CPU Model Validation for Multi-core Processor Simulation

Defense date: 30 Juny 2016

Author: Elisabet Valle Breix

Directors: Alejandro Rico, Miquel Moretó and Eduard Ayguadè
Degree in Informatics Engineer
Computer Architecture specialization

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Abstract

Designing new architectures is one of the ways to improve the performance of the computers that we used in our days. This improvements allow us to increase productivity, to simulate physics, quimics, etc. that were not be able to do it before in a realistic amount of time or even to be able to have a smartphone that could check whether tomorrow will rain or not.

But, designing new architectures implies testing them, in order to do so, one could implement the design and test it. In the case of processors, this implies a huge cost in terms of money. So, it is not feasible to build every single version of the microprocessor that it is design.

Then, what we can do? One answer is to simulate the behavior of the new architecture by using architectural simulators. The main issue with this solution is that these simulations require a lot of computational power. In order to minimize this computation, several approaches about how these simulations have to be done have been specified. Nowadays, the most used approach is execution-driven simulation due to the fact that the level of detail it allows to achieve is really high. Even though, execution-driven simulators are expensive in terms of computational power. Another kind of simulators which are less detailed are the ones included in the group of trace-driven simulators.

The TaskSim simulator uses both execution and trace-driven approaches in order to minimize the computational power required for the simulation and maximize the level of detail obtained from it. For this, it presents different models which are more or less detailed. One of these models is the memory model, which is focused on representing the best as possible the memory of the architecture simulated.

During this project, we will study how precise is this memory model of TaskSim when simulating a real architecture that is already in production. This will allow to discover the level of certainty TaskSim provides at one specific mode of simulation that has never been studied before.

Abstract Spanish

El diseño de nuevas arquitecturas es una de las formas de mejorar el rendimiento de los computadoras usadas hoy en día. Estas mejoras nos permiten mejorar nuestra productividad diaria, realizar simulaciones físicas, químicas, etc. que no hubiéramos sido capaces de ejecutar antes en un tiempo realista; e incluso ser capaces de disponer de un teléfono móvil capaz de decirte si mañana lloverá o no.

Pero, el diseño de nuevas arquitecturas implica tener que probar su validez primero, para ello, podría fabricarse la nueva arquitectura y probarla. Sin embargo, en el caso de los procesadores, el coste de fabricación es alto en términos económicos. Esto hace que no sea viable fabricar cada una de las versions de procesador que se diseñan.

¿Entonces qué podemos hacer? Una respuesta seria simular el comportamiento de la nueva arquitectura. El principal problema de esta solución es que las simulaciones son muy costosas en términos de cómputo. Para minimizar este coste computacional, diversos métodos sobre cómo y qué debe simularse se han diseñado. A día de hoy, la más utilizada es la denominada *execution-driven* ya que proporciona una fiabilidad muy alta, aún a costa de ser computacionalmente muy exigente. Otro tipo de simuladores, los cuales son menos detallados, son los denominados *trace-driven*.

El simulador TaskSim utiliza una combinación de los dos tipo de simuladores ya mencionados en orden de minimizar el coste computacional y, a la vez maximizar el nivel de detalle obtenido de la simulación. Para ello, presenta diferentes modelos los cuales difieren en la cantidad de detalle que reportan. Uno de estos modelos es el modelo de memoria, el cual se centra en simular la memoria de la arquitectura.

Durante este proyecto, estudiaremos como de preciso es este modelo de memoria cuando se simula una arquitectura real de una máquina en producción. Esto nos permitirá descubrir cuál es el nivel de fiabilidad de TaskSim en uno de sus modos de simulación que nunca se ha estudiado antes.

Abstract Catalan

El disseny de noves arquitectures és una de les maneres de millorar el rendiment de les computadores emprades avui dia. Aquestes millores ens permeteixen millorar la nostra productivitat diaria, realitzar simulacions físiques, químiques, etc. que no podíem haver realitzat abans en un temps realista; inclós el fet de ser capaços de tenir un teléfon móvil capçaç de dir si demà plourà o no.

Però, el disseny de noves arquitectures és un procés costós degut al fet de que aquestes san de testejar primer. Això es pot fer fabricant la nova arquitectura per exemple. Però, això és un procés molt costós en termes economics ja que el cost de fabricació és alt. Això provoca que no sigui viable fabricar totes i cada una de les versions de processador que s'en dissenyen.

Llavors, qué podem fer? Una resposta sería simular el comportament de la nova arquitectura. El principal problema d'aquesta solució és que les simulaciones son molt costosses en termes de còmput. Per a minimitzar aquest cost, diversos mètodes de simulació han sigut dissenyats. Avui día, el més utilitzat es el denominat *execution-driven* ja que en proporciona una fiabilitat molt alta, encara que requereix de un poder computacional molt gran. Un altre tipus de simuladores, els quals són menys detallats, són els anomenats *trace-driven*.

El simulador TaskSim utilitza una combinació de ambdós tipus dels simuladors abans mencionats per tal de minimitzar el cost computacional i, alhora, maximitzar el nivell de detall ofert per la simulació. Per això, presenta diferents models els quals es diferencien en la quantitat de detall que ofereixen. Un d'aquests models és el model de memòria, el qual és centrat al voltant de la simulació de la memòria de l'arquitectura estudiada.

En el trancurs d'aquest projecte, estudiarem com de precís és aquest model de memoria quan es simula una arquitectura real de una màquina en producció. Això ens pemetrà descobrir quin es el nivell de fiabilitat de TaskSim en un dels seus modes de simulació que mai abans s'ha estudiat.

Table of contents

1 Introduction 7

1.1 Motivation 7

1.2 Stakeholders 7

1.3 State of the Art (Related work) 8

1.4 Objectives 10

1.5 Project Scope 10

2 Planning, budget and sustainability 13

2.1 Gantt chart 13

2.2 Tasks 17

2.3 Resources 21

2.4 Budget 22

2.5 Sustainability 24

3 Benchmarks 26

3.1 Mont-Blanc benchmarks 26

3.2 PARSEC benchmark suite ###

4 Working environment set-up 28

4.1 OmpSs programming model 28

4.1.1 Mercurium 29

4.1.2 Nanos++ runtime 29

4.2 Paraver 30

4.3 Extrae 30

4.4 Mare Nostrum 3 31

5 TaskSim simulator 32

5.1 TaskSim's operating modes 33

6 TaskSim Memory evaluation with Mont-Blanc benchmarks 36

6.1 Starting point 36

6.2 Default configuration 36

6.3 Performance evaluation 38

6.4 Searching for improvement: vector-operation 39

6.4.1 First proposed configuration 39

6.4.2 Second proposed configuration 41

6.4.3 Third proposed configuration 42

6.4.4 Fourth proposed configuration 44

6.4.5 Fifth proposed configuration 45

6.4.6 Sixth proposed configuration 47

6.4.7 Seventh proposed configuration 48

6.4.8 Eight proposed configuration 50

6.5 Searching for improvement: sparse-matrix vector multiplication 52

6.5.1 First proposed configuration 52

6.5.2 Second proposed configuration 54

6.5.3 Third proposed configuration 55

6.5.4 Fourth proposed configuration 56

6.5.5 Fifth proposed configuration 56

6.5.6 Sixth proposed configuration 58

6.5.7 Seventh proposed configuration 59

6.5.8 Eight proposed configuration 60

6.6 Issue detected (1) 61

6.6.1 Solution 64

6.7 Issue detected (2) 66

6.7.1 Solution 68

6.7.2 Performance evaluation 68

6.8 New Starting point 69

6.9 Searching for improvement: bandwidth study 71

6.9.1 Performance evaluation 71

6.9.2 Setting 1st improvement point 72

6.9.3 Gain analysis 74

7 Conclusions 76

Glossary 77

Bibliography 78

List of Figures

1 Introduction

1.1 Motivation

TaskSim is a leading-edge computer architecture simulator, designed for architecture exploration of future many-core processors and parallel programming models. TaskSim can simulate parallel applications with multiple levels of abstraction, modeling the processor pipeline, memory hierarchy or just synchronization of the parallel application.

A major problem with current simulators is the increasing gap in performance between simulation and real execution on a modern complex many-core architecture. When the simulator is to be used for architecture exploration, an embarrassingly parallel problem, there is little benefit from parallelizing the simulator itself.

TaskSim takes a different approach, by increasing the level of abstraction of both the application, and the architecture. This approach requires trace-driven simulation. An existing simulator, Dimemas, operates at the MPI level.

One of the TaskSim levels of abstraction models the memory accesses of the CPU using the Reorder-buffer Occupancy Analysis model. Previous works have evaluated the accuracy of this model compared to execution-driven simulators. However, there is no prior work validating this model against real processors.

The main purpose of this project is to validate the memory accesses of this model, in order to carry out this task, the methodology will consist on compare and study the simulation of different benchmarks against their native execution.

1.2 Stakeholders

During this section will expose the different parties that could be interested on this project (i.e., the stakeholders).

Even though this project's target audience is an specific one, the stakeholders could be split into several groups including the developers of the TaskSim simulator, the project's directors which are in charge of the Tasksim simulator development as part of one of the tools developed at BSC and, of course, myself as the project relaizer. Also, we cannot forget computer architecture researchers that use simulators for performing their experiments. The next list shows the stakeholders of this project.

TaskSim project

The TaskSim is still a new simulator born from a project that is still alive and in constant developing. Until today, there hasn't been any studies using benchmarks to gather data and performance statistics as support.
This project would complement the TaskSim project adding the needed data that is currently missing.

Project's directors

In this case, both directors are interested in this project because they are part of the BSC and have direct interest on the TaskSim project and, consequently, in this one as well.

Myself

For now, I will be the only realizer involved in this project, under the supervision of the project's directors. I am part of the target audience not only because I do have interest in this project as a bachelor's thesis worker, but also because personal interest about the project's subject.

Developers

This project will provide extra help to future developers that wants to use the TaskSim simulator.
As said before, this project targets potential developers that want to simulate hardware by using the TaskSim simulator as developing platform, or any other functionality for which a multi-core architecture simulator would be needed.

OmpSs developers

The benchmark suite that will be used for this project, and more specifically, the version of each of the benchmarks included on it that we will execute and simulate, use the OmpSs programming model for exploiting intra-node parallelism. By using the OmpSs version of the benchmarks, we will be able to offer new behavior data regarding the use of OmpSs in conjunction with the TaskSim simulator.

1.3 State of the Art (Related work)

Nowadays, most of the conference papers that use simulation to evaluate different architecture use an execution-driven simulators. Even more, usually most of the works utilize a custom performance model on top of another simulation infrastructure like Simics [1], SESC [2], Simplescalar [3] or M5 [4], basically, modifying or extending those models.

So, execution-driven simulators have become more popular than other models due to the inability of traces, which are used on a trace-driven simulators like TaskSim, to capture the timing-dependent execution of multithreaded applications. However, execution-driven simulation is not perfect, and even though the level of detail provided is really high, each simulation spends a huge amount of time since it requires the full execution of target applications for its proper operation. In order to minimize this, one solution is to combine sampling and checkpointing.

When using sampling, only the most representative execution intervals are simulated, thus dramatically reducing simulation time. But, since only some intervals of the whole execution are simulated, checkpointing the architectural state just before one interval starts is required. Afterwards, the simulation will start from this checkpoint to collect statistics for the interval execution.

Anyhow, checkpointing ends up being almost the same as obtaining and simulating applications traces. So, the saved parallel state of the application cannot be changed

for different target architectures or, for example, different thread managers. At the end, you lose the benefits of using and execution-driven simulator.

Another kind of simulators utilize functional emulators or dynamic binary instrumentation tools that provide information to the performance model. COTSon [5] uses an AMD functional emulator that is feeded from the performance simulation engine in order to adjust the execution throughput, achieving a reasonable combined timing. But, these kind of simulators are used to “trace” the application online and then generate events that are utilized by the models of specific parts of the target architecture (e.g., network, memory system). These approaches end up needing the same information as execution-driven simulation, having to fully execute the target’s workload. However, they tend to be fast since they only emulate or execute natively. The main issue they face is that are less detailed, so they cannot be used for wide evaluations.

Trace-driven simulators, like TaskSim, are widely used in the industry due to the fact that, sometimes, benchmarks and applications are confidential, therefore, it could be difficult modifying or knowing which are the intervals of interest to be simulated. At other hand, traces provide a stable reference for the comparison in the evolution of different processor generations. In top of this, sampling has also been widely used in trace-driven simulation, thus dramatically reducing the simulation time with little effort since application data is not needed, neither techniques like checkpointing.

However, different limitations of trace-driven tools on true multithreaded applications restrict their utilization to single-core studies or multi-cores running multithreaded workloads.

TaskSim methodology is actually between trace and execution-driven approaches. Application code independent to the dynamic multithreaded behavior is captured and simulated by employing a trace-driven simulation engine. At other hand, the parallelism management operations that depend on the architecture and the state of the machine are dynamically executed by an execution-driven component.

Multithreaded applications can now be simulated using trace-driven environment, thus getting the flexibility and the time savings of trace-driven simulation and the

ability of execution-driven approaches to reproduce dynamic multithreaded behavior for the target architecture.

1.4 Objectives

The main goal of this project is to validate the 'memory' CPU model of a multi-core architecture simulated on TaskSim against the real hardware.
In order to accomplish this, the project is built around one main objective but hopes to accomplish, or at least to contribute in, a secondary objective.

- Main objective:

- Identify error bounds (correlations) in the memory.

- Secondary objective:

- Tune the model in order to minimize these errors, to faithfully mimic the real processors.

1.5 Project Scope

In order to keep the project in track, the scope needed for this project has to be specified correctly beforehand.

The project will start with simulating different benchmarks while emulating the MareNostrum III cluster on the TaskSim simulator. In order to check the memory generated different metrics will be used.

Afterwards, the same benchmarks will be run in the actual hardware, MareNostrum III, and the same metrics applied. This way, the results will be studied in order to validate the memory simulation of the selected multi-cores architecture.

The Marenostrum III is a cluster which nodes are composed of 2x SandyBridge-EP E5-2670 (8-core)@2.6 GHz. So, the architecture used as validation target is a Intel Xeon 45-2670 core.
Since the Marenostrum III is a cluster that is an active cluster that is currently being accessed by a lot of users in different projects, we can say that the validation will be against cores that are actually being by a wide range of users.

The benchmarks used for the simulations are the OmpSs version of the Mont-Blanc benchmarks, and the PARSEC benchmarks.

In order to check the CPU model's memory, some different metrics were considered: execution time, cache misses (L1, L2, L3) and IPC.
After considering the different options, speedup is the metric selected for this project. In the case of the Mont-Blanc benchmarks, the speedup is on the parallel section of the executions and simulations; while in the case of the PARSEC benchmarks the speedup is calculated on the whole executions and simulations.

After analyzing the results, a study of the results will show where to focus in order to modify the configuration of the architecture in order to tune the simulator to emulate the native executions with a smaller error range.

The outcome of the project will be a validated CPU model for simulation of multi-cores alongside a measure of the potential performance deviations TaskSim users could expect when simulating.

Obstacles

Some obstacles could surface during the project, including time-related problems, budget-related problems or even finding the complexity of the project to escalate above the lines in which a PFC should be restricted.

During the realization of the project, some tasks could encounter data differing more than the expected from the awaited, slowing the project's pace and ending generating problems complying the deadlines. Making the project to cover less than initially expected.

Another problem could arise if some microprocessors can be obtained due to its elevated cost. In this case, then such cores would be excluded from the simulations.

Tasks

Here it can be added a small list and description of the main tasks / estimated days it will take, needed in order to complete these steps.

2 Planning, budget and sustainability

2.1 Gantt chart

In order to explain the Gantt chart properly, first some information and description about the time restrictions for the project will be needed. Followed by the list of the main tasks and their estimated duration during the first task, project management; plus resources. From there, a **critical path will emerge**. With all this in mind, the Gantt chart will be shown and finally the detailed explanation of all the tasks.

Chronology

The project started 1st March 2015 and originally planned to end 30th June 2015. Due to unforeseen complications and underestimate the complication of some tasks, the project ended 20th June 2016. This will be covered in section 2.2.

Time distribution

The tasks are distributed by weeks, with the maximum concurrency factor of two different tasks for week. The working days are Monday, Tuesday, Wednesday, Thursday and Friday; the resting days are Saturday and Sunday. In addition, one special resting day will be placed after the completing of every task, in order to gain some freshness before starting the next task.

Some resting days may be used as work days in case of any tasks encounters unforeseen problems that requires more time to complete the task.

T1 - Project management

This task consists where the design of the project is done, meaning it will be the first task.

T2 - Research

This task is mainly focused on researching the information needed to complete the project.

T3 - Get familiar with working platform

The purpose of this task is to get used at how the working platform works, in order to compensate the start inexperience on the tools used for this project.

T4 - Run the benchmarks on the native platform

This task is one of the main points of the project; the learned architecture will be used for obtaining the native execution of the benchmarks.

T5 - Simulate the benchmarks on the simulated architecture

In this task, the learned architecture will be emulated on the TaskSim while simulating the benchmark executions. Which shares a similarity with the previous task, but in contrast will show a big difference in workload due to the time increase of the benchmark's execution over their native execution.

T6 - Analysis and comparison

The results obtained in all the executions will be studied and analyzed in this task. In order to appreciate the difference between the simulations and the native executions.

T7 - Modification Proposal and implementation

After the previous task is complete, the differences will be reasoned and a modification will be proposed for the architecture configuration, in order to minimize the difference in the executions between the simulations and the native executions.

This task will be repeated after every iteration of simulation and analysis and comparison.

T8 - Evaluation

The objective of this task is mainly to reach a conclusion after studying all the obtained data, which will be reflected in the final results of the project.

T9- Control meetings

Every week there will be an hour meeting with the project director in order to monitor the state of the project.

T10 - Documentation

This task will cover all the written documentation done for the project, also the defense and presentation of the project.

Task name	Task code	Estimated time	Dependences	Resources assigned
Project management	T1	2 days		- Research assistant - Laptop
Research	T2	5 days	T1	- Research assistant - Laptop
Get familiar with working platform	T3	2 days	T1 T2	- Research assistant - Laptop - MareNostrum III - TaskSim
Run the benchmarks on the native platform	T4	15 days	T2	- Research assistant - Laptop - MareNostrum III - TaskSim
Simulate the benchmarks on the simulated architecture	T5	20 days	T2	- Research assistant - Laptop - MareNostrum III - TaskSim
Analysis and comparison	T6	3 days	T3 T4	- Research assistant - Laptop - Paraver - Python graphic library
Modification Proposal and implementation	T7	2 days	T5	- Research assistant - Project's director - Project's co-director (when needed) - Laptop
Evaluation	T8	3 days	T7	- Research assistant - Laptop
Control meetings	T9			- Research assistant

				- Project's director - Project's co-director (when needed) - Laptop
Documentation	T10	35 days	T2	- Research assistant - Laptop - Word processor

Table: Main tasks codification

Task	Subtask	Real Duration
Project management		
	Project's size	- 1st March 2015 - 15th March 2015
	Task designing	- 5th March 2015 - 15th March 2015
Research		
	Study of pipeline's behaviour in superscalar processors	- 1st March 2015 - 26th March 2015 - 6th June 2016 - 22th June 2016
	Study of TaskSim	- 3rd March 2015 - 3th Sept. 2015
	Study of MN3' architecture (Intel SandyBridge)	- 27th March 2015 - 30th April 2015 - 1st Sept. 2016 - 14th Sept. 2016 - 9th April 2016
Get familiar with working platform		
	Getting TaskSim working	- 27th March 2015 - 19th Sept. 2016
	Getting Mont-Blanc benchmarks working	- 19th May 2015 - 24th Feb. 2016
	Getting PARSEC benchmarks working	- 25th April 2016 - 30th April 2016
	Learning Paraver's usage	- 6th April 2016 - 2nd June 2016
Run the benchmarks on the native platform		
	Execute the Mont-Blanc benchmarks on MN3	- 2nd June 2015 - 12th July 2015

		- 14th March 2016 - 20th March 2015
	Execute the PARSEC benchmarks on MN3	- 6th June 2016 - 16th June 2016
Simulate the benchmarks on the simulated architecture		
	Generate the Mont-Blanc benchmarks' traces with TaskSim	- 3rd Sept. 2015 - 24th Feb. 2016
	Simulate the Mont-Blanc benchmarks on emulated MN3 Obtain number of instructions of the parallel section	- 24th Feb. 2016 - 14th March 2016 - 14th March 2016 - 6th April 2016 - 6th April 2016 - 9th April 2016 - 9th April 2016 - 7th June 2016
	Generate the PARSEC benchmarks' traces with TaskSim	- 30th April 2016 - 6th June 2016
	Simulate the PARSEC benchmarks on emulated MN3 Obtain number of instructions of the parallel section	- 6th June 2016 - 20th June 2016
Analysis and comparison		
	Generate the speedup of the native execution	- 14th March 2016 - 20th March 2016
	Generate the speedup of the simulations	- 24th Feb. 2016 - 14th March 2016 - 14th March 2016 - 6th April 2016 - 6th April 2016 - 9th April 2016 - 9th April 2016 - 7th June 2016
	Generate a graphic comparison against older configuration	- 24th Feb. 2016 - 14th March 2016 - 14th March 2016 - 6th April 2016 - 6th April 2016 - 9th April 2016 - 9th April 2016 - 7th June 2016
Modification Proposal and implementation		
	Observe and understand the data obtained	- 24th Feb. 2016 - 14th March 2016

		- 14th March 2016 - 6th April 2016 - 6th April 2016 - 9th April 2016 - 9th April 2016 - 7th June 2016
	Propose a configuration with potential improvement	- 24th Feb. 2016 - 14th March 2016 - 14th March 2016 - 6th April 2016 - 6th April 2016 - 9th April 2016 - 9th April 2016 - 7th June 2016
	Implement the configuration proposed	- 24th Feb. 2016 - 14th March 2016 - 14th March 2016 - 6th April 2016 - 6th April 2016 - 9th April 2016 - 9th April 2016 - 7th June 2016
Evaluation		- 19th
Control meetings		- 1st March 2015 - 23 June 2015 - 24th Feb. 2016 - 17th June 2016
Documentation		
	GEP	- 1st March 2015 - 26th March 2015 (deliverables) - 27th May 2016 (milestone)
	Written memory	- 24th May 2016 - 27th May 2016 - 6th June 2016 - 22th June 2016
	Prepare slides	-23rd June 2016 - 29th June 2016

2.2 Tasks

- Project management
    - Project's size

First, is important to delimitate the scope of the project. Define the main, and secondary, objectives that want to be accomplished during the realization of this project. In order to define them, a meeting involving the project's director and co-director is needed. It is really important to establish from the beginning the importance of every task, its content and the resources needed to carry out the task.

- Task designing

In order to achieve a task in a good peace, a good specification of the projects' tasks is needed.
In this task, all the tasks concerning the realization of the project and the memory, will be carefully established. Once the workload is known, the duration is established so that the deadlines can be approximated.

- Research
  - Study of pipeline's behaviour in superscalar processors

The main activity of this task will be reading papers and documentation in order to learn how the pipeline is structured and used in superscalar processors. The idea is to gain a general knowledge of the process that will be performed during the execution of the tests.

- Study of TaskSim

The purpose of this task is to learn not only how to utilize the TaskSim simulator works and how to install it, but also to learn about the TaskSim project itself, and some of the projects currently working also with the simulator.
A good methodology to achieve this objective will be to assist the monthly TaskSim meetings, where all the projects involved with the simulator expose their progress.

- Study of MN3' architecture (Intel SandyBridge)

The principal objective of this task is to gain enough knowledge of the architecture used in the nodes that integrates the MareNostrum III. Since these nodes use an Intel SandyBridge architecture with Intel Xeon 45-2670 cores.
This knowledge will be needed for the TaskSim in order to configure the simulated architecture correctly.

- Get familiar with working platform
  - Getting TaskSim working

This task covers the whole installation of the TaskSim simulator on the MareNostrum III system. It's important to take into account that TaskSim has Nanox (Nanos++) and Mercurium as dependences.

- Getting Mont-Blanc benchmarks working

This task covers the installation and setup needed to start using the Mont-Blanc benchmarks on the MareNostrum III.

