Extending *OmpSs* programming model with task reductions: A compiler and runtime approach

Author: Ferran Pallarès

Director: Eduard Ayguadé (DAC) *Co-director:* Vicenç Beltran (BSC-CNS)

Degree in Informatics Engineering
Computer Engineering specialization

Barcelona Supercomputing Center - *Centro Nacional de Supercomputación* (BSC-CNS)
Barcelona Schools of Informatics (FIB)
Polytechnic University of Catalonia (UPC) - BarcelonaTech

24$^{\text{th}}$ January, 2017

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

Acknowledgments

I would like to express my sincere gratitude to my director Eduard Ayguadé who trusted and gave me the opportunity to get involved in the BSC and work in the field of High-Performance Computing (HPC).

I sincerely thank Vicenç Beltran for encouraging me to get engaged in this project and providing the necessary guidance to go through it.

The completion of this study could not have been possible without the expertise and invaluable help of my advisors Sergi Mateo and Josep Maria Pérez, always willing to help and clear any of my doubts out. Thank you for having the patience of teaching me.

Finally I need to thank my partner, my family and friends with special affection not only for this step but for being always at my side believing in me and supporting my decisions. I would not be where I am without them.

Abstract

After a brief overview of the task-based *OmpSs* programming model and the related literature, this document aims to show the process of extending the current infrastructure of this model in order to support reductions. The first step towards supporting reductions in *OmpSs* was the design and implementation of the `concurrent` clause, used to annotate tasks that access concurrently for updating some shared data, and thus providing a user-driven mechanism to compute reductions. Then, the underlying idea behind this mechanism was extended into a first implementation of the `reduction` clause based on a task privatization strategy. Next, a different approach for the clause based on task-and-CPU privatization was developed. Finally, both strategies were evaluated for a set of different architectures by subjecting them to the n-queens, dot product and unbalanced tree search benchmarks.

Resum

Després d'una breu descripció del model de programació basat en tasques *OmpSs* i altres estudis relacionats, aquest document té per objectiu mostrar el procés d'extensió de la infraestructura d'aquest model per tal d'oferir suport a les reduccions. El primer pas en aquesta direcció va ser el disseny i implementació de la clàusula `concurrent`, utilitzada per anotar tasques que accedeixen concurrentment a certes dades per actualitzar-les i, per tant, proveint un mecanisme a nivell d'usuari per poder computar reduccions. A continuació, la idea subjacent rere aquest mecanisme es va estendre cap a una primera implementació de la clàusula `reduction` basada en una estratègia de privatització per tasca. Més endavant es va considerar un enfocament diferent pel desenvolupament de la clàusula basat en una privatització per tasca i CPU. Finalment, ambdues estratègies es van comparar per un conjunt d'arquitectures diverses per mitjà de sotmetre-les a un joc de proves format per els *benchmarks n-queens*, *dot product* i *unbalanced tree search*.

Resumen

Después de una breve descripción del modelo de programación basado en tareas *OmpSs* y otros estudios relacionados, este documento tiene por objeto mostrar el proceso de extensión de la infraestructura de este modelo con tal de ofrecer soporte a las reducciones. El primer paso en ésta dirección fue el diseño y implementación de la cláusula `concurrent`, utilizada para anotar tareas que acceden concurrentemente a ciertos datos para actualizarlos y, por lo tanto, proveyendo un mecanismo a nivel de usuario para poder computar reducciones. A continuación, la idea subyacente a este mecanismo se extendió hacia una primera implementación de la cláusula `reduction`, basada en una estrategia de privatización por tarea. Más adelante se consideró un enfoque distinto para el desarrollo de la cláusula basado en una privatización por tarea y CPU. Finalmente, ambas estrategias se compararon por un conjunto de arquitecturas diversas por medio de someterlas a un juego de pruebas formado por los *benchmarks n-queens*, *dot product* y *unbalanced tree search*.

Contents

List of Figures

List of Tables

Listings

Acronyms

API Application Programming Interface. 1, 3, 6, 34, 37, 39–41, 44

BOE *Boletín Oficial del Estado*. 20

BSC-CNS Barcelona Supercomputing Center - *Centro Nacional de Supercomputación*. 1, 3, 18, 21, 22, 64, 67

CCPI Cavium Coherent Processor Interconnect. 50

CPU Central Processing Unit. vii, 1, 30, 39–44, 62, 65

DDR4 Double Data Rate type 4 Synchronous Dynamic Random-Access Memory (DRAM). 48

DIMM Dual-Inline Memory Modules. 48

DRAM Dynamic Random-Access Memory. viii, 48

ECTS European Credit Transfer and Accumulation System. 13

FIB Barcelona Schools of Informatics. 1, 3, 13

FPGA Field-Programmable Gate Array. 1

GCC GNU Not Unix (GNU) Compiler Collection. 18, 49, 50, 58, 60

GEP Project Management Course. 13

GNU GNU Not Unix. viii, 18, 19, 67

GNU GPLv3 GNU General Public License version 3. 67

GPFS General Parallel File System. 64

GPU Graphics Processor Unit. 1

HPC High-Performance Computing. i, 1, 20

KNL Knights Landing. iv, vi, vii, 48, 49, 56–58, 60, 70, 72

LSF Platform Load Sharing Facility. 47

MCDRAM Multi-Channel DRAM. 48

MIC Many Integrated Core. 48

NUMA Non-Uniform Memory Access. 42, 47, 49, 54, 56, 60

SHA-1 Secure Hash Algorithm 1. 53, 70

SLURM Simple Linux Utility for Resource Management. 47

SoC System on Chip. 49, 50

SSH Secure SHell. 14, 19

UDR User-defined Reduction. 40, 62, 63

UPC Polytechnic University of Catalonia. 1

UTS unbalanced tree search. vi, 52, 54–61, 70, 71

VPN Virtual Private Network. 14

1 Introduction

Today, multiprocessors are present in most modern computer systems. From the most powerful supercomputers, where good performance is imperative, to embedded systems like mobile phones, designed for purely entertainment, having more than one CPU has become a very common trait.

The increase in multiprocessor systems has entailed an equally-proportionate rise in the amount of software willing to take advantage of them, by means of parallel programming. When it comes to HPC and scientific research, the need to create optimized software for those architectures is even greater. Moreover, if these multiprocessor systems are enhanced by including powerful accelerators like graphic cards, coprocessors or Field-Programmable Gate Arrays (FPGAs), the performance can be vastly increased in many situations.

On the downside, programming for those heterogeneous systems can sometimes be tiresome and complicated. It requires not only precise knowledge of the system but also having to produce specific code for each of the devices that are to be supported, having to maintain many different versions of the code and therefore having to develop, test and optimize every new feature for all of them.

Having these limitations in mind, some programming models like *OmpSs* were created to take advantage of the benefits these heterogeneous systems offer while providing the developer user-friendly abstractions to deal with them.

1.1 *OmpSs* programming model

OmpSs is a task-based programming model composed of a set of directives and library routines that can be used in conjunction with a high-level programming language in order to develop concurrent applications.[7] This model is being actively developed by the Programming Models group of the Computer Sciences department of the *BSC-CNS*.

The goal of *OmpSs* is to provide a productive environment to develop applications for modern HPC systems. In particular, *OmpSs* aims to extend *OpenMP* with new features related to asynchronous parallelism and device heterogeneity (like Graphics Processor Units (GPUs)).[7,9] *OpenMP* is a well-known programming model for shared memory parallel programming, almost a standard for the such paradigm in the world of HPC. *OpenMP* consists of a set of compiler directives and library routines that implement a multi-platform Application Programming Interface (API) for C/C++ and Fortran.[3]

Some of the contributions of the *OmpSs* programming model that are already part of the *OpenMP* standard include:

- Task dependences (included in *OpenMP* 4.0)
- *OpenMP* SIMD extensions (included in *OpenMP* 4.0)
- Task priorities (included in *OpenMP* 4.5)

Tasks are the mechanism used to provide asynchronous parallelism in *OmpSs* and can be defined as its elementary unit of work: A task represents a specific instance of an executable code. The users specify tasks in their source code by using the *pragma oss* directive. In addition, they are encouraged to annotate what data is going to be accessed by a task and how these accesses will be by using the `in`, `out` and `inout` clauses (standing for input, output and input/output respectively). This information can be used at run-time to determine the dependences between those tasks and schedule their execution accordingly to avoid data races. This concept is known as data flow.

In the code listing 1 we can see an example of a taskified code in *OmpSs* where the data accesses have been annotated. The corresponding generated dependence graph can be seen in figure 1. For our example, tasks `T2` and `T3` could run in parallel provided enough resources are available.

The `taskwait` clause is used in *OmpSs* to specify a task synchronization point, every created task up to that point needs to finish before being able to continue with the execution.

Listing 1: Data flow example in *OmpSs*

```
int a, b = 0;
int tmp;

#pragma oss task inout(a, b) label(T1)
{
    b += compute_value(...);
    a += compute_value(...);
}

#pragma oss task in(a) out(tmp) label(T2)
tmp = a + compute_value(...);

#pragma oss task in(a) label(T3)
print(a);

#pragma oss task in(b, tmp) label(T4)
print(b*tmp);

#pragma oss task inout(a, b) label(T5)
{
    b *= compute_value(...);
    a *= compute_value(...);
}

#pragma oss taskwait
```

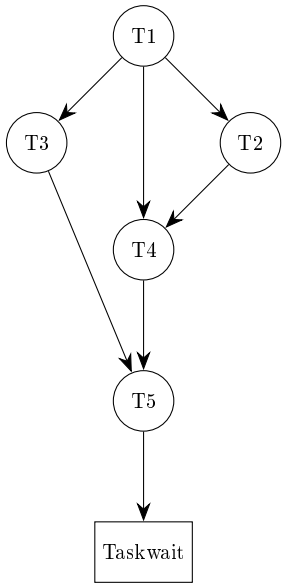


Figure 1: Task dependences in *OmpSs*

One of the main differences between dependences in *OmpSs* and *OpenMP* is that the *OmpSs* specification requires keeping track of the dependences between child tasks while the *OpenMP* specifications does not.

To illustrate this point consider the example in the code listing 2. The expected value of the a variable after the program is executed is 1.

Keeping in mind the differences between both models, in order to generate an equivalent code in *OpenMP* a synchronization point (i.e. `taskwait` or `taskgroup`) is required inside the task labeled *W1*. The reason of this necessary `taskwait` is that the *OpenMP* specification does not specify that any dependence should be registered between the tasks *W2* and *R* for being in a different task nesting level. The complete equivalent code for the *OpenMP* model can be seen in the code list 3.

Listing 2: Nested dependences in *OmpSs* example

```
int a = 0;

#pragma oss task inout(a) label(W1)
{
    #pragma oss task inout(a) label(W2)
    a++;
}

#pragma oss task in(a) label(R)
printf("A: %d\n", a);

#pragma oss taskwait
```

Listing 3: Dependences in *OpenMP* example

```
int a = 0;

#pragma omp task depend(inout:a) shared(a)
{
    #pragma omp task shared(a)
    a++;

    #pragma omp taskwait
}

#pragma omp task depend(inout:a) shared(a)
printf("A: %d\n", a);

#pragma omp taskwait
```

The reference implementation of the *OmpSs* specification developed at *BSC-CNS* is composed of the *Mercurium* compiler and the *Nanos6* runtime library.

1.1.1 *Mercurium* compiler

The *Mercurium* compiler is a *source-to-source* compiler developed by the Programming Models group of the Computer Sciences department of the *BSC-CNS* for C, C++ and Fortran.[8]

Mercurium can be used to transform the *OmpSs* annotations included in the input source code into standard C, C++ and Fortran: new routines and calls to a runtime library. Then, it can handle the control to the native compiler so it is possible to generate the final executable binary.

While its main task is to perform the previous transformation, it is also capable of detecting and applying some compiler optimizations, resulting in a more efficient code.

1.1.2 *Nanos6* runtime library

Nanos6 is the name of the reference library that provides runtime support for the *OmpSs* programming model. It is the software piece responsible for implementing the API calls defined by the model standard. It keeps track of the created tasks, computes data dependences between them and provides a threading paradigm.

A range of operation modes exists to be able to tune the runtime execution. In particular, we can find instrumentation modes that register the program activity during its execution and generate traces for later analysis. Those traces can be very helpful to reason about the correctness of the code and for performance evaluation.

1.2 This project

The project described in this document corresponds to a Bachelor's thesis of the Informatics Engineering degree, Computer Engineering specialization, by the *FIB* developed in collaboration with the *BSC-CNS*.

This project is entirely contained within the *OmpSs* project and there resides its complete utility. As a result, it shares with it the final goal of providing a user-friendly environment for the development of high-performance parallel applications.

In particular, the aim of this project is to extend the *OmpSs* infrastructure with a new feature: **Task reductions**. Specifically, the work will be centered in scalar values, leaving reductions on arrays and other structures for future research.

In order to provide a complete implementation of task reductions for scalars, both the *Mercurium* compiler and the *Nanos* runtime library will need to be extended.

It is worth pointing out that this project will be extending the next release of the *Nanos* runtime, code-named *Nanos6*. *Nanos6* has been redesigned and written completely from scratch, thus having no other relation with the current production version of *Nanos* than its function.

In short, the work of the project can be divided into four different parts: The first one is basically extending the *OmpSs* infrastructure from a theoretical point of view. The second and third sections focus on implementing task reductions on scalars for both the *Mercurium* compiler and the *Nanos6* runtime. Finally, the last section presents the evaluation of this new feature in different computer architectures.

1.3 Document structure

This document serves the purpose of giving the reader a broad view of the project topic and a more detailed description of the project itself.

The document is divided into three differentiated parts. The first part focuses on the management aspects of the project itself, including but not limited to the project context and its reason to be, the methodology to follow and a planning and budget estimation.

The second part is about the work done in the development and implementation phases of the project, explained from a technical point of view. After, the evaluation and results are presented, concluding the part with the conclusions and a short section about future work.

Finally, the third and last part of the document aims to perform an objective revision on the planning and budget estimations shown in the project management part in order to show what the reality has been.

In addition, references and appendixes are compiled in attached sections so that the reader can verify the information sources or delve into the project work if interested.

2 Problem description and goals

Reductions are a common algorithmic pattern found in many scientific applications[11]. In a reduction, a collection of objects are *reduced* to a single object by combining them pairwise with a binary operator.[21,22]

A reduction is an iterative update of a variable, defined as:

$$var := op(var, expr)$$

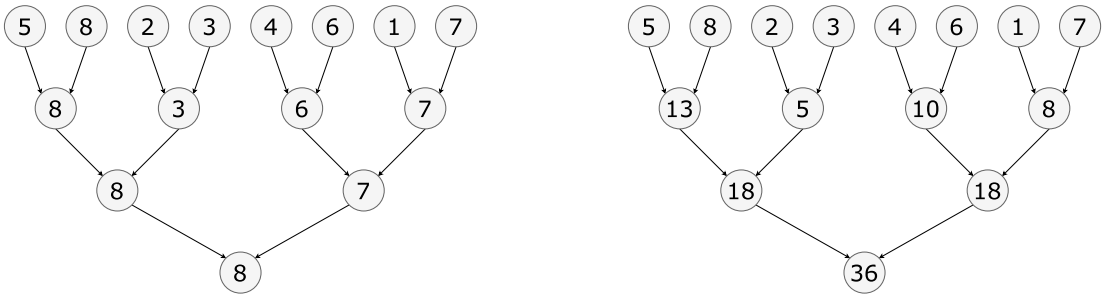
Where *var* is the variable where to combine the objects, *op* is the binary operator and *expr* is an expression that does not modify the value of *var*.

While reductions do not enforce that the operator satisfies the associative and commutative properties by definition, most times it does. When assuming this, parallel implementations of the pattern are possible, and considerable speedup can be obtained accordingly.

Special care must be taken when dealing with floating point arithmetic, as it is only approximately associative. In other words, different order in the operands can give different results due to round-off errors.[29]

Reductions are characterized for having non-atomic updates involving an accumulator variable and an expression, requiring exclusive access to ensure data consistency and making their execution computationally expensive and parallelization challenging.[11]

In figure 2 we can see a reduction pattern for *max* and *add* operators.



(a) Reduction pattern for *max* operator

(b) Reduction pattern for *add* operator

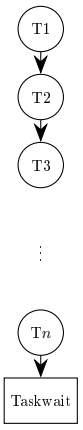
Figure 2: Reduction pattern examples

In the current *OmpSs* infrastructure and data flow model, the reduction pattern would be taskified using `inout` clauses as shown in the code listing 4. However, the dependence graph that corresponds to this scheme in figure 3 shows how those dependences would serialize the execution and thus a poor performance would be obtained.

Listing 4: Reductions in the current *OmpSs* infrastructure

```
int result = 0;
int *values = allocate(N);

#pragma oss task inout(values[0:N-1])
initialize(values, N);

for (int i = 0; i < N; ++i)
{
    #pragma oss task in(values[i]) inout(result)
    {
        result += function(values[i]);
    }
}

#pragma oss taskwait
```

Figure 3: Current dependence graph for reduction pattern in the current *OmpSs* infrastructure



That being said, the reduction pattern itself does not enforce such a rigid scheduling and, considering the associative and commutative properties are satisfied, a greater degree of concurrence can be attained. In detail, tasks participating in the reduction have an output dependence on the reduction variable for the accumulation of the partial computed result. In this sense, if we are able to postpone or handle these dependencies without causing data hazards, the resulting dependence graph should become similar to figure 4.

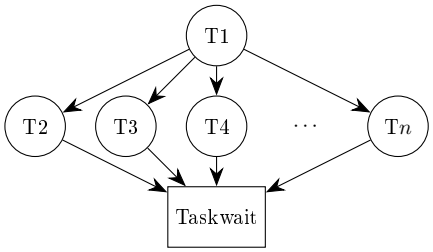


Figure 4: Desired dependence graph for reduction pattern in *OmpSs* infrastructure

Our work in this project is to study how this transformation can be better done.

2.1 Project objectives

The project objectives are listed as follows:

1. Extension of the *OmpSs* parallel programming model infrastructure to include the task reductions feature.
   (a) Theoretical analysis and API specification.
   (b) Extension of the *Mercurium* compiler.
   (c) Extension of the *Nanos6* runtime.
2. Evaluation of the task reductions feature.
   (a) Determination of appropriate metrics for an objective comparison.
   (b) Design of test sets that allow gathering relevant execution traces.
   (c) Comparison of different implementations on different computer architectures.

2.2 Scope

The purpose of this section is to reason about what belongs to this project and what, being the reason either time limitations or differences in the subject, does not.

In the first place, it should be recalled how this project is fully contained within its parent project *OmpSs*. Therefore, it is not intended in any way that this project goes beyond the scope of the *OmpSs* project. Considering this, the extent of this project is clearly limited to the analysis, implementation, comparison and testing of scalar-value reductions within the *OmpSs* task-driven parallel programming model.

Due to lack of time and the extra complexity involved, reductions of arrays and structures other than scalars themselves remain outside the scope of this work.

When it comes to computation and management of dependences between tasks, two distinct approaches are to be considered: On the one hand, **discrete** dependences, where an address (or *lvalue*[25,12] in general) is considered to represent the dependence. On the other hand, **fragmented linear regions** dependences, where more complex structures are used to represent all memory positions involved in the dependence.

As far as this project is concerned, a complete version of task reductions based on discrete dependences will be preferred over an incomplete version supporting both discrete and region-based dependences. In this sense, an implementation involving task reductions based on the region-based dependence system will only be considered if the version based on discrete dependences is complete and there is still time left to fit it in this project duration.

Finally, it should be noted that the project is also limited to the current *OmpSs* infrastructure under development, composed by *Mercurium* and *Nanos6*, and has no intention of going beyond those components.

2.3 Obstacles

When it comes to the project definition and its scope, it is important to have a mindful study of the obstacles one may encounter during its course.

Having consciousness of the possible hindrances when carrying the project out may help us avoiding them, staying focused on the development and averting unforeseen work.

2.3.1 Design or implementation errors

Design errors in the initial stages of the project can lead to incorrect implementations and poor overall performance. Defining an adequate methodology with short cycles and feedback sessions can result in shorter reaction time and the subsequent reorientation to the desired direction.

Unnoticed errors introduced while developing the feature may be a problem in later stages of the project, inducing wrong results or longer execution times. To minimize their impact on the developing time, development should be interlaced with exhaustive testing, with the intention of detecting and solving them as soon as possible.

2.3.2 Unconnected errors that affect our implementation

It can be the case that some errors have been introduced in other parts of the involved projects but have not shown up for the moment. These errors can have a negative impact on the performance of the new feature, to the point of driving the execution to incorrect results in some situations. This must be minimized to the possible extent by detecting and reporting them to the responsible at the earliest opportunity. As in the previous obstacle, testing has an important role in preventing this situation.

2.3.3 Unavailability of machines for testing

Regarding the parallel requirements of the requested feature, it will be necessary to test its implementation on a range of different multiprocessor architectures. It is possible that, given external factors, some of the computation clusters are under high load or not available at all over a period of time.

While it is improbable that this situation has an extended duration, it will be convenient to interlace developing and testing phases, distributing the testing phase over a long period of time in order to minimize the risk of facing high loads during a testing session.

This strategy also allows a proper rescheduling of tasks if such situation is detected.

3 Contextualization

3.1 Actors

In the project context, we find a set of people that are affected either in a direct or indirect way. Details of who they are and what their role within the project is are listed in this section.

3.1.1 Main developer

The developer is the final responsible for the project outcome. Its job consists in not only developing the project in its entirety but it is also responsible for its management, documentation and testing process. Its work needs to ensure being able to meet the project deadlines and budget.

3.1.2 Directors

The directors of this project are Eduard Ayguadé Parra and Vicenç Beltran Querol. Their job consists in supervising the project and guiding it towards the desired direction, verifying that the work has the expected quality and that the objectives are accomplished. In addition, the directors can also assist the developer in both technical and academic matters in case of necessity.

3.1.3 Beneficiaries

- Direct beneficiaries The direct beneficiaries of this project are the developers whose software relies on the *OmpSs* programming model to deliver a good performance. The new feature will allow them to use reductions in a straightforward way, cutting on development time while avoiding implementation errors and providing optimized code.
- Indirect beneficiaries The indirect beneficiaries are the final users of the scientific applications or general software which uses the *OmpSs* programming model. Although they need not know about the existence of this project nor the *OmpSs* programming model, they will experience shorter execution times resulting in a better overall experience.

3.2 State of the art

This section intends to review the current literature on the topic in question: reductions in parallel programming models. Previous studies and similar approaches to the problem are discussed, explaining similarities and differences.

