



UNIVERSITY
OF TRENTO

Department of Information Engineering and Computer Science

HIGH PERFORMANCE COMPUTING FOR DATA SCIENCE

2021/07/13

PROJECT REPORT

Carlo Corradini

223811

Massimiliano Fronza

220234

Academic year 2020/2021

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 1 |
| 1.1 | Source Code | 2 |
| 1.2 | Libraries | 2 |
| 1.3 | Build | 2 |
| 1.3.1 | Cmake | 3 |
| 1.3.2 | Procedure | 3 |
| 1.3.3 | CMake flags | 3 |
| 1.4 | Compile | 4 |
| 1.4.1 | GNU Make | 4 |
| 1.4.2 | Procedure | 4 |
| 1.5 | Execute | 4 |
| 1.5.1 | PBS | 6 |
| 2 | Architecture | 7 |
| 2.1 | Master & Worker model | 7 |
| 2.2 | Communication | 8 |
| 2.2.1 | Message | 9 |
| 2.3 | Input | 10 |
| 2.4 | Output | 13 |
| 3 | Optimization choices | 17 |
| 3.1 | OpenMP | 17 |
| 3.1.1 | Is parallelization always a good choice? | 17 |
| 3.1.2 | Terminate all threads if something goes wrong | 17 |
| 3.1.3 | Parallelization of a double for loop with collapse | 18 |
| 4 | Elaboration | 19 |
| 4.1 | Java and C comparison | 19 |
| 4.2 | Fluids visualization | 19 |
| 4.2.1 | Simulations | 20 |
| 5 | Benchmarks | 21 |
| 5.1 | PBS scripts and features (OpenMP & IO) | 21 |
| 5.2 | OpenMP schedule | 23 |
| | Bibliography | 25 |
| A | Benchmarks in depth | 26 |

1 Introduction

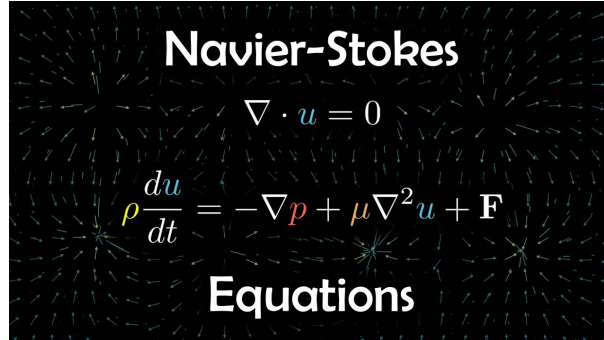


Figure 1.1: Navier-Stokes Equations

For this project, we decided to start from an already existing open source project implementing the Navier-Stokes equations¹, a set of partial differential equations describing the motion of viscous fluid substances.

Since the Navier-Stokes equations are really hard to understand and also to solve we took some knowledge from two YouTube videos where it's shown and explained "for the public":

- <https://youtu.be/Ra7aQlenTb8>
- <https://youtu.be/ERBVFcutl3M>

The original project where we took the inspiration from can be found at:

<https://github.com/deltabrot/fluid-dynamics>.

Our idea was to start from the Java simulation algorithm of the Navier-Stokes equations in the repository, re-write it in C and then turn the obtained sequential code into a parallel one, using the MPI and OpenMP libraries for high-performance computing.

The Java application presented itself as a GUI showing a grid where it was possible to click to increase the density of the fluid in a certain region and to apply forces that could make it react in the space, decreasing its density cell by cell (tick by tick).

The simulation was lead by ticks, units of time that regulated the progression of particles in the grid. The number of ticks, as well as their duration in seconds was set to 0.01 seconds for our tests. The greater the duration value per tick, the higher the precision.

After re-writing the Java part in C, we wrote a Jupyter Notebook in Python to see if they had the same behavior. The automatic time tick that was called every 10 ms in the Java scenario was also removed in favor of an interactive one, and we added the possibility to statically perform the placement of fluids in the space, as well as the forces acting on them and the amount of ticks that we wanted to perform. This was intended to make a fair 1 to 1 comparison between the two versions of the program.

We modified the original Java application and our C version to generate output files and compared them for verification in the notebook.

The next step was to entirely focus on the C program, by analyzing its computationally slowest regions for a fine-grained optimization based on OpenMP. This part, plus the MPI integration, is discussed in the *Architecture* and *Optimization choices* chapters.

In the *Elaboration* part, we've reported more details about the output comparisons with the Java application, the further elaboration of the C program results, and the gifs visualizing the fluids moving around the grid.

¹https://en.wikipedia.org/wiki/Navier-Stokes_equations

In the last chapter, called *Benchmarking*, we analyzed the performances of our parallel code in comparison with its serial version.

1.1 Source Code

The source code of the entire project is available at the following URL:

https://github.com/carlocorradini/hpc4ds/tree/main/navier_strokes

Note that in the same repository there are other HPC related projects.

1.2 Libraries

In the project we used several libraries that helped us reduce the overall amount of work and complexity.

Note that **OpenMP** (`omp.h`) and **MPI** (`mpi.h`) libraries are not listed since they are mandatory.

The following is the list of libraries used:

- **cJSON**

Website: <https://github.com/DaveGamble/cJSON>

Ultralightweight JSON parser in ANSI C. cJSON aims to be the dumbest possible parser that you can get your job done with. It's a single file of C, and a single header file. As a library, cJSON exists to take away as much legwork as it can, but not get in your way[2]. This library is used to parse and generate JSON string.

A string representation of a JSON simulation, read from `simulations.json` file, is sent to each worker node by the master node. The received string is then *parsed* to extrapolates the data. When the simulation has been completed the resulting data are written to a JSON.

We could have chosen another data-interchange format but JSON is what we are more familiar with. JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate[4].

- **argparse**

Website: <https://github.com/cofyc/argparse>

A command line arguments parsing library in C (compatible with C++). inspired by `parse-options.c` (git) and python's `argparse` module. Arguments parsing is common task in cli program, but traditional getopt libraries are not easy to use. This library provides high-level arguments parsing solutions. The program defines what arguments it requires, and `argparse` will figure out how to parse those out of `argc` and `argv`, it also automatically generates help and usage messages and issues errors when users give the program invalid arguments[1].

Heavily used when the executable is launched to check if the arguments (`const char **argv`) are valid and/or defined (see section 1.5 for more information).

- **log.c**

Website: <https://github.com/rxi/log.c>

A simple logging library implemented in C99[6].

A super-useful yet super-simple library that helps to better understand what the program is current doing and it's overall state simply by printing *logging* information.

Note that this library has been imported directly copying `log.c` and `log.h` files without the needs of *CMake*.

Note that linking libraries to an executable is not easy, but thanks to **CMake** (see next chapter) this procedure is much (much) easier, portable and maintainable.

1.3 Build

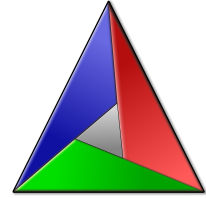
To build the executable we used Cmake, a standard de-facto software that makes easier the overall procedures. CMake is used to define the project structure (it interfaces with various

IDEs) and to gather all the information needed during the compilation phase, such as dependencies/libraries to include and Debug or Release mode, producing a directory hierarchies and a Makefile.

1.3.1 Cmake

CMake is cross-platform free and open-source software for build automation, testing, packaging, and installation of software by using a compiler-independent method. CMake is not a build system but rather it generates another system's build files. It supports directory hierarchies and applications that depend on multiple libraries. It is used in conjunction with native build environments such as Make, Qt Creator, Ninja, Android Studio, Apple's Xcode, and Microsoft Visual Studio. It has minimal dependencies, requiring only a C++ compiler on its own build system[7].

Website: <https://cmake.org>



1.3.2 Procedure

The building phase is separated by three different steps:

1. Create build directory

In the project root, create a *build* directory that will be used to store the generated files.

```
$ mkdir build
```

2. Change directory context

Change the current working directory to the newly created *build* directory.

```
$ cd build
```

3. Launch cmake

Launch **cmake** and build the project in **Release** mode to optimize the executable.

```
$ cmake -DCMAKE_BUILD_TYPE=Release ..
```

Note that in the third step we've passed a flag to CMake to configure the building phase. In the following section are described two flags that are the most important for our needs.

