Master of Computer Science at Grenoble
Specialization in *Parallel Distributed Embedded Systems*

On the Impact of Asynchronous I/O on the performance of the Cube re-mapper at High Performance Computing Scale

Riyane SID LAKHDAR

September, 2017

Research project performed at the ***Advanced Institute for Simulation*** within the ***Jülich Supercomputing Center***

Under the supervision of:
Dr. Pavel SAVIANKOU

Defended before a jury composed of:
Dr. Arnaud LEGRAND (President)
Dr. James CROWLEY (Examiner)
Dr. Bruno RAFFIN (External expert)

September 2017

Abstract

The high performance computing (HPC) ecosystem is, by design, obsessed with performance optimization. Developing an HPC-specific application requires the proper performance profiling and analysing tools. The high number of *compute* cores and the complexity of an HPC platform lead these utilities to generate and deal with very large performance-trace files. In this context, we have considered enhancing the I/O-access of the *Cube re-mapper*, a state-of-the-art trace-analysis software for HPC executions. We propose an overlapping-I/O *write* approach to outperform the time-response of the *Cube re-mapper*. Thanks to a theoretical study of the general pattern followed by the *Cube re-mapper*, we show that our method may bring an improvement up to 75% on this pattern. We also show that our custom implementation of the *Cube re-mapper* allows to reduce significantly the perturbation introduced by overlapping the *write* threads. Our most enhanced version is thus shown to improve the time-response of the *Cube re-mapper* by up to 64%.

Résumé

L'environnement des calculateurs a haute performance (HPC) est, par essence, tourné vers l'optimisation des performances. Développer une application destinée á un environnement HPC requiére l'utilisation des outilles adéquats pour profiler et analyser ses performances. Le grand nombre de cores (CPU) et la complexité des plate-formes HPC aménent ces utilitaires á générer et á gérer de trés larges fichiers représentant ces performances. Un accés efficace aux ressources sur disque (ROM) est donc primordiale pour analyser en un temps raisonnable ces traces. Dans ce contexte, nous nous sommes attelés á optimiser les accés au disque effectués par le *Cube re-mapper*, un outil d'analyse de performance dédié aux executions de logiciel sur HPC. Nous proposons ainsi une méthode permettant de superposer les accés en écriture au disque dans le but de réduire le temps de réponse du *Cube re-mapper*. Grace á notre étude théorique de l'architecture global constituant le *Cube re-mapper*, nous démontrons que notre méthode permet un gain allant jusqu'á 75% du temps de réponse de cette architecture. De même, nous démontrons que notre implantation du *Cube re-mapper* permet de réduire significativement les perturbations introduites par toute méthode de superposition des écritures sur disque. Ainsi, nous démontrons que notre version la plus élaborée du *Cube re-mapper* permet un gain de l'ordre de 60% par rapport a la version actuelle.

Acknowledgement

I would like to express my sincere gratitude to Dr. Pavel SAVIANKOU for doing me the honour of accepting me as part of his team within the JSC laboratory. His priceless assistance and his tendency to call into question every statement I made have definitely increased the standards of our research.

I would also like to thank Dr. Christian FELD for his very precious and accurate explanations. His help to understand and patch the *Score-P* software has been a real blessing. Not to mention our very inspiring conversation on the potential evolutions of the software.

Meanwhile, I am very thankful to Ilya ZHUKOV for providing the data samples used for our experimentations. I would also like to express all my friendship to my colleague Margarita LONGSWORTH for her cheering energy and her *joie de vivre*. Not to mention her priceless reviews of my work.

Last but certainly not least, I would like to offer my deepest and most sincere gratitude to Dr. Halim LEHTIHET for his invaluable help and high-standard advises. His unique support, through the years, and his constant push to strive for excellency are the main reasons why I am currently allowed to defend this thesis.

Contents

List of Figures

— 1 —
Introduction

For the end of the fifties, the computer science world has known a real race to increase the computation power. This phenomenon was partly observed through the development of new algorithmic paradigms and through the design of more application-oriented kernels[1]. However, the evolution of the hardware architectures has probably had the most prominent impact.
Indeed, one commonly-cited hardware-processor limitation is linked to the transistor limitation predicted by *Moore*'s law[2]. In order to circumvent it, various approaches have been deployed, based on a distribution or an aggregation of concurrent *compute cores*.

Our work deals with the flagship of these massively parallel supercomputers: the *High Performance Computers* (HPC). The present report describes an experimentally-driven study of the pattern implemented by an HPC-performance analysis tool (namely: the *Cube re-mapper*). In this study, we propose and evaluate an asynchronous I/O approach with the aim of outperforming the *Cube re-mapper*. We also emphasize the perturbations introduced by such an approach; Then, we propose and assess a succession of enhanced variants to overcome these drawbacks.

1.1 Context: the high performance computing ecosystem

The term *HPC* refers to an atomic hardware platform with a high level of computing, storage and communication resources (compared to a general-purpose desktop or server). It gathers a high number of computation units[3] on the same physical platform.

In addition to the important hardware resources, common HPC platforms implement some modern hardware designs in order to tackle the contention of the numerous parallel instructions. For instance, the platform we consider in this work (see section 3.3.3) implements hardware

[1] Such as the *Micro-kernel* and the *Elastic-kernel*

[2] "Moore's law is the observation that the number of transistors in a dense integrated circuit doubles approximately every two years." [1]

[3] Up to roughly 10^7 CPU cores for the number one in the *TOP500* ranking (*Sunway TaihuLight*, 2016)

instruction pipelining[4] and *vector processing*[5].

The computation power offered by the HPC platforms is not a simple theoretical achievement. It fulfils a real need of the scientific community to process very-large-scale data (exascale). Indeed, various scientific and simulation fields have different computational requirements and expectations. However, they often have a common need to processing-scalability. For instance, algorithms for modern weather-forecast or medical-imaging have to handle complex numerical models. The numerous parameters involved and their inter-dependency within these models make such computations require up to 10^{18} floating-point operations (exaFLOP). Meanwhile, these same computations require a massive parallel execution to be run in a reasonable time. The HPC platforms allow to solve such distributed operations in a fully parallel (physically) execution. Moreover, thanks to the CPU-core proximity, this level of parallelization is reached with a reduced intercommunication overhead.

1.2 HPC-application development: performance-analysis driven

The performance analysis tools are a major and founding part of the HPC ecosystem. They are a mandatory material for the development and the deployment process of HPC-specific applications. Indeed, the HPC environment is, by design, obsessed with efficiency and performance improvement. The whole idea beneath its hardware architecture is to merge computation cores[6] in a purely performance-driven vision.

Performance-profiling and performance-analysis are significantly hard when dealing with HPC because of the complex hardware architecture. The large application scale[7] that is considered makes the performance-profiling task even harder. Thus, a regular debugger may no longer be sufficient to detect efficiency bottlenecks. In fact, the amount of potential performance bottlenecks has sky-rocketed with the increase in the complexity of the hardware and software. Consequently, the human intervention for detecting them is no longer achievable without analysis-software and automated processes.

In this context, using the proper tool-set is primordial. To lead the experimental part of our study, we have relied on *Score-P*[10] to profile the execution of each of our custom implementation as well the benchmark one. We expected from *Score-P* to record the behaviour of these implementations regarding some performance-critical parameters (such as the execution time, the I/O resource access, the cache access and misses). Then, we have used the *Cube*[21] utility in order to plot and analyse the previously-generated execution profiles. We will show how the

[4]"Instruction pipelining is a technique that implements a form of parallelism called instruction-level parallelism within a single processor"[2]

[5]"A vector processor is a central processing unit (CPU) that implements an instruction set containing instructions that operate on one-dimensional arrays of data, compared to scalar processors, whose instructions operate on single data items."[5]

[6]It also focuses on optimizing the corresponding communication and interference

[7]massively parallel threads, process or MPI threads

results of these utilities have highlighted some unexpected weaknesses in our preliminary custom implementations. It has also driven us to design and validate the corresponding solutions.

One of the major challenges for the HPC-specific performance analysis tools comes from the enormous amount of data generated and analysed. Tracing a large-scale application running on several thousands of compute cores requires to record, among other things, the execution on each core and the communication between them[8]. Thus, the key facet of our study was to tackle this efficiency issue on a part of the *Cube* tool-set: the *Cube re-mapper*.

1.3 Objective: introducing an asynchronous I/O approach to the *Cube re-mapper*

Our main objective is to optimize the response-time of the *Cube re-mapper*. In order to reach this goal, we first identify the performance limitations of this software. As a part of the HPC performance-profile tools, the *Cube re-mapper* is dealing with a significantly large amount of data on hard disk (corresponding to the execution-trace of some profiled applications). Such an I/O resource access is well-known for introducing a relatively large time overhead (at processor scale) as well as making the processor stall.

Considering the I/O resource-access issue, we propose and evaluate a new approach to deal with the data stored to the hard disk. Our approach to reach this goal is based on an asynchronous I/O method. We propose an algorithm of the *Cube re-mapper* based on an overlapping of the *compute* and the I/O *write* operations. We also try to take advantage of the low-overhead parallel-access to the I/O resource at hardware level in the HPC[9]. The primarily considered design of the corresponding hardware-disk architecture being the multiple access-heads that might be used in parallel.

1.4 Structure and contribution

The *Cube re-mapper* follows a pattern shared by several scientific-computation applications (see listing 2.1). Our attempt to improve the performances of the *Cube re-mapper* might thus be seen as a general-purpose process that could be applied beyond the scope of this software.

The present report is organized as follows:
In chapter 2, we present the state-of-the-art regarding the performance profiling and analysing tools specific to the HPC platforms. We principally focus on the *Cube* and the *Score-P* software. We also introduce the existing synchronous-I/O version of the *Cube re-mapper*, which will be used as a benchmark.
In chapter 3, we describe our solution to enhance the performances of the *Cube re-mapper* and reduce the corresponding processor-stall time. We give the implementation details of our asynchronous I/O strategy and the way it has been shipped to the existing implementation of the

[8]The number of communication channel (at software level) grows exponentially with the number of concurrent compute cores

[9]In our experimentation, we used a *General Parallel File System* (see section 3.3.3)

Cube re-mapper. We also offer theoretical models to predict the response-time of our approach as well as the gain brought by our solution. We then highlight the parameters that influence most this gain.
In chapter 4, we give the experimental results obtained via our approach and its variants. These results are compared to those obtained via the existing implementation of the *Cube re-mapper* which, in our case, plays the role of a benchmark. We identify the deviations of our proposal with respect to its predicted behaviour. We then adapt our proposal accordingly.
We should mention that, in a preamble step, we do not use the full version of our implementation to generate our experimental results. We start with a simplified implementation (referred to as the *simulation test-bed*) which still implements the main functions of the pattern but through lightweight algorithms. The purpose is to get ride of the potential interferences created by the complex behaviour of the *Cube re-mapper*.

Our contribution is, first, a theoretical study of the pattern followed by the *Cube re-mapper*. We have designed a *simulation test-bed* to observe this pattern running using different strategies[10]. Using this custom test-bed, we have located the optimal domain to run an asynchronous I/O solution depending on the hardware architecture (I/O *write* time, *compute* time and number of parallel I/O access heads).
Using our asynchronous I/O approach, we have implemented a custom version of the *Cube re-mapper*. Finally, by observing the deviations with respect to the predicted behaviour, we have proposed and assessed a sequence of improved variants for our proposal. We show that our most enhanced variants of the *Cube re-mapper* outperforms the response time of the existing version by an average of 64%.

[10]We used different algorithms to simulate the computation function and the writing function (synchronous and asynchronous)

— 2 —
State of the art

This study has been conducted as a part of the *Scalasca*[1] [27] project, a tool-set that provides highly scalable performance measurements and analysis primarily for leadership-HPC platforms[2]. Thus, this study aims to develop a tool that eases the identification and the treatment of performance-bottlenecks during computations at HPC scale.

A performance study at HPC scale is generally significantly complex due to the number of parameters involved[3]. Tackling such a difficulty requires the use of the proper tools to measure accurately the performance parameters and identify the performance-critical sections. It also requires a user-friendly way to represent them in order to simplify the human intervention.

In this chapter, we first present the tools that we relied upon during the experimental assessments of our solutions. We identify the dimensions assessed by these tools as well as the relevancy of these dimensions when evaluating our solutions. Then, we introduce the performance-profiling software that our study attempts to outperform; namely: the *Cube re-mapper*. We present a global view of its implementation. We also focus on the main specifications of this implementation which, according to us, represents a limitation to the overall time-response of the *Cube re-mapper*.

2.1 Working framework

2.1.1 Execution profiling: *Score-P*

Profiling an application is the first step toward the identification of its performance bottlenecks. The objective of this phase is to observe the behaviour of the considered application at runtime.

In order to profile our solutions, we have used *Score-P*[10], a performance measurement infrastructure, which is being jointly developed by leading HPC performance tools groups[4].

[1]The project is developed by the *Jülich Supercomputing Centre* (Jülich, Germany) and the *German Research School for Simulation Science* (Aachen, Germany)

[2]Such as the IBM Blue Gene/Q

[3]The considered applications are influenced by the high number of concurrent work-flows (process, kernel or *MPI* threads) as well as their inter-communication and memory accesses

[4]Including but not limited to the "*Jülich Supercomputing Centre*" (Jülich, Germany)", the "*Technische Universitat München*" (München, Germany) and the "*University of Oregon*" (Eugene, USA)

The *Score-P* (Scalable Performance Measurement Infrastructure for Parallel Codes) is a plug-in to the regular compiler (primarily *gcc*, *g++* and *gfortran*). It consists of an instrumentation framework shipped with several runtime libraries. At compile time, *Score-P* allows to inject some measurement routines within the input user code[5]. When running the generated binary file, these injected routines will collect some relevant data regarding the execution of the input code. At the end of this execution, an execution-profile file is generated. It is accessible to the user for a later analysis. Thanks to this architecture and functional designs, *Score-P* offers two major interests for our study.

