelism (SPMD).

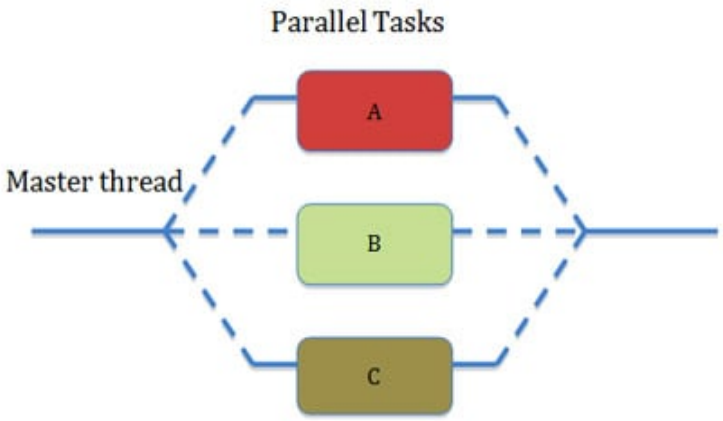

Figure 1.2: Execution model of OpenMP

OpenMP consists of a set of:

- compiler directives, responsible of defining which part of the code will run in parallel.
- Library routines, a set of user level routines like, for instance, the one used for retrieving the number of threads in a parallel region.
- Environmental variables, for adjusting the runtime behavior of parallel aplication.

OpenMP is a shared memory architecture, meaning that all the multiple threads, that we decide to use, can access to the same whole main memory. A single process, in fact, has its own memory, file ecc. . . , and it can create a certain number of threads that share some process state information, like the memory. For this reason, threads are considered very lightweight tasks, since they can reduce memory overhead. This API is also very diffused in parallel programming because it is available with almost all the compilers, like GCC or ICC, and also is always updated for dealing with new programming methods and technologies.

1.3 What is MPI?

Message Passing Interface (MPI) is a portable message-passing standard designed to function on parallel computing architectures. The aim of this API is to allow you to use multiple processes (not threads) for computing parallelizable code. The MPI standard defines the syntax and semantics of library routines that are useful for writing portable message-passing programs, again in C, C++ and Fortran.

Differently from OpenMP, in MPI the resources are local, this means that each process operates in its own environment and so it has to communicate with the others thanks to the usage of messages, while, like OpenMP, is also a parallelism based on Single Program Multiple Data criteria.

Even if MPI is very portable and scalable (and also well known) suffers from some performance issues, like high memory overhead and error-prone nature of message-passing system. So for this reason is always suggested to spend as little time as possible communicating data.

The three main operations that form the core of MPI are:

- Send(message), for sending data from a sender process to a receiver one.
- Receive(message), used by a receiver process for receiving message from a sender process.
- Synchronization, a collection of synchronization functions to regulate the work of many processes.

The MPI communication, instead, can be subdivided in two main types:

- Point to point communication, is the basic communication method between two processes for exanching data, one has the role of the sender and the other has the role of the receiver.
- Collective communication, is a different method of communication which involves all the processes defined inside a communicator (an environment in which happens the exchange of data).

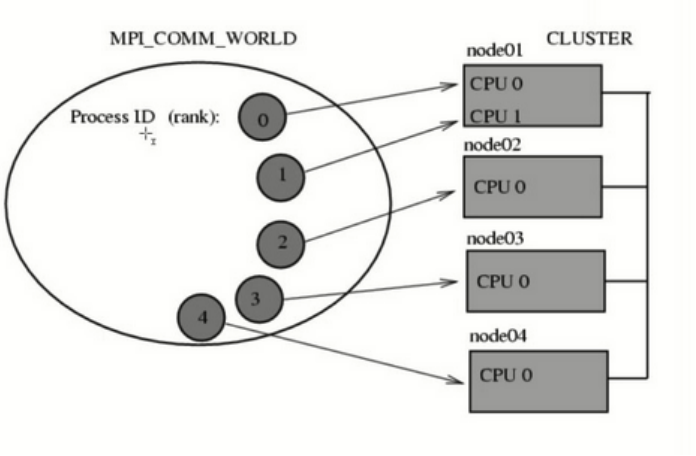

Figure 1.3: MPI message passing procedure

1.4 What is CUDA?

CUDA (Compute Unified Device Architecture) is a developer toolkit to compile programs and run them easily in an heterogeneous systems, using a set of high level programming language extensions to use a GPU (Graphical Process Unit) as a co-processor for speed up compute-intensive application, exploiting the power of GPUs for the parallelizable part of the computation.

If we chose CUDA in a programming language (like in C, C++, Python and Fortran) we have the possibility to use API to manage host and device components. A general CUDA program is divided in many serial sections of code (host) that are performed by the CPU and in few parallel ones that are instead performed by GPU (device), again following the SIMD mode. Host and device have separate memory and so it is necessary to transfer data between CPU and GPU using a way similar to the point to point message passing.

The CUDA GPUs parallel execution of code is based on stream computing, that is the subdivision of dataset into stream of elements. A single computational function (called kernel function) works on each element. In this scenario multiple cores handle multiple elements in parallel launching multiple threads.

The CUDA GPUs hierarchy is a 2-level hierarchy, it is organized in many stream-

ing multiprocessors, each of which is in turn composed by lots of cores.

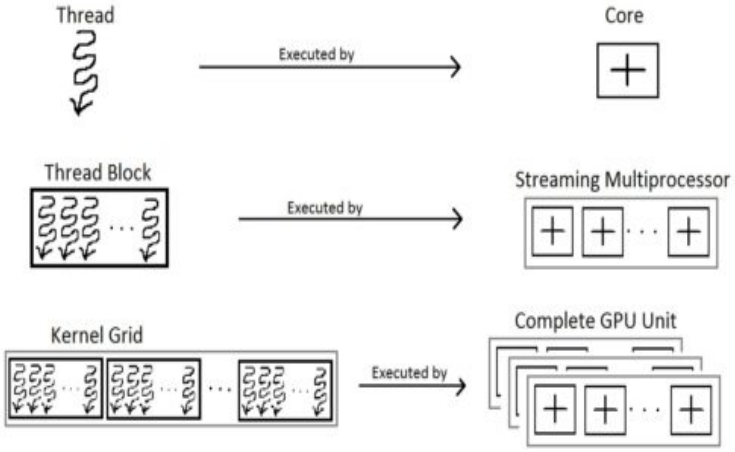

Figure 1.4: How CUDA execution and hierarchy work

Chapter 2

Parallelization of Recursive Function

Now that we know what is recursion (even if there are always new mystery to discover about this topic!) and that we know what are the main ways to perform parallel execution of programming code, we can dive more into details of parallelization of recursive function.