1.3.3 CMake flags

- **-DNO_OPEN_MP=<On | Off>**

Defines if the executable must be compiled with **OpenMP** or not.

Note that by default **OpenMP** support is enabled.

Values:

- **On**
Disable **OpenMP**.

- **Off**
Enable **OpenMP**.

Note that **Off** has the same behavior as omitting the flag.

- **-DCMAKE_BUILD_TYPE=<Release | Debug>**

Set the build type.

Note that the default value is **Debug**.

Values:

- **Release**
Enable compiler optimization and don't include debug features.
Note that the compilation time is greater but the overall performance is better.
- **Debug**
Include debug features and don't enable any compiler optimization.
Note that the compilation time is lower but the overall performance is worse.

1.4 Compile

To compile the executable we used GNU Make, another standard de-facto software to make the overall procedures easier to perform. GNU Make is used to build the final executable running various *recipes*, generated by CMake, defined in the Makefile. Launching **make** (without any recipes specified) will run the default recipe, that is defined by default as compiling the executable using the **mpicc** compiler.

1.4.1 GNU Make



Make is a build automation tool that automatically builds executable programs and libraries from source code by reading files called Makefiles which specify how to derive the target program. Though integrated development environments and language-specific compiler features can also be used to manage a build process, Make remains widely used, especially in Unix and Unix-like operating systems. Besides building programs, Make can be used to manage any project where some files must be updated automatically from others whenever the others change.

There are now a number of dependency-tracking build utilities, but Make is one of the most widespread, primarily due to its inclusion in Unix[8].

Website: <https://www.gnu.org/software/make>

1.4.2 Procedure

The compilation phase is very straightforward, since it requires a single command to start it. Note that the following command must be executed in the previously **build** directory as current working directory.

```
$ make
```

After the compilation has terminated an executable file called **navierstokes** has been generated.

1.5 Execute

To *execute*, the program **mpiexec** (or equivalent) is required.

The program requires at least 2 processes since is based on a master-worker architecture (see section 2.1 for more information).

The executable accepts several arguments. The majority of them are optional but **--simulations** and **--results** are required since they are used to configure the Navier Stokes simulations data computation and results.

The following is a list of all supported arguments:

- **--help**
Show help message and exit.
- **--simulations=<str>**
Required
Path to the JSON simulations file.
The value must be a *correct* JSON file with read permission otherwise the executable will exit with a failure code.
- **--results=<str>**
Required
Path to a folder used to save all JSON simulation results.
The value must be a folder with write permission otherwise the executable will exit with a failure code.
- **--loglevel=<str>**
Default: INFO
Set the minimum logger level.

A logging level is a way of classifying the entries in the log file in terms of urgency. Classifying helps filter log files during search and helps control the amount of information. Values:

1. **FATAL**
Designates very severe error events that will presumably lead the application to abort.
2. **ERROR**
Designates error events that might still allow the application to continue running.
3. **WARN**
Designates potentially harmful situations.
4. **INFO**
Designates informational messages that highlight the progress of the application at coarse-grained level.
5. **DEBUG**
Designates fine-grained informational events that are most useful to debug an application.
6. **TRACE**
Designates finer-grained informational events than the **DEBUG**.

Note that setting a high logger level will cause to ignore lower levels. On the contrary, setting a low logger level will cause to *log* a lot of information that can be useless.

- **--colors**

Default: **disabled**

Enable logger output with colors. Very useful if the **stdout** is printed directly to a terminal and not to a file.

Note that this option simply add the corresponding ANSI color code before the log string.

Below is an example on how to test **navierstokes** executable directly using **mpiexec**. Note that the logger level is set to **DEBUG** and the output is with colors enabled.

```
$ mpiexec -np 2 ./navierstokes --simulations=./simulations.json
--results=./results --loglevel=DEBUG --colors
```

Launch the program with the **--help** argument and some useful information will be showed. Note that the program can be executed with the **help** flag directly without the needs of **mpiexec** or equivalent.

```
$ ./navierstokes --help
Usage: mpiexec -np 2 ./navierstokes --simulations=./simulations.json
--results=./results
or: mpiexec -np 2 ./navierstokes --simulations=./simulations.json
--results=./results --colors
or: mpiexec -np 2 ./navierstokes --simulations=./simulations.json
--results=./results --loglevel=DEBUG
or: mpiexec -np 2 ./navierstokes --simulations=./simulations.json
--results=./results --colors --loglevel=DEBUG

Navier Stokes simulations in high performance computing environment
v.0.0.1

Options:
-h, --help show this help message and exit
--simulations=<str> Path to JSON simulations file
--results=<str> Path to folder used to save JSON simulation results
--loglevel=<str> Logger level. Default to 'INFO'
--colors Enable logger output with colors
```

1.5.1 PBS

The program is designed to be executed on the unitn cluster using the PBS scheduler. Portable Batch System (or simply PBS) is the name of the computer software that performs job scheduling. Its primary task is to allocate computational tasks, i.e., batch jobs, among the available computing resources. It is often used in conjunction with UNIX cluster environments[9].

In the folder `hpc/pbs` there are several *PBS shell* scripts that can be used to test the program in different environments. The default script to be used is named `navierstokes_multi_node_multi_worker.sh` and requires 8 nodes and 8 processes (one for each node).

Below is an example on how to test `navierstokes` executable on the unitn cluster. Note that the current working directory must be `hpc/pbs`.

```
$ qsub navierstokes_multi_node_multi_worker.sh
```

For more information see section 5.1.

2 Architecture

As for the solution design, we thought about having a list of different simulations organized in a JSON file collecting their parameters, such as fluid viscosity, diffusion and the grid's dimensions. Our original idea was to use MPI to manage different sections of a single world, but due to serious data dependencies, we had to change the approach. Each cell, indeed, needed data from all the surrounding cells at the same time, not just from some direction or at different time steps; for this reason, we decided to go for a fine-grained parallelization with OpenMP inside the single simulations, and an MPI-based system to collect the results of each world's computation.

MPI is used primarily for scheduling the simulations across workers and keeping the communication active between the master and the worker.

OpenMP, on the other hand, is used to parallelize the workload among the available processor cores on the worker node to increase the overall computational time.

Combining both MPI and OpenMP, led to a hybrid distributed-shared memory architecture that exploited all the resources and technologies available in the cluster.

In the next sections, all the architectural aspects will be discussed and shown in-depth to give a more detailed view & knowledge as possible.

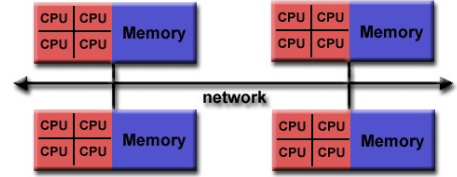


Figure 2.1: Hybrid distributed-shared memory architecture model

2.1 Master & Worker model

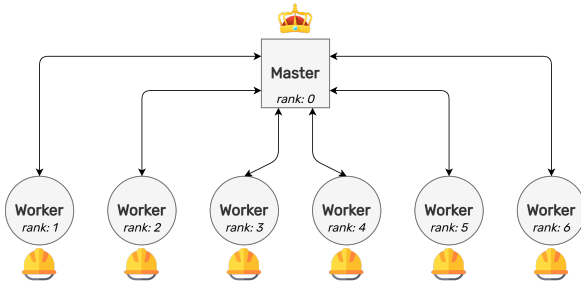


Figure 2.2: Master & Worker communication model

Master/Slave or Master/Worker is a model of asymmetric communication or control where one device or process (the "master") controls one or more other devices or processes (the "workers") and serves as their communication hub. In some systems, a master is selected from a group of eligible devices, with the other devices acting in the role of slaves[5].

Due to its intrinsic nature of the model, the overall architecture needs at least 2 processes to work properly, one for a master and one for a worker. If not, the program will log an error message and terminate (abort) via

`MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE).`

Note that all the communications are made directly from `master` to `worker`. No worker can communicate with any other worker.