On one hand, *Score-P* allows an accurate and non-invasive measurement of the runtime performance. This commonly-known observation objective is achieved thanks to a reduced memory and time foot-print of the measurement routines.
Indeed, the generated performance profile (representing the runtime statistics) is simply stored during the target program execution[6]. No analysis routine is executed while running the assessed software. The analyses of these raw data are performed *post-mortem* (for both performance purposes and compatibility with the profile-analysis tools).
At the same time, the information gathered by *Score-P* is reduced to a set of basic dimensions[7]. Such a design optimizes the number of overhead instructions executed at runtime and thus creates a limited interference with the observed application. These data might be transformed later on (post-mortem) to unveil derived dimensions [8].
Finally, the dimensions assessed by *Score-P* are often relative to kernel-level statistics[9]. Thus, their assessment may be processed by the kernel it-self during its management time[10]. Hence the reduction of the contention on the observed application.

On the other hand, *Score-P* enables to measure the execution time of a target application. This time dimension suits the theoretical study we came with on our proposed asynchronous model (see section 4.1) and allows its experimental verification.
It is not common to assess experimentally parallel (multi-threaded or multi-process) applications using their execution time[11]. This dimension is, as a matter of fact, subject to non-negligible random variations due to the OS scheduler behaviour. Time measurement may also hardly be paused when the observed thread is un-scheduled by the kernel. Hence a delayed effective runtime compared to the total execution time.
Thanks to its user-code re-factoring, *Score-P* ensures that the measurement routines only run during the processor-quantums of the observed code. Thus, the time-measurement routine is perfectly in phase with the observed thread.

[5]The C/C++ and Fortran are the main programming languages. Different frameworks ensued from these languages are also handled such as *MPI*[8], *CUDA*[11] and *PGAS*[26]

[6]This storage is performed (till a given threshold) on live memory(RAM) for performance purposes

[7]No single data might be deduced from a set of other ones

[8]Such a data-transformation is performed in order to be presented to the user in a more user-friendly way. The *Cube re-mapper*(see section 2.2) is an example of software that performs it

[9]Such as the I/O-resource access or the process-heap consumption and leak

[10]Different from a system-call which is executed by the kernel while the caller process is waiting

[11]Commonly, the state-of-the-art studies of parallel applications use CPU cycles instead of time

2.1.2 Performance analyses: *Cube*

The statistic file (PPF) generated by the execution-profiling (*Score-P*) tool is mainly intended to allow a *post-mortem* analysis of a program execution. Another advantage of such a file is to allow a translation of its content to several data standards. The purpose being the compatibility with different standards and performance-analysis tools that implement them.
In this context, *Score-P* is compatible with different execution-profiling standards including *TAU*[22], *Scalasca*[27] and *Vampir*[9]. The one that we have considered in this study is the *Cube*[21].

The *Cube*[21] is a tool-set that allows a graphical representation and analysis of a runtime execution profile. It also defines a data model to store these execution profiles[12] as well as a set of translators to map the *Cube* data model with other data models[13].
The *Cube* framework targets especially the large-scale applications running on several thousands of *compute* cores (such as HPC-specific applications). The execution-profile file (generated by *Score-P*) is thus subsequently analysed by *Cube* to identify performance-critical parts of the corresponding program. It thus fulfils the intrinsic necessity of the HPC world to analyse and optimize complicated parallel performance behaviours.

2.2 The *Cube re-mapper*

The performance-analysis framework described in Section 2.1.2 is the basis of our research. Indeed, the main purpose of our study is to optimize the performance of a part of the *Cube* tool-set, namely: the *Cube re-mapper*.

The *Cube re-mapper* is one of the numerous *Cube* tools used to pre-process the raw data contained in an performance-profile file (PPF). The purpose of such a transformation being to map the basic data contained in the the PPF with some inherited dimensions. It might also be used to pre-process an PPF that has been built using other standards.

The existing version of the *Cube re-mapper* is based on a common scientific-computation pattern (see listing 2.1). After parsing the original PPF, the *Cube re-mapper* loops over all the parsed dimensions. For each dimension, a *computation* (*mapping* function) is applied to the corresponding data. Then, the result of the computation (potentially more than one dimension) is written in an output PPF. The output file of the *Cube re-mapper* is intended to be used by a profile-analyse tool (ex: the *Cube*).

[12] Mainly used by the *Scalasca* parallel performance tool
[13] Ex: *TAU*[22] and *Vampir*[9]

```
1  void mainCubeRemapper
2  {
3      File* inputFile = new Cube(inputPPF);
4      Cube* input     = new Cube(inputFile);
5
6      for (int i=0; i<nbMetric;++i)
7      {
8          File* outputFile = openFile("w");
9          compute(input, i, &bufer);
10         write(buffer, outputFile);
11     }
12 }
13
```

Listing 2.1 – *Cube re-mapper*: simplified pattern

The *write* function used by this implementation is a standard (synchronous) one. The considered buffer is forwarded to the kernel in order to be written on the disk (system call). The *write* function waits till the data is effectively transmitted to the I/O resource before it returns. To the best of our knowledge, no specific I/O optimization[14] is being used on common desktop and HPC kernels[15].

Our concern about the *Cube re-mapper* implementation comes from the *write* function. Given the large size of data written[16], the *write* operation may be responsible for a significant time overhead (compared to the computation time).
Likewise, the created processor-stall (due to the I/O access wait time) might lead to a significant time overhead during the memory (live memory) and the cache accesses. Indeed, during the time of the considered *write* operation, the processor is mainly yielded and left accessible to other instructions than that of the *Cube re-mapper*. Hence, the process corresponding to the *Cube re-mapper* is scheduled-out by the kernel. The more this delay increases, the more likely the data corresponding to the *Cube re-mapper* process will be swapped-out from the RAM and from the caches. Hence, a potential overhead[17] during the next access (after the *write* operation).

In Section 3.1, we will introduce our approach to optimize the *Cube re-mapper* writing strategy. This approach is based on an overlapping of the *write* operation with the *compute* one. The potential gain that we expect from this method is linked to the ideal data access-pattern (though time) within the *Cube re-mapper*.
Indeed, one can notice that the data written at a given iteration is never accessed (read nor write) afterwards. Thus, this data could be written at any more convenient moment[18].
Meanwhile, the algorithm of the *Cube re-mapper* shows that each I/O-write is done within an independent file. Thus, a parallel access to the I/O resource (at hardware level) could be done with a minimal overhead, assuming that independent I/O disk-heads are available.

[14] Ex: disk page mapped to the process heap(*mmap* system-call)

[15] See section 3.3.2 for more details on the considered operating system distributions

[16] The considered data files corresponds to the performance statistics collected regarding an applications running on HPC system. It can thus be overwhelming due to the high number of dimensions assessed and the number of compute cores involved

[17] RAM page miss or cache(L2, L2, L3 or TLB) miss

[18] In our case, this *write* will be done in parallel with one of the next *compute* operations

— 3 —
Materials and Methods

The *asynchronous I/O* write is a process that allows to limit processor stall during the *I/O* writing time. It makes use of independent threads in charge of the effective *write* in memory. It also requires the usage of different kernel mechanisms (such as interprocess signalling and synchronization) and low-level interfaces (such as *RAM* controller). A clear vision of the asynchronous I/O implementation is mandatory to avoid a dramatic performances collapse while using it.

This chapter first describes the key concepts of our asynchronous I/O *write* strategy. We present the hardware and software requirements of this implementation and the way we have chosen to embed it within the developed applications.
Second, we give a bird-eye view of our two implementations which are based on this asynchronous I/O *write* strategy. The first implementation, which we call the *simulation test-bed* , is a simplified implementation of the *Cube re-mapper* under different *write* strategies (on demand). The second implementation, which is our main proposal, is a full-fledged version representing our customized implementation of the *Cube re-mapper*.
Furthermore, we present different models for the response time of our proposal. We use these models to highlight the parameters that influence most the gain of our solutions.
Finally, we describe the experimental set-up and the protocol applied for all our experimental assessments.

3.1 Custom POSIX-based asynchronous I/O implementation

3.1.1 The POSIX asynchronous I/O standard

The POSIX asynchronous I/O (AIO) standard[6] is a description of an asynchronous access to the I/O resource. It is provided by all the Linux operating systems since the Linux 1.3 kernel as the standard "aio" (or "pt_aio") library. This UNIX standard library is the angular-stone of all the implementations developed in our work.

Different approaches maybe used to implement an asynchronous access to the I/O resource. The one implemented in the POSIX AIO library is based on a work-flow split (see Figure 3.1): when a thread requires an access to the I/O resource, it simply forwards the request and returns before the request is physically processed. A dedicated thread (*write* thread) is woken up in

order to process the effective I/O access in parallel of the user thread. The delegated *write* thread becomes in charge of triggering the system call[1] and waiting for the operation to be effectively processed by the kernel. Then, it may, on demand, alert the initial user thread about the end of the operation.

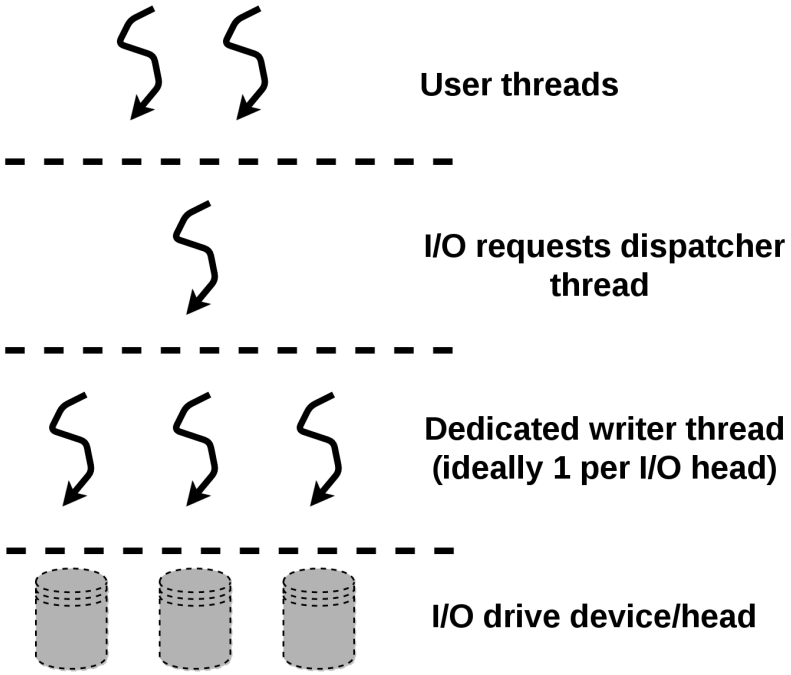


Figure 3.1 – POSIX asynchronous I/O (AIO) standard

Such an approach presents an obvious performance advantage. The caller thread does not need to wait for the end of the asynchronous I/O access. It might execute other tasks that do not require the end of the resource access. Given the relatively long time to access the I/O resource (at processor scale), this might induce a significant reduction of the processor stall time.

Despite these advantages, an intensive usage of the AIO library might be counter-productive in many ways. On one hand, the memory footprint of the system may easily sky-rocket, leading to a harmful intensive kernel-swap process (for both user and AIO library threads). Indeed, each I/O request may create a potentially large buffer[2] which cannot be freed straightforwardly[3]. Given the very small time needed to enqueue an I/O request, compared to the time to execute it, the user might enqueue a significantly large number of these pending requests.

On the other hand, a miss-calibration of the AIO library[4] might lead to a concurrent access to a single I/O device head, resulting in a dramatic increase of the access time to a given I/O block. For instance, let us consider the case where two threads are simultaneously accessing

[1] Kernel interface to access the I/O resource

[2] The considered buffers are intended to store a memory block which size might be comparable to the live memory (RAM) size. Hence the explosion of the live memory footprint

[3] In order to allow the user thread to access it asynchronously

[4] A calibration that does not suit to the hardware specification.

two different blocks[5] on the same I/O device (consisting of a single read/write head). While the head will be scanning a given block, it will be regularly interrupted by requests to scan parts of the other block[6]. According to Patterson *et al*[14], the resulting back-and-forth movement of the I/O head leads to multiply the access time to a block by up to 10-fold.
To avoid this hardware interference, all the implementations we present have been tuned to the optimal number of *write* threads: one *write* thread per I/O device head.

Our choice of the POSIX AIO library as a foundation of our implementations first aims to limit the engineering effort. As a matter of fact, the AIO library manages the whole life cycle of all the *write* threads. It also implements the dispatcher (proxy) thread that receives I/O requests and forwards them to the *write* threads. Likewise, the POSIX AIO library might be tuned (number of I/O devices, number of threads, signal to notify the end of a request process) in order to fit the hardware specifications and the specific I/O access pattern of the application. Moreover, the AIO library fits our performance objectives. Indeed, it efficiently manages, at kernel level, the synchronization of the I/O resource between the *write* threads.
Considering these services proposed by the AIO library, our engineering effort will focus on the synchronization between the user (caller) thread and the I/O device-local (*write*) threads.

3.1.2 Synchronizing the *compute* thread and the *write* threads

The AIO library manages intrinsically the synchronization between the internal threads that it creates[7]. It also implements a basic communication mechanism to notify the user about the end of a request's processing[8]. However, using this mechanism as provided might significantly downgrade the efficiency of the asynchronous strategy within the considered I/O pattern (see pattern definition in Section 3.1.4).
In this section, we first describe the need we have to implement a synchronization between the user and the *write* threads. Then, we give an overview of the synchronization mechanism provided by the AIO library. We explain how it might harm our pattern performances and we describe our solution to enhance it. Finally, we list the principles that we have followed in our implementations[9] to ease and fit this synchronization.

The need of synchronization

The main reason why an efficient synchronization mechanism is vital for asynchronous I/O is to prevent from memory foot-print explosion. Indeed, as introduced in Section 3.1.1, the asynchronous I/O (and the overlapped accesses to I/O in general) might lead to a significantly large number of pending requests and of corresponding buffers. Given the potentially large amount of data carried by these buffers (relative to I/O disk size), this may easily lead to run out of usable live-memory addresses. It might also dramatically reduce the overall performances due to the subsequent OS-swap (RAM swap).

