Characterization of Parallel Scientific Simulations

Borja Miñano Maldonado

Master Project Thesis in Computer Engineering

High Performance Computing

Francesc Guim Bernat, Thesis supervisor

Josep Jorba Esteve

January 15th, 2017

DOCUMENTATION DATA SHEET

Thesis title:	*Characterization of Parallel Scientific Simulations*
Author name:	*Borja Miñano Maldonado*
Thesis advisor name:	*Francesc Guim Bernat*
Tutor name:	*Josep Jorba Esteve*
Delivery date (mm/aaaa):	01/2017
Degree:	*Master in Computer Engineering*
Department:	*High Performance Computing*
Thesis Language:	*English*
Keywords	*HPC, Performance, Characterization*

Resumen del Trabajo (máximo 250 palabras)

Las simulaciones de ámbito científico demandan una cantidad elevada de recursos computacionales. Es por ello que la computación de altas prestaciones se hace imprescindible para resolver sus problemas. Uno puede estar tentado a pensar que la única manera de mejorar el rendimiento de dichas aplicaciones es mejorando el hardware de las máquinas. Sin embargo, la metodología y protocolos con los que se programan las simulaciones son factores críticos de cara a obtener un buen rendimiento.

La caracterización y posterior análisis de dichas simulaciones permiten a sus desarrolladores a conocer su comportamiento en detalle de cara a mejorar sus puntos flacos y

cuellos de botella.

El objetivo de este proyecto es analizar tres simulaciones científicas reales en un entorno de computación de altas prestaciones de cara a detectar sus cuellos de botella y proponer posibles mejoras. Para ello se proponen las métricas a obtener y un estudio de posibles herramientas con las que almacenar los datos. Paralelamente se explican brevemente como funcionan las tres simulaciones escogidas y la razón de su elección. Seguidamente, se exponen los resultados y se realiza el análisis individualizado de cada simulación, donde se extraen conclusiones respecto a cada una de ellas. Para finalizar, en el apartado de conclusiones generales, se exponen las lecciones aprendidas de cada una de estas fases y del desarrollo del proyecto.

Abstract (in English, 250 words or less):

Scientific simulations are high demanding computational applications. For that reason, high performance computing is absolutely necessary to solve their demands. One can be tempted to think the only way to improve their efficiency is using hardware with better performance. Nevertheless, the methodology and protocols implemented in the simulations are critical to obtain an accepted performance.

The characterization and analsysis of those simulations rise unknown behaviour that can be seized by skilled programmers in order to improve the bottlenecks and weakest points.

The project objective is the analysis of three real scientific simulations in a high performance computing environment to detect their bottlenecks and make improvement proposals when possible. To do so, a set of metrics are proposed, and a research among the possible tools to obtained them. Simulataneously, the three chosen simulations are briefly presented and the justification of their usage explained. Subsequently, the results are

exposed, and the analysis for every simulation is made, leading to individual conclusions. Finally, in the conclusions section, the lessons learnt during the development of the project are shown.

Table of Contents

Table of images

Table of tables

Introduction

Motivation

High Performance Computing (HPC) is a set of tools and protocols that are widely applied in different areas such as business analytics or weather forecasting. However, scientific and engineering simulations are two of the most demanding fields for this technology, also making huge efforts in improving it. For instance, the simulation of one percent of the human cerebral cortex needs the usage of new supercomputers as IBM's Blue Gene (1). In a near future, more powerful HPC systems are to be necessary to fulfill the new simulation requirements to come.

Nevertheless, increasing the power of supercomputers is not the only issue that simulation developers must be concern. The classical approach to solve demanding scientific computing is to parallelize as many sections of code as possible. Usually, the data of a simulation is partitioned in multiple subdomains, all of them being distributed to different nodes in a HPC system. This process is called domain decomposition. Each subdomain may need, at some point, the information of adjacent subdomains for computing purposes. Consequently, communication is needed between the processes to synchronize the data. Choosing a wrong communication pattern or protocol can affect negatively to the performance of the simulation. This problematic, and many others, emphasizes that not only computing power is needed, but also protocols and expert programming skills are required to run a simulation with proper performance (2).

The requirements of one simulation may be different to another one. For instance, a simulation may need a full bandwidth communication pattern, or a shared memory infrastructure. Or, maybe, a simulation needs to write to disk intensively or read from memory in a high rate. Then, the characterization of an HPC simulation is extremely important in order to exploit the computing resources. Program analysis tools are used to understand program behaviour. These tools focus on the analysis of the applications and the identification of critical sections in the source code.

As a member of the active research Institute of Applied Computing and Community Code (IAC3), developing scientifically parallel simulation code, it is strongly recommended to profile and characterize our simulations for a better understanding of the current and future simulation codes and the way of increasing their performance.

Objectives

The main objective for this project is the analysis and characterization of a set of real simulations with different resource requirements. There are also some extra objectives:

- Reporting a clear profile of the simulations based on a set of metrics
- Detection of bottlenecks
- Proposing, when possible, improvements of performance

Project Requirements

A HPC machine or cluster is the only hardware requirement. It is needed that the machine has at least two nodes to allow parallel execution. The computing power of the machine is not relevant in this project since the purpose is not to quantify the speed of the simulations but to characterize them.

In order to run the simulations, the following libraries must be installed:

- Gcc 4.5.0 or 4.6.0
- OpenMPI 1.6
- HDF5 1.8.5-patch1
- Silo 4.7.2
- Boost 1.45
- SAMRAI 3.6.3-beta

Simulation Test Cases

Parallel applications can be classified in different ways, depending on how the instructions and data are handled. According to Flynn's taxonomy, the simulation test cases chosen for this project are SPMD (Single Program, Multiple Data), which means that the same algorithm is executed by all the processes over a different set of data (3).

These simulations have diverse requirements on a regular machine basis. Despite the difficulty to isolate a bottleneck in scientific simulations, in this project there has been tried to choose three test cases in which there is a dominant resource demand among the others.

For the whole set of simulations the execution consists in 3 main phases, an initial framework setup, the initialization and the evolution. The first phase depends on the framework, and use to be used for network and environment setup. In the initialization phase, the memory allocation and initial data setting are performed. The initialization and setup phases are usually much faster than the evolution, being undepictable in real simulations. The evolution phase is the main core of the simulation, where the calculations are done. The evolution is a set of time steps executed sequentially, and at the end of each time step there is an optional dump to disk of the current simulation state.

Memory-bound simulation

RAM memory is the most demanded resource in this kind of simulations. The data is usually shared between different nodes due to the size of the simulation domain.

The test case is a basic Advection equation, which physically models the transport of a substance. The computing of the Advection equation has an extremely simple solution. This is the reason to choose this problem as a memory-bound example. By increasing the domain size as much as desired, the complexity of the simulation does not increase, minimizing the CPU user usage.

Applications

The advection simulation is one of the simplest equations modelled by Partial Differential Equations. It is used in different fields, as engineering, physics or meteorology, to simulate the motion of a scalar field while it is advected by a known velocity vector field. For instance, the equation models how fog is advected by air flow (4).

The equation in Figure 1 shows the Advection's partial differential equation.

$$\frac{\partial \psi}{\partial t} + u * \nabla \psi = 0$$

Figure 1. Advection equation

Being Ψ the field wanted to observe and u the vector describing the movement of the environment.

Simulation objective

The test case consists in a sinus profile being advected in a periodical domain. So, at the end of the simulation, the sinus should be at the same position with the same shape as at the beginning.

In order to solve partial differential equations computationally, some approximations must be made when discretizing the continuous equations. Several numerical discretization schemes can be used, some of them fit better in a kind of problems than others. In order to test the goodness of a discretization, the simulation of this test case is absolutely appropriate. The nearer the final sinus shape is to the initial one, the better the numerical discretization algorithm is.

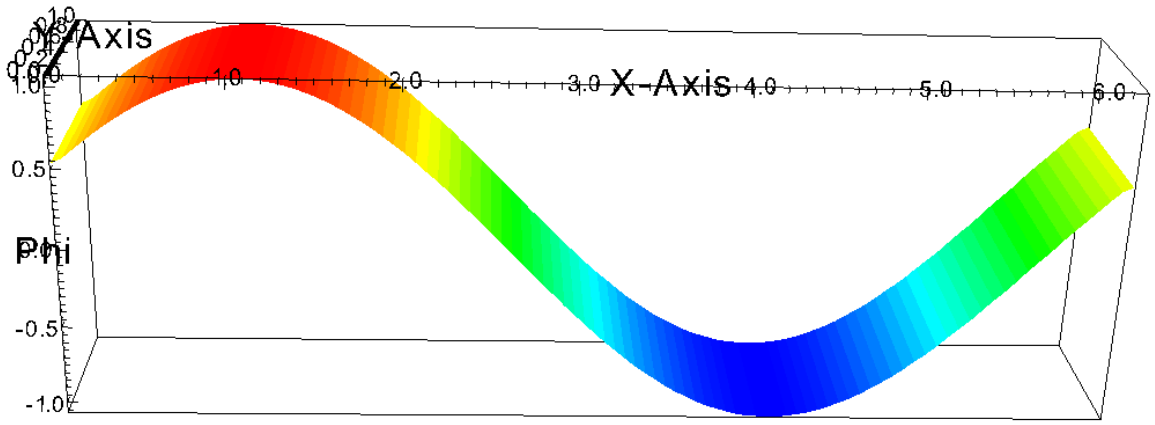


Figure 2. Advection simulation output

Quantitative information

The Table 1 lists the parameters used to profile this simulation case.

Dimensions	2
Number of time steps	10
Output frequency	1
Domain size	20000 x 20000
Number of variables	3 doubles + 7 integers
Expected size	20 GB data + overhead

Table 1. Advection simulation parameters

CPU-bound simulation

Simulations using large and complex code are excellent test cases for CPU-bound tests. Those simulations usually require many evolution steps in order to converge to a solution. As their domain may not be very extensive, domain decomposition is not always the correct approach to improve execution time. The expected behaviour is that the CPU usage dominates over the rest of resources.

The test case is an Euler Vortex, which emulates the movement of a vortex flow under Euler equations. Euler equations describe the movement of incompressible and inviscid flow. The simulation code is much more complicated than the Advection code. In this test, the domain chosen does not need to be extremely big in order not to hide the effects of the CPU usage.

Applications

Euler equations are a set of hyperbolic equations governing adiabatic and inviscid flow. They are constantly used in engineering to test stress on component parts in contact with water or wind. Simulations with Euler equations help to discover possible breaking points in components (5).

$$\frac{\partial \rho}{\partial t} + \rho * \nabla u = 0$$
$$\frac{\partial u}{\partial t} + \nabla \frac{p}{\rho} = g$$
$$\frac{\partial e}{\partial t} + \frac{p}{\rho} * \nabla u$$

Figure 3. Euler equations

The equations in Figure 3 represent the conservation of mass, balance of momentum and energy.

Simulation objective

The test case consists on a fluid vortex that is moving quickly and, after several iterations, it returns to the initial location as seen in Figure 4.

The aim of the test is the same as the Advection test case, checking the goodness of discretization methods. In this test, the discretization method is a WENO[1] method, an absolutely more complex discretization method than the one used in the previous test case. Consequently, the complexity of the simulation is higher, with many more calculations and variables.

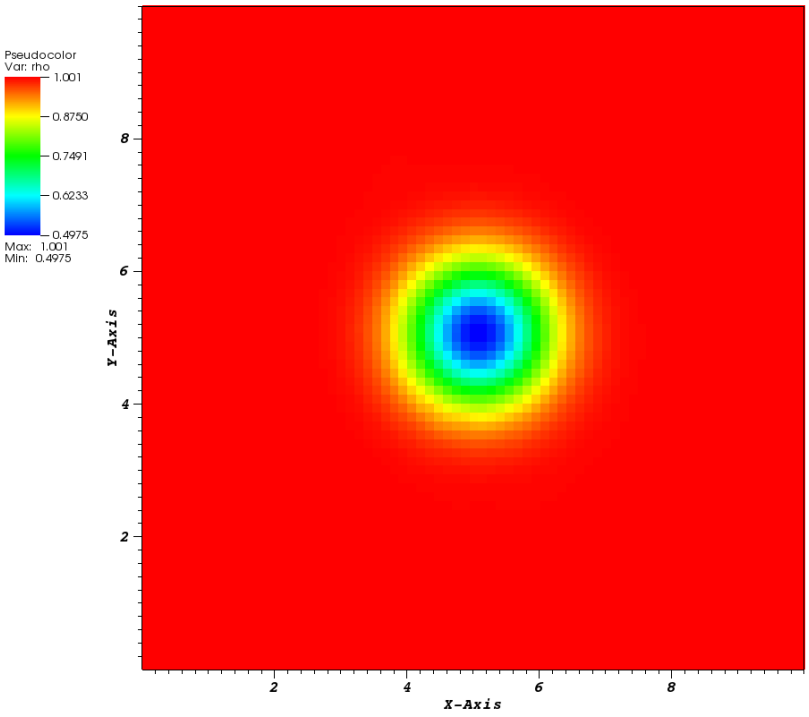


Figure 4. Euler simulation output

Quantitative information

The Table 2 lists the parameters used to profile this simulation case.

Dimensions	2
# of time steps (duration)	50
Output frequency	50
Domain size	5000 x 5000

[1] http://www.scholarpedia.org/article/WENO_methods

# of variables	61 doubles + 5 integers
Expected size	12 GB data + overhead

Table 2. Euler simulation parameters

I/O-bound simulation

The results of the simulations have to be written to disk for a later analysis by scientists. What is more, even intermediate states are written to control if the simulation is running properly or to have results with time dimension. An excessive output of this data to disk may create an I/O-bound problem.

The test case chosen is Cash and Goods. This is a toy Agent Based Model for trading. Its memory and CPU requirements are negligible, so choosing a high rated output should push the I/O capacity of the system.

Applications

This test case differs from the two previous ones in essence. The two previous cases simulate physical behaviour to solve problems based on nature events. This test case simulates a complex system. A system is considered complex if shows at least one of the following properties:

- Feedback. The output of a step is injected as part of the input in the next step
- Spontaneous order. From a primary chaos or disorder some structure emerges
- Robustness
- Emergent organization
- Numerosity
- Hierarchical organization

The cash and good model is an agent based model (ABM) to emulate a simplistic trading system market. An ABM is a computational simulation where every item or agent has its own individual behavior and it is able to communicate with other agents or the environment. From the interactions between the agents it can emerge an unexpected behaviour. That is the reason to use ABM to study complex systems.

The cash and goods algorithm consists on the following rules repeated cyclically:

- Calculate demand of goods and cash from the neighbour agents
- Calculate the price for offering goods in function of stock and demand
- Break interactions with neighbours offering better prices or not offering enough goods
- Exchange goods and refresh cash and goods values after transactions

Simulation objective

The aim of every complex system simulation is to generate enough output for a later analysis. As complex systems usually have some random behaviour it is important to simulate using different seed or initial conditions to have a proper statistical population.

This test case runs a single simulation, which is not appropriate for a later analysis, but enough for the scope of this project. The result of the simulation, see Figure 5, is a graph of connected and unconnected nodes with the values of cash, goods and price of every agent.

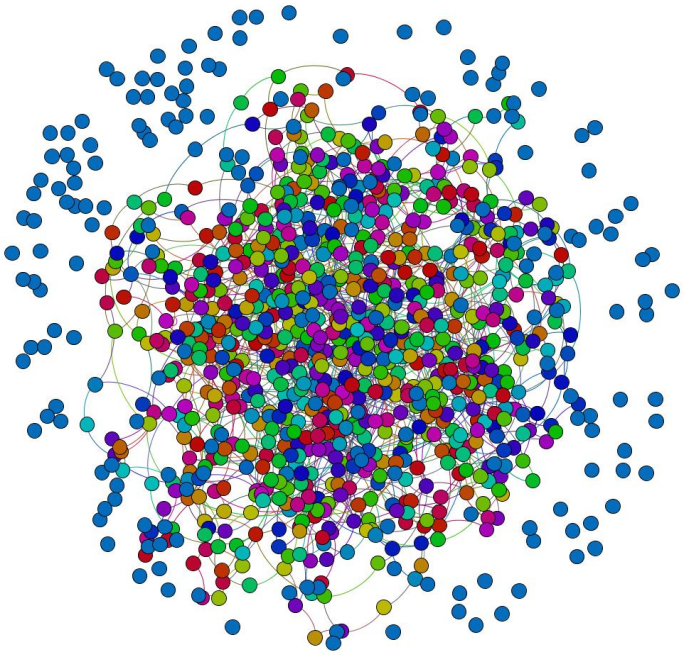


Figure 5. Cash and goods simulation output

Quantitative information

The Table 3 lists the parameters used to profile this simulation case.

Dimensions	Adimensional
# of time steps (duration)	10
Output frequency	1
Domain size	2000000 nodes
# of variables	5 doubles
Expected size	100 MB data + overhead

Table 3. Cash and goods simulation parameters

Test architecture

Domain decomposition

The three tests are based on domain decomposition in order to run in parallel. Every process in which the simulation is divided executes the same piece of code; the only difference is the data on the subdomain that is being executing in the process.

The Figure 6 and Figure 7 show the partition of the domain in mesh based simulations (Advection and Euler test cases) and graph based simulation (cash and goods test case).

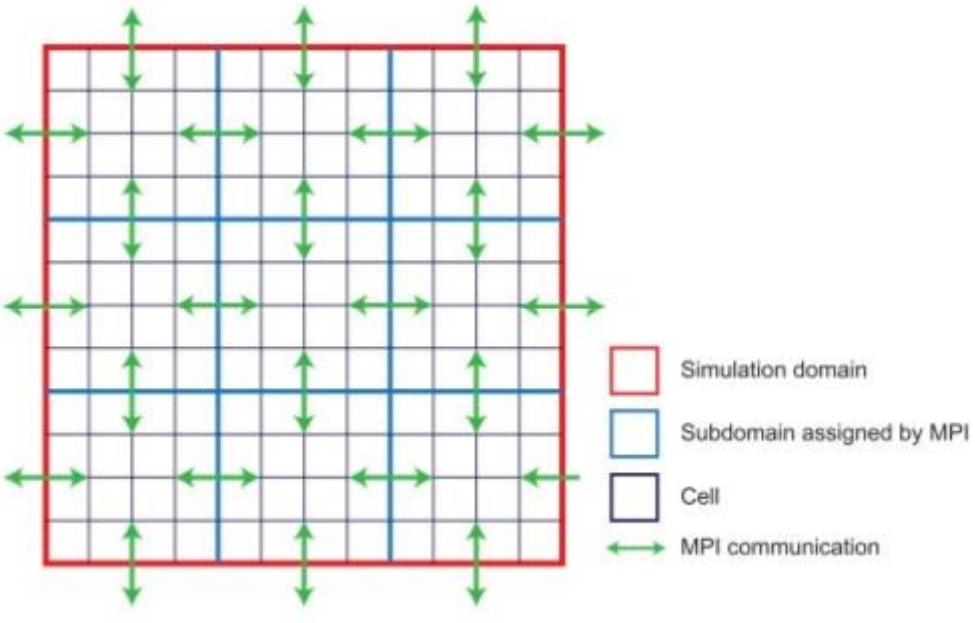


Figure 6. Domain decomposition on a mesh

In the first case, the mesh is decomposed in subsets of adjacent cell patches. The boundary cells on each process have to be communicated with the neighbour processes to synchronize information when required. The decomposition process is simple and is managed by SAMRAI, the simulation framework used in the mesh based tests.