As we said in the introduction, we have tried to parallelize this specific kind of function with three different APIs (OpenMP, MPI and CUDA) used inside in C language programs. For each tools we propose an implementation for exploiting all the parallelization features, we describe the result in terms of execution time, overhead and correctness, confronting them with a more classical approach of recursion.

The algorithm used as sample for recursive function parallelization is the well-know, didactical oriented and famous Fibonacci algorithm. This procedure is, by definition, very suitable for recursion, in fact, Fibonacci is a series in which the sum of every subsequent term is composed by the sum of previous two terms in the series.

```
int fibonacci(int number){ // number is the number for wich we want
                           // to retrieve its fibonacci value in the series
    if(number == 0){
```

```
        return 0;         // base case
    }else if( number == 1){
        return 1;
    }

     int x = fibonacci(number - 1); // recursion
     int y = fibonacci(number - 2); // recursion

    return x + y;

}
```

2.1 Fibonacci with OpenMP

The first API used for parallelizing code is Intel multi-threading parallelism interface OpenMP.

Initially, the first thing that we can do, and that it is suggested to do, to exploit code optimization, is to use the provided compiler's options and flags. For our purpose the used compiler is icc, a compiler for Intel processor-based systems, available for Windows, Linux, and macOS operating systems.We have made this choice because this compiler has more optimization features with respect to the basic gcc for C language. So an example of icc compilation command for OpenMP's optimization can be the following:

```
$ icc -O2 -qopenmp -xHost file.c -o file.out
```

in which we have used some compilation options like:

- The -O2 flag; this option allows to have the second level of icc optimization of the code (it is also the default if you don't specify any level). O2 level is a local op-

timization trade-off between compilation speed, optimization, code accuracy and executable size.

- The -qopenmp flag; it enables the usage of OpenMP directives inside the program and activates the loops' auto-vectorization. Vectorization is the unrolling of a loop combined with the generation of packed SIMD instructions by the compiler. Of course, for recursion, vectorization isn't performed since we don't have loop by definition of recursion's structure.

- The -xHost option; it is called in question for using the largest available register's size (in our case, the used machine has instruction set's maximum size of 512b) to speed up the computation.

After the compilation optimization, we have to manipulate the code to insert the useful pragma directives for having the explicit parallelization of the desired portion of code, that in our case are the recursive function's call and the implementation of the function itself.

```
 #include <omp.h>
int fibonacci(int number);
int main(int argc, char* argv[]){
    int res;
    double start_time = omp_get_wtime();
    #pragma omp parallel num_threads(128)
    {
        #pragma omp single
        {
            res = fibonacci(atoi(argv[1]));
        }
    }
    double run_time = omp_get_wtime() - start_time;
    printf("Total DFTW computation in %f seconds\n",run_time);
    printf("The fibonacci of %s is %d\n", argv[1], res);
```

```
}

int fibonacci(int number){
    if(number == 0){
        return 0;
    }else if( number == 1){
        return 1;
    }
    int x, y;
    #pragma omp task shared(x)
    {
        x = fibonacci(number - 1);
    }
    #pragma omp task shared(y)
    {
        y = fibonacci(number - 2);
    }
    #pragma omp taskwait

    return x + y;
}
```

As we can see in the code above, the first used OpenMp pragma directive is the pragma omp parallel that defines the beginnig of the code's parallel region. We can also specify how many threads we want inside this region, for finding the best configuration in terms of scalability and efficience. Inside the paralell region, we define the pragma omp single directives that allows, to only one of the previous number of threads, to enter the recursive function fibonacci(). This single thread is responsible of the creation of the two initial tasks inside the function. Each of the two pragma omp task has its own variable (x for task1 and y task2) shared because, by default, the variables inside a task are firstprivate, but only shared variables can be ineherited.

The taskwait directive, at the end of the function, is required for synchronization purpose because, in this way, the current task wait the completion of child tasks.

This first method for parallelizing recursive fibonacci function doesn't reach a nice execution time, that is even worse than the time obtained by the normal way of implementing this procedure. This behavior is caused by the high overhead of creating lot of tasks.

So we can do additional modifications to the code.

```
#include <omp.h>
int fibonacci(int number);
int main(int argc, char* argv[]){
    int res;
    double start_time = omp_get_wtime();
    #pragma omp parallel num_threads(128)
    {
        #pragma omp single
        {
            res = fibonacci(atoi(argv[1]));
        }
    }
    double run_time = omp_get_wtime() - start_time;
    printf("Total DFTW computation in %f seconds\n",run_time);
    printf("The fibonacci of %s is %d\n", argv[1], res);
}

int fibonacci(int number){
    if(number == 0){
        return 0;
    }else if( number == 1){
        return 1;
    }
```

```
    int x, y;
    #pragma omp task shared(x) \
    if(number > 30)
    {
        x = fibonacci(number - 1);
    }
    #pragma omp task shared(y) \
    if(number > 30)
    {
        y = fibonacci(number - 2);
    }
    #pragma omp taskwait

    return x + y;
}
```

The only difference with the previous code are the two if cluases specified in each omp task definition. With this method we are able to avoid the creation of task for small value of "number", reducing parallelization overhead when it is not needed. The performance now is better but not so wonderful as expected. In fact there is a way for skipping the OpenMP overhead.

```
#include <omp.h>
int parallel_fibonacci(int number);
int serial_fibonacci(int number);
int main(int argc, char* argv[]){
    int res;
    double start_time = omp_get_wtime();
    #pragma omp parallel num_threads(128)
    {
        #pragma omp single
        {
```

```
            res = parallel_fibonacci(atoi(argv[1]));
        }
    }
    double run_time = omp_get_wtime() - start_time;
    printf("Total DFTW computation in %f seconds\n",run_time);
    printf("The fibonacci of %s is %d\n", argv[1], res);
}

int parallel_fibonacci(int number){
    if(number == 0){
        return 0;
    }else if( number == 1){
        return 1;
    }
    if(number <= 30){
        return serial_fibonacci(number);
    }
    int x, y;
    #pragma omp task shared(x)
    {
        x = parallel_fibonacci(number - 1);
    }
    #pragma omp task shared(y)
    {
        y = parallel_fibonacci(number - 2);
    }
    #pragma omp taskwait
    return x + y;
}

int serial_fibonacci(int number){
```

```
    if(number == 0){
        return 0;
    }else if( number == 1){
        return 1;
    }
    int x = serial_fibonacci(number - 1);
    int y = serial_fibonacci(number - 2);
    return x + y;
}
```