[5] An I/Oblock is potentially spanning different I/O sectors

[6] Indeed, scanning a memory block at hardware level is not atomic. It might have different granularities down to a sector

[7] Dispatcher and *write* threads

[8] Effective access to the I/O resource

[9] Namely the *simulation test-bed* and the *Cube re-mapper*

By synchronizing the *compute* thread (one that produces the buffers) with the *write* threads (those that consume them), the user can expect to access the buffers right after they are processed then remove them. Hence the reduction of the memory footprint. However, the important question that remains is: how to make such a synchronization harmless to the performances of the *compute* thread. Moreover, how to extend this synchronization in order to regulate the amount of simultaneous buffers before they are created[10].

The synchronization mechanism proposed by the AIO library consists in sending a signal (UNIX-kernel signal) to the caller thread after the I/O request has been effectively processed. The user thread may decide to execute a custom routine handler to this signal or ignore it. This mechanism fulfils the synchronization purpose described. However, two main issues might follow on from its usage.

Thread-safety issue of the AIO synchronization

The signal handler may compromise the correctness of a critical section between the caller and the *write* threads. As this signal is received asynchronously, the user (caller thread) cannot decide at which moment to execute the handler. The handler might thus occur while the caller thread is trying to enqueue a new request to the structure shared with the *write* threads. As the handler might also require an access to this structure, the single-access principle of this critical section is violated.

Our solution to avoid thread-safety corruption on the *caller* thread is to forward the execution of the handler to a delegated thread. This prevents the *caller* execution from being interrupted by any handler. It also spares the user from implementing a specific critical section to prevent from the handler's execution at inappropriate moments.

Time overhead on the *compute* thread

Executing the handler corresponding to the asynchronous I/O signal might create a significant time overhead on the caller (*compute*) thread. As a mater of fact, this handler interrupts the execution of the caller thread in order to process its own code. Even more damaging, this handler might require a significant time to process. It might, for instance, require to access its corresponding buffer[11] (in order to remove it or to check the correct execution of the asynchronous I/O request). Given that this buffer has been processed by another thread (*write* thread), its access requires to import it from a potentially distant cache on a potentially distant *NUMA* socket and invalidate it. Considering the size of such a buffer, this remote *NUMA* node importation is very likely to create false-sharing (see Section 4.2.3). Hence, the increase of the response time of both the current thread (*caller*) and the one running on the remote *NUMA* node (*writer*).

Our solution to avoid time overhead on the *caller* thread is the same as that addresses the thread-safety issue. It consists in forwarding the execution of the handler to a delegated thread. This allows the *caller* thread to have no delay due to the handler[12]. It also spares the user from

[10]This regulation must be triggered on demand: when the memory footprint exceeds a given critical threshold, any creation request should be delayed

[11]Buffer which execution has triggered the current handler

[12]Assuming a sufficient number of CPU-cores

implementing a specific critical section to prevent from the handler's execution at inappropriate moments. Ideally, this delegated thread would be the *write* thread that has just processed the corresponding request (for core and cache proximity reasons).

Improving the AIO synchronization

In our implementations, applying any modification to the thread creation and management is not straightforward. Indeed, these threads are created by the POSIX asynchronous I/O library; a UNIX standard library with no certified open-source version available. Thus, it was not possible to change the thread creation part of this library nor to re-assign the workload between threads.
Our solution was then to wrap up the so-called *pthread* library.

Our solution is to make the thread-creation-request (and few other thread-management-requests) called by the standard asynchronous I/O library to be forwarded to our custom wrapper of the *pthread* library. Our wrapper manages the thread creation and affinity with unique CPU-cores[13]. It also allows to attribute custom functionalities to some specific threads.

This solution has led us to build a custom implementation of the *Cube re-mapper* referred to as the *asynchronous I/O-pinned thread* version.

Despite the interest of this custom wrapper library, this solution has not been shipped to our final version of the *Cube re-mapper*. Indeed, wrapping the standard *pthread* library with our custom one requires a non-negligible human intervention before running the *Cube re-mapper*[14]. Such a requirement would make the *Cube re-mapper* less user-friendly. Therefore, unless otherwise specified, our custom wrapper library will not be used in the experiments which will be presented in the next chapter.

3.1.3 Data distribution among threads

Custom shared data structure

The two previous solutions aim to enhance the synchronization between the *compute* thread and the *write* thread. The efficiency of such solutions relies on an adequate thread-proximity of the data. The pattern that we consider is, by design, adapted to such a data distribution (see section 3.1.4). Our engineering effort has thus focused on the data structure used for the synchronization of the asynchronous I/O requests (see Figure 3.2). Our structure implements all the used blocking operations through the *enqueue* and *dequeue* operations. It also enhances thread proximity by using separate ends for the *enqueue* (*compute* thread) and *dequeue* (*write* or *handler* thread).

[13]The specification and the deployment process of the custom thread library have been released in https://github.com/simbadSid/cubeRemapper_perfBenchmark

[14]For instance, the user would need to set the *LD_PRELOAD* environment variable with our custom library. And an unexpected exit of our software would make our custom library be called by default by any other application

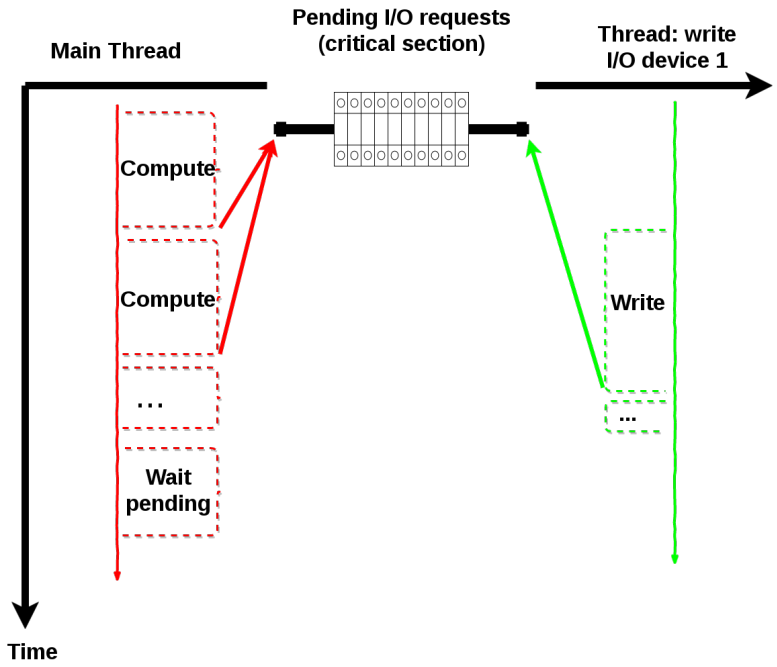


Figure 3.2 – Synchronization scheme of the *compute* thread with the AIO threads

Reducing the impact of *false-sharing*

Among the factors that can limit the efficiency of concurrent algorithms, the shared memory is probably the one that has the deepest impact. In section **??** we have presented our try to reduce the number of shared memory addresses (at RAM level) and lighten the impact of synchronization between threads. Let us now reduce even more this shared data impact at cache level. Our purpose is to address the well-known issue of *false-sharing*.

Let us consider two independent memory buffers used by two concurrent threads. For clarity, let us also consider that each thread is running on an independent core. Although the two buffers share no common address, they might still be stored by the same cache line (see Figure 3.3). As the granularity of a cache is a line, then modifying one buffer will lead to the invalidation of the other one. Likewise, accessing the unmodified buffer will require to import the content of the remote cache line. Thus a significant overhead at each *read* and *write* access. This drawback of multithreaded applications is known as ***false-sharing***.
Knowing that in our implementation all the concurrent threads (*compute* and *write* operations) are processing *write* accesses, we can easily imagine the high frequency of this costly and idle back-and-forth cache routine. An assessment of this cost will presented in next chapter (Section 4.2.3).

Our solution to avoid *false-sharing* between concurrent threads is to align each buffer accessed by the concurrent threads to a cache line. By doing so, we ensure that each buffer is stored within an independent cache line. Thus, each access to this buffer will be done independently from any other thread.

This solution has led us to build a custom implementation of the *Cube re-mapper* referred

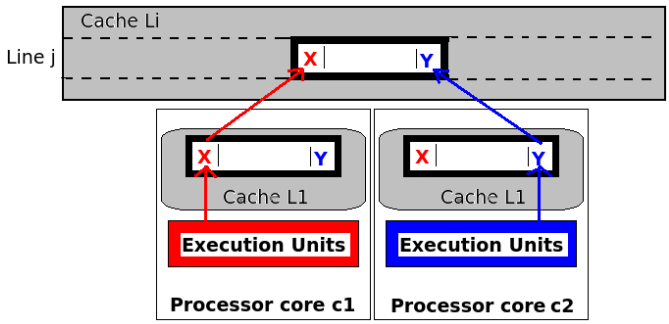


Figure 3.3 – *False-sharing* concept

to as the *asynchronous I/O-buffer aligned* version.

Custom dynamic memory allocation

Let us now see how to go further in limiting the contention on shared data at cache level (primarily L3). Our objective is to smooth the interaction between the *compute* and the *write* threads.
Our main purpose here is to tackle the dynamic-memory-allocation drawback. The objective being to make the dynamic memory allocation (invoked at each *compute* operation) less time-consuming and better fit the specification of the allocation-pattern of our application.
To do so, we describe in this section two solutions to undermine the impact of two main performance bottlenecks (relative to multithreading) of a general-purpose memory allocators[15].
Applying this approach on the *Cube re-mapper* has led us to implement a custom dynamic memory allocator.

We should mention that our memory allocator has, by design, a significant advantage on the general-purpose allocators (in addition to the three key points described bellow). Indeed, our allocator is based on a user-level-managed heap chunk. Thus, the *allocation* and *free* operations require no system calls (unlike the general-purpose allocators). The inherent advantage of this design choice is not discussed neither assessed in this report.

The first design of our custom memory allocator is to take advantage of the specificity of our memory allocation pattern. This is done through managing the free memory as fixed-size blocks (unique size equal to a cache line). Indeed, each allocation needs to be aligned to a given address (see Section 3.1.3). Thus, it is useless to split such a memory block during the allocation: the extra memory (till the next alignment address) will never be required.
Consequently, our dynamic memory management is simplified and reduced to pointer arithmetic. This considerably reduces the management time and memory footprint.
Our custom allocator has no external fragmentation to deal with. Meanwhile, the effect of the created internal fragmentation is lightened by the OS: as the extra addresses are not know by the user, they will most likely never be used. Thus, the OS will swap them out[16], which will

[15] Such as the Linux-standard glibC library: ptmalloc[18]

[16] As a memory blocks has a cache-line size, it will be swapped out into single page. It will also create no residual in the RAM

almost completely remove any time and memory space impact.

The second design we came with is to store our free memory blocks within a structure with two independent ends: one to allocate a block and the other to return a freed block (see Figure 3.4). Indeed, when couple with our asynchronous I/O strategy, this structure ensures that the thread that allocates memory (the compute thread) is not the one that frees it (after the write thread work). Thus, our distributed architecture allows to soften the contention on the shared free memory structure. As the *free* operation may be processed in parallel, we have designed the *free* end of the shared memory structure as a "Treiber" stack [15]: a lock free stack that ensures thread-safety using the hardware atomic primitive: *compare-and-swap*[25]. This ensures a thread-safety to this structure without any software synchronization.

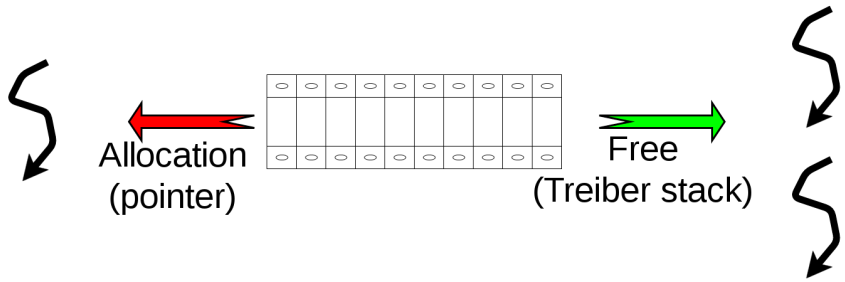


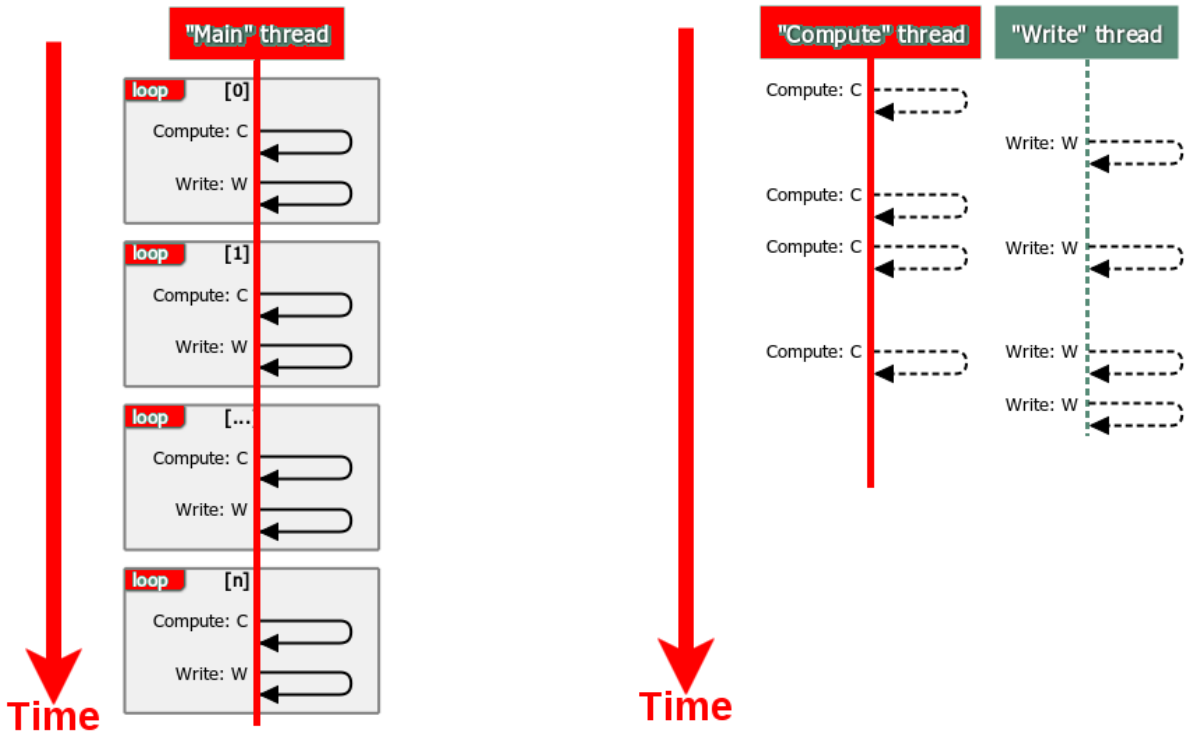
Figure 3.4 – Free dynamic memory architecture

The incorporation of this custom memory allocator has led us to build a custom implementation of the *Cube re-mapper* referred to as the *asynchronous I/O-full-fledged* version.