- Getting PARSEC benchmarks working

This task covers the installation[a] and setup needed to start using the PARSEC benchmarks on the MareNostrum III.

- Learning Paraver's usage

This task wasn't contemplated during the first phase of the project, but needed to be added as the work progressed.
The task covers the learning of the Paraver needed for working with the trace results obtained from the simulations.

- Run the benchmarks on the native platform
  - Execute the Mont-Blanc benchmarks on MN3

The Mont-Blanc suite will be executed on the MN3 and be referred as native execution.

- Execute the PARSEC benchmarks on MN3

The PARSEC suite will be executed on the MN3 and be referred as native execution.

- Simulate the benchmarks on the simulated architecture
  - Generate the Mont-Blanc benchmarks' traces with TaskSim

The reason of this task is to generate a valid TaskSim trace of the Mont-Blanc suite, in order to be emulated by the TaskSim.

- Simulate the Mont-Blanc benchmarks on emulated MN3

The TaskSim trace of the Mont-Blanc suite will be simulated on the emulated MN3 using TaskSim. The objective of this task was originally to obtain the execution time, but later on, the objective evolved to obtain a Paraver trace.

- Obtain number of instructions of the parallel section

In this task, the parallel section’s number of cycles will be obtained from the generated Paraver trace using the Paraver tool.

- Generate the PARSEC benchmarks’ traces with TaskSim

The reason of this task is to generate a valid TaskSim trace of the PARASEC suite, in order to be emulated by the TaskSim.

- Simulate the PARSEC benchmarks on emulated MN3

The TaskSim trace of the PARSEC suite will be simulated on the emulated MN3 using TaskSim. The objective of this task was originally to obtain the execution time, but later on, the objective evolved to obtain a Paraver trace.

- Obtain number of instructions of the parallel section

In this task, the parallel section’s number of cycles will be obtained from the generated Paraver trace using the Paraver tool.

- Analysis and comparison
  - Generate the speedup of the native execution

Once the parallel section’s execution time is obtained from the native execution, for every n-core configuration, the speedup will be calculated.

- Generate the speedup of the simulations

Once the parallel section’s cycle count is obtained from the simulations Paraver trace, for every n-core configuration, the speedup will be calculated.

- Generate a graphic comparison against older configuration

After obtaining the speedup comparison from the native executions and the simulations, a graphic comparison with previous configurations can be done in order to comprehend the differences obtained between the different configurations.

- Modification Proposal and implementation
    - Observe and understand the data obtained

  The graphic comparisons obtained from the task just described, the configuration modification at study will show clearly if the simulations have become closer to the natives executions or not.
  In some cases, differents configurations won't show significant differences, in those cases, coming with a feasible reasoning will be way to proceed.

    - Propose a configuration with potential improvement

  This task will focus on proposing a different configuration that could reduce the differences between the speedups from the simulations and the native executions.

    - Implement the configuration proposed

  Once a new proposal has been made and the configuration changed accordingly, the following tasks will be repeated in a loop:

        - Simulate the Mont-Blanc/PARSEC benchmarks on emulated MN3
        - Generate the speedup of the simulations
        - Generate a graphic comparison against older configuration
        - Observe and understand the data obtained
        - Propose a configuration with potential improvement
        - Implement the configuration proposed

  This will be kept until a satisfactory comparison emerges between the native executions and the simulations or a feasible reason is given for which the simulations' speedup won't become more similar to the native executions'.

- Evaluation

  After reaching a satisfactory reading from the different test, conclusions will be reached. Furthermore, an evaluation of the data given.

- Control meetings
  A good methodology can be obtained by meeting weekly and commenting the different results obtained.

- Documentation
  - GEP

  During the realization of this task, 7 deliverables will be done about the management of this project, plus a 5 minute video and a presentation.

  - Written memory

  This task is actually split in concurrency with long part of the project, the reason for that is to start writing the memory as soon as possible, in order to avoid encountering an overload of workload at the end of the project. The importance of the memory is to provide a good support for future usage, if needed, in case this project has to be reuse or expanded.
  The written document will include the background, motivation, state of art, evaluation and results of the project.

  Part of this task will be realized during the previous explained GEP task, where part of the deliverables will be adapted to fit this project.

  - Prepare slides

  During this task, some slides will be designed in order to help as visual material during the defense and presentation of the project.

  - Prepare presentation

  In order to prepare a better defense, during this task the slides will be checked and the defense of the project will be prepared and rehearsed.

  - Defense

  This task covers the defense of the project in front of a board, which will last 30 minutes plus another 30 minutes reserved for the purpose of preparing and setting up the needed material needed for the defense.

2.3 Resources

In order to develop this project, the following resources will be needed. The resources have been separated in hardware and software requirements.

Hardware

- Nowadays normal performance laptop.
- Physical and quiet space in order to work on the project.
- Mare Nostrum III.

Software

- Text processor.
- Matplotlib, a Python library used for making graphs.
- Python based terminal.
- TaskSim.
- Paraver tool.

2.4 Budget

This section exposes an estimation of the project's cost as well as its sustainability. The budget is split in different categories; human resources budget, hardware budget, software budget and indirect costs.

Concept	**Estimated cost**
Human resources	12.200,00 €
Hardware	777,60 €
Software	0,00 €
General expenses	76.868,05 €
Total	**89.891,65 €**

Table: Total estimated budget for the project

Human resources

Despite the fact that this will be a one-man project, woman in this case, if we want to be exactly accurate, there are several roles that would be taken during the realization of the project. Some other roles needed, like maintenance of the systems, will be provided by the Barcelona Supercomputing Center (BSC), likewise, this service will be included in the budget, reflecting the actual costs of renting the working platform.

The estimation costs for the project's roles are shown in the following table:

Role	Estimated hours	Est. price/hour	Total est. cost
Project Manager	80 h	40,00 €	3.200,00 €
Computer Architect Specialist	90 h	25,00 €	2.250,00 €
Test Runner	160 h	25,00 €	4.000,00 €
Analyst	50 h	30,00 €	1.500,00 €
Technical Support	50 h	25,00 €	1.250,00 €
TOTAL			**12.200,00 €**

Table: Estimated costs

Hardware

In order to realize all the tasks of the project, a set of hardware will be needed. The most important tools will be a laptop and the nodes rented from the BSC, the Mare Nostrum nodes, the real hardware on which we will be running the tests, to compare with the simulated hardware tests.

This two Mare Nostrum nodes will be required in order to compile and execute and run the tests in a semi concurrent way. One node will be used to compile the applications while the other one will be used to execute the application, using the job scheduler provided already by the supercomputer.

The following table shows all the hardware needed during the project:

				Estimated	**Total estimated**

Product	Price	Units	Service life	residual value	amortization[1]
Macbook Pro 13''	1.329,00 €	1	5 years	500,00 €	165,80 €
Mare Nostrum node	3.130,98 €	2	5 years	1.500,00 €	326,00 €
Samsung LS22D300HY LED Monitor 22''	96,01 €	1	10 years	50 €	4,60 €
HDMI Adaptor	6,98 €	1	5 years	1,00 €	1,20 €
TOTAL	**7.393,95 €**				**777,60 €**

Table: Hardware budget

Software

The software that will be used is during this project has no extra cost, at the time this document is written. The software included is Xcode, Nanos++, Latex, Linux distributions, OSX (there is not any extra fee since the hardware machine Macbook Pro is included in the budget), Extrae, Paraver and GNU gprof.

If the project requires more software, this budget will be reviewed during the realization of the project's tasks.

General expenses

This section includes the estimated indirect cost produced not considered in the previous sections, like electricity and unforeseen expenses.

We can't power only the nodes needed for our project, since the supercomputer must be powered completely before using it; the cost of running the whole cluster will be applied.

Product	Price	Units	Total estimated cost
Mare Nostrum power consumption	0,148832 €/kWh[17]	772.200 kWh	76.537,99 €
Working Office electricity	0,148832 €/kWh[17]	202.00 kWh (4.800 hours)	30,06 €

[1] The amortization may be obtained using the formula:
Amortization = (Value - Residual value) / Service Life

Unforeseen expenses	300 €	1	300 €
TOTAL			**76.868,05 €**

Table: MareNostrum node's electrical fee cost

Budget monitoring

It will be mandatory to possess a mechanism to constantly surveil the project's budget in order to avoid skyrocketing the pressupost.

Our technique would consist on checking and updating the budget after the completion of each main task.

This monitoring will allow us to maintain a more real budget as the actual time spent in each task and its indirect costs, like for example, electricity costs.

2.5 Sustainability

In this section, we will discuss about the sustainability of this project. The sustainability of a project can be defined as the ability of a project to maintain its finality and benefits during its project life time. In this line of thought, the market needs, the quality (of the project) and the investment are the angular points to analyze the project's sustainability.

Marked needs

TaskSim is a leading-edge architecture simulator developed by the BSC in order to satisfy and actual need in the market.

This project has as goal, to provide performance data for the TaskSim and analyze it. The simulator has this need; so the project is actually fulfilling a market need.

Quality

This criteria is always important, the quality of the data collected will have an important influence over the sustainability of the project.

The more accurate is the data collected, the higher the sustainability will be.

Investment

With more investment, more tools can be used to provide more accurate analysis of the data obtained from the simulations; so more meaning would be reached. Raising the project’s sustainability.

3 Benchmarks

3.1 Mont-Blanc benchmarks

For this project, we used the Mont-Blanc benchmark suite in order to validate the memory model of the TaskSim simulator. This benchmark suite is composed by nine HPC kernels and was developed within the Mont-Blanc european project.

The Mont-Blanc benchmarks cover a wide range of algorithms usually employed at HPC applications and stress various architectural features. Also, they have been already used in previous studies [6], [7] to evaluate the suitability of embedded platforms. The following list describes each of the benchmarks:

- **Sparse Vector-Matrix Multiplication (spvm)** benchmark multiplies a vector and a sparse matrix and stores the result into a new vector. It is useful as metric to measure performance in cases of load imbalance.
- **Vector Operation (vecop)** benchmark perform an addition of two one dimensional arrays in an element-by-element basis. This benchmark is basically memory-bounded since it does stress the memory bandwidth.
- **Histogram (hist)** benchmark performs the computation of the histogram from the values present in an array. For this task, it uses a configurable size as well as local privatization that requires a reduction stage which can become a bottleneck on highly parallel architectures.
- **3D Stencil (3dstc)** benchmark produces a new 3D volume from a in input 3D volume. Each point of the output is a linear combination of the point with the same coordinates in the input and neighboring points on each dimension. This benchmark is useful to evaluate the performance in presence of memory accesses with regular strides.
- **Reduction (red)** applies the addition operator to produce a single scalar output value from an input array. Reduction is a common operation in many computational kernels and allows to measure the capability of the compute accelerator to adapt from massively parallel computation stages to almost sequential execution.
- **Atomic Monte-Carlo Dynamics (amcd)** benchmark performs a number of independent simulations using the Markov Chain Monte-Carlo method. Initial atom

coordinates are provided and a number of randomly chosen displacements are applied to randomly selected atoms which are accepted or rejected using the Metropolis method.

- **N-Body (nbody)** takes as input a list of bodies described with a set of parameters (position, mass initial velocity) and updates their information after a given simulated time period based on gravitational interference between each body.
- **2D Convolution (2dcon)** produces a new matrix from an input matrix of the same size. Each point of the output is a linear combination of the point with same coordinates in the input and the neighboring points. Even though the description is quite similar to 3D Stencil computation, in this case neighboring points can include points with the same coordinates as the input point plus a positive or negative offset in one or two dimensions. This benchmark is useful to evaluate the performance in presence of spatial locality and strided memory accesses.
- **Dense matrix-matrix Multiplication (dgemm)** performs the multiplication of two dense input matrices. Matrix is a really common computation in many numerical simulations and measures the ability of the compute accelerator to exploit data reuse and compute performance.

Each of the benchmarks is implemented in six different versions with single and double floating point precision:

- Serial
- OpenMP
- OmpSs
- OpenCL
- OmpSs + OpenCL
- CUDA

Serial version of the benchmarks were designed to execute on a single core. The OpenMP version were designed to execute in parallel on several CPU cores. OmpSs version uses also several CPU cores but by using the OmpSs programming model, this is the version used within the project. OpenCL version accelerates the computation by using GPU as accelerator. OmpSs + OpenCL version is basically the same as the old one but exploiting both CPU and GPU at the same time. The last one, CUDA version, exploits the GPU by using Nvidia CUDA programming model. None of the six versions use vectorization since they were designed to be executed on ARM 32 bit platforms which don’t have a SIMD unit IEEE-754 compliant..

4 Working environment set-up

4.1 OmpSs programming model

OmpSs is a programming model currently developed at Barcelona Supercomputing Center which aims to extend OpenMP with new directives in order to add support for asynchronous parallelism and heterogeneity. One of OmpSs capabilities is to execute regions of code in a parallel way even if they are not explicitly declared as parallel with respect other regions. This can be done because OmpSs parallelism declarations are made by using data dependencies for each task. So, a section of code (i.e., an OmpSs task) will be queued in pool of ready tasks only when its data dependencies are satisfied. Once the tasks are queued, they will be executed once a thread will be available.

Regarding heterogeneity, OmpSs provides support for executing parts of the application at the GPU just adding an OpenMP-alike directive to the code. This way, one can see that one of the most important features of OmpSs is keeping things simple. This translates into clear-cut code which is easy to maintain afterwards. It allow the developer to forget about complex and architecture-dependent code, allowing him to just focus on the algorithm that need to be implemented.

From the beginning, OmpSs development team has been very close to OpenMP development, trying to push OmpSs' new features into OpenMP programming model. Some of them has been actually included into OpenMP as, for example, tasks with data dependencies, which were included on OpenMP version 4.

As already mentioned, OmpSs creates a pool of tasks that will be, at some point of the execution, binded to a thread. In order to know which of them is available, a pool threads is created as well by the runtime at the beginning of the application. One of those will be the master thread, responsible of executing the serial parts of the applications. The rest of the threads are called worker threads. Each of these threads could also convert themselves into master threads in case of nested parallelism.

In OmpSs programming model, each parallel region of code is defined as task. Each of them has its own dependencies, which are declared by using *in*, *out* and *inout* directives. These directives are set by the user and using them only helps the runtime to construct the dependency graph. This dependency graph will be used later during the execution to decide whether a task can be executed or not by checking if all the data dependencies of the task are fulfilled. Understanding what each of the directives indicates is pretty straightforward. *In*

directive tells the runtime the task needs to read the data indicated at the clause; *out* that needs to write on the region of memory where the data is contained; and *inout* that it will read and write at that memory region during the task's execution. Also, some other directives are available to indicate, for example, the priority of the task so the runtime will try to execute it as soon as possible.

Figure 1 shows how OmpSs programming model works. Basically, you compile the annotated source code with the Mercurium source-to-source compiler, which generates an executable. This executable will use Nanos++ libraries, i.e., the runtime, so it can be executed in a parallel way.

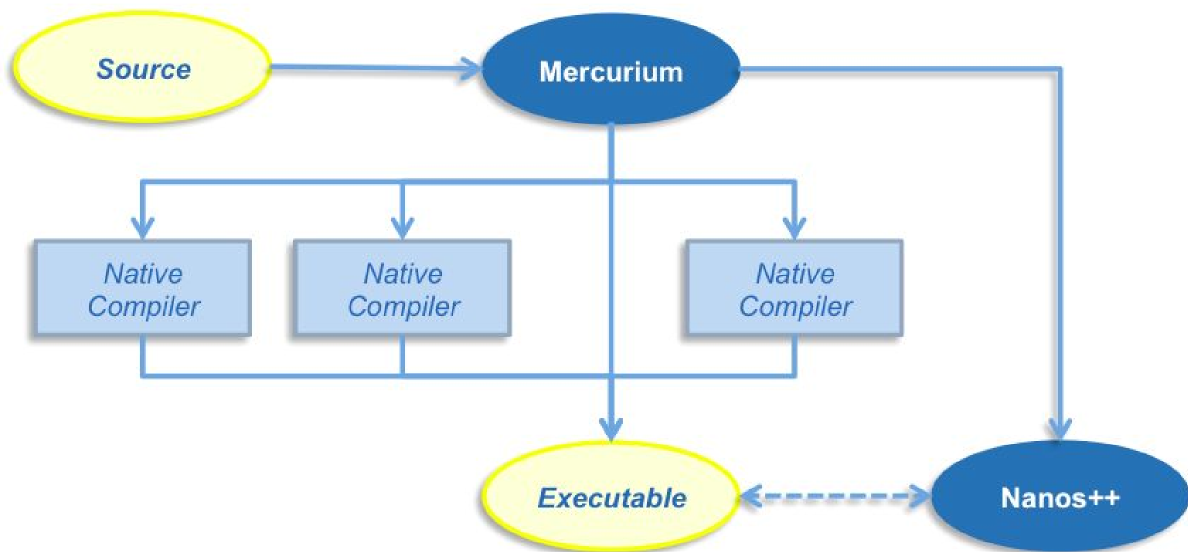


Figure 1: OmpSs schema

4.1.1 Mercurium

Mercurium is a source-to-source compiler currently developed at BSC, it supports C, C++ and Fortran programming languages. It is mainly used along with Nanos runtime to provide support for OmpSs and OpenMP programming models as well as the TaskSim simulator. All of this features are include in a form of plugins written in C++ and dynamically loaded by the compiler depending on what configuration the programmer chooses. Mercurium compiler has stable support for several programming languages:

- C/C++
- Fortran
- CUDA
    - C/C++ and Fortran
- OpenCL

- C/C++ and Fortran

4.1.2 Nanos++ runtime

Nanos++ is a runtime library (RTL) mainly developed in C++ which aims to give support for parallel environment. Its main objective is to be in charge of dependencies generated by OmpSs directives, even though is also used by the TaskSim simulator to simulate parallel architectural setups like the ones of this thesis. It also supports OpenMP and Chapel.
OmpSs tasks are implemented by Nanos++ with user-level threads when available (g.e., x86, x86_64, ia64, aarch64, etc.).
One of the most important features of Nanos++ runtime is that the programmer does not need to deal with complex usage when executing his applications.
Nanos++ structure is mainly an idle loop in which idle worker threads are waiting to be called by the master thread. Once this happens, the master thread will assign work to the worker thread. For this task, Nanos++ uses data structures where a task is described, each of the data structures is called *work descriptor* in Nanos++ runtime library.
Last but not least, Nanos++ package can be compiled in four different versions: performance, debug, instrumentation and instrumentation-debug. The one that will we use during this project is basically the instrumentation one. This version is instrumented so several information of the execution can be obtained afterwards in order to, for example, obtain traces of the application executed.

4.2 Paraver

Paraver is a development tool currently developed at the Barcelona Supercomputing Center and aims to respond to the need of having a way to visualize in a graphical view the behavior of an application in order to obtain an analysis of it. One of its features is to be able to read traces generated by Extrae.
What is showed and how of those traces is configured by the developer in order to allow him to debug or to know what is really happening at every part of the application.
Another important feature is that Paraver's trace format has no semantics. This translates into support for new programming models with no cost. Furthermore, metrics are programmed within the tool. To compute them, the tool offers a large set of time functions, filters and mechanisms to combine two time lines. This means the developer can obtain a practically unlimited number of metrics with the available data provided by the trace. Of course, once views are configured, this configuration can be saved for using it later.

Some other Paraver features are support for:

- Detailed quantitative analysis of program performance
- Concurrent comparative analysis of several traces
- Customizable semantics of the visualized information
- Cooperative work, sharing view of the trace in form of configuration files
- Building of derived metrics

4.3 Extrae

Extrae is a development tool currently developed at Barcelona Supercomputing Center which allows the developer to generate traces from the execution of its application. These traces can be analyzed later by using Paraver. Extrae tool uses different mechanisms to obtain significant information of the execution like hardware counters or function calls from both user and system space. Extrae instrumentation package supports the following programming models:

- MPI
- OpenMP
- CUDA
- OpenCL
- pthreads
- OmpSs

All parallel programming models are supported in conjunction with MPI, except MPI itself, of course.

Extrae configuration is made via an XML file, which contain which kind of counters, events or executions states will be gathered by the tool.

4.4 Mare Nostrum 3

For this project we used Mare Nostrum 3, which is hosted at the Capella building, located at the *Campus Nord* of the *Universitat Politècnica de Catalunya.* It was used for running, debugging, evaluating and simulating each of the benchmarks used in this project. This supercomputer is based on Intel Sandy Bridge processors, iDataPlex compute racks interconnected via an InfiniBand network and runs Linux Operating System.

Further information about the platform is provided at the following list:

- Peak Performance of 1.1 Pflops
- 100.8 TB of main memory

- Homogeneous nodes
    - 3056 compute nodes
    - 2x Intel Sandry Bridge-EP E5-2670/1600 20M 8-core at 2.6 GHz
    - 8x 4GB DDR3-1600 DIMMS (2 GB/core)
- Heterogeneous nodes
    - 42 compute nodes
    - 2x Intel Sandry Bridge-EP E5-2670/1600 20M 8-core 2.6 at GHz
    - 2x Xeon Phi 5110P
    - 8x 8GB DD3-1600 DIMMS (4 GB/core)
- 2 PB of disk storage
- Interconnection networks:
    - InfiniBand FDR10
    - Gigabit Ethernet
- Operating System: Linux – SuSe Distribution

5 TaskSim simulator

TaskSim is a leading-edge computer architecture simulator currently developed at the Barcelona Supercomputing Center under the RoMoL project running on the same center. This simulator design aims to two different objectives:

- Architecture exploration of complex multi-/many-core chip multiprocessors with large numbers of cores
- Research on parallel programming models

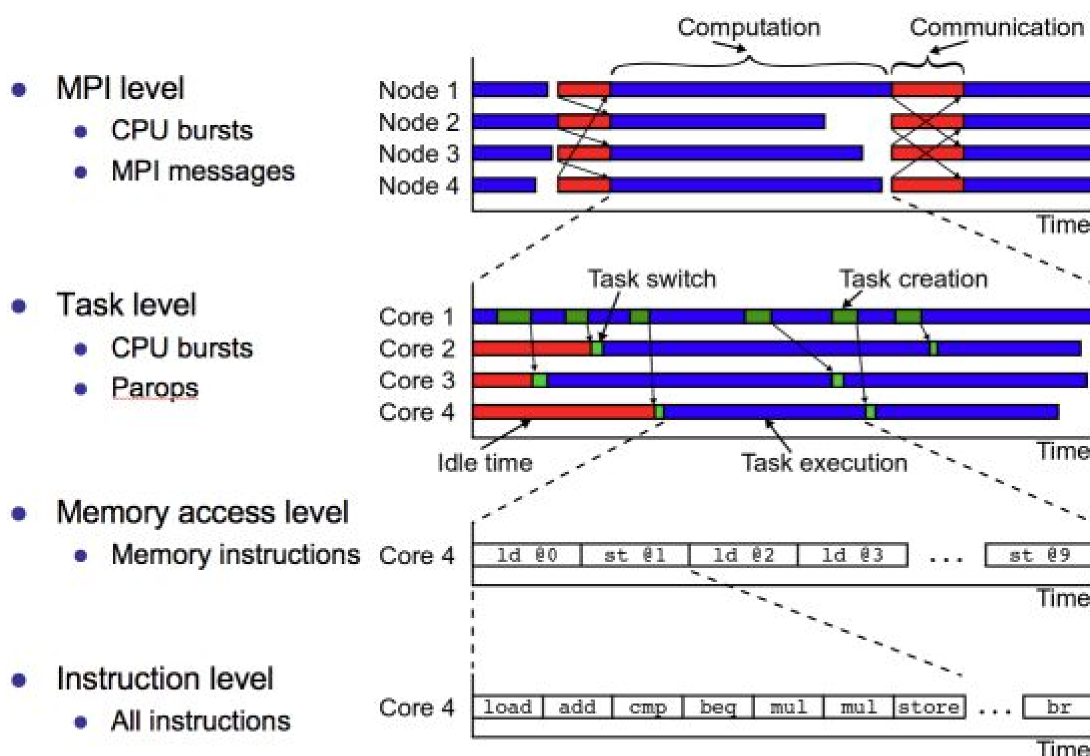


Figure 2: Different level of abstractions of TaskSim

A big issue with current simulator is the increasing gap in performance between the simulation and the real execution on a modern complex multi-core architectures. Actually, even when simulations are used for an architecture exploration, which is an embarrassingly parallel problem, the benefit you can achieve by parallelizing the simulator itself is small.