The last version of parallel Fibonacci, as we said before, avoid overhead for creating omp directives, calling the parallel version of the function only when "number" is greater than a certain specific value, in our case, greater than 30. Otherwise we execute a basic version of fibonacci not to waste time in useless parallelization actions. With this new configuration we obtain an execution time of the function that is a quite better compared to the normal version's time.

The following table summarizes the difference between the basic recursive approach of Fibonacci and the three OpenMP methods described before.

Fibonacci Input Number 40	
Approach	Execution time
Basic	4.252604 seconds
OpenMP V1	6.379852 seconds
OpenMP V2	4.712478 seconds
OpenMP V3	0.325947 seconds

The image below, in a similar way, shows the different trends of the execution time curves, of the various parallelization approaches, respect to the basic one, considering different input number of Fibonacci. As we can see the third OpenMP approach is really the best one, in fact, its execution time is almost constant for these intervals of numbers. The same is not true for the others three cases.

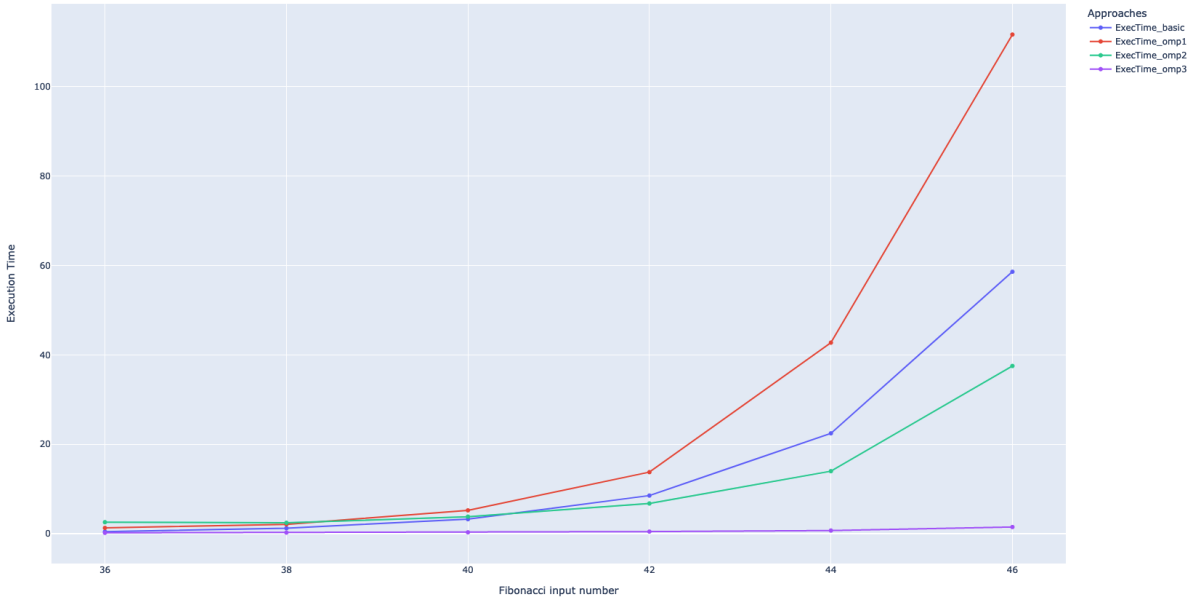

Figure 2.1: Difference between basic and omp approaches of Fibonacci recursion

2.2 Fibonacci with MPI

MPI paralellization of the Fibonacci recursive function can be implemented using MPI Comm spawn. This function takes the following parameters as argument:

- command, name of program to be spawned (string, significant only at root).
- argv, arguments to command (array of strings, significant only at root).
- maxprocs, maximum number of processes to start (integer, significant only at root).
- info, a set of key-value pairs telling the runtime system where and how to start the processes (handle, significant only at root).
- root, rank of process in which previous arguments are examined (integer).
- comm, intracommunicator containing group of spawning processes (handle).

The MPI Comm Spawn is the main function which on it's own is not recursive. To make the recursive paradigm, we need to create two files:

- Main file (root.c), the access point for starting the whole process of recursive paralellization.
- Worker file (fib.c), in which Fibonacci function is really implemented.

Using Spawn, we will create children with their own binary files and, at the mean time, the parents (in which children are created) will wait for the results from their children. Explicitly, we can't see recursion when we look at each program, solely, but, if we follow the procedure of calling children and parents, we will see that the overall program creates children recursively when the "command" parameter is the same in fib.c file. In other words, it creates process of itself with different number of fibo computation and it waits for it to be completed. Hence, the way we are creating processes and dividing jobs between new children follows a recursive paradigm.

To explain the procedure better we will start from root (mpi-fib-start.c) file:

```
Char[] command= ./ fib ;
if (n < 2){
  printf ("fib(%ld)=%ld\n", n, n);
  exit (0);
}
else{
  printf ("<root> spawning recursive process, n = %ld\n", n);
  MPI_Comm_spawn (command, argv, 1, local_info, myrank, MPI_COMM_SELF,
  &children_comm, errcodes);
}
printf ("<root> waiting receive\n");
MPI_Recv (&fibn, 1, MPI_LONG, MPI_ANY_SOURCE, 1, children_comm,
MPI_STATUS_IGNORE);
printf ("fib(%ld)=%ld\n", n, fibn);
MPI_Finalize ();
```

At root file we are creating children with another executable file called "./fib" which is in the same directory. Till now recursion has not been occurred. But in the ./fib file which is demonstrated below, we will create children of the same file which share no memory and wait foreach other to finish. They will continue to create processes as long as we meet the condition.