3.2.1 *OpenMP* reductions

The *OpenMP* specification currently includes a mechanism to perform reductions: the `reduction` clause. This clause can be found in a `parallel` construct or a `for` work-sharing construct.[2]

The syntax of the `reduction` clause in *OpenMP* is the following:

```
reduction(reduction-identifier :  list)
```

The `reduction` clause takes as arguments a *reduction-identifier* used to identify the reduction operation and a list of items to be reduced.

All valid *OpenMP* implementations should support a range of implicit reduction-identifiers for the most common operations (arithmetical, logical, bitwise, etc.). These identifiers include but are not limited to: `+` (addition), `*` (product), `&&` (logical and), `max` (maximum), etc. Moreover, *OpenMP* offers a mechanism so that the user can define its own reduction identifiers.

The *OpenMP* specification states that, for each item, a private copy is created in each implicit task, and is initialized with the initializer value of the *reduction-identifier*. After the end of the region, each list item is updated with the combination of the private copies by the combiner operation associated with the *reduction-identifier*.[2]

In the code listing 5 we can see how the `reduction` clause can be used within a `for` work-sharing construct in order to parallelize the dot product between two vectors.

Listing 5: *OpenMP* reduction clause example

```
float dot_product(const float *A, const float *B, unsigned int length)
{
    float result = 0;

    #pragma omp parallel for reduction(+:  result)
    for (unsigned int i = 0; i < length; i++)
    {
        result += A[i]*B[i];
    }

    return result;
}
```

In addition to the described `reduction` clause for the `parallel` and the `for` constructs, extended support for the *OpenMP* tasking model is expected for the next release of the model, as seen in the *OpenMP* 5.0 technical preview.[4]

In detail, the new clauses `task_reduction` and `in_reduction` will be added to the model. The first will be used as a reduction scoping clause, used to delimit the domain of the reduction. The latter is defined as a reduction participating clause, used within the reduction domain to annotate a task participating in that reduction. The syntax of the clauses is the following:

```
task_reduction(reduction-identifier :  list)
 in_reduction(reduction-identifier :  list)
```

In the code listing 6 we can see how the dot product example could be implemented using these new clauses.

Listing 6: *OpenMP* reduction example with tasks

```
float dot_product(const float *A, const float *B, unsigned int length)
{
    float result = 0;

    #pragma omp taskgroup task_reduction(+:  result)
    for (unsigned int i = 0; i < length; i++)
    {
        #pragma omp task in_reduction(+:  result)
        result += A[i]*B[i];
    }

    return result;
}
```

As opposed to the `reduction` clause, which is explicitly defined to use private copies for each thread in the reduction domain, the new task-based reduction clauses are defined in more vague terms, specifying only

what the result should be after the reduction domain, and therefore allowing a greater degree of freedom to the implementations that comply with the standard.

3.2.2 Intel *Cilk Plus* reducers

Cilk Plus is an extension to the C and C++ languages that provides extra support to data and task parallelism.[18]. *Cilk Plus* can be differentiated from programming models like *OpenMP* or *OmpSs* in that *Cilk Plus* does not use *pragma* compiler directives but directly extends the language instead, by adding new keywords.

The base concept of the *Cilk Plus* extensions is the *strand*: A *strand* is a sequence of instructions that starts or ends on a statement which will change the parallelism. A strand is delimited by one of the following *Cilk Plus* keywords: `cilk_spawn`, `cilk_sync` (including the implied `cilk_sync` at the end of a function), or `cilk_for`.[20]

An example of parallelization with *Cilk Plus* can be seen in the code listing 7, where the n-th position in the *Fibbonacci* sequence is computed.

Listing 7: *Cilk Plus* example

```
int fib (int n)
{
    if (n < 2) return n;
    else
    {
        int x, y;

        x = cilk_spawn fib (n - 1);
        y = cilk_spawn fib (n - 2);

        cilk_sync;
        return (x + y);
    }
}
```

Cilk Plus describes a special mechanism for reductions called the *Cilk Plus reducers*. The *reducers* are special variables of the *Cilk Plus* framework that have the following properties:[13,19]

- Each strand has a private view of the reducer, so we do not need to use mutual exclusive regions to serialize access to the reducer. The views are combined by the *Cilk Plus* runtime by calling the `reduce()` function of the reducer's operation and type when views `sync`.
- The `reduce()` function is called so that the strands are combined in the order that would have occurred if the program were run with one worker.

In the code listing 8 we can see an implementation of the aforementioned dot product reduction by using the *Cilk Plus* extensions.

Listing 8: *Cilk Plus* reduction example

```
float dot_product(const float *A, const float *B, unsigned int length)
{
    cilk::reducer< cilk::op_add<float> > result(0);

    cilk_for (unsigned int i = 0; i < length; i++)
    {
        *result += A[i]*B[i];
    }

    return result.get_value();
}
```

4 Methodology

Before starting the development of the described objectives right away, it is best to first describe the appropriate methods which help us dealing with the project work and achieving those objectives.

4.1 Work method

The project work is divided in analysis, development and testing stages:

First of all, we will perform an in-depth analysis of the reduction operation and how it should be integrated into the current *OmpSs* programming model from a theoretical point of view.

Then, we will be required to understand the existing projects' environment in detail, becoming acquainted not only with the components that form them but also the software architecture of the projects themselves.

After having a complete knowledge of their current situation, the development of the first implementation stages of our project may begin. Both the *Mercurium* compiler and the *Nanos6* runtime should be extended concurrently so they are found in the same development stage at all times.

The well-known *Agile* iterative development approach *eXtreme Programming* with one-week long iterations will be used to ensure the project advances towards the proper direction.

4.2 Tracking tools

In order to ensure that the project is on the right track at all times and that the work is complete within the scheduled time frame, project tracking methods are required.

On one hand, weekly meetings with the directors will be a good opportunity to assess the project status. On the other hand, an exhaustive tracking of all processes and the time invested in each will allow comparing them to the estimated times as described in the schedule, thus being able to react as soon as possible in case of deviating from the original planning.

Git[14] is the tool chosen for the project tracking. It allows not only to track changes in a fine granularity but also to document them in an organised fashion.

4.3 Validation method

It is important to provide an objective validation method that checks whether the developed software is working as expected and generates correct results. For this matter, automated testing techniques may be appropriate.

Exhaustive sets of tests will be developed for both the compiler and the runtime projects in order to test each added feature. Tests will not be limited to common use cases and will try to evaluate corner cases as well. The current test base for those projects it is also to be considered when it comes to verification, as it guarantees that existent features still work after the new changes.

These testing mechanisms can also be automated so that they run on a set of different systems on every new version of the software. This can help us evaluate the software for different computer architectures without extra effort, providing extensive testing.

5 Planning

The amount of work that this project supposes is quantified to be at least 15 European Credit Transfer and Accumulation System (ECTS) credits for being a Bachelor's Thesis, as defined by the Bachelor's Thesis regulation document approved by *FIB*.[1] For each ECTS credit, the student is expected to devote it from 25 to 30 hours, summing up to a total development time ranging from 375 to 450 hours for the whole project.

To this amount, we still need to add the work that corresponds to the Project Management Course (GEP). This course takes place at the beginning of the semester, during the first stage of the project, and the work it supposes is quantified as 3 more ECTS credits. Accounting the GEP, the total workload ranges from 450 to 540 hours.

The starting date of the project corresponds to the first day of the semester, **September 12th, 2016**. The deadline to deliver the Bachelor's thesis report is the **17th of January** next year.

The temporal planning of the project is detailed in this section. To have a fine-grain control over the work and the time spent on it, the project work is decomposed into specific tasks and dependences between those.

5.1 Task specification

This section gives a concise but clear description of each task and what resources are required for it to be carried on.

Moreover, consider the laptop computer, the text editor and the LaTeXcompiler separately, as those are basic resources required for every single task and there is no point in repeating them for each task meaninglessly.

5.1.1 Project management

These first tasks are about the management of the project itself. A proper analysis of this project, how it will be organized and a viability study are to be done.

Those tasks correspond to the GEP. The detailed steps to follow are listed below:

1. Context and scope of the project
2. Planning
3. Budget and sustainability
4. Preliminary oral presentation
5. Condition specification
6. Final document and oral presentation

Aside from the aforementioned basic resources, a video camera will be required to record the preliminary oral presentation.

5.1.2 Study and familiarization with the model

It is important to have a complete knowledge of the current project before starting the implementation of a new feature that runs on it. Understanding the *OmpSs* model as a whole and the *Mercurium* and *Nanos6* projects that compose it gives us a broader view that will lead to a design of the new feature that fits with the project.

Reading of the model specification, detailed technical documents and effective learning with by programming and analysing test programs will be key actions in this task.

5.1.3 Analysis and design

Before considering the implementation details, it is a good idea to have a purely theoretical analysis of the reduction scheme and the possible designs considering a task paradigm.

A thoughtful analysis of how this new feature should be integrated into the current model is necessary to ensure its utility and avoid future implementation problems.

Then, a complete design of how the feature will behave and what use-cases should be supported will mandate the upcoming implementation.

This task will also be responsible for giving a detailed description of which particular parts of the software need to be extended to support the new feature.

5.1.4 Development

In short, the purpose of this group of tasks involves the specific implementation of the previous design, testing, and evaluation of the obtained results.

Regarding the required resources for the development tasks, the *MareNostrum III* supercomputer as well as some additional clusters yet to be determined will be used for testing and performance analysis purposes.

The details for the involved tasks are listed as follows:

Preparation of the work environment

Spending time in the proper preparation of the work environment is essential to for productive development phase. Downloading, compiling, installing and setting up the necessary tools will be done in this task.

In particular, it is important to set up the working environment for the *MareNostrum III* supercomputer and any other machine to be used. Additionally, this includes setting up the remote connection system (Secure SHell (SSH), Virtual Private Network (VPN), etc.) as well as learning how to use the execution queuing system in use.

Getting to know the specific configuration flags and setting up a remote *Git* repository to track the development is also to be done.

Extending *Mercurium* to support task reductions for point-based dependence model

Implementation of the task-reduction feature in the *Mercurium* compiler, considering only point dependences.

Whilst this implementation considers only the simplest dependence model, the developer will have to deal with every other aspect of the implementation. Future tasks will be using most of this work, while focusing only on extended dependence models.

Extending *Nanos6* to support task reductions for point-based dependence model

Implementation of the task-reduction feature in the *Nanos6* runtime library, considering only point dependences.

The reasoning about the dependence models in the previous tasks also applies for the runtime library.

Extending *Mercurium* to support task reductions for region-based dependence model

Implementation of the necessary modifications in order to support region-based dependences in the *Mercurium* compiler, where a possibly non-contiguous set of addresses is to be considered as a dependence.

Extending *Nanos6* to support task reductions for region-based dependence model

Implementation of the necessary modifications in order to support region-based dependences in the *Nanos6* runtime library, where a possibly non-contiguous set of addresses is to be considered as a dependence.

Result documentation, comparison and conclusions

When the implementation is complete its time to test the new feature, evaluate the results and reason about them.

In short, we are interested in testing how the feature performs for different problems and in different setups, while comparing it to other reference implementations. For more details in the validation method please refer to the methodology section of this project. 4.3

5.1.5 Final stage and defense preparation

While every task described up to the point considers its applicable documentation, some time needs to be done to finalise the documentation as a whole.

Moreover, a reasonable amount of time is needed when it comes to the preparation of the presentation support material and the defense itself.

5.2 Task duration estimation and assigned roles

Table 1 shows an estimation of the duration of the described tasks and the part it supposes to the total. The table also shows which role is assigned to that particular task.

Task	Duration (part)	Duration (time)	Assigned role
Project management	14%	75h	Project manager
Study and familiarization	4%	25h	Analyst, programmer
Analysis and design	18%	100h	Analyst
Environment preparation	2%	10h	Programmer
Extending *OmpSs*	44%	240h	Analyst, programmer, tester
Result evaluation	11%	60h	Analyst, programmer, tester
Final stage and defense	7%	40h	Project manager
Total	100%	550h	-

Table 1: Task duration estimation and roles

5.3 Dependences between tasks

Figure 5 shows a *Gantt* diagram where explicit dependences and ordering between tasks are shown in a clear way.

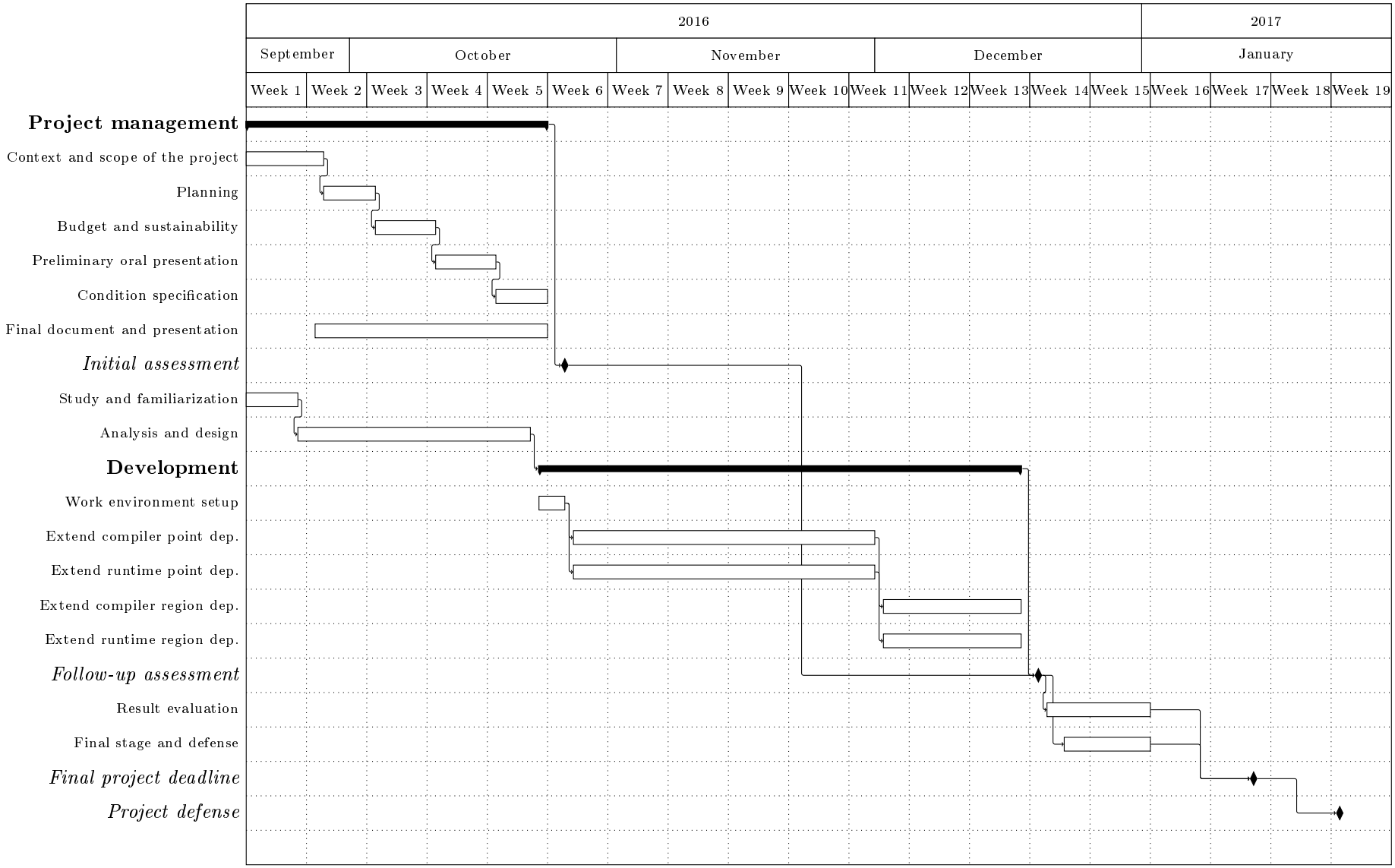


Figure 5: Gantt chart, showing dependences and ordering between tasks

5.4 Deviations from planning and action plan

Even if a planning is accurate and well-done, deviations can still happen. The reason of this is the uncertainty of the future. One can make predictions about how much time expects a particular task to take or try to foresee any complication, but still will not be assured until that point.

Taking this into consideration, a reasonable margin of time of two weeks is left in the planning in case any task takes longer than was originally expected. Even with this margin, it may be the case that a deviation is big enough to require even more time. Then, the developer will need to increase its workday in order to cope with the increased workload.

Finally, in the undesirable case where the planning is considerably mistaken, resulting in a major deviation, drastic actions will need to be taken. In particular, the planning will be discussed with the project directors, deciding which tasks are preferential for the project and which can be put aside given the situation.

When it comes to deviations, having a short reaction time is as important as the subsequent decision, if not more. In this matter, the short development cycles and the weekly meetings policies described in the methodology section 4 can help detecting and correcting the deviation as at the earliest opportunity.

At last, it is convenient to try to anticipate any deviations by having the possible obstacles in mind. As detailed in the obstacles section 2.3 the development phase is the most prone to encounter obstacles, like unnoticed errors unavailability of testing machines to verify whether the implementation is correct.

5.5 Resources

In this section, we describe the necessary resources to carry on the tasks specified in the previous section. Economic resources are treated separately in the budget section 6, describing the project costs in detail.

5.5.1 Hardware resources

Following we can find the necessary hardware for this project.

- **Laptop computers:** Main development tool. Two laptop computers will be used for the development. On the one hand, a *Dell Latitude E7450* provided by the *BSC-CNS*. On the other hand, a *Dell XPS13 9343* that belongs to the developer of the project.
- ***MareNostrum III* supercomputer:** To be used for testing and performance assessment, operated remotely using the laptop.
- **Additional clusters:** Set of machines with different hardware architectures yet to be determined depending on availability. To be used for testing and performance assessment for those particular architectures being operated remotely using the laptop.
- **Video camera:** Either the laptop integrated web-cam, mobile phone camera or conventional camera can be used.

5.5.2 Software resources

In addition to hardware resources, some software the following software resources are necessary to accomplish the described tasks. They are listed as follows. Note how all of them are open-source software.

- **Arch Linux:** Operating system installed in the personal laptop computer.
- **Autotools:** Automated build system.
- **Debian Linux:** Operating system installed in the laptop computer provided by the *BSC-CNS*.
- **GNU Compiler Collection (GCC):** GNU compilers for C, C++ and Fortran among others.

- **Git:** Robust yet efficient control version system.
- **Gitlab:** *Git* repository manager.
- **Gprof:** GNU profiling tool.
- **LaTeXcompiler:** Text composition system widely used for scientific reports.
- ***Mercurium* compiler:** *OmpSs* reference compiler for C, C++ and Fortran.
- **OpenSSH SSH client:** Required to remotely connect to the *MareNostrum III* supercomputer and the other computing clusters.
- **Oprofile:** Profiling tool for Linux that supports reading hardware counters.
- **Vim editor:** Simple, highly-configurable, command-line editor.

5.5.3 Human resources

Human resources have the most important role in the development of the project.

The author and main developer of the project stand out for investing the most time, he is also responsible for the proper functioning of the feature, as he will test it.

It is also very important to consider the time invested by the directors and support research engineers in supervising the project and guiding the developer to achieve the objectives.

6 Budget

In this section, we describe the economic needs of the project and a detailed budget is presented. Meeting the budget is essential to guarantee the project's economic viability.

6.1 Costs specification

According to *Boletín Oficial del Estado* (BOE)[5], the expected annual working time is 1.826 effective hours. Considering this, the following formulas can be used to describe the required calculation to compute the amortization of the resources imputed to this project.

$$\text{cost/hour}_{\text{effective}}(€/\text{hour}) = \frac{\text{acquisition price } (€)}{\text{useful life (years)}} \times \frac{\text{1 annual working time}}{\text{1.826 hours}} \tag{1}$$

$$\text{imputed cost}(€) = \text{units} \times \text{usage (hours)} \times \text{cost/hour}_{\text{effective}} \tag{2}$$

Where *usage* is calculated according to the duration of the tasks on which the resource is used, as seen in the planning section 5.

6.1.1 Software costs

Costs involving the acquisition and usage of the required software resources to carry the project out. All software resources used in this project are released under an Open-Source license, i.e. no software costs are to be imputed to this project.

6.1.2 Hardware costs

Costs attributed to the hardware resources necessary for the project. In order to calculate these costs accurately, it is important to consider their amortization period. In Spain, the tax office states that the maximum amortization period for a computer system is up to eight years[28]. Nevertheless, considering the nature of the HPC field, the hardware amortization period can be no more than four years. After this period, the hardware is considered obsolete.

Table 2 shows the breakdown of the hardware-related costs.

Resource	**Units**	**Cost/unit (€)**	**Useful life (years)**	**Usage (hours)**	**Imputed cost (€)**
Dell XPS13 9343 laptop	1	1.469	4	140.0	28
Dell Latitude E7450 laptop	1	2.529	4	400.0	139
Video camera	1	0*	4	1.5	0
MareNostrum III supercomputer	1	-*	-	20.0	-*
Additional clusters	-	-*	-	(each) 20.0	-*
Total	-	**3.998**	-	-	**167**

Table 2: Hardware costs estimation

When computing the hardware costs it is important to note some special cases (marked with an asterisk symbol in the table).

In the first place, the imputed cost of the *MareNostrum III* supercomputer could not be calculated. The reason for this being the lack of information regarding its complete cost, maintenance cost, useful life and amortization period.

On the other hand, if we use the laptop integrated web-cam as a video camera, no extra cost is to be recorded.

As a side note, the cost of the *Dell Latitude E7450* laptop is fully covered by the *BSC-CNS*.

6.1.3 Human resources costs

The project will be carried out as a whole by one developer, who will have four different roles based on the task to execute: Project manager, analyst, programmer and tester.

In table 3 we can see the global cost attributed to each of those roles, based on the corresponding salary and the dedication. In turn, the dedicated time is calculated according to the duration of the tasks where the role takes part, as seen in the task description section. 5.1

Role	**Salary (€/hour)**	**Dedication (hours)**	**Cost (€)**
Project manager	50	115.0	5.750
Analyst	35	212.5	7.438
Programmer	30	122.5	3.675
Tester	20	100.0	2.000
Total	-	**550.0**	**18.863**

Table 3: Human resources costs estimation

6.1.4 Indirect costs

In addition to the costs related to the explicitly required resources specified up to this point, we need to consider other costs that affect to the project budget in an indirect way.

Specifically, we should take into account the cost of electricity, internet access and office rent, among other general expenses. However, as the project will be developed in its totality in the *BSC-CNS*, it is not possible for us to specify the cost of these services, being this information private and out of our hands.