3.1.4 The *Cube re-mapper* custom implementation (asynchronous)

As described in Section 2.2, the execution time of the *Cube re-mapper* has a significant portion of processor idle time. Indeed, at each iteration of this software, the *compute* operation is followed by a *write* operation. Due to the important amount of data stored on the hard disk, this operation has a significant impact on the overall execution time.
Our objective is to reduce the stall time of the processor by overlapping the *compute* with the cause of the processor stall: the I/O *write* function. The AIO library and the custom synchronization method that we have previously described are the foundations of the improvement we want to bring to the *Cube re-mapper*.

The main idea behind our implementation of the *Cube re-mapper* is summarized in Figure 3.5. Let us consider the main loop of the *Cube re-mapper*. Thanks to the AIO library, we expect to overlap the execution of each *write* operation with one of the next computations.
For such a strategy to succeed, one must ensure that the two operations (*compute* and *write*) are using totally independent memory addresses (except the one used for the synchronization).



(a) Synchronous case

(b) Asynchronous case

Figure 3.5 – Asynchronous I/O applied to the *Cube re-mapper* pattern

3.1.5 The *simulation test-bed*

In order to assess the solution we propose for the *Cube re-mapper*, one could simply ship it to the *Cube re-mapper* and evaluate its performances. However, the *compute* operation of the *Cube re-mapper* involves different and highly-random memory access patterns (at RAM level). It also allocates and deals with significantly large chunks of the process stack. Thus, it might have unexpected impacts on the *write* operation that we implement. It might also have a performance impact on the synchronization between the *compute* thread and the *write* threads. Finally, it might be extremely sensitive to remote memory-accesses on the data it generates[17]. In Section 4.2, we will present an experimental assessment of this potential interferences and we will show the huge impact that they might have on our implementation of the *Cube re-mapper* (basic version).

To avoid dealing with these interferences (in a preamble step), we have decided to simplify the global pattern of the *Cube re-mapper* through a custom and independent program (*simulation test-bed*). This *simulation test-bed* allows to reduce the response time of the system down to the two operations: *compute* and *write*. It also gets ride of the numerous potential sources of perturbation by simplifying these two operations. It is used as a preliminary test for our theoretical models.
At the same time, our simulator may use different strategies to simulate the complex *compute*

[17]The interferences (between concurrent threads) that we consider here take place mainly at cache level. At RAM level, such an interaction is harmless due to the synchronization we have designed

operation of the *Cube re-mapper*. This allows to observe different behaviours of the *Cube re-mapper* without having to find the inputs that would trigger such a behaviour. One may, for instance, observe different *compute* times or live-memory access patterns. From a purely theoretical perspective, our *simulation test-bed* allows to assess different aspects of the *Cube re-mapper* pattern within different domains. This intends to confirm the potential performance gain of our solution. It would also help us identify the domain where such a gain would be optimal.

The main principle behind the *simulation test-bed* is described on the listing 3.1. The algorithm used for the *compute* and the *write* operations are set through the usage of different implementations of the same "Worker" interface.

```
1  int main(int argc, char **argv)
2  {
3      unsigned int nbIteration, computeTime, bufferSize, nbIoDevice, nbProc;
4
5      extractParameter(argc, argv, &nbIteration,
6                       &computeTime, &bufferSize, &nbIoDevice,
7                       &nbProc, &memAccessPattern);
8      // Pick the implementation to use for the "compute"
9      // and the "write" operations
10     Worker worker = Worker.generate(nbIteration, computeTime, bufferSize,
11                                     nbIoDevice, nbProc);
12
13     for (int i=0; i<nbiteration; ++i)
14     {
15         char *buffer = worker.compute();
16         worker.write(buffer);
17     }
18
19     worker.waitPendingRequest();
20
21     return 0;
22 }
23
```

Listing 3.1 – *Cube re-mapper simulation test-bed*

This code also shows the main parameters of the simulation that might be tuned:

- Number of iterations
- Compute time (per iteration)
- Number of hardware I/O devices (used to determine the number of concurrent *write* threads)
- Number of concurrent CPU-cores (used to initialize the AIO library internal thread synchronization)
- Buffer size used at each computation

3.2 Proposed theoretical models

In this section, we propose two theoretical models for the response-time of our custom (asynchronous I/O) implementations. We also try to enhance these models in order to fit the HPC platform specifications (multiple parallel I/O devices and reduced computation time).
These models are used to determine the parameters[18] that influence most our solution. They will also help us (in Section 4.1.1) to confirm the potential gain brought by our solution (from a theoretical perspective). Finally, they will help us determine the domain where our solution may bring a significant improvement.

Clearly, the gain brought by our proposed algorithm will be compared to the existing synchronous model given by the following Equation (3.1):

$$T_{synchronous} = n * (C + W) \tag{3.1}$$

In this model, all the instructions are serialized. Thus, the total execution time is simply the sum of all the execution times.

To conduct our experimentations, we assume that the size of the buffer written at each iteration is constant (for a given experimentation). We also assume that the computation time C at each iteration is constant (for a given experimentation).
The following notations are used to express the theoretical model of our asynchronous version of the *Cube re-mapper*:

- C: computation time at each iteration
- W: size of the buffer written at each iteration
- n: number of iterations
- P: number of computation-cores available
- N_{io}: number of independent I/O access heads (only used in section 3.2.3)

3.2.1 First approach: simple write/compute representation (single I/O device)

In this first approach, we present a simple model by introducing the following hypotheses:

- Constant writing time W of each buffer.
- Perfect parallelization model (execution time with p computation cores $T(p) = \frac{T(1)}{p}$).

Thus, the execution time of the asynchronous pattern would be:

$$T_{asynchronous} = C + W + (n-1) * max(C, W) \tag{3.2}$$

[18] Example: *compute* time, *write* buffer size

Indeed, the first computation time and the last writing time (the two first terms of the above equation) cannot be avoided or softened. Then, the *compute* and the *write* operations are executed by two independent threads (ideally with no interference between them). Hence, we only retain the maximum execution time of the two threads: $max((n-1)*C,(n-1)*W) = (n-1)*max(C,W)$.

In this model and for the seek of simplicity, we deliberately get ride of the scheduling and pseudo-parallelization[19] overhead. This decision is also motivated by the limited impact of these phenomena regarding the range of the considered writing and computing times (see the *write* time evaluation in Section 4.1.1).

By analysing the asymptotic behaviour of Equation (3.2), one may use the following approximations:

$$T_{asynchronous} \overset{\text{if C « W}}{\approx} n*W+C \approx n*W \tag{3.3}$$

$$T_{asynchronous} \overset{\text{if C » W}}{\approx} n*C+W \approx n*C \tag{3.4}$$

3.2.2 Second approach: modelling the perturbations (single I/O device)

The previously-generated model allows a simple estimation of our version of the *Cube remapper* pattern. However, it relies on hypotheses that make it hardly scalable. While trying to enhance this model, we have experimentally observed (see Section 4.1.1) that the difference between the theoretical model and its experimental evaluations seems constant (with respect to the computation time). Based on this finding, we propose the following improved second model:

$$T_{asynchronous} = C + W_{perturbation} + (n-1)*max(C,W_{perturbation}) + n*req \tag{3.5}$$

Where:

- *req* is the time to transmit the *write* request (or enqueue it for later execution).
- $W_{perturbation}$ is the delay of the *write* operation introduced by the payload and the contention on the I/O device.

In this second model, we try to express the previously-observed perturbations[20] by introducing the two expressions *req* and $W_{perturbation}$. These two expressions do not depend on the computation time; hence, each one of them can potentially explain the constancy of the observed perturbation.

[19] Pseudo-parallelization happens when 2 concurrent threads are executed on the same mono-threaded core

[20] Gap between the theoretical model and the experimental evaluation

Modelling the writing time ($W_{perturbation}$)

When we estimate the overall time response of the *Cube re-mapper* pattern, the experimental errors of the *write* time at each iteration are summed up. This may eventually have a significant impact on the total response time. In this section, we no longer consider the *write* time as constant and try to fit more accurately its variations.

Providing an accurate measurement of the *write* time maybe very complex (even when we consider buffers with constant sizes). The *write* time may highly vary depending on the processor family[21], the previously processed I/O requests, the detected I/O access pattern[17], or even some unexpected external parameters (such as the system temperature or some magnetic interferences)[12]. However, we still need to estimate it to validate our theoretical model.

Our solution to overcome this difficulty is to consider the writing time as constant but only at saturation regime. This solution is inspired from the "I/O saturation method" introduced by R. Robert *et al*[20]. It consists in measuring the time to write a given buffer after flooding the I/O system with concurrent *write* access (following a given access pattern). In order to measure the required I/O-saturation level[22] and apply the I/O-saturation during the write-time measurement, we have used the code released by the above-cited reference.

Modelling the asynchronous I/O request time (req)

Let us consider an asynchronous I/O request submitted by the *compute* thread. Such requests are always processed by the main thread in the considered approach. Thus, their time foot-print cannot be avoided neither divided upon processor cores or I/O devices.
Within the scope of our study, we consider the asynchronous I/O request-time as constant (for a given hardware platform). Indeed, submitting an asynchronous I/O corresponds to simply forward the request-address from the *compute* thread to the *write* thread. No other computation nor data transfer is performed at this step[23].

Our method to evaluate the asynchronous I/O request-time[24] is similar to the I/O write-time evaluation (see Section 3.2.2). It consists in assessing the time T to submit an asynchronous I/O request for different number n of iterations (number of requests submitted): $T = req * n$. Then, determining req is equivalent to determining the linear model that fits the best the assessed values (T, req).

Thanks to the generated linear-model, we can determine the value of the request time as the slope of the linear model. We may also confirm our hypothesis (constant asynchronous I/O request-time) using the constant term[25].

[21] Here we mainly refer to the number of cores and physical threads.

[22] Depends on the hardware I/O specification

[23] The buffer to be written might be imported by the writer thread later-on when the effective I/O operation will be processed

[24] For a given hardware platforms

[25] Other methods could be applied in order to assess the validity of the linear model (example: the "coefficient of determination" R^2). But these would require a benchmark evaluation of the acceptable error

3.2.3 Introducing multiple parallel I/O devices

In this section, we suppose that the hardware platform has at its disposal N_{io} independent I/O devices. Hence, at most, N_{io} independent I/O operations maybe performed concurrently. For simplicity, we assume in this section that the parallel I/O model is perfect: two concurrent I/O operations on two different I/O devices will not create any interference on each others. We also assume that the number N_{io} of I/O devices is much smaller than the number n of iterations (for the simplicity of the model expression).

Thanks to Equation (3.4), we can notice that when $C >> W$, the time response of our system does not depend on the writing time W[26]. Thus, introducing additional parallel I/O devices will not affect this response time (see Figure 3.6a). Hence, our model in this case remains:

$$T_{asynchronous}(N_{io}) \overset{\text{if C » W}}{\approx} n*C+W \approx n*C \tag{3.6}$$

By simply extending this model to the case where $C \approx W$, we can see (Figure 3.6b) that additional I/O devices are still useless. The model in this case remains:

$$T_{asynchronous}(N_{io}) \overset{\text{if C = W}}{\approx} n*C+W \approx n*C \tag{3.7}$$

Finally, in the case where $C << W$, the number of I/O devices becomes relevant (see Figure 3.6c). The new inflection point (computation time where this model becomes applicable) is $C = \frac{W}{i}$. The theoretical model in this case becomes:

$$\begin{aligned}
T_{asynchronous}(N_{io}) &\overset{\text{if C « W}}{\approx} (n-1)*max(\frac{W}{N_{io}},C)+C+W \\
T_{asynchronous}(N_{io}) &\overset{\text{if C « } \frac{W}{N_{io}}}{\approx} (n-1)*\frac{W}{N_{io}}+C+W \\
T_{asynchronous}(N_{io}) &\overset{\text{if C } \in [\frac{W}{N_{io}},W]}{\approx} (n-1)*C+C+W \approx n*C+W
\end{aligned} \tag{3.8}$$

3.3 Target hardware platform and compatibility

3.3.1 Hardware platform

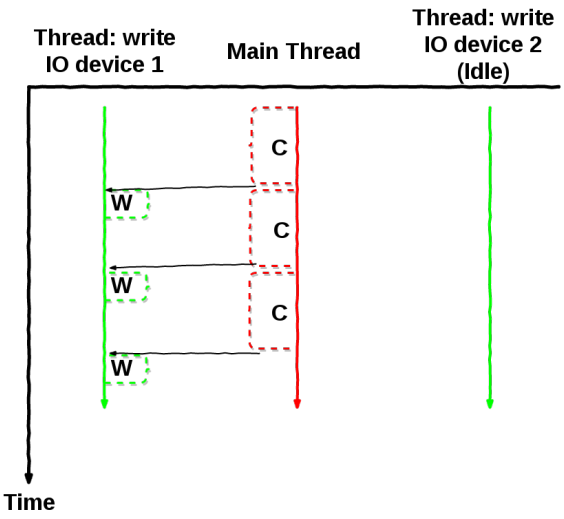
The performance profiling and measurement tools that we are developing are intrinsically designed for highly-multithreaded (physical thread) hardware platforms. There principle purpose is to track and profile performance-bottlenecks that are due to the contention of concurrent (threads or processes) execution. These tools aim to assess each unitary[27] execution. They also aim to evaluate the communication and the interaction between these executions[28]. These dimensions can only be set and observed on a hardware platform that physically[29] allows an execution at such concurrent scale.