TaskSim takes a different approach, by increasing the level of abstraction of both the application and the architecture [8]. For this, trace-driven simulation is required. The highest

level of abstraction of TaskSim is task mode, at this mode, TaskSim simulates the application at the

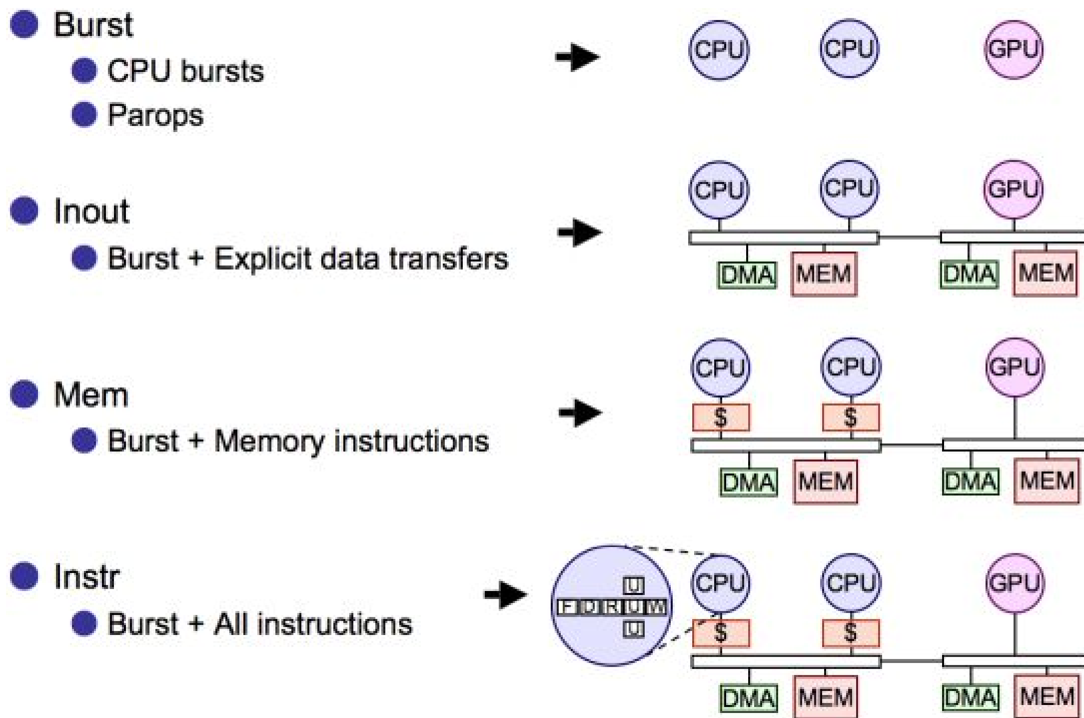


Figure 3: Different modes of TaskSim

level of OmpSs tasks and the architecture as CPU burst an parallel operations (parops). This mode is useful for scalability studies of the parallel application. The memory model, which is the one that is validate within this project, provides a greater level of simulation accuracy at a slowdown of just 25x compared with native execution. Finally, the instruction mode offers the greatest level of accuracy. All of this modes by just using the same simulation tool. Figure 2 shows in a graphic manner the different levels of abstraction TaskSim can represent. At Figure 3 you can also see the different modes the simulator operates.

5.1 TaskSim's operating modes

As mentioned before, TaskSim can operate at four different levels of abstraction.

- Burst mode
- Inout mode
- Memory mode
- Instruction mode

Each of the different levels allow different level of detail of the simulated architecture, being the most detailed the instruction mode and the least the burst mode. The following part of this document will explain in a deeper detail how each of the levels work.

Burst Mode

The burst mode implements a simple event-driven core model. Basically, one trace for the main task is provided (i.e., a trace of the master thread of the application). Also, one trace for every other task executed in the application is needed.

A task trace includes all computation burst as CPU events, and the required synchronization events for a correct parallel execution. Since, as said before, each of the tasks traces are separated, they can be individually scheduled to available cores and thus be executed at the same time in a parallel way.

Inout Mode

This mode, built on top of the burst mode, provides precise simulation of multi-core architectures with non-coherent distributed shared memory.

In this mode, the burst mode is extended with specific events that indicate when explicit memory transfers (e.g., DMA) start as well as the synchronization with their completion before bursts reading that data.

Memory Mode

This mode adds on top of the input mode the possibility to also simulate the memory accesses performed by the application. For this, the burst mode is extended to add trace memory simulation to the computation bursts in the target application. For each computation burst, a trace of all corresponding memory accesses is captured and a link to the resulting memory access trace is included in the associated CPU event. Then, at simulation time, when a CPU is event is processed, the computation burst duration in the trace is ignored, and the corresponding memory access trace is simulated instead.

In order to model the core performance, the memory access trace also includes the number of non-memory instructions between memory accesses. The records in the trace include following information related to each memory access:

- Access type: Load or Store

- Virtual memory address
- Access size in bytes
- Number of instructions since the previous memory instruction

This information is employed in the core model to simulate the performance of an out-of-order core by modeling the re-order buffer (ROB) structure.

Instruction Mode

This is the most detailed mode of the TaskSim simulator. The instruction mode extends the burst operation mode to allow the employment of instruction traces for computation bursts, similarly to the extension applied for the memory mode. The instruction flow is captured at tracing time by using PIN, a dynamic binary instrumentation tool, and an instruction trace file is generated for every computation burst. Then, as for the memory mode, the CPU event format is extended to include an extra field for storing a reference to the corresponding instruction trace.

The core model in instruction mode operates exactly as in burst mode, but the duration of CPU events is ignored, and the corresponding instructions are simulated instead. There are many ways to encode an instruction trace but, in general, they require the inclusion of the information for every instruction:

- Operation code
- Input and output register lists
- Memory address (and data size): for memory instructions
- Next instruction (for branch instructions)

This mode allows a detailed core simulation. Obviously, the more accurate the core model the longer the simulation takes to complete. In TaskSim though, different instruction-level core models can be used, and even switched from one to another at simulation time.

6 TaskSim Memory evaluation with Mont-Blanc benchmarks

During the next section we present the results we obtained that enabled us to validate the TaskSim memory model. Before going into deeper details, some definitions should be explained before in order to allow a good comprehension of the following sections

Parameters
With parameters we refer to the options we set to TaskSim (i.e. memory bandwidth, memory latency, processor frequency, etc.). These values will be used later at simulation time to emulate the different architecture's components behavior.

Native/Target architecture
When talking about native and target architectures, the first one refer to the machine that runs the simulator as well as the architecture that we want to simulate; the later one refers to the architecture that is actually simulated during the TaskSim simulation.

Speedup
Each of the speedups that will be showed is computed by dividing the parallel execution time of the specific case by the base case, which is always the same configuration but only using one core.

6.1 Default configuration

The next step is to simulate the Mont-Blanc benchmarks on the emulated MN3 with TaskSim using the starting configuration of MN3. The results obtained with this simulations will be used as "default configuration" for evaluating the gain or loss of speedup due to the proposed modifications, in the attempt to minimize the simulation errors.

The principal parameters that will concern our performance are:

- cpu_freq_mhz

- The frequency at which MN3 runs is 1000 MHz.
- Bandwidth
    - The MN3 has a memory bandwidth of 16 GB/s.
- Latency
    - The memory latency is calculated to be at 140 cycles.

In the following chart a comparison between the native execution speedup (top) and the simulation's speedup (down), is shown, using 1, 2, 4, 8 and 16 nodes:

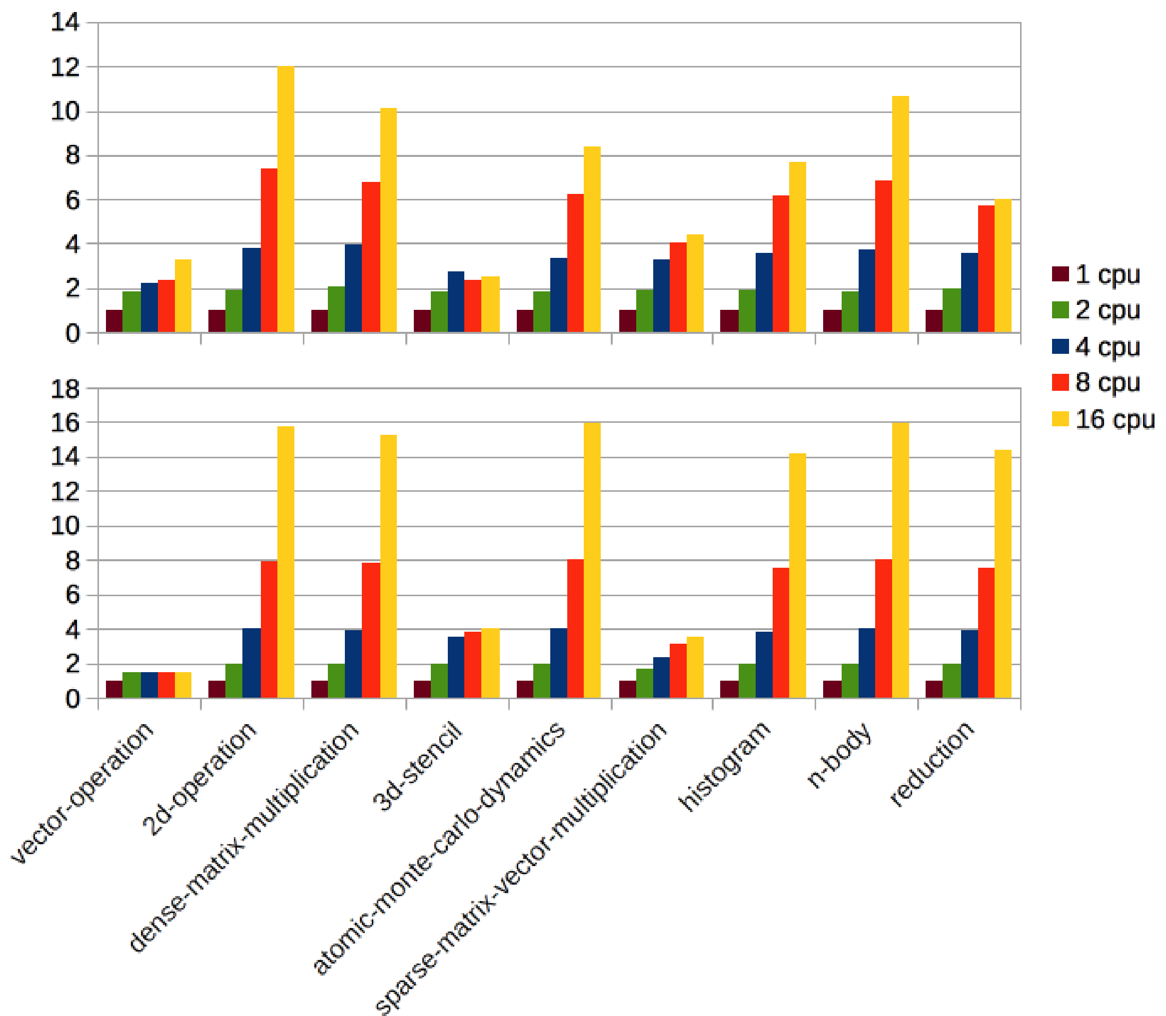


Chart xx: Speedup comparison between native execution (top) and simulated MN3 default configuration (bottom)

As a reminder, the TaskSim simulations using 16 cores is not able to fully reproduce the MN3 real architecture, and that is due to the NUMA effect. Because of that, the simulations with 16 cores are not representative.

The NUMA effect

To explain later

Non-Uniform Memory Architectures (NUMA) is an architecture configuration that consists on a single motherboard where multiple CPU are plugged. Each of these also have its own memory, which is connected only to its CPU.

The NUMA effect is an artifact that you can experience when accessing from one CPU to the memory of the other. What basically happens is that, since the CPU is not accessing directly your memory, a coherence method must be executed to make sure that cache memories from both CPU have the same data. This process can affect drastically the the memory bandwidth, thus decreasing the performance. MareNostrum 3 presents such an architecture configuration, where two Intel Xeon processors are connected to the motherboard, having each of them their own private memory.

At the end, the drop in performance creates an special case where both the memory bandwidth and latency for some memory accesses is decreased. These specific cases cannot be simulated at TaskSim due to the following reasons:

- Memory bandwidth and latency are specified as a unique value on TaskSim's configuration files
- Once TaskSim reads the configuration file, the different parameters remain unchanged across the whole simulation

At this point, it is clear that TaskSim cannot simulate different memory accesses with different memory bandwidths and latencies, therefore the cases where MareNostrum 3 uses both CPU's at the same time (i.e. 16 cores) cannot be simulated properly.

6.2 Performance evaluation

Let's analyze the results obtained. Some of the benchmarks simulated show quite similar speedups to the native's execution (e.g. 2d-operation and n-body); While other benchmarks simulated are quite distant from the expected results (e.g. vector-operation and 3d-stencil). The rest of the Mont-Blanc benchmark something in between, the results are not perfect but acceptable.

Add table showing close to native and far away to native

Usually, a simulator will overestimate the behaviour of the benchmarks simulated. This behaviour can be observed in most of the emulated benchmarks [the nice chart with all the benchmarks in bars] how the simulations show a better speedup than the native executions, except for the vector-operation and sparse-matrix-vector-multiplication benchmarks.

For now, the project will focus on proposing modifications for those two benchmarks that show more discord between native execution and simulations, hoping to faithfully mimic the native execution.

6.3 Searching for improvement: vector-operation

6.3.1 First proposed configuration

If we take a closer look to the vector-operation speedup comparison:

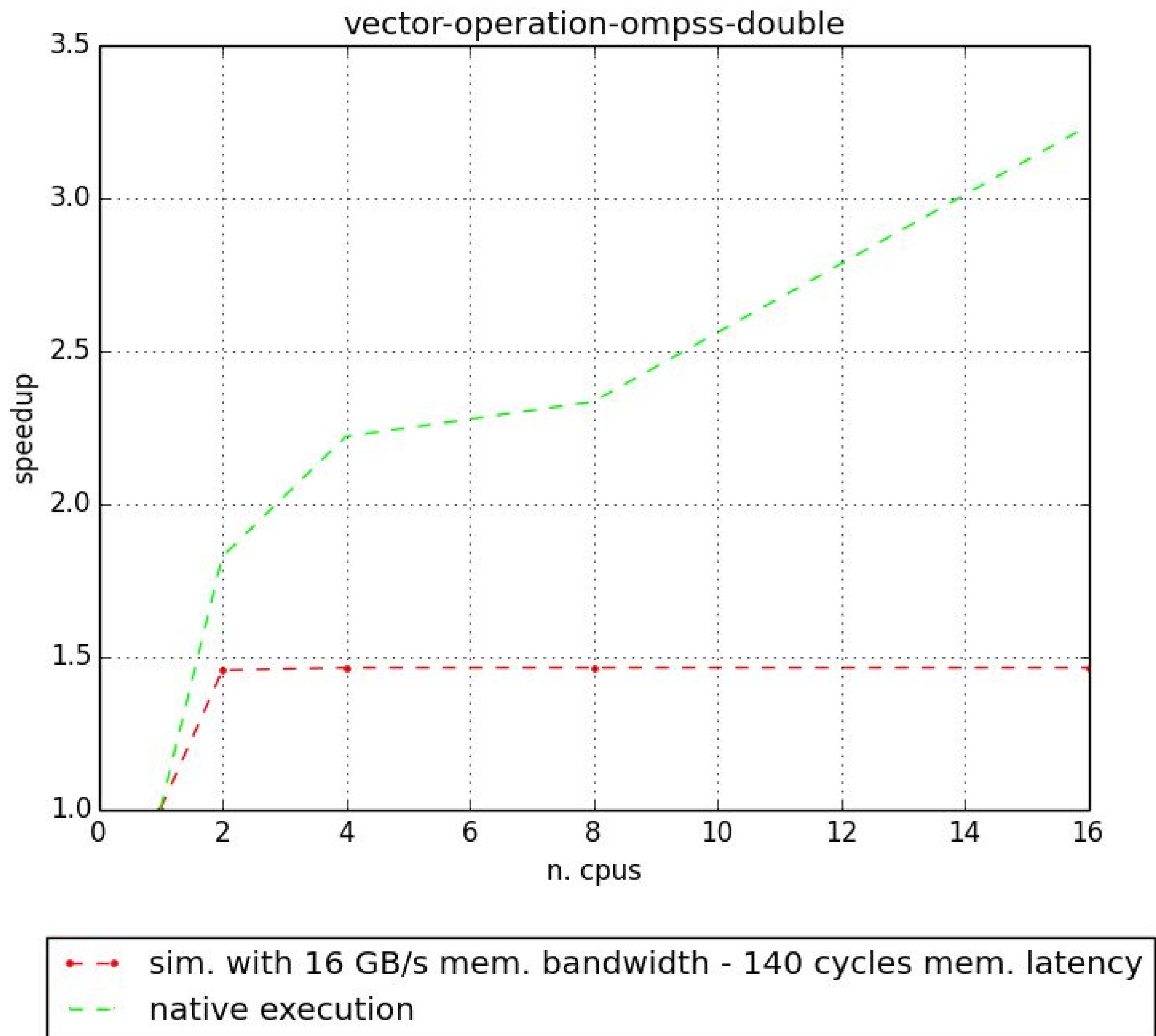


Chart xx: Speedup comparison between native execution and MN3 default configuration

The simulation shows a ceiling pattern around the speedup of 1.5, while the native execution keeps scaling even if not exactly linearly.

In order to get a more feasible simulation, the MN3's configuration will have to be modified. Even if we yet not known the most appropriate parameters for this benchmark, since we know the benchmark behaviour, we can deduce a few points that could lead us to the right direction:

- We know the vector-operation has a lot of memory access (i.e., bandwidth memory bounded).

- We can observe that the increment of cores shows the same improvement as two cores, we can deduce it does not seem to be “enough work” to do for more cores.

One big reason for this would be that the bandwidth is too low, and the cores have to wait too much time to write the data, thus decreasing the performance of the benchmark, which directly translates into the scalability observed.

The proposed configuration in order to search more improvement would be to increase memory bandwidth value, for the purpose of proving this theory we proposed the following configuration:

- cpu_freq_mhz
  - 1000 MHz
- Memory bandwidth
  - 32 GB/s
- Memory latency
  - 140 cycles

In the following chart, the speedup obtained with this configuration can be compared with the native execution:

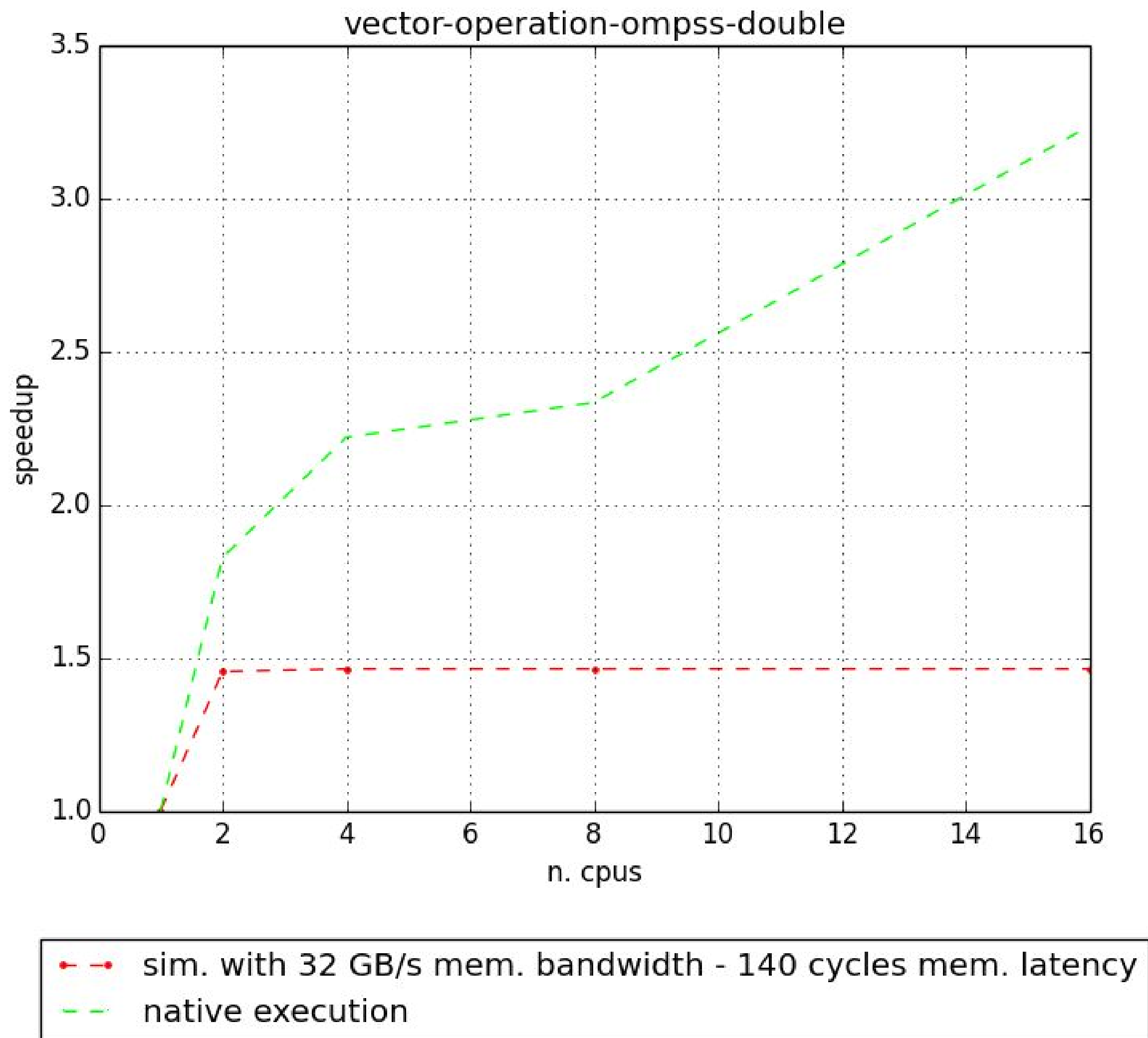


Chart xx: Speedup comparison between native execution and MN3 proposed configuration

Performance evaluation

It does seem like increasing the bandwidth while keeping the rest of the memory parameters the same does not make the simulation to be more realistic as we thought it would happen at first.

The reason for this behavior could be to explain later.