The master node is always the process with MPI rank 0. This choice has been made for two main reasons. Firstly, rank 0 is always associated with a work scheduler so it's more familiar for other HPC programmers to understand which is the master. Secondly, in a HPC environment the rank 0 process is commonly the only process with "full" permissions such as reading a file or stdin interactive mode. This "role" is crucial in our project since all the simulations to compute are stored in a JSON file that must be read only by the master node.

In opposite, each worker can have a rank that is in the range of 1 (inclusive) to `MPI_Comm_size - 1` (inclusive). The program can also be executed with more workers than the available number of simulations in the file. In these cases, however, a warning message will be printed in the terminal log, informing that some workers are useless and will not be involved in the processing. Logically, the number of workers can also be less than the number of simulations, always respecting the presence of at least one; when a worker finishes its simulation a new one (if available) will be scheduled for it to process. This aspect is better described in the next section.

In conclusion, the overall number of workers must be correctly balanced between the available resources and the total number of simulations. It's a tradeoff between time and resource usage: if the number of workers is too low, all resources will be used but the computation time will be higher.

If the number of workers are too high, maybe not all the requested resources will be used, but the computation time will be certainly lower.

2.2 Communication

The communication between the master and the worker nodes is asynchronous.

Initially, the workers are listening for a message and the master has an array of all available workers (and their status) and the list of simulations to compute.

The master must schedule all the simulations across all the available workers. If all the workers are *processing* and some simulations are still missing to elaborate, the master has to wait until one of them becomes free, scheduling one of the missing simulation to it. This process is repeated until all the simulations have been computed. After which, a **terminate** message is sent to all workers causing all the processes to shutting down.

Following is a more in depth description on the scheduling procedure:

The master has to schedule a simulation **X** to a worker node **Y**. Note that the worker node **Y** must be not working. The master sends a message to the worker node **Y** telling the simulation identifier (`simulation_id`) and to continue to work (**terminate** set to *false*). The worker now knows that it must continue to work and little by little a simulation (JSON string) is going to be sent to him. The master sends the simulation data, obtained from the input file (`simulations` array), as a JSON string. The worker parse it and start the elaboration. The master can continue to schedule new jobs or wait for a new worker to be available. Once the worker **Y** has finished computing the simulation **X** it informs the master that the job has been successfully completed sending a message containing the simulation identifier of the just yet computed simulation. If there are more simulations to compute, the master schedule another simulation to the worker **Y** and continue working. Otherwise, it sends a terminate message (**terminate** set to *true*) and the worker node shuts down.

In the next page is shown a figure that shows an example of communication between one master node and two-worker nodes for computing four simulations to compute.

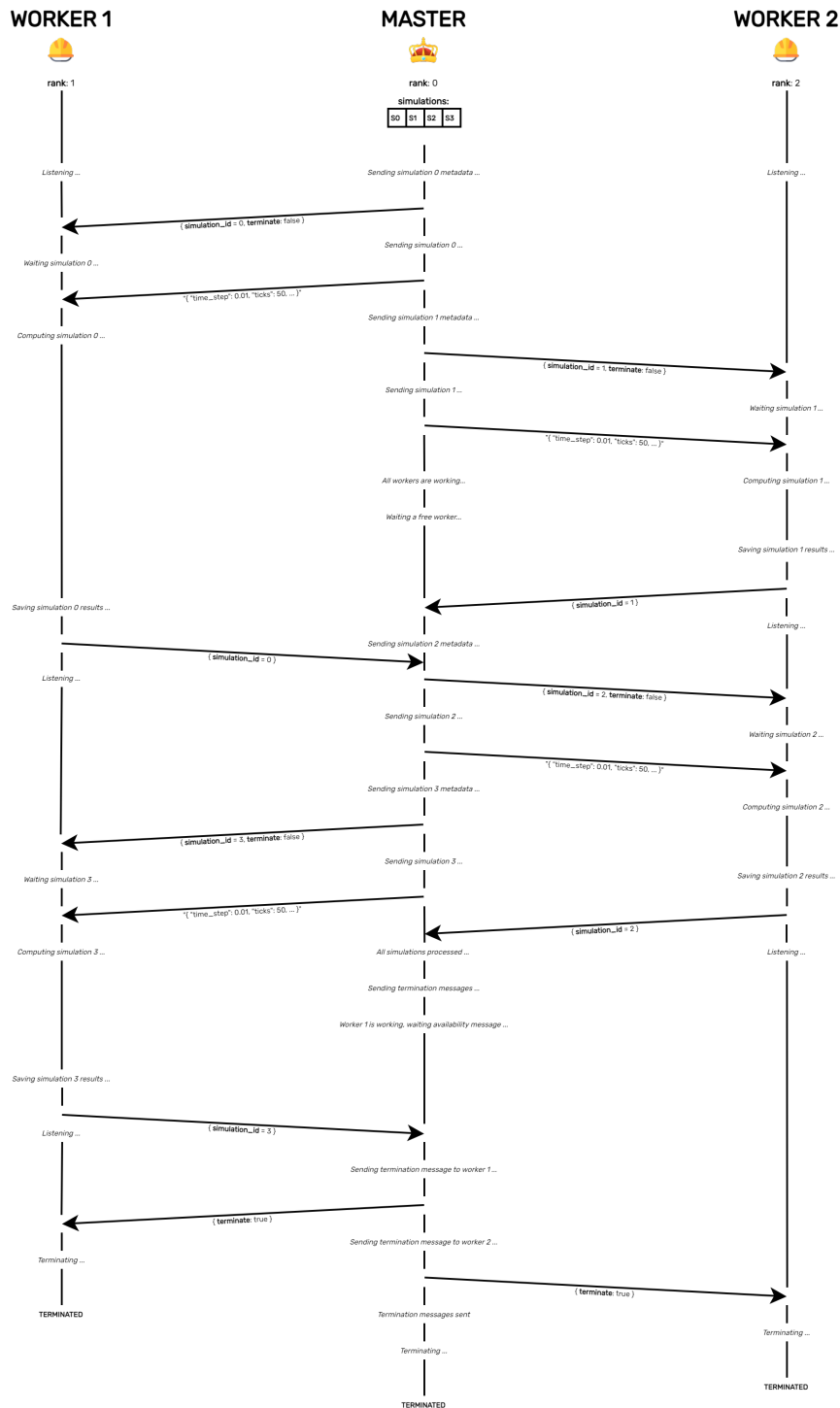


Figure 2.3: Communication phases between **master** and **workers** [https://imgur.com/a/CLeA7r9]

2.2.1 Message

To send a message between two processes we created a custom MPI data type called `com_message_t`. Its purpose is to instruct the worker on its behavior: keep working on a new simulation, informing the master that the worker has finished the elaboration or shutting down the worker. The struct is composed by two fields:

- **terminate**

Type: bool

Determinates if the current worker node must terminate working and shut down. This applies only if the value is **true**.

- **simulation_id**
Type: unsigned long int
Stores the simulation identifier.

To *build* the message data type, a function has been defined. It's called `com_message_MPI_datatype(...)` and accepts a pointer to a `MPI_Datatype`. The pointer is used to store the struct data type definition in the caller context.

Unfortunately, the process to define a custom `MPI_Datatype` is not very easy and the overall procedure is particular, since we have to compute all the addresses offsets of each field in the struct.

Following, the function signature and body:

```

1 void com_message_MPI_datatype(MPI_Datatype *message_type) {
2     if (message_type == NULL) return;
3
4     // Number of items
5     enum { n_items = 2 };
6
7     // How many elements for each item
8     int block_lengths[n_items] = {1, 1};
9
10    // Type of each item
11    MPI_Datatype types[n_items] = {MPI_C_BOOL, MPI_UINT64_T};
12
13    // Calculate offsets
14    MPI_Aint offsets[n_items];
15    struct com_message_t m;
16    MPI_Aint base_address;
17    MPI_Get_address(&m, &base_address);
18    MPI_Get_address(&m.terminate, &offsets[0]);
19    MPI_Get_address(&m.simulation_id, &offsets[1]);
20    offsets[0] = MPI_Aint_diff(offsets[0], base_address);
21    offsets[1] = MPI_Aint_diff(offsets[1], base_address);
22
23    // Create the struct type
24    MPI_Type_create_struct(n_items, block_lengths, offsets, types, message_type);
25    MPI_Type_commit(message_type);
26 }

```

2.3 Input

All the simulations to compute are saved in a JSON file. Initially, the master has to read the file given the path as an argument (`--simulations`). After the file has been correctly read and parsed, the master starts to schedule the computation across workers.

