Run-time support for multi-level disjoint memory address spaces

Javier Bueno Hedo

Advisor: Xavier Martorell Bofill
Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

Thesis submitted in fulfillment of the requirements of the degree:
Doctor per la Universitat Politècnica de Catalunya

July 2015

Abstract

High Performance Computing (HPC) systems have become widely used tools in many industry areas and research fields. Research to produce more powerful and efficient systems has grown in par with their popularity. As a consequence, the complexity of modern HPC architectures has increased in order to provide systems with the highest levels of performance. This increased complexity has also affected the way HPC systems are programmed. HPC users have to deal with new devices, languages and tools, and this is can be a significant access barrier to people that do not have a deep knowledge in computer science.

On par with the evolution of HPC systems, programming models have also evolved to ease the task of developing applications for these machines. Two well-known examples are OpenMP and MPI. The former can be used in shared memory systems and is praised for offering an easy methodology of software development. The latter is more popular because it targets distributed environments but it is considered burdensome to use. Besides these two, many programming models have emerged to propose new methodologies or to handle new hardware devices. One of these models is OmpSs.

OmpSs is a programming model for modern HPC systems that is based on OpenMP and StarSs. Developed by the Programming Models group at the Barcelona Supercomputing Center, it targets the latest generation of HPC systems while benefiting from the ease of use of OpenMP. OmpSs offers asynchronous parallelism with the concept of *tasks* with data dependencies. These tasks allow the specification of sections of code that can be executed in parallel while the dependencies specify the restrictions about the order in which the tasks can be executed. With this, OmpSs programs can adapt to a many different system configurations while fundamentally still being sequential programs with annotations.

This thesis explores the benefits of providing OmpSs the capability to target architectures

with complex memory hierarchies. An example of such systems can be the new generation of clusters that use accelerators to power their computing capabilities. The memory hierarchy of these machines is composed of a first level of distributed memory formed by the memory of each individual node, and a second level formed by the private memory of each accelerator devices. We propose a reference implementation that enables OmpSs programs to run on a cluster with or without accelerators while also providing a competitive performance when compared with other programming models.

We also discuss the enhancement of the OmpSs programming model with the support of non-contiguous regions of data. Offering this feature allows applications with complex data accesses to be easily annotated with OmpSs. This is important to widen the spectrum of applications that can be handled by the programming model. We present an implementation and evaluation of the performance and programmability impact of supporting non-contiguous memory regions.

Acknowledgments

I would like to thank Dr. Xavier Martorell for giving me the chance of pursue a doctorate at the Departament d'Arquitectura de Computadors at UPC, and tutoring this work. Also special thanks to Dr. Alejandro Duran who also provided invaluable guidance to develop this thesis, undoubtedly this work would not have possible without his support. This work was supported by the European Commission through the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the HiPEAC-3 Network of Excellence (ICT FP7 NoE 287759) the ENCORE project (FP7-248647), the TERAFLUX project (FP7-249013) and the TEXT project (IST-2007-261580), the Spanish Ministry of Education (TIN2007-60625, TIN2012-34557, and CSD2007-00050) and the Generalitat de Catalunya (2009-SGR-980).

I consider an important part of this thesis the people I have had the pleasure to meet while working at the Barcelona Supercomputing Center. Big thanks to Roger Ferrer, Xavier Teruel, Jordi Vaquero, Alberto Miranda, Jairo Balart, David Ródenas, Juanjo Costa, Marta García, Judit Planas, Guillermo Miranda, Sergi Mateo, Victor López, Sara Royuela, Claudia Rosas, Jan Ciesko, Diego Caballero, Jorge Bellón, Marçal Solà, Diego Nieto, Marcos Maroñas and many others who I have crossed paths with. Thanks for sharing many moments that made this thesis such a way easier and enjoyable experience. Definitely meeting you is one of the most valuable things I have got from doing this work.

I also want to thank my family for all the support they have given to me. My parents have worked hard during their lives to ensure that I received a good education. Their effort is also part of this thesis.

Finally, special thanks go to my wife, Sonia, who has been patient and supportive during these years. She excels in making me happy every day which compensates some of the feelings that were caused by this thesis. Thank you.

Contents

List of Figures

1

Introduction

1.1 Motivation

General purpose CPUs have reached a performance level where it is difficult to obtain more performance from a single unit. This has affected the way high-performance computers are organized, shifting to designs with a higher number of CPUs, since relying on future, more powerful, designs is no longer a viable option. This trend has brought new challenges in how processors are interconnected in order to allow systems to achieve a good performance at a reasonable cost.

Multiprocessor systems can be typically classified into two architecture types depending on how they are interconnected with the system memory. *Shared memory* systems are those where all CPUs in the system can access the whole available memory. The other type, *Distributed memory* systems, are those where each CPU can access only its own private memory, and an extra communication subsystem is needed to interconnect all the CPUs of the computer.

In the last decade, distributed memory computers have become the design of choice of the most powerful computers. The main reason is that the technology required to implement a shared memory architecture is not capable of efficiently handling several thousands of CPUs while offering a decent performance. Distributed memory systems have kept up with the challenge and cluster systems have been capable of managing thousands of CPUs at a reasonable cost. The effects of this can be seen in Figure 1.1. The chart shows the

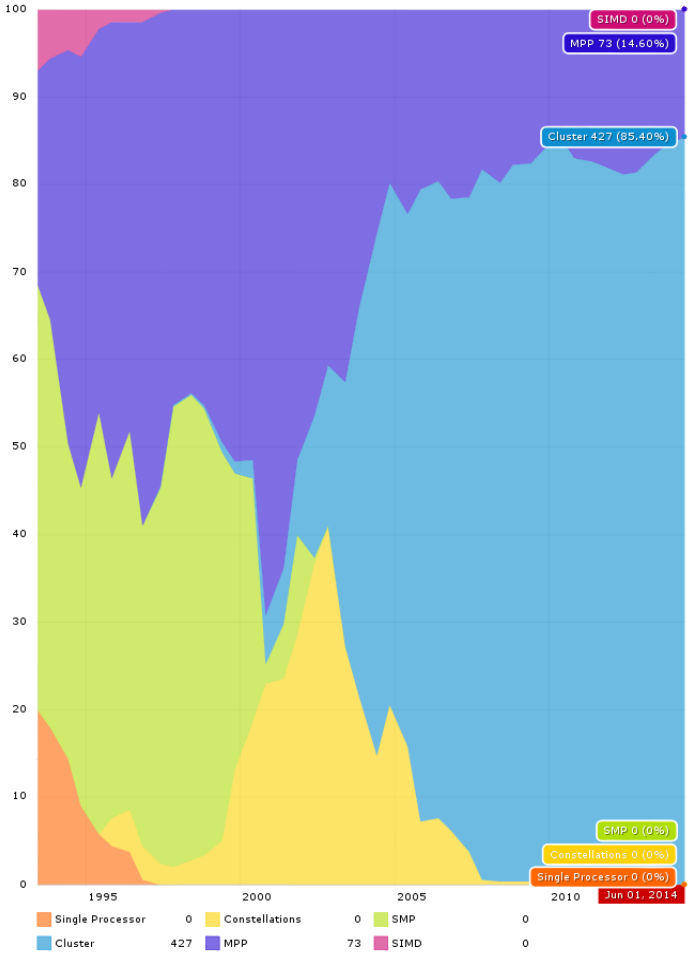


Figure 1.1: Top 500 Supercomputers types over the time as of June 2014

evolution over the years of the architecture type of the Top 500 supercomputers. Shared memory computers represented a 50 percent of the population until 1999, moment when the distributed memory computers became more cost efficient.