6.3.2 Second proposed configuration

Since our last configuration did not make any change on the simulator behavior, it looks like looking for other parameters to tune. In this line, we wanted to try a new configuration where we could see if the memory latency affected the performance of the application. For this, we proposed the following configuration:

- cpu_freq_mhz
    - 1000 MHz
- Memory bandwidth
    - 16 GB/s
- Memory latency
    - 280 cycles

After running again the scalability experiments we observed the behavior showed at the following chart:

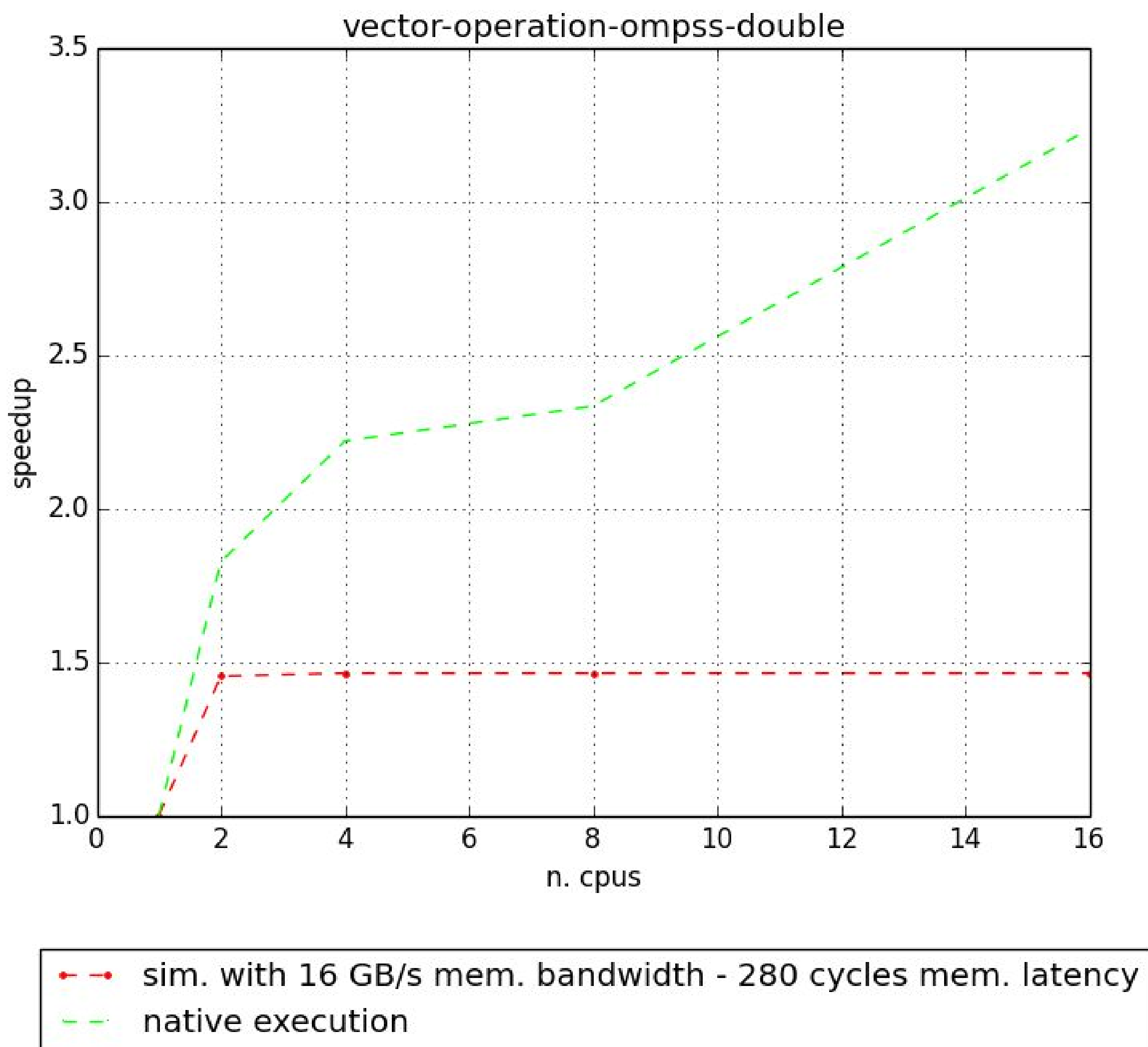


Chart xx: Speedup comparison between native execution and MN3 proposed configuration

Performance evaluation

Increasing the memory latency does not seem to give a higher speedup, neither a worse one. That is indeed the expected behaviour for a non memory latency dependant benchmark like the vector operation.

6.3.3 Third proposed configuration

So we already saw that neither increasing the memory bandwidth nor the memory latency while keeping the frequency at the same value had any impact on the simulation of the vector operation benchmark. Then, our next step is to modify the CPU frequency. In this line, the next configuration is proposed:

- cpu_freq_mhz
  - 2600 MHz
- Memory bandwidth
  - 16 GB/s
- Memory latency
  - 140 cycles

Again, the same experiments were executed. The results can be observed at the next chart:

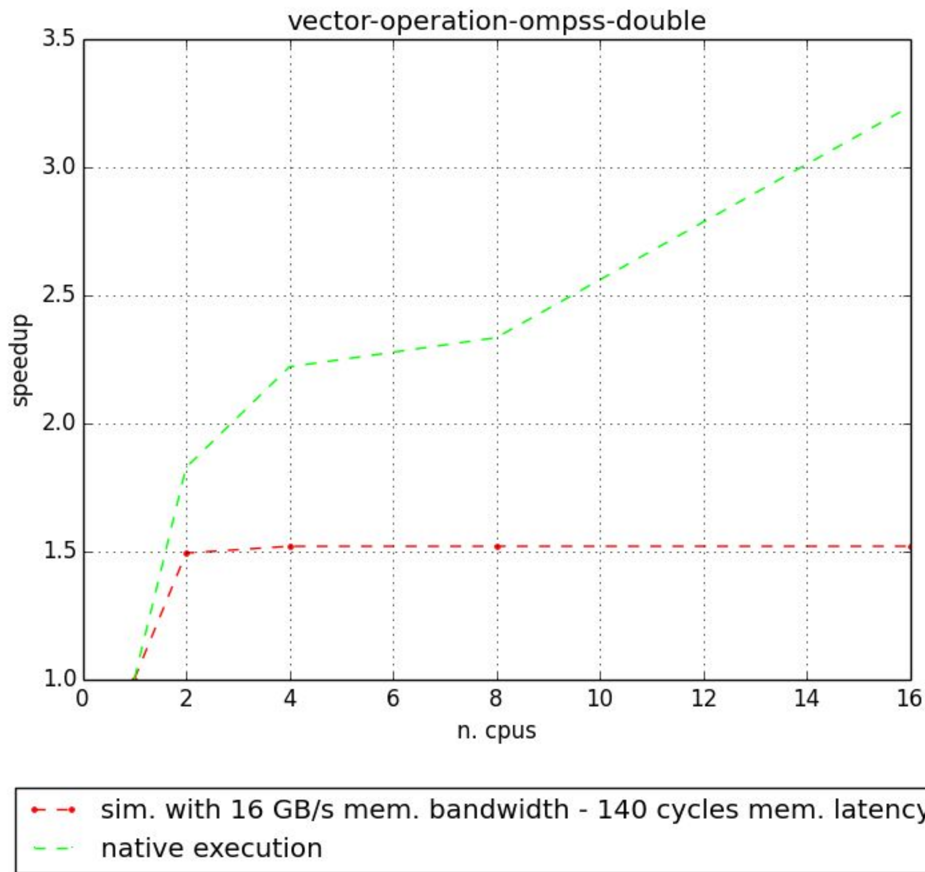


Chart xx: Speedup comparison between native execution and MN3 proposed configuration

Performance evaluation

Even though not too much, there is some difference in the speedup for 4 and 8 CPUs. So, it seems like increasing the CPU frequency actually affects somehow the performance of the application. This can be seen more clearly on the next chart where both first and third proposed configuration are compared against the native execution:

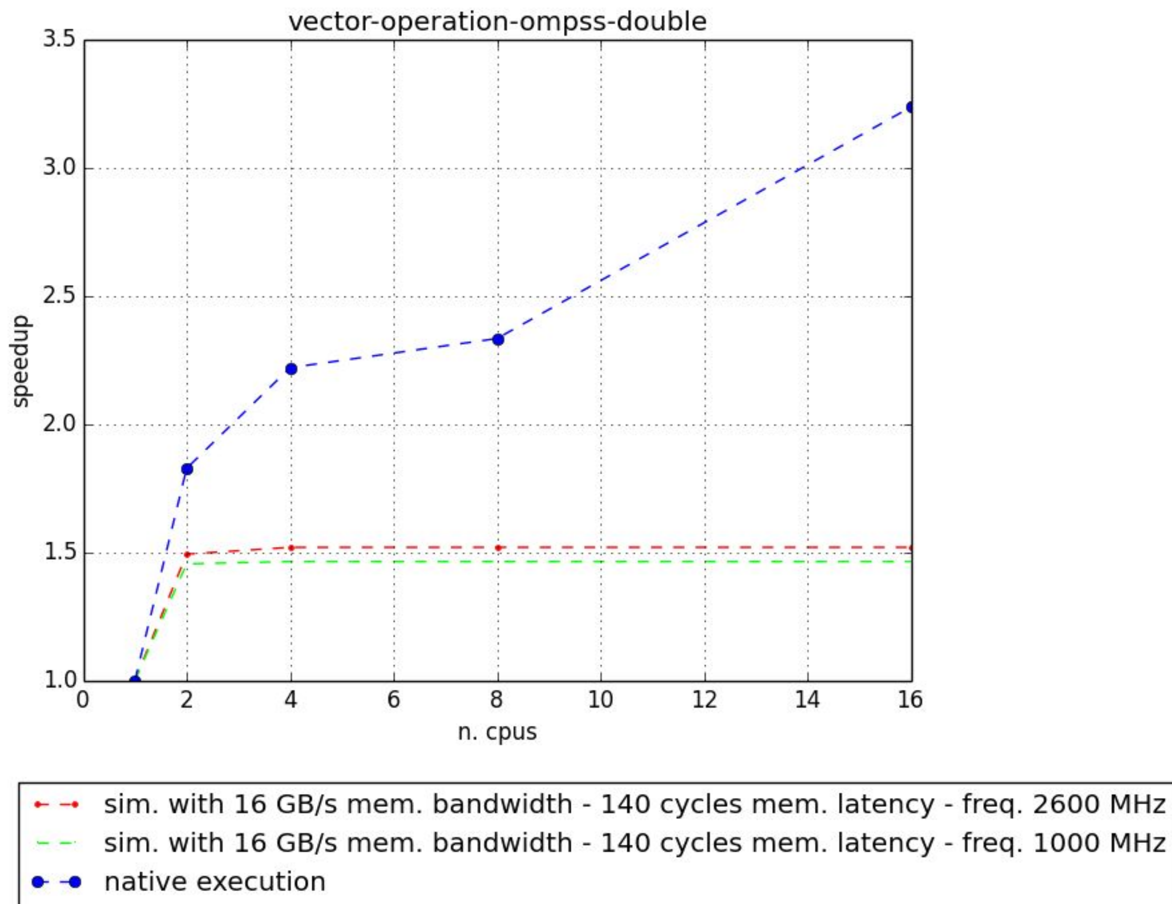


Chart xx: Speedup comparison between native execution and MN3 default and proposed configuration

What we learn of this is that, even though the default simulation configuration for MareNostrum 3 uses a CPU frequency of 1000 MHz, it does not seem to be the proper one to reflect the target architecture for memory bounded applications. Then, for now on, we will keep the CPU frequency as the real one (2600 MHz) while we will modify both memory bandwidth and latency to see if we could find a configuration that really mimic the target architecture.

6.3.4 Fourth proposed configuration

After modifying the bandwidth and the latency of the memory and then the frequency we did not obtain much difference in speedup starts to give us the impression that something is not exactly going as it should.

Since the ceiling pattern appears afters the usage of 2 cpus, there is a change that the problem that could indicate by a resource that gives the same performance for two or more cpus.
In order to check if the memory is the causant, we try to start simulating with perfect memory latency using the following configuration:

Stuff to say about, since no changes, maybe we were wrong. Something is going on. Nothing changes. Try to put latency perfect. In the tasksim, if latency = 1, then perfect. Than way, we can see if the freq is 2600 or 1000.

Proposed parameter configuration:

- cpu_freq_mhz
    - 2600 MHz
- Memory bandwidth
    - 16 GB/s
- Memory latency
    - 1 cycles

A perfect memory latency is when each memory access can be handled in one cycle.

Performance evaluation

All these TaskSim simulations usually uses around 2-6h depending of some parameters.

Lols, basicament, user designed signal 2, les simulacions triguen masa masa. Algo rara pasa i el simulador es queda penjat? Maybe latency = 1 is not perfect. We check the code and it seem it’s when latency = 0. Or not, I can’t remember it...

6.3.5 Fifth proposed configuration

After the impossibility to check the simulator execution performance while trying to simulate a memory without latency, our next step was to decrease the memory latency to a value that were low enough to show us if decreasing the memory latency

while keeping the CPU frequency at 2600 MHz and the bandwidth at the default value will make any change. Then, we propose the following configuration:

- Cpu_freq_mhz
  - 2600 MHz
- Memory bandwidth
  - 16 GB/s
- Memory latency
  - 50 cycles

After running the usual scalability performance we obtained the results that can be seen at the next chart:

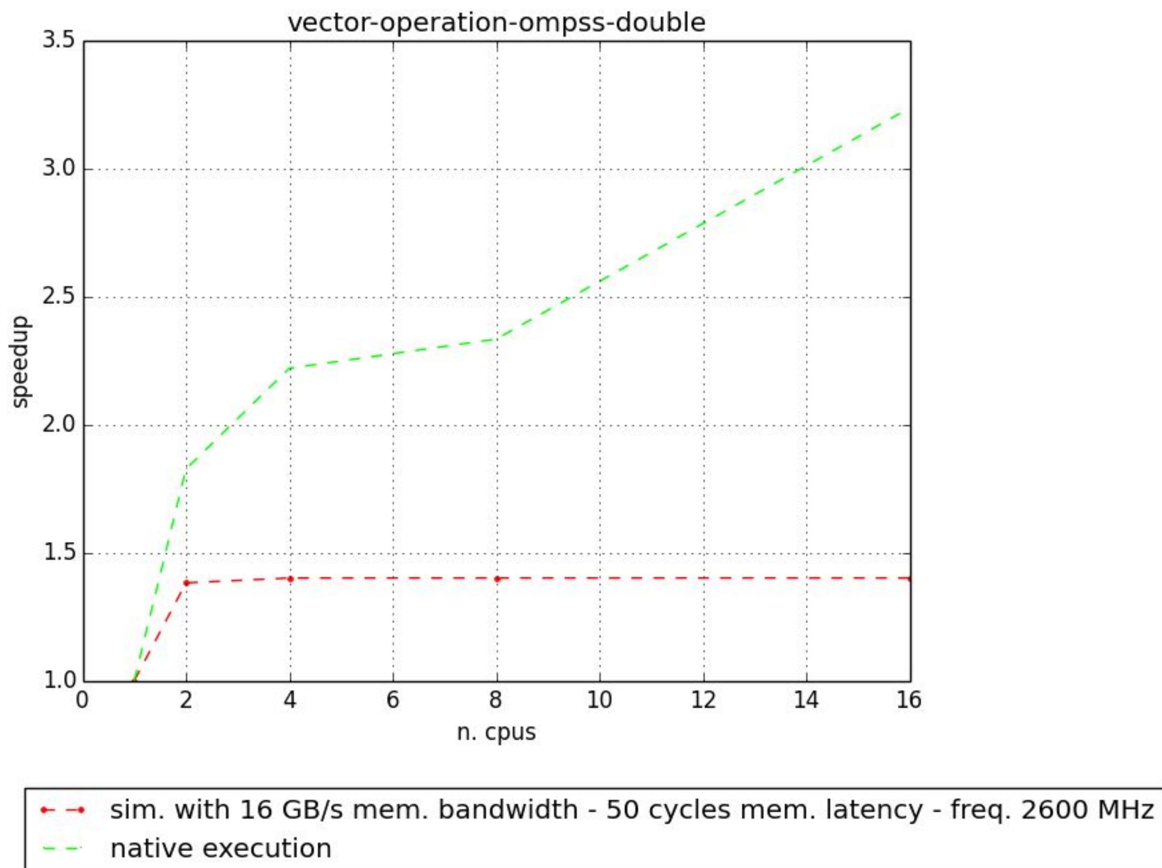


Chart xx: Speedup comparison between native execution and MN3 proposed configuration

Performance evaluation

By looking at the chart, we can see that no improvement was achieved. Actually, if we compared this configuration to the default one, we can see in a better way that

the performance of the simulation execution decreased. This is shown at the next plot:

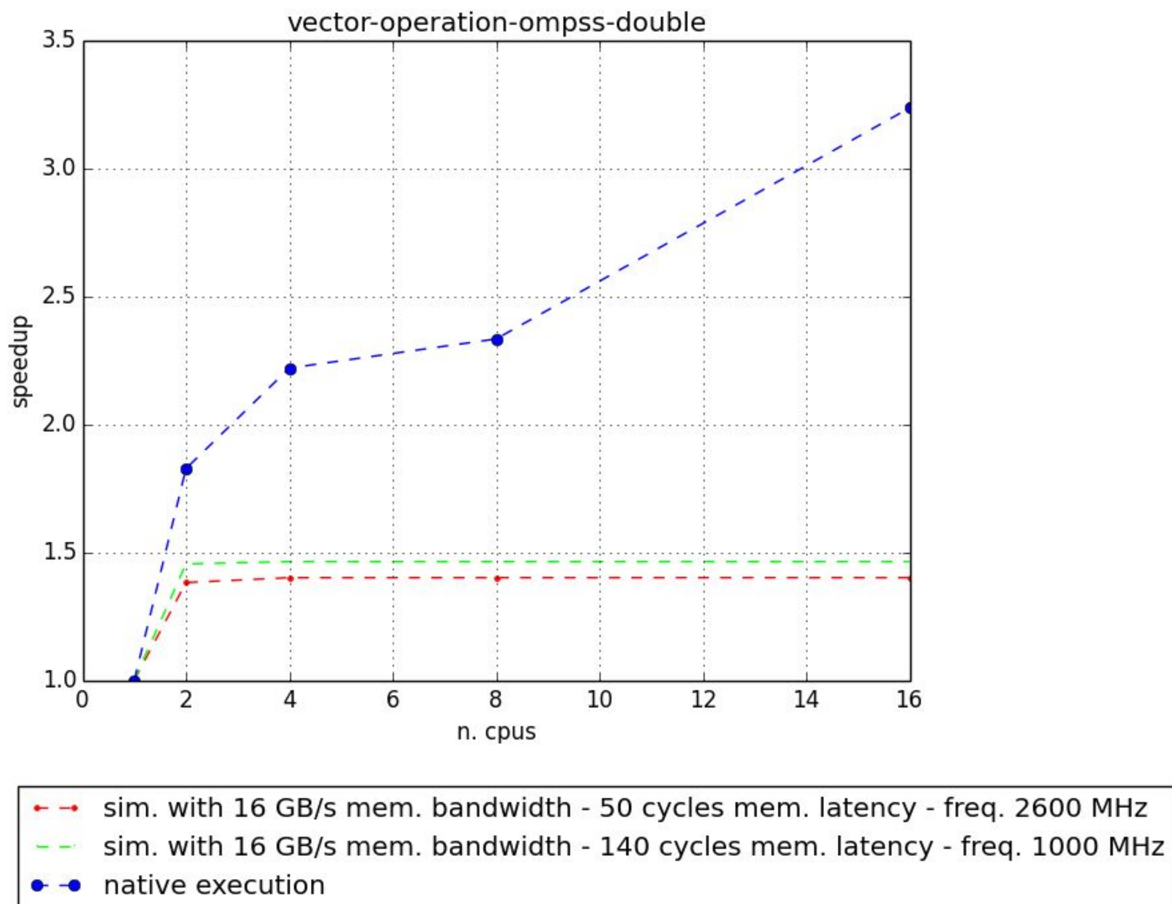


Chart xx: Speedup comparison between native execution and MN3 default and proposed configuration

The reason of this behavior is that, while the memory latency should allow us to increase the performance and the scalability a bit, it does not mean that the scalability has to improve as well. Actually, it could even hurt the scalability if the performance for one core is greater.

6.3.6 Sixth proposed configuration

As now, any of the configurations proposed is really modifying the behavior of the simulator execution in a way that closes the gap between the native and the simulator executions. Anyhow, by setting CPU frequency to 2600 MHz and memory latency to 50 cycles we do observe some minimal changes. For this reason, in this iteration we will keep these values the same and then we will increase the memory bandwidth. So, this is our sixth proposed configuration:

- cpu_freq_mhz

  - 2600 MHz
- Memory bandwidth
  - 32 GB/s
- Memory latency
  - 50 cycles

The scalability results for this configuration can be observed at the next chart:

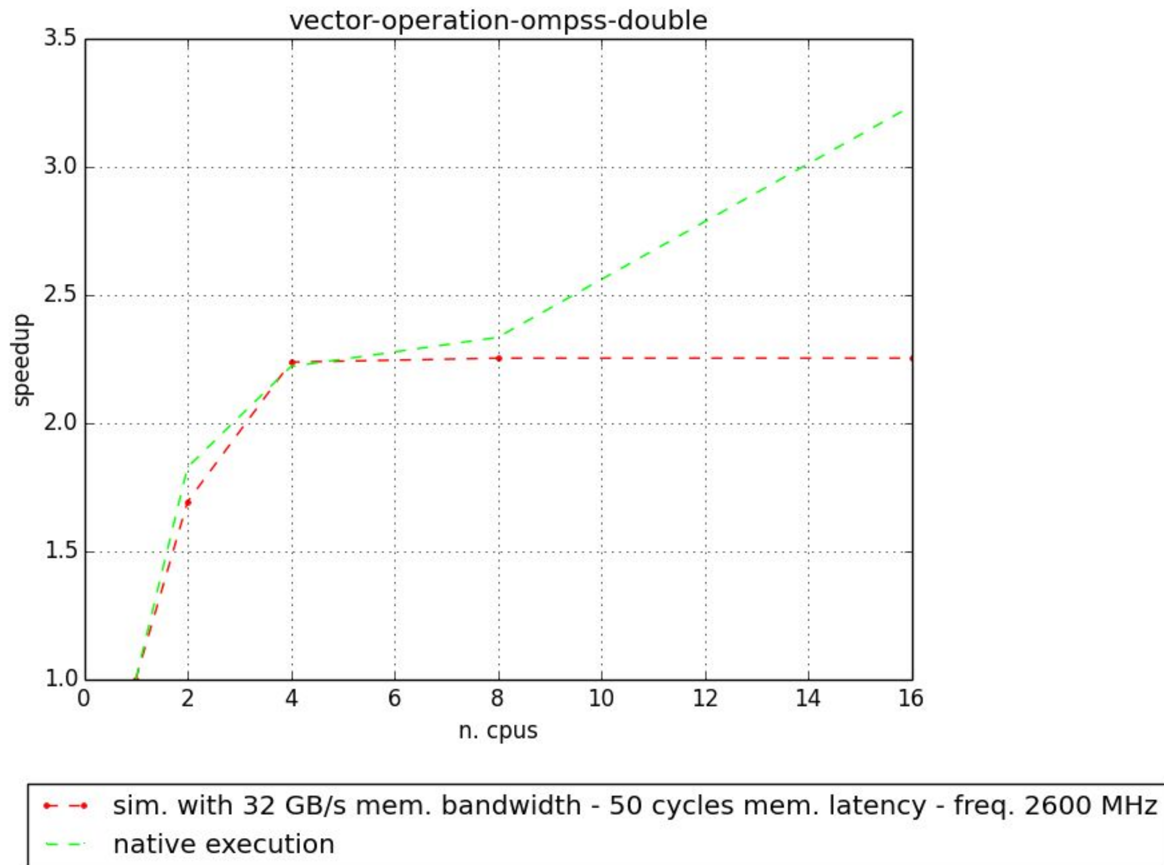


Chart xx: Speedup comparison between native execution and MN3 proposed configuration

Performance evaluation

Now, the simulator execution performance seems to mimic properly the native execution up to 8 CPUs. As already mentioned before, the case of 16 CPUs is not considered due to the inability of the simulator to reflect different memory bandwidths and latencies as one can find on a NUMA configuration like the one of MareNostrum 3. However, now the increase of the memory bandwidth do affects the performance as we expected on a memory bounded benchmark like vector operation.

6.3.7 Seventh proposed configuration

Now that we got a configuration that seems good enough, what we will try to do is to do is a further tuning to try to make the simulation as realistic as possible. In order to do so, we will keep the memory bandwidth and the frequency as it was for the last proposed configuration. The configuration proposed is the following:

- cpu_freq_mhz
  - 2600 MHz
- Memory bandwidth
  - 32 GB/s
- Memory latency
  - 140 cycles

After finishing the executions with different amount of cores, the scalability plot is shown at the following charts. The first one shows only a comparison of the native execution against the configuration proposed. The second chart shows it is the same plot but by just adding also the last configuration.