[26]However W appears in this equation, it is a simple added element with no coefficient. Thus this write time W can not be split over multiple devices.

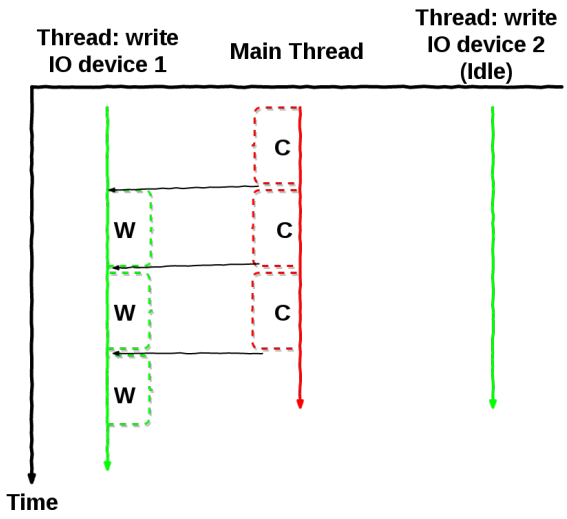
[27]Thread, process or job execution

[28]*MPI* or kernel-level (ex: pipe, socket) communication

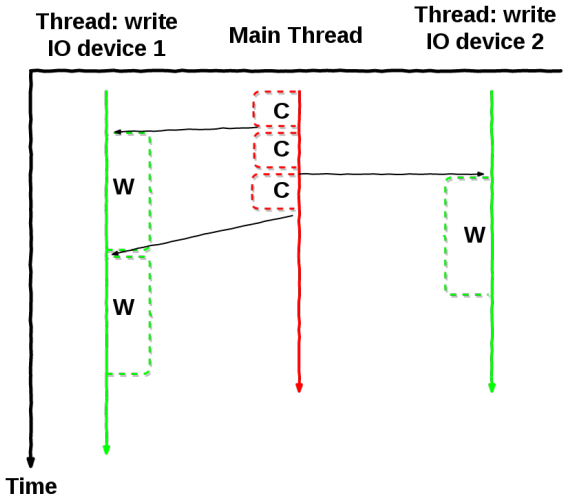
[29]In this study we do not consider virtual hardware environments.



(a) Computation time >> Write time



(b) Computation time ≈ Write time



(c) Computation time << Write time

Figure 3.6 – Asynchronous I/O pattern diagram with multiple I/O devices

As we consider highly-concurrent hardware platforms, HPC platforms are obvious candidates. For the experimental assessment of our study, we made such a choice (see section 3.3.3) for both practical and functional intends.

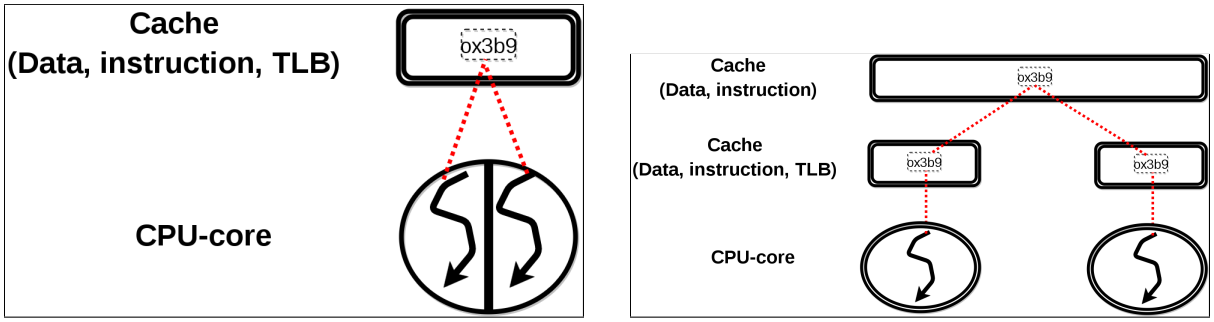
On one hand, our study has been conducted within an environment of cutting-edge HPC-specific performance track researches. Our goal to outperform the *Cube re-mapper*[21] is part of a global project[30] that aims to develop a set of tools that provide highly scalable performance measurement and analysis for computation at HPC scale. Our work is thus part of a framework to enhance and automate the profiling of parallel and distributed applications at HPC scale.

On the other hand, an HPC platform represents an ideal experimental set-up for our theo-

[30]Scalasca project[27]

retical study about overlapped I/O-accesses. The HPC platforms we considered (see Section 3.3.3) has at its disposal several parallel and independent I/O devices, managed by a shared file system. Moreover, as mentioned in Section 3.1.1, the number of concurrent *write* threads is limited by the number of independent I/O devices. Hence, thanks to the HPC's numerous independent I/O devices, one could expect that the asynchronous I/Owould bring a significant additional gain.

Thanks to the specification of the considered HPC hardware (see Section 3.3.3), the overlapping and the parallelization implemented by our solution maybe reached with a reduced time overhead. In this context, the high number of CPU-cores and the large cache size are two commonly-considered helping factors. However, the *Hyper-Threading* technology[13][31] is probably the most influencing design of such a hardware architecture on our approach.
The *Hyper-Threading* technology allows two threads to run concurrently on the same CPU-core: there relative micro-instructions (except the load/store instructions) being processed in parallel as if they where running on two distinct CPU-cores. As shown in Figure 3.7, the advantage of such a design, compared to physical parallelization[32], comes from the optimal cache proximity of the shared data at all cache levels (L1, L2, L3 (data and instruction) and *Translation Lookaside Buffer*).
In the ideal case, this would allow the *compute* thread (producing the buffers) and the *write* thread (consuming the same buffers) to run simultaneously on the same CPU-core. By doing so, we take advantage of the parallelization[33]. Additionally, the shared buffer will not need to be moved from the producer core to the consumer core. Hence the optimal cache proximity.
The same cache proximity advantage can be noticed on the *Translation Lookaside Buffer* (TLB). The address translation[34] used by the *compute* thread is similar to the one used by the *write* thread (in the specific execution case described earlier). Thus, the content of the TLB might be reused by the *write* thread without cache-misses.



(a) Hyper-Threaded core

(b) Multi-core

Figure 3.7 – *Intel*'s Hyper-Threading technology concept

It is worthwhile to mention that the thread-distribution advantage brought by the *Hyper-*

[31]The *Hyper-Threading* technology is not specific to the HPC platforms. It is an *Intel* technology that is also deployed on some modern Intel servers

[32]Two concurrent threads running on two independent CPU-cores

[33]Unlike pseudo parallelization where two concurrent threads run on the *same CPU-core* by interrupting each others

[34]Virtual-physical and physical-virtual address translation

Threading to our overlapping solution is highly random. Moreover, it cannot be controlled nor enforced at software (user or kernel) level. The evaluation of such an impact on our solution is out of the scope of this study.

3.3.2 Operating-system portability

The main family of operating-systems (OS) targeted by our research and implementations is the *UNIX*-like system[4] (with a primer for *Gnu-Linux* OS). The reasons for this choice are relative to both implementation-compatibility and kernel-level requirements.

On one hand, our global project (*Scalasca*) aims to develop an HPC utility (*Cube re-mapper*). As a matter of fact, the HPC ecosystem is well known for its dependency on UNIX-like systems. To the best of our knowledge, only few alternative systems[35] exist for the UNIX-dominated HPC market. No future evolutions seem likely to affect this trend. Consequently, deploying our version of the *Cube re-mapper* on non *UNIX*-like systems would have no real practical interest. Such a release would target an insignificant set of potential users.

On the other hand, the whole idea beneath our asynchronous approach is based on mechanisms that are generally specific to *UNIX* architectures. For instance, let us consider the synchronization that we have implemented between the *compute* thread and the *write* threads (see Section 3.1.2). Our intend was to reduce the memory footprint and ease the communication between the producer and consumers of the I/O requests. To do so, we have made use of inter-process signals as well as condition pipes: event-driven communication. Such a programming paradigm is intrinsically linked to the process-design of *UNIX* platforms; hence the difficulty to consider this paradigm for non-*UNIX* deployment.
Some state-of-the-art libraries allow to emulate *UNIX*-specific mechanisms on non *UNIX* platforms [19]. However, such solutions are not natively implemented by these systems[36]. Thus, their usage may lead to a significant time and memory overhead due to extra system-calls and interprocess polling; not to mention the obvious library- and name-space-compatibility issues.

Despite this compatibility limitation, we still have considered *windows*-OS portability for our implementation of the *Cube re-mapper*.
The main issue when we consider deploying a multithreaded *UNIX*-based application on *windows* comes from the thread-synchronization: unlike most *UNIX*-based OS architectures, the *windows* OS does not natively implement signals and related event-oriented mechanisms. Thus, the whole work-flow of our multithreaded algorithm can no longer be implemented.
Our solution to deploy the *Cube re-mapper* on *windows* platforms was to use the *Cilk Plus*[19] library [37]. Then, through a custom wrapper of this library, we have been able to fit all the kernel-level requirements of our algorithms and benchmarks. Using this wrapper, we have also limited the engineering effort by adapting this interface to the standards followed by the *Cube re-mapper* (name spaces, function names and parameters).

[35]The main example is the *Microsoft Windows* HPC Pack 2008. But even this project has been dropped by *Microsoft* and is no longer supported

[36]Unlike the signal and all event-driven mechanisms on *UNIX*-compatible OS (implemented and executed at kernel level)

[37]Library proposed by *Intel* to emulate *UNIX* process and relative thread-synchronization mechanisms on *windows* platforms at user level (user mode)

This kind of user-level emulation of a kernel process is well known for its potential performance downgrade[24]. Indeed, the *Cilk Plus* library manages the process address-space (and principally the heap's dynamic memory) through costly system calls to the host kernel (instead of being natively executed by the kernel). Likewise, the implementation of these signals and these synchronizations is based on *active-waiting* and *polling*.
The assessment of such an overhead on our asynchronous approach (deployed on *windows* OS) is beyond the scope of this work.

For simplicity and clearness, all the considered implementations in the rest of this work will be implicitly developed, implemented and assessed on UNIX platforms.

3.3.3 Experimental setup

The experimental evaluations that will be presented in the next chapter have been obtained on two x86 machines.
The first one is a *JURECA T-Platforms V-Class* (HPC) consisting of two 24-core (*Intel Xeon* CPU E5-2680 v3 (2.5 GHz) *Haswell*) chips. Each core has a double physical thread support (*Hyper-Threading*). The machine accesses its I/O resource using *JUST*[3]: a *General-Parallel-File-System* storage cluster. Thanks to the hardware distribution of its underlying disks, this file system may perform (*physically*) several simultaneous accesses to the I/O resource. The access to this file system is performed through a 100 GiB/s connection. Finally, on this machine, a *Gnu-Linux* (3.10.0-514.26.2.el7) operating system has been used based on the *CentOS* kernel (7.3.1611). The default kernel dynamic memory allocation policy (*first touch*) has been set.
For all the presented experimentations on the *JURECA HPC* nor intermediate virtualization, or job-encapsulation[38] has been used.

The second considered machine is an *ASUS M32CD-US014T* consisting of four 2-core (*Intel Core* CPU i7-6700 (3.4 GHz)) chips. Each core has a double physical thread-support (*Hyper-Threading*). All the I/O accesses on this machine have been performed on a single disk (*Samsung SSD 850-EVO, 2.5" ATA*) of 256 GiB. To the best of our knowledge, no specific hardware or software optimization has been used regarding this disk. Finally, on this machine, a *Gnu-Linux* (4.4.74-18.20) operating system has been used based on the *openSUSE* kernel (*Leap* 42.2). The default kernel dynamic memory allocation policy (first touch) has been set.

All the programs that implement our algorithms of the *Cube re-mapper* and the *simulation test-bed* have been implemented in C++ language (using the ISO C++11 standard). They have been compiled using *g++* (*GCC* 5.4.0). The option -O3 (maximum optimization level) has been used for all the compilation processes. Additionally, the compiler has been coupled with the *Score-P* (4.0-orphaned-pthreads) to inject the performance-profiling routines to the assessed applications (see Section 2.1.1). A custom patch of *Score-P* has been implemented in order to profile asynchronous signal-handler.

Finally, the software we are trying to outperform is the *Cube re-mapper* (part of the *Cube* (4.4)). This version of the *Cube re-mapper* has been used as a performance-benchmark to test our custom implementations. It has also been used as a basis to our custom implementation of

[38]Using jobs (through a batch scheduler) is a standard way to access an HPC or a cluster platform

the *Cube re-mapper*.

Although the *JURECA HPC* has been used to evaluate our approach, we have still confirmed our observations on a regular workstation (*ASUS Intel Core* CPU i7-6700). We have dedicated our effort to separate as much as possible the performance gain brought by our design from the hardware performance of this HPC. Consequently, the observed experimental gain, which will be highlighted in the next chapter, is not simply due to the performance of the storage cluster on the considered HPC.