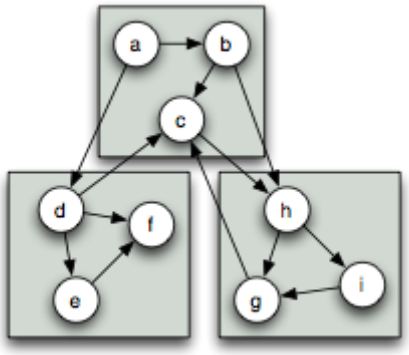


Figure 7. A graph decomposition for parallelization

In the case of graphs, the decomposition of the graph in different subgraphs is not as straightforward as the mesh decomposition. Good graph decomposition minimizes the connections between the nodes of the subgraphs in order to reduce the communication time. For this purpose, Metis library is used to partition the graphs by the parallel Boost Graph Library, the library for the graph based test.

Communication and synchronization

The communication pattern is critical in architectures using domain decomposition. Simulation computations can be classified in two groups, local updates or gathering calculations. The first group uses information stored locally in a cell or a node, while the second group needs information of the adjacent cells or nodes. Then, to have the right results, previously to a gather calculation, the information from adjacent cells and nodes must be available.

The common pattern to solve the stated problem in domain decomposition is to have some extra cells from adjacent processes. The region of the extra cells, called ghost zone, is synchronized between processes as shown in Figure 8. The ghost zone areas correspond to internal zones of adjacent processes, so subdomains overlap.

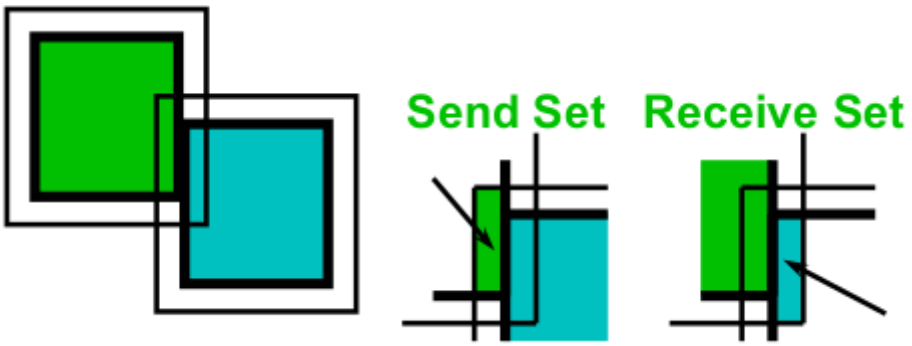


Figure 8. Adjacent mesh patches overlaping

In order to minimize the communication time, only the needed information is synchronized and only at specific times before a gather calculation.

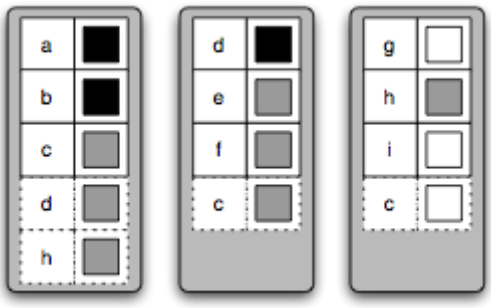


Figure 9. Subgraphs with own and ghost information

Regarding graphs, the strategy is similar. Nodes connected to another subgraph are also stored in the local partition, with all their values, as seen in Figure 9. A synchronization refresh ghost node values using information from their respective subgraphs.

Metrics

The characterization of a simulation is determined by its behaviour, which depends on multiple factors. The most important factor is the simulation code itself, but also the decision on parallel domain decomposition can change the performance. A metric objectively quantifies a property in a system. A set of metrics is proposed in order to profile the simulation test cases.

The metrics proposed can be classified in two levels. The first level consists in a CPU, Memory, I/O, and Parallel Communication usage. This level aims to classify a global bottleneck of the simulation. For the first level, the measures are obtained over time in order to have temporal profiles.

The second level consists on specific metrics of each subgroup. The detailed behaviour of a simulation code is shown through these metrics. This level of detail can help the developer to improve the source code in order to gain performance.

CPU Metrics

This set of metrics is oriented to decompose CPU usage. The metrics used in this group are:

- Arithmetic operations
- Function calls
- CPU usage detail

Ideally CPU usage will be obtained for every processor. And presented data will be average and standard deviation.

Memory Metrics

This set of metrics is oriented to study memory accesses. The metrics used in this group are:

- Reads
- Writes

- Page and cache faults

Memory usage will be obtained for each process.

I/O Metrics

This set of metrics is oriented to register disk requests. The metrics used in this group are:

- Reads
- Writes

The system only has one physical disk, so data will be written simultaneously by all the processes in the same disk.

Parallel Communication Metrics

This set of metrics analyzes communication patterns. The metrics used in this group are:

- Sending
- Receiving

Unless communication showed an unexpected behaviour, only a summary of communication usage will be shown.

Project Planification

The current project can be decomposed in the following tasks:

Inicial documentation

03/10/16 - 09/10/16 - 7 days

In this initial task the project has been set up. The requirements both for software and hardware have been established.

The test cases are chosen trying to challenge the spectra of main machine resources. This is made theoretically, since the run results and posterior analysis can lead to different conclusions.

Also a minimum set of metrics is established. It is not excluded to add new metrics to the study, especially in the second level, if they are found relevant.

Simulation environment setup

10/10/16 - 30/10/16 - 21 days

The simulation cases use a set of libraries that need to be installed in the HPC machine. In this task, the setup of the environment is done.

Once the environment is ready, some tests are performed using the selected test cases. These first runs are useful to test the machine limits in order to choose proper parameters for the real runs.

Metric documentation and implementation

10/10/16 - 07/11/16 - 29 days

An extensive survey is needed to choose the proper profilers. In this task, the metrics are deeply reviewed and can lead to changes in the list of the initial proposed list.

The implementation and test of the metrics is part of this task. To test the implementations, some small runs are performed in the environment.

Running and metric capturing

8/11/16 - 14/11/16 - 7 days

In this phase of the project, all the runs are launched using the profiler tools to capture the information required for a later analysis. They all are run according to the parameters yield from the machine setup tests.

Result analysis

15/11/16 - 09/12/16 - 25 days

The data obtained from the runs is analysed and reports are created to characterize each test case. When possible, performance proposals are described.

The analysis of the metric data is the aim of this project and one of the most important tasks.

Conclusions and documentation

10/12/16 - 24/12/16 - 15 days

This is the last task of the project, in which the final documentation is created, describing the whole process. The experiences and conclusions of the project are gathered.

Note: The Gantt diagram of the planification and tracking can be found in the Annex II – Gannt diagram, initial planification and tracking.

Implementation

Environment setup

The first step in the project is the configuration of the environment. The proposed hardware for the runs is an UOC shared memory cluster with the following features:

- 24 core (12 physical + 12 Hyperthreading) Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz
- 15 MB SmartCache
- 128 GB RAM
- Intel Corporation I350 Gigabit Network Connection Up to 1Gbps
- Seagate 4TB Enterprise Capacity HDD 7200RPM SATA 6Gbps 128 MB Cache
- CentOS Linux release 7.2.1511

In order to run the simulations, a list of libraries is required. Most of the libraries have to be compiled from sources in a specific way. The required libraries are described as follows:

- **Gcc 4.5.0 or 4.6.0.** This is the GNU compiler. The framework the simulations use is limited to these two versions. Later versions of Gcc do not work.
- **OpenMPI[2] 1.6.** When running parallel code, information exchange between nodes is necessary. OpenMPI is an implementation of the Message Passing Interface (MPI[3]) standard.

[2] https://www.open-mpi.org/

- **HDF5[4] 1.8.5-patch1.** The simulation output is stored using this scientifically wide-spread technology. HDF5 efficiently manages extremely large and complex data collections.
- **Silo[5] 4.7.2.** This library also manages the writing of scientific data to binary. Silo is used by the simulation framework to write data from particle method simulations.
- **Boost[6] 1.45.** The framework uses Boost utilities library to manage common structures. The specific Boost Graph Library[7] is also used in Cash and Goods simulation as the library which manages the graph environment and partitioning.
- **SAMRAI[8] 3.6.3-beta.** SAMRAI is the framework that Advection and Euler simulations use. This library manages the mesh partitioning, output and running conditions.

Actually, the GNU compiler version 4.6.4 and OpenMPI version 1.6 were already installed in the system. The sources from the rest of libraries were downloaded, compiled and properly setup with environmental variables. The detail of the commands can be read in the Annex I - Library Installation Commands

Metric implementation

This section describes the research among profiler tools and the implementation of metrics using the selected tools.

Profiler types

Profilers use a wide variety of techniques to collect data. All of them have advantages and disadvantages, as shown in the following list.

---

[3] http://mpi-forum.org/

[4] https://support.hdfgroup.org/HDF5/

[5] https://wci.llnl.gov/simulation/computer-codes/silo

[6] http://www.boost.org/

[7] http://www.boost.org/doc/libs/1_45_0/libs/graph/doc/index.html

[8] http://computation.llnl.gov/projects/samrai

- Event-based profilers collect data on the occurrence of determinate events from a specific event set.
- Statistical profilers collect data from sampling. They are less accurate than event-based profilers, but add less overhead.
- Instrumentation is a technique that adds instructions to the program to profile. As they interfere directly with the source code, they can lead to inaccurate results depending on the information collected and level of detail required.
- Interpreter instrumentation is a kind of instrumentation made on the language interpreter, so they do not interfere on the source code directly.
- Hypervisor/Simulator. The program is run over a hypervisor or a simulator platform which collects the data.

Research and documentation

There are a wide variety of profilers and utilities to measure, characterize and analyse the performance of applications; from proprietary to open source. In the research of this project, only the free and open source tools were considered. And, as the system in which to measure is CentOs, the tools must run in Linux based systems.

The metrics in which this project is interested are CPU, Memory, Communication, and I/O statistics. Apart from summary metrics, a profile in time is interesting for the test cases, since they have different phases which can show different behaviour.

Several tools were considered. The following sections show a brief detail and the strongest and weakest features that were taken into account for the election of the proper tools.

Perf

Perf is a performance analyzing tool in Linux using event-based profiling. This tool supports hardware and software counters, tracepoints and probes of several events, depending on the ones available by the kernel and the physical processors. It is used from command line and its simple interface provides the following commands:

- **Perf stat**. Event counter summary.
- **Perf record**. Record the events in timeline.
- **Perf report**. Reports the requested data based on the “Perf record” outcome.
- **Perf annotate**. Adds annotations to assembly or source code.

- **Perf top**. Live counters.
- **Perf bench**. Runs different kernel microbenchmarks.

The two first commands were considered to obtain the metrics. The **stat** command aggregates all counter data and shows it at the end of the execution, while the **record** command samples the counter data and reports based on functions.

Most of the events are related to assembly counters. For this project, the list of events that perf has available in the simulation machine are most convenient for summary reporting. Concretely, the events chosen are related to cache and L1 cache hits and miss ratios, memory access, and floating point calculations.

Perf stat was chosen as one of the tools to get summary metrics. Perf record could have been useful for function analysis in order to increase performance, but was discarded in favour of another tool.

Sysstat

Systat is a package of utilities to monitor usage and system performance. Most of these utilities are common in UNIX systems. There are tools for collecting data and others for reporting on the data collected. Most of these tools overlap in some metrics with Sar. Iostat, Pidstat and Mpstat output can be gathered with Sar using the proper flags. The only tools not overlapping with Sar are related to tape drives (Tapestat) and CIFS systems (Cifsiostat), features that does not have the system. Therefore, Sar was the only tool inside Sysstat used in this project.

Sar collects and reports a wide variety of system statistics, CPU, I/O, memory, network, paging and much more. This command line tool is configured via flags and can save the information to disk periodically. The frequency and amount of information to store are two parameters of the tool. In Metrics setup section there is the command line set for the study with an explanation of every flag.

Oprofile

Similar to Perf in features, Oprofile provides system and application profiling by sampling, and also provides counters on CPU level. Their commands are the following:

- **Operf**. Profiler through events.

- **Ocount**. Counter summary.
- **Opreport**. Reporting on the operf collected data.
- **Opannotate**. It annotates source code or assembly listings.
- **Opgprof**. Reports the operf collected data in a format similar to GNU gprof command.

Oprofile does not provide any extra feature than perf. As a consequence, after checking that perf provides the desired features, it has been decided not to use Oprofile for this project. Moreover, Perf is already installed in the system and Oprofile needs to be compiled and installed manually.

Valgrind (Callgrind, Massif)

Valgrind is one of the most well-known instrumentation frameworks for dynamic analysis. It includes six main tools and three experimental tools which perform profiling and debugging of programs.

- **Memcheck** is a debugging tool for memory error detection in source code, as memory leaks.
- **Cachegrind** is a cache and branch-prediction simulator. Cachegrind simulates how the program interacts with the machine cache hierarchy.
- **Callgrind** is a program function profiler. It creates a call graph with statistics from the functions and instructions called during the program execution.
- **Helgrind** and **DRD** are thread error detectors in multithreading that use POSIX pthread primitives. The difference between the two tools is that they use different analysis techniques.
- **Massif** is a heap profiler. It stores information related with memory usage and relates to the source code demanding that memory.
- **DHAT** is an experimental heap profiler. It helps detecting issues related with memory blocks and layout inefficiencies.
- **SGcheck** is an experimental tool to detect overruns of stack and global arrays.
- **BBV** is an experimental tool that generates basic block vectors for analysis with SimPoint tool. In computer architecture research, this tool helps reducing benchmark slowdowns.

Discarding debugging tools, since the simulation test cases have already been checked for bugs, there are interesting profiling tools in Valgrind. Cachegring, Callgring and Massif have been considered for this project.

Cachegrind is a cache simulator whose behaviour overlaps with Perf cache counting. While Cachegring is a simulator and Perf performs real cache counting on the processor, the tool from Valgrind has been dropped in favour of Perf.

Callgrind was strongly considered for function profiling and some initial tests were made on the test cases. However, it was discarded due to the excessive overhead it added. It was tested in parallel with the similar-featured Gprof, which turned out to be much lighter.

The last tool considered was Massif, the heap profiler. The output from this tool is extremely useful, giving temporal memory statistics and snapshots with memory usage detail. The metrics obtained are useful for both profiling and performance analysis. Consequently, this tool has been selected as part of the tools to be used in this project.

MpiP

Mpip is a specific library for MPI application profiling. It reports on the MPI functions and measures the time in MPI routines that an application spends in. It is useful to detect Communication bottlenecks, specially the ones that are risen by a wrong implementation design.

Despite it seems very useful for parallel application profiling; it is not going to be used in this project. There are two reasons for this decision. The first and most important is that the communication pattern in the simulations test cases are based on external libraries that have already proven their correct design and scalability. Then, since simulation analyses have not shown an unrecognisable pattern of bottleneck, it has not been necessary to use this tool.

The second reason is that the communication information wanted is related to sending and receiving rate, and this tool does not provide this kind of data.

Gprof

Gprof is a GNU profile analysis tool for UNIX applications. It uses hybrid instrumentation and sampling techniques. This tool generates elapsed time, statistics information and call graphs of functions and instructions.

The information aids detecting slow parts of the code and parts repeated more times than expected, being utterly useful for performance analysis.

Gprof generated data is similar to Valgrind's Callgrind. Nevertheless, during metric testing, Gprof has been discovered to introduce less overhead than Callgrind. Thus, Gprof has been selected for this project.

Vmstat

One of Linux commands interesting for getting statistics is vmstat. It reports information about processes, memory, paging, I/O and CPU activity.

Most of this information is obtained using Sysstat Sar command. So this command has been discarded for reporting in this project as it does not contribute with any extra information.

HPCToolkit

HPCToolkit is a suite for measurement and analysis of program performance on HPC systems. It uses statistical sampling of timers and hardware counters. It provides call graphs and line-by-line profiling.

Despite it includes all the functionality and yields complete statistical analysis, it was prefered to perform the metrics and analysis with the previous tools from this section for the sake of a better detailed control and apprenticeship on profiling techniques.

Metrics setup

Sar

Sar has plenty of flags to customize the measures to obtain. For the purpose of this study, the instruction executed is as follows:

sar -ubRB -n DEV -P ALL -o file INTERVAL

The flag description is:

- **-u**. CPU usage
- **-b**. I/O usage
- **-R**. Memory statistics
- **-B**. Paging statistics
- **-P ALL**. Provides cpu information of all processors independently

- **-o file**. Stores the result in the file provided.

The output intervals chosen for the Advection, Euler and Cash and goods simulations are 2, 10 and 5 seconds as the pieces of simulation run are 2, 10 and 5 minutes long. So they have a similar number of snapshots.

The output generated by the profiler provides several statistics, the following list expose the ones used in this thesis:

- CPU metrics:
    - user. Percentage of CPU utilization that occurred while executing at the user level (application).
    - nice. Percentage of CPU utilization that occurred while executing at the user level with nice priority.
    - system. Percentage of CPU utilization that occurred while executing at the system level (kernel). This field includes time spent servicing hardware interrupts.
    - iowait. Percentage of time that the CPU was idle during which the system had an outstanding disk I/O request.
    - steal. Percentage of time spent in involuntary wait by the virtual CPU while the hypervisor was servicing another virtual processor.
    - idle. Percentage of time that the CPU was idle and the system did not have an outstanding disk I/O request.
- I/O metrics:
    - rtps. Number of read requests per second issued to physical devices.
    - wtps. Number of write requests per second issued to physical devices.
    - bread/s. Amount of data read from the devices in blocks[9] per second.
    - bwtn/s. Total amount of data written to devices in blocks per second.
- Memory metrics:
    - frmpg/s. Number of memory pages[10] freed by the system per second. A negative value represents a number of pages allocated by the system.