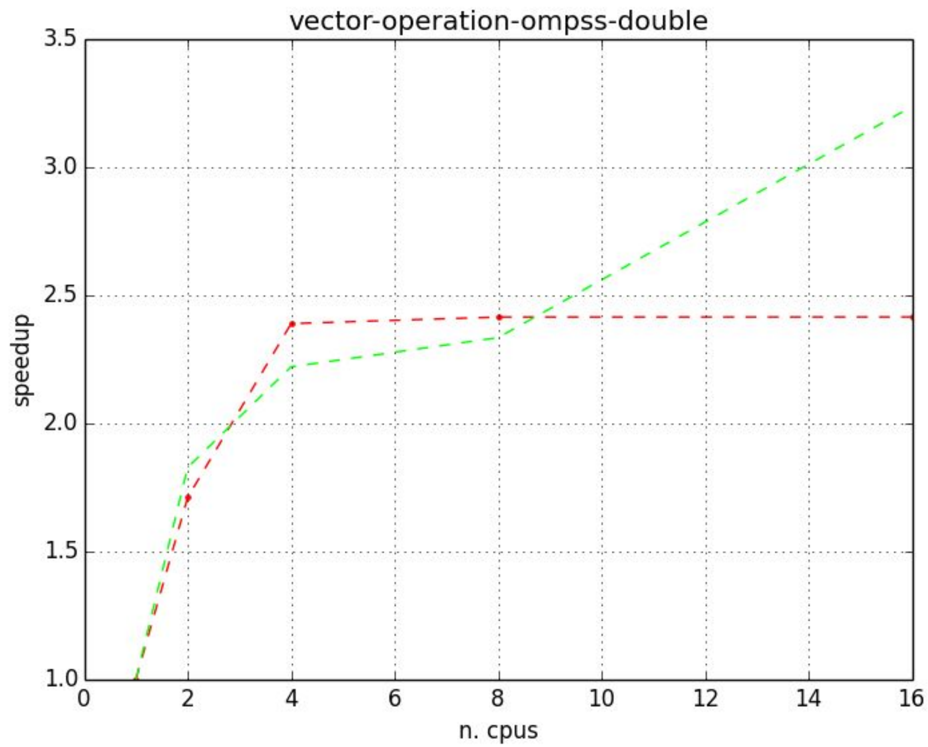


Chart xx: Speedup comparison between native execution and 7th MN3 proposed configuration

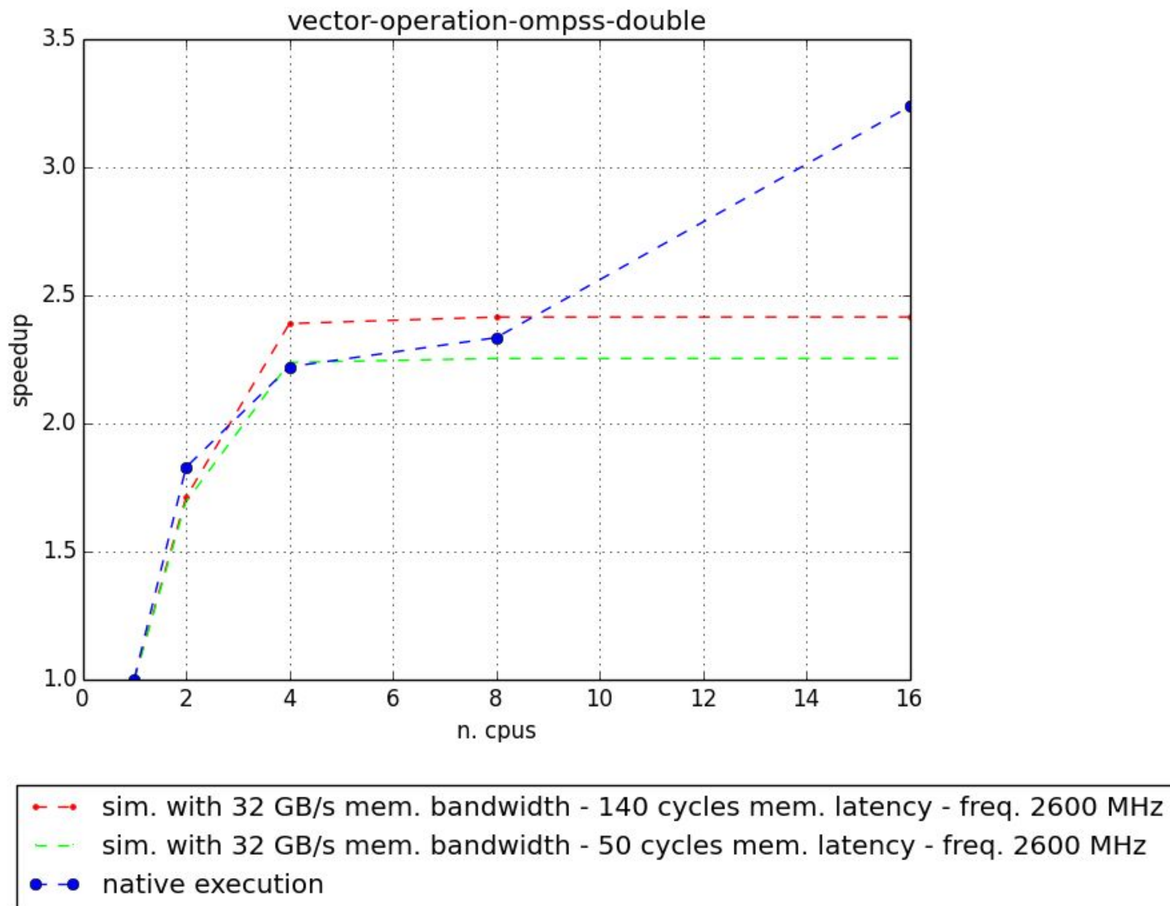


Chart xx: Speedup comparison between native execution and 6th and 7th MN3 proposed configuration

Performance evaluation

We can see at the charts that now, increasing the memory latency actually increases the scalability. This could be explained due to the fact that increasing the memory latency affects the performance of the application for the case of one CPU only. Therefore, we could actually say that this configuration also mimics properly the behavior of the target architecture.

6.3.8 Eight proposed configuration

Stuff to say about, since low latency seem to work really well with double bandwidth here, how the default MN3 configuration with low latency:

At this point, we saw that decreasing the latency seems to do a good job when trying to mimic the target architecture as well as increasing the frequency and the bandwidth. So, at this point, what we wanted to try was to, starting from the MareNostrum 3 default TaskSim configuration (i.e., 100MHz of CPU frequency, 16 GB/s of memory bandwidth and 140

cycles of memory latency), just improve the latency and see if only this change is enough to mimic properly the target architecture. In this line, we propose the following parameters:

- cpu_freq_mhz
    - 1000 MHz
- Memory bandwidth
    - 16 GB/s
- Memory latency
    - 70 cycles

The next chart shows the last two configuration proposed (i.e., 6th and 7h) along with the configuration just proposed and the native execution.

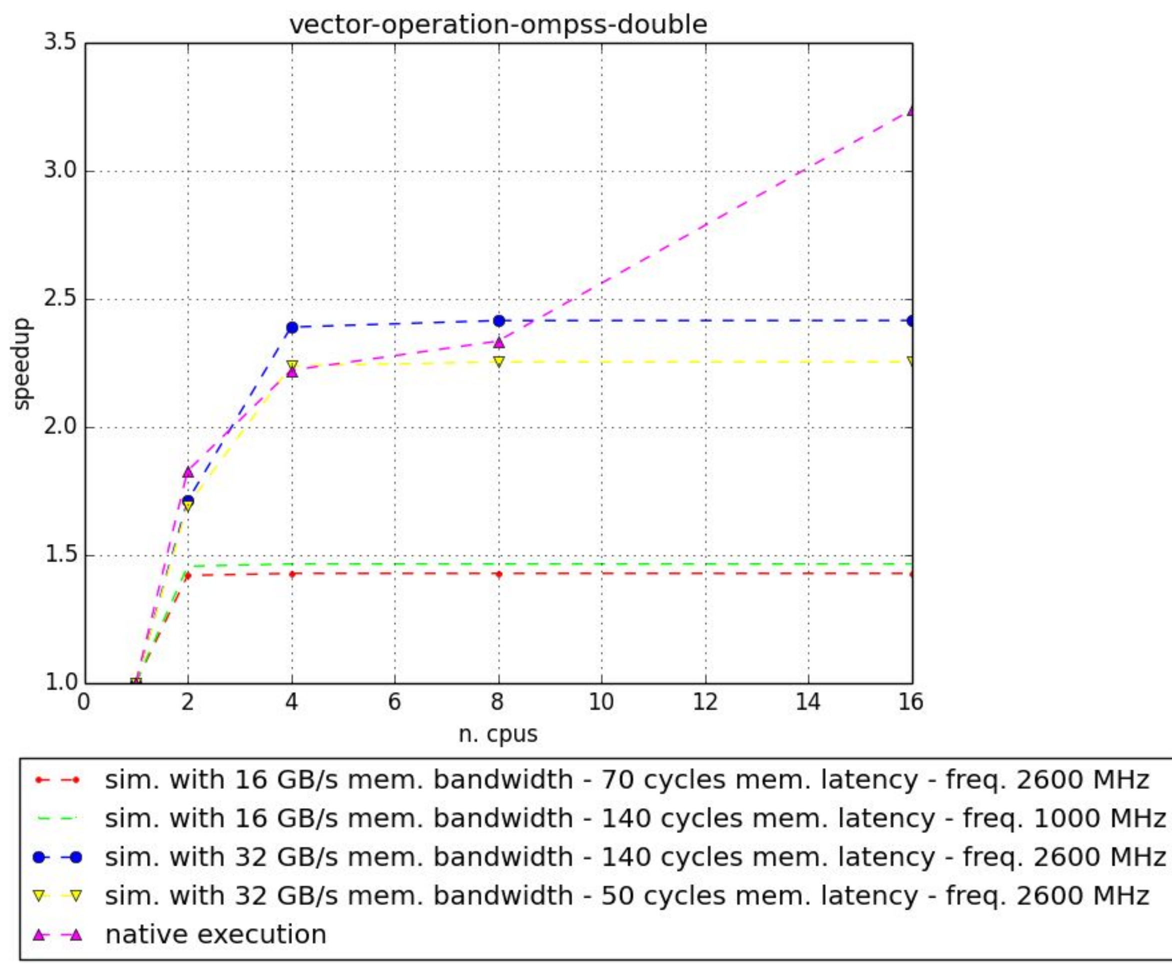


Chart xx: Speedup comparison between native execution and default, 7th and 6th MN3 proposed configuration

Performance evaluation

The performance obtained with this configuration is not the want we wanted at first, so we could discard it.
In summary, it seems like using the default MareNostrum 3 TaskSim parameters does not report a realistic target architecture at simulations, neither decreasing the latency. At other hand, by setting the real CPU frequency and a more realistic memory bandwidth seems a good configuration for representing the target architecture.

6.4 Searching for improvement: sparse-matrix vector multiplication

6.4.1 First proposed configuration

If we take a close look to the sparse-matrix-vector-multiplication comparison:

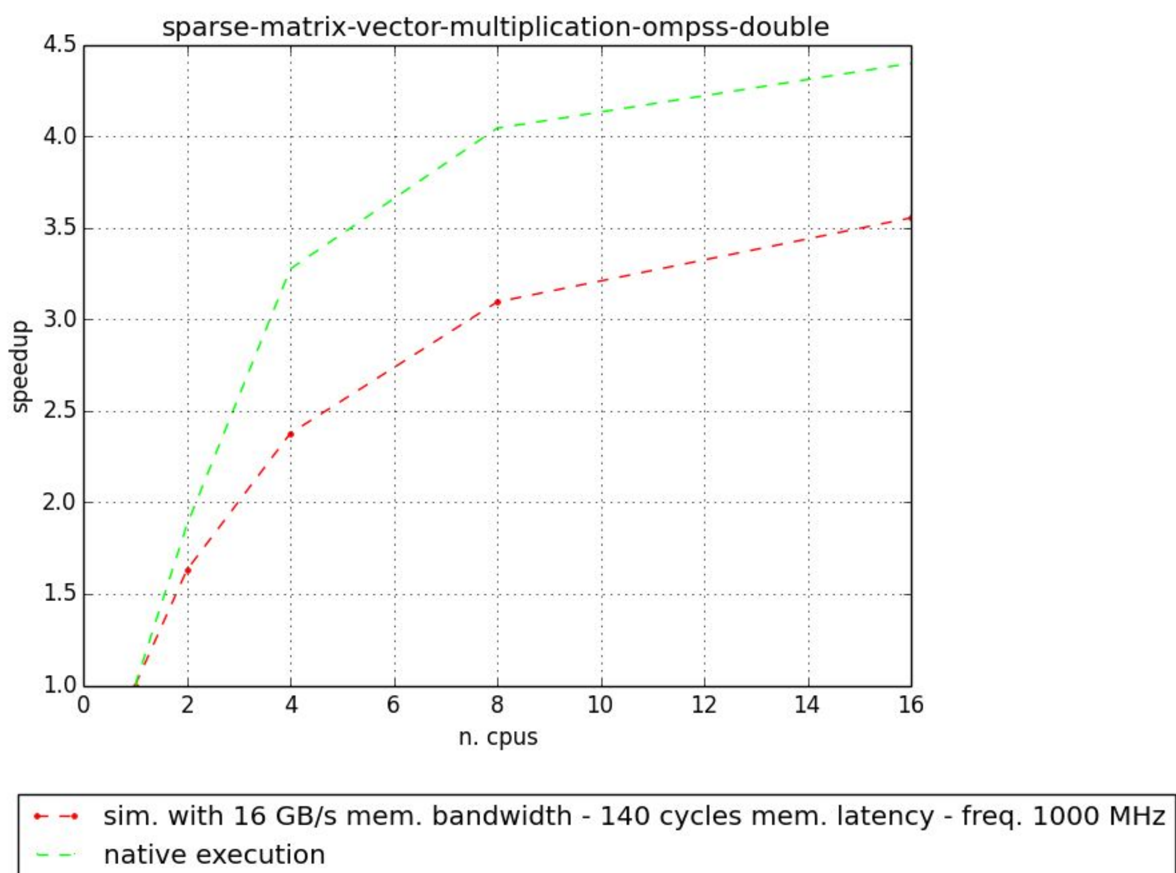


Chart xx: Speedup comparison between native execution and MN3 default configuration

This simulation shows an interesting behaviour, though the default configuration do not obtain the same speedups as the native execution, has a similar curve. Another interesting thing that can be observed here, is that while nearly all the other benchmark simulations are getting better speedups than their homolog native execution, this simulation presents a lower speedup.

In order to get a more feasible simulation, the MN3's configuration will have to be modified. Even if we yet not known the most appropriate parameters for this benchmark, since we know the benchmark behaviour, we can deduce a few point that could us to the right direction:

- Has a random memory access pattern
- It has load unbalance across the different threads, so memory accesses are not uniform across threads

Since we are using the default MareNostrum 3 configuration for TaskSim, it is supposed to have a valid latency value, so it could be that the issue here is the memory bandwidth or the CPU frequency. In this line, we will try first to increase the memory bandwidth. For this, we propose the following TaskSim configuration:

- cpu_freq_mhz
    - 1000 MHz
- Memory bandwidth
    - 32 GB/s
- Memory latency
    - 140 cycles

In the following chart, the speedup obtained with this configuration can be compared with the default configuration speedup and the native execution speedup:

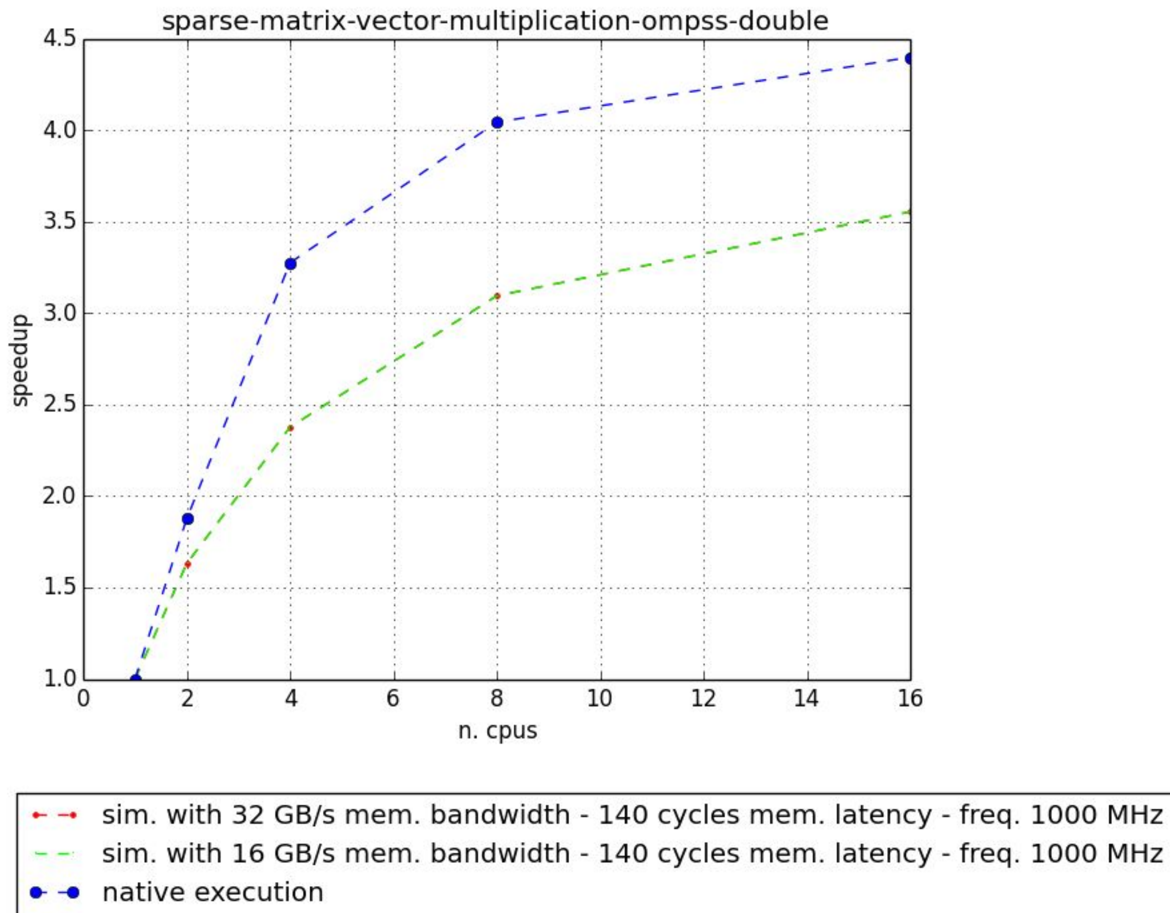


Chart xx: Speedup comparison between native execution and MN3 default configuration

Performance evaluation

Basically the performance with both configurations is basically the same when increasing the memory bandwidth. It actually makes sense when doing computation with sparse matrices. The point is that, all the valuable data is scattered all along the matrices, so little cache of data can be done. This means that memory latency pays a lot when accessing each value.

6.4.2 Second proposed configuration

From the last configuration proposed, it seemed like it was good enough to present the same behavior but with different values. Since what we did was to improve different values, now, starting from the default configuration we will try to increase the memory latency in order to check if this change could lead to a more precise target architecture simulation. The following configurations is proposed:

- cpu_freq_mhz
    - 1000 MHz
- Memory bandwidth
    - 16 GB/s
- Memory latency
    - 280 cycles

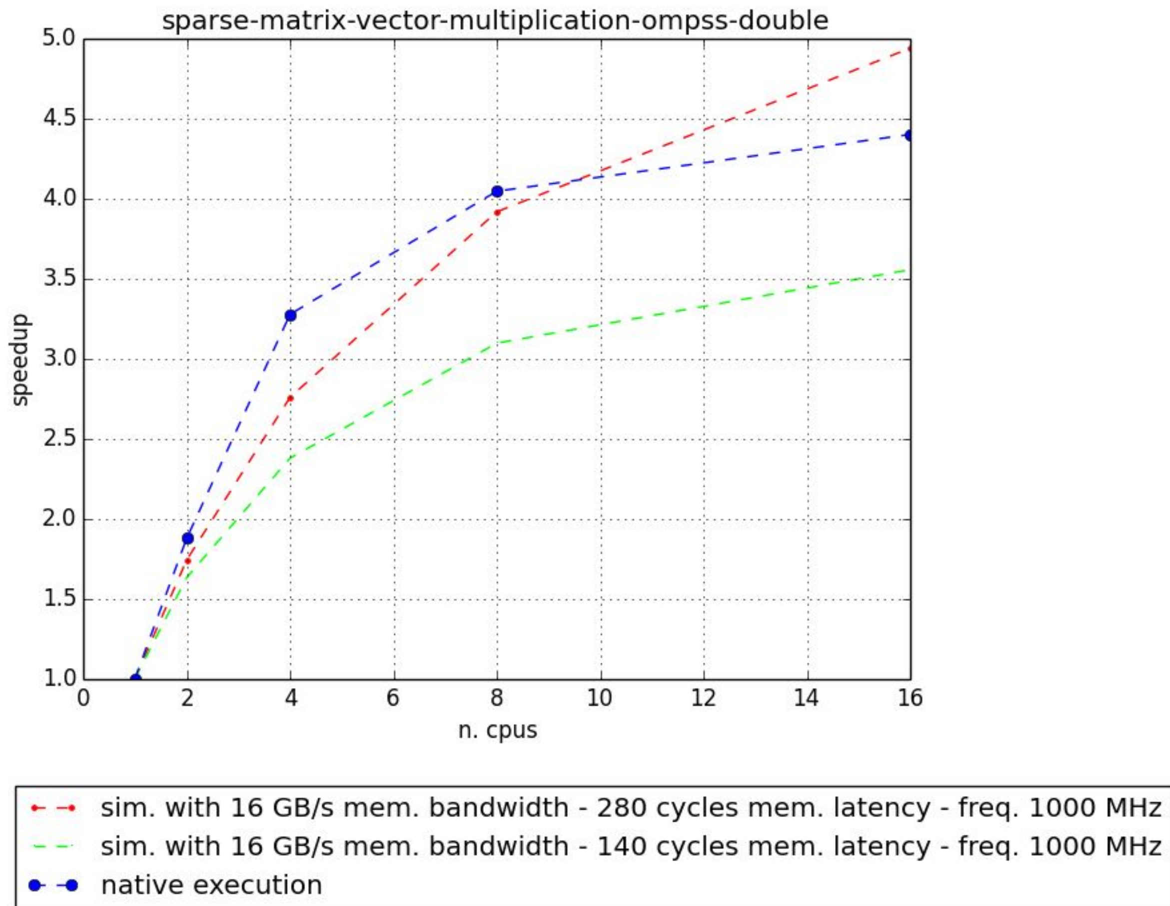


Chart xx: Speedup comparison between native execution and MN3 default configuration

Performance evaluation

We can see how increasing the latency lead to a more similar curve, compared to the native execution, than the obtained with the first proposed configuration. What we can understand from these results is that the real memory latency from MareNostrum 3 is higher than the one from the TaskSim configuration or that the effective memory latency from this benchmark is not the same as the real one due to issue with contention or umbalance within the benchmark. More research has to be done anyhow.

6.4.3 Third proposed configuration

Now, since MareNostrum 3 CPUs actually run at 2600 MHz, we will try to modify the TaskSim parameters so the only value modified is the frequency. For this, we propose the following configuration:

- cpu_freq_mhz
  - 2600 MHz
- Memory bandwidth
  - 16 GB/s
- Memory latency
  - 140 cycles

The next chart shows the results obtained.