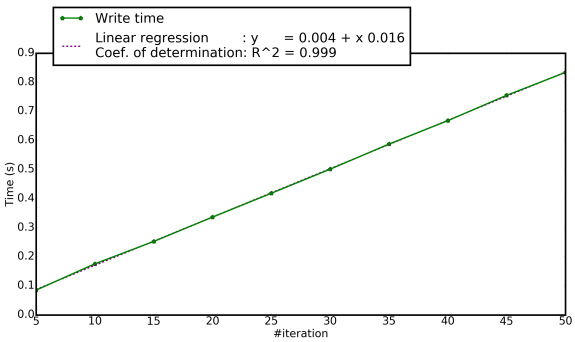
— 4 —
Results and Discussion

In this chapter, we use our custom simulation test-bed to assess the models we presented in both synchronous and asynchronous cases. In Section 4.1, we evaluate the accuracy of these models. We also give an experimental evaluation of the gain brought by our solutions on the considered pattern.
In Section 4.2, we compare the performances of our asynchronous I/O solution using both our *simulation test-bed* and our custom implementation of the *Cube re-mapper*. We show how our solution might interfere with a real-life application (namely: the *Cube re-mapper*software). We show how we identified the causes of these different perturbations. We also evaluate the sequence of custom solutions that lighten such perturbations.

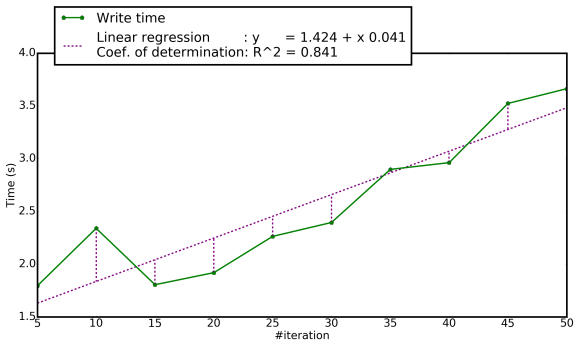
4.1 Asynchronous model and improvements

4.1.1 First model assessment (single I/O device)

In order to assess the first theoretical model (see Section 3.2.1), one should first be able to assess the *write* time W of each buffer. According to the model hypotheses, this *write* time is considered as constant (for a fixed data size and a given hardware platform). Figure 4.1 shows the experimental evaluation that we have obtained on the two considered platforms.



(a) *Intel Core* CPU i7-6700 @ 3.4 GHz

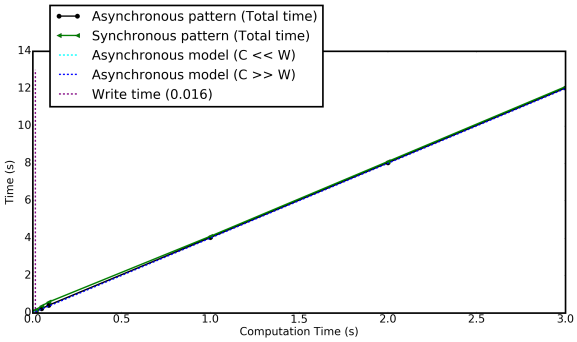


(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

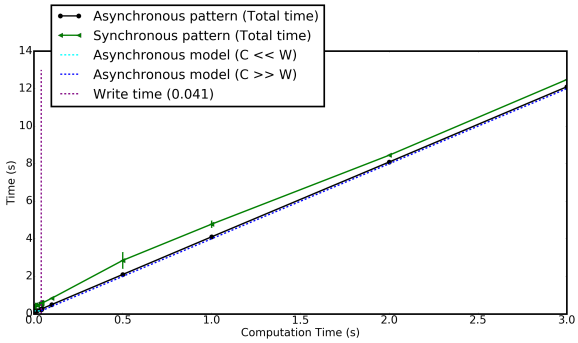
Figure 4.1 – Experimental assessment of the time for writing 50,000,000 bytes

One could notice on these experimental results that the assessed *write* time is roughly constant through different assessments (excluding the outliers which are regularly observed at the beginning of each experimentation on the HPC platform[1]). Indeed, the value of the *write* time W fluctuates within a range of 10^{-2} seconds (for both considered hardware platforms). As the measured computation times (C) fluctuates within a higher range of 10^{-1} seconds (see Section 4.2), then the variation of W could be considered as comparatively negligible. The retained *write* time approximation is the slope of the linear regression on Figure 4.1. This value (for writing $50,000,000$ bytes) is $W = 0.016$ seconds for the *Intel Core* CPU i7-6700 (Figure 4.1a) and $W = 0.041$ seconds for the HPC *Intel Xeon* CPU E5-2680 v3 (Figure 4.1b).

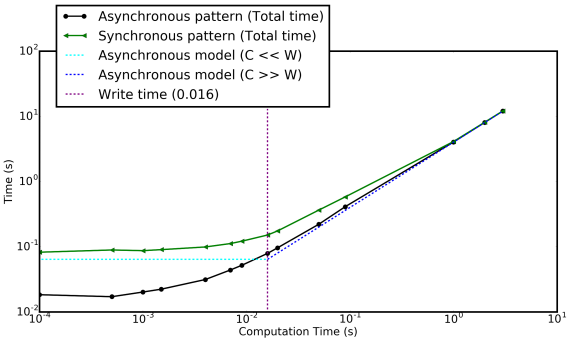
Using the simulation test-bed described in Section 3.1.5, we compared the asynchronous implementation with the synchronous one and the theoretical model (see Figure 4.2).



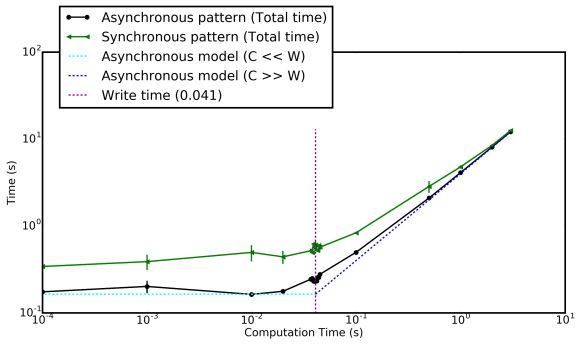
(a) *Intel Core* CPU i7-6700 @ 3.4 GHz (Linear scale)



(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz (Linear scale)



(c) *Intel Core* CPU i7-6700 @ 3.4 GHz (Log-Log scale)



(d) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz (Log-Log scale)

Figure 4.2 – Experimental comparison of the asynchronous solution with its theoretical model and the synchronous one (50,000,000 bytes per write, 4 iterations)

First of all, it is clear on Figure 4.2 that, on both targeted platforms, our custom asynchronous I/O *write* approach might bring a significant improvement compared to the original synchronous one. When the computation time is in the neighbourhood of the *write* time, an

[1]This outlier corresponds probably to a warm-up phase before the pre-loader of the I/O device becomes fully efficient (for the currently accessed I/O pattern)

improvement up to 45% is observed on the *Intel Core* CPU i7-6700 and 38% on the HPC *Intel Xeon* CPU E5-2680 v3). Hence, our global approach to optimize the *Cube re-mapper* seems comforted at this stage.
Second, the trends modelled by Equations (3.1) and (3.2) are clearly highlighted and matched by the experimental response times of the considered pattern on Figure 4.2. We can notice the accuracy in the position of the inflection point on both targeted platforms.

Finally, Figure 4.2 allows to identify the region[2] where the improvement brought by our asynchronous I/O *write* algorithm is the most significant. Indeed, we can notice on both platforms that this gain is maximal when the *compute* time is in the neighbourhood of the *write* time.

Such a property of our asynchronous I/O approach is complicated to use in a real-life software (such as the *Cube re-mapper*). As a matter of fact, it is not easy to adapt the *compute* function and make its execution time match a given value. Furthermore, this targeted value of the *compute* time is totally dependent on the host hardware platform.
However, this property might be used to analyse the performance of the considered software (once shipped with our solution). A *compute* time that is too far from the *write* time could lead to a poor performance-gain of our solution.

4.1.2 Second model assessment (single I/O device)

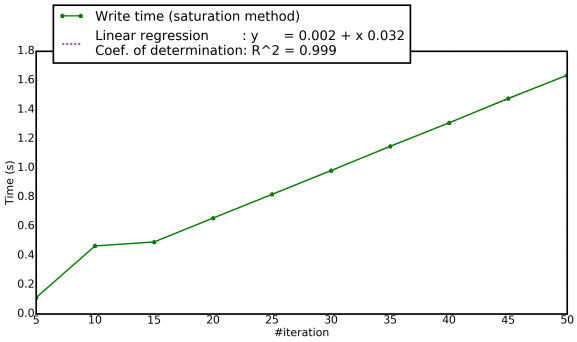
We notice on Figure 4.2 that, in one case (*Intel Core* CPU i7-6700), the predictions of our first asynchronous model (Equation (3.2)) are well-confirmed by the experimental results (Figures 4.2b and 4.2d). This is however not always the case as Figures 4.2a and 4.2c exhibit a significant gap between the theoretical model and the experiment for the *Intel Core* CPU i7-6700 hardware. One could notice that this deviation seems roughly constant[3] (for $C << W$ and $C >> W$). Therefore, the enhanced second model proposed by Equation (3.5) maybe more accurate. In this section, we experimentally test this second model by assessing its two founding parameters: the *write* time ($W_{perturbation}$) and the asynchronous I/O *request* time (req).

Modelling the *write* time ($W_{perturbation}$)

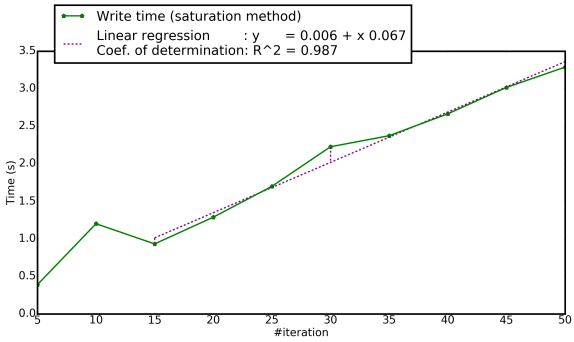
The experimental approximation of the *write* time previously presented (Section 4.1.1) has a relatively low fitting-coefficient[4] equal to 0.84 on the HPC *Intel Xeon* CPU E5-2680 v3 platform (see Figure 4.1b). Consequently, we have considered using the *saturation method* (see Section 3.2.2) in order to attempt to model the *write* time more realistically.

In Figure 4.3, we have represented our measurement of the *write* time ($50,000,000$ bytes) using the *saturation method*. One can notice that, after a warm-up phase[5], the *write* time reaches a given threshold. Then, its fluctuations almost vanish.

[2] Set of computation times
[3] Does not depend on the computation time
[4] *Regression coefficient of determination R^2 of the linear model*
[5] Corresponding to the time to flood the file system with I/O requests



(a) *Intel Core* CPU i7-6700 @ 3.4 GHz



(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

Figure 4.3 – Experimental assessment of the time for writing 50,000,000 bytes using the *I/O saturation method*

Thanks to the *saturation method*, the fitting-coefficient of the *write* time estimation has been improved to 0.987 (see Figure 4.3b) on the HPC *Intel Xeon* CPU E5-2680 v3 platform. Furthermore, the constant term in the linear approximation of the *write* time is now closer to zero (0.006 seconds) compared to the previous value of 1.42 seconds (see Figure 4.1b). Therefore, the following linear function:

$$f(iteration) = W_{perturbation} * iteration$$

is more realistic.
Note that the currently-presented experimentation has been obtained by excluding, in the linear regression, the samples which are bellow the worm-up threshold (two first points). This is done in order to remove any bias which would be introduced during the warm-up phase to our linear regression.

Thanks to this *saturation method*, we have a more stable and more realistic evaluation of the *write* time. However, it is clear that this measured value is artificially increased; hence the importance to show its constancy[6]. Indeed, a constant overhead in the *write* time (for all the experimentations) will not affect a benchmark study as long as it is incorporated in *all* the compared algorithms.

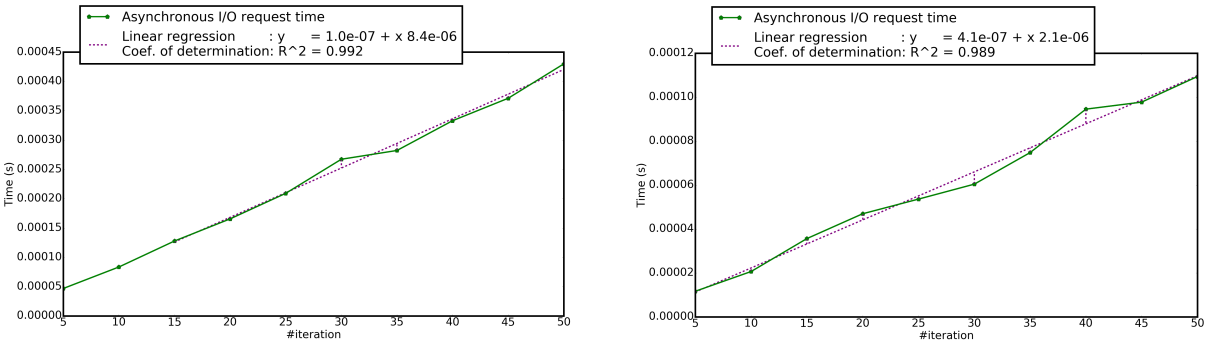
Modelling the asynchronous I/O request time (*req*)

Here, we evaluate the time needed to submit an asynchronous I/O request on the considered hardware platform. Let us also check the validity of its constancy (stated in Section 3.2.2) with respect to the number of submitted requests).

In Figure 4.4, we present our experimental evaluation of the asynchronous I/O request time (*req*) within the two considered hardware platforms. Using the numerical method described in Section 3.2.2, we estimate the value of the request time as well as the validity of the hypothesis: constant request time (for a given hardware platform).

The value of *req* is the slop of the linear regression (Figure 4.4). This value is, respectively, $8.4e^{-6}$ seconds and $2.1e^{-6}$ seconds for the *Intel Core* CPU i7-6700 and the HPC *Intel Xeon*

[6]Fitness of the linear regression with the experimental points



(a) *Intel Core* CPU i7-6700 @ 3.4 GHz (b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

Figure 4.4 – Experimental assessment of the time for forwarding a request to the asynchronous *write* thread

CPU E5-2680 v3 platform. Moreover, the *coefficient of determination* ($R^2 = 0.992$ and $R^2 = 0.989$) do not contradict our hypothesis about the constancy of the asynchronous I/O request time.