In addition, during the last years a new trend has emerged to increase the computation capacity of supercomputers. System designers have started to include accelerators that can be used by applications to speed up some of its parts. The Cell Broadband Engine (Cell/B.E.) [25, 49, 55] was one of the first modern processors to include accelerators. These accelerators used a different instruction set than the main processor, each one had its own private memory and they were specialized in running vector code. More recently, NVIDIA® graphics processors (GPUs) [60, 52] are also being added to supercomputers to fulfill a similar role

as the accelerators of the Cell. Clusters with GPUs have shown a higher peak performance while costing less than CPU clusters [42]. Still, the usage of GPU clusters in a production environment faces new challenges that have to be considered [59].

Accelerators have increased the efficiency of supercomputers in terms of energy consumed and cost. They can do more operations than a regular CPU while consuming less energy, and the overall cost per performance is also lower. Because of this there is active research to develop new kinds of accelerators. Field-Programmable Gate Array (FPGA) devices are a promising technology to implement accelerators. They are formed by reconfigurable hardware that can be set up to solve specific problems. Current research is trying to demonstrate the viability of these devices in the HPC field [33, 53]. Also new commercial products have been launched. The Intel® Xeon Phi™[26] is probably one of the most successful as currently is present in some of the most powerful supercomputers.

Figure 1.2 illustrates the relevance of accelerators in modern supercomputers [74]. The table shows the five most powerful supercomputers in the world and the top two run accelerators. The first one uses the Intel® Xeon Phi™accelerator and the second one uses NVIDIA® K20x GPUs. In addition, in the top ten we can find two more systems with accelerators. The trend can also be observed in the top 100 where 27 systems use accelerators, 17 of them are NVIDIA® GPUs and 10 Intel® Xeon Phi™. This demonstrates that accelerators have become key components of current HPC systems, and their usage is not expected to decline during the upcoming years.

This evolution has also affected the software side of the High Performance Computing (HPC) world, since languages and tools have had to deal with these architectural changes. This thesis tries to address some of the challenges that the software components have to face in order to adapt to the latest hardware trends previously exposed.

1.1.1 Programming models for High Performance Computing

The ability to express and manage parallelism has been the key concept of programming models and tools targeting HPC environments. The main mission of those is to maximize the productivity of the programmer of such systems. Productivity can be quantified as a measure of how much effort is put into the development cycle of applications and the performance achieved by them.

As with the hardware systems, parallel programming models can be classified in the same two big categories: programming models for shared memory systems and programming models for distributed memory.

Rank	Site	System	Cores	Rmax (TFlops/s)	Rpeak (TFlops/s)	Power (kW)
1	National Super Computer Center in Guangzhou *China*	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P *NUDT*	3120000	33862.7	54902.4	17808
2	DOE/SC/Oak Ridge National Laboratory *United States*	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x *Cray Inc.*	560640	17590.0	27112.5	8209
3	DOE/NNSA/LLNL *United States*	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom *IBM*	1572864	17173.2	20132.7	7890
4	RIKEN Advanced Institute for Computational Science (AICS) *Japan*	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect *Fujitsu*	705024	10510.0	11280.4	12660
5	DOE/SC/Argonne National Laboratory *United States*	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom *IBM*	786432	8586.6	10066.3	3945

Figure 1.2: Top 5 supercomputers as of June 2014

Programming models for shared memory architectures are generally considered to be easy to use since program data can be accessed by all CPUs on the system. This is more intuitive when the parallel algorithm is being adapted from a sequential one.

By contrast, programming models for distributed memory architectures require an extra effort to develop parallel applications because algorithms must be adapted to fit in a distributed memory architecture. This process is error prone and usually requires a higher grade of knowledge about the computer programming concepts. The advantage of distributed memory programming models is that they are capable of taking advantage of distributed memory systems, which offer more performance and nowadays are more popular than shared memory systems. MPI [75] is an example of such models. In MPI, communication is explicitly stated in the program code and data distribution and synchronization must be manually handled by the programmer.

Still, some shared memory programming models have offered many ideas and design principles that have been well considered. Many of these ideas are related to the ease of use and the capability to shorten parallel application development time. The use of annotations in a sequential code to allow the compiler to produce a parallel program is an idea that OpenMP [35] popularized on the shared memory scenario.

Other programming models have been designed with the idea of offering an environment that resembles a shared memory architecture, but targets distributed systems. Partitioned Global Address Space (PGAS) languages like Split-C [34], UPC [29] or Co-Array FORTRAN [79] are examples of this.

Another approach to make shared memory programming models available to distributed memory systems is the use of Software Distributed Shared Memory (SDSM) systems, software components capable of offering a virtual shared memory environment on top of a distributed system. With such systems tools and applications that were originally developed for shared memory environments can run without any modification on a distributed machine. This usually has an important impact on application performance, and its successful usage greatly depends on the application characteristics. Cluster OpenMP [54] is one of these systems and it was distributed by Intel®.

In the later years, parallel programming models have evolved in their way to offer new forms of expressing parallelism. Originally, programming models were only capable of parallelizing regular code structures (mainly loops). Recently, programming models have been extended or new ones have emerged to support irregular parallelism (recursive calls, complex data structures) in a more generic way. Examples of these are Cilk [14], the new task model of OpenMP 3.0 [83], or the new Asynchronous PGAS (APGAS) programming models, like X10 [23] or Chapel [21].

Also, to deal with the new heterogeneous systems, new programming technologies like CUDA [81] and OpenCL [58] have been integrated into the existing programming models, or have also been the starting point to create new tools. However, CUDA and OpenCL face the same problems as MPI, they are very explicit APIs and the developer requires a deep knowledge about the GPU architecture. With all, programming with them has a very significant impact on the application code. As with MPI, many tools have started to emerge in order to try to hide those complexities from the end programmer.

1.2 Objective of the thesis

Programming models supporting distributed and heterogeneous systems face many challenges in order to accomplish the mission of being productive tools that offer both ease of use and a decent performance. One of the biggest problems is handling the memory organization of such systems. Memory is not only distributed across a set of nodes, but also these nodes may contain additional resources with independent memories. Partitioning and adapting sequential code to these systems is not a trivial task, and it requires a deep knowl-

edge on the underlying hardware. Besides the cost of application development in terms of writing, testing and debugging the code, many issues may appear that affect the performance of the final program. A badly distributed data may lead to bottlenecks in several parts of the memory hierarchy, which may degrade the performance dramatically.

We believe that using a programming model that hides the memory management from the application code is critical to provide an easy and fast environment for developing applications. With such a model, many aspects that can have a significant impact on performance can be delegated to the run-time environment, which can contain the logic to take better decisions at run-time than the programmer at develop time. In addition, the run-time can be configurable in order to also allow the developer to provide additional information to be used when the application runs.

This thesis proposes to develop support for distributed and multi-level memory hierarchies for the OmpSs programming model. OmpSs is an annotation based programming model that tries to hide many of these complexities from the programmer, and places many critical decisions, that may affect performance, on the underlying run-time environment. The OmpSs run-time environment is composed of Nanos++, the run-time library, and the Mercurium C/C++ source to source compiler. At the time of starting this project, Nanos++ supported only two system architectures, SMP systems and GPU systems.

This task will be done by extending the original implementation of the Nanos++ run-time library. The goal is to provide support for distributed and heterogeneous architectures. We expect that all changes will be kept inside the run-time library and the compiler, thus applications will not be affected and could benefit from the new features without any significant changes.

OmpSs applications running on top of this environment must also be capable of delivering a reasonable performance, therefore, evaluating our implementation will also be an important task to do during the thesis. Specific optimizations may be developed if performance issues are detected during the evaluation.