6.1.5 Taxes

This project development will take place in the *BSC-CNS*, in Barcelona. For this reason, the project budget will be subject to the Spanish taxing laws. In particular, we need to consider value-added taxes (VAT) and the electricity tax.

Table 4 shows the expenses related to the aforementioned taxes.

Note that some of the taxes could not be calculated as the base cost is unknown, they are shown in the table for the sake of completeness. Above all, note that the tax rate for the electricity tax is unknown as it depends on the specific consumption and power.

Concept	Base cost	Tax rate	Cost(€)
Software	0	21	0
Hardware	167	21	35
Human resources	18.863	21	3.961
Internet access*	-	21	-
Electricity (VAT)*	-	21	-
Electricity tax*	-	-	-
Total			**3.996**

Table 4: Taxes estimation

6.1.6 Summary

To sum up, the project costs have been gathered in table 5.

Concept	Cost(€)
Software	0
Hardware	167
Human resources	18.863
Indirect costs	-
Taxes	3.996
Total	**23.026**

Table 5: Project costs summary

Due to the environment where this project takes place it has been impossible to determine some of the costs that rely on private information from the *BSC-CNS*. For this reason, the total sum presented should be taken as a lower bound of the project cost and not a reliable estimation.

6.2 Deviations from budget

Deviations from budget will mostly be related to one or more risks listed in the planning section of this document. 5.4

In this case, any deviation from the budget will be caused by an increment in the accounted dedicated hours for a given task. In particular, we will need to face the overrun caused by the extra fees that correspond to the salary of the roles involved in the deviated task. Moreover, the resources amortization will need to be readjusted in terms of the new use time.

As stated in the planning section, we are not expecting major deviations from the original plan. In case of minor deviations, from an hour to 24 hours work-time, the extra cost can be covered by the deviations budget item created for that matter. This item extends the budget by a 6% and will be used to solve any of the deviations described in this section. With the costs computed in this document, the deviation budget item will consist of 1.381€. This amount should not be spent unless a deviation occurs and it is expected to cover any unforeseen cost.

In the worst-case scenario, a 24 hours deviation corresponding to the project manager role will result in an overrun cost of 1.200€, which can be fully covered by the deviations budget item.

6.3 Expense control

Monitoring of the resources used for each task takes an important part in the expense control. To ensure this is done properly, a weekly report can be written to track down the resources that have been used during that period and the dedicated time. This way, not only the budget can be controlled but also deviations from the project planning can be discovered within a short time.

7 Project sustainability

In this section we evaluate the project sustainability and its compromise with the society. This evaluation considers the project's environmental, economic and social aspects, which are assessed separately.

The evaluation method consists in answering a set of questions related to each of the aspects. Each question grants a positive or negative amount of points which are recorded in the aspect score and are then used for the final sustainability evaluation.

The sustainability matrix 6 shown below shows the obtained score for each aspect as well as the final sustainability score.

Aspect	**Project development**	**Exploitation**	**Risks**
Environmental	Consumption design 5/10	Ecological footprint 18/20	Environmental risks -2/-20
Economic	Project bill 9/10	Viability plan 18/20	Economic risks -5/-20
Social	Personal impact 8/10	Social impact 15/20	Social risks -2/-20
Sustainability range	22/30	51/60 64/90	-9/-60

Table 6: Sustainability matrix

7.1 Economic aspect

Several points in this document support the score for the economic aspect shown in the sustainability analysis.

On the one hand, there is a detailed cost evaluation that includes not only direct costs attributed to necessary material resources but also human resources, as well as indirect costs. Then, contingency and unexpected costs are also considered in the project budget, ensuring that the project is economically viable in any case.

On the other hand, the resources described in this document are minimal and well-justified for carrying the project out: these are basically development and testing tools, as well as the human resources in charge of the development itself.

Being a research project, it is important to keep in mind that its economic viability does not require developing a project competitive in a commercial sense. In addition, the feature developed in this project will be included in the *OmpSs* programming model, which is distributed openly and free of cost to everyone.

7.2 Social aspect

In the first place, this project has a great implication in a personal level. Not only is this the first project of this magnitude the developer carries out but it also differs from previous projects in terms of how important its proper organization is. The experience is both enriching and challenging.

Although one may think there is not direct social compromise involved in this project, it can be shown how it can benefit the society both directly but, above all, indirectly.

For the user of the *OmpSs* programming model, it will allow a hassle-free programming of the tedious reduction schemes, increasing the model ease of use while avoiding programming errors and simplifying the

overall code being developed. Additionally resulting in a shorter development time and ensuring an optimized execution. Eventually, this will make the model user not only more efficient and focused on its real job but also happier, which is the objective we are after.

Regarding indirect benefits to society, one can think of mostly anything. For example, consider an application that takes advantage of the features offered by the *OmpSs* programming model. With an optimized implementation of the reductions scheme, the application will reduce its execution time, providing the result to the final user quicker while consuming fewer resources to do the job. The user does not need to know about the existence of this project to benefit from it.

Risks originated from a malfunction in the developed feature are not a major issue, the user can always decide not to use this feature and continue using its implementation of the reduction scheme.

7.3 Environmental aspect

The project in question is almost completely environment-friendly. As far as the final product is concerned, being a software application means that no product will be manufactured and, therefore, no natural resources will be consumed in that matter. Moreover, any optimization applied during the development of the feature will reduce the computing time and maximise the parallel usage of computer resources, eventually minimizing the resource consumption in time and the electricity bill.

This being said, the only environmental impact can be caused during the development stages of the project. Even if this is the case, caution has been taken to account for all the required resources and any possible deviation in their usage. Furthermore, as seen in the resources description section, little hardware will be used in this project.

As a particular case, the *MareNostrum III* supercomputer is a very powerful machine and has a significant environmental impact as a result of its building process and electrical consumption. However, we will be only using some of its nodes and for a short period of time, resulting in an almost negligible environmental impact attributable to this project.

All in all, the project has definitely a small ecological footprint.

8 Project base

This project is based in the existing *OmpSs* infrastructure presented in section 1.1 that will be extended during the development phase. In this section we aim to give a more detailed description of the base project components internal functioning required to be able to understand how our work applies on top.

8.1 *Mercurium* compiler internals

The *Mercurium* compiler design is characterized by a set of successive phases that create a tree structure and subsequently enrich it by adding information onto its nodes as well as creating new ones.

For the processing of *OmpSs pragmas*, the compiler uses different node types to represent their information in the tree depending on the compiler phase. After the completion of the front end phase, the *OmpSs pragmas* are represented in the tree as generic *pragma* nodes storing their information as text. In this point, the nodes are handed to a generic *OmpSs* phase, where the *pragma* nodes that correspond to *OmpSs pragmas* are analysed and any restrictions imposed by the models are checked. If no problems have been found, the generic *pragma* nodes are replaced by specific *OmpSs* nodes.

After the *OmpSs* phase, the node tree can be passed onto a posterior, *Nanos6*-specific lowering phase which is responsible for elaborating or *lowering* the *OmpSs* nodes into meaningful nodes of the underlying language.

When this process is over, the completed tree can be passed onto the code-generation phase to generate the output C, C++ or Fortran code. Finally, the native compiler can be called to transform this code into an executable binary.

For this project, our extensions in the *Mercurium* compiler will be limited to the *Nanos6* lowering phase.

As mentioned above, in this phase we are going to receive an enriched tree structure with nodes representing *OmpSs pragmas*, and our task will consist in traversing the tree gathering the necessary information in order to be able to replace these *OmpSs pragma* nodes in favour of new nodes and symbolic information corresponding to meaningful statements of the underlying language, according to what our specific purpose is.

Generic utility classes such as visitors are provided in *Mercurium* to ease the job of efficiently traversing the tree structure looking for the nodes we need to treat.

8.2 *Nanos6* runtime library internals

As explained in section 1.1, *OmpSs* is a model completely based in the dataflow. *Nanos6*, being the reference runtime implementation, needs to be fully conscious and efficient in handling dependences between data accesses.

The capability of the *Nanos6* runtime library of effectively recognizing dependences between tasks of different nesting levels comes from its particular dependence management design:

The *Nanos6* runtime keeps track of the dependences between tasks by considering their data accesses separately.

A data access is defined as any access annotated in the task by the existence of any dependence-annotating clauses (such as `in`, `out`, `inout`, etc.). A data access can refer to either a variable or memory position.

In the *Nanos6* runtime, a task is satisfied and ready to be executed only when all its data accesses are satisfied.

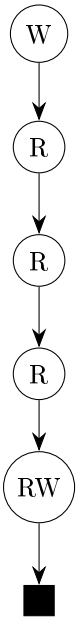
When a new task is created, each of its data accesses is registered in a chain containing all data accesses referring to the same variable or memory position. This structure is called `DataAccessSequence`. When a

task is finished, its data accesses are removed from the corresponding `DataAccessSequence`s. This concept is illustrated by showing a representation of the `DataAccessSequence` generated from the code listing 9 in figure 6.

Listing 9: Example sequence of tasks

```
int a;

#pragma oss task out(a)
{
    a = initialize();
}

for (int i = 0; i < 3; i++)
{
    #pragma oss task in(a)
    {
        display_value(a);
    }
}

#pragma oss task inout(a) label(R)
{
    a++;
}

#pragma oss taskwait
```

Figure 6: `DataAccessSequence` example



Each `DataAccessSequence` is responsible for managing the data accesses to a single variable or memory position, determining when an access is satisfied and executing its originating task or holding it when there is a pending dependence with a previous data access.

Figure 7 shows an example of how a `DataAccessSequence` could evolve over time:

1. In the first time step, there is a satisfied write access ready to be performed. In order to avoid any data hazards, the tasks that originate the other accesses are held (not satisfied).
2. In the second step, the write access is over and all read access become satisfied. Note how the satisfiability is propagated between accesses of type read as they can be performed concurrently over the same variable or memory position. The last (read-write) access remains unsatisfied.
3. In the third step, two of the three read accesses are done. Note how a dependence between the remaining read access and the subsequent read-write access is created.
4. In the last step, all read accesses are finished and the satisfiability is propagated towards the read-write access.

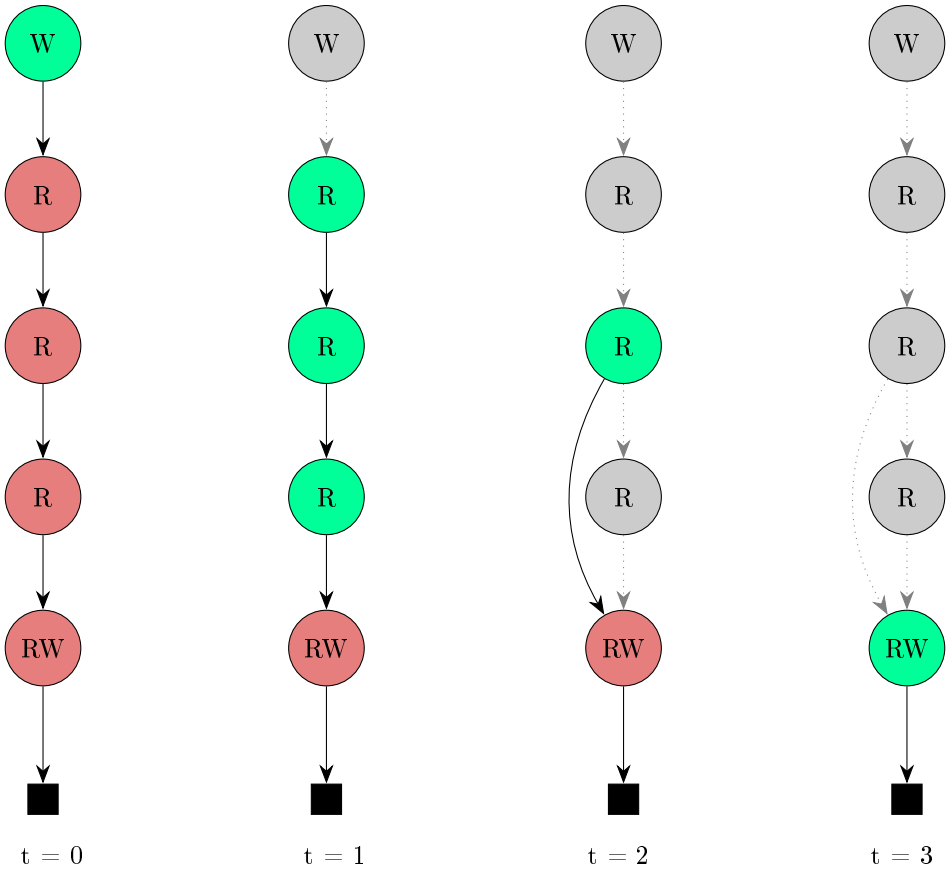


Figure 7: Data access sequence example

When a task creates nested tasks that have data accesses referring the same items, those data accesses are placed in a `DataAccessSequence` contained within the parent data access. This way, when the parent task has finished, those data accesses can be rose one level and merged into the `DataAccessSequence` where the parent data access was found.

Figure 8 shows an example of this procedure, representing the accesses originated by nested tasks in the parent access' private `DataAccessSequence`. Note that the steps presented here correspond to an arbitrary execution ordering chosen to illustrate this interesting scenario, the ordering could change from an execution to another:

1. In the first time step, there is a satisfied write access ready to be performed. The other accesses are held (not satisfied).
2. In the second step, the write access is over and all read access become satisfied. Note how the satisfiability is also propagated towards inner nesting levels. The last (read-write) access remains unsatisfied.
3. In the third step, the nested task originating the last read access has finished its execution and the access is deleted.
4. In the last step, the parent task originating the read access has finished, the access is deleted and the accesses in its private `DataAccessSequence` are merged up at the position where the access was, creating an explicit dependence with the previous and posterior accesses.

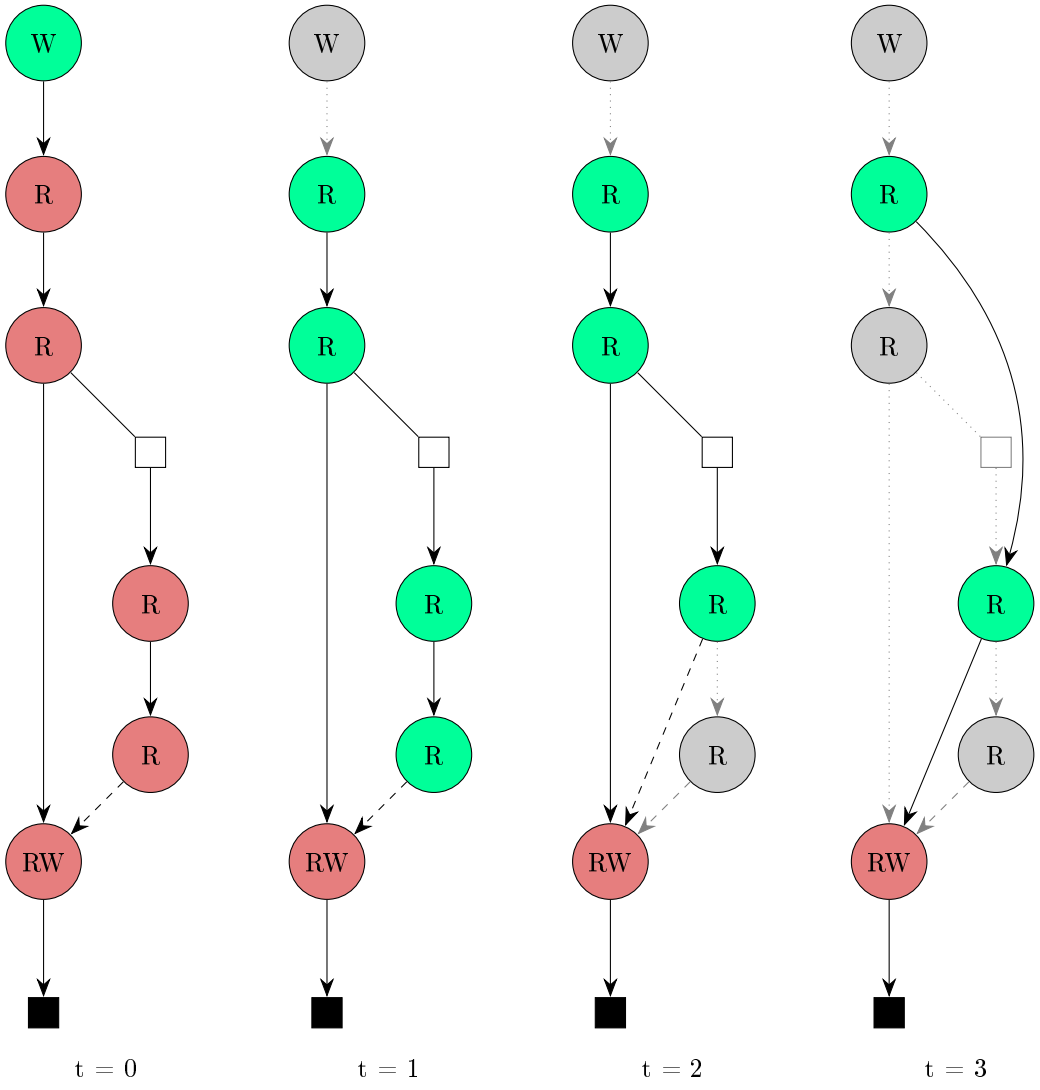


Figure 8: Nested data access sequences

9 Towards supporting reductions in *OmpSs*

In the following sections we aim to guide the reader through the different stages undergone during the development of this project.

The first step in the process was the development of the `concurrent` clause, which can be used to write user programs that compute reductions. This is the most low-level approach, where the user has to handle the reduction process.

The second step is about the task privatization approach for the development of the `reduction` clause. This approach supersedes the use of the `concurrent` clause for reductions in all situations by abstracting the process and adding functionality without compromising the performance in any case.

Finally, an alternative approach that includes privatization per CPU with support from the *Nanos6* runtime library is presented.

9.1 Implementing the `concurrent` clause in the *OmpSs* infrastructure

In this first section we are going to give a detailed explanation of the design and implementation of the `concurrent` clause. First, we are going to introduce the `concurrent` clause, showing its usage and how the *OmpSs* model defines it. Then, describe the decisions and considerations taken to design the feature in a generic manner and finally we will reveal the corresponding implementation details of the *OmpSs* infrastructure including the *Mercurium* compiler and the *Nanos6* runtime support library.

9.1.1 Introduction to the `concurrent` clause in *OmpSs*

The aim of the `concurrent` clause is to provide a new data access type for tasks that allows a greater freedom on the scheduling of their execution. In particular, the idea is to allow all tasks sharing a concurrent access on the same variable to run concurrently and in any order once the previous non-concurrent accesses are satisfied, being the user's responsibility to protect any write access to the accessed variable.

To clarify, the generic usage of `concurrent` accesses can be seen as having a *cloud* of tasks that have dependences with previous and posterior accesses but having no dependences between accesses in the cloud itself, as illustrated in figure 9.

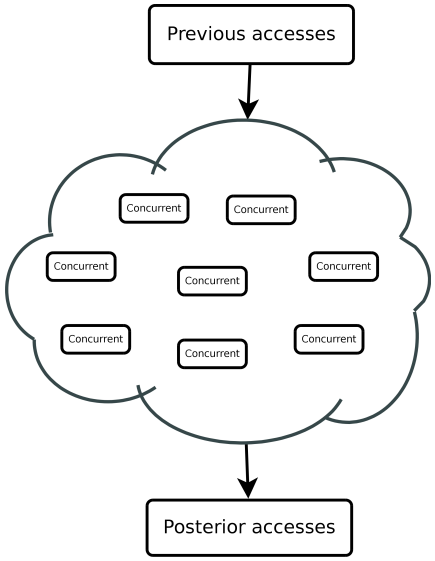


Figure 9: Concurrent accesses seen as a cloud

The `concurrent` data access type is very similar to the `in` data access type in the sense that, when the first access becomes satisfied, the consecutive accesses of the type also become satisfied. That being said, the justifications of implementing a new access type are clear:

On the one hand, there is an intrinsic semantic difference between an `in` access type and a `concurrent` access type. An `in` access type is only valid for reading the data but not modifying while the `concurrent` clause allows reading and modifying the data indistinctly. Using the `in` clause for both cases would be semantically incorrect.

In addition, separating the `concurrent` clause from the `in` clause gives us an extra degree of freedom by being able to specify a dependence between a group of reading tasks (annotated with `in`) and a group of concurrently writing tasks (annotated with `concurrent`). If both clauses were unified, a task synchronization point would be required between the groups. The only way to avoid the performance implications of having the task synchronization point in such situation is to implement the `concurrent` data access and describe the interaction (dependences) between it and the other access types.

For the time being, the behaviour of the `concurrent` accesses when either preceding or following an access of any type other than `concurrent` itself is to block the posterior access until the task originating the anterior access is over.

The design is open to allow different interactions with any new data access types to be added into the *OmpSs* model.

As the concurrent clause is provided to annotate concurrent accesses, its syntax is determined by the set of clauses used to annotate data accesses like the clauses *in*, *out* or *inout*. The specific syntax of the clause is as follows:

```
#pragma oss task concurrent(list)
```

Where *list* is the list of items which will be accessed by the task under the defined concurrent access terms.

In the code listing 10 we can see an example of how the concurrent clause could be used in a real program. The code defines two distinct concurrent regions where tasks that can be executed concurrently: The first region is composed of the tasks C1 and C2, while the second is composed of the tasks C3 and C4. The resulting task dependence graph derived from the code can be seen in figure 10.

Listing 10: Concurrent access example

```
int var;

#pragma oss task out(var) label(W)
var = initialize();

#pragma oss task concurrent(var) label(C1)
atomic_add(var, compute_value());

#pragma oss task concurrent(var) label(C2)
atomic_add(var, compute_value());

#pragma oss task inout(var) label(RW)
var = normalize(var);

#pragma oss task concurrent(var) label(C3)
atomic_product(var, compute_factor());

#pragma oss task concurrent(var) label(C4)
atomic_product(var, compute_factor());

#pragma oss task in(var) label(R1)
printf("Var: %d\n", var);

#pragma oss task in(var) label(R2)
update_database(var);

#pragma oss taskwait
```

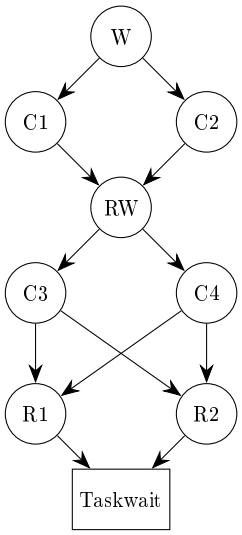


Figure 10: Concurrent access example graph

9.1.1.1 Manual reductions using the `concurrent` clause

A manual implementation of the reduction pattern can be achieved by the user by using the `concurrent` clause and carrying out the synchronization manually. While this is a powerful mechanism to implement reductions, it is important to note that it has clear drawbacks in functionality, as the reduction has to be completely managed by the user.