The JSON input file has a strict structure that must be respected, otherwise it cannot be parsed and the execution is aborted.

Each simulation object is composed by the following fields:

- **time_step**
Required
Type: Positive Real
Time step for each tick. The representative execution **time** to compute on each tick.
- **ticks**
Required
Type: Positive Real
The total number of ticks to compute.
The overall simulation time is represented by the interpolation of **ticks** and **time_step**.
Note that the computed ticks are always **ticks + 1** since the first tick is used to define the initial world.

- **world**

Required

Type: Object

Defines the simulation world data dimensions.

The same world with different dimensions (keeping the same aspect ratio) can be used to obtain different levels of precision.

Data Object:

- **width**

Required

Type: Positive Integer

World's width.

- **height**

Required

Type: Positive Integer

World's height.

Note that the total number of the world's cells is made by the multiplication of `world.width` * `world.height`.

- **fluid**

Required

Type: Object

Define the fluid's data.

Data Object:

- **viscosity**

Required

Type: Positive Real

Fluid's viscosity.

- **density**

Required

Type: Positive Integer

Fluid's density.

- **diffusion**

Required

Type: Positive Real

Fluid's diffusion.

- **mods**

Optional

Type: Array

Defines the modifications to apply to the world.

A modification can mutate the fluid's density and/or apply a *new* force in a specific cell .

A modification can be applied at different ticks: starting from 0 (inclusive) to `ticks` (inclusive).

The modification is represented by a `Object` composed by the following fields:

- **tick**

Required

Type: Positive Integer

Tick's identifier on when to apply the current mod.

- **densities**

Optional

Type: Array

Array of **x** and **y** coordinates object on where to increase the fluid's density.

Data Object:

- * **x**
Required
Type: Positive Integer
X coordinate.
- * **y**
Required
Type: Positive Integer
Y coordinate.

– **forces**

Optional

Type: Array

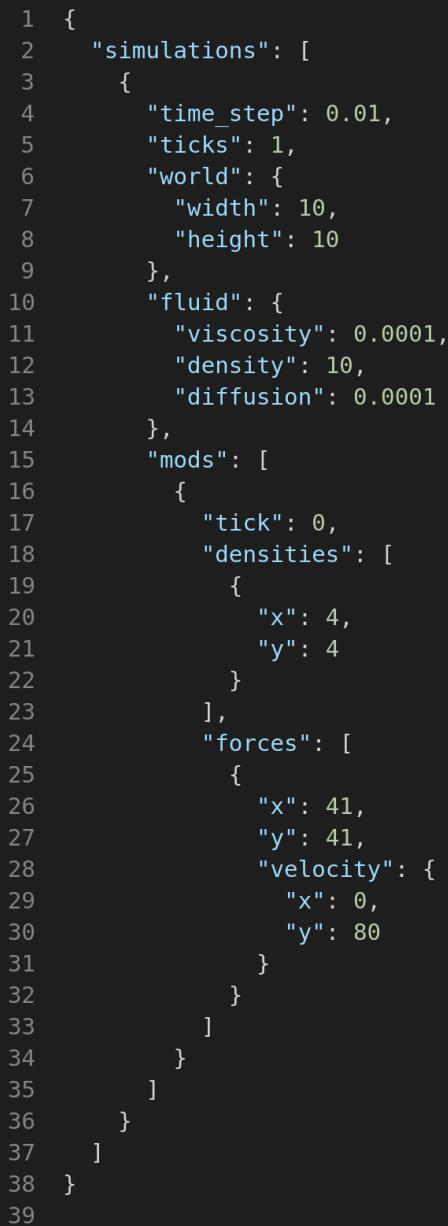
Array of coordinates object on where to apply a force and in which direction.

Data Object:

- * **x**
Required
Type: Positive Integer
X coordinate.
- * **y**
Required
Type: Positive Integer
Y coordinate.
- * **velocity**
Required
Type: Object
Object composed by **x** and **y** fields that determines the velocity direction and strength of the current force.
Data Object:
 - **x**
Required
Type: Real
X velocity and strength.
 - **y**
Required
Type: Real
Y velocity and strength.

All simulation objects are contained in a **Array** called **simulations**. This array is also the first entry key in the input file.

The figure in the next page shows an example of the content of a possible input JSON file (simulations.json). Note the strict hierarchy between Arrays/Objects/Fields.



```

1  {
2    "simulations": [
3      {
4        "time_step": 0.01,
5        "ticks": 1,
6        "world": {
7          "width": 10,
8          "height": 10
9        },
10       "fluid": {
11         "viscosity": 0.0001,
12         "density": 10,
13         "diffusion": 0.0001
14       },
15       "mods": [
16         {
17           "tick": 0,
18           "densities": [
19             {
20               "x": 4,
21               "y": 4
22             }
23           ],
24           "forces": [
25             {
26               "x": 41,
27               "y": 41,
28               "velocity": {
29                 "x": 0,
30                 "y": 80
31               }
32             }
33           ]
34         }
35       ]
36     }
37   ]
38 }
39

```

Figure 2.4: Generic structure of an input JSON file

2.4 Output

After a worker node has correctly computed a simulation, it saves an output JSON file that contains all the computed data.

The output file name is hardcoded and is composed as follows:

`simulation_[SIMULATION_ID]_[WORKER_NODE_RANK]` where `SIMULATION_ID` is the simulation identifier and `WORKER_NODE_RANK` is the rank of the worker node that has computed the simulation.

The save location is obtained by reading the `--results` argument from the environment. Note that the save location must be a valid folder with write permissions otherwise the program will

abort with an error message.

Each simulation result object is composed by the following fields:

- **id**
Type: Positive Integer
Simulation identifier.
- **metadata**
Type: Object
Metadata object containing useful information about the simulation. Most of the information in the object are the one given in the input simulation.
Data Object:
 - **time_step**
Type: Positive Real
The representative execution **time** to compute on each tick.
 - **ticks**
Type: Positive Real
The total number of ticks to compute.
Note that the computed ticks (in the snapshots-array) are always **ticks + 1** since the first tick is used to define the initial world.
 - **world**
Type: Object
Defines the simulation world data dimensions.
Data Object:
 - * **width**
Type: Positive Integer
World's width.
 - * **height**
Type: Positive Integer
World's height.
 - * **width_bounds**
Type: Positive Integer
World's width with bounds.
Note that the width bounds are always one cell on the left and one cell on the right.
 - * **height_bounds**
Type: Positive Integer
World's height with bounds.
Note that the height bounds are always one cell on the top and one cell on the bottom.
 - **fluid**
Type: Object
Define the fluid's data.
Data Object:
 - * **viscosity**
Type: Positive Real
Fluid's viscosity.
 - * **density**
Type: Positive Integer
Fluid's density.

* **diffusion**
Type: Positive Real
Fluid's diffusion.

- **snapshots**

Type: Array of Array

Snapshots array contains all the world's snapshots trough all the ticks.

The inner array contains an object that describes the state of a cell at the specific tick (identified by the index of the outer array). The size of this array is represented by the total number of the world's cells (including bounds).

The outer array contains the inner array. The size of this array is represented by the number of `ticks` + 1.

The final result is an array of the world's state in all the ticks.

Cell data object:

- **x**
Type: Positive Integer
X coordinate.
- **y**
Type: Positive Integer
Y coordinate.
- **d**
Type: Positive Real
Density of the fluid in the cell.
- **u**
Type: Real
X force velocity and strength.
- **v**
Type: Real
Y force velocity and strength.

For more information on how the results data are processed and how they are transformed from RAW JSON to a GIF image see section 4.2.