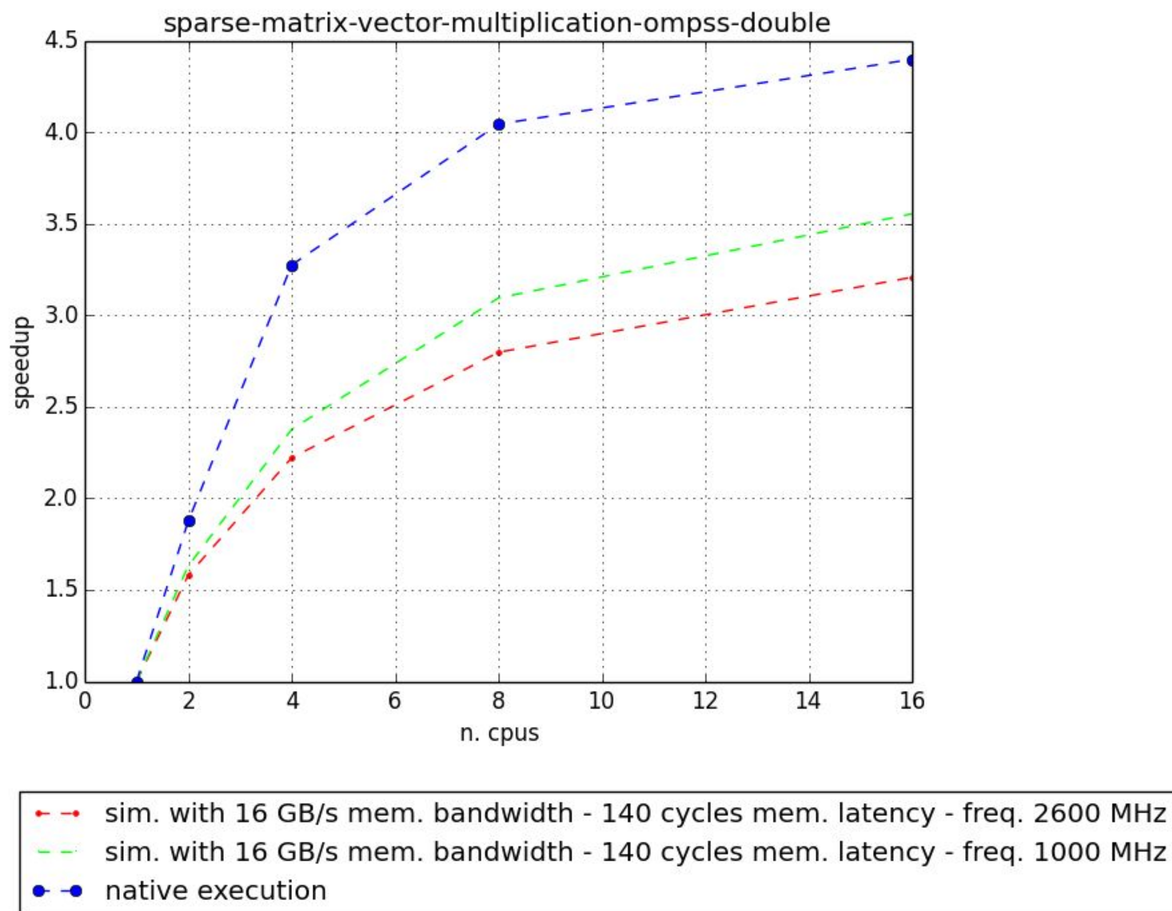


Chart xx: Speedup comparison between native execution and MN3 default and proposed configuration

Performance evaluation

The result obtained is even worse than the default configuration with a non-realistic CPU frequency. The explanation for this is an increment for the proportional increment of the memory latency overhead.
This means that, when increasing the frequency, the portions of code that do computation spent less time than before, while the memory accesses spent the same amount of time. This leads to more percentage of time waiting for data while the computation time decreases. This leads to an even worse scalability.
Finally, we can see that the curve is very similar, so increasing the frequency is not the solution we were looking for.

6.4.4 Fourth proposed configuration

Starting from the last configuration proposed, we want to know if the memory latency is the issue we were looking for. So, we will try to minimize the memory latency to its minimum. The following configuration is proposed:

- cpu_freq_mhz
  - 2600 MHz
- Memory bandwidth
  - 16 GB/s
- Memory latency
  - 1 cycles

The only problem we had with this configuration is that it spent that much time to execute that it was virtually impossible to run the simulations. This is still an open issue with the TaskSim developers. Basically it seems that, when using almost zero memory latency, the simulator does not work properly.

6.4.5 Fifth proposed configuration

After the issue of the last configuration, we will try to use an small memory latency value, but not that small. The next configuration is proposed:

- cpu_freq_mhz
  - 2600 MHz
- Memory bandwidth
  - 16 GB/s

- Memory latency
    - 50 cycles

This time the executions were performed correctly, providing results that are presented on the following chart:

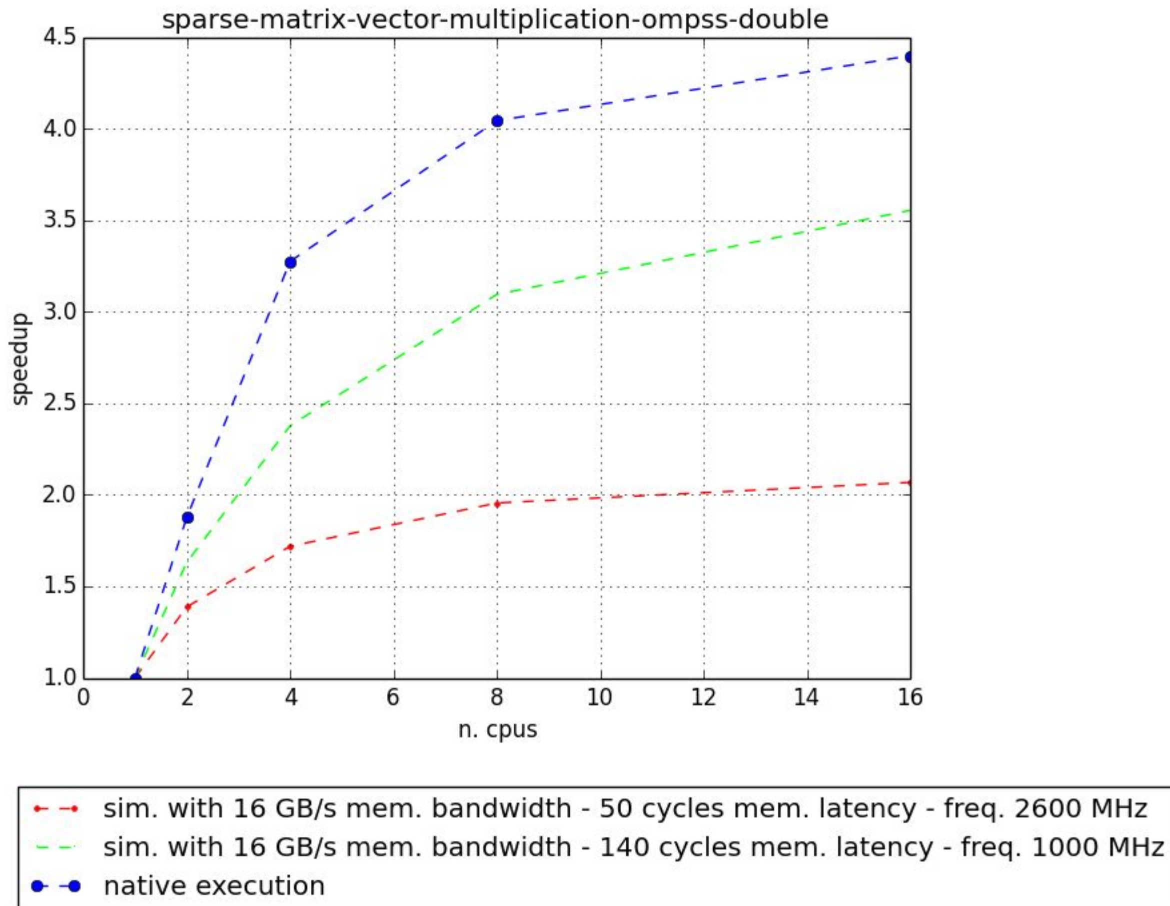


Chart xx: Speedup comparison between native execution and MN3 default and proposed configuration

Performance evaluation

Decreasing the memory latency has the opposite effect as the want we expected. Actually, we were expecting a better scalability than the native execution, or at least closer.

The following chart show all the configuration tested until now:

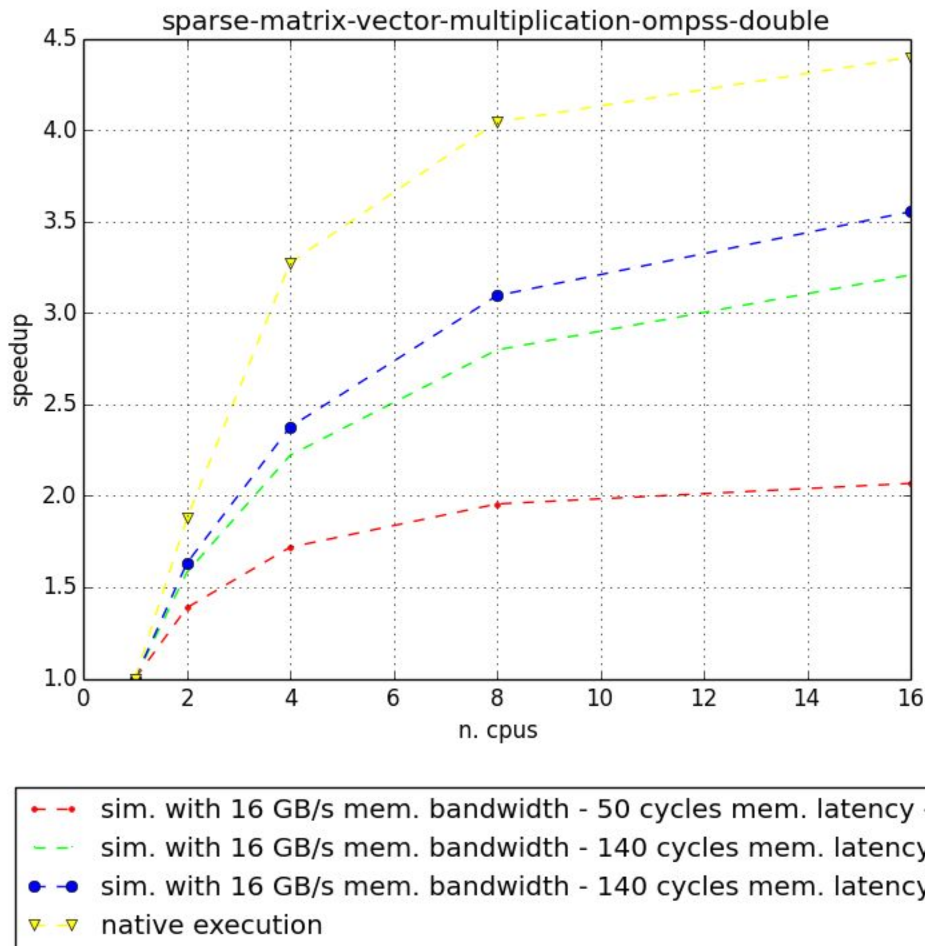


Chart xx: Speedup comparison between native execution and 3rd and MN3 default and proposed configuration

We can see how the closer simulation to the target architecture is the default TaskSim's MareNostrum 3 configuration, even though is not good enough.

6.4.6 Sixth proposed configuration

Even though the benchmark studied is not memory bandwidth bounded, we will try to increment it to see how it affects the simulation of the benchmark. In this line, we propose the following configuration:

- cpu_freq_mhz
  - 2600 MHz
- Memory bandwidth
  - 32 GB/s
- Memory latency

○ 50 cycles

The following chart shows the results obtained.

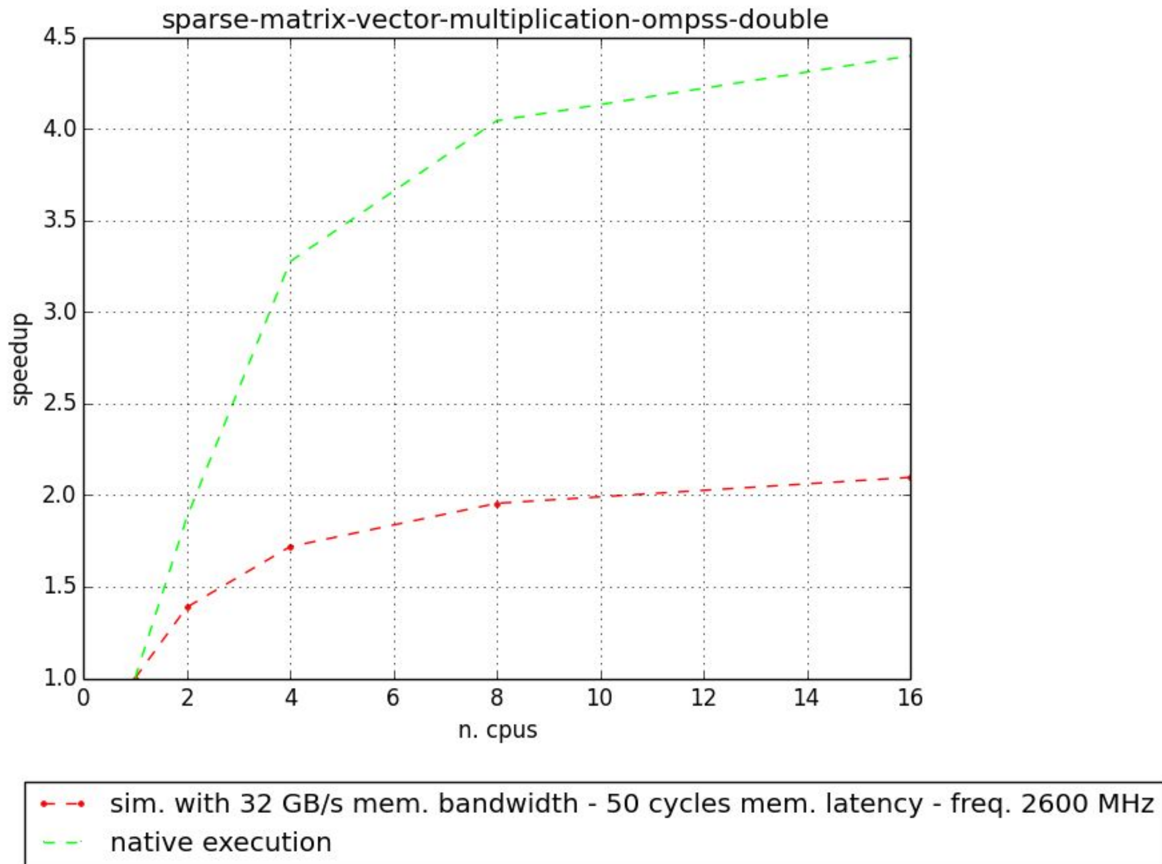


Chart xx: Speedup comparison between native execution and MN3 proposed configuration

Performance evaluation

So, as expected, increasing the memory bandwidth does not have any impact on the simulation.

6.4.7 Seventh proposed configuration

Even though the last configuration was not modifying anything, now we will try to use the default MareNostrum 3 value for memory latency. In this line, the following configuration is proposed:

- cpu_freq_mhz
  - 2600 MHz

- Memory bandwidth
  - 32 GB/s
- Memory latency
  - 140 cycles

The following chart presents the results we obtained from running scalability experiments.

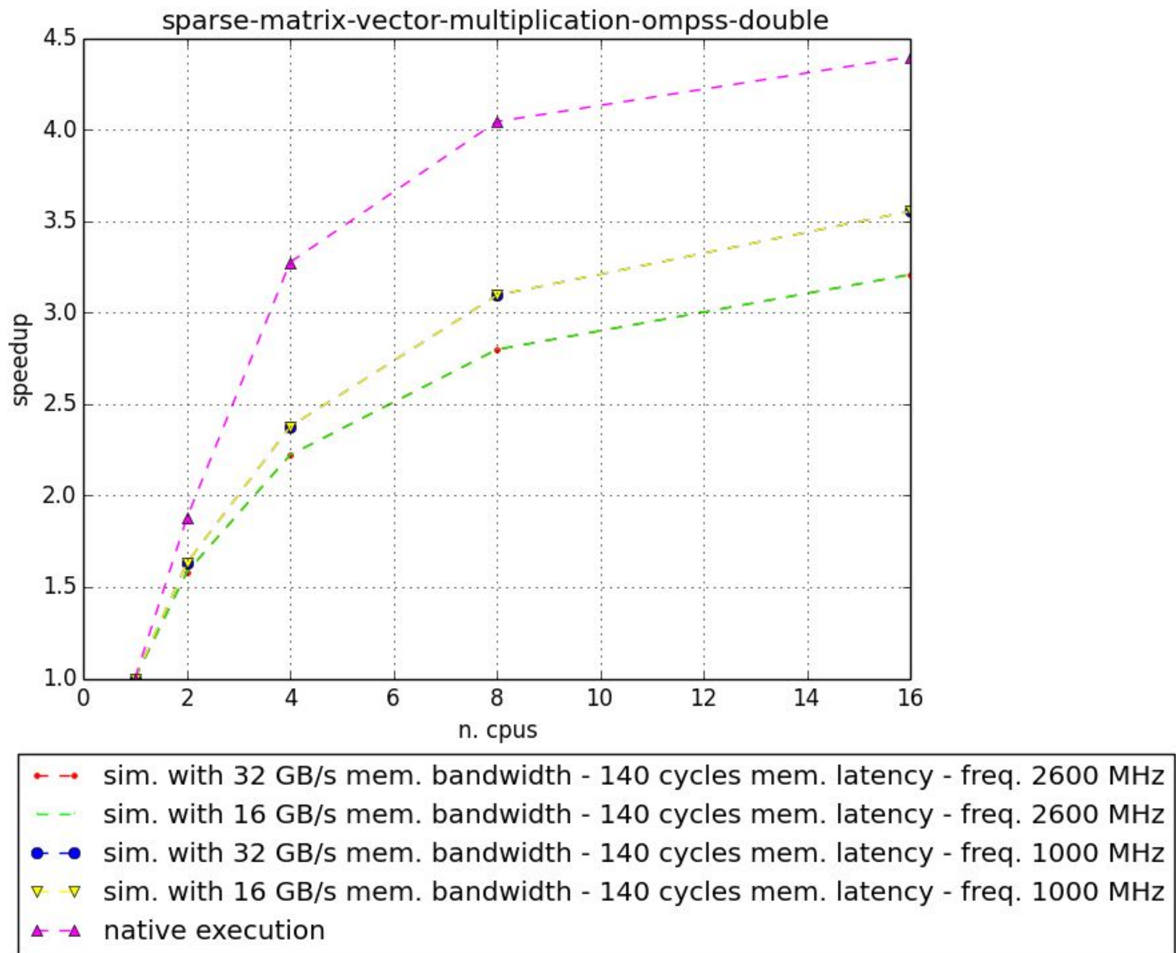


Chart xx: Speedup comparison between 7th, 3rd, 1st proposed modifications, default configuration and native execution

Performance evaluation

The proposed configuration actually presents almost the same scalability curve from the third proposed configuration. So, we are again with the same curve and close to the same performance results than in the native execution at the target architecture.

6.4.8 Eight proposed configuration

Now, starting from the default MareNostrum 3 default TaskSim configuration, we try if dividing by two the memory latency affects somehow the results obtained. The following list shows the configuration used for the experiments.

- cpu_freq_mhz
  - 1000 MHz
- Memory bandwidth
  - 16 GB/s
- Memory latency
  - 70 cycles

The following chart shows the results obtained:

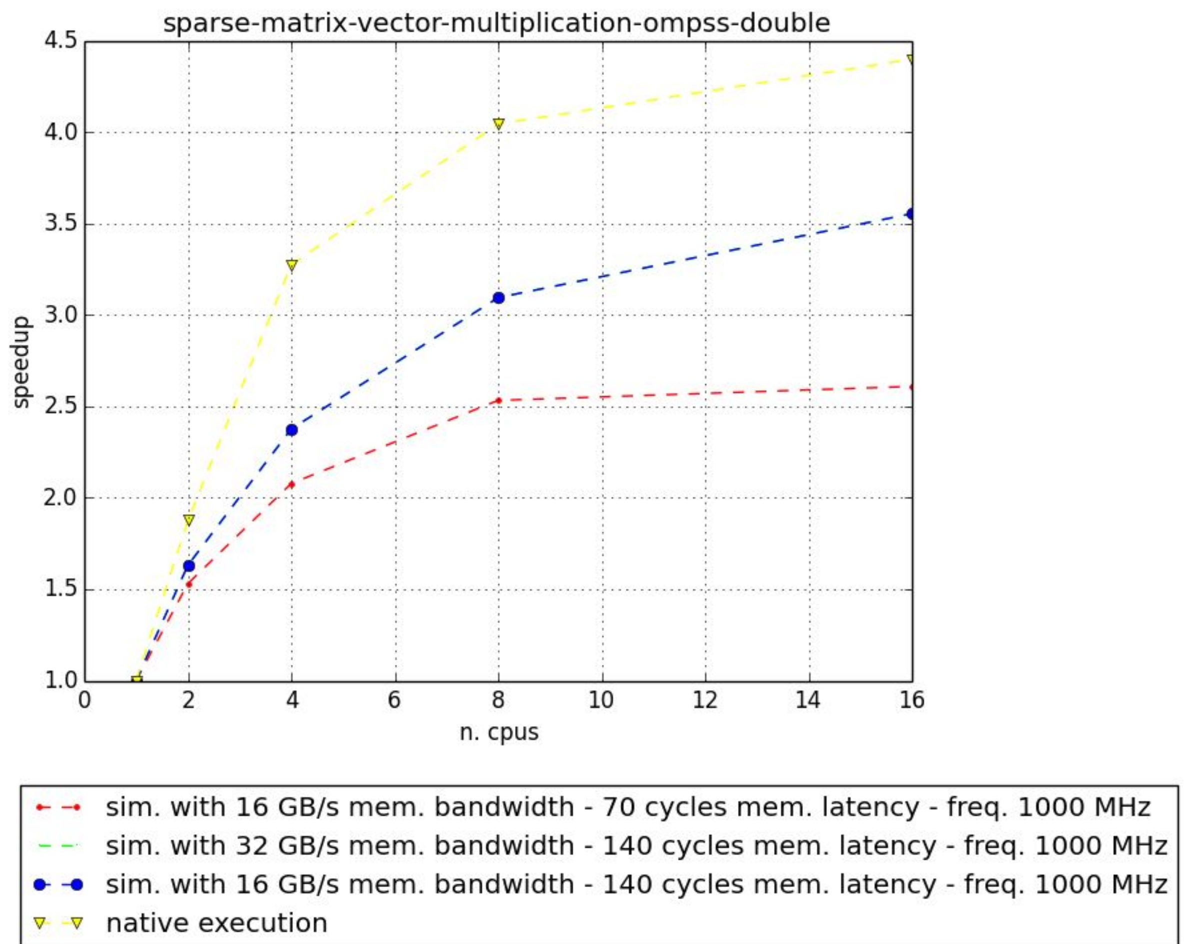


Chart xx: Speedup comparison between 8th and 1st proposed modifications, default configuration and native execution

Performance evaluation

As can be seen on the charts, modifying the bandwidth does not change anything on the simulator. Basically, the most close that we got to the native execution was by using the default configuration for MareNostrum 3. Besides this fact, it seems that modifications on the CPU frequency and memory latency are the determinant factors from the results observed across all the configurations proposed.

6.5 Issue detected (1)

We know that MareNostrum 3 has a latency of 140 cycles, but if we take a look at the following charts, we can see something interesting.