1.3 Thesis contributions

This thesis presents the implementation of the cluster support for the Nanos++ library. It enables OmpSs applications to run on a distributed environment with minimal changes to the source code. This implementation has gone through an iterative process of enhancements and additions in order to improve the performance of applications and to add features required by the applications.

The first iteration produced a basic implementation that supported distributed memory architectures. OmpSs applications were capable of running transparently on a cluster system and Nanos++ handled the required data transfers automatically. Multi-level parallelism was also supported. A few optimizations were developed during this stage: a scheduling policy, named *affinity*, that tries to execute the program tasks where their data is located, and the addition of emphslave-to-slave transfers. Both techniques are transparent to OmpSs applications and aim to minimize the amount of network activity during the execution of the application. The performance of this version of the OmpSs run-time environment was evaluated with a set of benchmarks. We also compared the results achieved with the performance of the same benchmarks developed with MPI, a well-known programming model for distributed systems. This work has been described on the following publication:

[17] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta. **Productive cluster programming with OmpSs**. In *Proceedings of the Euro-Par 2011 Parallel Processing*, pages 555-566, Springer, 2011.

As clusters are ubiquitous in nowadays HPC environments, new designs include accelerators in order to provide more computational capabilities. Part of the burden of programming accelerators comes from dealing with the data distribution and movement, a problem that is similar to programming for distributed environments, therefore it made sense to extend our initial implementation to handle this new kind of clusters. Nanos++ originally supported CUDA devices, this support was integrated with the cluster support implemented initially in order to allow GPU applications to be able to run on clusters of GPUs. Again this support was completely transparent and OmpSs applications did not require modifications to be able to use it. A few optimizations were also needed to achieve a good efficiency. The *affinity* scheduling policy was updated to take GPUs into consideration. We implemented a task pre-send mechanism and data forwarding mechanism in order to overlap communication with computation automatically. This new features were evaluated again with a set of GPU applications. In this evaluation we also compared OmpSs applications against their MPI+CUDA counterparts, and we also tried to measure the programmability of both programming models. The results were published in the following paper:

[18] Javier Bueno, Judit Planas, Alejandro Duran, Rosa M. Badia, Xavier Martorell, Eduard Ayguadé, Jesús Labarta. **Productive programming of GPU clusters with OmpSs**. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium*, pages

557-568, IEEE, 2012.

The initial OmpSs implementation had some limitations when it came to specifying complex regions of data. Nanos++ did not allow the specification of non-contiguous and overlapping data. These limitations, while not being critical, hindered the implementation of some applications and did not allow for some more efficient implementations of others. As a result, we implemented this support in the OmpSs run-time library and measured the benefits of having this feature in terms of productivity and performance. The following paper describes this work:

[16] Javier Bueno, Xavier Martorell, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta. **Implementing OmpSs support for regions of data in architectures with multiple address spaces**. In *Proceedings of the 27th international ACM conference on International Conference on Supercomputing*, pages 359-368, ACM, 2013.

The implementation of OmpSs for clusters has also been used by fellow colleagues to do other research projects. Jan Ciesko et al. have implemented a mechanism to improve the performance of reductions in distributed applications. They used the Nanos++ implementation developed in this thesis.

[27] Jan Ciesko, Javier Bueno, Nikola Puzovic, Alex Ramírez, Rosa M. Badia, Jesús Labarta. **Programmable and Scalable Reductions on Clusters**. In *Proceedings of the 27th International Parallel and Distributed Processing Symposium*, pages 560-568, IEEE, 2013.

1.4 Thesis organization

The rest of this document is organized as follows. Chapter 2 is dedicated to the OmpSs programming model, describing the most relevant concepts and showing examples of usage. Chapter 3 introduces the two OmpSs development and run-time environment components: the Mercurium source-to-source compiler and the Nanos++ library. The Chapter also shows the technical details about the design and implementation of the latter, as much of the work of this thesis has consisted in extending the original Nanos++ design. The first contribution of this thesis is the topic of Chapter 4. It describes the design and implementation of the cluster support for the Nanos++ library, which enables OmpSs applications to be run on distributed systems. The Chapter also includes a performance evaluation of the imple-

mented infrastructure and a comparison against another well-known programming model: MPI. Chapter 5 details the second contribution of the thesis, the extension of our work to support cluster systems with GPUs on each node. A performance evaluation and a comparison against another programming model is also presented. The third contribution of this work is explained in Chapter 6. It illustrates the limitations of the original Nanos++ design when dealing with non-contiguous data, presents the proposed design and implementation, and evaluates this solution. Finally Chapter 7 reviews the state of the art in the field of programming models for distributed an heterogeneous architectures, and Chapter 8 summarizes the main conclusions produced by this work.

2

The OmpSs programming model

This chapter introduces the OmpSs programming model. While the development of OmpSs is not a contribution of this thesis, providing an introduction to it is needed in order to understand the concepts and the context behind the contributions presented. The following sections provide a general description of the OmpSs programming model. Section 2.1 comments the general philosophy behind OmpSs. Section 2.2 describes how parallelism can be expressed with OmpSs. Section 2.3 explains the mechanisms available to synchronize the parallel parts of the application. Section 2.4 shows how the programmer can provide memory references where OmpSs requires it. Section 2.5 covers the details about how OmpSs can be used to develop applications on systems with multiple address spaces. Section 2.6 shows how OmpSs can also target heterogeneous systems. Finally section 2.7 covers the latest features that have been added to the programming model, and section 2.8 briefly comments the influence of OmpSs in the OpenMP programming model.

2.1 Introduction

OmpSs is a programming model composed of a set of directives and library routines that can be used in conjunction with a high level programming language [1] in order to develop parallel applications. The name *OmpSs* comes from the name of two other programming models,

[1] C, C++ and FORTRAN are currently supported.

OpenMP and *StarSs* [85, 87, 8, 12]. The design principles of these two programming models form the basic ideas used to conceive OmpSs.

The goal of OmpSs is to provide a *productive* environment to develop applications for modern High-Performance Computing (HPC) systems. Two concepts contribute to make OmpSs a *productive* programming model: performance and ease of use. By performance we understand that programs developed with OmpSs must be able to deliver a reasonable performance when compared to other programming models that target the same architectures. Ease of use is a concept difficult to quantify but OmpSs has been designed using principles that have been praised by their effectiveness in that area.

OmpSs takes from OpenMP its philosophy of providing a way to, starting from a sequential program, produce a parallel version of it by introducing annotations in the source code. These annotations do not have an explicit effect in the semantics of the program. Instead, they allow the compiler to produce a parallel version of it. This characteristic feature lets programmers parallelize applications incrementally. Starting from the sequential version of an application, the programmer can add new directives to parallelize different parts of the application. This methodology simplifies the development process, easing the search for errors and the performance analysis, which leads to an increase of the overall productivity provided by the programming model.

This methodology proposed by OmpSs contrasts with other approaches that have to be usually taken when using more explicit programming models. Generally these expose resources that have to be managed by the programmer in order to implement the parallelism. An example of this could be the MPI programming model, where part of the network logic must be coded into the application to send and receive messages that will allow the application to exploit the parallel resources of the system. This exposure of resources is typically dependent on the underlying technology, thus binding an specific implementation of a parallel application to a single technology. All of this results in a more complex development cycle where the code must be redesigned to fit the parallelism requirements. In addition this also increases the effort required to debug and test the code.

Star Superscalar (StarSs), is a family of programming models that also offer implicit parallelism through a set of compiler annotations. It differs from OpenMP in some important areas. StarSs uses a *thread-pool* execution model whereas OpenMP implements a *fork-join* model. StarSs also includes features to target heterogeneous architectures while OpenMP only targeted shared memory systems until its 4.0 version. Finally StarSs offers asynchronous parallelism as the main mechanism of expressing parallelism whereas OpenMP only started to implement it since its version 3.0.