For this manual implementation, the user will need to ensure that every access to the original variable is protected by means of surrounding it with a synchronization mechanism in order to avoid data races. For a better performance, atomic operations should be used in place of locking synchronization strategies when possible. The code listing 11 shows an example of a manual reduction of N tasks over the `result` variable using the `concurrent` clause.

Listing 11: Manual reduction using `concurrent` clause

```
int result = 0;
for (int i = 0; i < N; ++i)
{
    #pragma oss task concurrent(result)
    {
        mutex_lock();
        result += compute_value();
        mutex_unlock();
            ...
        mutex_lock();
        result += compute_other_value();
        mutex_unlock();
    }
}

#pragma oss taskwait

return result;
```

In addition to this, the user is also capable of developing a more sophisticate reduction strategy using the `concurrent` clause. This strategy consists in declaring a local variable in the beginning of the task, and replacing all appearances of the original variable for it. Right after finishing the meaningful computation of the task related to the reduction, the local variable should be combined into the original variable by means of the desired operator. With this changes, we will only have to protect the single access to the original variable, the combination statement, reducing considerably the amount of synchronization required. This strategy is commonly known as *privatization*, and its application can be seen in the code listing 12

Listing 12: Manual reduction using `concurrent` clause with privatization

```
int result = 0;

for (int i = 0; i < N; ++i)
{
    #pragma oss task concurrent(result)
    {
        int result_local;

        result_local = compute_value();

            ...

        result_local += compute_other_value();

        mutex_lock();
        result += result_local;
        mutex_unlock();
    }
}

#pragma oss taskwait

return result;
```

9.1.2 Implementation of the `concurrent` clause

The design of the `concurrent` clause can be directly derived from the model specification.

A new access type needs to be introduced in the runtime task scheduler so that it fulfils the described behaviour:

- Holding the execution any tasks willing to perform a concurrent access when there are previous non-concurrent accesses pending.
- Allowing the concurrent execution of all tasks willing to perform a concurrent access on the same data once all previous tasks performing non-concurrent accesses are finished.
- Holding the execution of all tasks willing to perform a non-concurrent access when there are previous concurrent accesses pending.

9.1.2.1 Internal details

The simplest valid implementation of the concurrent clause is the pure serialization of the tasks as in an `inout` clause. However, this implementation is also the most (unnecessarily) constraint and, as we want to take advantage of the parallelism the clause offers, we are going to provide an optimized implementation and keep the serialization only as a fall-back implementation.

The modifications in the *Mercurium* compiler to support the new `concurrent` clause are minimal. First, we need to replace the `concurrent` clause with a `shared` clause over the same items, then, we need to place

an API call to the *Nanos6* runtime to register the concurrent access for each of those items. The API calls are placed in the task creation routines.

The extensions required for the *Nanos6* runtime are also deduced from the design: First, the API needs to be extended accordingly to the *Mercurium* compiler calls. Then, a new `concurrent` data access type needs to be introduced in the dependence management system based on the data access sequences, specifying under which conditions a `concurrent` access should be satisfied and under which held (materialize a simple logic from the designed behaviour).

9.2 Implementing the `reduction` clause in the *OmpSs* infrastructure

The `reduction` clause is the fundamental mechanism for computing reductions in *OmpSs*.

Being the main feature to be developed in this project, special care has been taken in the design and implementation phases of the `reduction` clause. Multiple versions have been designed, all having different implementation complexity and different strengths and weaknesses. In the following sections we describe two different approaches and propose an implementation.

9.2.1 Introduction to the `reduction` clause in *OmpSs*

As opposed to *OpenMP*, the *OmpSs* programming model is purely based on tasks. For this reason, any mechanism in *OmpSs* is designed on top of tasks. The `reduction` clause is no exception.

The specific syntax of the clause is as follows:

```
#pragma oss task reduction(reduction-identifier:  list)
```

Even though the similarities between the syntax of the `reduction` clauses in *OmpSs* and *OpenMP* may induce us to think both clauses are equivalent, they are very different in terms of the semantics. In fact, the `reduction` clause in *OmpSs* is much closer in meaning to the `in_reduction` clause than to the `reduction` clause in *OpenMP*.

The `reduction` clause specifies a reduction-identifier and one or more list items. Each list item corresponds to a variable or memory position. From the user point of view, each of those list items can be accessed within the task scope as if it were a private variable initialized to the neutral element. It is important to remark that the model wants to remain open to any implementation that computes the result of the reduction into the original list item, for this reason, the number of copies is unspecified and implementation-dependant.

The reduction domain is defined to be the region starting where the reduction is firstly defined and is extended until the subsequent task synchronization point, being it a dependence or a `taskwait` in the precise nesting level. After the end of the region, the original list item is guaranteed to contain the updated value, computed by applying the operator associated with the reduction-identifier to each private copy, if any.

The code listing 13 shows a real example of how the `reduction` clause can be used. In the example, the dot product of two arrays is computed concurrently. The arrays are broken up into blocks and, for each pair of blocks, an *OmpSs* task is instantiated: Each of those tasks is responsible for computing the dot product of its blocks. The tasks are annotated with a `reduction` clause specifying the `result` reduction variable and the addition (`+`) reduction identifier. Within each task, `result` can be seen as a local copy of the original `result` variable. For each pair of corresponding elements in the blocks, its product is accumulated in the `result` local copy as explicitly stated in the code. Finally, all local copies are combined into the original `result` variable once the `taskwait` is reached.

Listing 13: *OmpSs* reduction clause example

```
float dot_product(const float *A, const float *B, unsigned int length)
{
    float result = 0;

    for (unsigned int i = 0; i < length; i += BLOCK_SIZE)
    {
        unsigned int bound = MIN(i + BLOCK_SIZE, length);

        #pragma oss task firstprivate(i) in(A[i:bound - 1], B[i:bound - 1]) reduction(+:  result)
        {
            for (unsigned int j = i; j < bound; j++)
            {
                result += A[j]*B[j];
            }
        }
    }

    #pragma oss taskwait

    return result;
}
```

Only a `taskwait` or a data dependence on the same nesting level where the reduction was declared should trigger the reduction finalization.

The previous statement is important in the sense that it is allowed by the model to have nested tasks within a reduction task and task synchronization points like `taskwait` s that do not finish the reduction. The code listing 14 shows an example where this happens. In the example we can see how a `reduction` task creates two nested tasks to compute some factors, those tasks are waited for and the local copy `result` is set to the product of those factors. The combination of the `reduction` task with its counterparts is not produced in the nested `taskwait` where the nested tasks are waited for.

Listing 14: `taskwait` within reduction task example

```
#pragma oss task reduction(+:  result)
{
    int a, b;

    #pragma oss task
    a = compute_value(...);

    #pragma oss task
    b = compute_value(...);

    #pragma oss taskwait

    result = a*b;
}

#pragma oss taskwait // At this point we compute the reduction
```

9.2.1.1 Supported operations

This section describes the supported reduction operators and their identifier symbol. Table 7 shows the supported reduction identifiers and their initializer and combiner functions.

Identifier	Initializer	Combiner
+	`oss_out = 0`	`oss_out += oss_in`
*	`oss_out = 1`	`oss_out *= oss_in`
-	`oss_out = 0`	`oss_out += oss_in`
&	`oss_out = ~0`	`oss_out &= oss_in`
|	`oss_out = 0`	`oss_out |= oss_in`
^	`oss_out = 0`	`oss_out ^= oss_in`
&&	`oss_out = 1`	`oss_out = oss_in && oss_out`
||	`oss_out = 0`	`oss_out = oss_in || oss_out`
max	`oss_out` = least representable number in the type	`oss_out = oss_in > oss_out ? oss_in : oss_out`
min	`oss_out` = largest representable number in the type	`oss_out = oss_in < oss_out ? oss_in : oss_out`

Table 7: Supported reduction operators

9.2.2 First implementation of the `reduction` clause: Task privatization strategy

In this very first section, the *task privatization* reduction strategy is exposed.

For this reduction strategy we are looking for a task-level privatization or, in other words, for each instantiation of the reduction task, a different private variable will be used replacing each appearance of the original variable within the task. When the computation of the task is over, the partial result computed in this private copy will be combined into the original variable by means of the combining operator.

9.2.2.1 Internal details

This version is mostly implemented in the compiler, requiring the runtime support library only when extending the support to nested reductions.

First, the *Mercurium* compiler needs to declare a local copy for each list item specified in the `reduction` clause and initialize them to the neutral element of the reduction as shown in the previous table 7.

Then, the compiler needs to parse the task code and replace each appearance of the reduction item for the local copy.

The last step required for the privatization to be completed is placing a combining statement as the last statement of the task code. This statement will combine the local copy into the original variable, accumulating the computation done for the task. This statement is the only point where the task will access the original variable directly and therefore the access needs to be synchronous with any other tasks trying to achieve the same.

While placing the access in a mutual exclusion region is a valid synchronization mechanism, the project is centered in scalars and using atomic operations to perform the combination will always lead to a better performance if supported by the underlining architecture.

Finally, for each list item item specified in the reduction we register a dependence of the `concurrent` type by the corresponding API call.

By automatizing the described compile-time transformations, all reductions that do not require nesting should be supported.

The code listings 15 16 show an example code and how the code could be transformed after having applied the task privatization transformations.

Listing 15: Task privatization: original code

```
...

unsigned int bound = MIN(i + BLOCK_SIZE, length);

#pragma oss task in(A[i:bound-1], B[i:bound-1])
reduction(+: result)
{
    for (unsigned int j = i; j < bound; j++)
    {
        result += A[j]*B[j];
    }
}
```

Listing 16: Task privatization: transformed code

```
...

unsigned int bound = MIN(i + BLOCK_SIZE, length);

#pragma oss task in(A[i:bound-1], B[i:bound-1])
concurrent(result)
{
    int result_local = 0L;
    for (unsigned int j = i; j < bound; j++)
    {
        result_local += A[j]*B[j];
    }

    atomic_add(result, result_local);
}
```

Support for nested reductions

With only the transformations stated up to this moment, not all nested reduction patterns are directly supported. Let us first consider the supported code listing 17 and its transformation 18 as explained up to this point.

Listing 17: Nesting in task privatization: original code

```
#pragma oss task reduction(+:  result)
{
    #pragma oss task reduction(+:  result)
    {
        result++;
    }

    result++;

    #pragma oss taskwait
}
```

Listing 18: Nesting in task privatization: transformed code

```
#pragma oss task concurrent(result)
{
    int result_local_1 = 0L;

    #pragma oss task concurrent(result_local_1)
    {
        int result_local_0 = 0L;
        result_local_0++;

        atomic_add(result_local_1, result_local_0);
    }

    result_local_1++;

    #pragma oss taskwait

    atomic_add(result, result_local_1);
}
```

We can see how the task privatization has been systematically applied to each of the tasks, from innermost to outermost (note the `_0` and `_1` suffixes). If we analyse the result in more detail, we can see how the transformation has introduced a data race between the parent and the nested task over the `result_local_1` variable: only the child is accessing it atomically.

This behaviour is unacceptable, but even if this scenario was not supported by the model, problems would arise when removing the `taskwait` construct placed before the outer task performs its combination. If the outer task happened to finish before than its child, the child would be accessing the finished parent's stack, resulting not only in an invalid memory access but also losing the child contribution in the reduction.

In order to overcome this problem, we need to create a mechanism in the *Nanos6* runtime that allows us to know when a reduction is registered over a local copy and what original variable corresponds to that copy

so that the task accesses can be properly updated: A map containing the relation between those objects in every task should be enough.

In fact, a slightly different approach was used in the development of the *Nanos6* runtime to avoid introducing overhead to every task with this new structure. For our implementation, the declaration of the local copies was moved from the beginning of the task into the `args_block`, the structure used to maintain the task execution arguments. In detail, each local copy is placed just next to its corresponding original variable pointer in the `args_block`. After this modification, when a task creates a nested task that defines a reduction over the same variable, proceeds to register its data accesses as usual and calls an additional *Nanos6* API function.

This function is responsible for checking if the reduction variable is pointing an address compressed within the parent's `args_block` (meaning it will be pointing a local copy) and, in this case, update it into the original variable by using pointer arithmetics.

The listing 19 aims to clarify this process, which happens entirely in the *Nanos6* runtime.

Listing 19: Privatization moved into the `args_block`

```
// args_block structure of the parent task
struct nanos_task_args_parent
{
    int x_local; // local copy moved into the args_block
    int *x; // points to the original x variable
};

// args_block structure of the child task
struct nanos_task_args_child
{
    int *x; // originally pointing &task_parent->x_local, to be updated to *task_parent->x
};
```

9.2.3 Second implementation of the `reduction` clause: CPU-and-task privatization strategy

Another version considered during the design stage intended to continue decreasing the thread synchronization cost within the reduction by taking a step further in the privatization process. After having privatized the reduction variable in the task scope, as seen in the previous design, the next level was to privatize the variable for each CPU participating in the reduction.

The idea is to have a private space for each reduction and CPU for it to store its contribution to that reduction, being able to accumulate the values computed on every task executed by that CPU without requiring any synchronization mechanism (each CPU will access a different memory position).

Be aware that, in this version, we will still be applying the task privatization as shown in section 9.2.2. The main difference with that implementation is that, instead of accumulating the local copy into the original variable at the end of the task, we are accumulating in into another private copy per CPU, avoiding any kind of synchronization.

We have no way to implement this mechanism with the information known at compile time and hence we need to implement it inside the runtime support library. In detail, we will at least need a mechanism to register and maintain information related to every reduction into the runtime and a way to retrieve the internal private copy memory address for a given CPU and reduction in order to be able to accumulate the task contribution into it.

It is important to remark that, for scalars, the CPU-and-task privatization strategy can not possibly outperform the previous task-privatization strategy in a platform where the combination of the private copy per task is done via atomic operations. The reasoning behind this claim is that, if the amount of reduction tasks on the same element were so big that collisions in the combination statement were significant, the mutual exclusion regions found within the *Nanos6* runtime for accessing the internal structures and scheduling

queues would already suffer from such contention in both scenarios that the cost of the atomic operations would not even be noticeable.

That being said, the CPU-and-task privatization strategy would then be a preferable alternative if the architecture did not provide native support for atomic operations and mutual exclusive regions were required. In addition, if a reduction were to be combined into a variable for which a previous long computation has to be done, this strategy could be optimized to schedule the computation of the reduction instantly, without waiting for the previous value to be computed. Then, as soon as the previous data had finished its computation, the combination of the reduction copies per CPU would take place. This scenario is shown in the simulated traces of figures 11 and 12, where the red chuck represents the previous data computation, the orange part represents the computation of the reduction tasks and finally the yellow region corresponds to the combination of the reduction.

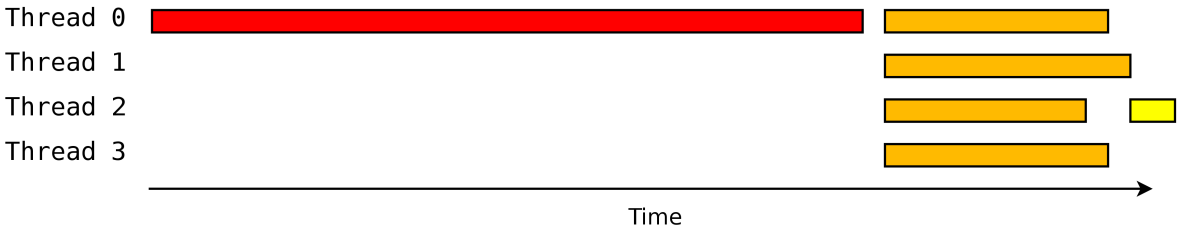


Figure 11: Early beginning optimization: task-privatization

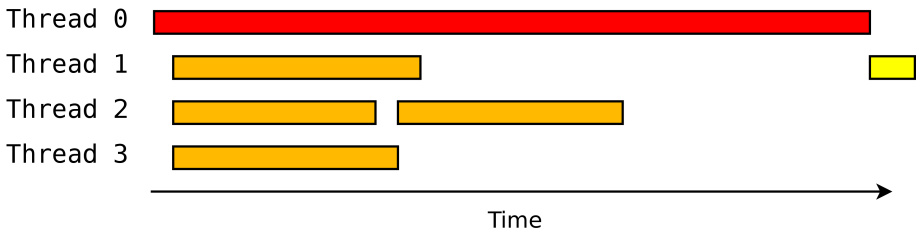


Figure 12: Early beginning optimization: task-and-cpu privatization

Finally, the CPU-and-task privatization strategy would also be favourable when extending the implementations for array regions or structs (via User-defined Reduction (UDR)), where no atomic operations to combine the whole object are possible.

9.2.3.1 Internal details

Despite being based on it, the implementation of the *Mercurium* compiler differs from the previous task privatization implementation in some aspects.

First, instead of registering the `reduction` access as a `concurrent` access, a new API call should be used to register the reduction within the task, specifying the address of the initialization and combination functions to be called from inside the runtime. It is the responsibility of the compiler to generate this functions based on the reduction identifier as seen in the table shown in the section 9.2.1.1. The specific prototype of those functions is the following:

```
void nanos_reduction_initialize(void *element)
void nanos_reduction_combine(void *accumulator, void *element)
```

Finally, in the last step of the user code transformation, instead of placing a combination statement accumulating into the original variable, it should be replaced to accumulate into the CPU copy, in conjunction with

placing an API call before it to obtain its address. This statement does not require synchronization.

The code listings 20 21 show an example code and how the code could be transformed after having applied the CPU-and-task privatization compiler transformations.

Listing 20: CPU-and-task privatization: original code

```
...

unsigned int bound = MIN(i + BLOCK_SIZE, length);

#pragma oss task in(A[i:bound-1], B[i:bound-1])
reduction(+: result)
{
    for (unsigned int j = i; j < bound; j++)
    {
        result += A[j]*B[j];
    }
}
```

Listing 21: CPU-and-task privatization: transformed code

```
void nanos_red_sum_init(double *oss_out)
{
    *oss_out = 0;
}
void nanos_red_sum_comb(double *oss_out, double *oss_in)
{
    *oss_out += *oss_in;
}

...

nanos_register_task_dependences(...)
{
    nanos_register_reduction_access(...,
        nanos_red_sum_init, nanos_red_sum_comb);
}

unsigned int bound = MIN(i + BLOCK_SIZE, length);

nanos_unpack_task(int *result_local, int *result)
{
    // (recall local variable is moved to args_block)

    for (unsigned int j = i; j < bound; j++)
    {
        result_local += A[j]*B[j];
    }

    int *cpu_storage;
    nanos_get_reduction_storage(
        (void*)&result, (void**)&cpu_storage);
    *cpu_storage += result_local;
}
```

As far as the runtime support is concerned, the following sections show how the management of the ongoing reductions has been implemented in the *Nanos6* runtime.

Memory management

First, during the initialization of the runtime itself, a fixed-size memory block is allocated for each CPU to be used to hold its private copies that store the contributions to every running reduction. The total allocated space can be calculated as shown in the equation 3.

$$\text{total allocated space} = \#\text{reduction slots} \cdot \#\text{CPUs} \cdot \text{element size} \tag{3}$$

Here, a reduction slot identifies a system-wide reduction. Figure 13 shows how the reduction slots memory space is organized.

Note how indexing the memory space by reduction slot first and then by CPU allows not having to add padding in order to avoid the false-sharing performance-degrading pattern.

Moreover, if we set the product $\#reductionslots \cdot elementsize$ to be exactly the system page size (4KB in most systems), every CPU will have its private copies in a separate memory page and the Non-Uniform Memory Access (NUMA) effect should be minimized.

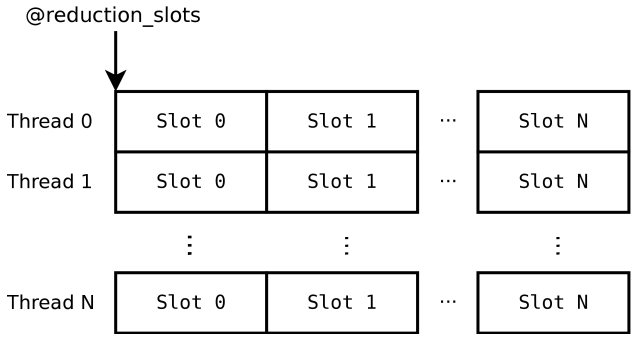


Figure 13: Reduction slots memory space

Every time a new task with a `reduction` clause is registered in the system, the runtime checks whether it corresponds to an already registered reduction or to a new one instead. If it corresponds to an existing reduction, the task will be assigned the specific reduction slot id, whereas in the opposite situation a new reduction will be registered and a new reduction slot id will be assigned.

It is required to have a data structure responsible for keeping track of which reduction slots are free and available to be assigned to new reductions and which are being used instead. For our implementation, a stack-like data structure is overlapped in the very same memory space of the *CPU 0* reduction slots.

Going into detail, we have a *next slot* pointer that points to a free slot, or to `nullptr` if there are no slots left. Every free slot points at its time to the next free slot, up to the last slot that points to `nulllptr`. This structure is initialized during the initialization phase of the *Nanos6* runtime. Figure 14 shows the described structure.

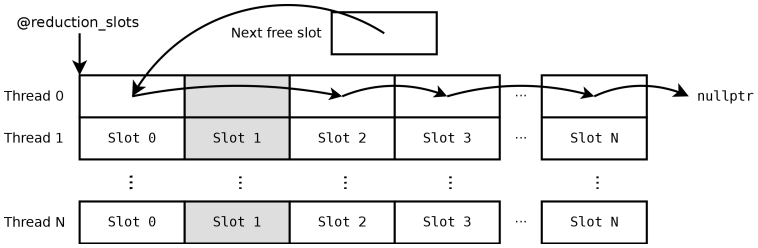


Figure 14: Reduction slots management

When a slot is to be assigned to a new reduction, its address is taken directly from the *next slot* pointer. Then, the *next slot* pointer is updated to the value found in the assigned slot as shown in figure 15.