fib.c (executable file should be compiled and the name of output file should be the same as command below):

```
Char[] command="./fib"; // Name of the executable file
if (n < 2){ // Condition that finishes the process and sends the result to
    parent
  printf ("<%ld> returning fib(n) < 2\n", n);
  MPI_Send (&n, 1, MPI_LONG, 0, 1, parent);
}
else{
  printf ("<%ld> spawning new process (1)\n", n);
  MPI_Comm_spawn (command, argv, 1, local_info, myrank,
  MPI_COMM_SELF, &children_comm[0], errcodes); //Recursion Occurs here
  printf ("<%ld> spawning new process (2)\n", n);
  MPI_Comm_spawn (command, argv, 1, local_info, myrank,
  MPI_COMM_SELF, &children_comm[1], errcodes); //Recursion Occurs here
  printf ("<%ld> waiting recv fib(n-1) > 2\n", n);
  MPI_Recv (&x, 1, MPI_LONG, MPI_ANY_SOURCE, 1,
  children_comm[0], MPI_STATUS_IGNORE); );//Receive result from children
  printf ("<%ld> waiting recv fib(n-2) > 2\n", n);
  MPI_Recv (&y, 1, MPI_LONG, MPI_ANY_SOURCE, 1,
  children_comm[1], MPI_STATUS_IGNORE);//Receive result from children
  fibn = x + y; // computation
  printf ("<%ld> returning fib(n) > 2\n", n);
  MPI_Send (&fibn, 1, MPI_LONG, 0, 1, parent);//Send the result to the parent
}
```

```
printf ("<%ld> returned (isend) fib(n) \n", n);
MPI_Finalize ();
```

Here, we will use spawn twice since the computation of fibo series depends on two calls: fibo(n)=fibo(n-1)+fibo(n-2). Since the command refers to the same program, recursion occurs. Root file will be executed once.

For compiling the MPI program we have to use the specific mpiicc Intel compiler for this kind of program. A typical compilation command for our case can be:

```
$ mpiicc file.c -o file.out
```

In this configuration it is also very important the way by which we execute the MPI program:

```
$ mpirun -np 1 ./file.out
```

This mpirun command starts the execution of the program with the specif -np number of processes (we can also specify the number of processors that we want to use, for example with -np 10 and -hostfile list.txt -perhost 1 means that we execute 1 of the 10 processes in each machine specified in a file). In our case, we execute only one process in one single machine because we want to associate only one process to the execution of the starting point program (mpi-fib-start.c), since it will create the other number of processes for doing recursion.

This version is not suitable for recursion at all, so it is better not ot use MPI for this kind of problem, in fact our implementatio works only with few number of input for Fibonacci and takes a huge amount of time so we don't have any kind of result because are useless and it is also difficult to run the program. Some Possible performance issues are listed below:

- Since we are opening a new binary file using spawn, we shall meet a limit based on OS that we are using. In every OS, there's a limit for the opening files. In the cluster that we used , default value was 1024 which can only support fibo(10).

Using the command unlimit -n 2048 we can increase this support value which we can compute fibo(12) maximum.

- Since we are creating processes with no shared memory there will be a huge amount of overhead, meaning that we will waste lots of time for communications. Hence this program didn't provide an acceptable result in terms of time consumption

2.3 Fibonacci with integration between OpenMP and MPI

In this section we want to show a possible implementation of the second approach of recursive function's parallelization, the MPI multi-processes configuration, with the integration of the multi-threading paradigm offered by the OpenMP interface, used inside each process, since the first version of MPI had some problems, especially for computing the big numbers of fibo.

2.3.1 OMP + MPI Version 1

In the first version of the integration between OpenMP and MPI, in few words, we split the execution of the program in the available processors (in the used cluster we have 8 machines), for exploiting the hardware as much as possible, then, in each processor, we compute the recursion using the third, and the best, version of the OpenMP approach for parallelizing it, only when we reach the "leaf" of the recursion tree. Since we have 8 hosts machines, we can reach only the 4th depth of the recursion tree and we starts the recursion in the leafs present at this level.

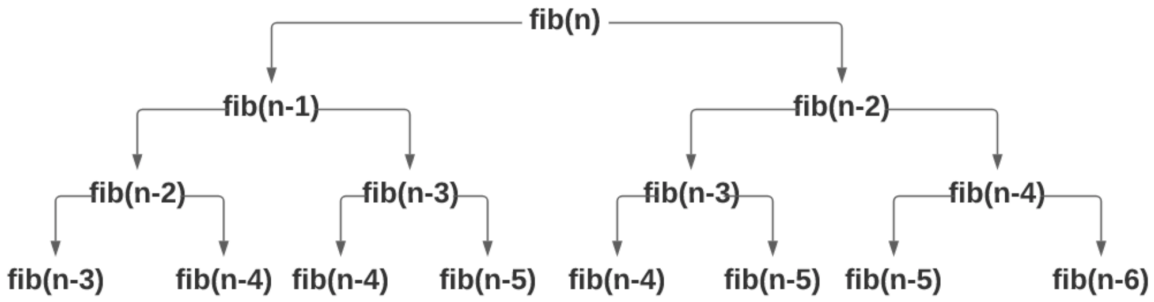

Figure 2.2: Partial tree of Fibonacci recursion

Now the execution doesn't occur using the MPI paradigm but inside the OpenMP multithreading approach (OMP version 3), and this is more efficent, in terms of overhead and execution time. The code below explains what we have just said.

```
int main(int argc, char* argv[]){
   err=MPI_Init(&argc, &argv);
   err=MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   err=MPI_Comm_size(MPI_COMM_WORLD, &size);
   int final_res=0;
   int res0=0,res1=0,res2=0,res3=0,res4=0,res5=0,res6=0,res7=0;
   if (rank==0){
       res0=callOMP(atoi(argv[1])-3,rank);
       MPI_Recv(&res1, 1, MPI_INT, 1, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
       MPI_Recv(&res2, 1, MPI_INT, 2, 0, MPI_COMM_WORLD,
           MPI_STATUS_IGNORE);
       .
       .
       .

       MPI_Recv(&res7, 1, MPI_INT, 7, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
```

```
    final_res=res0+res1+res2+res3+res4+res5+res6+res7;
    }
    else if (rank==1 ){
        res1=3*callOMP(atoi(argv[1])-4,rank);
        MPI_Send(&res1, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }
    else if(rank==2){
        res2=3*callOMP(atoi(argv[1])-5,rank);
         MPI_Send(&res2, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    .
    .
    .

    else if(rank==7){
        res7=callOMP(atoi(argv[1])-14,rank);
         MPI_Send(&res7, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
    }

    err = MPI_Finalize();
    return 0;
}
int callOMP(int number, int rank){
    int res;
    #pragma omp parallel num_threads(128)
    {
        #pragma omp single
        {
            res = parallel_fibonacci(number);
        }
```

```
    }
    printf("We are in rank%d and we are computing fibo(%d) \n result:%d
        \n",rank,number,res);
    return res;
}
```

The right way for compiling this program is the following:

```
mpiicc -qopenmp -xHost omp_mpi.c -o omp_mpi.out
```

in which we use the MPI compiler but also we activate the OpenMP directives (and the largest register's size available). Inside each host we are using 128 threads which were empirically good options in OMP version 3.