[9] Blocks are equivalent to sectors and therefore have a size of 512 bytes.

[10] Page size in the current machine is 4 kiB.

    - campg/s. Number of additional memory pages cached by the system per second. A negative value means fewer pages in the cache.
    - fault/s. Number of page faults (major + minor) made by the system per second.
- Network metrics
    - rxpck/s. Total number of packets received per second.
    - txpck/s. Total number of packets transmitted per second.
    - rxkB/s. Total number of kilobytes received per second.
    - txkB/s Total number of kilobytes transmitted per second.

Valgrind Massif

Massif is integrated as a tool in Valgrind toolkit. It can be run in a parallel system by:

mpirun -np 24 valgrind --tool=massif ./Application

The output files can be read using:

ms_print massif.out.procId

Where the file provided is the output file generated by the previous instruction. One file is created for each of the 24 parallel processes.

Perf

Perf stat can precisely count every event providing there are enough counters available. Otherwise, it samples data. In order to maximize the precision of events, three runs of perf stat were launched separately.

perf stat -e cache-references:u,cache-misses:u,cycles:u,instructions:u - o perf_ipc.stats

This command captures the cache accesses and misses, the number of total CPU cycles and number of total instructions in the user space. In order not to capture system or other user interferences, the flag **:u** to obtain only user counters is specified. It stores the result in a file.

perf stat -e mem-loads:u,mem-stores:u,L1-dcache-loads:u,L1-dcache-load-misses:u -o perf_misses.stats

This command captures the L1 cache accesses and misses and the number of memory accesses in the user space.

perf stat -e r530110:u -e r532010:u -e r538010:u -o perf_fops.stats

This command captures the floating point operations in the user space using Intel Xeon (Sandy Bridge) processor registers (6).

The output generated by Perf is detailed as follows:

- cache-references. Last Level Cache accesses.
- cache-misses. Last Level Cache misses.
- mem-loads. Number of memory load accesses.
- mem-stores. Number of memory stores.
- cycles. Number of CPU cycles.
- instructions. Number of instructions served by CPU.
- L1-dcache-loads. L1 cache loads of data.
- L1-dcache-load-misses. L1 cache loads of data misses.
- Floating point registers:
    - r530110. Traditional 8087 style 80bit floating point operations
    - r531010. SSE double-precision on packed data (128 bit registers, so this is two operations).
    - r532010. One single-precision operation
    - r534010. Four single-precision operation (32bit single precision packed into 128 bit register).
    - r538010. One double-precision operation.

Gprof

In order to profile an application using gprof, the application must be compiled using the tag **-pg**. Additionally, if a line-by-line profiling is desired, the tag **-g** should also be specified.

Grof was not designed to perform parallel profiling. A run of an application would generate a gmon.out file with the statistics. If running in parallel using mpi (*mpirun -np X ./App*), every MPI process would generate a gmon.out which would overwrite the previous one. However, there is a solution, setting a gprof environmental variable to specify the filename as *export GMON_OUT_PREFIX='gmon.out-'`/bin/uname*. Then, an output file will be generated for every process (7).

After executing the simulation, the profile information regarding elapsed function time can be retrieved using:

gprof Application_binary gmon.out-procId > gprof.output-procId

For a line-by-line information:

gprof Application_binary gmon.out-procId -l -p > gprof_detail.output-procId

Performance characterization

This section shows the analysis of the results obtained from the application of the tools described in the previous sections to the test cases. Through the metrics obtained, the analysis may confirm the expected behaviour, Memory, CPU and I/O-bound; or it may show unexpected issues.

The three test cases are analyzed separately in different sections. However, they are compared among them in relevant metrics for the sake of referencing.

Test case 1 Advection

The Advection simulation test consists in a very simple algorithm, with only 3 mesh variables and few calculations, using a relatively high resolution mesh, and expecting a run prone to memory saturation.

As stated before, the run can be divided in three parts, the framework setup, the simulation initialization and the execution of the simulation evolution steps. The heavy core of the simulation is the last phase, which in production simulations takes almost the whole time. For this project, the simulation part is long enough to visualize the trend of a long-time simulation.

The following graphics show the behaviour of the Advection test case.

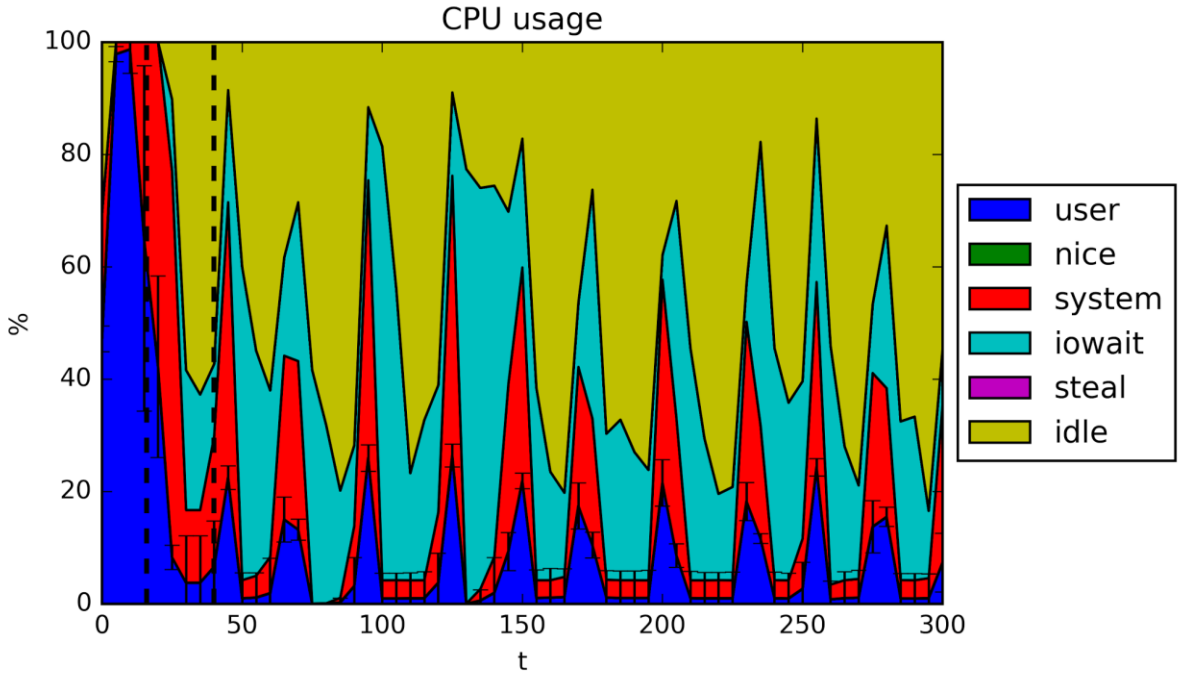


Figure 10. Advection CPU usage

The CPU usage shows a very low rate of user usage since the initialization phase (first vertical dashed line). The system remains idle and waiting for I/O operations at high rate. And, periodically, the system usage is also considerable. The I/O peaks coincides with the output from the simulation.

Note: the CPU graphic shows the average of the 24 processors used in the tests. The standard deviation on user usage is shown as vertical error bars at each point.

With these results, it can be inferred that it seems to be a bottleneck in the disk I/O. With the following graphics, this assertion can be reaffirmed.

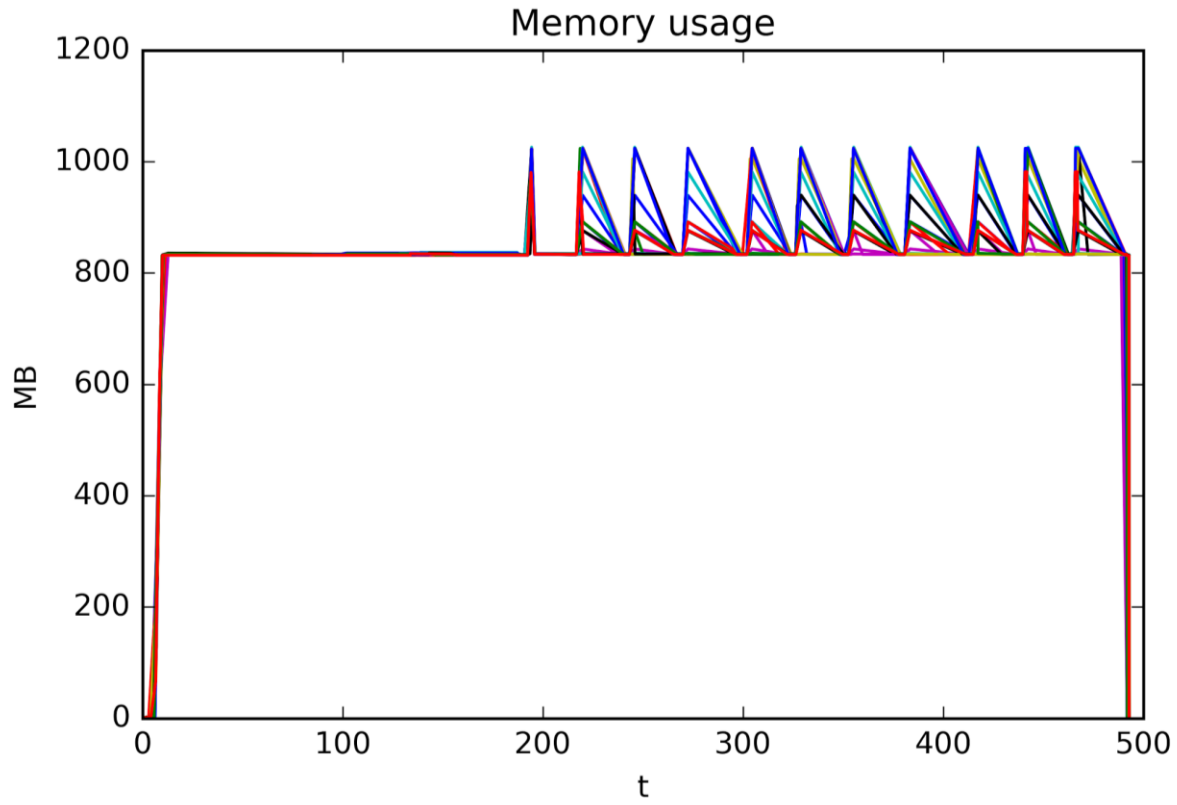
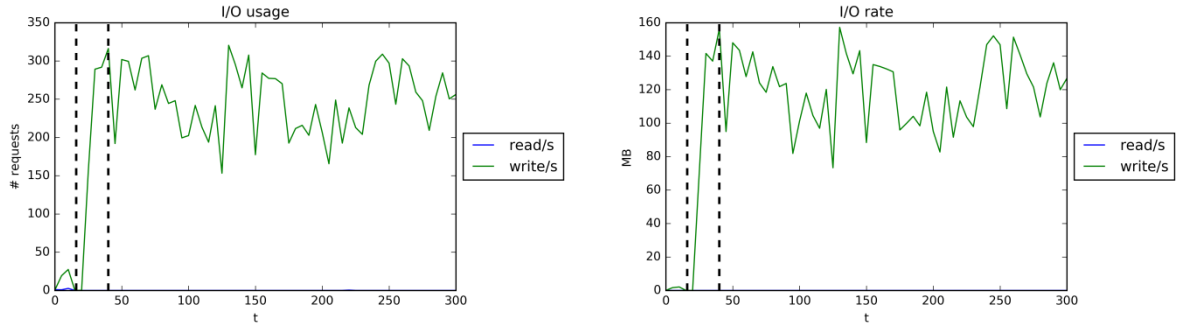


Figure 11. Advection memory consumption

The Figure 11 shows the 24 processors heap memory usage. It does not only include the resident memory, but all the virtual memory the process can potentially use. In this case, the difference is minimal.[11]

From the very beginning, all the processes reach near 800 MB. At initialization, all the in-memory variables are allocated. The peaks at the end of the simulations match with the disk output, which seems to require an extra 25% memory to handle the write to disk.

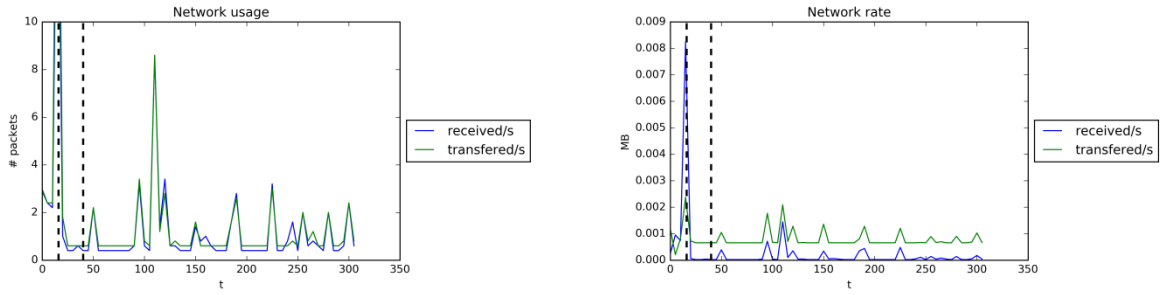
[11] Massif introduces an important overhead that slows down the simulation. That is the reason for the time to be longer than in graphics taken by Sar.



a. I/O usage **b. I/O rate**

Figure 12. Advection I/O metrics

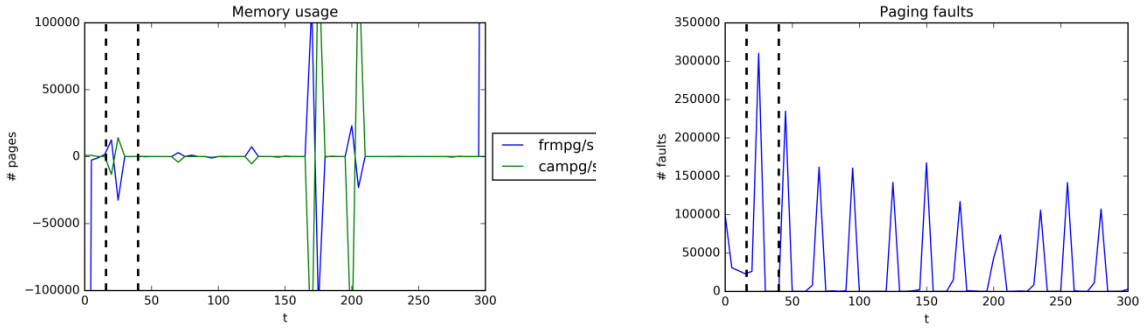
The I/O usage and rate shows a continuous disk write demand. The profiles of Figure 12 do not match with the I/O peaks from the CPU usage. The 24 processes need to write 130 MB simultaneously, with a total size of nearly 3 GB. It can be concluded that the size of the output and the concurrent access to the only physical disk overpass the capacity of the system. The simulation forces the system to be permanently writing, and, as a result, this situation drives to an I/O bottleneck.



a. Network usage **b. Network usage**

Figure 13. Advection network metrics

The network usage and rate are not high-demanding. The capacity of the system is able to deal with the network requirements of the simulation.



a. Memory usage **b. Paging faults**

Figure 14. Advection memory metrics

The Figure 14.a shows a relative constant liberation and allocation of memory pages and additional pages cached. Only two peaks between time 150 and 200 show a discontinuity, but they does not seem important enough as to consider a memory problem.

Regarding paging, there are some peaks of page faults reported periodically. Despite a page fault does not implies that the problem is related with I/O (8), due to the previous discoverings, the faults can be considered consequence of I/O.

Event	Count	Events/ins	Events/s	Extra
cache-references	404.975.325	0,34 %	1,34 M	
cache-misses	98.301.151	0,08 %[12]	0,32 M	23,27 % of cache refs.
mem-stores[13]	15.755.671.325	13,25 %	52 M	
L1-dcache-loads[14]	33.589.508.436	28,25 %	110,86 M	
L1-dcache-load-misses	560.140.347	0,47 %	1,85 M	1,67 % of L1 cache refs.
FLOPS[15]	4.647.643.271	3,91 %	15,34 M	

[12] This value represents the percentage of kernel instructions, not cache-references. The percentage of misses by references is shown in the extra column.

[13] There is no data referent to mem-loads due to the inability of perf to access that counter. Some processors do not support all perf events.

[14] The L2 cache was not available in the system.

instructions	118.897.660.369		392,41 M	1,11 IPC

Table 4. Advection Perf metrics

The perf counters show a summary information from all the run, which usually is not enough to characterize the code. Nonetheless, this information added to the temporal results given by Sar and Massif complete the analysis of the simulation.

The information by itself is not very clear. Comparing it with the other test cases the following conclusions rise:

- The number of FLOPS is very low. In a computational simulation it is a signal of poor performance.
- The rate of all cache misses is lower than the other test cases. The simulation code has very few variables and is simple. As a result, there is more probability of the data required to be in cache.
- The IPC is the highest. The instructions per cycle (IPC) statistic is a common performance indicator. It measures the number of instructions executed per CPU cycles. The higher the rate, the better profit of the resources. This test harnesses better the CPU when it is required than the other tests. However, the I/O bottleneck makes the CPU be idle most of time.

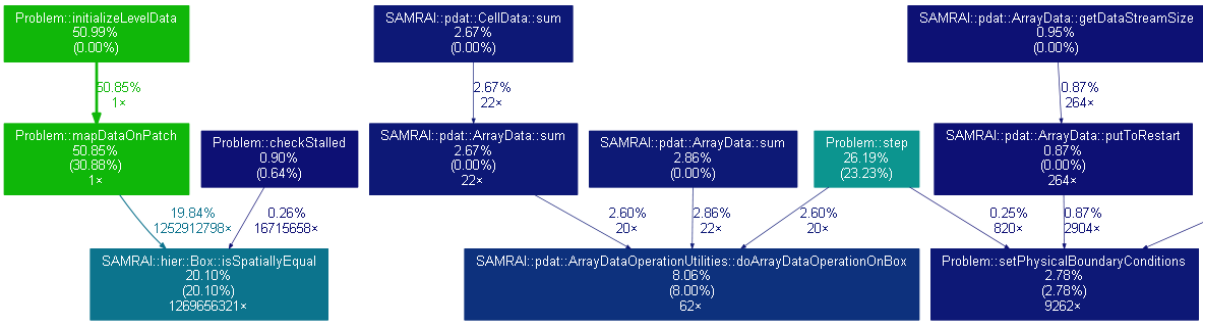


Figure 15. Advection remarkable call graph extract

[15] The Intel Xeon processors of the test machine belong to the Sandy Bridge family. There are five registers of the processors counting floating point operations available to perf: r530110, r531010, r532010, r534010 and r538010 (6). This metric summarize the five registers.