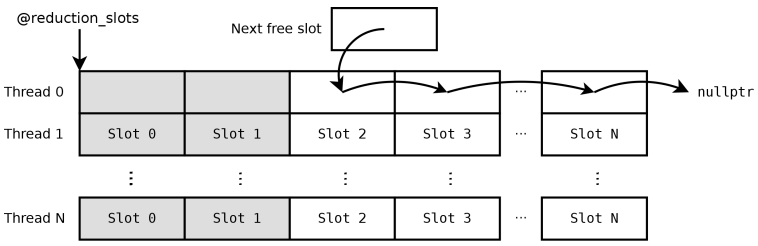


Figure 15: Reduction slots assignation process

When an ongoing reduction is over, the runtime proceeds to free the used slot and push it in the top of

the structure. First, the freed slot is set to the address pointed by the *next slot* field, then the *next slot* is set to point to the freed slot, placing it in the first position of the structure. The process can be seen in figure 16.

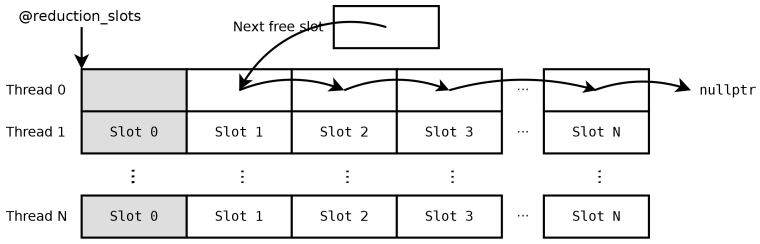


Figure 16: Reduction slots freeing process

Finally, when assigning a slot to a new reduction, it can be the case that there are no more slots available, as the number of allocated reduction slots is fixed. In this case, a whole new reduction slots block will be allocated and initialized. Then, linking the newly allocated block with the previous existing one is as simple as making the *next slot* field point to the new memory space. The described assign/free mechanism will keep the structure properly linked and consistent.

Figure 17 shows how two different reduction slots blocks would be linked after a slot is freed in the original block. For the sake of simplicity, the figure considers there are only four reduction slots in every block.

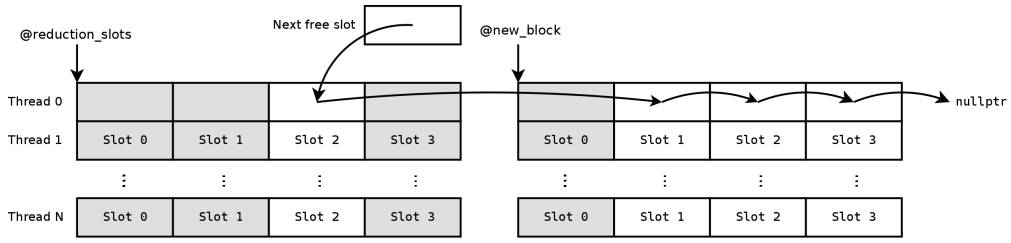


Figure 17: Reduction slots structure with multiple blocks

Note that with this procedure as many blocks as needed can be allocated at runtime.

After the reduction slot id is assigned, the task code can be executed normally over a task-local variable. When the user code is finished, the address of the CPU private copy is retrieved in order to accumulate the performed computation.

Initialization of the private copies

When a new reduction is registered, after having assigned it an identifier and a slot as explained in the previous sections, the runtime will initialize the internal private copies that correspond to that slot with the reduction neutral value.

In detail, the runtime will be calling the initialization function generated by the compiler for every private copy. This function is accessible from the runtime since it was provided when the reduction was registered.

Once the initialization is complete, the reduction tasks can be scheduled and ultimately executed.

Combination of the private copies

Recalling how the reductions are defined in the *OmpSs* model, the value of the original variable or memory position should be updated by the end of the reduction domain, delimited by either a dependence or a `taskwait` in the precise nesting level.

A single mechanism was designed so that both scenarios were supported. The logic behind the mechanism can be summarised in triggering the reduction combination when one of the following conditions is met:

- A reduction task finishes its execution, there are no previous accesses in the access sequence and there are posterior accesses of a type other than reduction. If the posterior access is of type reduction but it corresponds to a distinct reduction, the combination should also be triggered.
- A reduction task finishes its execution, there are no previous nor posterior accesses in the access sequence and the access sequence is marked to have reached a `taskwait`.
- A new access is registered, there are no previous accesses in the access sequence and there is a pending reduction annotated in the access sequence. If the new access is of type reduction, the combination should be triggered only if the annotated reduction and the new access reduction are distinct.
- A `taskwait` is reached and there are empty access sequences with a pending reduction annotated.

Note how the logic is distributed into three different points: the registration of data accesses, the termination of data accesses and the `taskwait` synchronization point.

For the mechanism to work it is required that we modify some existing data structures and add a new one.

First, a field needs to be added in the access sequences so that the last pending reduction can be annotated in it. This information is used by the third and fourth points in the described logic. This field should be updated every time a new access (of any type) is registered.

Secondly, a new data structure is required to keep track of access sequences have an ongoing (uncombined) reduction. New accesses need to ensure the sequence where they belong is present in the container and combining the reductions corresponding to those sequences will invalidate them in the container. For our implementation, an intrusive (with hooks within the objects) doubly linked list makes these operations possible in constant time. This container will be traversed when a `taskwait` is reaching, either combining the access sequence if empty as stated in the fourth point or flagging them to be combined when empty by the second point.

Finally, it is important to clarify that when the combination of a reduction is cleared, the runtime is the one responsible for performing the accumulation onto the original variable by calling the function generated by the compiler provided when registering the reduction via the API call.

Implementation considerations and limitations

The following assumptions were made when implementing this reduction strategy:

- All CPU participate in every reduction, if the reduction computation is too small and the number of CPUs is big, we could notice the overhead of creating and initializing the private copies for each CPU. Another implementation keeping track of which CPUs have participated in the reduction could be beneficial in those situations, in exchange for making the scenario where all of them participate slower.
- Reduction slots are fixed-size. The internal runtime structures will hold fixed-size copies (say 8 Bytes) regardless of the real size of the original reduction variable. However, if the size of the variable is smaller, the implementation will still work.

9.2.3.2 Debugging the implementation

Debugging a parallel application is never easy, different thread schedulings and timings can hide potential data races and other issues, exposing them once every many executions and thus complicating the detection and debugging process. Moreover, using a debugger or memory-sanity tool like *Valgrind* may slow down the execution, and problems that show up during normal execution could be covered up here.

Debugging a runtime library like *Nanos6* is no different. This section shows what measures have been taken in order to ease the process of detection, identification and repair of programming errors.

The *Nanos6* runtime has different execution modes to be selected at load time. The default mode is optimized to provide a good performance, but alternative modes are available for other purposes.

The modes described in table 8 have been of high utility to discover what was really happening inside the runtime.

Execution mode	Description
optimized	Default value, optimized for good performance
debug	Enables assertions and produces debugging symbols
extrae	Instrumented to register *Extrae* events and generate a *Paraver* execution trace
profile	Instrumented to produce a function and source code execution profile
stats	Instrumented to produce a summary of execution metrics (number of tasks, etc.)

Table 8: Used *Nanos6* execution modes

Besides the described execution modes, two additional modes were explicitly extended to provide detailed information of how reductions where being internally managed.

On the one hand, the *graph* execution mode, used to display in a graphical way the data accesses, the dependences between those, and how they evolve in the system. The *graph* mode was extended to support both concurrent and reduction data accesses, labeling them properly and annotating their status changes. Figure 18 shows a sample of the generated steps for a simple execution of seven tasks.

On the other hand, the *verbose* execution mode, used to display text-based information of occurring events and actions taken inside the runtime. The *verbose* execution mode was updated to inform about the following reduction-related events:

- When a new reduction slots block is added to the system (showing the total number of allocated slots).
- When a new reduction access is registered over an address
- When a reduction is combined and in which of the four possible situations described in section 9.2.3.1 the combination happens.
- When a reduction chain is flagged to be combined.
- When a reduction is annotated onto an access sequence.

Figure 19 shows an example output of the text-based information shown by using the *verbose* execution mode. Note that the text `original: 1010` is the program regular output.

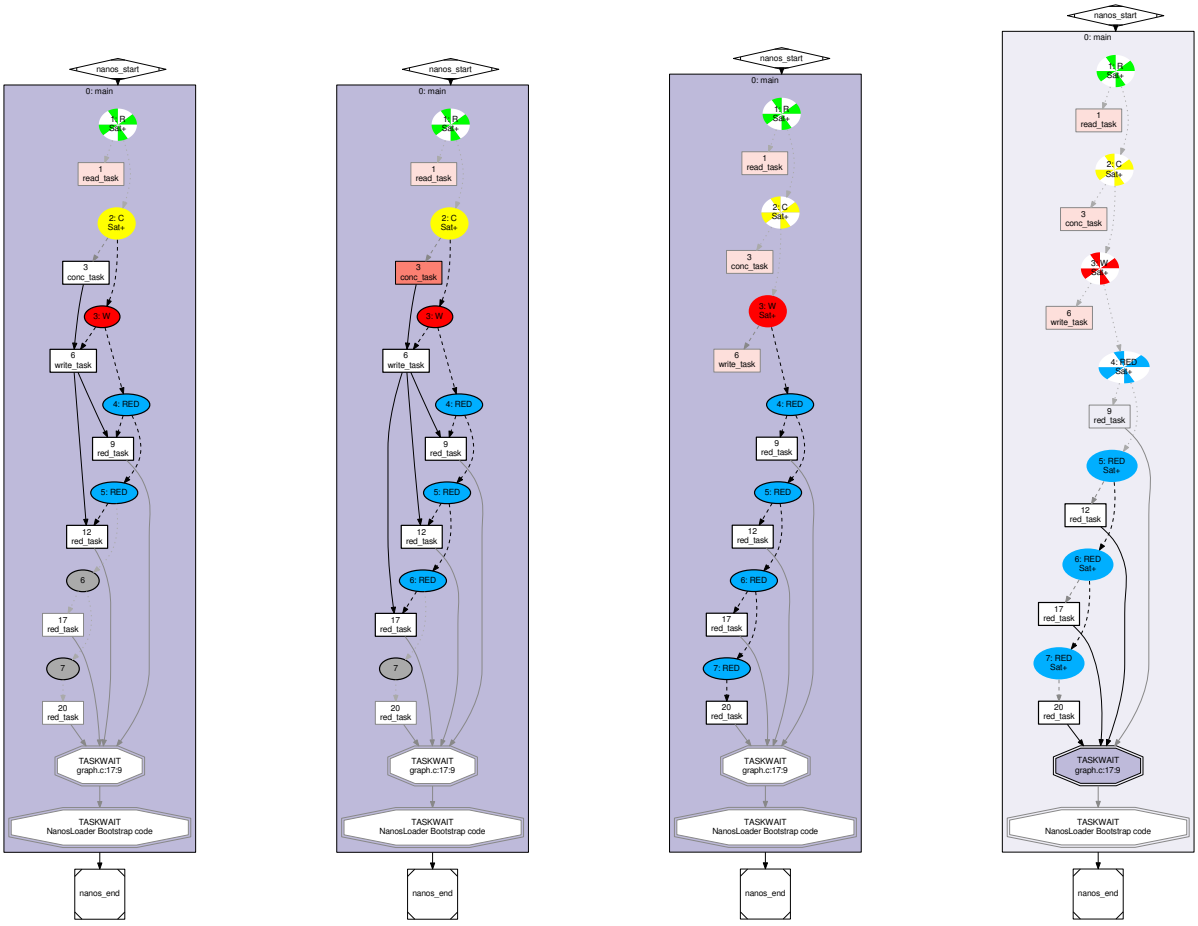


Figure 18: *Nanos6 graph* output example

```
ferran@bsc:/tmp$ NANOS6=verbose NANOS_VERBOSE='reductions,!leaderthread' ./exhaustive
original: 1010
34125.586004510 Thread:external CPU:ANY <-> Reduction slots incremented by 512, total number of slots is 512
34125.586209381 Thread:0xd6cac0 CPU:0 --> Reduction 0 registered for address 0x7f41f9a64d74, reduction private elements initialized
34125.586214594 Thread:0xd6cac0 CPU:0 <-> Access sequence reduction set to 0 when adding a new access
34125.586221060 Thread:0xd6cac0 CPU:0 <-> Access sequence reduction 0 cleared when adding a new access
34125.586226981 Thread:0xd6cac0 CPU:0 --- Reduction data access start address updated: 0x7f41f4000900 -> 0x7f41f9a64d74
34125.586228217 Thread:0xd6cac0 CPU:0 <-> Access sequence reduction set to 0 when adding a new access
34125.586229017 Thread:0xd6cac0 CPU:0 --- Reduction data access start address updated: 0x7f41f4000900 -> 0x7f41f9a64d74
34125.586254082 Thread:0xd6cac0 CPU:0 --- Reduction data access start address updated: 0x7f41f4000900 -> 0x7f41f9a64d74
34125.586255952 Thread:0xd6ccf0 CPU:1 <-> Access sequence reduction set to 0 when adding a new access
34125.586263187 Thread:0xd6ccf0 CPU:1 <-> Access sequence reduction set to 0 when adding a new access
34125.586291315 Thread:0x7f41f4001a20 CPU:0 <-- Reduction 0 combined by finishing a reduction access with an existing posterior
```

Figure 19: *Nanos6 verbose* output example

10 Evaluation and results

In this section we provide a description of the methodology used to evaluate our contribution as well as the obtained results.

10.1 Benchmark methodology

In order to have a fair and efficient benchmarking some were taken.

All tests were executed with exclusiveness of the node. Node resources were not shared with other processes during the performance measures. To achieve this, the Platform Load Sharing Facility (LSF) and Simple Linux Utility for Resource Management (SLURM) workload management systems and their queuing system were used.

Five repetitions were run for each configuration in the benchmark. Then, the measured times for each of the five executions were averaged in order to minimize the effect of the measure variability between executions.

10.2 Machines description

In this section we give an insight of the architecture of the machines used for testing and benchmarking.

It is important to note that our benchmarks will focus on shared-memory architectures, which are the main target of the *OmpSs* programming model. For this reason, all executions will be contained within a single cluster node for all machines.

10.2.1 *MareNostrum III* supercomputer

This machine has been considered as the most reliable platform where to benchmark our implementation. The *MareNostrum III* supercomputer is a member of the PRACE network[24] and, as such, is relied on by many researchers performing their experiments.

Each compute node of the *MareNostrum III* supercomputer is composed of two Intel Xeon E5-2670 processors with 8 cores each and 32GB of RAM. As far as the memory hierarchy is concerned, each socket forms a NUMA node and all the cores that belong to the socket share the L3 cache. The other cache levels are private for each core. Both processors are connected by using a *QPI*[17] connection, forming a NUMA shared memory node.

Even though these processors support *HyperThreading*, it is disabled in this machine, and therefore there is only one thread per core.

Finally, the compiler used in this platform is *Intel icc 17.0.1*.

Details of this cluster node are shown in table 9. Further information about the cluster node topology can be found in the appendix B.

#NUMA nodes	2
Memory/NUMA node	16GB
#Sockets	2
#Cores/socket	8
#Threads/core	1
#Total threads	16
L3 size	20MB
L2 size	256KB
L1d size	32KB
L1i size	32KB
L3 shared amongst	socket
L2 shared amongst	core

Table 9: *MareNostrum III* node summary

10.2.2 KNL cluster

The KNL cluster gets its name from its processors: the second generation of the Intel Xeon Phi products, codenamed Knights Landing. The underlaying architecture of the Xeon Phi processors and coprocessors is known as the Intel Many Integrated Core (MIC) architecture. As opposed to the *MareNostrum III* supercomputer, the KNL cluster is a much smaller machine (4 nodes) currently used for research in the field of computer architecture.

Each compute node of the KNL cluster is composed by a KNL Xeon Phi processor of 68 cores and 4 threads per core. Every node has a total of 94GB of RAM.

This nodes have the special characteristic that they use an additional memory of type Multi-Channel DRAM (MCDRAM). This memory type stands out for physically sitting atop the processor, providing higher bandwidth and lower latencies when compared to the conventional Dual-Inline Memory Moduless (DIMMs)[6]. For running our benchmarks, this memory has been configured to cache the accesses to the main Double Data Rate type 4 Synchronous DRAM (DDR4) memory.

A peculiarity from this machine with respect to the others is that the L2 level of cache is shared between every two cores.

Finally, the compiler used in this platform is *Intel icc 17.0.0*.

Details of this cluster node are shown in table 10. Further information about the cluster node topology can be found in the appendix B.

#NUMA nodes	1
Memory/NUMA node	94GB
#Sockets	1
#Cores/socket	68
#Threads/core	4
#Total threads	272
L3 size	-
L2 size	1MB
L1d size	32KB
L1i size	32KB
L3 shared amongst	-
L2 shared amongst	every two cores

Table 10: KNL node summary

10.2.3 *Power8* cluster

As the KNL machine, the *Power8* cluster is a 4-node machine mainly used for research about computer architecture.

Each compute node of the *Power8* cluster is a S822LC system[16], composed of 2 Power8 processors of 10 cores each and 256GB of RAM. Each core has 8 threads and both processors are interconnected forming a NUMA shared-memory machine.

Finally, the compiler used in this platform is *GCC 4.9.3*.

Details of this cluster node are shown in table 11. Further information about the cluster node topology can be found in the appendix B.

#NUMA nodes	4
Memory/NUMA node	128GB
#Sockets	2
#Cores/socket	10
#Threads/core	8
#Total threads	160
L3 size	8MB
L2 size	512KB
L1d size	64KB
L1i size	32KB
L3 shared amongst	core
L2 shared amongst	core

Table 11: *Power8* node summary

10.2.4 *ThunderX* cluster

The *ThunderX* cluster differs from the other clusters by using based in the *ThunderX* System on Chip (SoC) developed by Cavium, which is based in the ARMv8 architecture.

Each compute node of the *ThunderX* cluster is composed of 2 SoC of 48 cores each and 128GB of RAM.

The *ThunderX* SoC is the first ARM based SoC that is fully cache coherent across dual sockets. To achieve this, Cavium developed its own proprietary protocol called Cavium Coherent Processor Interconnect (CCPI).[10]

Finally, the compiler used in this platform is *GCC 6.1.0*.

Details of this cluster node are shown in table 12. Further information about the cluster node topology can be found in the appendix B.

#NUMA nodes	2
Memory/NUMA node	64GB
#Sockets	2
#Cores/socket	48
#Threads/core	1
#Total threads	96
L3 size	-
L2 size	16MB
L1d size	78KB
L1i size	32KB
L3 shared amongst	-
L2 shared amongst	socket

Table 12: *ThunderX* node summary

10.3 Benchmarks description

10.3.1 N-queens

The n-queens problem is the generalization of the 8 queens puzzle, where 8 chess queens need to be placed in an 8×8 chessboard so that no two queens threaten each other[31]. In the generalized version n queens need to be placed in an $n \times n$ chessboard for a given natural n.

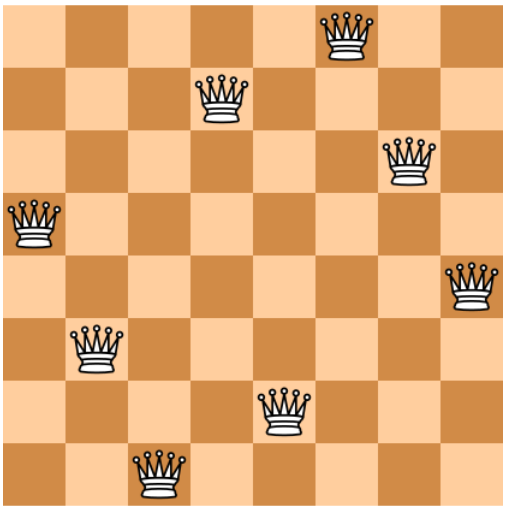


Figure 20: One of the 92 solutions for the 8 queens problem

While finding a single solution to the problem for a given n can be done efficiently in polynomial time[26], finding every possible solution requires an exhaustive search and has an exponential cost.

The proposed solution of the problem is based in a recursive depth-first search, where the board is traversed either row-wise or column-wise (it does not matter since the board is square) while placing a queen in a safe square at each step.

If a partial solution to the problem is expressed as a linked list of positions where a queen is placed, the remaining solution tree can be explored in parallel. In particular, after the placement of a queen, a new task is created for every safe position in the adjacent column to count for all possible solutions sharing the positioning up to that point.

The reduction pattern can be found in the counting for the number of solutions to the problem. Each task will be either accumulating a found solution to the global counter if it represents a leaf node or forward the reduction by creating more tasks that will possibly do so at some point (either them or their descendants). This process defines a single reduction across the whole solution tree where all tasks participate.

It is important to note how the defined reduction is recursive, and therefore this problem is not only a real problem where nested reductions are required but also will provide a valid benchmark to evaluate the effect of having a `taskwait` clause in each nesting level.

By using the described strategy the number of created tasks will grow exponentially to the size of the problem, making the overhead of managing such a big amount of tasks disproportionate to the benefits the parallel execution could bring. To stop the creation of tasks once reached a certain level of parallelism the `final` clause will be used. The depth specified in the `final` clause will determine how many levels of recursion (or rows/columns in the chessboard) will be explored until no more tasks are instantiated. Once a thread enters in final mode, it will finish exploring its local branch of solutions until there are no combinations left to try.

It should be taken into consideration how the depth value passed to the `final` clause determines a trade-off between the amount of tasks in the system and their granularity: The bigger the value, a greater number of finner tasks and vice-versa. Having a small number of coarse tasks can reduce overheads but it can also unbalance the amount of work to be done for each thread, increasing the execution if the solution tree is not balanced (which is not).

In figure 21 we see a binary solution tree that exemplifies this last point by showing how setting a final depth too small would result in an unbalance the threads workload. See how under the final threshold there is not a fair distribution of the workload among the threads (illustrated in the figure by using different colors).

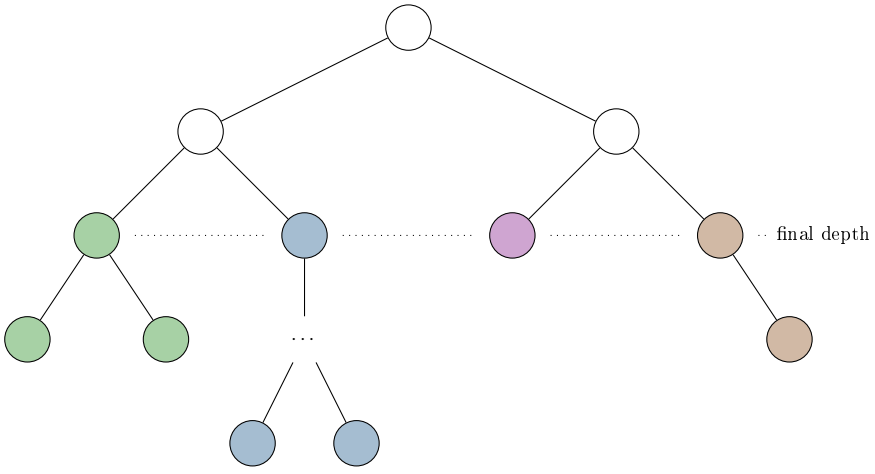


Figure 21: Unbalanced n-queens solution tree

For this benchmark, the impact of the `taskwait` clause will be evaluated by comparing the results of the same configurations obtaining by placing or removing the `taskwait` clause in each nesting level of the reduction, as shown in figures 22 and 23.