StarSs raises the bar on how much implicitness is offered by the programming model. When programming using OpenMP, the developer first has to define which regions of the program will be executed on parallel, then he or she has to express how the inner code has to be executed by the threads forming the parallel regions, and finally it may be required to add directives to synchronize the different parts of the parallel execution. StarSs simplifies part of this process by providing an environment where parallelism is implicitly created from the beginning of the execution, thus the developer can omit the declaration of parallel regions. The definition of parallel code is based on the concept of *tasks*, which are pieces of code which can be executed asynchronously in parallel. When it comes to synchronizing the different parallel regions of a StarSs application, the programming model also offers a dependency mechanism which lets programmers annotate the program data that each individual *task* will use and produce. With this information, the programming model can generate dynamic constraints that contain the information about which *tasks* can be run in parallel and which ones require a specific order of execution. This mechanism allows to achieve a much finer grain of synchronization between parallel *tasks* than using explicit synchronization clauses, like barriers or locks, which leads to a more efficient use of the parallel resources of the system.

OmpSs tries to be the evolution that OpenMP needs in order to be able to target newer architectures. For this, OmpSs takes key features from OpenMP but also new ideas that have been developed in the StarSs family of programming models.

2.2 Expressing parallelism

OmpSs offers an execution model where an implicit parallel region is automatically created when the program starts. From there, the programmer can annotate pieces of code that will be run in parallel with the rest of the application.

2.2.1 Tasks

OmpSs allows the expression of parallelism through *tasks*. *Tasks* are independent pieces of code that can be executed by the parallel resources at run-time. Whenever the program flow reaches a section of code that has been declared as *task*, instead of executing the task code, the program will create an instance of the task and will delegate its execution to the OmpSs run-time environment, which will eventually execute the task on a processing element.

The programmer can specify a *task* using the `task` directive. This directive can appear

inside any code block of the program, which will mark the following statement as a *task*.

Figure 2.1 shows a sample code that declares a task inside a function code. When the control flow reaches line 5 and 7, a new task instance is created, however when the program reaches line 12, the previously created tasks may not have been executed yet by the OmpSs run-time. The directive in line 13 waits until all created tasks have been completed.

```
1   float x[3] = { 0.0,  1.0, 2.0};
2   float y[3] = { 0.5,  0.2, 0.1};
3   float z[3] = {22.2, -1.2, 4.5};
4   int main() {
5      #pragma omp task
6      do_computation(x);
7      #pragma omp task
8      {
9         do_computation(y);
10        do_computation(z);
11     }
12
13     #pragma omp taskwait
14     return 0;
15  }
```

Figure 2.1: Inline task example with OmpSs

The task directive can also be used in the declaration of a function, which transforms the function into a *task function*, meaning that each invocation of the given function will create a task instead of doing a normal function invocation. Figure 2.2 is an example of how task functions are used. Invocation of `do_computation_task` in line 8 will create an instance of a task.

```
1   extern void do_computation(float a);
2   #pragma omp task
3   extern void do_computation_task(float a);
4
5   float x = 0.0;
6   int main() {
7      do_computation(x); //regular function call
8      do_computation_task(x); //this will create a task
9      return 0;
10  }
```

Figure 2.2: Task function example with OmpSs

2.2.2 Loops

Another way to express parallelism in OmpSs is using the `for` directive. This directive has a direct counterpart in OpenMP and the OmpSs behavior is almost identical. It must be used in conjunction with a for loop (in C or C++) and it encapsulates the iterations of the given for loop into tasks. The number of tasks created is determined by the OmpSs run-time, however the user can specify the desired scheduling with the `schedule` clause. Figure 2.3 shows an example of a loop parallelized using the `for` directive. The number of tasks created will depend on the number of available processing elements. Also, according to the scheduling policy specified, the run-time environment will try to assign the same number of iterations to each task.

```
1  float x[N];
2  int main() {
3     #pragma omp for schedule(static)
4     for (int i = 0; i < N; i++) {
5        do_computation(x[i]);
6     }
7     return 0;
8  }
```

Figure 2.3: Parallelizing a loop with OmpSs

2.2.3 Multi-level parallelism

Tasks can also create new tasks. This allows the definition of multiple levels of parallelism, which can provide better application performance [6, 95, 70].

Whenever a task t_1 creates another task t_2, we say that t_2 is a *child* task of t_1 and t_1 is the *parent* task of t_2.

Actually all OmpSs applications use at least two levels of parallelism, since the sequential code can be considered as a task implicitly created when the application starts. Because of this, all tasks have a parent task that is either a user defined task or the implicit task.

The underlying OmpSs run-time environment can exploit different aspects of having multiple levels of parallelism to achieve a more optimal program execution. Also, multi-level parallelism is needed to allow the implementation of parallel recursive algorithms.

2.3 Synchronizing parallelism

Synchronizing the parallel tasks of the application is required in order to produce a correct execution, since usually tasks depend on data computed by other tasks. The OmpSs programming model offers two ways of synchronizing tasks: (i) data dependencies, and (ii) explicit directives to set synchronization points.

2.3.1 Dependencies

Tasks usually require data in order to do meaningful computation. Typically a task will use some input data that has been produced by preceding tasks to perform some calculations and produce new results that can later be used by other tasks or parts of the program.

OmpSs lets the programmer express this kind of relationships specifying the data that each task will use and how the task will access it, if it will be written, read or both. This information is added when declaring a task and is what we name as *data dependencies*.

The following clauses can be used along with the `task` directive to specify this data:

`in(memory-reference-list)` It specifies that the construct depends on some data which will be only read, and therefore, it is not eligible for execution until any previous construct with an `out` clause over the same data is completed.

`out(memory-reference-list)` It specifies that the construct will generate some data–only written–, and therefore, it is not eligible for execution until any previous construct with an `in` or `out` clause over the same data is completed.

`inout(memory-reference-list)` It specifies a combination of `in` and `out` over the same data, that is, the task will read and also update the referenced data.

All of these clauses receive a list of memory references as argument. The syntax permitted to specify memory references is described in Section 2.4.

When an OmpSs programs is being executed, the underlying run-time environment uses the data dependence information and the creation order of each task to perform dependence analysis. This analysis produces execution-order constraints between the different tasks which results in a correct order of execution for the application. We call these constraints *task dependencies*, and they are formed by pairs of tasks. There is a task dependence between t_1 and t_2 if the following properties are satisfied:

1. t_1 is created before t_2 and both have the same parent task.

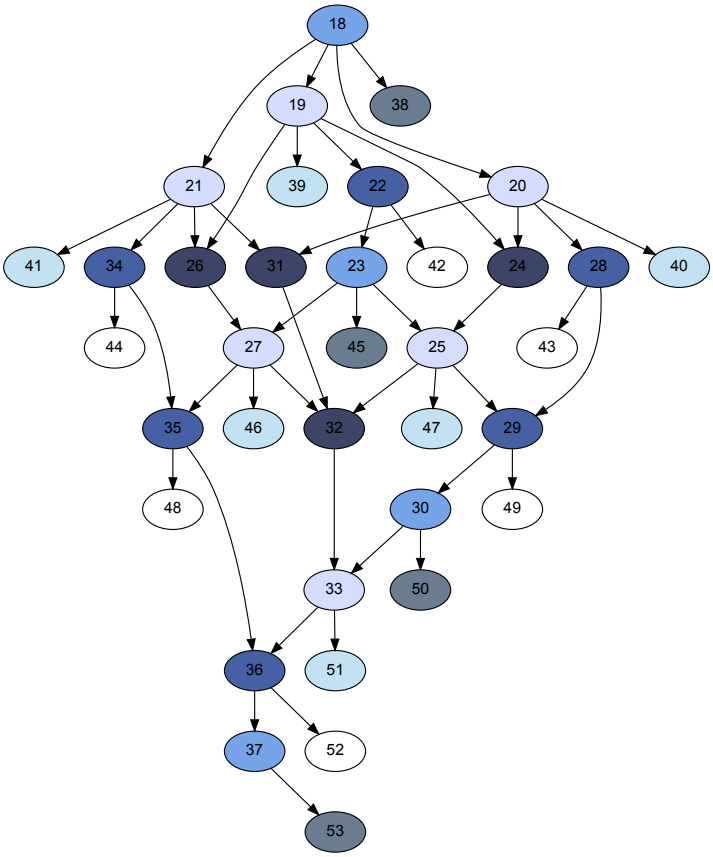
2. t_2 has a memory reference that overlaps with a memory reference of t_1 and at least one of these references is declared using an `out` clause.