4.2 Real-life experimental case: the *Cube re-mapper*

Let us now see how the previously presented results may be transposed to a real-life example: the *Cube re-mapper*.
The objective of this section is to check whether the previously noticed improvements brought by our asynchronous *write* implementation may still be achieved within a real-life application. We also aim to identify any additional perturbation that our asynchronous I/O solution may introduce on a general-purpose computation. In a second step, we evaluate our custom solutions to eliminate (or at least to reduce) these undesirable interferences.

All the presented experimental results are obtained following the same protocol. Each considered point is assessed (experimental run) 10 times in a row. The presented corresponding value is the average of the result of these 10 runs. We also present an error bar showing the distance between the maximal and minimal value of these 10 runs.
It is also worthwhile to mention that for all the following experimentations, the used inputs[7] of the *Cube re-mapper* have not been processed nor modified. Thus, the observed executions of the *Cube re-mapper* are performed in a real-life conditions and with real-life data samples.
All the used input data samples (performance profiling file) have been obtained by profiling the execution of the *NAS parallel benchmark*[8]. They all represent the execution of about 10^5 threads. Each file is about 10^9 Bytes large.

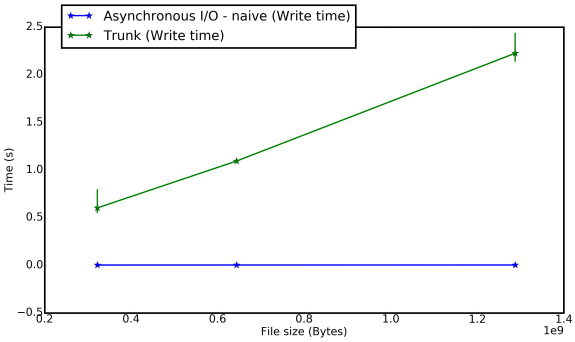
4.2.1 Basic POSIX-based asynchronous implementation

In this first step, we embed our basic asynchronous writing strategy within the *Cube re-mapper*. The experimental assessments of this implementation are compared in Figures 4.5 and 4.6 to

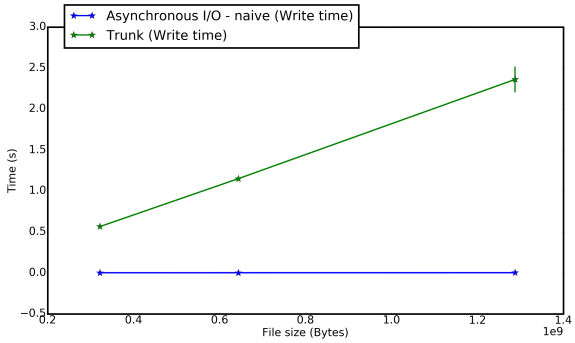
[7]Performance profile file (see Section 2.1)

[8]Set of programs designed by *NASA* to help evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications

the state-of-the-art (synchronous) implementation of the *Cube re-mapper* .



(a) *Intel Core* CPU i7-6700 @ 3.4 GHz



(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

Figure 4.5 – Experimental comparison of the *write* time: proposed *asynchronous I/O* (naive) VS. *state-of-the-art* (trunk synchronous) implementation of the *Cube re-mapper*

As one could expect, Figure 4.5 shows that the *write* time of our custom basic implementation is now almost null (compared to the synchronous *write* time). Instead of waiting for a costly I/O *write* time at each iteration, we simply wait for the time to enqueue a request within a shared queue. Given the reduced *write* time (compared to the *compute* time), the asynchronous I/O strategy has a negligible final waiting[9] time.

However, the significant performance gain achieved in the *write* operation is not transposed on the total time. Indeed, Figures 4.6c and 4.6d show that, on both considered platforms, the overall performance of our custom implementation is roughly comparable to that of the *state-of-the-art*. The gain achieved in the *write* operation is lost in the *compute* operation[10]. More concerning is the *compute* operation in our implementation. Its value fluctuates within a range from 40 to 80%. Moreover, although our custom version and the state-of-the-art version use the same *compute* function, we obtain a different response-time when this function is shipped to our implementation of the *Cube re-mapper*.

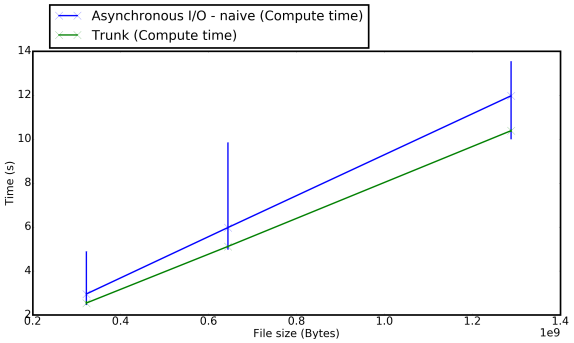
4.2.2 Improving the asynchronous-thread scheduling policy

Based on our previous experimental observations, one could notice the interference between the *compute* operation and our custom *write* implementation. Indeed, the *compute* operation becomes less efficient once shipped to our version of the *Cube re-mapper* (despite the fact that it is the same implementations).
Given the instability of this perturbation (see Figures 4.6a and 4.6b), our first hypothesis to explain this phenomenon is related to the OS-scheduler policy.

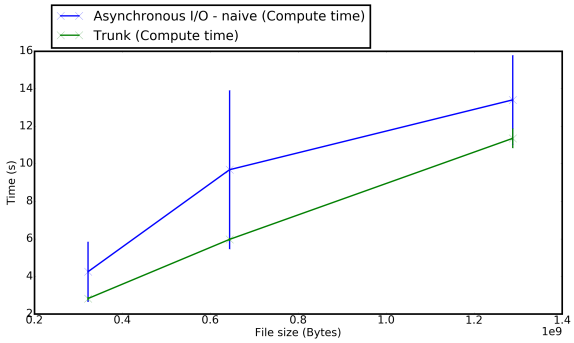
As all the considered threads (those processing the *compute* and the *write* operations) belong to the same process, the OS scheduler will tend to run them on the same CPU-core as often

[9]Time to wait for the pending asynchronous I/O request to be executed once all the *compute* operations have been executed

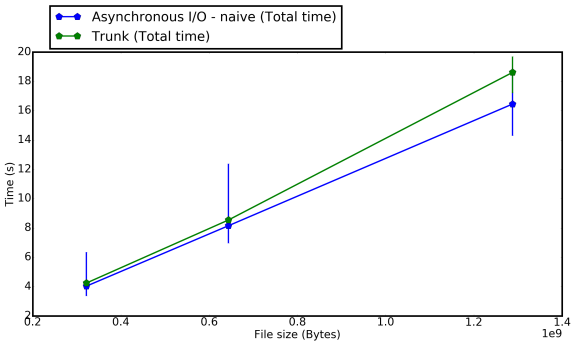
[10]This performance is shown by Figure 4.6



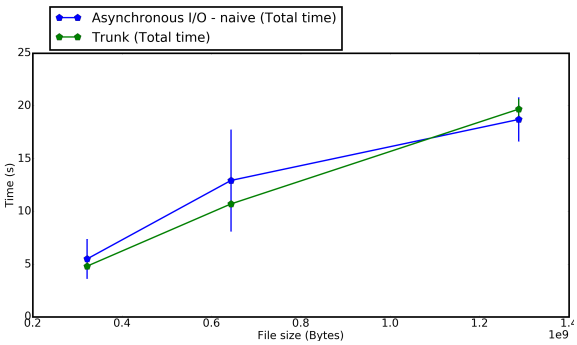
(a) *Intel Core* CPU i7-6700 @ 3.4 GHz



(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz



(c) *Intel Core* CPU i7-6700 @ 3.4 GHz



(d) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

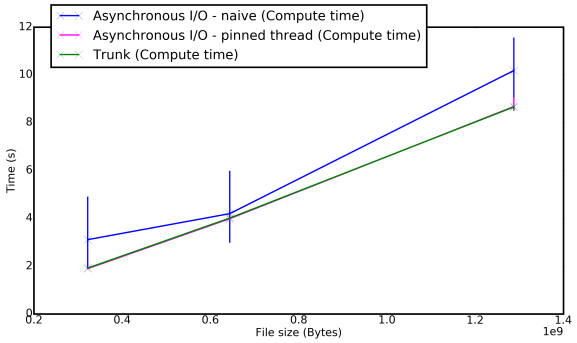
Figure 4.6 – Experimental comparison of the *compute* and *total* time: proposed *asynchronous I/O* (naive) VS. *state-of-the-art* (trunk synchronous) implementation of the *Cube re-mapper*

as possible (for cache proximity purpose). Thus, the execution time of the *compute* thread is time-sliced and regularly interrupted for the execution of the *write* threads, most likely on the same CPU-core. This may explain the delay observed on the experimental assessment of the compute *operation* in our custom implementation (see Figures 4.6a and 4.6b).

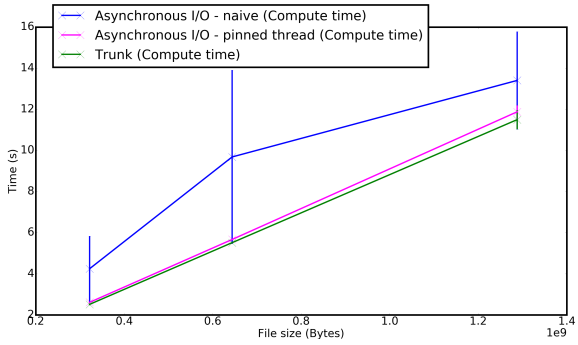
Our solution to this resource-multiplexing issue is to pin each running thread on an independent CPU-core. Assuming that our implementation has a relatively reduced amount of shared data and instructions between these threads (see Section 3.1.3) then, having threads running on independent cores would significantly improve the parallelization with a reduced time overhead[11].
In order to pin the threads created by the asynchronous I/O library, we have used the custom wrapper of the *pthread* library described in Section 3.1.2.

Figure 4.7 shows the impact of setting the thread affinity on the *compute* operation in our custom implementation. Indeed, Figures 4.7a and 4.7b show that the gap between the response time of the *compute* operation in our enhanced custom implementation and the state-of-the-art one has been decreased to respectively 5 and 2% (compared to that of Figures 4.6a and 4.6b) on the HPC *Intel Xeon* CPU E5-2680 v3and the *Intel Core* CPU i7-6700platforms.
Likewise, the size of the fluctuation of this operation has significantly collapsed (compared to that of Figure 4.6a and 4.6b).

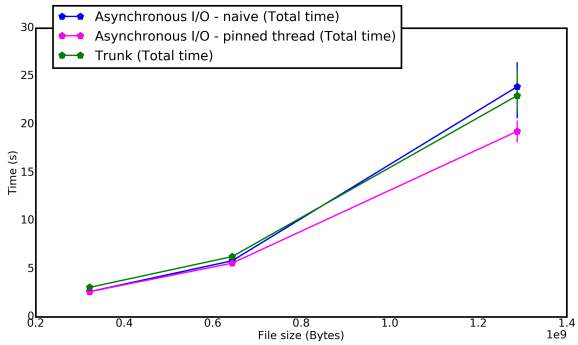
[11]Minimal shared data is equivalent to minimal synchronization required



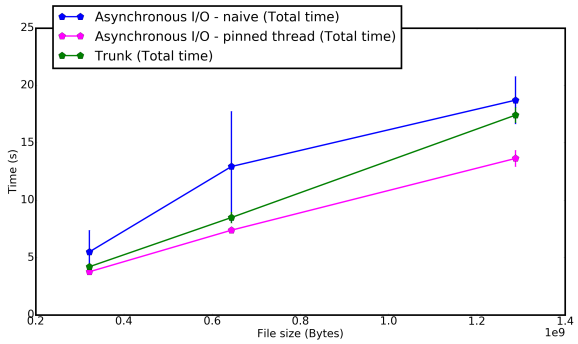
(a) *Intel Core* CPU i7-6700 @ 3.4 GHz



(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz



(c) *Intel Core* CPU i7-6700 @ 3.4 GHz



(d) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

Figure 4.7 – Experimental comparison of the *compute* and *total* time: proposed *asynchronous I/O* (*pinned thread*) VS. proposed *asynchronous I/O* (naive) VS. *state-of-the-art* (trunk synchronous) implementation of the *Cube re-mapper*

Consequently, the overall gain achieved by our enhanced asynchronous strategy is now clearly confirmed. In Figures 4.7c and 4.7d, the total time of our implementation is about 60% more efficient than that of the state-of-the-art. Moreover, the error margin of the *compute* operation has been dramatically reduced. Thus, the observed overall improvement is now clearly achieved and in a more robust manner (regardless of the error margin).

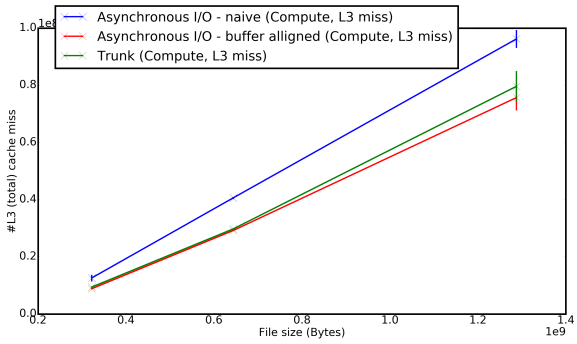
4.2.3 Reducing the impact of false-sharing

Despite our effort to reduce the number of shared memory addresses and reduce the impact of synchronization between threads, Figures 4.6 and 4.7 show that the overhead due to synchronization might still be reduced[12]. Given that we managed to use independent buffers between concurrent threads (RAM level), our explanation is that this interference is due to *false-sharing* (cache level).