Listing 22: Nested reduction with `taskwait`

```
int x = 0;
#pragma oss task reduction(+:  x)
{
    #pragma oss task reduction(+:  x)
    x += ...

    #pragma oss task reduction(+:  x)
    x += ...

    #pragma oss taskwait
}
```

Listing 23: Nested reduction without `taskwait`

```
int x = 0;
#pragma oss task reduction(+:  x)
{
    #pragma oss task reduction(+:  x)
    x += ...

    #pragma oss task reduction(+:  x)
    x += ...
}
```

10.3.2 Unbalanced tree search

The next problem we consider in our benchmark is the UTS. As stated by its name, this problem is characterized by completely traversing an unbalanced tree while doing some computation for each node.

Like in the n-queens problem, our algorithm is based in a recursive depth-first search. However, there are meaningful singularities in the shape and growth of the recursion tree that make them completely different problems.

For our benchmark, of the generation of the tree will occur dynamically during runtime by means of the *Galton-Watson* branching stochastic process[30]. Specifically, the algorithm will decide whether the node is traversing will instantiate children or will become a leaf node otherwise.

The branching factor at all nodes except the root follows an identical binomial distribution. A node has m children with probability q, or no children (becoming a leaf) with probability $(1-q)$. The expected branching factor is then $q \cdot m$. The number of children for the root node is explicitly defined as part of the execution input.

Figure 22 shows an example of such trees where the root branching factor is 5, all non-root nodes have 2 children with a probability of 0.49 and therefore the expected branching factor for all non-root nodes is 0.98 (cropped at depth 8).

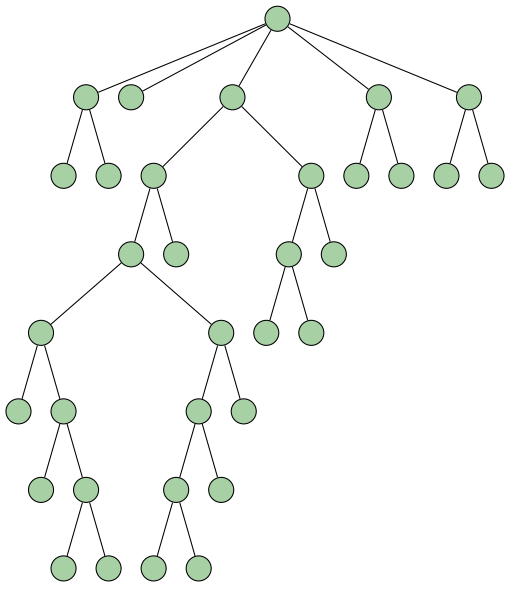


Figure 22: Stochastically branched unbalanced tree

Considering the unbalanced shape and growth mechanism of the generated trees, it is impossible to determine a reasonable cut-off depth to use in combination with the `final` clause, making it useless for this problem. For this reason, the runtime support library will have to deal with the overhead of creating a task for every node of the tree.

In our benchmark, the computation to be done for each node consists in a number of Secure Hash Algorithm 1 (SHA-1) operations. The number of SHA-1 operations to be done is determined by an input parameter, allowing the comparison of different task granularities.

In this case, the reduction pattern can be found in the counting for the number of nodes of the unbalanced tree (remember how the trees are dynamically generated). As opposed to the n-queens problem, each task will be accumulating the node count to the global counter and not just the leaf nodes. Again, a single recursive reduction is defined across the whole tree where all tasks participate.

By including a recursive reduction, the problem is also useful to further evaluate the effect of having a `taskwait` clause in each nesting level and the described strategy is the previous section will also be used for evaluating its impact on this benchmark.

10.3.3 Dot product

The dot product or scalar product is an algebraic operation that takes two vectors of numbers and returns a single number. The result is computed the sum of the products of each corresponding position of the two vectors.[32]

As expected, this benchmark computes the dot product between two vectors.

The dot product definition above leads us straight into the reduction pattern, which is to combine each of those products into a single number by adding them.

As opposed to the previous benchmarks that follow a recursive reduction scheme, the dot product follows a flat reduction scheme. The input vectors are broken into fixed-size blocks and a task is instantiated for each pair of corresponding blocks, reducing the product of each pair of elements within the blocks into the global result. This tasking scheme can be seen in figure 23, where the product of two given `A` and `B` vectors wants to be computed.

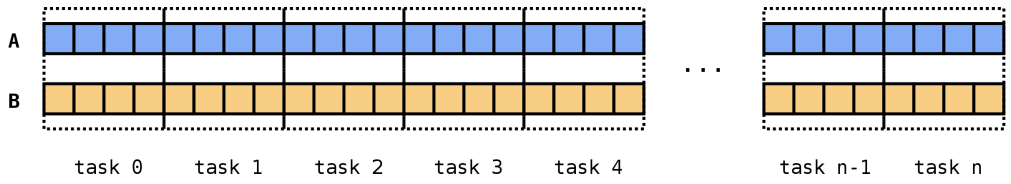


Figure 23: Dot product tasking scheme

This problem is also highly memory-bound and vectorizable, meaning that very little computation is done for every memory access.

The algorithm we consider for our benchmark is based in having one thread instantiating initialization tasks for each block of the vectors followed by instantiating the computation tasks for those blocks with dependences on the initialization tasks. The aim of doing this is for facilitating it for the scheduler to schedule the computation task for some blocks on the same thread the initialization of those blocks was executed. This way, we can take advantage of reusing those memory pages as much as possible memory, avoiding having to access different pages and maximizing the use of the memory hierarchy. This scheduling strategy is called immediate successor.

10.4 Results

In this section we present the results obtained for our implementations of the *OmpSs* `reduction` clause when executing the benchmarks described in section **??** on the different platforms presented in section 10.2.

In order to provide fair conditions to evaluate the strong scalability of our implementations, we have tunned the benchmarks' input parameters with the intention to find a fixed problem size for each benchmark that is suitable for every platform. For the sake of reproducibility, those settings are displayed in the appendix A.

The strong scalability test consists in fixing the problem size and gradually increasing the number of execution resources to see if the obtained benefit is equally proportionate to the increment. This experiment is useful to find the optimal execution configuration for a given problem size and can show how in some situations incrementing the execution resources does not imply an increment in performance.

In addition, we present a comparison of our implementation against *OpenMP* using the best-performing configurations. As far as the *OpenMP* implementation of the reductions is concerned, a user-driven thread-privatization have been used by declaring thread-private copies to be used within the tasks and combining them in the end of the parallel region by using the *OpenMP* `reduction` clause.

As a general comment, in this section we present the results for *OmpSs* without doing any distinction among versions. The reason of this decision is that the proposed implementations are equivalent performance-wise in this context, as detailed in section 9.2.3.

10.4.1 Results in *MareNostrum III*

In this section the results obtained for the *MareNostrum III* supercomputer are presented.

For this platform we will discuss the scalability of our implementations based on the results of executing the benchmarks using from 1 to 16 threads, doubling the number of threads in each step.

The results obtained for strong scalability in the n-queens and UTS benchmarks are shown in figures 24 and 25 respectively. We can see how in this platform both problems scale almost perfectly in the number of threads.

For the n-queens benchmark, no differences in time have been found in any machine between executions having a `taskwait` clause in each recursion level and those not having it. The reason of this is that the `final` clause greatly reduces the number of tasks and consequently the effect of the `taskwait`.

In *MareNostrum III* in particular, the `taskwait` has not affected the execution time of the UTS benchmark either, possibly due to having only 16 cores.

When it comes to the dot product, figure 26 shows a worse scaling if compared to the previous benchmarks, this is expected due to the memory-bound nature of the benchmark. In addition, we can see how doubling the number of threads in the last step (from 8 to 16) does actually slow down the execution. The reason of this is that when we go from a single NUMA node to two, the slow-down that supposes having some threads accessing to the memory of the other node outweighs any advantage given by the increment in the number of computing resources.

It is worth commenting the peculiar situation found while adjusting the compute granularity for the UTS problem seen in figure 27. The displayed chart shows how while a x10 increase (from 10 to 100) in the computation weight (amount of work to be computed) for each task does not seem to cause any increment in the final execution time but a drop instead. The explanation for this behaviour is that in the first scenario our runtime has a really poor performance due to having a lot of contention in the scheduling queue, but very little computation to perform. In the second scenario, however, the tasks have already a weight large enough to make the threads spend more time computing useful work rather than frenzy asking for tasks to execute. Figures 28 and 29 show an execution trace for the first and second situations respectively. In the traces, the time spent in the runtime creating the tasks, registering their accesses, etc. is shown in

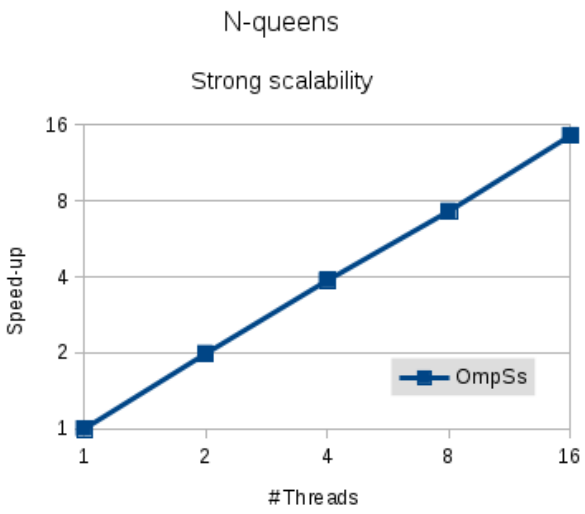


Figure 24: N-queens *OmpSs* in *MareNostrum III*

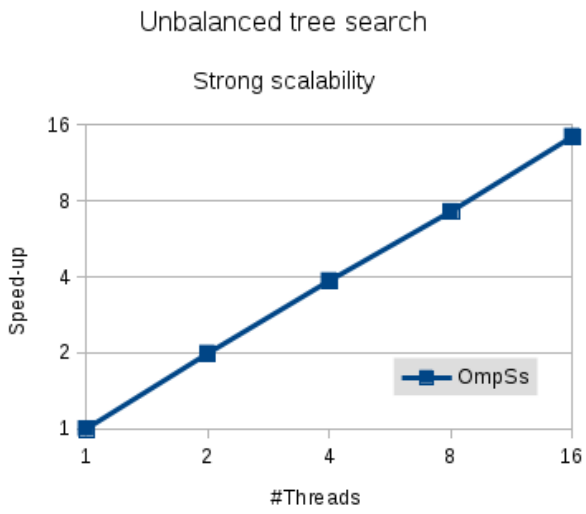


Figure 25: UTS *OmpSs* in *MareNostrum III*

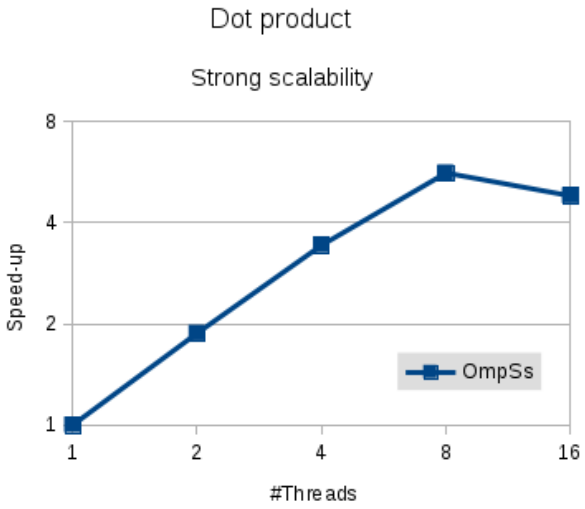


Figure 26: Dot product *OmpSs* in *MareNostrum III*

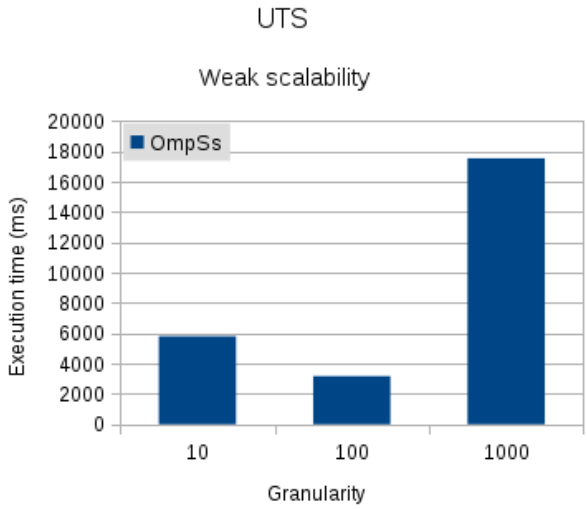


Figure 27: UTS granularity study in *MareNostrum III*

pink, while the execution of the proper task code is painted yellow. In the upper figure we can see a lot of time is spent in the creation of the tasks, specifically, we can see a lot of synchronization between threads (painted in red in the first thread). On the other hand, in the lower figure we can see how the creation time and synchronization has been cut down and the execution of the proper task code has incremented (due to the increase in the computation weight). However, the total cost for creation plus compute is still smaller in the second scenario. This situation happens for every platform, the specific results can be seen in the appendix A.

Finally, figure 30 shows the comparison of the best scenario in the previous figures (16 threads for n-queens and UTS, 8 for the dot product) against Intel *OpenMP*.

In the figure we see how both parts have a similar behaviour for the UTS and dot product benchmarks, whereas *OmpSs* is much faster than *OpenMP* when running n-queens. That being said, the underlying

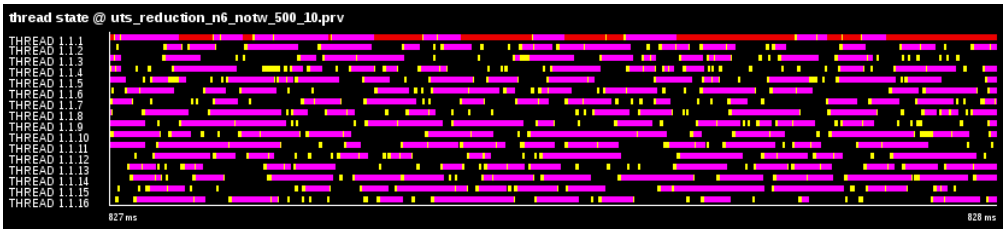


Figure 28: Execution trace of UTS in *MareNostrum III* with granularity 10

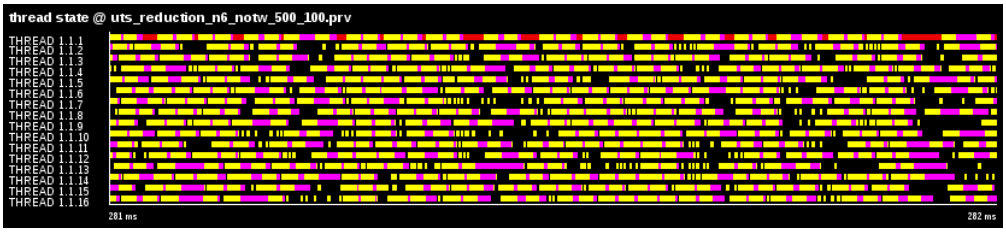


Figure 29: Execution trace of UTS in *MareNostrum III* with granularity 100

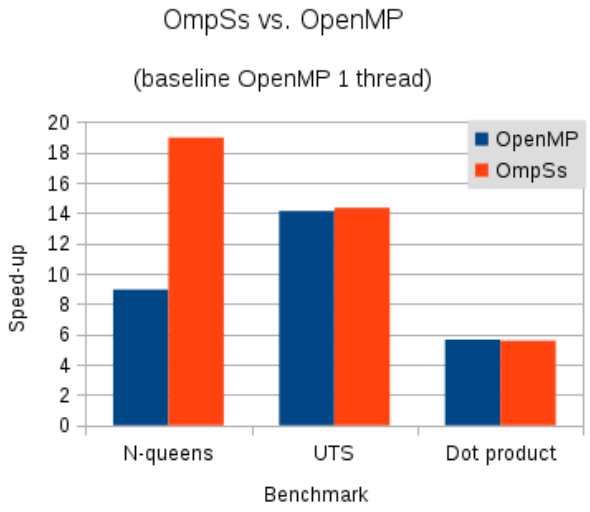


Figure 30: *OmpSs* vs. *OpenMP* in *MareNostrum III*

reason of such difference is mainly due to the optimizations done for the `final` clause in *OmpSs* and have little to do with our implementation of the `reduction` clause.

10.4.2 Results in KNL

In the KNL cluster there are a total of 272 threads distributed in 68 cores arranged in a single socket (a NUMA node itself). For this architecture, our scalability tests consist in executing the benchmarks on the whole node starting by using a thread for each core (68 in total), and increasing by one the number of threads per core in each step until all resources are used (272 in total).

The strong scalability results for the n-queens and the dot product can be found in figures 31 and 32 respectively. The first conclusion we draw from the first figure is that this machine has a good scalability up to having one thread per core, from this point onwards, incrementing the number of threads in the cores

brings a much smaller speed-up.

If we focus on the dot product, the problem has an extremely bad scalability. Be that as it may, the comparison against Intel *OpenMP* below shows that this problem does not suit the KNL architecture at all.

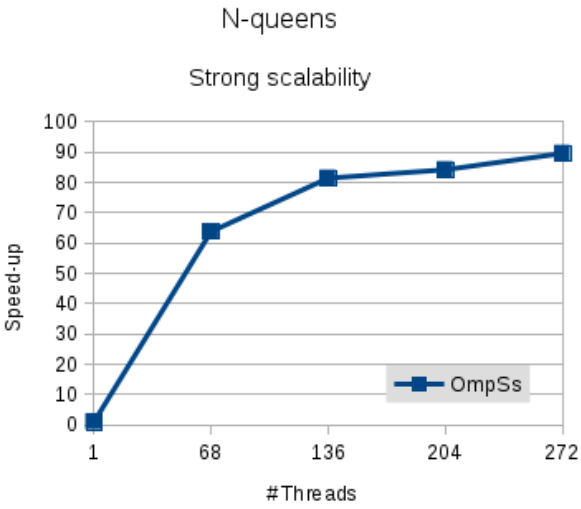


Figure 31: N-queens *OmpSs* in KNL

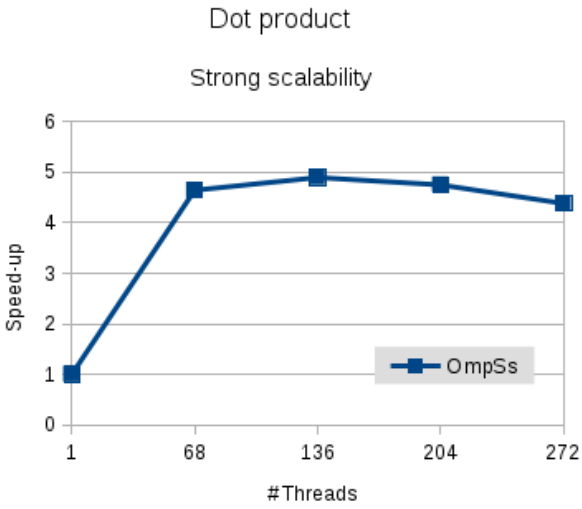


Figure 32: Dot product *OmpSs* in KNL

Regarding the results obtained for UTS in figure 33, we can see the effect of the contention caused by having a big number of threads trying to access the task queue structure either for adding new tasks (an effect of the recursive algorithm) or for retrieving tasks to execute. In addition, we can see how an important speed-up could be obtained from avoiding the `taskwait` synchronization in each recursion level when compared to the version having it. As opposed to the n-queens benchmark, the UTS instantiates a big number of tasks that have to be waited for when a `taskwait` clause is encountered, producing an important overhead.

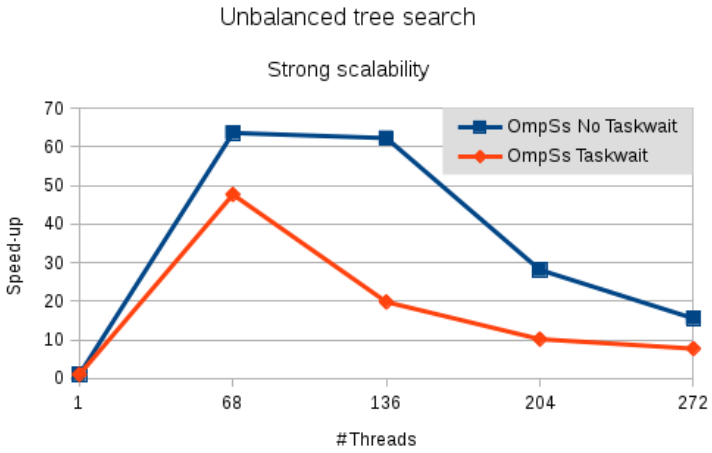


Figure 33: UTS *OmpSs* in KNL

Finally, figure 34 shows the comparison of the best found scenario as seen in the previous figures against Intel *OpenMP*. Similarly to what happens in *MareNostrum III*, a great speed-up is achieved in n-queens in *OmpSs* due to the optimizations in the `final` clause.

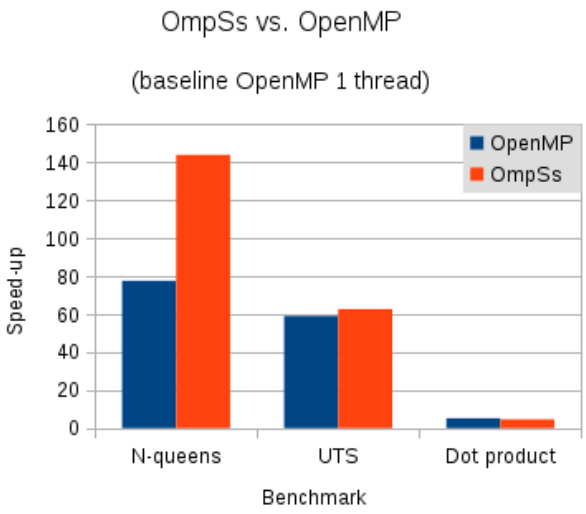


Figure 34: *OmpSs* vs. *OpenMP* in KNL

10.4.3 Results in *Power8*

In the *Power8* cluster, the cores are arranged in 2 sockets of 10 cores each, and for each core there are 8 threads. For this architecture, our scalability tests consist in executing the benchmarks on the whole node starting by using a thread for each core (20 in total), and doubling the number of threads per core in each step until all resources are used (160 in total).