The Figure 15 shows the most important part of the call graph obtained by Gprof. As it can be seen, the most consuming function is the initialization (Problem::initializeLevelData), which takes near 51% of time. Then there comes the evolution (Problem::step) with a 26%.

Spending more time in initialization than evolution is an unexpected behaviour. However, as Gprof measures only user time (not system time) (9) and detected an I/O bottleneck in this simulation, the graph call makes sense. The initialization only requires one write to disk, so the user time is seized completely. On the other hand, the evolution wastes plenty of time waiting for I/O, using a few time in user space, where the computations are calculated, which is what Gprof accounts.

Even in the call graph it keeps clear this simulation has a bottleneck related with system calls, in this case I/O saturation.

Performance modeling

# of experiment	Output frequency
1	1
2	2
3	4
4	8
5	16
# of time steps (duration)	16
# of processes	24
Domain size	20000 x 20000
Metrics	CPU usage, I/O usage

Table 5. Advection performance modeling parameters

For a better understanding on the parameters affecting this simulation, some extra tests have been run. As facing an I/O-bound simulation, the objective is to check the behaviour of the test case with different I/O demand. The size of the domain remains fixed while the frequency for writing to disk is decreased. For a total of 12 time steps, the frequency values are 1, 2, 4, 8 and 16. The complete list of parameters is shown in Table 5.

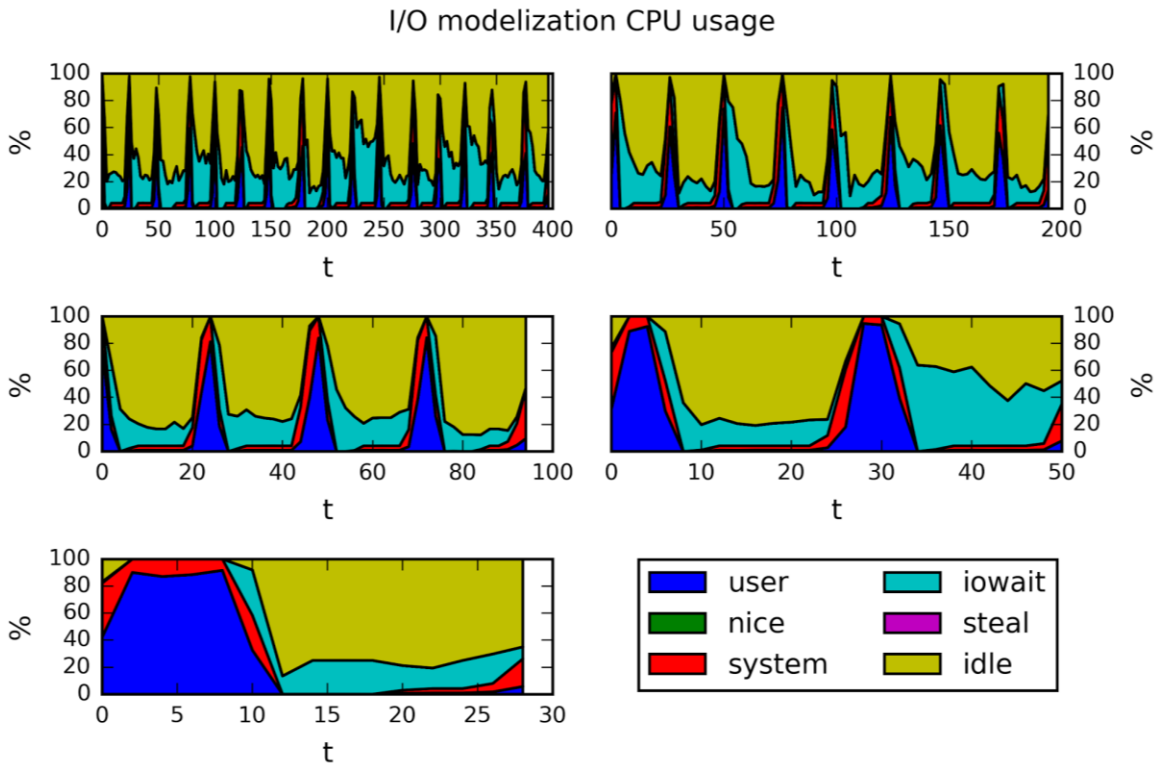


Figure 16. Advection CPU usage in performace modeling

The Figure 16 shows the CPU usage during the simulation phase[16] for the different frequencies. Even decreasing the frequency of output, the simulations remain I/O-bound. For the most relaxed frequency, which only writes once, the CPU user is dominant only in less than the 50 % of the simulation time.

[16] Framework setup and initialization times have been discarded in performance modelling for all the test cases.

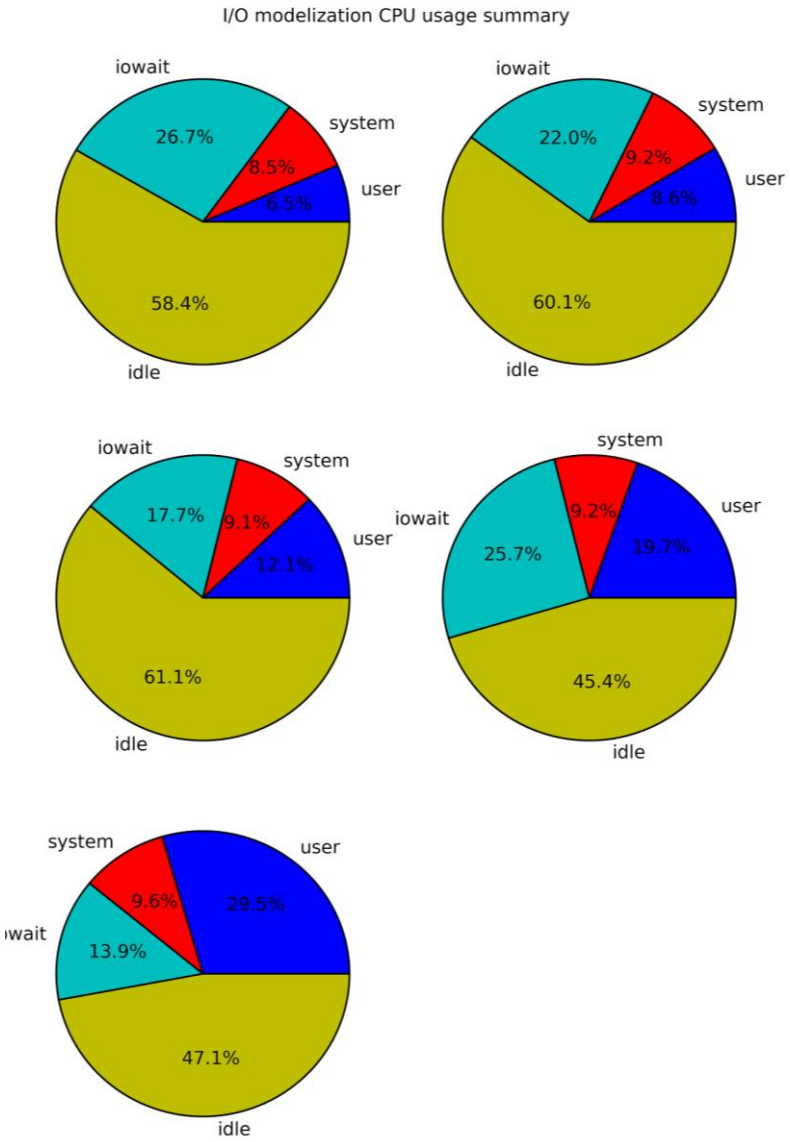


Figure 17. Advection CPU usage summary in performance modeling

The Figure 17 shows the summary of data represented in the Figure 16. Although the I/O still blocks the system, the user time is increasing as the number of output decreases.

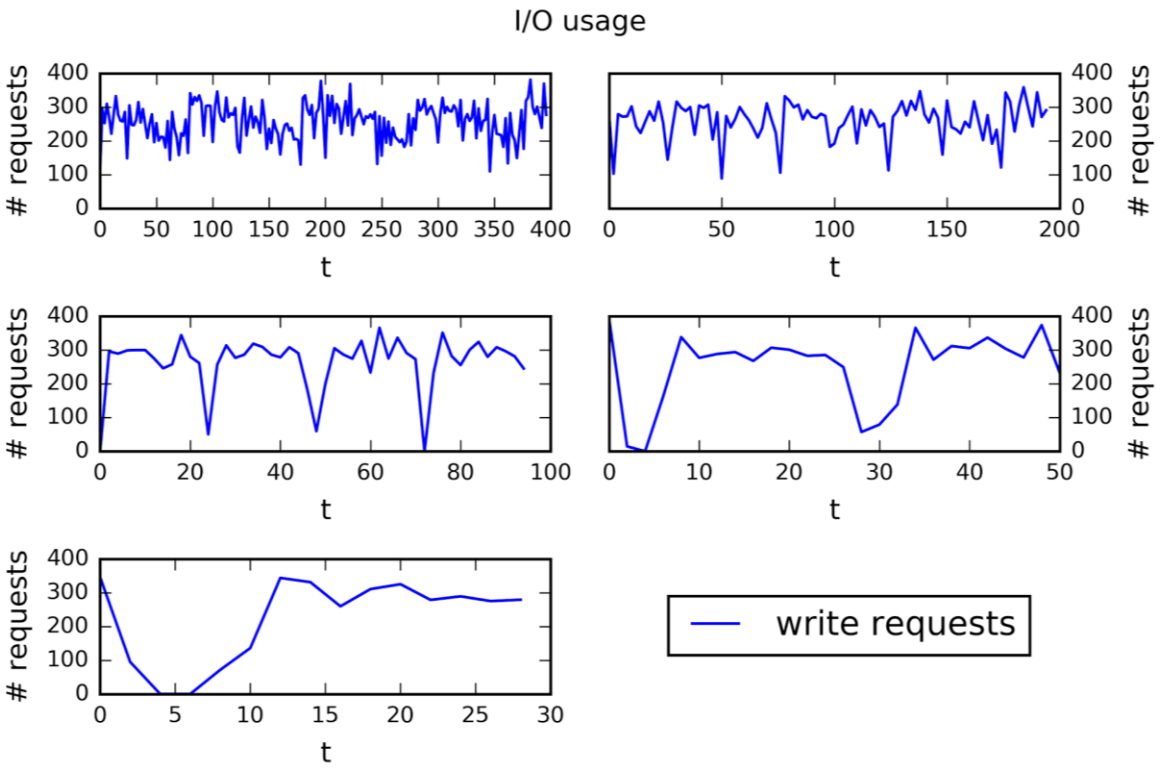
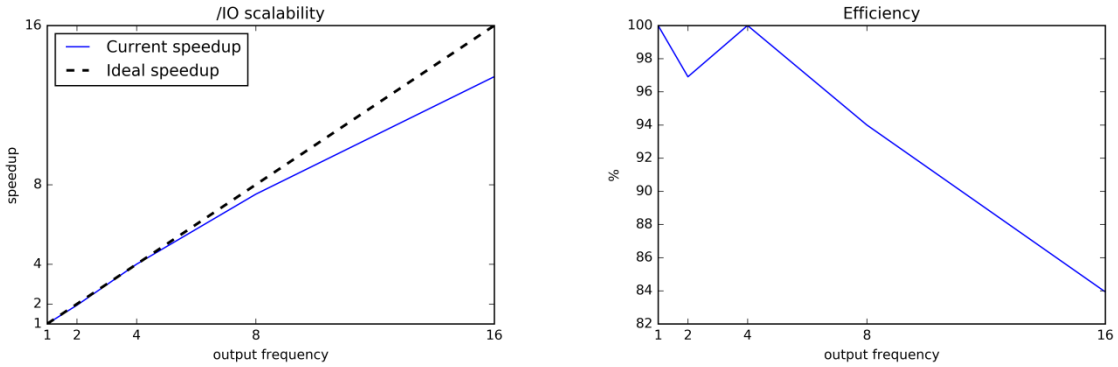


Figure 18. Advection I/O write requests in performance modeling

The write request profiles still show a relatively constant demand of disk writes. This is a confirmation to the thesis that this simulation is I/O bound, even when reducing the output frequency. Nevertheless, the elapsed time changed significantly in the different runs.



a. Scalability

b. Efficiency

Figure 19. Advection I/O scalability and efficiency

With a similar idea than classical CPU scalability tests, the Figure 19.a and Figure 19.b were generated. The I/O scalability was supposed to check how much scalable is the I/O, and the result is astonishing. Changing the frequency have notable consequences, the speedup is quite near to the ideal speedup. This fact reflects that the simulation test is completely I/O-bound. Without changing the domain size and the algorithm complexity, the simulation time is reduced proportionally to the output frequency. At some point it is expected the frequency to saturate and not offering any gain to the simulation time, though it was not found in these tests. However, the limit would appear when the writings were separated enough for the system to deal with one write without overlapping with the next write.

Conclusion

The Advection simulation test is a completely I/O-bound simulation. Apparently, despite the domain size is large, it does not visibly affect to memory performance. Nevertheless, when writing the output to disk, the domain size affects critically to the performance of the system. The simplicity of the computations is also helping to the saturation; the simulation computations run so speedy that the system cannot end writing the previous step output when it receives another writing request.

The suggestions to remove or reduce the bottleneck are:

- Using a system with a higher I/O throughput rate.
- Using a system with a distributed architecture, in which the processes use different nodes with separated disks in order to share I/O accesses.
- Output lighter or less frequently snapshots from the simulation as seen in the performance modeling.

Test case 2 Euler

The Euler Vortex simulation test has high number of stored in memory variables and a complex algorithm, with the most demanding computation needs in this project. Consequently, a CPU-bound run is expected.

The following graphics show the behaviour of the Euler test case.

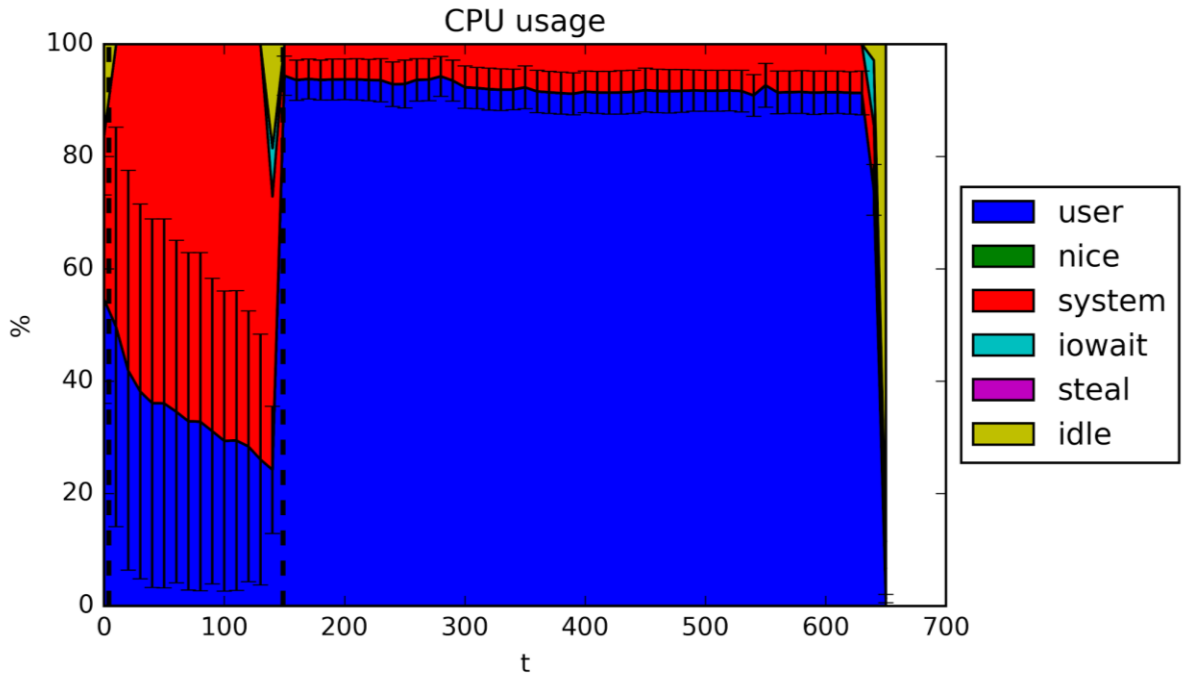


Figure 20. Euler CPU usage

The CPU usage figure shows variability on user usage during initialization phase. The error bar sizes show a relevant variability during that phase, with processors working in the user space at different rates. This variability implies a relative poor efficiency of the algorithm during initialization.

In contrast, when the evolution phase starts (second vertical dashed bar), the CPU usage rate is extremely high (near 90%) and remains constant until the end of the simulation. Then, the evolution phase can be considered high performing in terms of CPU usage. Although the CPU usage is not 100%, the access to memory registers also spend the 10%, a 90% CPU user time can be considered a CPU-bound simulation.