The figure in the next page shows an example of the content of a possible output JSON file (`simulation_6_7.json`).

```

1  {
2    "id": 6,
3    "metadata": {
4      "time_step": 0.01,
5      "ticks": 1,
6      "world": {
7        "width": 2,
8        "height": 2,
9        "width_bounds": 4,
10       "height_bounds": 4
11      },
12      "fluid": {
13        "viscosity": 0.0001,
14        "density": 10,
15        "diffusion": 0.0001
16      }
17    },
18    "snapshots": [
19      [
20        {
21          "x": 0,
22          "y": 0,
23          "d": 0,
24          "u": 0,
25          "v": 0
26        },
27        {
28          "x": 1,
29          "y": 0,
30          "d": 0,
31          "u": 0,
32          "v": 0
33        },
34        /* ... */
35      ],
36      [
37        {
38          "x": 0,
39          "y": 0,
40          "d": 25,
41          "u": 2.98,
42          "v": 0.1
43        },
44        {
45          "x": 1,
46          "y": 0,
47          "d": 30,
48          "u": -45.2,
49          "v": -7.1839
50        },
51        /* ... */
52      ]
53    ]
54  }

```

Figure 2.5: The generic structure of an output JSON file

3 Optimization choices

Here we'll describe how we used some OpenMP directives to parallelize the code and improve its performances. We'll also explain with comments and observations the choices about some chunks of code that were optimized thanks to these library.

3.1 OpenMP

The project had some critical parts that were parallelized using `OpenMP` to reduce the overall execution time.

We started by applying OpenMP directives to spawn a thread for each loop iteration whenever we thought it was needed. In those cases the operations inside the "for" loop acted on separated data structures and variables to avoid tainting the other threads' spaces. For example, whenever we cycled through all the matrices we added the relative OpenMP directive. Since this was an intensive operation, a huge performance increase was already observable in terms of execution time and CPU cores usage.

3.1.1 Is parallelization always a good choice?

We want to point out that using `OpenMP` isn't always a secure performance gain since spawning threads (parallel regions) is costly. If the section of code to execute is very simple and/or the number of iterations of a loop are low, it could be better not to use any form of parallelization and continue with the serial code alone. This evaluation is not as easy as it seems, since every system/architecture behaves differently and the operations of benchmarking and testing between a serial and a parallel code can cause false positives due to the overhead of the test itself. Furthermore, is not always possible to automatically decide whether to parallelize or not some piece of code relying on general assumptions. As a practical example, in the project we parallelized every `for` loop that processed the world's matrix. For small matrices this is a waste of resources and the overall performances are lower. As opposite, for large matrices, using the parallelization is a huge performance gain.

3.1.2 Terminate all threads if something goes wrong

However, we've encountered a major problem: how to stop/skip the parallel region if a thread has encountered a problem? This challenge showed up when allocating heap memory for the world's matrix: if the returned value from `calloc`¹ is `NULL`, it means that (possibly) there is no memory left and all threads should terminate.

We solved this by adding a `shared bool` variable named `error`, initialized to `false`. In the parallel region, we first check whether the `error` variable is `true`, and if so, something has gone wrong and the threads should stop. If not, we execute the for loop body as normal. At the end, if something went wrong, we update the value of `error` to `true` and the *information* is "propagated" to all threads in the parallel region. Since the variable `error` is *shared*, it must be protected from multiple concurrent writes using a `critical` section.

What has been described above is shown in the code below. Note that some parts have been removed to improve readability.

```
1 bool error = false;
2
3 #pragma omp parallel for \
4     schedule(DEFAULT_OPEN_MP_SCHEDULE) \
```

¹Allocates memory for an array of `num` objects of size and initializes all bytes in the allocated storage to zero. For more information see <https://en.cppreference.com/w/c/memory/calloc>

```

5      default(none) private(i) shared(ns, error)
6      for (i = 0; i < ns->world_height_bounds; ++i) {
7          if (error) continue;
8
9          /* ... */
10
11         if (something\_went\_wrong) {
12 #pragma omp critical
13             {
14                 error = true;
15             }
16         }
17     }

```

3.1.3 Parallelization of a double for loop with collapse

In the project, there's a significant number of parallel double **for** loop that are used to cycle through the matrix (double array). To parallelize not only the outer loop but also the inner loop we used the **collapse** *OpenMP* directive.

The OpenMP collapse-clause is used to increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread. If the amount of work to be done by each thread is non-trivial (after collapsing is applied), this may improve the parallel scalability of the OMP application[3].

In the code below, it's shown the usage of the **collapse** directive in a parallel region having two nested **for** loop. Note the value of **collapse**: it reflects the number of nested loop that are parallelized.

```

1  int i;
2  int j;
3
4  #pragma omp parallel for collapse(2) \
5      default(none) private(i, j)
6  for (i = 0; i < I; ++i) {
7      for (j = 0; j < J; ++j) {
8          /* ... */
9      }
10 }

```

4 Elaboration

4.1 Java and C comparison

This first small introduction serves to clarify the details about the process of comparison between the Java application and the original C translation of it.

The Jupyter notebook referred to in this section is called "java_c_comparison.ipynb" and the analysis performed inside of it is based on the equality of the cells' density throughout the simulation.

We made both the Java application and the C version generate output files(just text) containing a 3-tuple per row with x,y coordinates of each cell and its relative density, all grouped by frame(i.e. tick/snapshot of the simulation). The notebook can read these files, pre-process and compare them using pandas DataFrames for simplicity since they can be very long. Through this, we verified that the C program produced the same values as the original Java application.

4.2 Fluids visualization

For this part, another notebook was created and named "elaboration.ipynb".

The ultimate goal of it was to have a visual representation of the activity on the grid through the generation of an animated GIF per simulation executed.

The format of the output file coming from a simulation was also changed to a more organized JSON, where we placed a "metadata" section, describing the events happening during the simulation, and a "snapshot" section, where the actual data composed by series of 5-tuples was stored.

The results folder defined in the notebook will contain a file for each simulation performed in the previous execution of the program, in the format of "simulations_0_1.json". The "0" in there serves as an ID of the simulation and the "1" is the rank of the worker that performed said simulation. From these outputs we can pick the one of interest to be analyzed. The file is then loaded as JSON and split between metadata and actual data.

Information contained in the metadata is then extracted in variables like the number of frames and amount of points per snapshot, and these are compared with the "data"(or "snapshots") section of the output file to check for consistency of results. Exceptions are eventually raised whether there was declared in the metadata something that's not matching with the actual data. Data is then copied in a list of pandas DataFrames, where each DataFrame represents a snapshot of the whole grid at a certain point in time. This was done to easily handle data along the elaboration.

In this notebook we only processed and visualized the density of each cell, but further inspection could also be done on the vertical and horizontal velocity components, "u" and "v", already written in the output files. In fact, the output was enriched with those values, making each cell a tuple of 5 elements. With little effort, this gives the possibility for possible future improvements in the analysis of the fluids' behavior; from a climatic and very simplistic standpoint, these force components could allow to predict the intensity of winds in specific regions at specific times.

In the last part of the notebook, density values are normalized between 0 and 255 and then organized into a dictionary to get processed by the final block of the notebook, the GIF creation code.

4.2.1 Simulations

In this subsection are shown three simulations that has been processed from a raw JSON to a GIF file.

Since the final result of this processing an animated GIF image, only three frames are shown: first, middle and last.