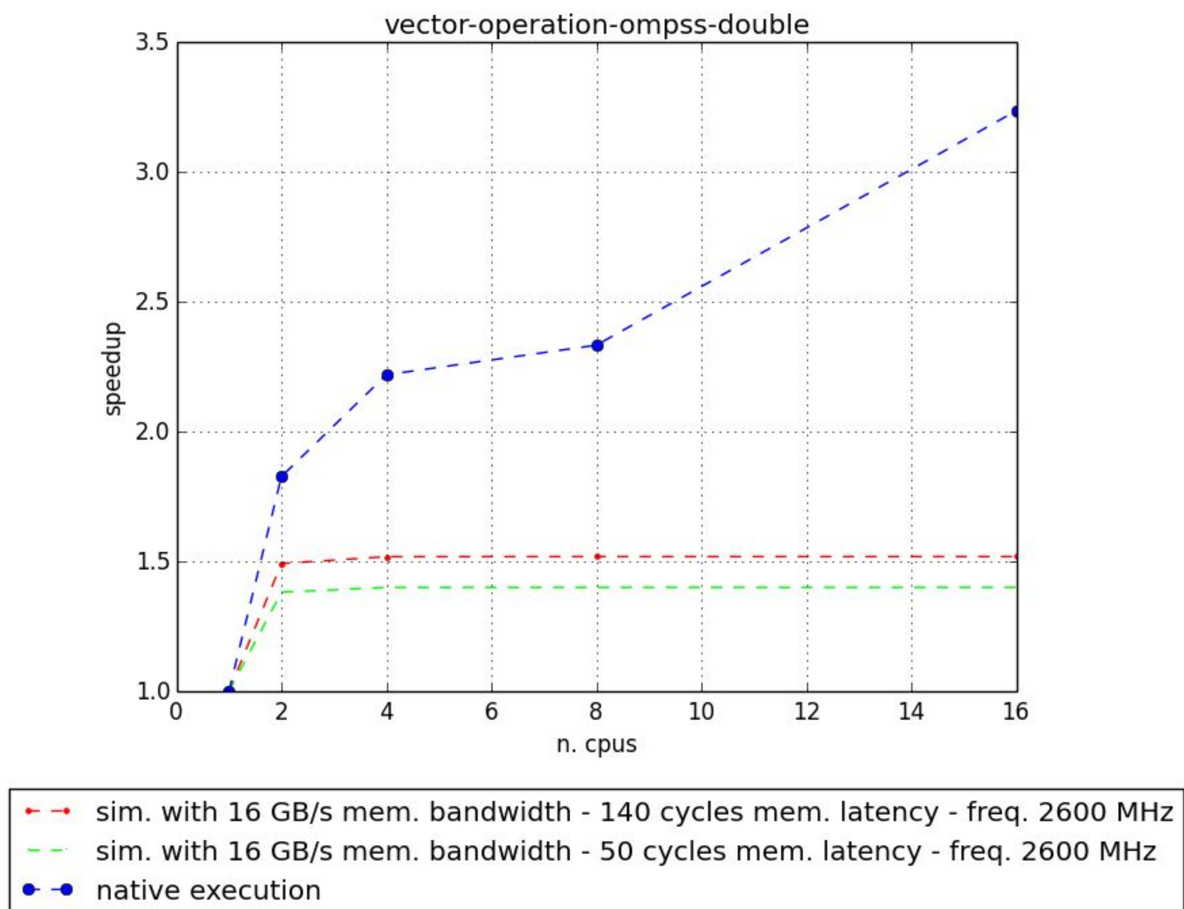
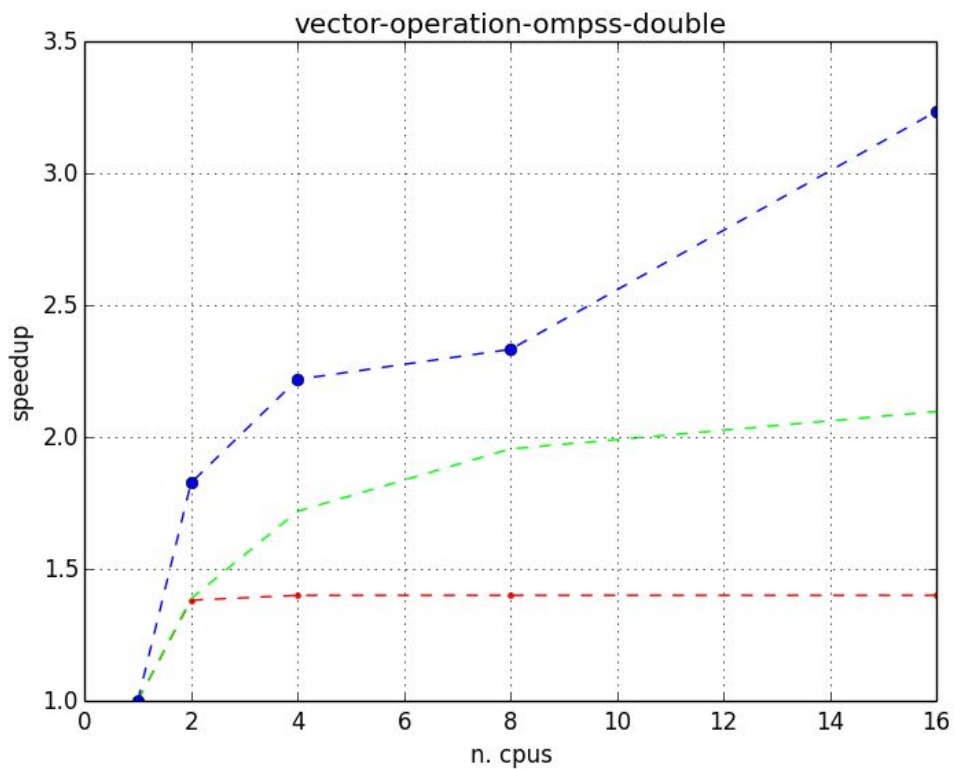


Chart xx: Speedup comparison between 3rd and 5th proposed modifications and native execution

In the chart [3rd vs 5th vs native] we can see that changing the memory latency doesn't generate big difference in speedups. Actually, decreasing the latency basically reduces scalability.



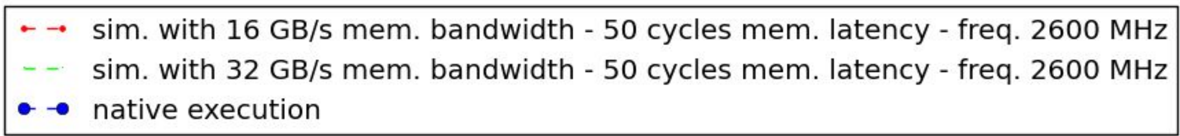


Chart xx: Speedup comparison between 5th and 6th proposed modifications and native execution

In the chart [5th vs 6th vs native], the difference in memory bandwidth generates quite a difference in the speedups, thus improving scalability while increasing number of threads. This is understandable, since the vector-operation benchmark is memory bandwidth bounded as we saw at section 3.1.

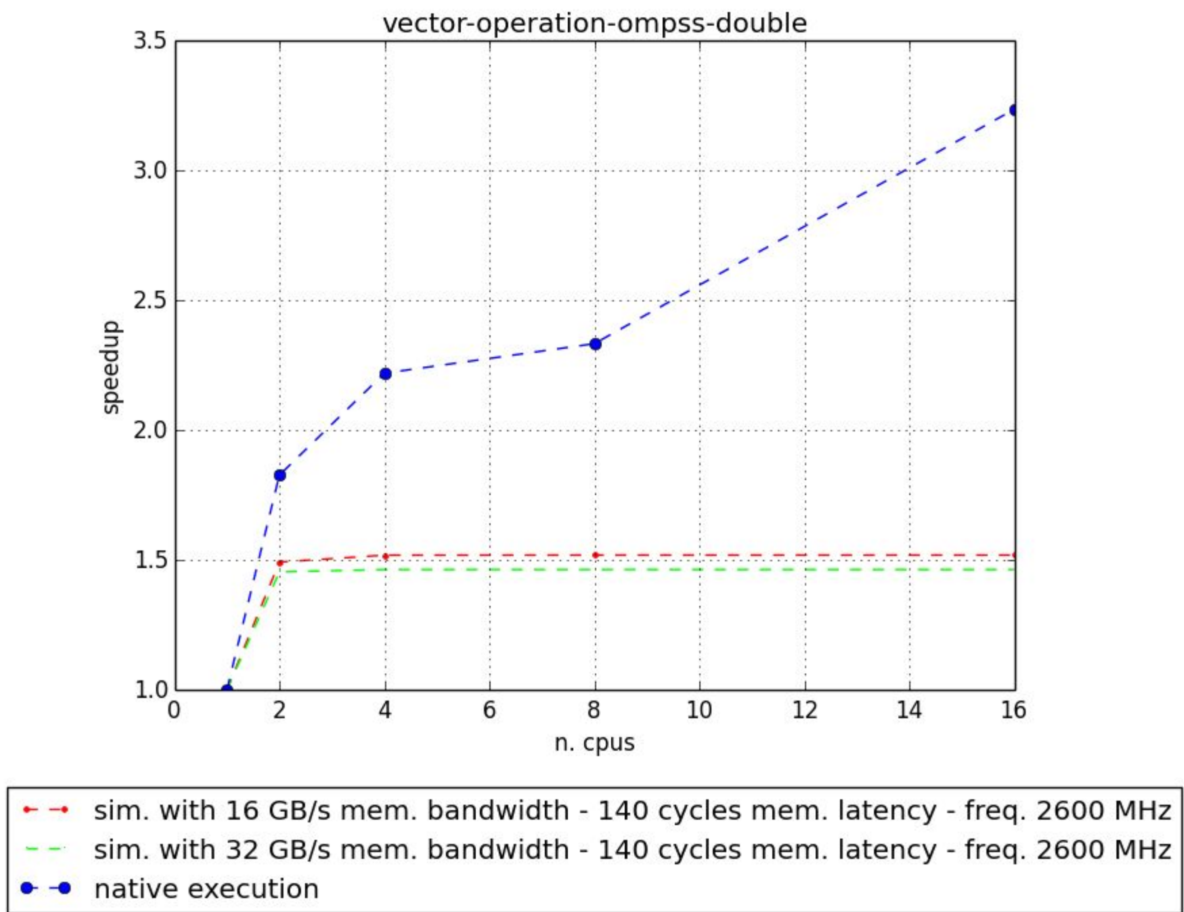


Chart xx: Speedup comparison between 3rd and 7th proposed modifications and native execution

In this last chart [3rd vs 7th vs native], we found unexpected results. With a memory latency of 140 cycles (remember that is the native memory latency of MareNostrum 3), the difference in bandwidth does not impact the performance, which translates into a bad scalability after four threads or more.

Vector operation benchmark is not supposed to be impacted by memory latency, but from the plots we can see that, modifying memory bandwidth has an impact depending on the memory latency selected.

6.5.1 Solution

In order to investigate the weird behavior when modifying latencies and memory bandwidth, we contacted the developers of TaskSim to try to discover what was really happening. After testing and debugging both our executions and the TaskSim simulator itself, we discovered that the computation of the latency that we have been made during the experiments was not correct at all. Basically, the problem was that, the memory latency value is used internally

after some computations and not as it is introduced. So, at the end, we found that the value of the memory latency has to be introduced as nanoseconds.

Actually, more useful information was gathered from these meetings with the TaskSim developers. In this case, we also found that the CPU frequency introduced at the configuration files is not used as it is as well, meaning that some computations with the value are done before using it internally at the TaskSim simulator. At the end, we found that this was the reason why the default MareNostrum 3 TaskSim configuration used 1000 MHz as a CPU frequency and not 2600 MHz as it is indicated at the spreadsheet of the processor that features each of the compute nodes of MareNostrum 3.

The following part of this section will explain how we decided the new default parameters for the TaskSim simulator so the values introduced were the same as the ones of the target architecture after all the transformations that TaskSim applies to the values.

Correct latency calculation

The values we want to use are the following:

- Frequency = 2.6 GHz
- Latency = 140 cycles

So, in order to know which values have to be introduced at the TaskSim configuration files, we have to apply some formulas. The following computation is the one that led us to the final value.

$$(140 * 10^{-9} cyc) / (2.6 * 10^{9}\ cyc/s) \Rightarrow (140 * 10^{-9} cyc * s) / (2.6 * 10^{9} cyc) \Rightarrow$$
$$(140 * 10^{-9} s)/(2.6 * 10^{9}) \Rightarrow 140ns/(2.6 * 10^{9}) \Rightarrow 54ns$$

From now on, the rest of the comparisons will take into account this number as the native execution memory latency. The following chart shows a comparison against different latencies configurations.

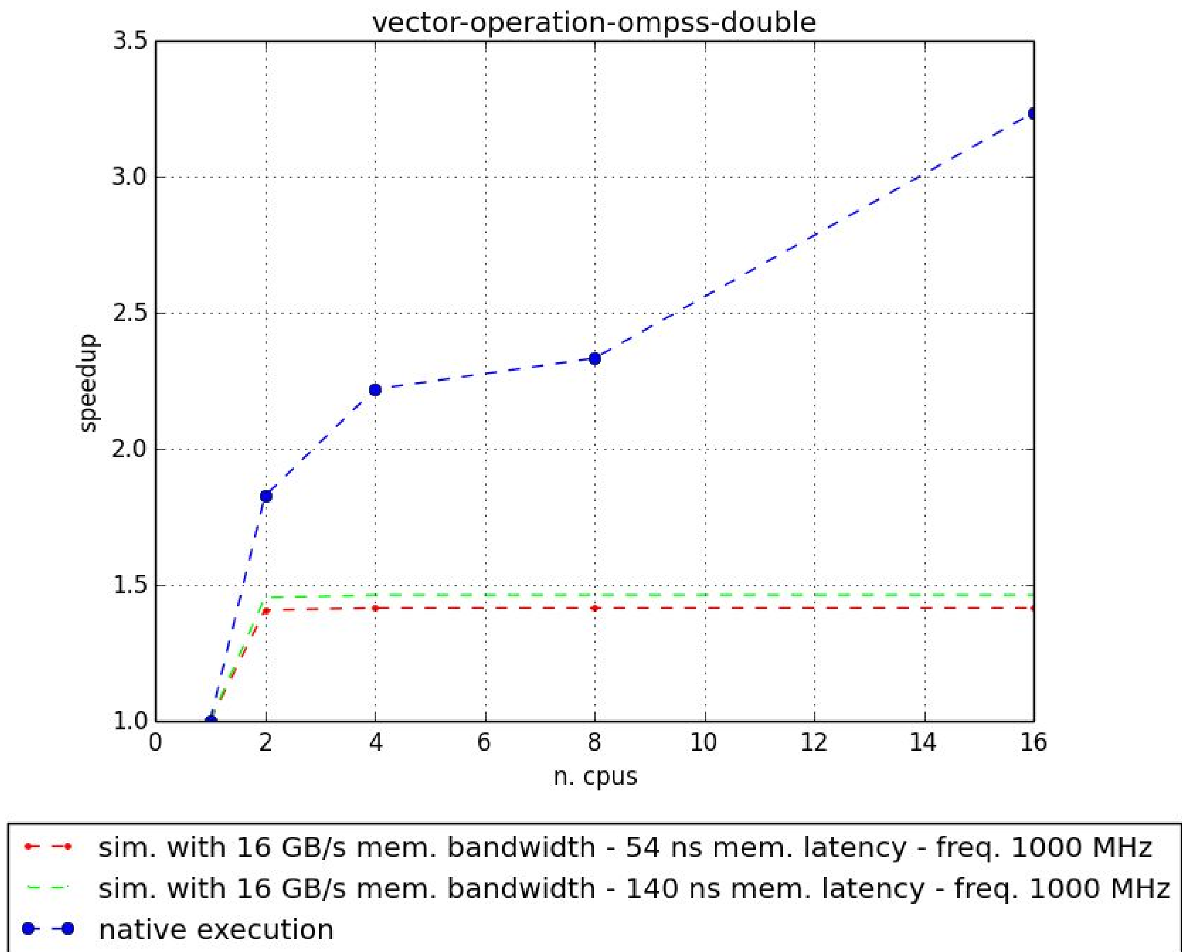


Chart xx: Speedup comparison between new configuration, default configuration and native execution

For vector operation we didn't see that much change. Basically, this benchmark is affected by the memory bandwidth more than for the memory latency, so the behavior observed is the expected.

6.6 Issue detected (2)

As said before, several meetings were done with the TaskSim developers. Besides the issues with the configuration files, we also detected some other "issue" for the benchmarking. Basically, after finishing the simulation, TaskSim reports the cycle count of it.

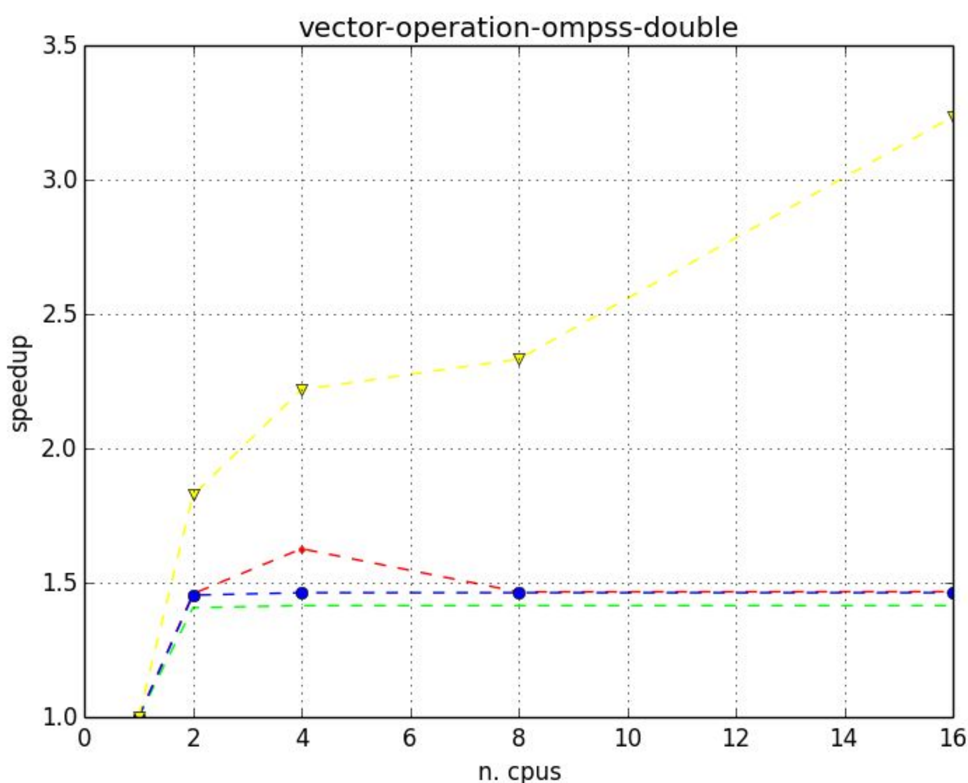
As already mentioned, the TaskSim simulator uses a combination of execution-driven and trace-driven simulations to improve its overall performance and thus decrease its execution time. This means that, some parts of the code (i.e., the parallel parts) are actually simulated while the serial parts are executed natively.

In summary, what the TaskSim developers told to us is that the reported cycle count is for the total execution (i.e., simulated plus natively executed parts), and not only for the parallel parts of the code as we initially thought.

Now that we fixed the issues we had with the values of the memory latency and the CPU frequency at the configuration files, as well as the issue with the metric reported, we wanted to see if these solutions fixed the weird behavior we found during all the experiments we have made.

In order to do so, more experiments have been run with the new configuration. The experiments were only repeated for the vector operation and the sparse matrix vector multiplication benchmarks, the reason is that they were the only ones that showed a different behavior when comparing the simulation against the native execution.

The following charts show how this change affected the performance of the simulations for the vector operation benchmark.



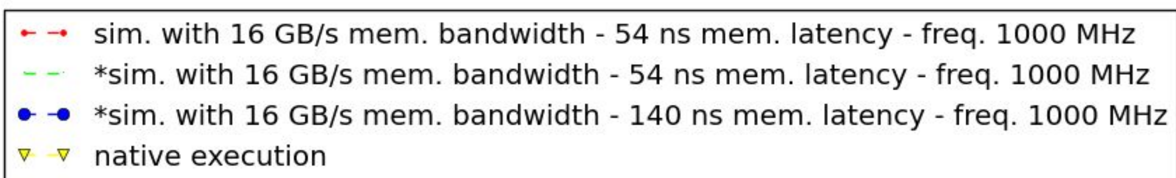


Chart xx: Speedup comparison between parallel speedup/correct latency, total speedup/correct latency, total speedup/incorrect latency and native execution

At Chart [Just up there ^] we saw the scalability of the native execution against the one from different simulations with different parameters. The star on the label indicates that not only the simulated part of the benchmark (i.e., the parallel part) but also the serial part for computing the speedup. The absence of a star on the label indicates that only the simulated part (i.e., the parallel part) has been used for computing the speedup.

Now, we can see that the behavior is actually different when computing the speedup by only using the parallel parts of the code. Also, only considering these parts gives a more fair metric since the speedup reported by the native execution was computed by only considering the parallel part of the benchmark.

Even though, the behavior observed is not as close as the behavior observed at the native execution.

6.6.1 Solution

Now that we know which are the exact metric that TaskSim reports (which is not the same as the one we want), we need to know how can we obtain the metric we want (i.e., cycle count of simulated code).

In order to do it, we used Extrae and Paraver. The first one was used to obtain traces from our simulations while the second one was used for visualizing, analyzing and gathering the metrics from the traces.

6.6.2 Performance evaluation

Now that both issues have been solved, it is time to rerun our simulations in order to see if they mimic better the native execution.

At chart [the one of the next page] we can see the scalability results for the sparse matrix vector multiplication benchmark. Again, a star in the label indicates that the cycle count from the whole execution was used and the absence of it that it was only used the cycle count of the parallel part.

What we can see is that, for the simulation measuring only the parallel part of the benchmark, the scalability is almost perfect. This does not represent at all the native execution, but, it opens more room for making more experiments since we detected that, for this benchmark, it makes a difference to measure only the parallel parts or the whole execution.

Anyhow, what we can also see is that, the behavior of the simulations is more close to the target architecture when the whole execution was computed. At least, this is true when considering how the speedup changes as we increase the number of CPUs used. This behavior should be studied in a deeper detail to try to understand it.

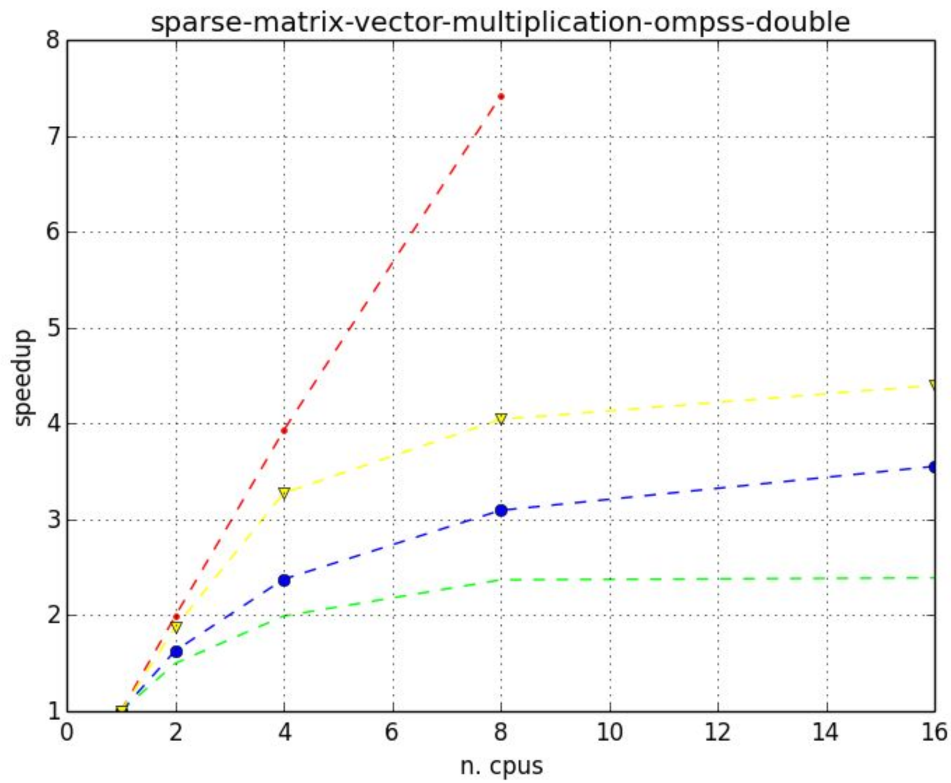


Chart xx: Speedup comparison between parallel speedup/correct latency, total speedup/correct latency, total speedup/incorrect latency and native execution

6.7 New Starting point

Basicament, dir que de les primeres simulacions, de 1000 MHz 16 GB/s 140ns es fa una comparativa entre simulació contant temps total, simulaicó contant temps parallel i natiu.