Instead, to execute it we launch the command:

```
time mpiexec -hostfile machinefile.txt -perhost 1 -np 8 ./omp_mpi.out 45
```

where we specifiy the file that contains the names of the 8 machines available and also we set the number of processes that we want to use, 8 in total, 1 for each machine. The time option before the command is used for taking the real time execution of the entire program.

2.3.2 OMP + MPI Version 2

A more detailed look at the computation procedure will reveal that we are not assigning the unique number of fibo for each host. As an example, in the previous version we computed fib(41) three times in three different hosts.

```
.
.
.
We are in rank2 and we are computing fibo(41)
result:165580141
```

```
.
.
.
We are in rank4 and we are computing fibo(41)
result:165580141
We are in rank1 and we are computing fibo(41)
result:165580141
.
.
.
```

What we can do here is to assign each host a unique number, so we are avoiding extra computations. In order to achieve this goal, we need to explore the tree a bit more to find some unique numbers to hosts. We will still go with the 4th depth but we will explore the tree more on fib(n-6). Two children of fib(n-6) are fib(n-7) and fib(n-8) which don't appear in 4th depth. We will continue fib(n-8) and find it's children on it's forth depth which are , fib(n-11), 3x fib(n-12), 3x fib(n-13) and fib(n-14). Since fib(n-12) appears three times, in the final summation in rank 0 we need to multiply it by three. And the same rule is true for other numbers. So in this version we have the following summation:

```
final_res= res0+ 3*res1+ 3*res2+ res3+ res4+ 3*res5+ 3*res6+ res7;
```

The point here is we are trying to avoid extra computation using assigning an efficient number to each host. Here, in each host we are computing smaller number of fibo and we still get the same or less amount of execution time. This implies that for the bigger numbers of fibo, this method will be efficient in a sense that previous version was computing the same number several times. Multiplying the results based on the number of times they appear in the tree will avoid computation for each host.

Using only MPI, we could only compute fibo till 12th number. Using this approach, we can compute fib(46) . The execution time decreased dramatically. From minutes we reached some seconds of computation.

Fibonacci Input Number 46	
Approach	Execution time
Basic	58.598 seconds
MPI + OpenMP	3.528 seconds

The table above shows the fact that for high number of Fibonacci's input the execution time is not so big respect at the basic approach, in fact the image below displays that the curve of the MPI + OMP approach's time remains almost constant respect to the basic one, in which increasing the number of Fibonacci, the time increases too.

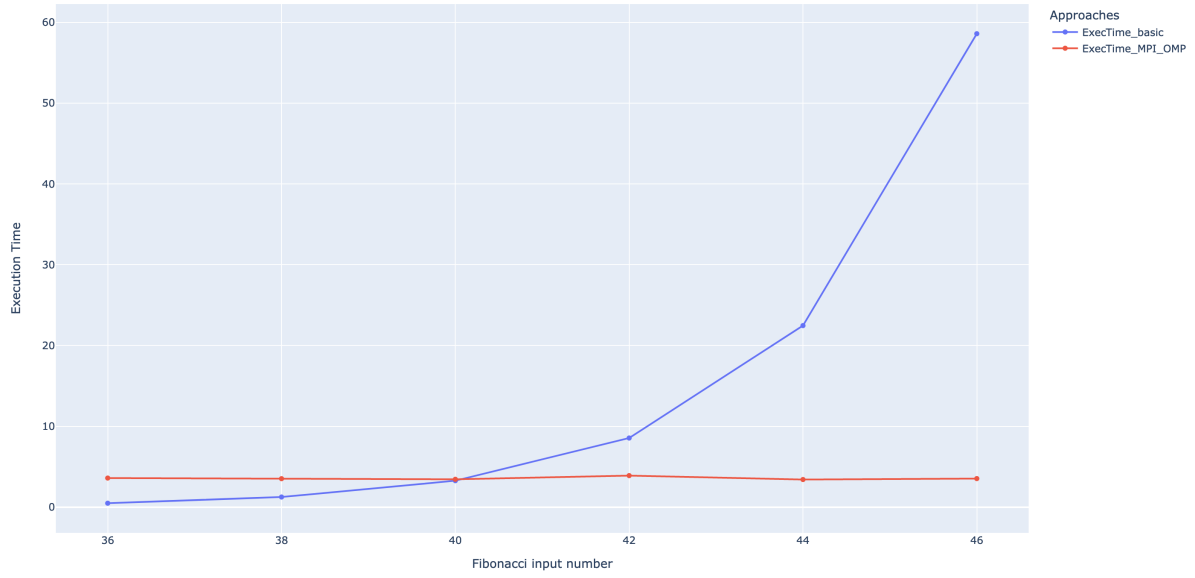

Figure 2.3: Difference between basic and MPI + OMP recursive Fibonacci

2.4 Fibonacci with CUDA

The last trial for parallelizing the fibonacci recursive function is the one with the usage of Gpu computation thanks to CUDA platform. In this case we don't have flags or options that could be used during compilation or execution for an optimization. The right command to compile a CUDA file (notice the fact that a C program of CUDA has the extension .cu instead of .c) is:

```
$ nvcc -rdc=true -o fibonacciCuda_par1.out fibonacciCuda_par1.cu
```

where:

- The nvcc is the NVIDIA CUDA Compiler used to hide the intricate details of CUDA compilation from developers.
- The -rdc=true option is the only used flag, due to the fact that CUDA dynamic parallelism requires separate compilation and linking.