There is a restriction on which data can be specified on a data dependence. A task can not specify references to memory that is not referenced as a data dependence by its parent task, and the access mode has to be consistent with the one specified by the parent. An example of this restrictions can be seen on figure 2.4. The task declared in line 6, the data dependence is invalid because the parent task, declared in line 3, does not reference the variable `y`. Also, the reference to `x` is incorrect because the parent task has specified that is an *input* data (it will only be read) but the child tasks have specified it as *inout*.

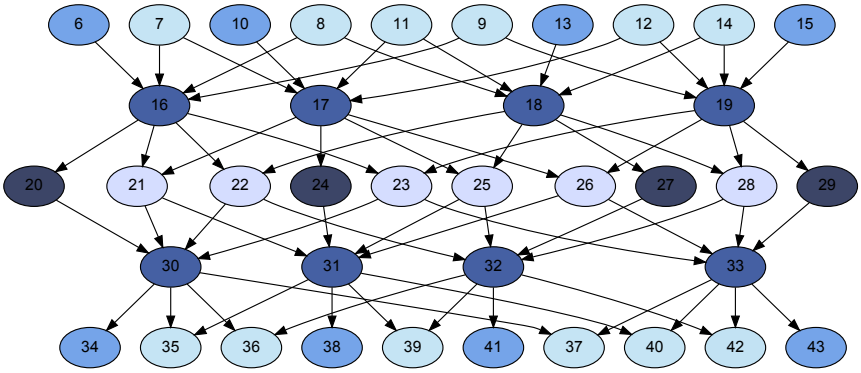
```
1  float x[10], y[10], z[10];
2  int main() {
3     #pragma omp task input(x) output(z)
4     {
5        ...
6        #pragma omp task inout(x, y)
7        {
8           ...
9        }
10    }
11 }
```

Figure 2.4: Invalid data dependency on a nested task.

Dependence analysis also exposes the parallelism between different tasks. OmpSs applications can be represented using a directed acyclic graph where nodes represent each created task and links represent task dependencies between two individual tasks. This graph reveals important information about the application, it exposes the parallelism available and shows how the tasks will be ordered for execution. Figure 2.5 shows two examples of task graphs that represent the parallelism of two different applications: a Cholesky decomposition (figure 2.5a) and a Fast Fourier Transform (figure 2.5b). Each node represents an OmpSs task, also each node contains the internal task identifier, which shows the order of creation of the tasks. The color of each task represents the program code that it executes. The directed links correspond to the data dependencies between tasks. The task graph corresponding to the Cholesky application reveals that, after a first stage where many tasks are created, the available parallelism decreases as the program progresses. In contrast, the FFT program shows a very regular parallelism in all stages of the execution.



(a) Task graph representing the Cholesky application



(b) Task graph representing the FFT application

Figure 2.5: Example task graphs, each node corresponds to an OmpSs task, the number is the internal identifier, the color identifies the program code, and the directed links show the data dependencies

Value dependencies

Sometimes a task needs to work with some values of data that are valid at the time of creating the task. A common example may be a piece of code where a loop creates some tasks and each of them require an induction variable, the first tasks needs the value 0, the second needs 1, and so on (figure 2.6). This induction variable only contains the desired value at the moment of creating each task, so using the `in` clause (figure 2.6a) would be incorrect because each task will read the value at the moment of execution, which is probably not the desired behavior in this scenario.

```
1  int main() {
2     for (int i=0; i<N; i++) {
3        #pragma omp task in(&i)
4        do_work(i);
5     }
6     return 0;
7  }
```

(a) `in` clause does not usually provide the desired behavior in this scenario.

```
1  int main() {
2     for (int i=0; i<N; i++) {
3        #pragma omp task firstprivate(i)
4        do_work(i);
5     }
6     return 0;
7  }
```

(b) `firstprivate` captures the value at the moment of creation.

Figure 2.6: Example of usage of `firstprivate`.

To handle this scenario, the clause `firstprivate` provides the desired behavior (figure 2.6b). It will capture the value of the specified data at creation time and the task will receive a private copy of it.

The specificaton of data in a *firstprivate* also interacts with the dependencies specified in other tasks through the `out` and `inout` clauses. In this case, however, the creation of a task that has a specification of `firstprivate` data is hold if there is already a created task that will generate the required values.

2.3.2 Explicit synchronization

In addition to the data dependencies mechanism, the programmer can set explicit synchronization points in the program code. These points are set using the `taskwait` directive. When the control flow reaches a synchronization point, it waits until all children tasks have completed.

OmpSs also offers synchronization points that wait until certain tasks are completed. These are set adding the `on(memory-reference-list)` clause to the previously men-

tioned `taskwait` clause. This clause receives a memory reference as argument, and it will make the synchronization point to wait until all tasks that produce the referenced data have completed their execution.

2.4 Referencing memory

Several OmpSs clauses take one or more memory references as an argument. This memory reference must be legal variables appearing in the program, and can be references to basic types or array types. However, OmpSs extends the syntax when specifying array types in order to be able to define partial regions over array variables. This syntax extension eases the declaration of tasks that access slices of arrays which is a common use case in some application domains, for example linear algebra applications.

When using the C/C++ programming language, valid memory references are those that are expressions that their evaluation results in a pointer type. A memory reference always provides a *base address* and a *object size*. When using the syntax to specify partial regions of arrays, the specified regions always fall into the address range defined by the base address and the object size.

Supposing that we have the following variable declarations:

```
1 int value;
2 int  value_ptr = &x;
3 float data[128];
```

The variable `value` can not be used as a memory reference because it is not a pointer type. Also, a reference using a dereferenced pointer (`*value_ptr` for instance) or a single array element (`data[2]`) can not be used as memory references as their evaluation does not result in a pointer type.

The pointer `value_ptr`, `&value`, or the base of the array `data` can be used as a valid memory references. The first two cases report an *object size* equal to the result of evaluating `sizeof(int)`, whereas the third reference reports an object size equal to `sizeof(float)*128`.

While referencing arrays, OmpSs extends the syntax of the array subscript to be able to specify sub-chunks of arrays. The extensions allow the following syntax constructs:

data[] An empty expression is considered to take all the elements of the array as the data that is referenced.

data[*start*:*end*] *start* and *end* represent a range of the elements that are referenced. Both elements are included in the range.

data[*start*;*size*] It specifies a range starting from index *start* and consisting of *size* number of elements.

There are no restrictions when using this syntax on multi-dimensional arrays. It must be noted that these extensions may end up referencing areas of memory that are not contiguous in memory when multi-dimensional arrays are involved, however this behavior is supported by OmpSs.

As an example of how these expressions are used consider a two-dimensional square matrix declared as `Array[N][N]`. The expression `Array[0;N/2][]` references the first half rows of it. If we want to reference a set of the columns instead, the expression could be `Array[][N-10:N-1]` used to reference the last 10 columns of it.