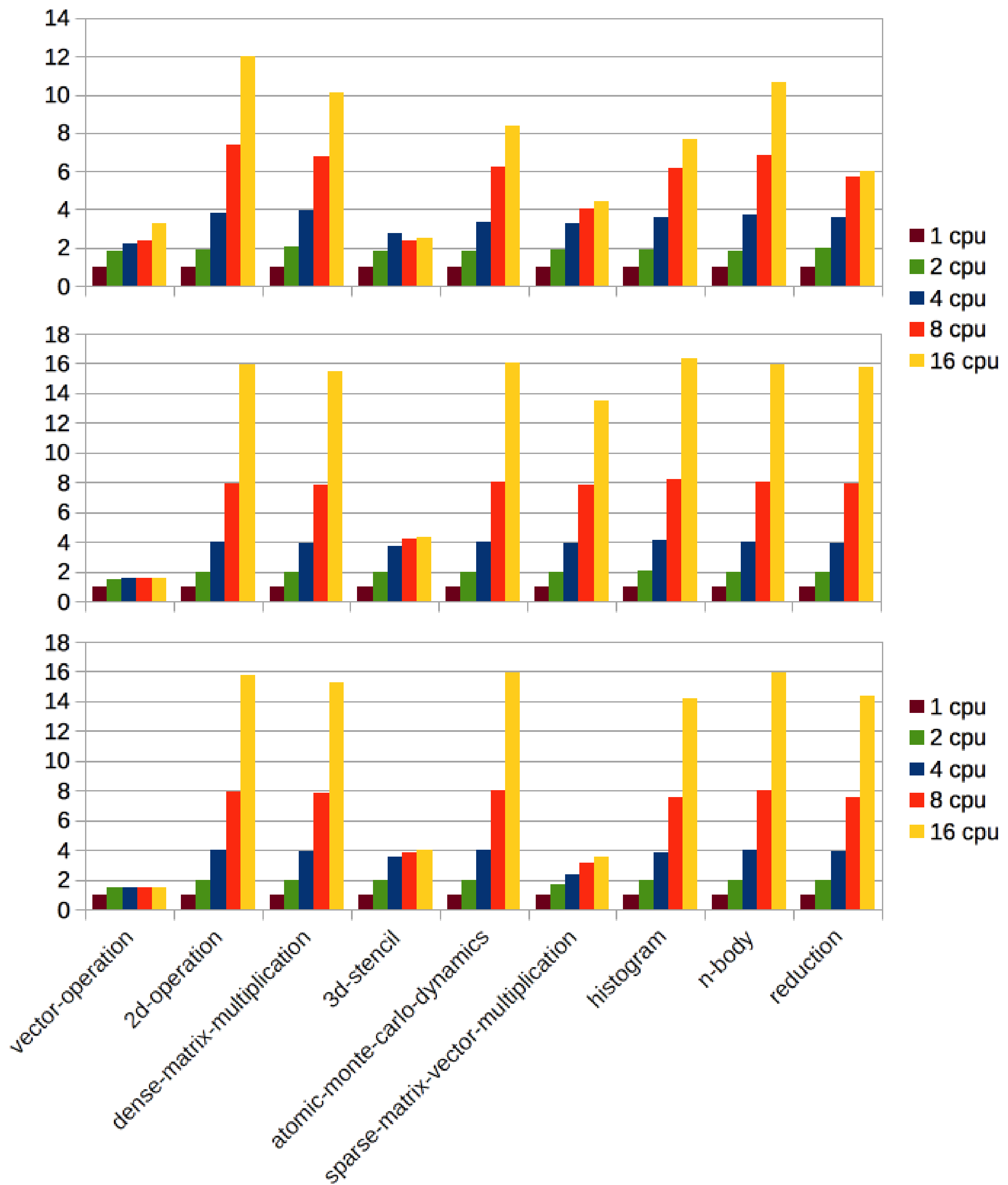


Chart xx: Top: native Mont-Blanc benchmarks suite parallel section speedups.

Middle: simulated Mont-Blanc benchmarks suite parallel section speedups.

Bottom: simulated Mont-Blanc benchmarks suite total simulation speedups.

On the chart above[] we can find a comparison, for all the benchmarks from the suite, between the parallels section speedup of the native execution. In the middle we see the total simulation's speedup. Finally, in the bottom there is the parallel section simulation's speedup.

As we can observe, we cannot find a significant difference for most of the benchmarks. Only the sparse-matrix-vector-multiplication shows significant difference. Also, vector, histogram and reduction show a slight difference.

In fact, we can say that there is not a lot of difference between using the parallel section cycle count or the total simulation cycle count in order to realize the speedups. Still, some benchmarks do show a difference. Knowing that, from now on, the speedups will always realized using the parallel section cycle count.

Meaning that, the parameters that will be used for the new default configuration will be:

- cpu_freq_mhz
    - 1000 MHz
- Bandwidth
    - 16 GB/s
- Latency
    - 54 ns (140 cycles)

It is time to keep searching for more improvements.

6.8 Searching for improvement: bandwidth study

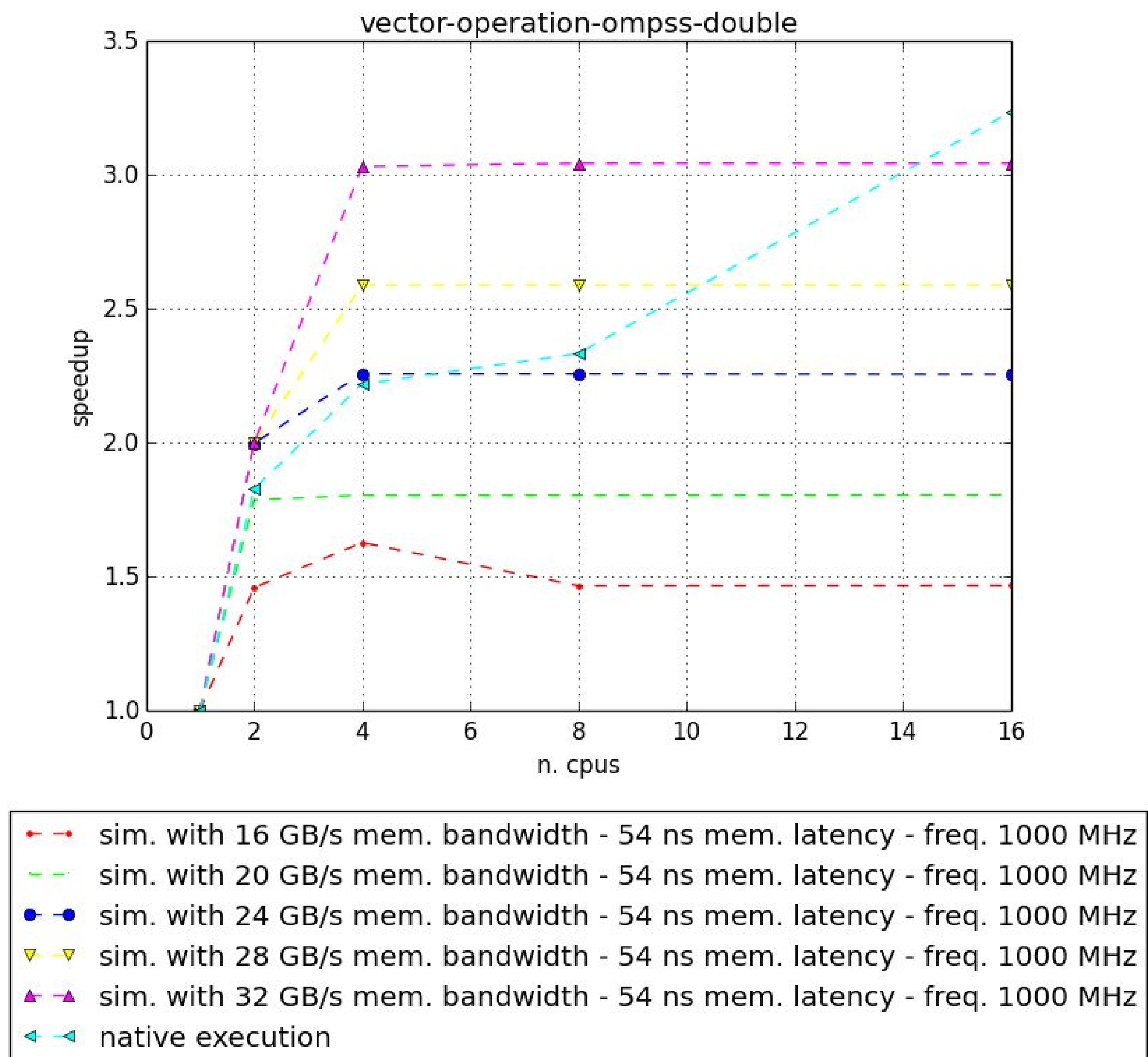
A really good approach in order to improve the vector-operation benchmark, which still show under performance than the native executions, is to realize a bandwidth study, since it is a bandwidth dependent benchmark.

A range of bandwidth parameters will be proposed, simulated and studied. The proposed bandwidth parameters are:

- cpu_freq_mhz
    - 1000 MHz
- Bandwidth
    - 16, 20, 24, 28, 32 GB/s
- Latency
    - 54 ns (140 cycles)

6.8.1 Performance evaluation

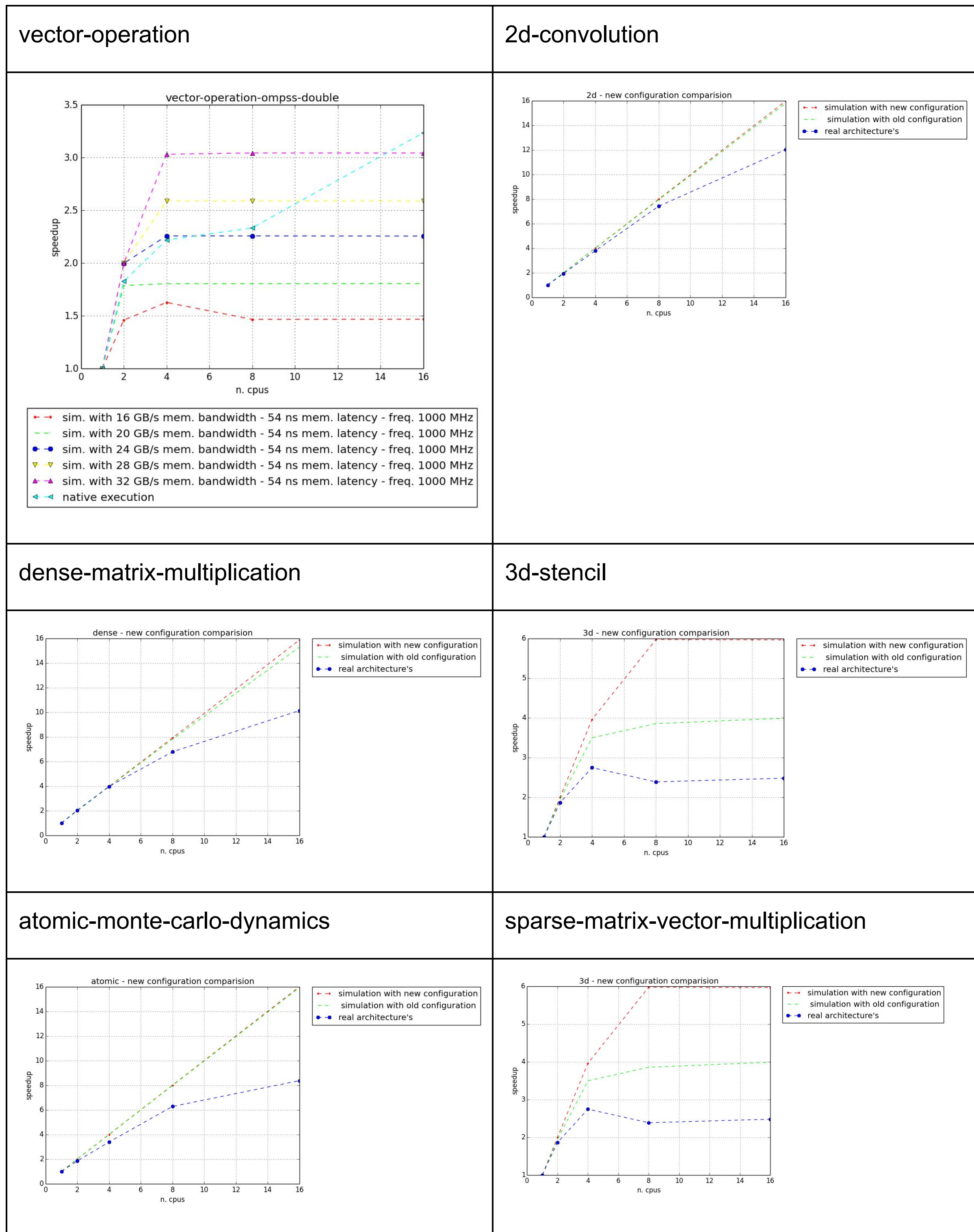
After realizing the simulations, we can observe good feedback in the following chart:

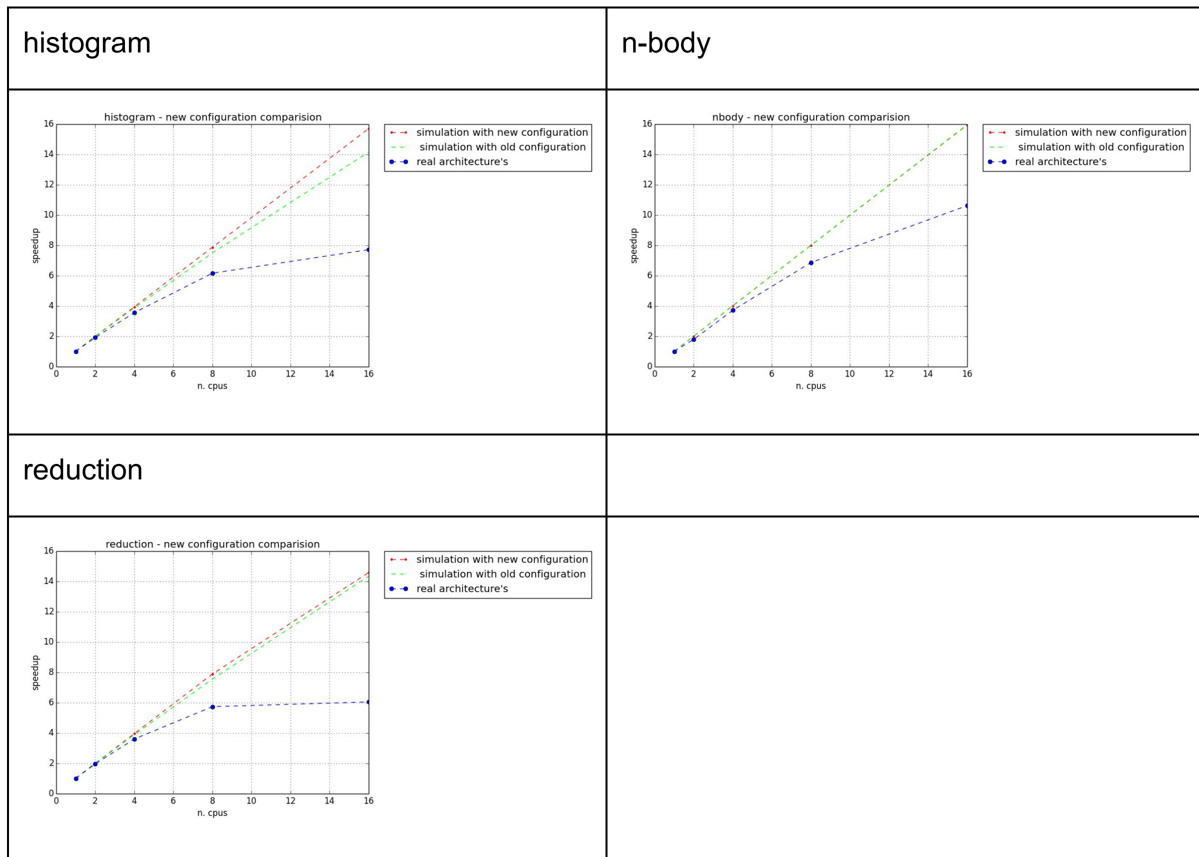


The study that can be found in the last chart, gives a really good feedback. One of the proposed parameter configurations gives a behaviour quite similar to the native execution for the vector-operation benchmark. We can say we just found a good configuration that lets us tune this benchmark for a better outcome.

6.8.2 Setting 1st improvement point

Now, that we found a configuration that really works to simulate the vector-operation, we will use this configuration and test it with all the other benchmarks from the Mont-Blanc benchmarks suite, and evaluate how the modifications affects their previous behaviour:



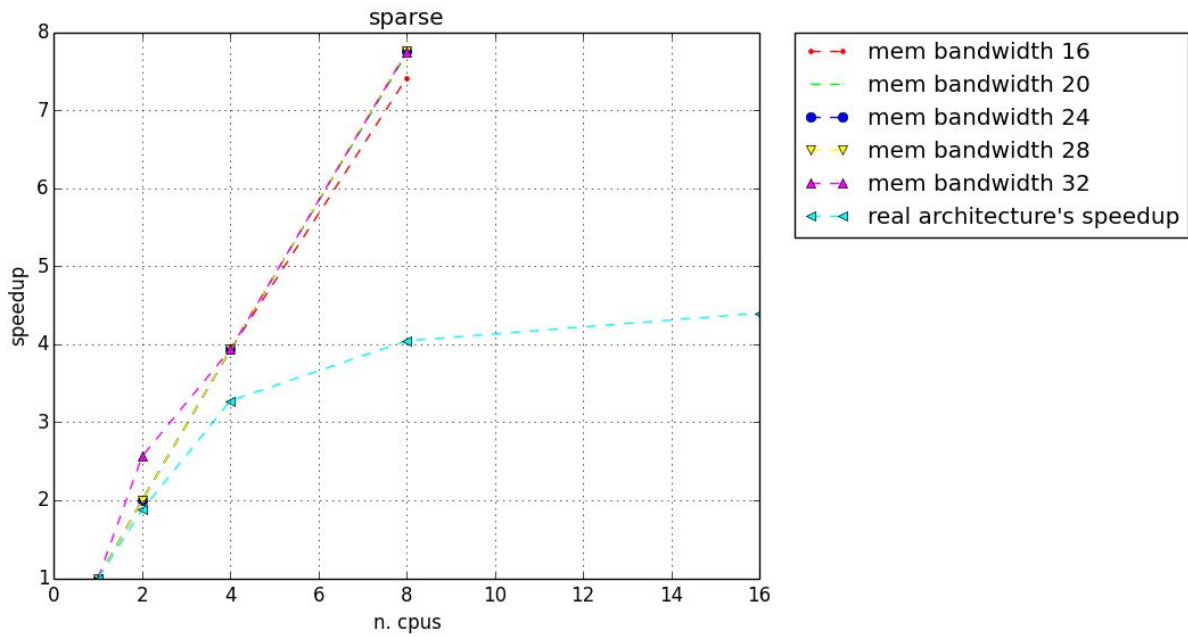


6.8.3 Gain analysis

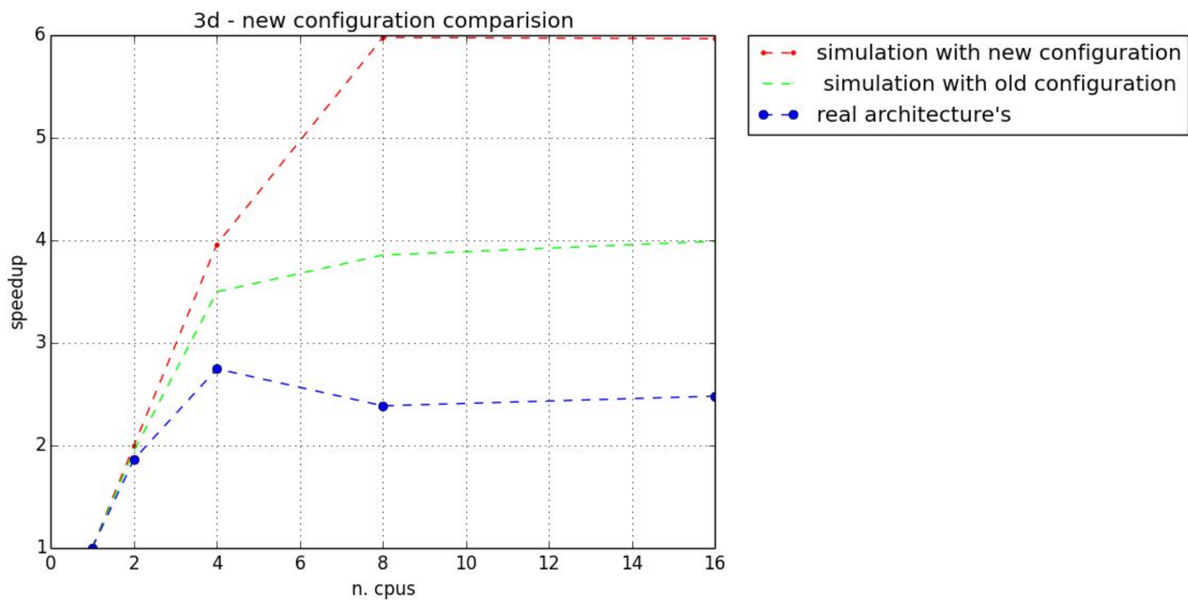
With this new configuration, a few thing can be easily observed. Most of the simulated benchmarks did not show any difference in their behaviour. Only a few did change:

Huge improvement	vector-operation
Same good behaviour	The rest
Same bad behaviour	sparse-matrix-vector-multiplication
Worsen	3d-stencil

As we can see in the following chart, the sparse-matrix-vector-multiplication benchmark did not change, is still not showing the wanted behaviour:



As for last, what surprised us more, was the 3d-stencil benchmark. A benchmark that shows a worse behaviour than before. Having a bigger bandwidth generates an increase in speedup on the simulation, that is not expected in the native execution:



7 Conclusions

This project had one main objective, to take the TaskSim simulation memory model for a driving test. No previous study had been done before for this powerful tool.
A simulator like the TaskSim is a big and complicated application, that is able to simulate a more complex hardware with just a simple description from the user. Adding a mask layer between the hardware and the user. Being able to verify that will have an expected behaviour is a complicated task but really interesting with a lot of meaning.

In order to accomplish this, the main task was to a benchmark suite with a lot of different memory dependent benchmarks, like the Mont-Blanc benchmarks.
This task has been carefully completed. We ended discovering a strange behaviour with the memory latency. With some benchmarks non-memory latency dependents, would end being influenced by the memory latency parameter. This creates the need to study this phenomena in a future project, in order to explain it and corrected it if necessary.

The second objective, was to think and propose new modifications in order to emulate more perfectly the benchmarks simulated. In other words, to give tips that could help tune the behaviour of the simulations with their homolog native execution. This objective was not only accomplish. Not only some propositions were made, but they were implemented, tested and analyzed repeatedly. With that, one of the benchmarks was able to be tuned to emulate quite nearly the same behaviour as the native execution.

For future projects, as commented in this section, a good study of the memory latency dencies found during the realization of this project would generate a more strong validation and help to improve the TaskSim simulator.
Another interesting continuation for this project, would be to keep on searching for the best parameters for every benchmark of the Mont-Blanc benchmarks suite.
And last, it would be really interesting to try other memory bounded benchmarks like the PARSEC benchmarks suite.

As personal gain, I would like to add that this project lead me to get a good methodology in how to approach observed problems in order to understand the behaviour that created them, also how to solve and document them accordingly.

Bibliography

[1] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg,J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A FullSystem Simulation Platform,”IEEE Computer, vol. 35, no. 2, pp. 50–58,2002.
[2] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze,S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, “SESC simulator,”http://sesc.sourceforge.net, 2005.
[3] T. Austin, E. Larson, and D. Ernst, “SimpleScalar: An Infrastructurefor Computer System Modeling,”IEEE Computer, vol. 35, no. 2, pp.59–67, 2002.
[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, andS. K. Reinhardt, “The M5 Simulator: Modeling Networked Systems,”IEEE Micro, vol. 26, no. 4, pp. 52–60, 2006.
[5] E. Argollo, A. Falćon, P. Faraboschi, M. Monchiero, and D. Ortega,“COTSon: infrastructure for full system simulation,”SIGOPS Oper. Syst.Rev., vol. 43, no. 1, pp. 52–61, 200
[6] N. Rajovic, P. Carpenter, I. Gelado, N. Puzovic, A. Ramirez,and M. Valero, “ Supercomputing with Commodity CPUs:Are Mobile SoCs Ready for HPC? ,” inSC, 2013.
[7] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, andA. Ramirez, “ Experiences with mobile processors for energyefficient HPC ,” inDATE, 2013, pp. 464–468.
[8] Rico, A. Cabarcas, F., Villavieja, C., Pavlovic, M., Vega, A., Etsion, Y., Ramirez, A., and Valero, M. 2012. On the simulation of large-scale architectures using multiple application abstraction levels. ACM Trans. Architec. Code Optim. 8, 4, Article 36 (January 2012).