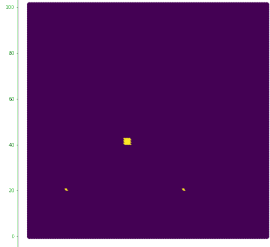
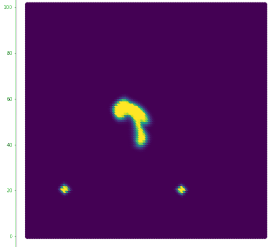
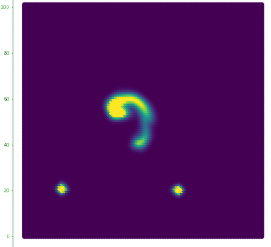
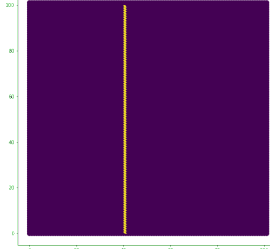
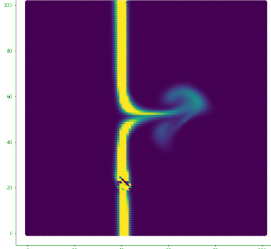
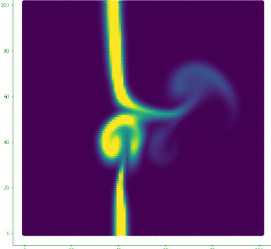
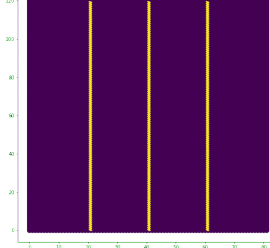
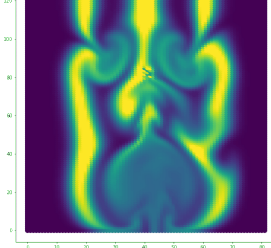
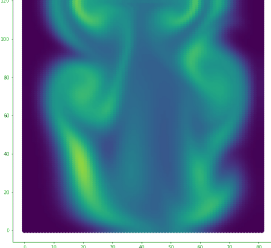
| Simulation Name | First Frame | Middle Frame | Last Frame |
|---------------------|--|---|--|
| Vortex ¹ |  |  |  |
| Wall ² |  |  |  |
| Bars ³ |  |  |  |

Table 4.1: Comparison between the first, middle and last frames of three different simulations computed

¹https://raw.githubusercontent.com/carlocorradini/hpc4ds/main/navier_strokes/notebook/animation_vortex.gif

²https://raw.githubusercontent.com/carlocorradini/hpc4ds/main/navier_strokes/notebook/animation_wall.gif

³https://raw.githubusercontent.com/carlocorradini/hpc4ds/main/navier_strokes/notebook/animation_bars.gif

5 Benchmarks

This chapter is dedicated to the comparisons between the computation time of the C project and its parallel version with OpenMP and MPI, and the overall performances of our optimized code.

5.1 PBS scripts and features (OpenMP & IO)

To facilitate our benchmarking activities we wrote 6 PBS scripts, each corresponding to a single Type of Table 5.1. What differs between them is:

- the number of nodes requested to the cluster for the computation
- the possibility of having MPI processes equally distributed one per node
- the amount of MPI processes that will be generated(one of them will always be the master, which doesn't count as worker process)

Here there's a further explanation of the scripts:

- *multi_node_multi_worker*
8 nodes and 8 MPI processes, one for each specific node.
- *multi_node_multi_worker_random*
8 nodes and 8 MPI processes.
- *multi_node_single_worker*
2 nodes and 2 MPI processes, one for each specific node.
- *multi_node_single_worker_random*
2 nodes and 2 MPI processes.
- *single_node_multi_worker*
1 node and 8 MPI processes are requested.
- *single_node_single_worker*
1 node and 2 MPI processes are requested.

From these scripts, we obtained the results reported in Table 5.1 and Figure 5.1. Each column of the histogram is an execution of simulations.json, so of the 7 different scenarios defined inside that file, under a different PBS script(whose names are reported below the Figure).

What can be seen is that, as expected, the worse case is when there's a single node on a single worker, which is equal to not having MPI parallelizing the work since there will only be one worker(this script schedules 2 MPI processes, and one will be the master). In this specific situation, when having just the I/O with no OpenMP directives, the computation time is logically equivalent to our original C serial code. This last situation falls of course in the highest column. The other results are all aligned with our expectations, with OpenMP reducing the computation time and the I/O activities increasing it.

By looking at single groups of columns the optimization given by OpenMP is clearly visible but may not seem to be that relevant. We don't have to forget that the parallelism was of fine-grained type, and also that the 7 testing simulations used for each column were not so big in grid size or ticks(to avoid stressing too much the cluster); about this last point, the overhead given by forking threads for workloads that are not huge can quite compensate the presence of multiple threads, reducing the overall speedup of the application.

Ideally, the project can be used to process multiple different regions of the Earth simultaneously, or simply, several scenarios of fluids with specific characteristics floating in space.

We want to point out that the benchmarks, especially the multi-node ones, are obtained as an average of the total execution time and this can be influenced by many factors. These factors can be: transfer speed, amount and complexity(grid size, ticks) of the simulations, PBS jobs number, cores per processor, the number of workers, and many more. Due to the intrinsic complexity of processing a simulation, a worker node can compute multiple simple ones and terminate before worker nodes that are processing even just one complex simulation. The resulting execution time is high due to the single complex simulation but the throughput (average number of simulations computed) remains high. As said in the previous chapters, the number of workers in combination with the number of nodes must be well balanced not to waste resources (unused workers) and to avoid waiting an elevated amount of time for the completion of all simulations.

An additional note on Figure 5.1: since the highest column logically represents the program without MPI nor OpenMP it's safe to assume that, in comparison with the alternatives using multiple nodes, multiple workers or both, the improvement given by our hybrid approach was a success, with the best simulation environments being able to execute the same tasks in approximately 1/7(around 50 against 350 seconds) of the time w.r.t. the worst-case scenarios.

| Type | OpenMP + IO | OpenMP (no IO) | IO (no OpenMP) |
|--|----------------------------------|----------------------------------|----------------------------------|
| <i>multi_node_multi_worker</i> | 48 765 675 μ s ~48.77 s | 37 150 211 μ s ~37.15 s | 60 469 822 μ s ~60.47 s |
| <i>multi_node_multi_worker_random</i> | 51 407 989 μ s ~51.41 s | 36 302 302 μ s ~36.3 s | 59 503 482 μ s ~59.5 s |
| <i>multi_node_single_worker</i> | 108 548 061 μ s ~108.55 s | 82 996 835 μ s ~83 s | 123 329 268 μ s ~123.33 s |
| <i>multi_node_single_worker_random</i> | 114 671 633 μ s ~114.67 s | 78 700 542 μ s ~78.7 s | 127 189 385 μ s ~127.19 s |
| <i>single_node_multi_worker</i> | 134 209 238 μ s ~134.21 s | 118 739 563 μ s ~118.74 s | 151 935 992 μ s ~151.94 s |
| <i>single_node_single_worker</i> | 305 416 664 μ s ~305.42 s | 257 811 506 μ s ~257.81 s | 358 784 128 μ s ~358.78 s |

Table 5.1: Time comparison between different PBS scripts and features (OpenMP & IO)

In the next page the graphic that compares the *six* different *PBS* scripts and OpenMP & textttIO features.