The strong scalability results for the n-queens and the dot product can be found in figures 35 and 36 respectively. At first sight we see good scalability for n-queens up to having one thread per core (first step) and still a decent speed-up when increasing the number of threads per core to 2 or even 4, specially if we compare it with the KNL cluster.

For the dot product, while slightly better than KNL, the problem does not scale for this platform either. The best configuration for this problem is having only a thread per core, this makes sense if we consider the memory bound nature of the problem and the fact that in *Power8* the cache memories are shared among all threads in a core: Adding more execution resources for the same memory hierarchy causes more disturbances among than benefit to the system.

Then, the results obtained for strong scalability of the UTS in figure 37 show a considerable divergence if we consider the impact of having a `taskwait` at each recursion level. The underlying reasoning is the same exposed above for the KNL machine. Again, this is a good example to justify our effort for avoiding synchronization restrictions in the *OmpSs* model whenever possible.

Finally, Intel *OpenMP* is not available in this machine, consequently the GCC implementation of the model has been used for the comparison in figure 38. In this architecture the models compare much differently than what was seen in *MareNostrum III* and KNL. First we see how *OpenMP* outperforms *OmpSs* in the n-queens problem. While the most probable explanation of this result is that the compiler is applying some optimization that *Mercurium* is incapable to apply, we have not had the opportunity to further investigate on this issue, leaving it for future work of optimizing *OmpSs* for this architecture. On the other hand, a speed-up in favour of *OmpSs* is found for the UTS benchmark, trait that did not show up in the previous architectures.

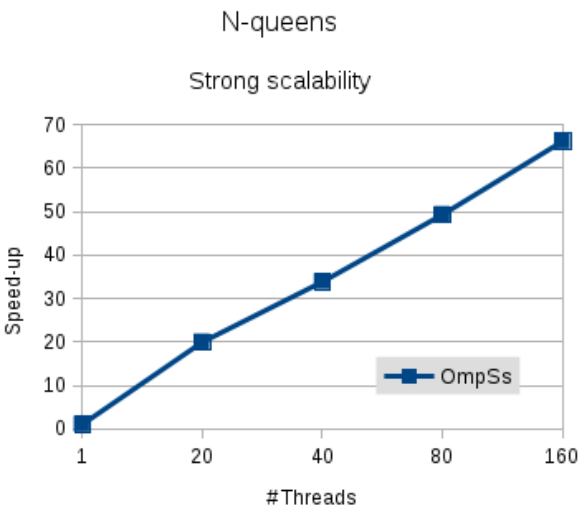


Figure 35: N-queens *OmpSs* in *Power8*

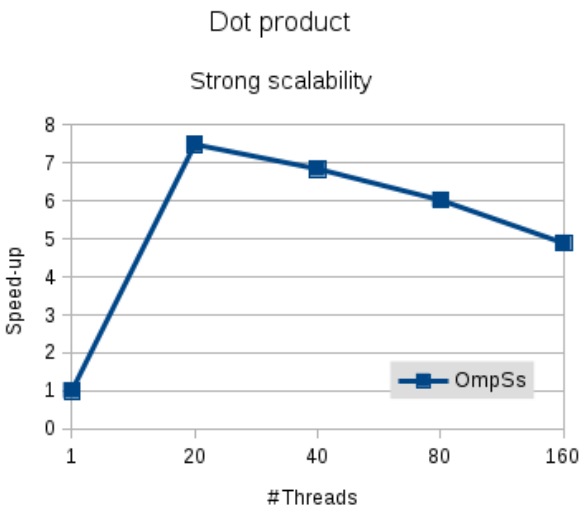


Figure 36: Dot product *OmpSs* in *Power8*

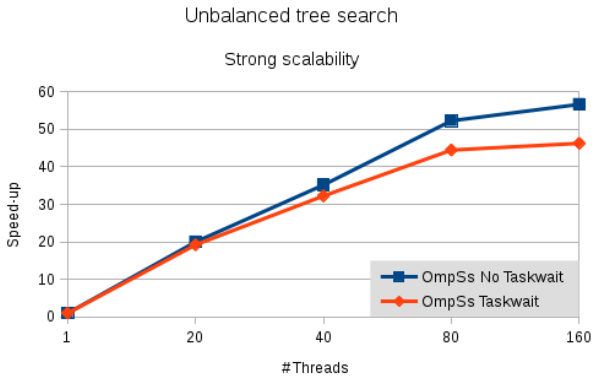


Figure 37: UTS *OmpSs* in *Power8*

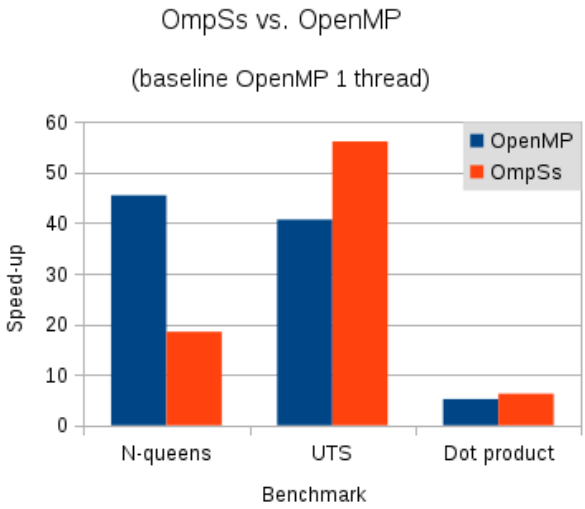


Figure 38: *OmpSs* vs. *OpenMP* in *Power8*

10.4.4 Results in *ThunderX*

Finally, for the *ThunderX* cluster, all 96 cores are arranged in 2 sockets of 48 cores each, there is only one thread in each core. For this architecture, our scalability tests consist in executing the benchmarks in a single core, a whole NUMA node (48 threads) and finally the whole machine.

In addition to those configurations, for this machine we also consider the configuration using half the total cores but distributed among the two NUMA nodes. On the one hand, this can be beneficial in the sense that both L2 caches will be shared only among half the cores. On the other hand, this configuration could be affected by the NUMA effect of accessing memory positions from the other NUMA node. This configuration is labeled as "48:2" in the figures, as opposed to using only one node, which is labeled as "48:1".

The strong scalability results for the n-queens and the UTS can be found in figures 39 and 41 respectively. As for this architecture the cores are not multi-threaded, both benchmarks show a scalability almost perfectly in the number of threads.

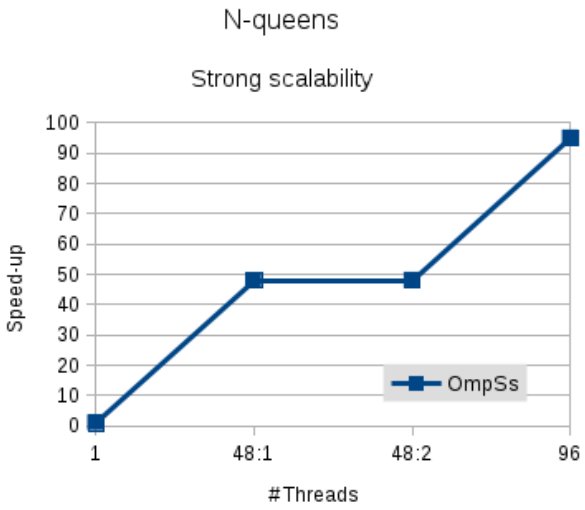


Figure 39: N-queens *OmpSs* in *ThunderX*

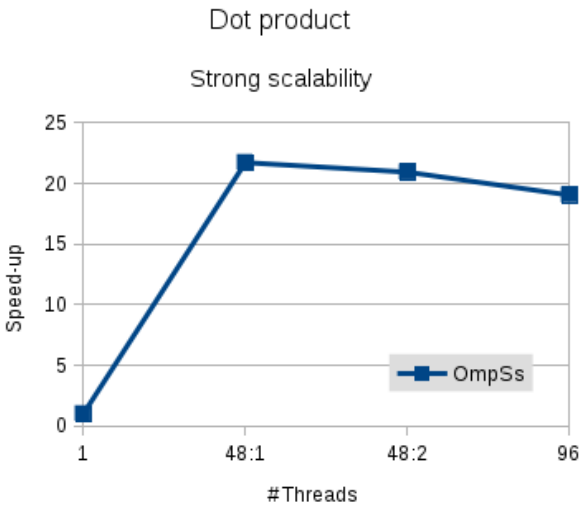


Figure 40: Dot product *OmpSs* in *ThunderX*

For the dot product, while definitely better than KNL and *Power8*, the benchmark struggles to scale in this platform as well. The best configuration for this problem is remaining within a single NUMA node. When it comes to the alternative configuration involving half the cores distributed among both sockets, we find that the better use of the memory hierarchy can not be compared to the cost of accessing memory positions from the foreign NUMA node.

Finally, as in *Power8*, Intel *OpenMP* is not available in this machine either and again the GCC implementation of the model has been used. The comparison can be found in figure 42. In this architecture *OmpSs* outperforms *OpenMP* in both n-queens and UTS, while both models have a considerably greater speed-up in the dot product benchmark as compared to other platforms. The reasoning behind this speed-up in the dot product is that each of the ARM processors in this machine have considerably lower computing power and more cores are required to reach the memory bound, thus showing better scalability until its reached. For the other machines, the memory bound is reached with far less computing resources.

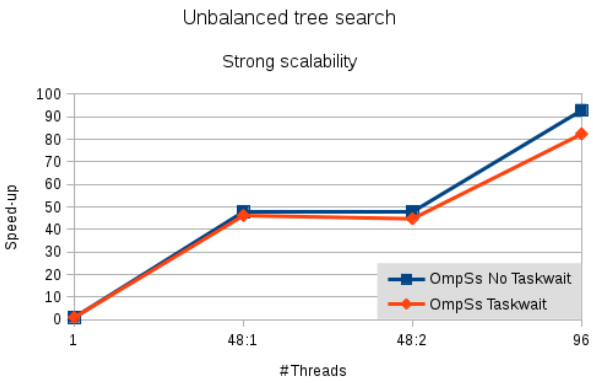


Figure 41: UTS *OmpSs* in *ThunderX*

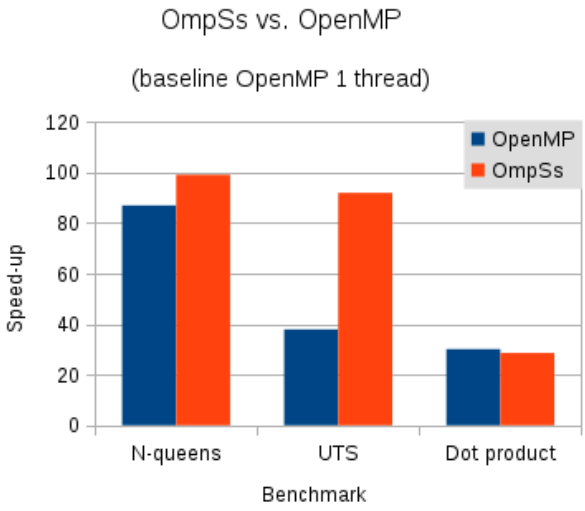


Figure 42: *OmpSs* vs. *OpenMP* in *ThunderX*

11 Conclusions

The work accomplished during the course of this project is an effort to make it easier for the *OmpSs* users to include reduction patterns in their parallel applications. The proposed way to achieve this purpose has been extending the current *OmpSs* infrastructure to support the `reduction` clause, a functional clause that lets the user specify the reduction pattern in a convenient, clean and pleasant way. This extension will help reducing the effort required to design parallel programs with this pattern, drastically reducing the probability of writing erroneous code and hence reducing the overall development time for the user.

The first step in this direction involved adding support to the `concurrent` clause, giving a greater degree of freedom to the users by providing them of a new way to access the data within the dataflow model and bringing room for new optimizations.

As far as the `reduction` clause is concerned, two strategies have been developed: The first one, based on a task-level privatization, consists mostly in compile-time transformations performed by the *Mercurium* compiler while the accesses are handled as regular `concurrent` accesses by the *Nanos6* runtime. In the second one, based on both task-level and CPU privatization, the reductions are registered into the runtime, which is responsible for managing the private copies and handling the combination processes.

While for scalar reductions in architectures that support atomic operations both strategies are equivalent in terms of performance, for arrays and User-defined Reductions only the latter would be competitive. In addition, by handling the control to *Nanos6*, run-time optimizations like early beginning of reduction tasks could be implemented.

Such feature would not be of much utility if the provided implementation were not competitive performance-wise. After all, the final goal of the *OmpSs* programming model is to facilitate ease programming for parallel architectures without compromising the performance.

In order to evaluate the performance of our implementation three distinct benchmarks have been designed. These benchmarks have been executed in four different architectures in order to study how the implementation performs on distinct scenarios, providing a complete evaluation on the feature.

The results obtained from each benchmark justify our work and support our efforts for eliminating the necessity of placing synchronization constructs within nested reductions.

To conclude, the initial goals of the project described in section 2.1 have been fulfilled and a well-performing mechanism to simplify the computation of reductions has been developed and evaluated on a range of different machines and applications.

12 Future work

Even though the project has fulfilled its initial goals and the obtained results are satisfactory, there are still many aspects related to the task reductions left to be dealt with.

The first and most natural extension of this project would be to extend its implementation to the *Nanos6* region-based dependence system. This would allow supporting array reductions, which is very convenient for some applications. Simple as it may seem at first, the generalization of the mechanism proposed in this project is not trivial and special care needs to be taken in recursive schemes, where the privatization of array regions could easily become unsustainable. We should certainly expect to see an implementation of the reduction scheme for regions in the upcoming years.

Closely related to the last point, it could be interesting to develop the new clauses `weakconcurrent` and `weakreduction`. Analogous to the already existing `weakin`, `weakout` and `weakinout` clauses, the proposed new clauses would be useful to declare tasks that do not directly access the data but instead create nested tasks annotated with a `concurrent` clause or a `reduction` clause respectively that do access those data. By annotating such tasks using the `weak` clauses as opposed to their regular counterparts would provide this extra piece of information that could led to further run-time optimizations. For region-based reductions, this could suppose avoiding the privatization of a whole array when not necessary.

An additional aspect to consider in future projects would be to support UDRs. The UDR is a mechanism that allows the user to define new reduction operators. This feature could be used not only for better ad-hoc solutions but also to support reductions in `structs`. UDRs are part of the *OpenMP* standard since the *OpenMP* 4.0 revision.

As a final note, further investigation on the behaviour of some benchmarks under some platforms, like for instance n-queens in *Power8*, could be useful to increase the understanding of how the runtime performs in those architectures and what could be done to optimize it.

13 Project revision

This section describes how the approach taken for this project has changed during its process, focusing in the work-plan but revising other main aspects defined during the initial assessment as well. Each of the following points focus on one of those aspects.

13.1 Task specification revision

While there have not been major changes in terms of the project objectives or the initially proposed schedule, some stages have been adjusted or adapted for better fitting the needs of the project.

In the first place, in an early meeting with the directors, the initially-proposed extension of scalar reductions for a region-based dependence system was thoughtfully discussed. It was pointed out that the natural extension of the project for the region-based dependence system was the development of array reductions over just scalar reductions support. Whilst the first is not yet implemented, an application requiring scalar reductions could as well be adapted to use the discrete-dependence system.

Extending the region-based dependence system with scalar reductions is a very logical step for setting up the bases and further extending them into array reductions. In this sense, before being able to provide an extendible implementation for scalar reductions in such system it is important to first deeply analyse and have a complete scheme of how array reductions will be implemented. Under that reasoning, and considering the effort that it would require to adapt our implementation to the region-based dependence system given the little utility of the result, it was agreed to finally consider this phase out of the scope of the project.

Conversely, the *OmpSs* `concurrent` clause that was not considered during the initial assessment of the project was completely implemented and tested for point dependences. The `concurrent` clause allows instantiating a set of tasks having dependences with both previous and posterior tasks, but no dependences between the tasks in the set, allowing their concurrent execution. The clause was developed as a first logical step in the process of developing task reductions for *OmpSs*, as shown in section 9.1.

Another development process that was not initially considered was the adaptation required for the applications to be run in the *OmpSs* runtime. For some applications, no adaptation was necessary besides using the reduction *pragma*, whereas in others, the original source code had to be redesigned to remove otherwise necessary `taskwait` s and synchronization points.

As far as time deviations are concerned, we could almost fit the described changes in the work-plan by using the time reserved for the extension of scalar reductions on the region-based dependence system.

13.2 Obstacles revision

During the project we have inevitably faced some of the obstacles foreseen during the initial assessment in the obstacles section 2.3.

On the one hand, the *BSC-CNS* machines *MareNostrum III* and *Power8*, which this project relies on for the execution of some tests, underwent some maintenance tasks, including the upgrade of the General Parallel File System (GPFS) in the case of *MareNostrum III*. This happened 12th December from midnight until 20.00h the same day. During this period, both machines were unavailable for the users. While it was planned to use the machines for testing during that day, those tasks could be rescheduled and advance further in the implementation stages instead, resulting with no time deviation for the global project schedule.

On the other hand, at some point during October, an unconnected error or *bug* was introduced in the runtime by some people working on other projects related with *OmpSs*. This error affected the main runtime scheduler in particular and, being a critical part of the runtime, caused the overall runtime performance to degrade considerably. In more detail, the error was responsible for some threads being incorrectly pinned to the

CPUs they had previously been attached to, resulting with more than one thread pinned to the same CPU at the same time and causing major problems in mutual exclusion regions.

This issue forced us to repeat all the testing and benchmarks done up to the point, as the measured performance was not valid. In consequence, the testing withing the development phase was prolonged a bit more that it would have been desirable.

13.3 Work-plan revision

Everything added up resulted in a deviation of around 15 hours, resulting in an expected amount of 565 hours for the whole project. The extra required time was fitted in the schedule by taking advantage of the reserved time-frame for deviations and increasing the developer workday.

Figure 43 shows a revised version of the original *Gantt* diagram where the up-to-date status of the project is shown in a clear way. Specifically, we can visualize the situation of the follow-up meeting in the project schedule, having a clear picture of which tasks are over and which are yet to be done.

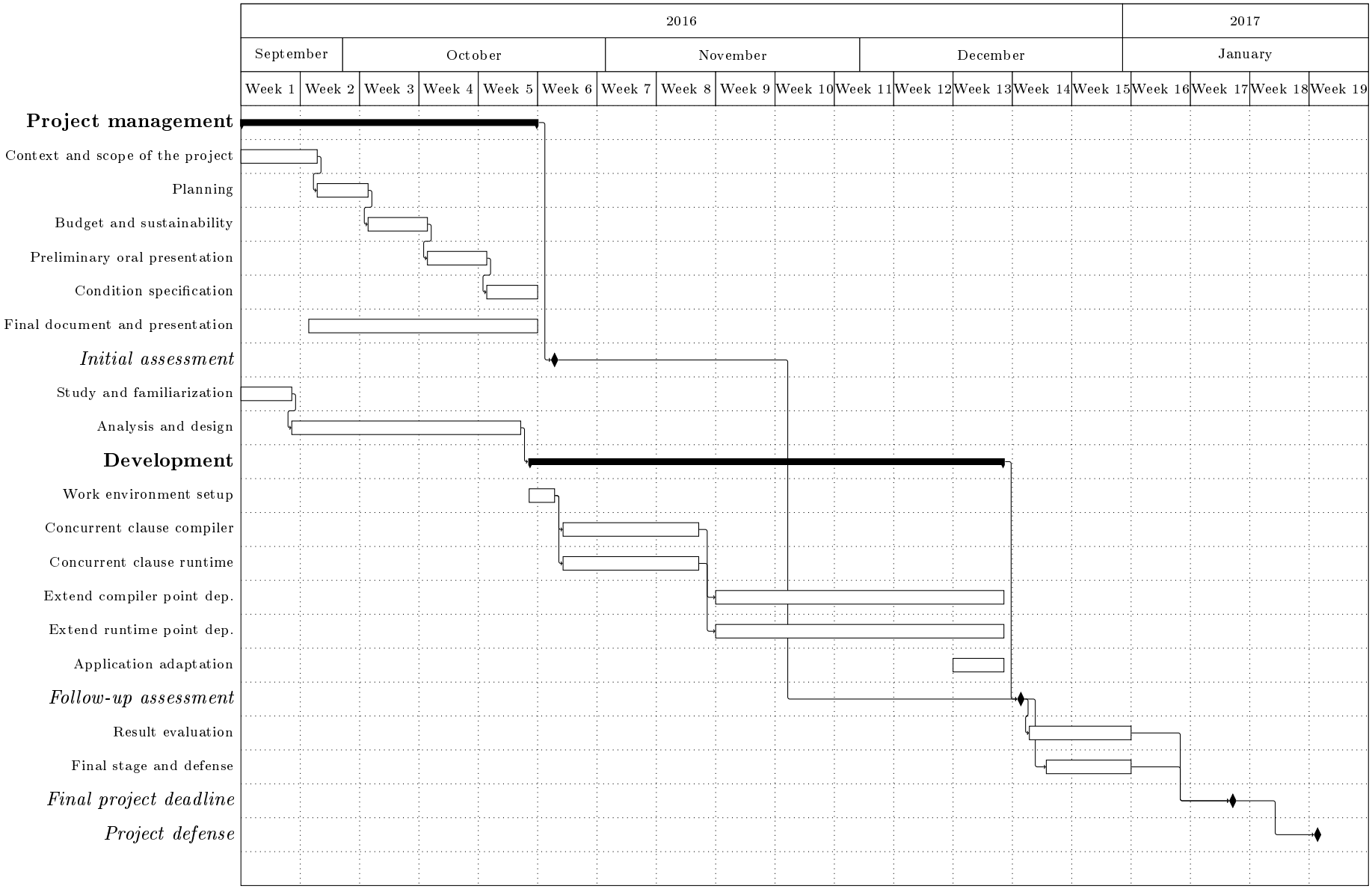


Figure 43: Revised Gantt chart, showing dependences and ordering between tasks

13.4 Costs revision

The increment of 15 hours of development time are distributed among the described roles as shown in table 13.

Role	**Salary (€/hour)**	**Dedication (hours)**	**Cost (€)**
Project manager	50	1.0	50
Analyst	35	3.5	122
Programmer	30	7.0	210
Tester	20	3.5	70
Total	-	**15**	**453**

Table 13: Human resources additional costs estimation

When it comes to other resources like computers or software, as they where already being used in overlapping phases, no extra cost is to be accounted for.