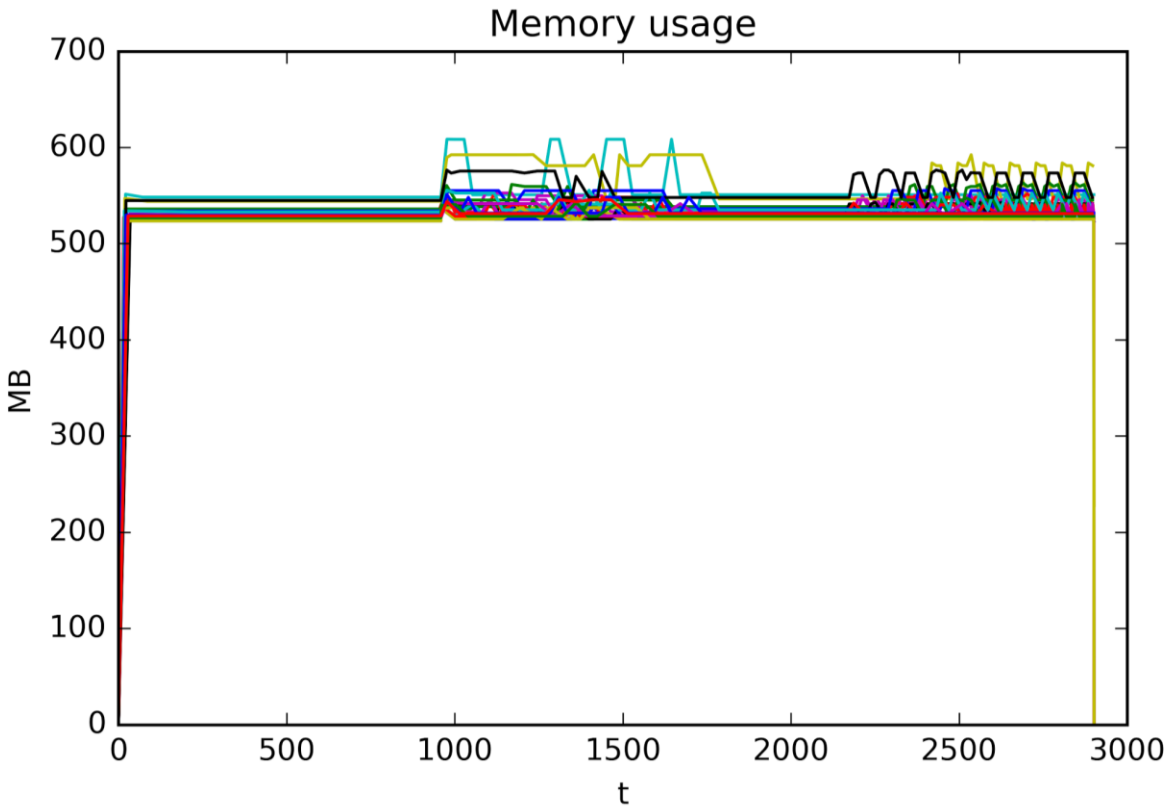
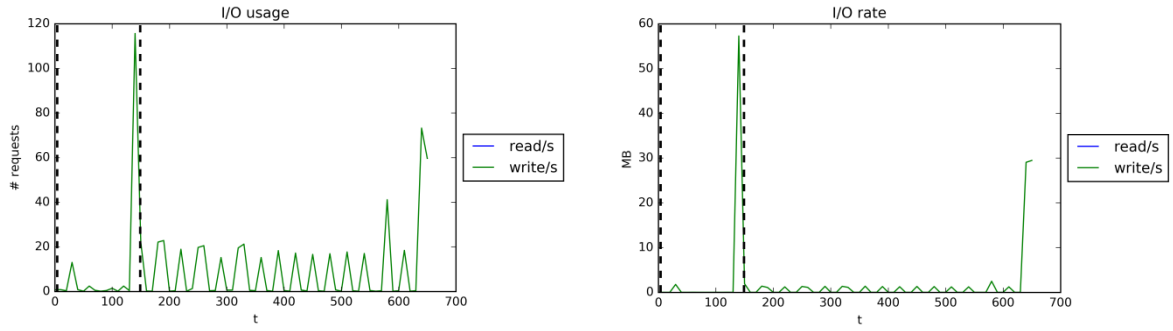


Figure 21. Euler memory consumption

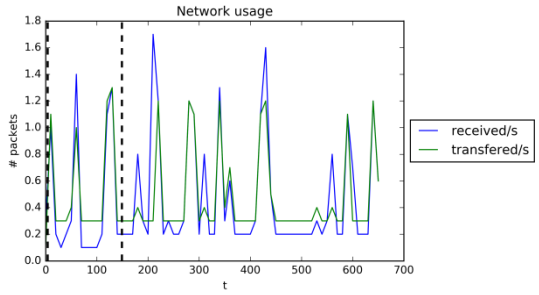
Similar to the Advection test case, from the very beginning of the simulation, all the variables gets to the 500 MB. There are perturbations in Figure 21 for some processes in the middle and at the end of the simulation. Studying the Massif raw data in detail, those perturbations are due to data communication between processors. The processes running the central parts of the domain requires larger communication buffers, this could be the reason for a higher memory demand in some of the processes.

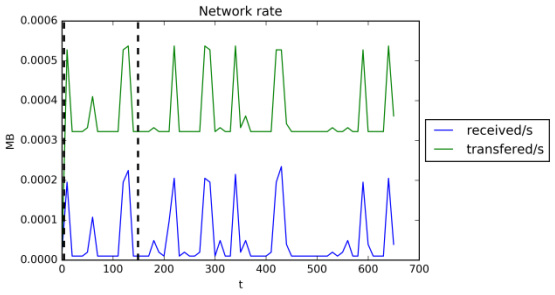


a. I/O usage **b. I/O rate**

Figure 22. Euler I/O metrics

The writing pattern of this simulation is not saturating, both the amount of requests and the I/O rate is negligible.

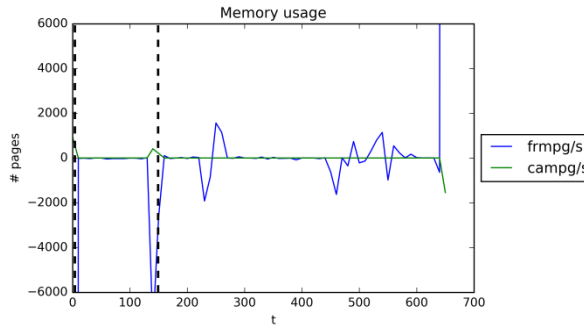


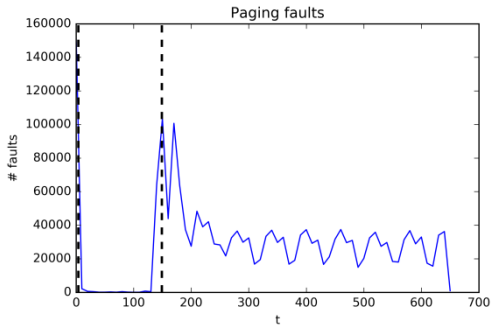


a. Network usage **b. Network rate**

Figure 23. Euler network metrics

The network usage is also negligible. Despite having more variables to communicate than the other tests, the size of the domain is much lower. So communication is not a bottleneck.





a. Memory usage **b. Paging faults**

Figure 24. Euler memory metrics

The memory usage does not show a pattern or size to be concern. There is a peak of memory allocation just before the evolution phase, which is due to a write to disk. But in general, it does not show any profile that could affect to the simulation performance.

The paging fault rate shows a periodical pattern during evolution, having values between 20000 and 40000 faults per second. This constant paging fault rate can be considered normal due to the nature of the simulation. The algorithm computes over a wide set of variables, accessing the arrays at every position in a relatively short time. The cache memory needs to be constantly loading

the proper registers, resulting in this constant rate. However, the system is able to fulfill this request and does not seem to saturate excessively the simulation.

Event	Count	Events/ins	Events/s	Extra
cache-references	3.522.270.172	0,36 %	5,38 M	
cache-misses	1.237.187.524	0,13 %	1,89 M	35,125 % of cache refs
mem-stores	110.211.636.380	11,22 %	168,34 M	
L1-dcache-loads	308.795.201.413	31,43 %	471,65 M	
L1-dcache-load-misses	25.636.359.337	2,61 %	39,16 M	8,3 % of L1 refs
FLOPS	409.563.438.375	41,69 %	625,56 M	
instructions	982.516.219.892		1.500,68 M	0,68 IPC

Table 6. Euler Perf metrics

The data gathered from Perf upholds the paging fault rate presented in Sar graphics. The cache misses and L1 cache load misses rates are the highest of the three test cases. Watching only at Perf data, it seems that this simulation has worse performance than the others. However, these miss rates are constant over time, so the distribution does not drive to a memory-bound run.

The FLOP rate is extremely significative, since the 42 % of CPU instructions are floating point operations, which is the full core of the simulation.

Nevertheless, the IPC is low, 0.68 instructions per cycle. This rate induces to think that there is some margin to improve the simulation.

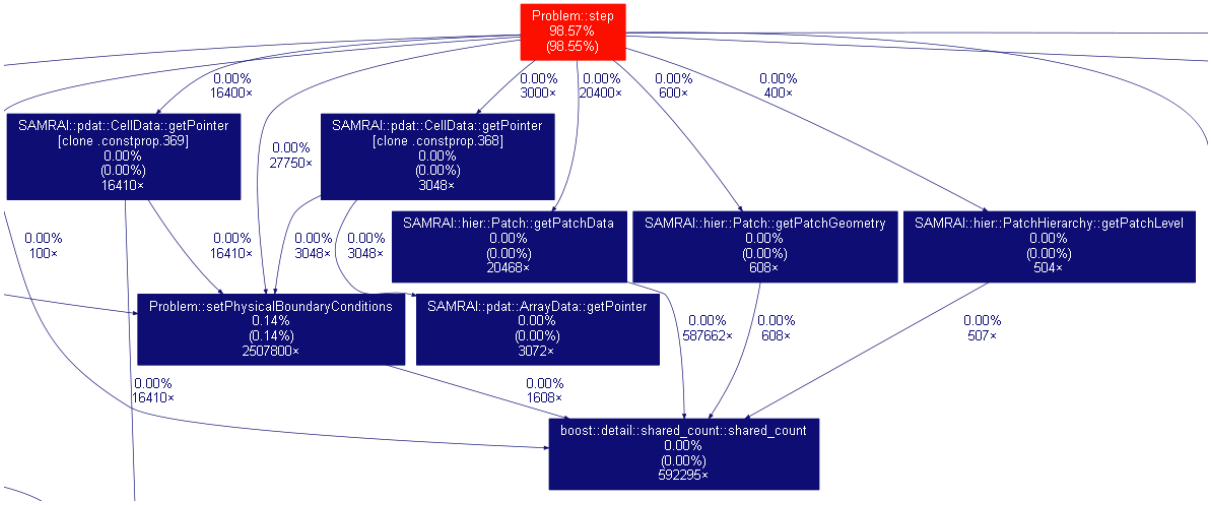


Figure 25. Euler remarkable call graph extract

The Figure 25 shows an extract of the function call graph where the whole simulation time concentrates in the evolution (Problem::step), 98%. Although the extract above does not show the complete graph, the functions called within the evolution does not takes more than 0.20% of total time. This means the computation instructions take most of the time, proving the simulation to have a great performance and being CPU-bound. Consequently, from implementation concern, the code does not need to be dramatically improved, since it is taken a proper calculation rate, with low overhead.

Performance modeling

# of experiment	# of processes
1	1
2	2
3	4
4	8
5	16
# of time steps (duration)	50
Output frequency	50
Domain size	5000 x 5000

Metric	CPU usage

Table 7. Euler strong scalability parameters

# of experiment	# of processes	Domain size
1	1	1000 x 1000
2	2	1000 x 2000
3	4	2000 x 2000
4	8	2000 x 4000
5	16	4000 x 4000
# of time steps (duration)	50	
Output frequency	50	
Metric	CPU usage	

Table 8. Euler weak scalability parameters

In order to have a better understanding on the parameters affecting this simulation, some extra tests have been run. When confronting a CPU-bound simulation, running scalability tests challenge the CPU usage. A strong and a weak scalability tests were executed and the following results were obtained.

For the strong scalability (Table 7), the size of the domain remains fixed, having used the same size as the original simulation test. For the weak scalability (Table 8), the size of the test is set depending using the number of processors as the factor. For instance, the domain size of the 4 processor test is four times the size of the uniprocessor test.

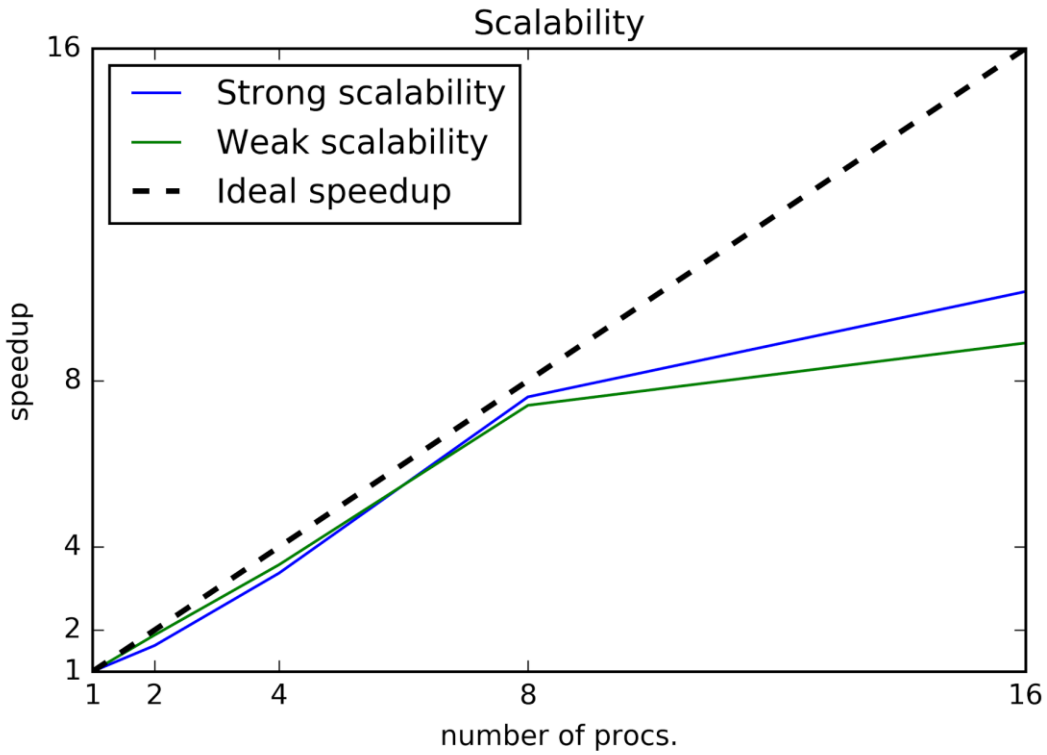


Figure 26. Euler CPU scalability

From the Figure 26, the first conclusion is that the simulation is not scaling at 16 processors at all. The main reason for this lost of performance is due to the fact that from the 24 processors, only 12 are physical and the others exists because of Hyper-threading. When using Hyper-threading, each processor creates another virtual processor for the operative system. However, the two processors share physical hardware, so they are competing for the same accesses to resources (10). Despite Hyper-threading is a good idea for several applications, this is not always valid for SPMD parallel applications, where all the processes used to require access to the same resources simultaneously. Moreover, for this test, a CPU-bound simulation, using Hyper-threading is discouraged (11). Even more, in some cases, Hyper-threading may harm parallel performance though not using more CPUs than physical (12). However, this might be not the case, since it seems to affect to systems under Windows operative systems.

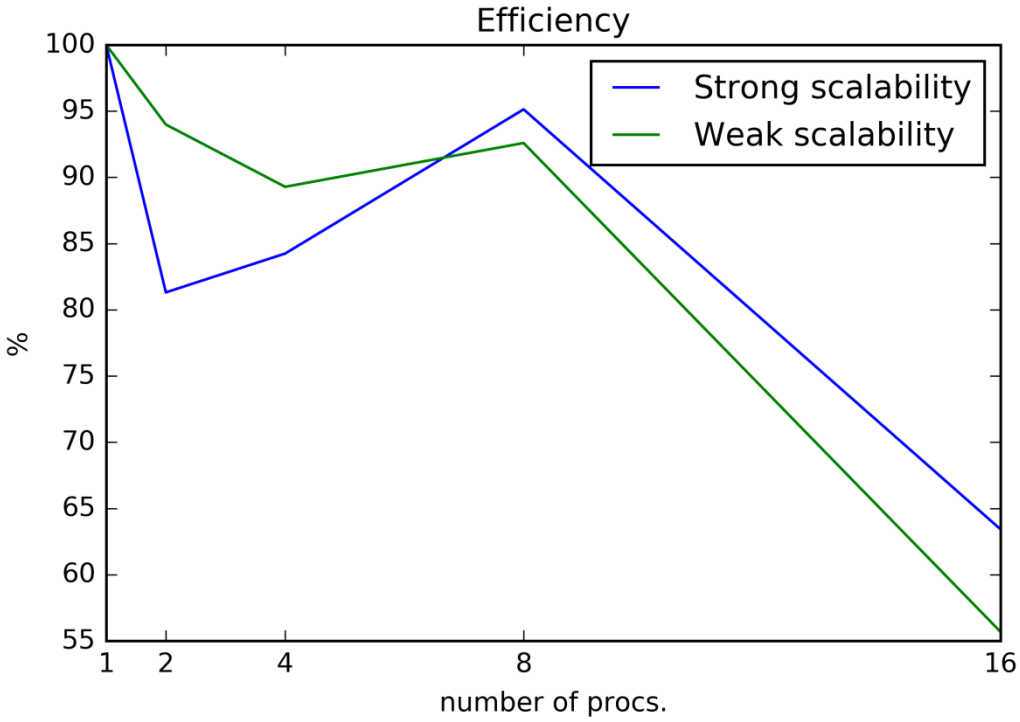


Figure 27. Euler scalability efficiency

As a result, not only the Hyper-threading is affecting the 16 processors test (and also the original 24 processors one), but it also can be influencing the efficiency of the rest, as can be seen in the Figure 27. To calculate efficiency, only the simulation phase has been taken into account, and the formulas are the following (13):

- Strong scaling efficiency

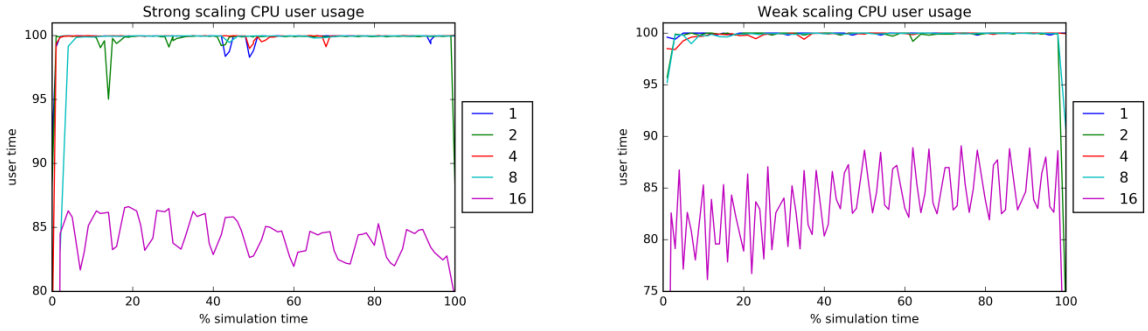
$$\frac{t1}{N \cdot tN} \cdot 100$$

- Weak scaling efficiency

$$\frac{t1}{tN} \cdot 100$$

Where *t1* is the elapsed time for uniprocessor execution, *tN* the elapsed time for *N* processors, and *N* the number of processors.