2.5 Disjoint address spaces

One of the most relevant features of OmpSs is to handle architectures with disjoint address spaces. By disjoint address spaces we refer to those architectures were the memory of the system is not contained in a single address space. Examples of these architectures would be distributed environments like clusters of SMPs or heterogeneous systems built around accelerators with private memory.

2.5.1 Single address space view

OmpSs hides the existence of other address spaces present on the system. Offering the single address space view fits the general OmpSs philosophy of freeing the user from having to explicitly expose the underlying system resources. Consequently, a single OmpSs program can be run on different system configurations without any modification.

In order to correctly support these systems, the programmer has to specify the data that each task will access. Usually this information is the same as the one provided by the data dependencies, but there are cases where there can be extra data needed by the task that is not declared in the dependencies specification (for example because the programmer knows it is a redundant dependence), so OmpSs differentiates between the two mechanisms.

2.5.2 Specifying task data

A set of directives allows to specify the data that a task will use. The OmpSs run-time is responsible for guaranteeing that this data will be available to the task code when its execution starts. Each directive also specifies the directionality of the data. The data specification

directives are the following:

`copy_in(`***expr***`)` The data specified must be available in the address space where the task is finally executed, and this data will be read only.

`copy_out(`***expr***`)` The data specified will be generated by the task in the address space where the task will be executed.

`copy_inout(`***expr***`)` The data specified must be available in the address space where the task runs, in addition, this data will be be updated with new values.

`copy_deps` Use the directionality specification clauses (`in/out/inout`) also as if they were `copy_[in/out/inout]` clauses.

The syntax accepted on each clause is the same as the one used to declare data dependencies.

Each data reference appearing on a task code must either be a local variable or a reference that has been specified inside one of the copy directives. Also, similar to the specification of data dependencies, the data referenced is also limited by the data specified by the parent task, and the access type must respect the access type of the data specified by the parent. The programmer can assume that the implicit task that represents the sequential part of the code has a read and write access to the whole memory, thus, any access specified in a user defined top level task is legal. Failure to respect these restrictions will cause an execution error. Some of these restrictions limit the usage of complex data structures with this mechanism, section 2.5.4 describes the problem with more detail.

The code in figure 2.7 shows an OmpSs program that creates two tasks: one that must be run on a regular CPU, which is marked as `target device(smp)`, and the second which must be run on a CUDA GPU, marked with `target device(cuda)`. This OmpSs program can run on different system configurations, with the only restriction of having at least one GPU available. For example, it can run on a SMP machine with one or more GPUs, or a cluster of SMPs with several GPUs on each node. OmpSs will internally do the required data transfers between any GPU or node of the system to ensure that each tasks receives the required data. Also, there are no references to these disjoint address spaces, data is always referenced using a single address space. This address space is usually referred to as the *host* address space.

```
1  float x[128];
2  float y[128];
3  int main() {
4     for ( int i = 0; i < N; i++ ) {
5        #pragma omp target device(smp)
6        #pragma omp task copy_inout(x)
7        do_computation_CPU(x);
8
9        #pragma omp target device(cuda)
10       #pragma omp task copy_inout(y)
11       do_computation_GPU(y);
12    }
13    return 0;
14 }
```

Figure 2.7: An OmpSs program with a SMP task and a GPU task.

2.5.3 Accessing children task data from the parent task

Data accessed by children tasks may not be accessible by the parent task code until a synchronization point is reached. This is so because the status of the data is undefined since the children tasks accessing the data may not have completed the execution and the corresponding internal data transfers. Figure 2.8 shows an example of this situation. The parent task is the implicitly created task and the child task is the single user defined task declared in line 4. The access to data on line 7 is illegal since the data may be still in use by the child task, the `taskwait` on line 10 is needed to guarantee that the access is legal.

```
1  float y[128];
2  int main() {
3     #pragma omp target device(cuda)
4     #pragma omp task copy_inout(y)
5     do_computation_GPU(y);
6
7     float value0 = y[64]; //illegal access
8                           //data not accessible
9
10    #pragma omp taskwait
11    float value1 = y[64]; //legal
12
13    return 0;
14 }
```

Figure 2.8: Illegal and legal accesses from the parent task.

Synchronization points, besides ensuring that the tasks have completed, also serve to synchronize the data generated by tasks in other address spaces, so modifications will be made visible to parent tasks.

However, there may be situations when this behavior is not desired, since the amount of data can be large and updating the host address space with the values computed in other address spaces may be a very expensive operation. To avoid this update, the clause `noflush` can be used in conjunction with the synchronization clause (`taskwait`). This will instruct OmpSs to create a synchronization point but will not synchronize the data in separate address spaces. Figure 2.9 contains an example of this situation. The access in line 8 may not produce the results computed by the previous task (line 5) since the `noflush` clause instructs the underlying run-time to not trigger data transfers from separate address spaces. The access in line 14 following a regular `taskwait` will access the computed values.

```
1  float y[128];
2  int main() {
3     #pragma omp target device(cuda)
4     #pragma omp task copy_inout(y)
5     do_computation_GPU(y);
6
7     #pragma omp taskwait noflush
8     float value0 = y[64]; //previous task has finished, but
9                           //'y' does not contain the data
10                          //computed by the previous task
11
12    #pragma omp taskwait
13    float value1 = y[64]; //contains the computed values
14
15    return 0;
16 }
```

Figure 2.9: Difference between using `noflush` or not on a `taskwait`.

The aforementioned `on` clause (section 2.3.2) can also be used to synchronize a specific piece of data. Figure 2.10 shows an example of this scenario, the value read in the access at line 9 corresponds to the value computed by the previous child task, however the access done on the previous line may not. The `taskwait on` at line 7 only synchronizes the data of the region of the array specified by the memory reference `y[32;64]`.

```
1  float y[128];
2  int main() {
3     #pragma omp target device(cuda)
4     #pragma omp task copy_inout(y)
5     do_computation_GPU(y);
6
7     #pragma omp taskwait on(y[32;64])
8     float value0 = y[0];  //not updated value
9     float value1 = y[64]; //this value has been updated
10
11    return 0;
12 }
```

Figure 2.10: Example usage of `taskwait on` to selectively synchronize data.

2.5.4 Limitations

The requirement of having to specify all memory references through `copy` clauses limits the data structures that can be effectively used within an OmpSs task that may run on a device with a disjoint address space.

Data structures that contain members that are references to memory are not well suited to be used as task data. A simple example to showcase this scenario can be seen on figure 2.11. In the example, OmpSs can handle the movement of the contents of `my_data`, however the data type contains two references that contain addresses pointing to main memory. These references will not be updated by OmpSs and are invalid within the context of the GPU. A workaround for this case could be adding both references in the `copy_inout` clause, which will solve the issue for this particular scenario.

The previous example was a simple case that could be fixed by manually adding the memory references. However sometimes the programmer may use opaque data structures which their implementations may be unknown. Examples of such data structures are the STL[94] containers. The exact implementation may vary across the different implementations and versions, and even though its technically possible to know the specific implementation of them it is not practical to program applications depending on it.

In some cases, knowing the specific implementation of a data type is not enough to overcome the OmpSs limitations. Some data structures can hold an arbitrary number of memory references that is unknown when the programmer is developing the application. An example of this is a linked list. Starting from the *head* element, each following element will contain a reference to the next. It is probably impossible to know how many references

```
1  struct my_data_type {
2     int value;
3     int  first_reference;
4     double  second_reference;
5  }
6  int main() {
7     struct my_data_type my_data;
8
9     my_data.value = 100;
10    my_data.first_reference = (int  ) malloc(128   sizeof(int));
11    my_data.second_reference = (double  ) malloc(64   sizeof(double));
12
13    #pragma omp target device(cuda)
14    #pragma omp task copy_inout(my_data)
15    do_computation_GPU(my_data);
16
17    #pragma omp taskwait
18
19    return 0;
20 }
```

Figure 2.11: Using a complex data structure within a `copy` clause.

a task accessing a linked list will require, so it is impossible to specify all of them when declaring a task.