2.4.1 Recursive CUDA approach

A possible implementation of recursive CUDA parallel Fibonacci algorithm could be the following:

```
int main(int argc, char* argv[]){
    printf("The input number for fibonacci is: %s\n", argv[1]);
    int number = atoi(argv[1]);
    printf("The fibonacci of %s is %d\n", argv[1], calc_CUDA_Fibonacci(number));
}
int initial_fibonacci_run(int fib, int currentDepth, int targetDepth){
    if (fib <= 1)
        return fib;
    if (currentDepth < targetDepth)
        return initial_fibonacci_run(fib - 1, currentDepth++, targetDepth) +
            initial_fibonacci_run(fib - 2, currentDepth++, targetDepth);
```

```
    int *d_fib, *d_result;
    int size = sizeof(int);
    cudaMalloc((void**)&d_fib, size);
    cudaMalloc((void**)&d_result, size);
    cudaMemcpy(d_fib, &fib, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_result, 0, size, cudaMemcpyHostToDevice);
    int threadsPerBlock = 64;
    int blocksPerGrid = (fib + threadsPerBlock - 1) / threadsPerBlock;
    // call the global function
    fibonacciParent<<<blocksPerGrid, threadsPerBlock>>>(d_fib, d_result);
    int result = 0;
    cudaMemcpy(&result, d_result, size, cudaMemcpyDeviceToHost);
    cudaFree(d_fib);
    cudaFree(d_result);
    return result;
}
__global__ void fibonacciParent(int* number, int* res){
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if(i < *number){
        fibonacci(*number, res); // call the device function
    }
    return;
}
__device__ void fibonacci(int number, int* res){
    if (number <= 1){
        *res += number;
        return;
    }
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int x, y;
    fibonacci(number - 1, res); //recursion on device
```

```
    fibonacci(number - 2, res);
}
```

Without going too much into the complex implementation details, in the code above the main part are the definition of the global (kernel) and device function. The global function can be called only inside "normal" function of the program (like the program main) and they cannot be called from other global function or from device function. The device function, instead, cannot be called from "normal" function, but only from, or a global function, or another device function. This digression for explaining the fact that to implement recursion in our program the first step was to call the global function inside the intialFibonacciRun(), and, only after doing this, we can trigger the execution of the recursive Fibonacci, that is, in fact, a device function, since it calls itself many times, and we have just said that only device function can do this. Just to clarify, the function named intialFibonacciRun() is needed for: allocating and de-allocating the GPU memory of the used variables (respectvely with cudaMalloc and cudaFree); sending these variables from the CPU RAM to the GPU RAM and, after the computation, to send back the result from GPU to CPU (with cudaMemcpy). We can notice also the initial if clause, used to compute a classical fibonacci recursion when the current depth of recursion is less than the target one.

Fibonacci Input Number 30	
Approach	Execution time
Basic	0.005608 seconds
CUDA	9.942477 seconds

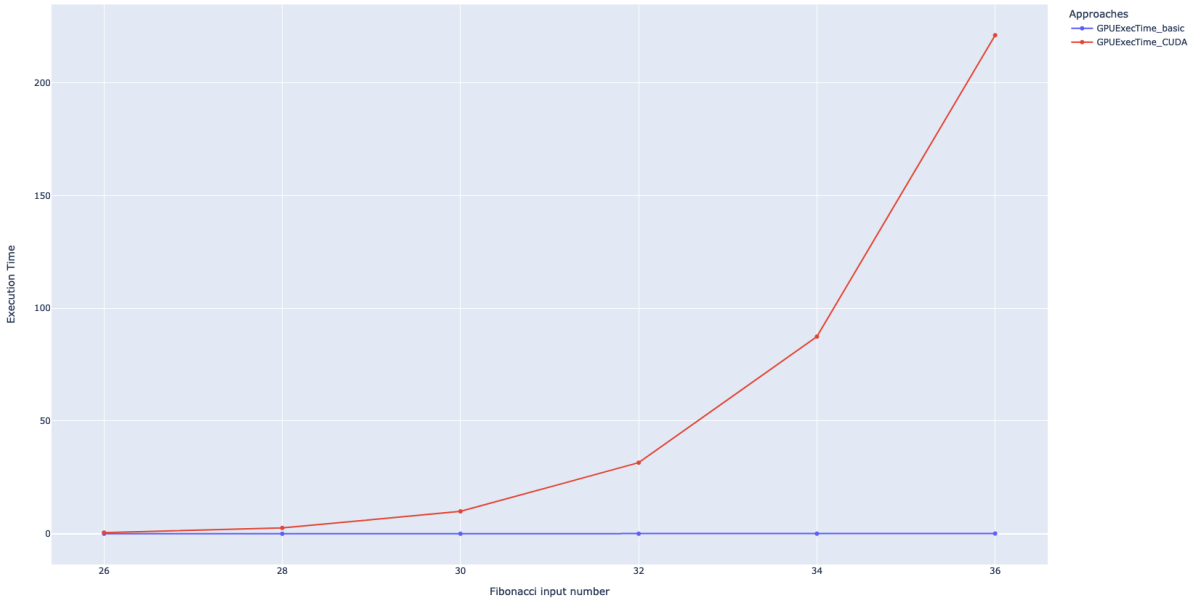

Figure 2.4: Difference between basic and CUDA approach of Fibonacci recursion

The first thing that we have to say is that in general CUDA is not suitable for recursion, as the tabular and image results above show. This behavior is due to the fact that, with recursion, the compiler cannot accurately determine the required stack size as the depth of the recursion which is unknown at compile time and your kernel may experience stack overflow at runtime. Note that recursion is supported, at least on newer GPU architectures, but you may have to manually adjust the stack size configured via the driver, as it cannot determine total stack usage at compile time. So, if your algorithm involves a lot of recursions, then support or not, it is not designed for GPUs, either redesign your algorthims or exploit in a good way the CPU will be better then do recurisons on GPUs. In the end it is possible to say that it may still be a good idea to remove the recursion for performance reasons when we usa CUDA parallel computation.