The total deviation can be covered by the budget item reserved for situations such as this. The amount of money available in this concept consist of 1.382€, fully covering our deviation of 453€.

No further deviations from the budget are expected.

13.5 Methodology revision

The *Agile* iterative development approach *eXtreme Programming* was chosen during the initial assessment of the project.

Throughout the course of the project, weekly meetings have been arranged with the project supervisors, fulfilling the initial plan of having one-week long development iterations. This has proven to be a good measure in order to have the project properly oriented at every moment.

In particular, it has helped detecting the described deviations at the soonest, minimizing them as much as possible while keeping in mind the final goals.

Having a cyclic methodology also took an important paper when we faced the maintenance tasks for some of the machines. If we had used a linear non-*Agile* methodology, we could not have rescheduled our tasks at that point, making the deviation much more worrying.

Test-driven development and code-revision sessions helped us ensure no errors where introduced while developing our features, and therefore avoid having to face more deviations for that reason.

All in all, the chosen methodology has proven to be adequate to this project.

13.6 Applicable laws and regulations

The only regulations that apply to a project like ours are the ones related to the intellectual property of the product being developed. *OmpSs* itself is an open standard freely available to use and edit. In particular, the *Mercurium* compiler is released under the GNU General Public License version 3 (GNU GPLv3).[15] Regarding the *Nanos6* runtime support library, it is not yet decided under which license it will be released, being the project still in its development phase.

In any case, and as far as this project is concerned, anything developed for it fully belongs to the *BSC-CNS* organization, and the developer of the project renounces any further rights over the intellectual property of the developed product.

14 References

[1] Facultat d'Informàtica de Barcelona. *Normativa TFG*. Mar. 2014. URL: http://www.fib.upc.edu/fib/estudiar-enginyeria-informatica/treball-final-grau/mainColumnParagraphs/0/document/NormativaTFG-GEI%20(document%20final).pdf (visited on 09/26/2016).

[2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface*. Nov. 2015. URL: http://www.openmp.org/mp-documents/openmp-4.5.pdf (visited on 09/27/2016).

[3] OpenMP Architecture Review Board. *OpenMP FAQ*. Nov. 2013. URL: http://www.openmp.org/about/openmp-faq/#OMPAPI (visited on 09/27/2016).

[4] OpenMP Architecture Review Board. *OpenMP Technical Report 4: Version 5.0 Preview 1*. Nov. 2016. URL: http://www.openmp.org/wp-content/uploads/openmp-tr4.pdf (visited on 11/25/2016).

[5] *Boletín oficial del estado*. num. 49, article 25. Feb. 2015. URL: https://www.boe.es/boe/dias/2015/02/26/pdfs/BOE-A-2015-1989.pdf (visited on 10/08/2016).

[6] Bevin (Intel) Brett. *Multi-Channel DRAM and High-Bandwidth Memory*. June 2016. URL: https://software.intel.com/en-us/articles/multi-channel-dram-mcdram-and-high-bandwidth-memory-hbm (visited on 01/04/2017).

[7] BSC-CNS. *Introduction to OmpSs*. URL: https://pm.bsc.es/ompss-docs/specs/01_introduction.html (visited on 09/26/2016).

[8] BSC-CNS. *Mercurium*. URL: https://pm.bsc.es/mcxx (visited on 09/26/2016).

[9] BSC-CNS. *The OmpSs Programming Model*. URL: https://pm.bsc.es/ompss (visited on 09/26/2016).

[10] Cavium. *ThunderX ARM Processors*. 2016. URL: http://www.cavium.com/ThunderX_ARM_Processors.html (visited on 01/04/2017).

[11] Jan Ciesko et al. *Towards Task-Parallel Reductions in OpenMP*. Springer International Publishing, 2015.

[12] CppReference. *Reference declaration*. Dec. 2015. URL: http://en.cppreference.com/w/cpp/language/reference (visited on 09/27/2016).

[13] Neil (Intel) Faiman. *An Introduction to Cilk Plus Reducers*. Feb. 2013. URL: https://software.intel.com/en-us/blogs/2013/02/26/an-introduction-to-cilk-plus-reducers (visited on 12/15/2016).

[14] *Git*. URL: https://git-scm.com (visited on 09/28/2016).

[15] GNU. *General Public License version 3*. June 2007. URL: https://www.gnu.org/licenses/gpl-3.0.en.html (visited on 12/14/2016).

[16] IBM. *IBM Power Systems S822LC Technical Overview and Introduction*. Dec. 2015. URL: http://www.redbooks.ibm.com/redpapers/pdfs/redp5283.pdf (visited on 01/07/2017).

[17] Intel. *An Introduction to the Intel QuickPath Interconnect*. Jan. 2009. URL: http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf (visited on 01/07/2017).

[18] Intel. *CilkPlus*. URL: https://www.cilkplus.org/ (visited on 12/15/2016).

[19] Intel. *Tutorial Cilk Plus Reducers*. URL: https://www.cilkplus.org/tutorial-cilk-plus-reducers (visited on 12/15/2016).

[20] Intel. *Tutorial Cilk Plus Terms*. URL: https://www.cilkplus.org/tutorial-terms (visited on 12/15/2016).

[21] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *A Pattern Language for Parallel Programming*. Addison Wesley Software Patterns Series, 2004.

[22] Timothy. G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Reduction Design Pattern*. Sept. 2005. URL: http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/SupportingStructures/Reduction.htm (visited on 09/30/2016).

[23] BSC Programming Models. *OmpSs Specification Documentation*. Section 2.6, page 9. Sept. 2016. URL: https://pm.bsc.es/ompss-docs/specs/OmpSsSpecification.pdf (visited on 09/27/2016).

[24] PRACE. *PRACE Resources*. URL: http://www.prace-ri.eu/prace-resources (visited on 01/07/2017).

[25] *Programming Languages - C++*. Section 3.10, page 74. ISO/IEC, May 2013. URL: https://isocpp.org/files/papers/N3690.pdf (visited on 09/27/2016).

[26] Rok Sosic and Jun Gu. "A Polynomial Time Algorithm for the N-Queens Problem". In: *SIGART Bull.* 1.3 (Oct. 1990), pp. 7–11. ISSN: 0163-5719. DOI: 10.1145/101340.101343. URL: http://doi.acm.org/10.1145/101340.101343 (visited on 12/20/2016).

[27] StudyinEurope. *The ECTS System*. 2016. URL: http://www.studyineurope.eu/ects-system (visited on 09/26/2016).

[28] Agencia Tributaria. *Tabla de coeficientes de amortización lineal*. Jan. 2015. URL: http://www.agenciatributaria.es/AEAT.internet/Inicio/_Segmentos_/Empresas_y_profesionales/Empresas/Impuesto_sobre_Sociedades/Periodos_impositivos_a_partir_de_1_1_2015/Base_imponible/Amortizacion/Tabla_de_coeficientes_de_amortizacion_lineal_.shtml (visited on 10/08/2016).

[29] Oreste Villa et al. *Effects of Floating-Point non-Associativity on Numerical Computations on Massively Multithreaded Systems*. Proceedings of Cray User Group Meeting (CUG), May 2009.

[30] H. W. Watson and Francis Galton. "On the Probability of the Extinction of Families." In: *The Journal of the Anthropological Institute of Great Britain and Ireland* 4 (1875), pp. 138–144. ISSN: 09595295. URL: http://www.jstor.org/stable/2841222 (visited on 12/20/2016).

[31] Wikipedia. *Eight queens puzzle*. Dec. 2016. URL: https://en.wikipedia.org/wiki/Eight_queens_puzzle (visited on 12/20/2016).

[32] Wikipedia. *Eight queens puzzle*. Nov. 2016. URL: https://en.wikipedia.org/wiki/Dot_product (visited on 12/20/2016).

A Additional results

In this section we can find the additional results mentioned in section 10.4.

If not explicitly shown otherwise in the figures, the parametrization for the execution of the benchmarks is the following:

- ***n*-queens:** N = 16, final depth = 2, 2500 tasks
- **UTS:** Non-leaf probability = 0.499, computation granularity (task weight) = 1000 SHA-1 operations per tree node, 1.300.000 tasks
- **dot product:** N = 2^{20} Kelements, block size = 2^{11} Kelements, 512 tasks

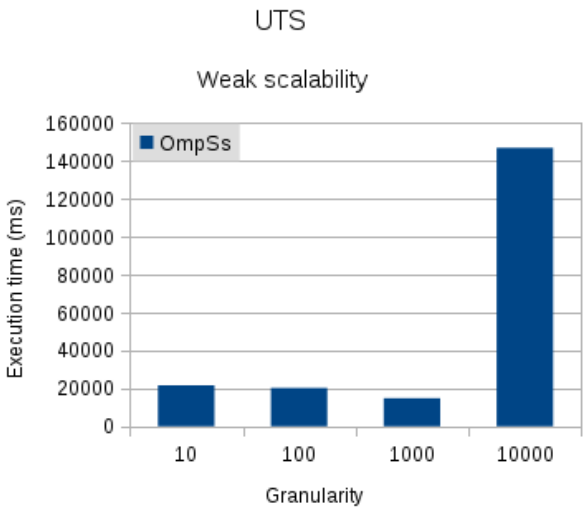


Figure 44: UTS granularity study in KNL

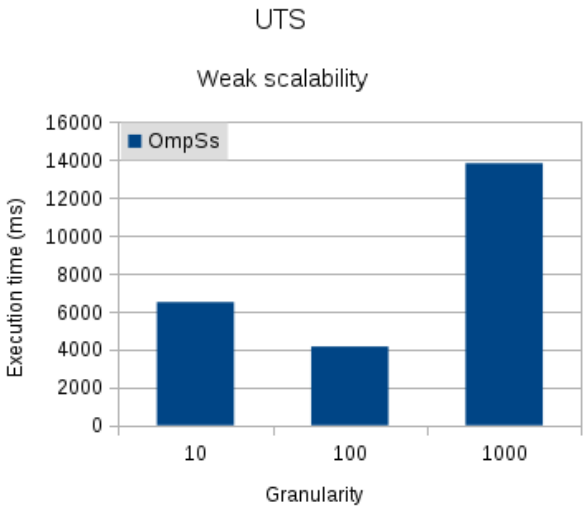


Figure 45: UTS granularity study in *Power8*

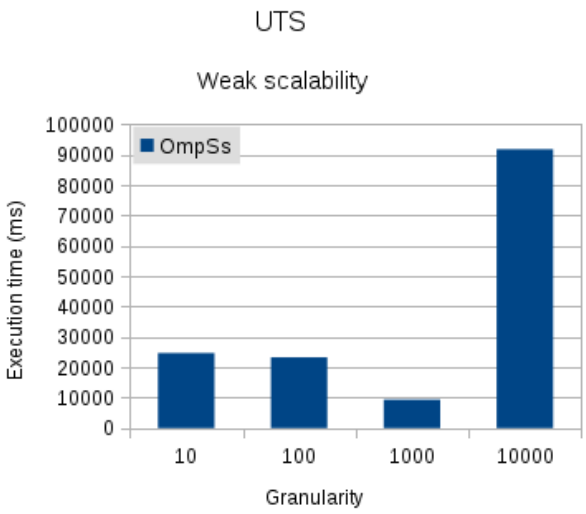


Figure 46: UTS granularity study in *ThunderX*

B Machine node topologies

This section shows the topology diagrams of the machine nodes used in the benchmark. The diagrams were extracted directly from the exact nodes by using the tool *lstopo* from the *hwloc* software suite.

Figures 47 48 50 49 show the extracted diagrams.

Note that no information about the caches is given in the *ThunderX* cluster node. This is due to a recognition/labeling error at the operating system level.

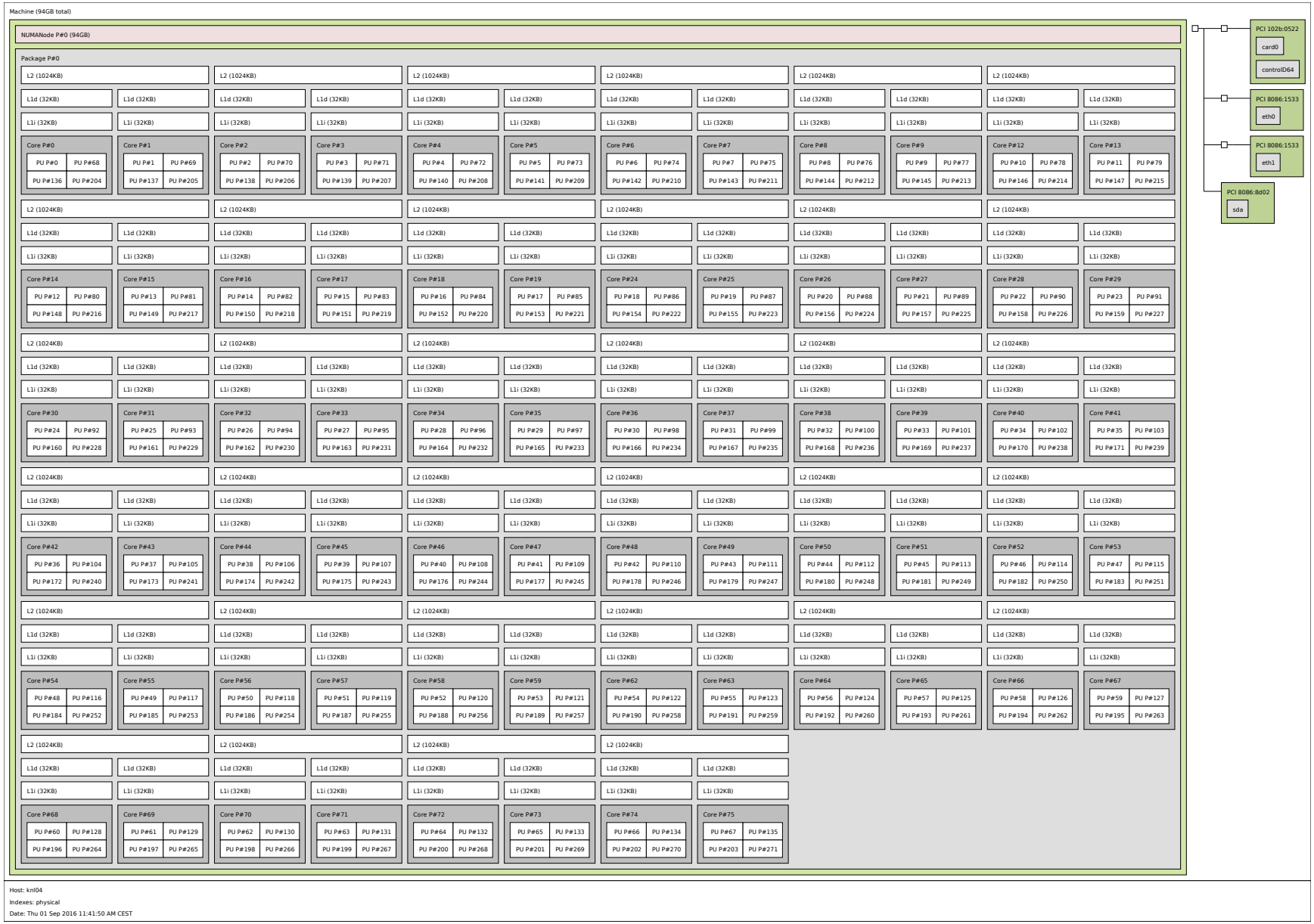


Figure 47: KNL node topology

Figure 48: *Power8* node topology

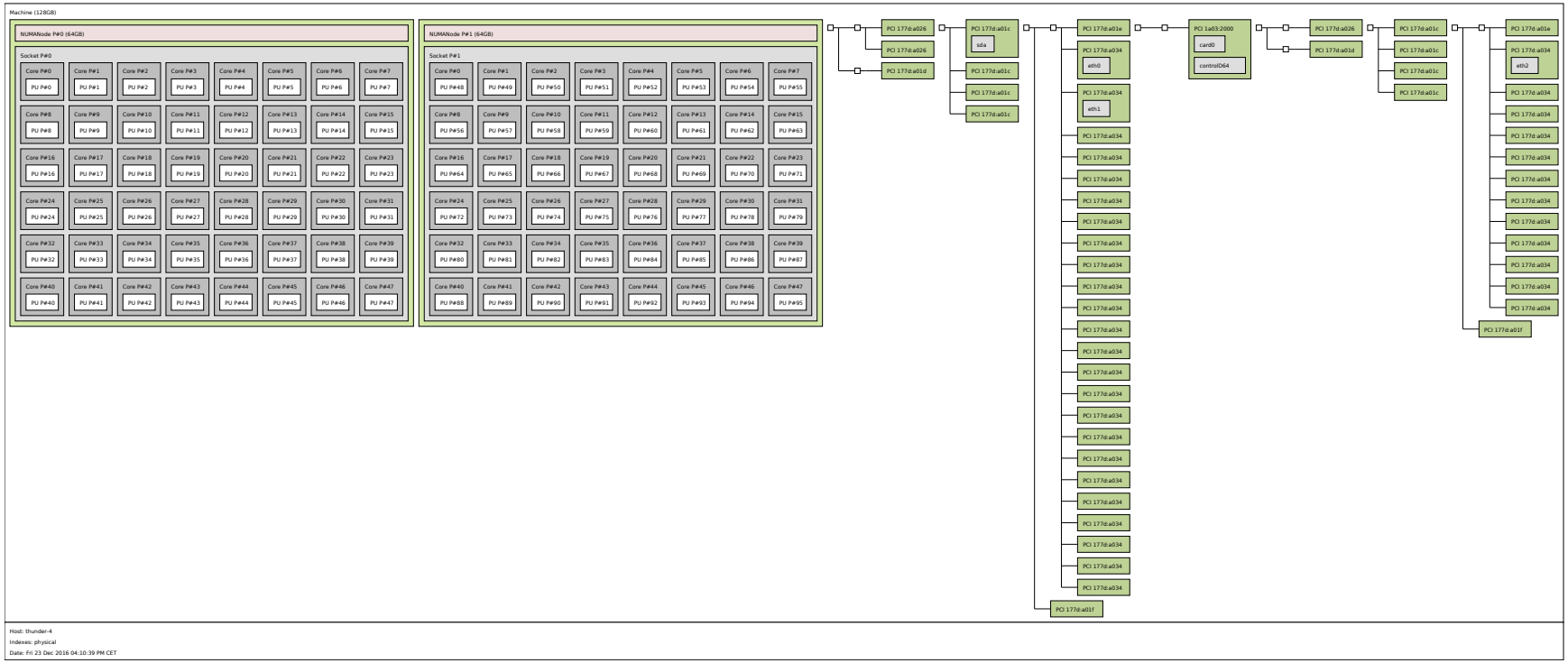


Figure 49: *ThunderX* node topology

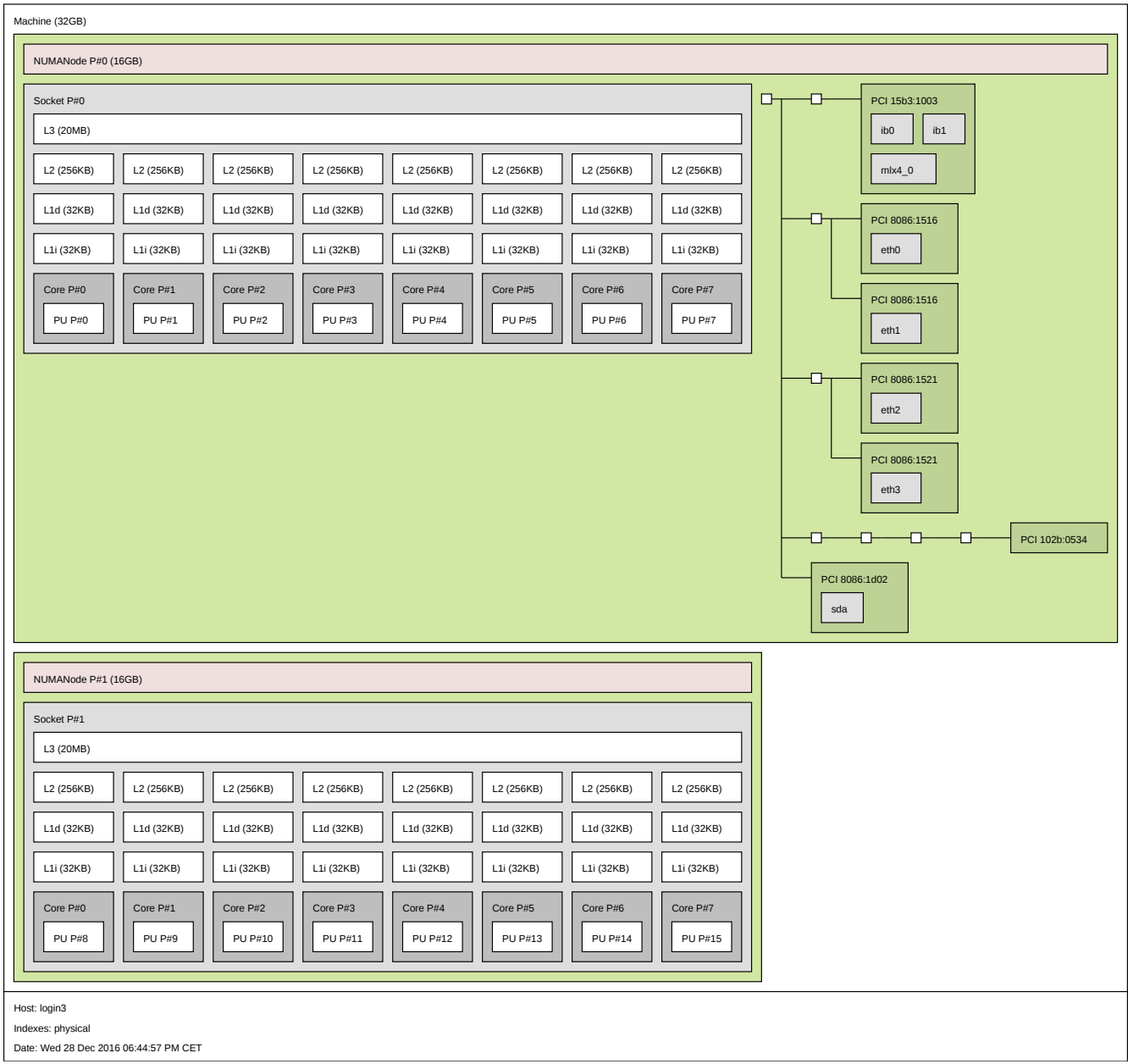


Figure 50: *MareNostrum III* node topology