Apart from performing scalability tests, the CPU usage has been recorded.



a. Strong scalability **b. Weak scalability**

Figure 28. Euler CPU usage in performance modeling

From Figure 28 it can be seen that, but for the non-scaling 16 processor test, the user usage is 100 % most of time at the simulation phase. Consequently, it can be concluded that the simulation is CPU-bound when using physical processors.

Conclusion

The Euler simulation test is a high-demanding CPU simulation. The data yield by the analysis tools fits with the predicted result. The complexity of the algorithm and the moderated domain size leads to a CPU-bound simulation.

To power up the simulation the following actions are possible:

- Run the simulation in more powerful CPUs. The simulation would execute faster.
- Parallelizing between more physical CPUs. However, this solution does not always fit. The scalability of a solution depends on multiple factors, one of them the size of the simulation. There is a limit in which the communication overhead (ratio communication/computation times, ratio surface/volume, Amdahl's law (14)) impacts negatively on the expected speed up (15).

Test case 3 Cash and Goods

The Cash and Goods simulation test has a relatively large graph, a limited number of variables and a straightforward algorithm. The output frequency is high; in the computation phase it yields a graph snapshot every time step. This behaviour should be I/O-bound. However, the analysis described below shows a different bound.

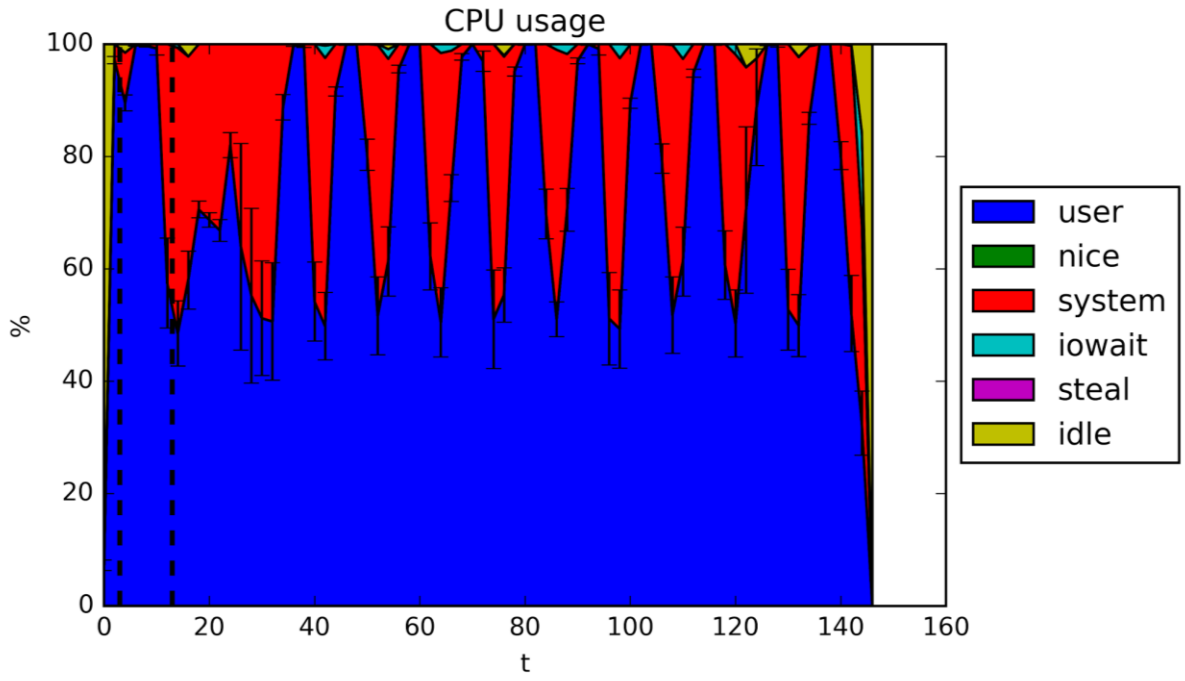


Figure 29. Cash and goods CPU usage

The CPU usage graphic shows an unexpected result. Formerly, the I/O resource was thought to be the bottleneck of the system due to the simplicity of the algorithm and the fast execution of the computational steps. Nonetheless, the iowait values are negligible. Knowing that, one could expect the CPU work at full user usage, however, there seems to be a system process blocking the CPU periodically, at the same time the output is stored in disk.

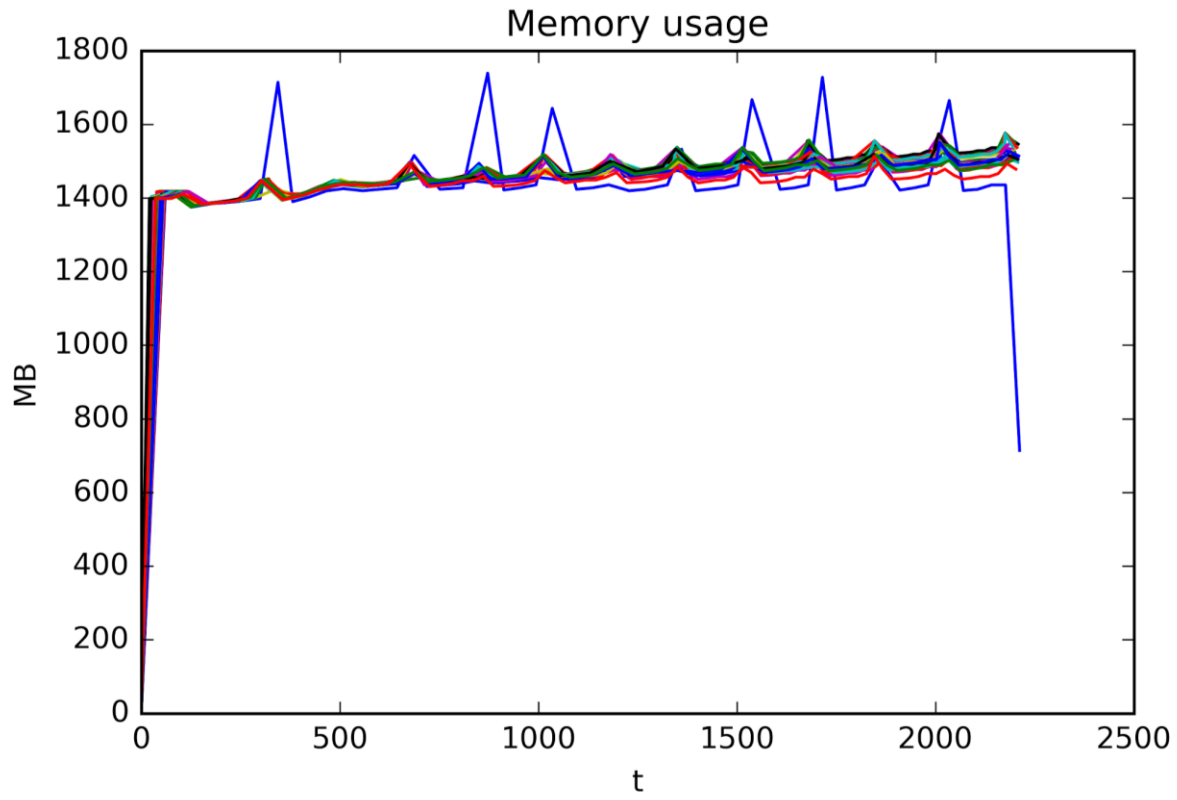
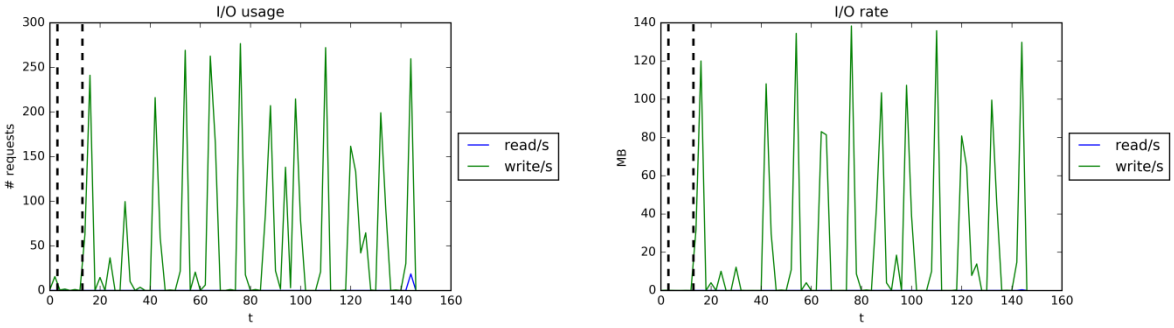


Figure 30. Cash and goods memory consumption

Observing Figure 30 there are two unexpected isues. The first one is the amount of memory taken by all processes, up to 1.4 GB and increasing. According to Massif manual, the tool records the Virtual Memory of the processes (16). The Virtual Memory does not need to reflect the real memory usage (resident memory), which was nearly a 20 % of Virtual Memory according to command line Top information. The Virtual Memory reflects the potential size of the execution, which may be not reaching (17). This excessive size could be generated by a memory leak in Boost graph library, concretely in graph creation time, which takes the 85 % of the memory.

The second issue in the graphic is the memory peaks appearing in only one of the processors. Crawling into Massif raw data, the extra memory taken by this process correspond to the disk writing process. Differently to the previous tests, the framework governing the Cash and Goods simulation is generating only one output file with all the graph data, while the other simulations generate one file per processor. As a result, one of the processors must gather the entire graph in memory prior to writing it to disks. This behaviour explains the memory peaks in only one process.

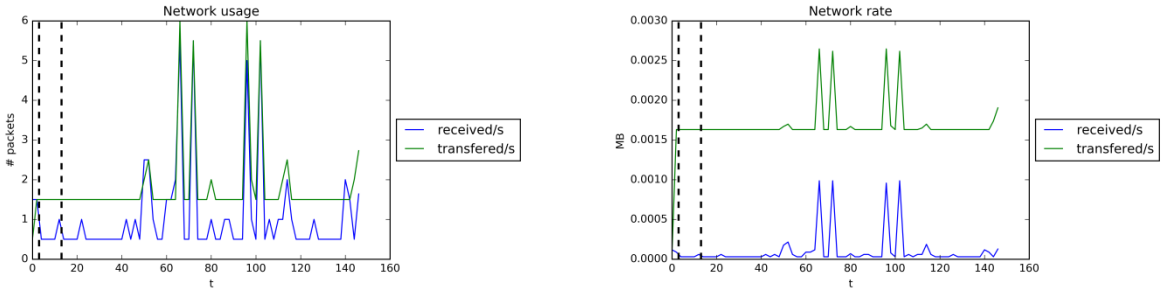


a. I/O usage **b. I/O rate**

Figure 31. Cash and goods I/O metrics

The I/O profiles show a periodic request of this resource. The requests are separated enough as not to block the CPU as it did happen in the Advection test case.

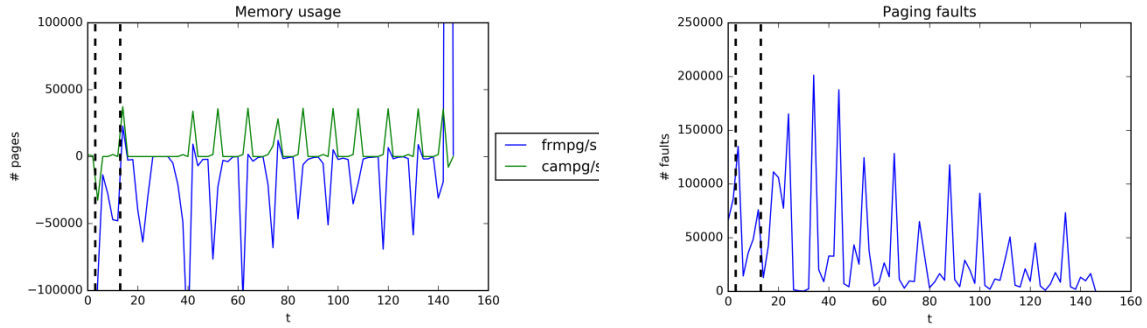
Although the number of requests and rates are similar in both tests, there are two factors that make the difference. The first is the size of the output, 300 MB graphs vs 3 GB shared by 24 files. The second is precisely that in the Advection test, 24 processes are trying to write on disk simultaneously. Meanwhile, in this case only one process requires access to disk, which obviously always get. Then, for this test case, the I/O is not a bottleneck.



a. Network usage **b. Network rate**

Figure 32. Cash and goods network metrics

Like in the other test cases, the communication is not a limiting resource. The graph nodes to communicate in comparison with the ones only used inside the local subgraph is extremely low, so communication overhead is negligible.



a. Memory usage **b. Paging faults**

Figure 33. Cash and goods memory metrics

The memory usage data of Figure 33.a is showing a periodical pattern that indicates the system is trying to allocate a large number of cache pages at the same time (18). Those peaks are repeated 10 times, at the same points as snapshot writing. Examining the CPU usage at the same time as one of those peaks and knowing that one of the processes has an extra demand on memory showed in the Massif graphic, there can be deduced that only the process in charge of writing to disk is allocating the pages. This unbalanced behaviour creates a bottleneck in the system related to memory access.

The paging fault rate also shows periodical peaks, which can lead to similar conclusion.

Event	Count	Events/ins	Events/s	Extra
cache-references	422.856.284	0,21 %	2,92 M	
cache-misses	129.333.986	0,06 %	0,89 M	30,586 % of cache refs
mem-stores	35.152.031.935	17,05 %	242,58 M	
L1-dcache-loads	67.372.289.327	32,67 %	464,93 M	
L1-dcache-load-misses	3.979.035.538	1,93 %	27,46 M	5,91 % of L1 refs
FLOPS	1.047.181.496	0,51 %	7,23 M	
Instructions	206.215.827.236		1.423,08 M	0,65 IPC

Table 9. Cash and goods Perf metrics

Perf data is shared between all the processes, so detecting an unbalanced pattern like the one detected using Sar and Massif tools is not straightforward. Probably, generating a Perf output

for every single process would have shown an unbalanced event, as the pattern present in the previous graphics.

The cache and L1 cache miss rates are slightly lower than Euler test case and do not show any special issue.

As expected, the number of FLOPS is the lowest of all test cases, reflecting the simplicity of the algorithm.

Without being excessively outstanding, the memory stores event has the highest rate. Nonetheless, it seems not relevant enough to extract any conclusion.

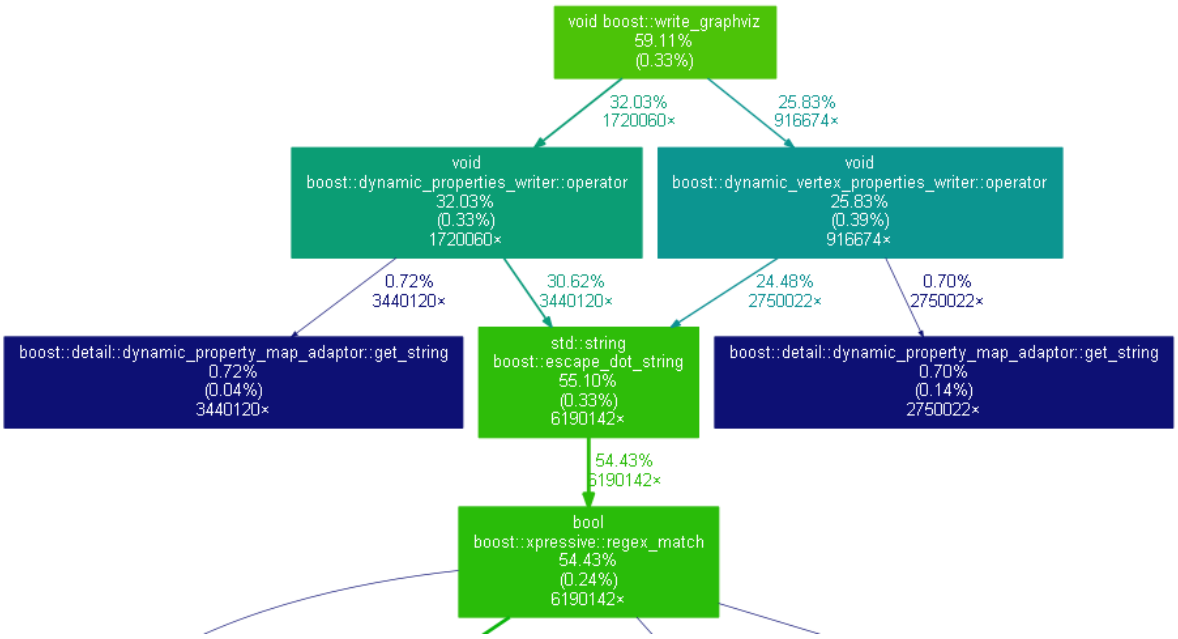


Figure 34. Cash and goods remarkable call graph extract

In the Figure 34, there can be seen the maximum function, by far, spending time in this simulation, write_graphviz. Writing the output takes a 59% from user time. However, going into detail, the inner functions spending that time are Boost Xpressive calls. The Xpressive library is a regular expression library, which seems to saturate in this problem. When executing write_graphviz all the data has already been communicated, so the process executes locally. Then, the saturation must be result from CPU or Memory saturation. Due to the previous results from Sar and Massif, it seems the size of the graph is too heavy, so memory is the bottleneck in this case.

The summation of computational routines takes approximately a 28% of the CPU time and communication 15%. So writing output routines (59%) nearly doubles the time computing and

communicating. In a proper simulation, computation time should clearly dominate over communication and I/O routines.

In Massif output there was discovered a process with a different behaviour from the others. This process was the one in charge to write the output file effectively to disk. The CPU usage distribution changes a bit from the rest of processors, spending 8% in computational routines, 29% in communication and 58% in I/O routines.

Performance modeling

The source of this memory-bound simulation is not absolutely clear. Surely, the size of the graph plays an important role in the bottleneck. However, the memory peaks occurring at the same time than the output and the information coming from function profiling make us think if the I/O frequency is also a concern in this simulation. To discover the reason, two sets of tests have been performed.