2.6 Heterogeneity

Another key feature of OmpSs is the support for heterogeneous architectures. Originally OmpSs was designed with the CUDA GPU architecture as a reference for future heterogeneous systems, and currently OmpSs is capable of running applications that use GPUs, through the use of CUDA or OpenCL environments, and FPGAs. This thesis has added the support for clusters of SMPs and clusters of GPUs.

Tasks can be created for a specific device using the clause `target device`. This construct takes an argument that specifies the target architecture. Tasks created for a specific architecture require the presence of the hardware on the system and the support for it available at the OmpSs run-time environment, otherwise programs may not execute correctly. Figure 2.7 shows a example of this clause. Two tasks are declared, the first is intended to run on a regular CPU and the second will be run on a CUDA GPU. If no `target device` appears on the declaration of a task, it is assumed that the desired architecture is SMP (which

is the architecture for regular CPUs).

Heterogeneous architectures usually have private memories where data needs to be placed in order to use its computing capabilities. By using the mechanisms described in Section 2.5 OmpSs transparently handles the data movement required. This results in a great productivity improvement when we compare applications coded by device-specific programming models (CUDA for example) against the same OmpSs implementations. The formers usually need to provide the code to perform the data transfers between host memory and the device memory, which can be fairly complex, whereas the OmpSs implementation can delegate this responsibility to the underlying OmpSs run-time library.

2.7 Additional features

OmpSs is in continuous development and new features have been added since its initial conception. The two most relevant additions are the *priority clause* and the *task versioning* mechanism.

2.7.1 Priority clause

The priority clause provides the programmer with a simple mechanism to feed the execution scheduler with additional information in order to achieve a better performance.

The `priority` clause can be used when declaring a OmpSs task. It is used to specify the criticality of the specified task with respect to the rest of the tasks. The clause takes an integer argument to model this information, higher values indicate a higher criticality. This information can be used by the underlying run-time library to schedule the execution of the tasks. High priority tasks will try to be executed before tasks with lower priorities.

2.7.2 Task versioning

In an effort to increase the adaptability of OmpSs applications to different system configurations, the programming model offers the *task versioning* mechanism. It allows the programmer provide different implementations of a task. The implementations provided may target different hardware devices, or they can be different algorithms providing the same functionality. At execution time the OmpSs run-time environment will be able to select between the different implementations to better use the available resources, or to select the most efficient implementation of a given task.

The code in figure 2.12 exemplifies how this mechanism is used. The invocation of `do_work` will create a task that can potentially use any of the three given implementations. If the system has a CUDA GPU available, some of the task invocations will be executed on it, while the CPUs execute the rest. Also, there are two implementations that target the CPUs, in this case the OmpSs run-time can measure the performance of both versions and later decide to use only the fastest.

```
1  #pragma omp target device(cuda)
2  #pragma omp task implements(do_work)
3  void do_work_gpu() {
4      ...
5  }
6
7  #pragma omp target device(smp)
8  #pragma omp task implements(do_work)
9  void do_work_cpu() {
10     ...
11 }
12
13 #pragma omp target device(smp)
14 #pragma omp task implements(do_work)
15 void do_work_cpu_second_version() {
16     ...
17 }
18
19 int main() {
20     while( !done ) {
21         do_work();
22     }
23 }
```

Figure 2.12: Usage of the task versioning mechanism.

The work done by by Planas et al. [88] has shown that the task versioning mechanism can achieve a better performance in hybrid CPU-GPU systems in addition to enhancing the productivity of the OmpSs programming model.

2.8 Influence in OpenMP

OmpSs and the family of StarSs programming models have been developed by the Programming Models group of the Computer Sciences department of the Barcelona Supercomputing Center (BSC). The group has always been an advocate of annotation based programming

models like OpenMP. In addition, there has also been research that has targeted OpenMP features with the objective of improving it.

Many OmpSs and StarSs ideas have been introduced into the OpenMP programming model. Starting from the version 3.0, released on May 2008, OpenMP included the support for asynchronous tasks. The reference implementation using the Nanos4 run-time library and the Mercurium C source-to-source compiler was developed at BSC, which was used to measure the benefits that tasks provided to the programming model [9, 7, 97, 96]. Later, on its version 4.0 released on July 2013, OpenMP introduced tasks dependencies, which allow fine-grain synchronization between tasks, and support for heterogeneous devices. Both features are already present in OmpSs.

3

The OmpSs run-time environment

The OmpSs run-time environment is currently composed by two software components: the Mercurium C/C++/FORTRAN source-to-source compiler and the Nanos++ run-time library. Both components have been developed by the Programming Models team of the Computer Sciences department at the Barcelona Supercomputing Center.

3.1 Mercurium compiler

The Mercurium compiler is the responsible for parsing OmpSs programs and generating an executable file. Mercurium is a source-to-source compiler that relies on a language specific compiler in order to perform the binary code generation. However, Mercurium performs the code transformations required by the OmpSs directives, and introduces the calls to the OmpSs run-time environment, which is implemented by the Nanos++ run-time library.

Mercurium also simplifies the process of compiling heterogeneous applications since it manages automatically the invocation of the device-specific compilers with the appropriate pieces of code.

The internal organization of Mercurium can be seen in figure 3.1. The front-end is the responsible of invoking the pre-processor, if needed, and parsing the source code. It produces a high level representation of the user code regardless of the original input language. The resulting representation goes through different compiler phases that transform the original program. The organization of the compilation process in different phases provides a

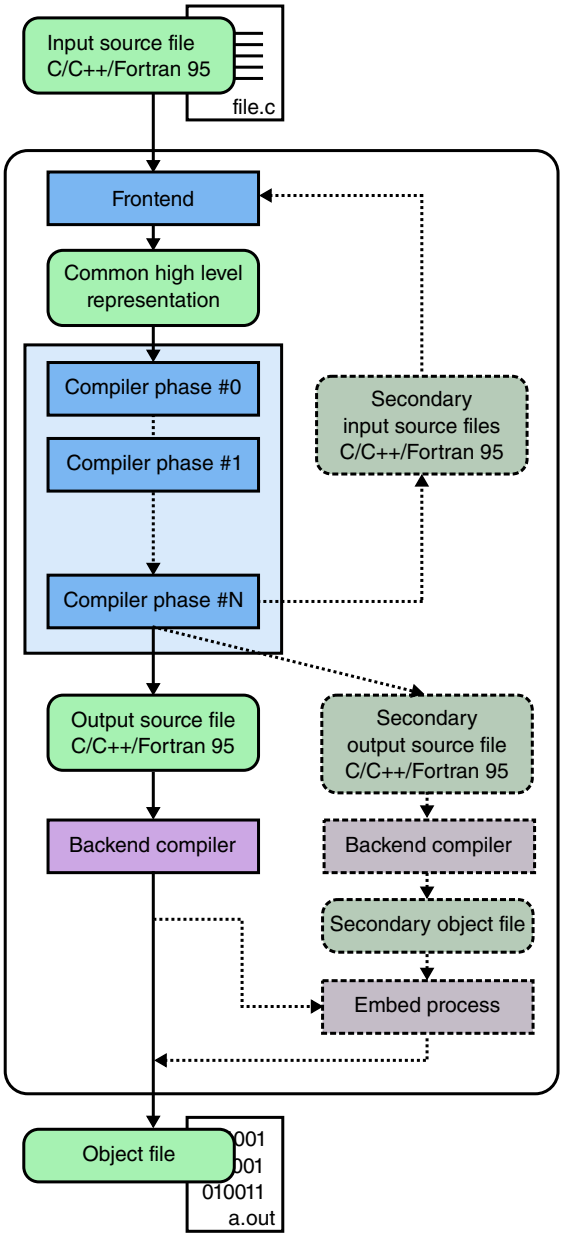


Figure 3.1: Mercurium C/C++/FORTRAN compiler internal organization

powerful way to extend the functionality of the compiler. The OmpSs transformations are implemented in a specific phase which is in charge of processing the OmpSs directives and transforming the program code accordingly. The result of the compiler phases is written in the original programming language in order to invoke the appropriate back-end compiler

that will perform the final compilation to produce binary files. In some cases multiple back-end compilers may be required, for example, when compiling CUDA code, the specific CUDA compiler is required to generate the final binary code.