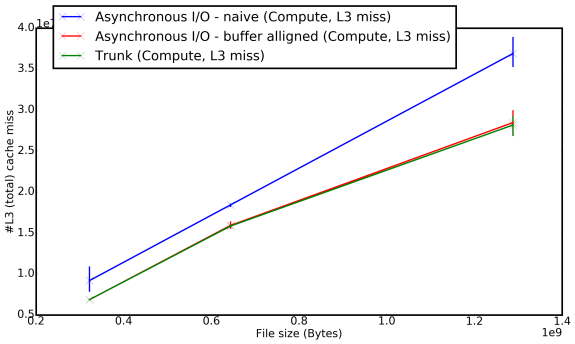
In Figures 4.8a and 4.8b, we have represented the cache access foot-print of each one of the *state-of-the-art* (*trunk*), the *naive* and the *aligned buffer* implementations of the *Cube re-mapper*. As one can notice, the number of cache misses in the *naive* version of the code is about 33% higher than that of the *state-of-the-art* one. Given that these two curves profile the

[12]The execution time of the *compute* function in our implementation is still slightly higher than the one in the *state-of-the-art* implementation

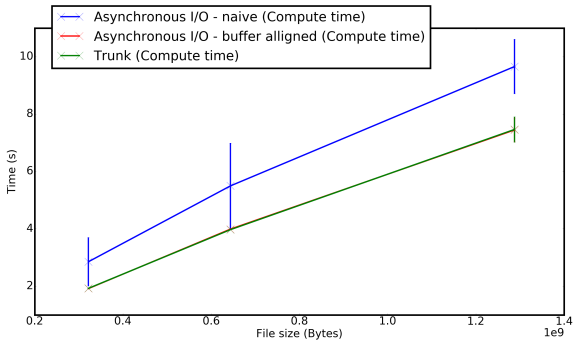
same implementation of the *compute* function, this shows the impact of *false-sharing* on our implementation. This result is also correlated with the one at time level (Figure 4.8c and 4.8d).



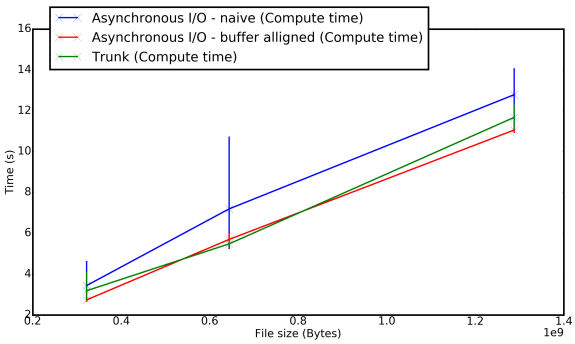
(a) *Intel Core* CPU i7-6700 @ 3.4 GHz



(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz



(c) *Intel Core* CPU i7-6700@ 3.4 GHz



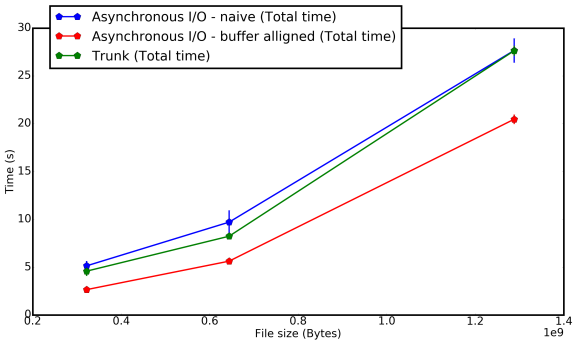
(d) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

Figure 4.8 – Experimental comparison of the *L3 cache-miss* (total) and *compute* time: proposed *asynchronous I/O* (buffer aligned) VS. proposed *asynchronous I/O* (naive) VS. *state-of-the-art* (trunk synchronous) implementation of the *Cube re-mapper*. For the seek of clarity the only results represented are the one linked to the *compute* operation

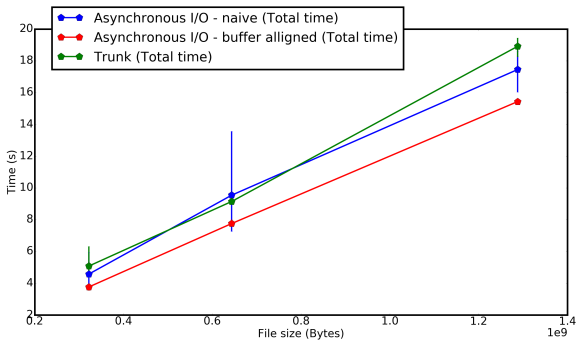
Furthermore, we see in Figures 4.8a and 4.8b that the rate of cache misses in our *buffer aligned* version drops down to the same level as that of the *state-of-the-art* implementation. Thus, our improvement allows to eliminate the cache-linked interferences between concurrent threads.
This improvement, noticed at cache level, has a clear impact on the overall performance of our implementation. Indeed, we observe in Figure 4.9 an average gain of 70% of the total time. Given the increase of the stability in the performance of this *buffer aligned* implementation, this gain is achieved regardless of the error margin.

4.2.4 Further improvement: adapting the dynamic memory allocation

So far, all the performance gain has been obtained by focusing on the *write* implementation. Likewise, the improvements described in Sections 4.2.2 and 4.2.3 were aimed at reducing the perturbation of our *write* implementation with respect to the *compute* one. The objective was



(a) *Intel Core* CPU i7-6700 @ 3.4 GHz

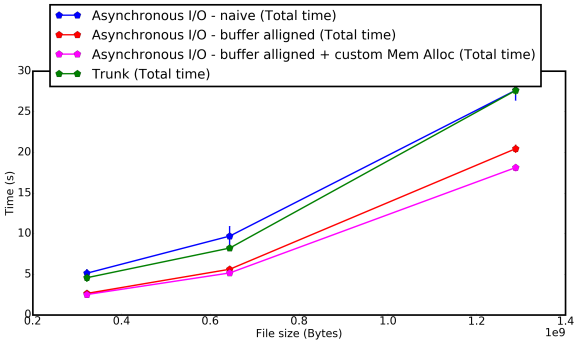


(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

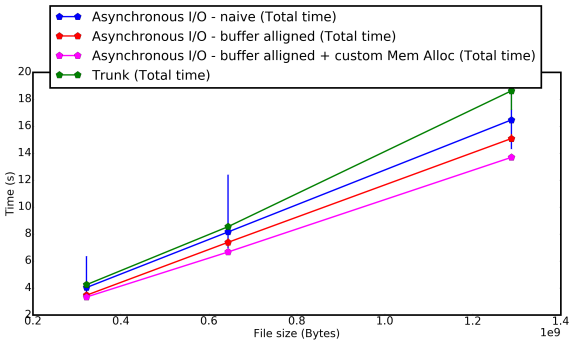
Figure 4.9 – Experimental comparison of the *total* time: proposed *asynchronous I/O* (buffer aligned) VS. proposed *asynchronous I/O* (naive) VS. *state-of-the-art* (trunk synchronous) implementation of the *Cube re-mapper*.

to bring the performance of this *compute* implementation [13] to the same level as that of the *state-of-the-art* version.

We now assess our full-fledged asynchronous I/O implementation of the *Cube re-mapper*. This final version consists in using the *buffer aligned* version coupled with our custom memory allocator (see Section 3.1.3). Figure 4.10, which presents the experimental response time, shows that our full-fledged implementation has been able to provide a slight average gain of 19% compared to the *buffer aligned* implementation. However, much more significant gain (64%) is achieved when compared to the original *state-of-the-art* implementation.



(a) *Intel Core* CPU i7-6700 @ 3.4 GHz



(b) HPC *Intel Xeon* CPU E5-2680 v3 @ 2.5 GHz

Figure 4.10 – Experimental comparison of the *total* time: proposed *full-fledged* (buffer aligned + custom mem alloc) VS. proposed *asynchronous I/O* (buffer aligned) VS. proposed *asynchronous I/O* (naive) VS. *state-of-the-art* (trunk synchronous) implementation of the *Cube re-mapper*.

[13] Unchanged through all the versions of the *Cube re-mapper* that we have presented till now

— 5 —
Conclusion

Performance-profiling and analysing is a mandatory step when developing an HPC-specific application. In this context, our objective was to outperform a specific tool for HPC-performance analysing: the *Cube re-mapper*. Given the amount of data (stored on the I/O disk) to be processed by such a software, our study has focused on enhancing the algorithms of the I/O-resource access.

From a theoretical perspective, our study has considered a quite commonly-used computation-pattern[1]. It has proposed an asynchronous I/O approach to enhance this pattern as well as the theoretical models related to this custom version of the pattern. This study has enabled to model accurately the response time of the considered pattern when the I/O *write* functions are overlapped with the *compute* ones. Our models also accurately predict the improvement brought by our overlapping approach on a specific implementation of the *Cube re-mapper* pattern (referred to as the simulation test-bed). More importantly, our study has enabled to unveil the parameters that influence most this improvement.

Consequently, our custom implementation of the overlapped I/O *write*[2] is shown to potentially outperform the response time of this well-known programming pattern: up to 75% improvement[3] compared to the original version of the pattern (assuming an equivalent *compute* and *write* times).

From a programmer perspective, our study has highlighted the limitations of using such a well-considered overlapping (asynchronous I/O) approach on the considered pattern (the *Cube re-mapper* is used as flagship of this pattern). This study has made an assessment of some targeted hardware dimensions (processor running/stall time, L3 cache miss-rate) to emphasize the interferences produced by the asynchronous I/O approach on the considered pattern. The study has also proposed two custom solutions[4] that dramatically soften the impact of such interferences. Finally, this study has demonstrated the usefulness of our custom memory allocator which was designed to enhance the interaction between the *compute* thread and the *write* threads.

[1] The pattern followed by the *Cube re-mapper* (see Section 2.2)

[2] Using the POSIX AIO library coupled with our custom synchronization implementation

[3] Experimental result relative to the considered hardware

[4] Namely: adapting the kernel thread-scheduling policy and aligning the dynamic memory allocation with the cache lines

Consequently, our best version of the *Cube re-mapper* is shown to bring an improvement up to 64% compared to the original version of this software.

In the pattern considered in this study, a buffer created by the *compute* function at a given iteration is never accessed (read nor write) by a later *compute* call. Thus, this buffer might be written at any more convenient time. This data-access pattern of the *Cube re-mapper* has led us to reduce the overall time-response by overlapping the I/O access with the *compute* function. However, it could also be an argument to go further and make the whole pattern of the *Cube re-mapper* parallel. In this context, the *compute* operation could be processed concurrently with the *write* operation as well as with other *compute* calls (at other iterations). A parallel loop (example: *MPI* or *Cilk* parallel loop) would thus be used in addition to the already overlapped *write* and *compute* thread.

Bibliography

[1] Fifty years of moore's law, https://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html.

[2] Instruction pipelining, https://en.wikipedia.org/wiki/Instruction_pipelining.

[3] Just: Juelich storage cluster, http://www.fz-juelich.de/ias/jsc/EN/Expertise/Datamanagement/OnlineStorage/JUST/JUST_node.html.

[4] Unix: multitasking, multiuser computer operating systems, https://en.wikipedia.org/wiki/Unix.

[5] Vector processor, https://en.wikipedia.org/wiki/Vector_processor.

[6] Suparna Bhattacharya, Steven Pratt, Badari Pulavarty, and Janet Morgan. Asynchronous i/o support in linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[7] Millad Ghane, Abid M Malik, Barbara Chapman, and Ahmad Qawasmeh. False sharing detection in openmp applications using ompt api. In *International Workshop on OpenMP*, pages 102–114. Springer, 2015.

[8] George Em Karniadakis and Robert M Kirby II. *Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation*. Cambridge University Press, 2003.

[9] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S Müller, and Wolfgang E Nagel. The vampir performance analysis tool-set. *Tools for High Performance Computing*, pages 139–155, 2008.

[10] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[11] William B Langdon and Mark Harman. Genetically improved cuda c++ software. In *European Conference on Genetic Programming*, pages 87–99. Springer, 2014.

[12] Yingping Lu and David HC Du. Performance study of iscsi-based storage subsystems. *IEEE communications magazine*, 41(8):76–82, 2003.

[13] William Magro, Paul Petersen, and Sanjiv Shah. Hyper-threading technology: Impact on compute-intensive workloads. *Intel Technology Journal*, 6(1), 2002.

[14] R Hugo Patterson and Garth A Gibson. Exposing i/o concurrency with informed prefetching. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 7–16. IEEE, 1994.

[15] Bradley J Pedersen and Werner Kurt Perry. Method for supporting an extensible and dynamically bindable protocol stack in a distrubited process system, October 20 1998. US Patent 5,826,027.

[16] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. Controlling cache utilization of hpc applications. 2011.

[17] Darko Petrović, Thomas Ropars, and André Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, page 17. ACM, 2015.

[18] William K Robertson, Christopher Kruegel, Darren Mutz, and Fredrik Valeur. Run-time detection of heap-based overflows. In *LISA*, volume 3, pages 51–60, 2003.

[19] Arch D Robison. Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST*, 18:25, 2012.

[20] Robert Ross, Daniel Nurmi, Albert Cheng, and Michael Zingale. A case study in application i/o on linux clusters. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, pages 11–11. ACM, 2001.

[21] Pavel Saviankou, Michael Knobloch, Anke Visser, and Bernd Mohr. Cube v4: from performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, 2015.

[22] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[23] Paulus Stravers and Jan-Willem van de Waerdt. Translation lookaside buffer, December 10 2013. US Patent 8,607,026.

[24] Ashkan Tousimojarad and Wim Vanderbauwhede. Comparison of three popular parallel programming models on the intel xeon phi. In *Euro-Par Workshops (2)*, pages 314–325, 2014.

[25] John D Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[26] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. Upc++: a pgas extension for c++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1105–1114. IEEE, 2014.

[27] Ilya Zhukov and Brian JN Wylie. Assessing measurement and analysis performance and scalability of scalasca 2.0. In *European Conference on Parallel Processing*, pages 627–636. Springer Berlin Heidelberg, 2013.