# of experiment	# of nodes (size)
1	2000000
2	1000000
3	500000
4	250000
5	125000
# of steps (duration)	10
Output frequency	1
# of processes	24
Metrics	CPU usage, memory usage

Table 10. Cash and goods performance modeling variable size parameters

The first tests consist in a variability of the memory size, reflected by the number of graph nodes. Starting from the original size, the factors of 2, 4, 8 and 16 are applied to reduce the number of nodes in the graph.

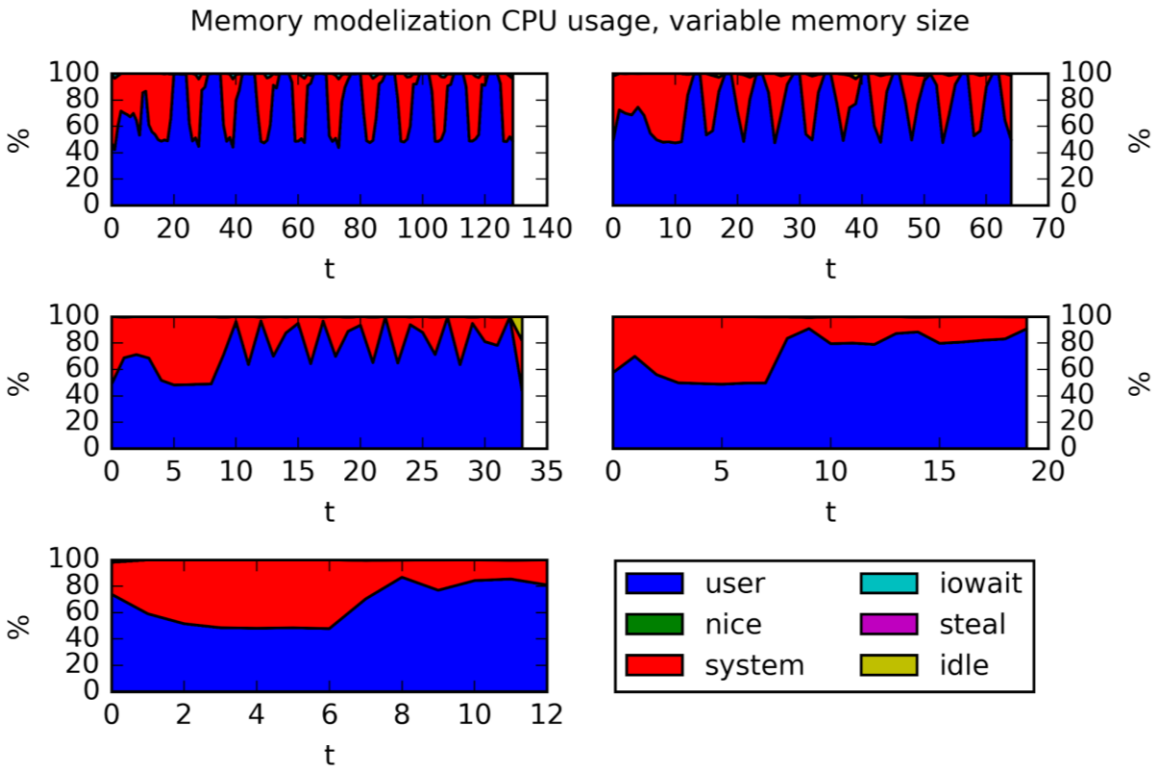


Figure 35. Cash and good CPU usage in performance modelling, variable size

From the CPU usage in Figure 35, starting from bigger size on the top left up to the minimum size in the bottom, an increase of the average user usage can be seen at the end of the simulation. The biggest sized simulation has peaks reaching the 50 % user usage when the memory problems appear, alternating with a 100 % usage. Reducing the size has an averaging effect, driving the usage up to the 80 %.

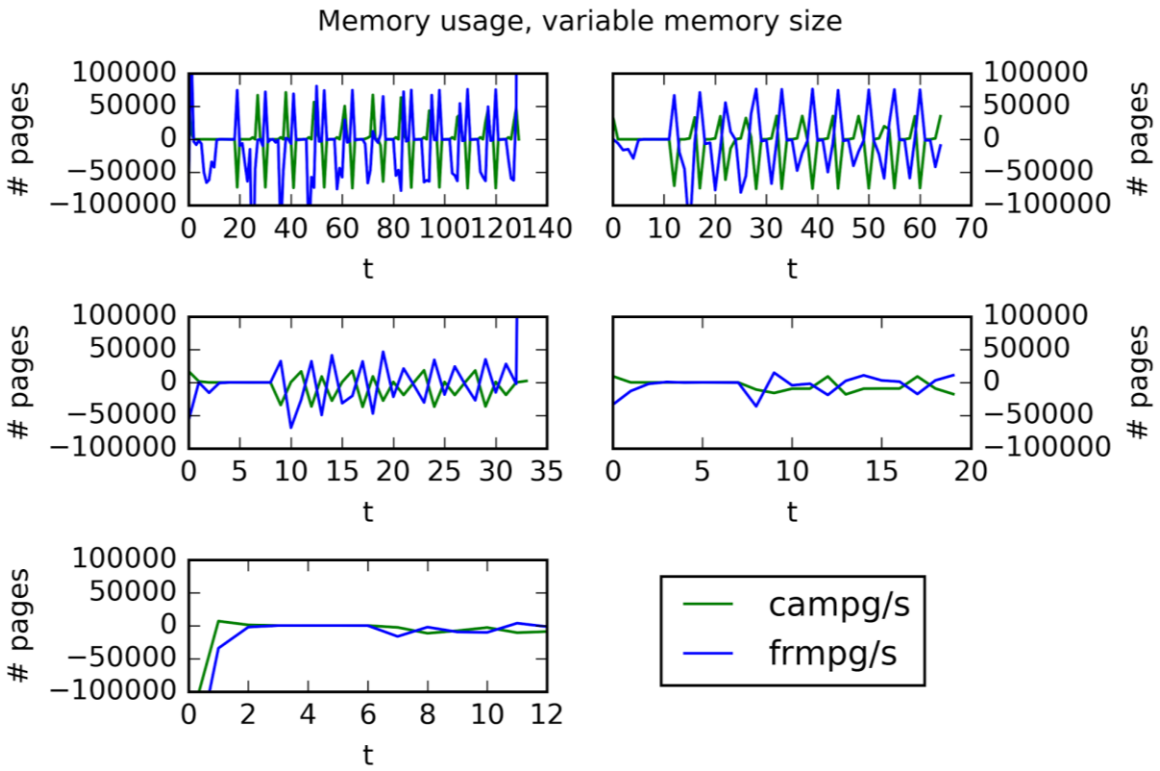


Figure 36. Cash and goods memory usage in performance modelling, variable size

The memory usage graphics clearly expose that memory size is a cause of the bottleneck for the original simulation. As the memory size is reduced, the periodical peaks reflecting the allocation of large number of pages are also considerably reduced. The reduction of the allocation peaks influences directly to the execution times, as can be seen in the graphic below.

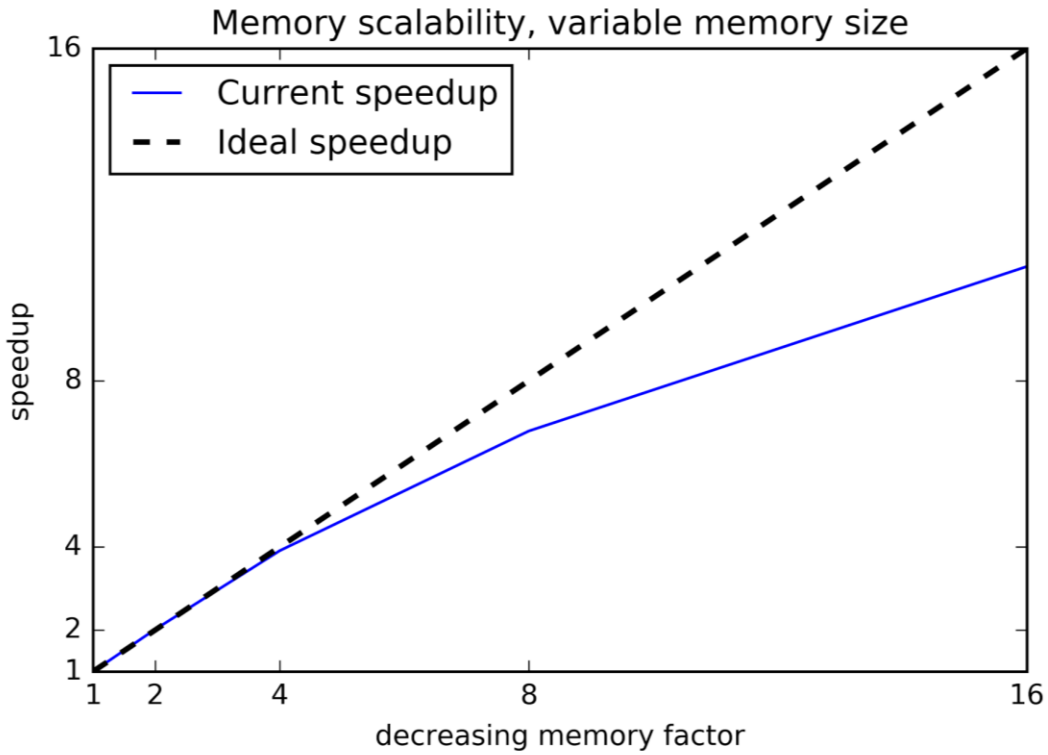


Figure 37. Cash and goods memory scalability, variable size

The scalability of the simulation when reducing the memory size is significant enough to show that the original simulation is memory-bound. Nevertheless, reducing memory maintaining the same number of processes will lead to a saturation when the ratio memory-per-process/memory-to-communicate reached certain low value, turning the simulation from memory-bound to communication-bound.

# of experiment	Output frequency
1	1
2	2
3	4
4	8
5	16
# of time steps (duration)	16
# of nodes (size)	2000000

# of processes	24
Metrics	CPU usage, memory usage

Table 11. Cash and goods performance modeling variable output frequency parameters

The second set of tests has the objective to check the impact the I/O frequency has in the memory-bound simulation. The peaks in CPU system usage and high allocation demand seem to occur at the same time than the write to disk. Fixing the problem size and testing different output frequencies (1, 2, 4, 8 and 16) must help in the simulation modelization.

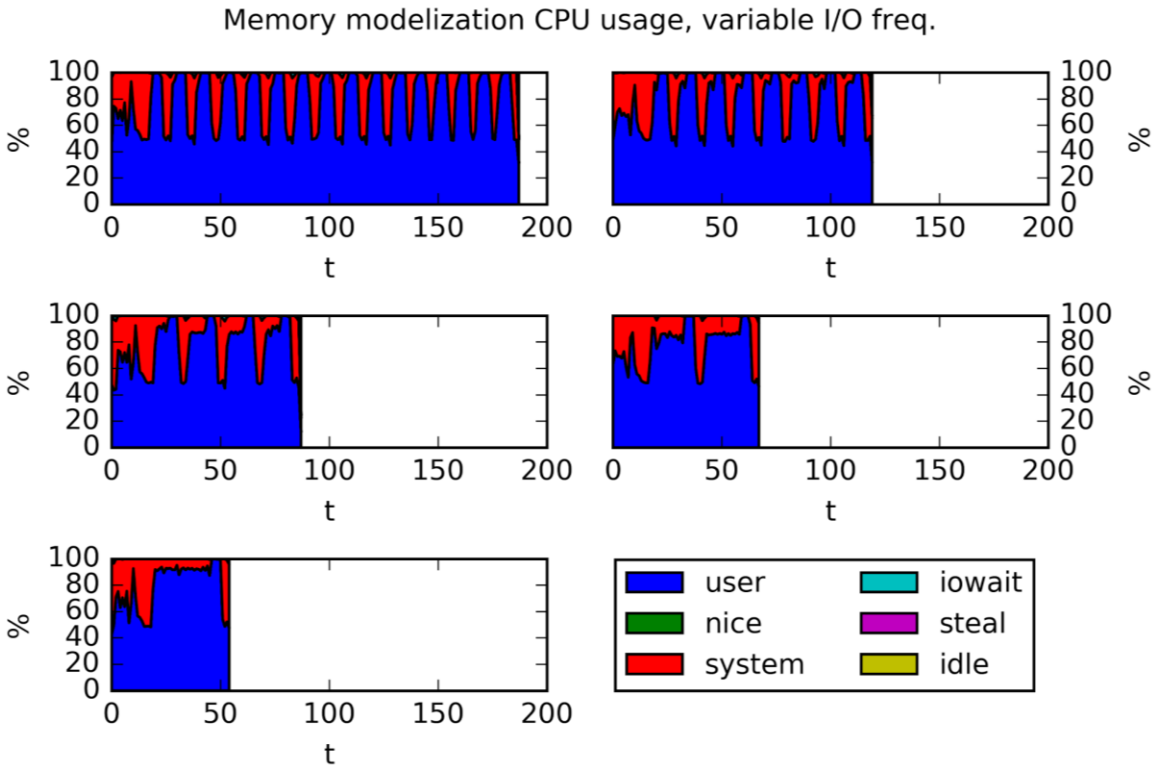


Figure 38. Cash and goods CPU usage in performance modeling, variable output frequency

The CPU usage (scaled in time for better visual comparison) show that decreasing the output frequency does not significantly changes the shape of the graphs when writing (red peaks). Each time disk write is requested the system usage takes the 50 % of the CPU. Providing the I/O was affecting significantly to the memory bottleneck, a decrease in elapsed time must be seen in the simulations. The elapsed time decreases, althoug not enough to be considered the main cause of the bottleneck.

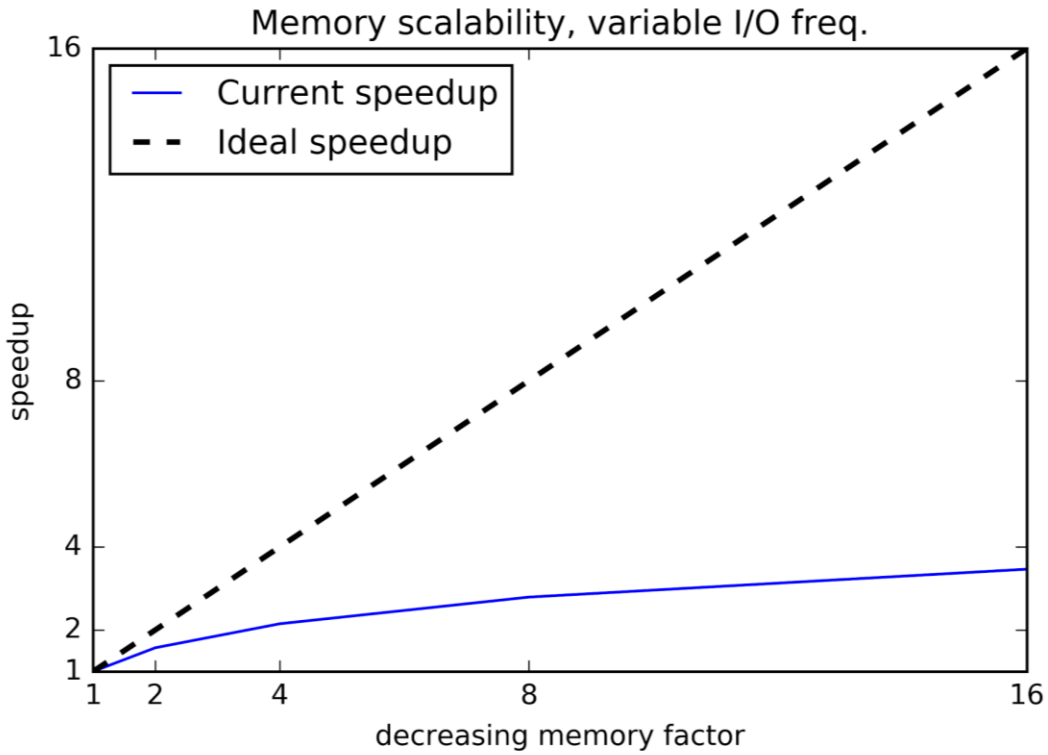


Figure 39. Cash and goods memory scalability in performance modeling, variable output frequency

The Figure 39 shows that reducing the frequency does not really improve the performance of the simulation. Surely, reducing the output frequency affects the elapsed time, but that is because less work must be done. To be considered a cause of the memory-bound a different graphic was expected, but this graph immediately saturates, the maximum speedup achieved is less than 4 when using 16 processors. Then it can be inferred that memory is still commanding the simulation.

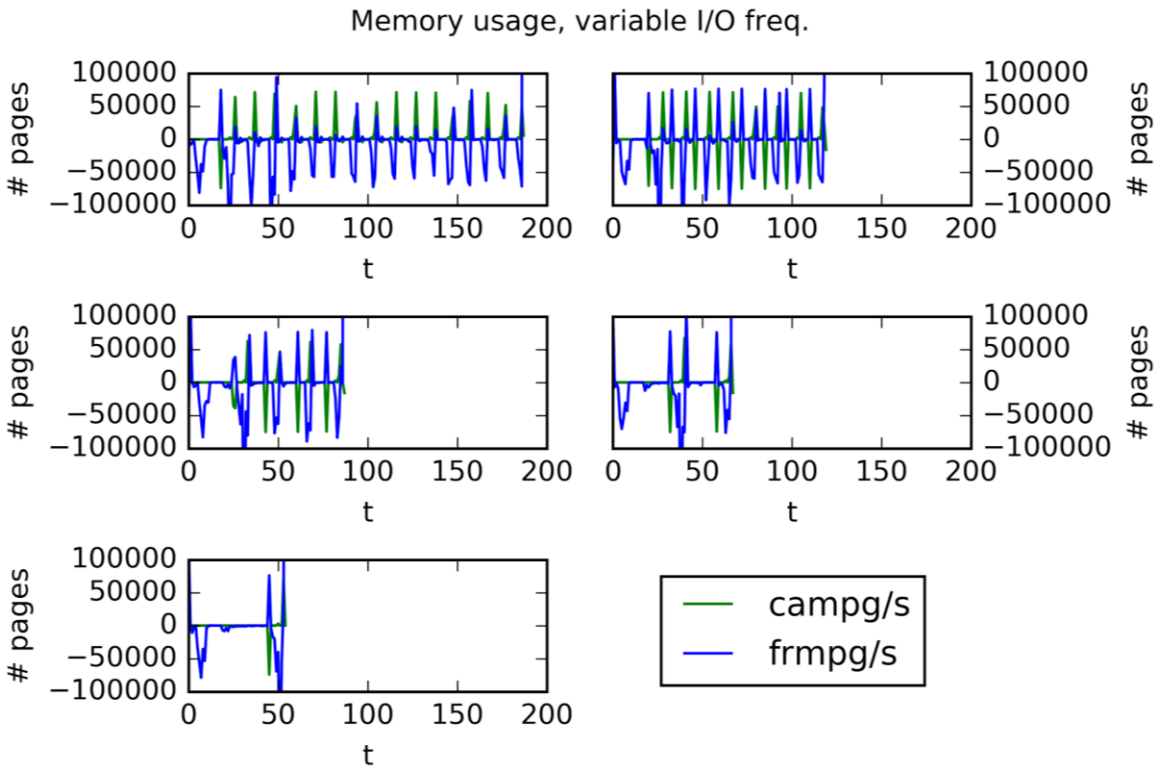


Figure 40. Cash and goods memory usage in performance modeling, variable output frequency

The Figure 40 confirms the conclusions from this modelization recently stated. The allocation peaks are not quantitatively reduced. Every write request requires extra memory pages in the same quantity, independently when there are less or more output requests.