2.4.2 Non Recursive CUDA Approach

Since with recursion using CUDA is not the best solution, for having something concrete to report, we present another kind of approach for finding the result of fi-

bonacci, always with a GPU computation, but for this configuration we don't have used a recursive implementation anymore.

```
__global__ void Fibonacci(double *ga, double *gb, double sqrt_five, double
    phi1, double phi2, int N)
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
    {
        gb[i] = (pow((double)phi1, ga[i]) - pow((double)phi2, ga[i])) /
            sqrt_five;
    }
}
int main(int argc, char *argv[])
{
     if(argc != 2){
        printf("Error: You have to insert the input number for fibonacci!\n");
        return 0;
    }
    int N = 1 + atoi(argv[1]);
    double ha[N], hb[N];// Host variable
    double *ga,*gb; //For GPU use
    double sqrt_five, phi1, phi2;
    sqrt_five = sqrt(5);
    phi1 = (sqrt_five + 1) / 2; //positive golden ratio
    phi2 = (sqrt_five - 1) / 2;// negative golden ratio
    // Initialize array on CPU
    for (int i = 0; i<N; i++)
    {
        ha[i] = i;
    }
    //Allocate memory on GPU
```

```
    cudaMalloc((void**)&ga, N*sizeof(double));
    cudaMalloc((void**)&gb, N*sizeof(double));
    //Copy array from CPU to GPU
    cudaMemcpy(ga, ha, N*sizeof(double), cudaMemcpyHostToDevice);
    //Kernel launching
    Fibonacci <<<2, 128 >>>(ga, gb, sqrt_five, phi1, phi2, N);
    //Copy results from GPU to CPU
    cudaMemcpy(ha, gb, N*sizeof(double), cudaMemcpyDeviceToHost);
    printf("Result of Fibonacci GPU of %d is: %lf\n", N-1, ha[N-1]);
    return 0;
}
```

The new method, showed in the code above, is based on the golden ratio of Fibonacci, a constant number equal to 1.618033988749895, used with both positive and negative value (phi1 and phi2). As before, at first it is needed to allocate the necessary memory in the CPU and GPU. Then, we copy the initialized array (ha) from CPU to GPU to use it (ga) inside the kernel Fibonacci function. After the execution of this function, we copy back the array to CPU (from ga to ha). As we can see, the main difference is inside the global Fibonacci function that uses the golden ratio mentioned before in a specific formula, called Binet's formula, for calculating the Fibonacci sequence without using, at each step, the previous two numbers of the sequence. Of course, since we are in a kernel function with multi-threading access, we have to remove a for loop with a simple if clause in which we specify that we compute the formula only if the id (the position of the thread in the CUDA grid) is less than the size N (the number for which we want to calculate the Fibonacci's result). With this new approach we obtain a result, showed in the table and image below, in terms of execution time, that is very wonderful with respect to the recursive one, as a proof that usually recursion in CUDA is not the best solution. The time seems to remain always constant since does not depend really on the size N but depends on a constant "magic" number (the golden ratio) used inside the Binet's formula.

Fibonacci Input Number 46	
Approach	Execution time
Basic	58.598 seconds
Non Recursive CUDA	0.000051 seconds

We should say that we are able to compute a completely correct result of Fibonacci sequence until an input value of 71 (for the next numbers we obtain a result that differs from the correct one for few numerics at the end of the number).

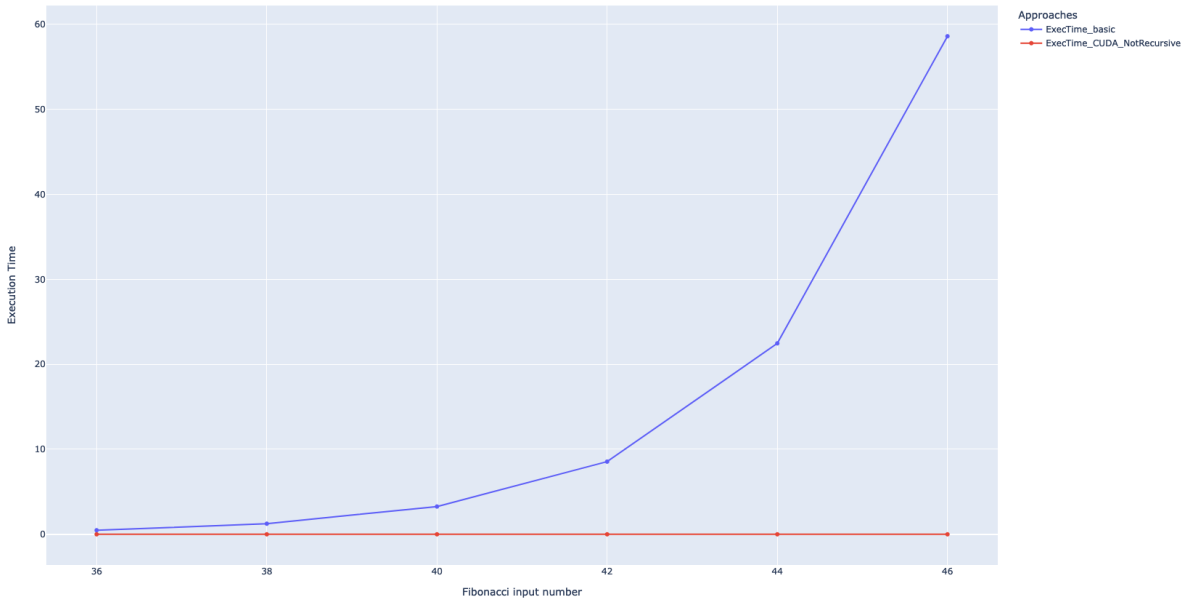

Figure 2.5: Difference between basic and Non Recursive CUDA approach of Fibonacci recursion

In this picture we notice that, as we said previously, the execution time's growth curve of the non recursive CUDA approach remains constant and so it is better than the basic one.

Chapter 3

Conclusion and Future Works

If we try to summarize everything, we are able to say now that a basic approach of recursion still be a good choice because some parallel interfaces are tailored more for loop than for a recursive approach, but, as this report and project shows, it is possible to obtain good results. For our case, in which the recursion of Fibonacci produces correct results only with a maximum input value of 46, the best solution seems to be the one with the usage the OpenMP (version 3) multithreading paradigm. Another good solution is the one relative to the integration approach between MPI and OpenMP. Theoretically, this configuration should works very good (in terms of execution time) for large number of input, while using only the OpenMP version is more suitable to compute Fibonacci recursion for small input values. The MPI approach, for now, is almost unrealizable, because recursion doesn't fit the MPI's paradigm of multi-processes. Also the CUDA method, if we don't consider the non recursive approach, doesn't work well. The results are correct only for small number of input values and also the execution time is, in general, very high.