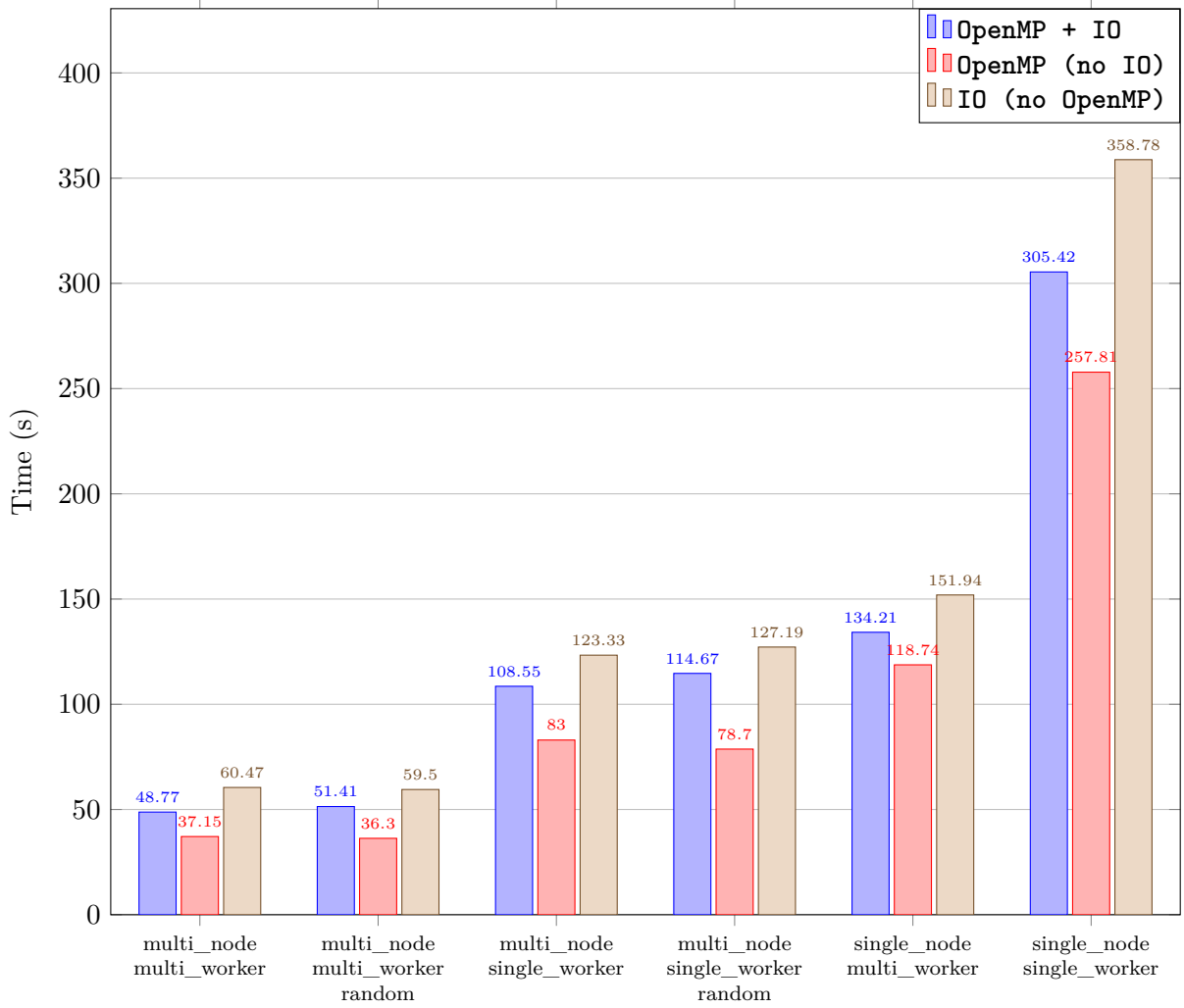


Figure 5.1: Time comparison between different PBS scripts and features (OpenMP & IO)

5.2 OpenMP schedule

We originally used an "auto" type of loop scheduling whenever it was necessary, and then we made the following analysis to understand whether it was the best possible choice. In Fig. 5.2 there's a histogram with our results.

The tests were performed with multiple workers among multiple nodes, 5 times for each column(the times reported are their average). Each execution was performed with our standard "simulations.json" file, containing 7 simulations each different in grid size, amount of ticks, and other input parameters.

What's obvious by looking at the picture and at the Table 5.2 is that, with a generic set of simulations for each run, the worse was generally obtained in the "dynamic" scenario. With this scheduling type, the iterations are broken up into chunks of a given `chunksize` amount of consecutive iterations, and each thread requests another chunk at run-time whenever it has done executing the previous. The overhead given by doing the assignment at run-time was clearly important in our simulations. By default, the chunksize is set to 1, and the worst result was obtained with exactly this number of consecutive iterations assigned. A variation to 10 greatly improved the performances, probably because threads asked for new chunks less frequently. Maybe the distribution among the threads was not so balanced as the case with a chunksize equal to 1, but it surely was faster.

In general, for the other types of scheduling, the computation times were all slightly similar.

| Schedule | Time |
|--------------------|---------------------------------|
| <i>auto</i> | 46 150 521 μ s ~ 46.15 s |
| <i>static</i> | 46 833 196 μ s ~ 46.83 s |
| <i>static, 10</i> | 47 941 849 μ s ~ 47.94 s |
| <i>dynamic</i> | 75 537 427 μ s ~ 75.54 s |
| <i>dynamic, 10</i> | 54 632 091 μ s ~ 54.63 s |
| <i>guided</i> | 43 158 797 μ s ~ 43.16 s |
| <i>guided, 10</i> | 48 294 501 μ s ~ 48.29 s |

Table 5.2: Time comparison between different *OpenMP* schedules

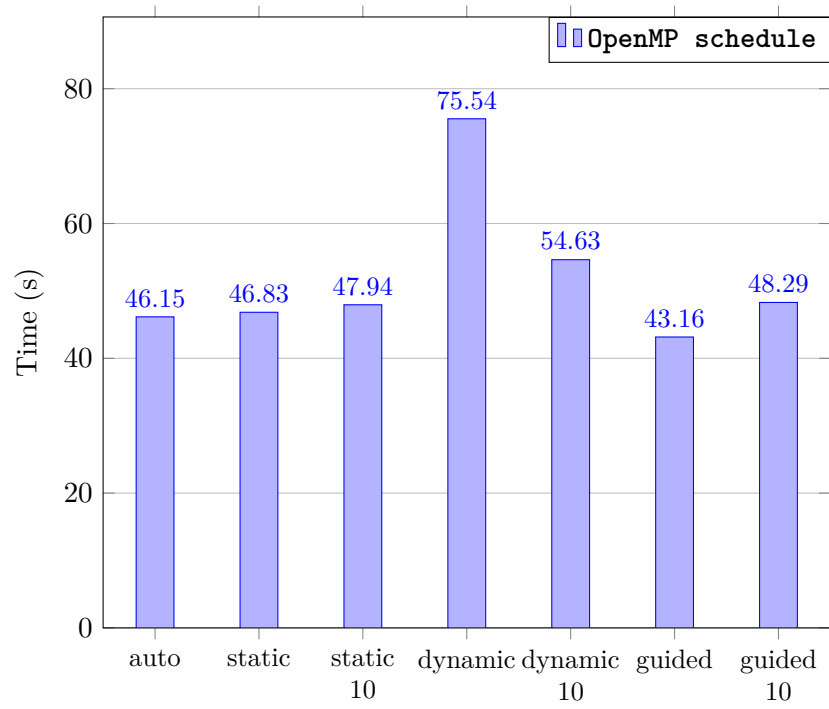


Figure 5.2: Time comparison between different *OpenMP* schedules

Bibliography

- [1] argparse contributors. argparse. <https://github.com/cofyc/argparse>.
- [2] cJSON contributors. cJSON. <https://github.com/DaveGamble/cJSON>.
- [3] Intel contributors. OpenMP Loop Collapse Directive. <https://software.intel.com/content/www/us/en/develop/articles/openmp-loop-collapse-directive.html>.
- [4] JSON contributors. JSON. <https://www.json.org>.
- [5] Wikipedia contributors. Master/slave (technology). [https://wikipedia.org/wiki/Master/slave_\(technology\)](https://wikipedia.org/wiki/Master/slave_(technology)).
- [6] log.c contributors. log.c. <https://github.com/rxi/log.c>.
- [7] CMake website. CMake. <https://cmake.org/>.
- [8] Make website. Make. <https://www.gnu.org/software/make/>.
- [9] PBS website. Portable Batch System (PBS). <https://www.openpbs.org/>.