Summarizing, the modelization has shown that the main cause of the memory-bound problem is related with the problem size, and not really to the output to disk.

Conclusion

The cash and goods simulation rise to be memory-bound. There were found hints that seemed to point to the I/O process, especially to one of the processes gathering all the data prior to writing to disk. All the same, the simulation modelization proved that the allocation required size was the main source of the problem and I/O was not enough significantly.

The possible solutions to reduce memory-bound could be:

- Move to a system with high cache read rate or higher cache capacity. Even to a system where each node has its own RAM memory.

- Despite not being the bottleneck, it would be interesting to change the output pattern to balance the disk writing between all the processes. Every process should be able to generate a separated file. It could improve the simulation when using high frequency output with a factor of approximately 4, as seen in the modelization.

Conclusions

Measuring applications is not only a mean to gauge performance, but also is a good practice to learn about the insights of applications. By the usage of metric tools, unexpected behaviour and problems may emerge, as occurred in this project.

This section shows the conclusions and lessons learned regarding tool selection, the simulation test cases and project management.

Tool selection

An appropriate selection of the metrics is fundamental and should be the initial step for a performance study. After the metrics were established, the study of the tools led to the necessity of using several tools, and that decision was the key to unravel hidden problems, as Cash and Goods memory issues.

As a matter of fact, using only one performance tool would not have been enough. For instance, Perf does not show temporal profile, hiding periodical issues as in Advection or unbalanced pattern as in Cash and goods. Also, Massif granularity has manifested an unknown behaviour regarding writing output files.

Moreover, it is a good practice to use different tools to consolidate the results. Some of the tools overlapped results, strengthening the conclusions.

Apart from using a wide variety of tools, the granularity is essential. Tools providing deeper levels of detail as call graphs are critical to find delicated issues. Unpredicted results can rise from hottest functions in tools like Gprof. And also granularity in terms of individual process data recording is crucial. Despite seeming not necessary in SPDM simulations (all the processes executes the same program), an unbalanced behaviour was detected due to processor granularity.

Simulation results

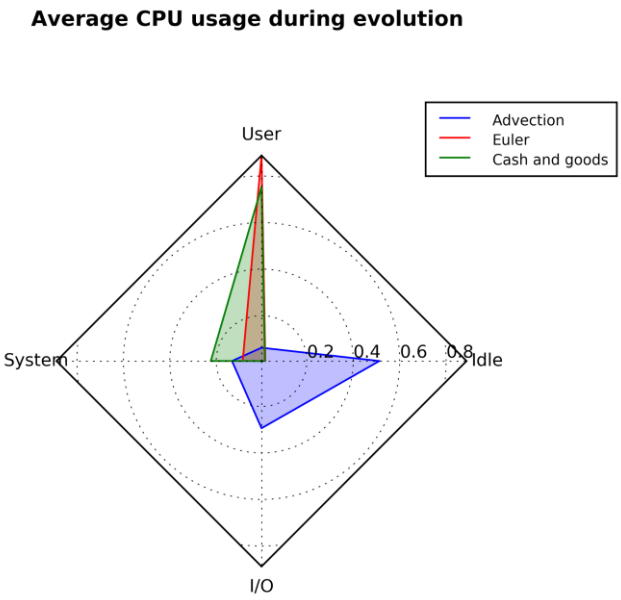


Figure 41. Average CPU usage

Initially, the three test cases, Advection, Euler, and Cash and goods, were thought to be Memory, CPU and I/O-bound respectively. In Figure 41 it is shown the average CPU usage during the simulation evolution phase. Advection resulted to be I/O-bound, Euler was CPU-bound as expected and Cash and goods, not being completely memory-bound, showed memory saturation periodically.

Studying the performance of these simulations has driven to deeper thoughts of their characteristics. How simulations behave under different circumstances, trying to reach the limits oversizing the memory or I/O capacity, and how to detect those bottlenecks have been two of the main pieces of knowledge from this project.

Project management

During the project, the initial plannification had sufferend a significant change. As can be seen in the Annex II – Gannt diagram, initial planification and tracking, the modelization of the three simulation test cases was not initially planned. Despite the metric documentation and implementation had suffered a slight delay, the result analysis was completed early enough to add extra studies. The addition of the new task, modelization, has been an excellent way to complete the project, aiding to consolidate the characterization results.

The result analysis was a bit overestimated, which in the end allowed us to add new content to the project.

Regarding the project objectives, fortunately, all of them have been achieved. A deep knowledge of the simulations has been obtained, allowing creating realistic performance proposals. The identification of clear bottlenecks was also accomplished, with solid results coming from the modelization scalability tests.

Finally, from the personal point of view, the knowledge obtained during the project was instantly and parallely applied in current performance studies driven in the author research institute.

Glossary

Agent based model	Computational model for simulating complex systems.
Bottleneck	A resource is considered a bottleneck when it is severely affecting an application performance.
Cache hit	A cache hit is a state in which data requested for processing by a component or application is found in the cache memory. It is a faster means of delivering data to the processor, as the cache already contains the requested data.
Cache miss	Cache miss is a state where the data requested for processing by a component or application is not found in the cache memory. It causes execution delays by requiring the program or application to fetch the data from other cache levels or the main memory.
Characterization	Methodology to generate descriptive characteristics and behaviour of an application.
Complex system	A system composed by different components interacting with themselves. From these interactions, a global behaviour may arise.
Debugging	The process of locating and removing computer program bugs, errors or abnormalities.
Discretization schema	The numerical recipes to convert a continuous physical model into a discrete one.
Domain decomposition	The process of decompose a problem domain into smaller subdomains in order to resolve the problem faster or easily.
Flag	A common way to specify options for command-line programs.

FLOP	A measure based on specific numbers of floating-point operations (or real number calculations).
Ghost zone	An area on a local process with copies of data from adjacent processes. The ghost zone is a strategy that minimizes the communications in parallel domain based simulations.
Heap memory	The portion of memory where *dynamically allocated* memory resides. The programmer is the responsible to allocate and deallocate the memory.
Hyper-threading	Technology that is designed to improve overall superscalar CPU performance by using simultaneous hardware multithreading.
Instrumentation	The ability to monitor or measure the level of application's performance, to diagnose errors and to write trace information.
Memory leak	A process in which a program or application persistently retains a computer's primary memory. It occurs when the resident memory program does not return or release allocated memory space, even after execution, resulting in slower or unresponsive system behavior.
Metric	A software metric is a standard of measure of a degree to which a software system or process possesses some property.
Paging	Paging refers to memory allocation. In a paging memory management scheme, data are stored and managed in identical consistent blocks referred to as 'pages.'
Partial differential equation	PDE is a differential equation that contains unknown multivariable functions and their partial derivatives. PDEs are used to formulate problems involving functions of several variables, and are either

	solved by hand, or used to create a relevant computer model.
Profiling	A form of dynamic program analysis that measures, for example, the space (memory) or time complexity of a program, the usage of particular instructions, or the frequency and duration of function calls. Most commonly, profiling information serves to aid program optimization.
Resident memory	The amount of physical memory currently allocated. A process can have a large amount of virtual memory allocated but still be using very little physical memory.
Scalability	In High Performance Computing, scalability is the ability of an application to efficiently perform when the resources increase.
Simulation Domain	The simulation domain limits the size of a simulation.
Snapshot	A set of files that keep the state of a simulation in a determined time step.
SPMD	Single Program Multiple Data is a technique employed to achieve parallelism. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.
Strong scaling	How the solution time varies with the number of processors for a fixed *total* problem size.
Virtual memory	Virtual memory is a feature developed for the kernel of an operating system that simulates additional main memory. This technique involves the manipulation and management of memory by allowing the loading and execution of larger programs or multiple programs simultaneously.

Weak scaling	How the solution time varies with the number of processors for a fixed problem size *per processor.*

Bibliography

1. **Kaku, Michio.** *Physics of the future.* New York : Doubleday, 2011, p. 416.

2. **Hwu, Wen-mei.** Three challenges in parallel programming. [Online] 2012. http://parallel.illinois.edu/blog/three-challenges-parallel-programming.

3. *Some Computer Organizations and Their Effectiveness.* **Flynn, M. J.** 9, 1972, IEEE Transactions and Computers, Vols. C-21, pp. 948-960.

4. **Wikipedia.** Advection. [Online] https://en.wikipedia.org/wiki/Advection.

5. —. Euler equations (fluid dynamics). [Online] https://en.wikipedia.org/wiki/Euler_equations_(fluid_dynamics).

6. **Nikolic, Bojan.** How to easily measure Floating Point Operations Per Second (FLOPS). [Online] 2016. http://www.bnikolic.co.uk/blog/hpc-howto-measure-flops.html.

7. **Tibirna, Cristian.** Using gprof for parallel profiling. [Online] 2014. http://giref.ulaval.ca/~ctibirna/parallel_gprof.html.

8. **page, Linux man.** Sar(1). [Online] https://linux.die.net/man/1/sar.

9. **Argonne National Laboratory.** gprof Profiling Tools. User guides. [Online] https://www.alcf.anl.gov/user-guides/gprof-profiling-tools.

10. **Mikalauskas, Aurimas.** Hyper-threading - how does it double CPU throughput? *Percona Database Performance Blog.* [Online] 2015. https://www.percona.com/blog/2015/01/15/hyper-threading-double-cpu-throughput/.

11. *An Empirical Study of Hyper-Threading in High Performance Computing Clusters.* **Leng, Tau.** St. Petersburg, Florida : LCI conference on high performance clustered computing, 2002.

12. **Kinghom, Donald.** Hyper-Threading may be Killing your Parallel Performance. [Online] 2014. https://www.pugetsystems.com/labs/hpc/Hyper-Threading-may-be-Killing-your-Parallel-Performance-578/.

13. **ScharcNet.** Measuring Parallel Scaling Performance. [Online] https://www.sharcnet.ca/help/index.php/Measuring_Parallel_Scaling_Performance.

14. *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities.* **Amdhal, Gene M.** 1967, AFIPS Conference Proceedings, Vol. 30, pp. 483-485.

15. **Skinner, David and Antypas, Katie.** Parallel Application Scaling, Performance, and Efficiency. [Online] 2010. https://www.nersc.gov/assets/NUG-Meetings/IntroMPIApplicationScaling.pdf.

16. **Valgrind user manual.** Massif: a heap profiler. [Online] http://valgrind.org/docs/manual/ms-manual.html.

17. **King, Graham.** Resident and Virtual memory on Linux: A short example. [Online] 2012. http://www.darkcoding.net/software/resident-and-virtual-memory-on-linux-a-short-example/.

18. **Smith, Dudley.** Linux CPU cache slowdown. [Online] 2013. http://stackoverflow.com/questions/16900680/linux-cpu-cache-slowdown.

19. **Technopedia.** Technopedia. [Online] https://www.techopedia.com/.

Annexes

Annex I - Library Installation Commands

The current Annex lists the commands used to compile, install and setup the required libraries in the cluster.

HDF5

./configure -- prefix=/home/bminano/software/hdf5 CFLAGS=-fPIC LDFLAGS=-fPIC CPPFLAGS=-fPIC CXX=/opt/gcc/4.6.4/bin/g++ CC=/opt/gcc/4.6.4/bin/gcc

make

make install

SILO

./configure --prefix=/home/bminano/software/silo --with-pic CC=/opt/gcc/4.6.4/bin/gcc CXX=/opt/gcc/4.6.4/bin/g++ --disable-fortran --enable-shared --with-hdf5=/home/bminano/software/hdf5/include,/home/bminano/software/hdf5/lib

make

make install

Boost

The Boost Graph Library in version 1.45 has a bug when running parallel graphs which need incoming information from nodes in different processor. The following line from file boost/graph/distributed/adjacency_list.hpp was replaced

&& !(i->source_processor == source(e, g).owner) && i->e == e.local)

by

&& !((i->e == e.local) && (i->source_processor == target(e, g).owner)))

After the correction, the sources can be configured.

./bootstrap.sh –prefix=/home/bminano/software/bootstrap

Then, in order to provide MPI support and set the GCC version, the file tools/build/v2/user-config.jam was edited to add the following lines:

using gcc : 4.6.4 : /opt/gcc/4.6.4/bin/g++ ;

using mpi : /opt/openmpi/1.6.5/bin/mpicc ;

Then, the library is compiled and installed.

sudo ./bjam install

SAMRAI

./configure --prefix=/home/bminano/software/SAMRAI --with-mpi-include=/opt/openmpi/1.6.5/include --with-mpi-lib-dirs=/opt/openmpi/1.6.5/lib --with-hdf5=/home/bminano/software/hdf5 --without-petsc CXX=/opt/gcc/4.6.4/bin/g++ CC=/opt/gcc/4.6.4/bin/gcc F77=/opt/gcc/4.6.4/bin/gfortran --with-x CXXFLAGS=-fPIC LIBS=-lmpi CPPFLAGS=-fPIC CFLAGS=-fPIC FFLAGS=-fPIC --enable-opt=-O2 --with-boost=/home/bminano/software/bootstrap LDFLAGS=-L/opt/openmpi/1.6.5/lib

make

make install

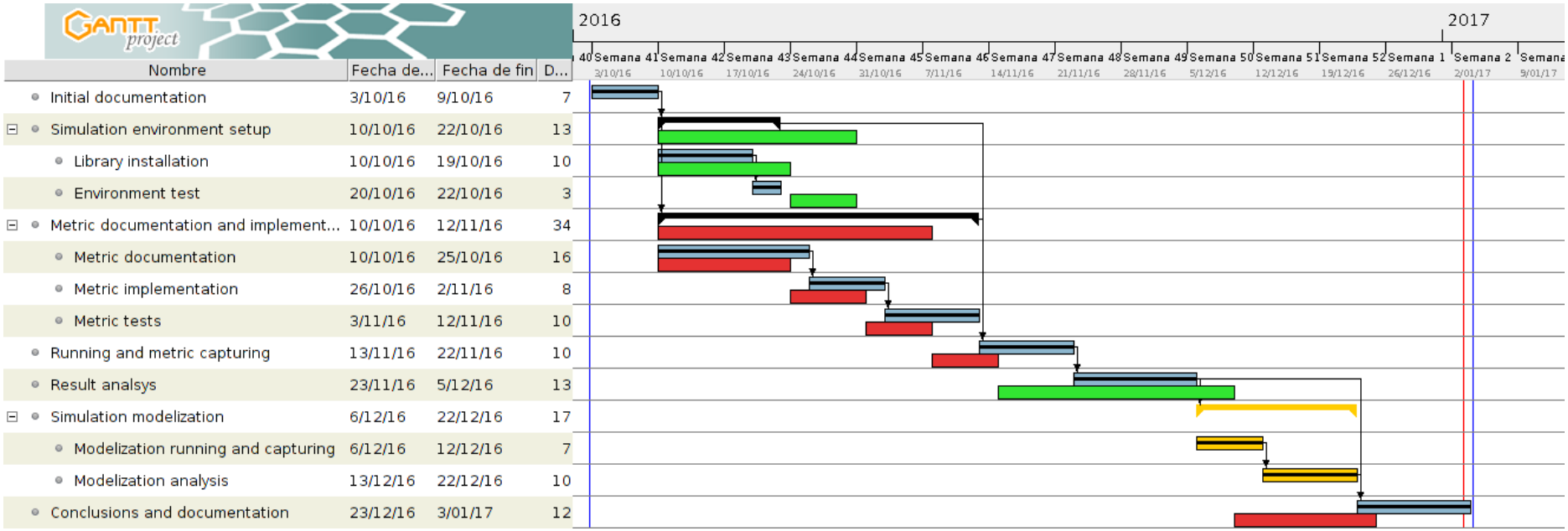
Environment variables

The .bashrc file from the user account was changed to set the PATH and LD_LIBRARY_PATH environmental variables according to the previous installations.

export PATH=$PATH:/opt/gcc/4.6.4/bin:/home/bminano/software/hdf5/bin:/home/bminano/software/silo/bin:/opt/openmpi/1.6.5/bin

export LD_LIBRARY_PATH=/opt/gcc/4.6.4/lib:/lib64:/home/bminano/software/hdf5/lib:/home/bminano/software/silo/lib:/home/bminano/software/bootstrap/lib:/opt/openmpi/1.6.5/lib:/home/bminano/software/SAMRAI/lib

Annex II – Gannt diagram, initial planification and tracking



Nombre	Fecha de...	Fecha de fin	D...
Initial documentation	3/10/16	9/10/16	7
Simulation environment setup	10/10/16	22/10/16	13
Library installation	10/10/16	19/10/16	10
Environment test	20/10/16	22/10/16	3
Metric documentation and implement...	10/10/16	12/11/16	34
Metric documentation	10/10/16	25/10/16	16
Metric implementation	26/10/16	2/11/16	8
Metric tests	3/11/16	12/11/16	10
Running and metric capturing	13/11/16	22/11/16	10
Result analsys	23/11/16	5/12/16	13
Simulation modelization	6/12/16	22/12/16	17
Modelization running and capturing	6/12/16	12/12/16	7
Modelization analysis	13/12/16	22/12/16	10
Conclusions and documentation	23/12/16	3/01/17	12

Figure 42. Gannt diagram with tracking

The Gannt diagram shows the initial tasks in blue, and two new tasks in yellow added during the development of the project. Every former task, except *Initial documentation*, has been modified a bit in its completion time. The final task *Conclusions and documentation* has been displaced due to the new tasks added, though its completion time was less than estimated.

The green tracks show the estimated time for the tasks that ended earlier. The red tasks show estimated tasks where they ended later.