Fibonacci Input Number 42	
Approach	Execution time
Basic	8.549 seconds
OpenMP V3	0.495 seconds
MPI + OpenMP	3.895 seconds
CUDA	221.086 seconds

The table above and image below demonstrate the differences among the various implemented recursive configurations, starting from the basic approach to the CUDA one. We can see, as we said before, that the only two good approaches are the OpenMP and the MPI + OpenMP, that, taking into account that our configuration works only up to Fibonacci of 46, are even better than the basic method for computing the recursive Fibonacci function.

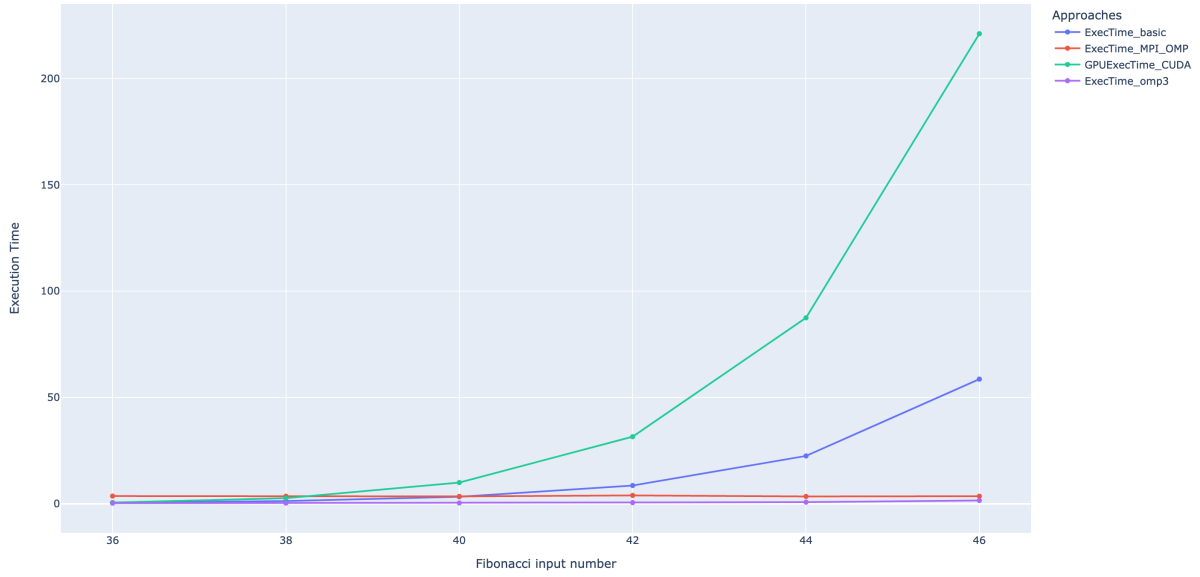

Figure 3.1: Summary of differences between all the approaches of recursive Fibonacci

We will also make some suggestions here to improve the execution time. First is the fact that, in order to compute bigger numbers, we can use long integer variables. If we do so, it is highly likely that the MPI-OMP version will perform better than the

OMP-v3. Another possibility in MPI-OMP is to explore the Fibonacci's tree to find bigger and unique numbers which theoretically can be more efficient than ours. We also encourage the readers to test the following approaches:

- Trying to implement the sequential computation of Fibonacci in each host in MPI-OMP version. Instead of using OMP, use sequential version.
- For numbers ,like fibo(46), try to allocate 3 or 4 hosts to avoid the overhead caused by communication in our MPI-OMP integration. Hence, in MPI-OMP first version, we could only divide the task among 4 hosts, instead of 8 hosts, in which each should compute the following numbers: fib(n-3), fib(n-4), fib(n-5) and fib(n-6). Finally, we could multiply the fib(n-4) and fib(n-5) by 3 which is the number of their appearance in the tree and sum up the results. Using this case, the number of communication among hosts will decrease by 2 times, hence we will avoid extra communications for the smaller numbers of Fibonacci's sequence.
- Another possibility is about OMP-v3 and it's integration with MPI. In both cases, for the numbers smaller than the threshold, we are computing the sequential computation of fibo which is recursive. Here we could apply sequential, but with the golden ratio. Certainly, computing with for loop, will be faster than recursive one.

Thank You for the attention

GitHub Repo:

`https://github.com/edo-pasto/HPC_Project.git`

Bibliography

[1] Tutorials Tonight. What is Recursion in Programming. https://www.tutorialstonight.com/what-is-recursion-in-programming

[2] HowToGeek. What is Recursion in Programming and how do you use it. https://www.howtogeek.com/devops/what-is-recursion-in-programming-and-how-do-you-use-it/

[3] AfterAcademy. What is Recursion in Programming. https://afteracademy.com/blog/what-is-recursion-in-programming/

[4] Technical-qa.com. What is MPI parallel computing. https://technicqa.com/what-is-mpi-parallel-computing/

[5] InfoWorld. What is CUDA? parallel programming for GPU. https://www.infoworld.com/article/3299703/what-is-cuda-parallel-programming-for-gpus.html

[6] Wikipedia. Message Passing Interface. https://en.wikipedia.org/wiki/Message_Passing_Interface

[7] Wikipedia. OpenMP. https://en.wikipedia.org/wiki/OpenMP

[8] Wikipedia. Intel C++ Compiler. https://en.wikipedia.org/wiki/Intel_C%2B%2B_Compiler

[9] Hpc Wiki. Introduction to Tasking. https://hpc-wiki.info/mediawiki/hpc_images/9/91/Hpc.nrw_05_Introduction-Tasking.pdf

[10] GitHub. Fibonacci GPU. https://github.com/branstanley/Fibonacci_GPU

[11] NVIDIA Developers Forum. Fibonaci sequence in CUDA? https://forums.developer.nvidia.com/t/fibonacci-sequence-in-cuda/10309/4

[12] PagodeWiki. mestrado:mpi2-fibonacci http://www.pagode.tuxfamily.org/doku.php?id=mestrado:mpi2-fibonacci