Attachment A Benchmarks in depth

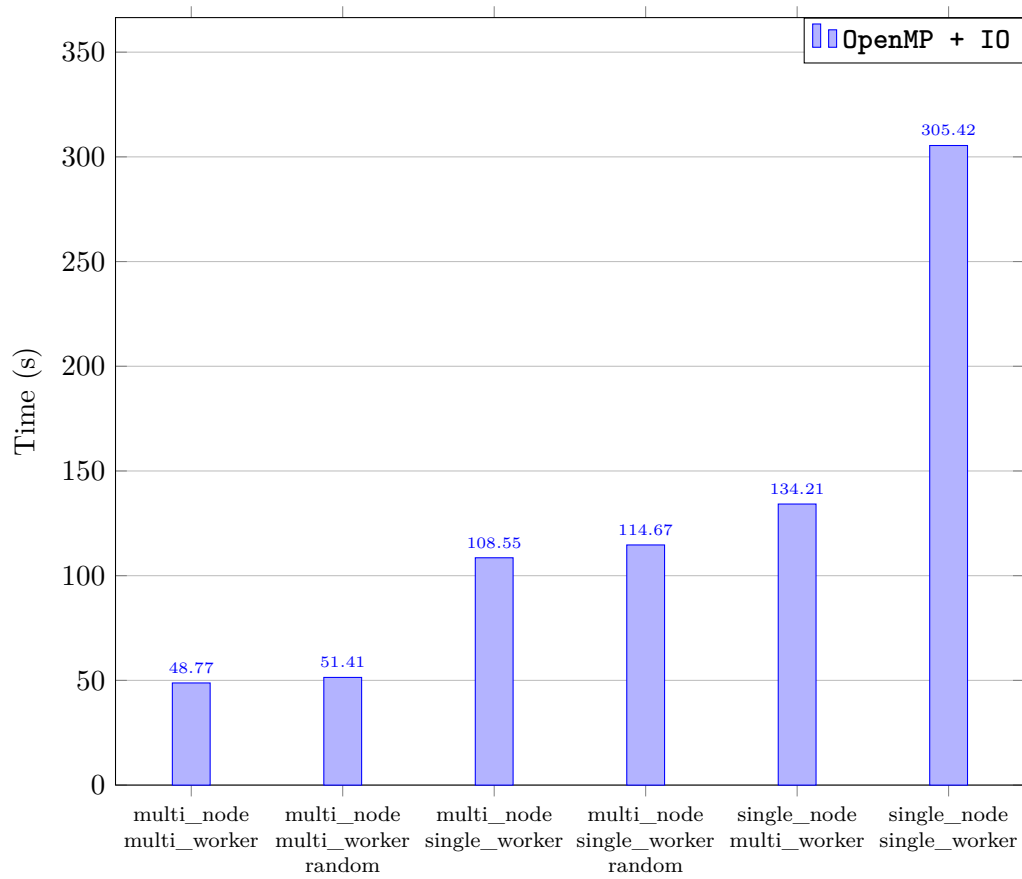


Figure A.1: Time comparison between different PBS scripts having OpenMP & IO features enabled

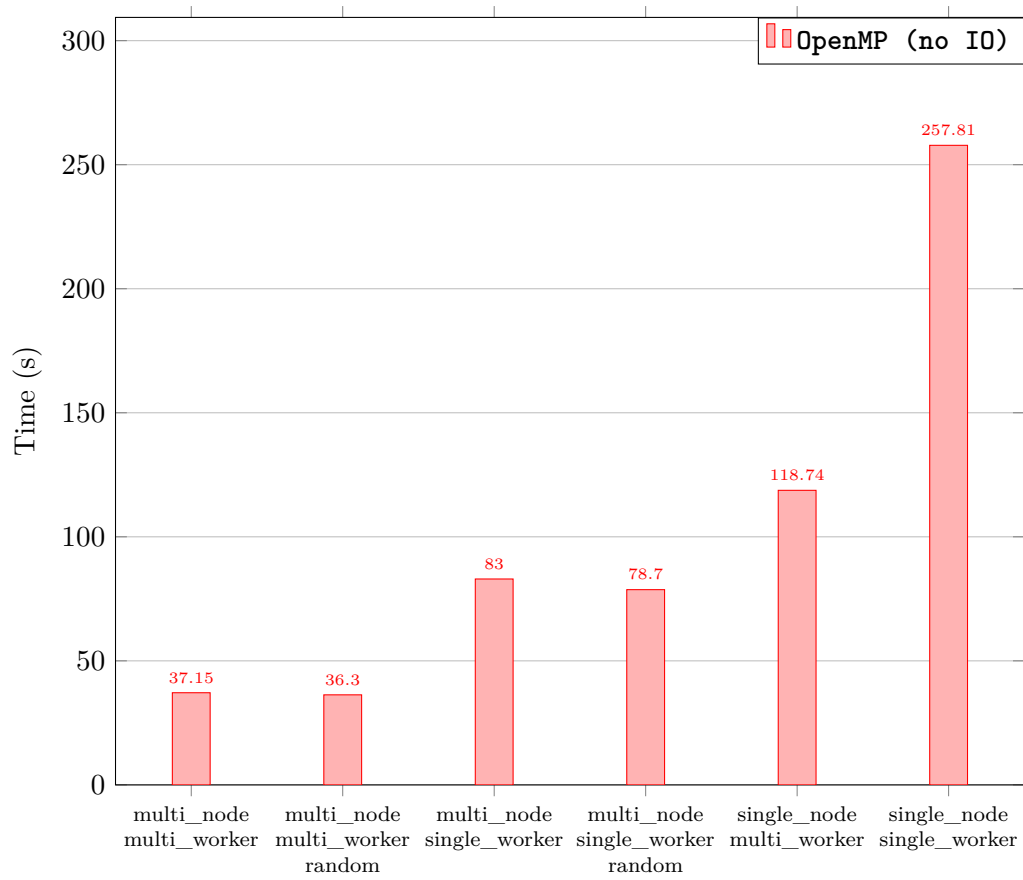


Figure A.2: Time comparison between different PBS scripts having **OpenMP** feature enabled and **IO** feature disabled

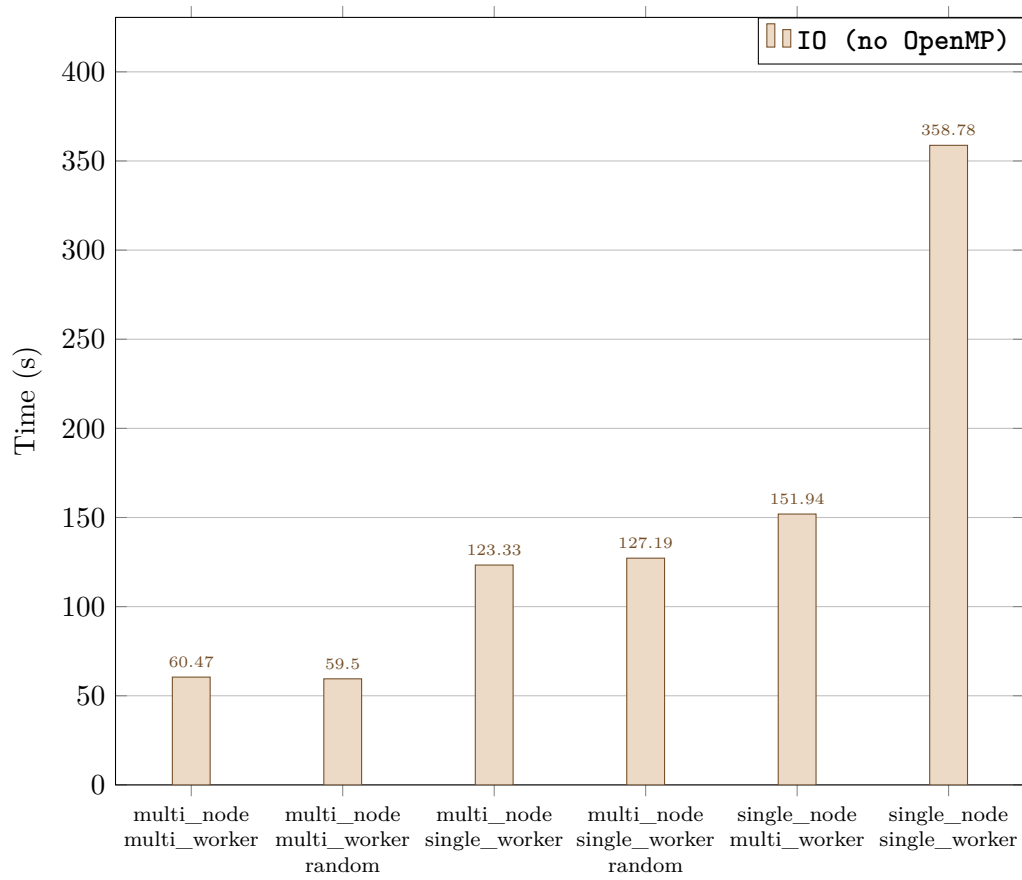


Figure A.3: Time comparison between different PBS scripts having IO feature enabled and OpenMP feature disabled