3.2 Nanos++ run-time library

The Nanos++ run-time library is the component responsible of executing OmpSs applications that have been compiled using the Mercurium C/C++ source-to-source compiler. The compiler transforms most of the directives into calls to Nanos++ and it restructures the code in order to be able to be handled by tasks. Nanos++ offers subsystems that handle the task creation, dependence analysis, and task execution.

Nanos++ main responsibility is to execute the created tasks as fast as possible. To do so, it uses the available hardware and software resources of the system. The hardware resources that Nanos++ manages are the CPUs, GPUs and system memory, whereas the software resources are the threads, provided by the operating system, and the program tasks, provided by the user application.

The internal design of Nanos++ can be divided in three parts, the representation of the program tasks, the architecture support and the behavior subsystems. The program tasks are represented internally by *Work Descriptors*, which are entities that represent the code that has to be executed by Nanos++. The architecture support is composed by a set of generic concepts that model the hardware and software resources managed by Nanos++. In order to support a specific architecture, a concrete implementation of these concepts must be provided. The behavior subsystems are in charge of executing the program tasks. Each subsystem fulfills a specific role of the process. but globally they provide the logic and the intelligence of the library to manage the system resources to execute the program tasks as efficiently as possible while ensuring a correct execution order. Nanos++ provides different implementations of the subsystems that can be selected by the user when running OmpSs programs in order to achieve a more optimal performance.

3.2.1 Work Descriptors

The internal representation of a OmpSs task is done through the concept of *Work descriptor*. Work descriptors represent a piece of code that can be run independently and in parallel with other work descriptors. In addition they also represent a piece of code that can be decomposed into an arbitrary number of pieces.

OmpSs tasks are transformed directly into individual work descriptors. In the case of loops, they are transformed into a work descriptor that will be decomposed at execution time. Each decomposed part will represent a subset of the iterations of the loop.

3.2.2 Architecture support

The Nanos++ design defines a set of generic concepts that model the components of a system architecture. This set is commonly named Nanos++ *device*. A Nanos++ *device* can be composed by up to four elements, two of them model hardware resources and two of them model software resources. Typically, they are also referred using its concrete implementation, for instance, the *SMP device* refers to the Nanos++ *device* that models the SMP architecture.

The two hardware resources defined by Nanos++ are the *processing elements* and the *device memory*. They model actual hardware instances present on the system and available to the user to execute OmpSs applications.

Processing element: It models a hardware component that is capable of executing code. Processing elements can execute program tasks but can also run auxiliary code needed by the run-time environment.

Device memory: It represents the memory space accessible by the processing elements. Work descriptors typically use a set of data that they read and produce. Whenever a work descriptor is executed on a given processing element, its required data must be available on the device memory accessible by the processing element. A processing element can access a single device memory area, but a device memory area may be accessed by different processing elements.

Besides the hardware resources, there are two software resources that play key roles defining what a system architecture is. These are the *execution threads* and the *work descriptor device data*.

Execution threads: They are entities that have the capability of executing work descriptors on a processing element. Nanos++ provides concrete representations of threads for each available kind of architecture available. Each thread can be used as an interface to execute tasks on a specific PE. Threads are always binded to a PE, and a PE can have multiple threads binded to it.

Work descriptor device data: It is information associated to each work descriptor that is needed to be able to execute it on a specific architecture. A work descriptor may have

more than one *device data*, which makes it capable of being executed on different architectures.

Supported architectures

The initial Nanos++ implementation provided support for two system architectures, the SMP architecture and the CUDA-GPU architecture. Figure 3.2 shows the summary of the hardware resources modeled by these two architectures, and how them map to the generic concepts proposed by the Nanos++ design.

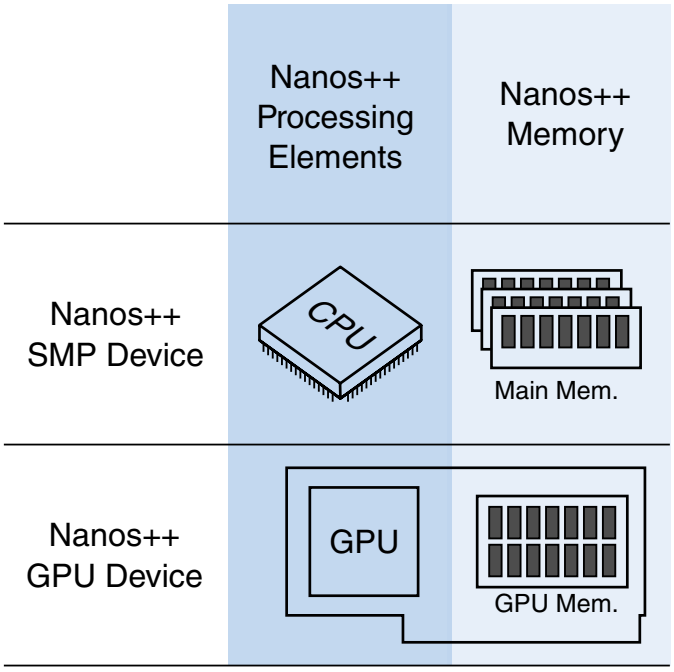


Figure 3.2: Nanos hardware resources

The SMP architecture is the most basic architecture supported by Nanos++ since it models the basic hardware resources that Nanos++ needs to execute applications. The Nanos++ *SMP device* provides the *processing element* implementation that models general purpose CPUs, and the representation of the main memory of the system. Memory of the SMP architecture is considered to be the *host memory* and it is accessible by all *SMP processing elements*. Nanos++ SMP threads are implemented using operating system threads, as they are the interface to access to the CPUs of the system. The *SMP work descriptor specific device data* contains the information needed to execute a program task on a CPU. It basically contains the address of a function that has been generated by the compiler which contains the task

code, and the argument information that the function needs.

The CUDA-GPU architecture adds the support for CUDA-GPU accelerators on a SMP system. The corresponding *GPU processing element* represents the hardware graphics processor that is capable of executing CUDA kernels. The *GPU device memory* provides the mechanisms to access the GPU Memory, which is where the graphics processor needs to have its accessed data stored. The *GPU thread* is actually implemented using an *SMP thread* because the GPUs must be accessed from regular CPUs. In the current implementation, a *SMP thread* is used to manage one single *GPU processing element*. These threads, however, are seen by the rest of the Nanos++ components as *GPU threads* and will only be capable of running tasks that target GPUs. Finally the *GPU work descriptor specific device data* is almost identical to the SMP one, since the execution of a task will also be done using a compiler generated function.

3.2.3 Behavior subsystems

Nanos++ is composed by a set of subsystems. Each subsystem has a specific responsibility in the process of executing OmpSs applications. Figure 3.3 depicts the four subsystems (dependencies, scheduler, data directory and data cache) and their roles in during the execution.

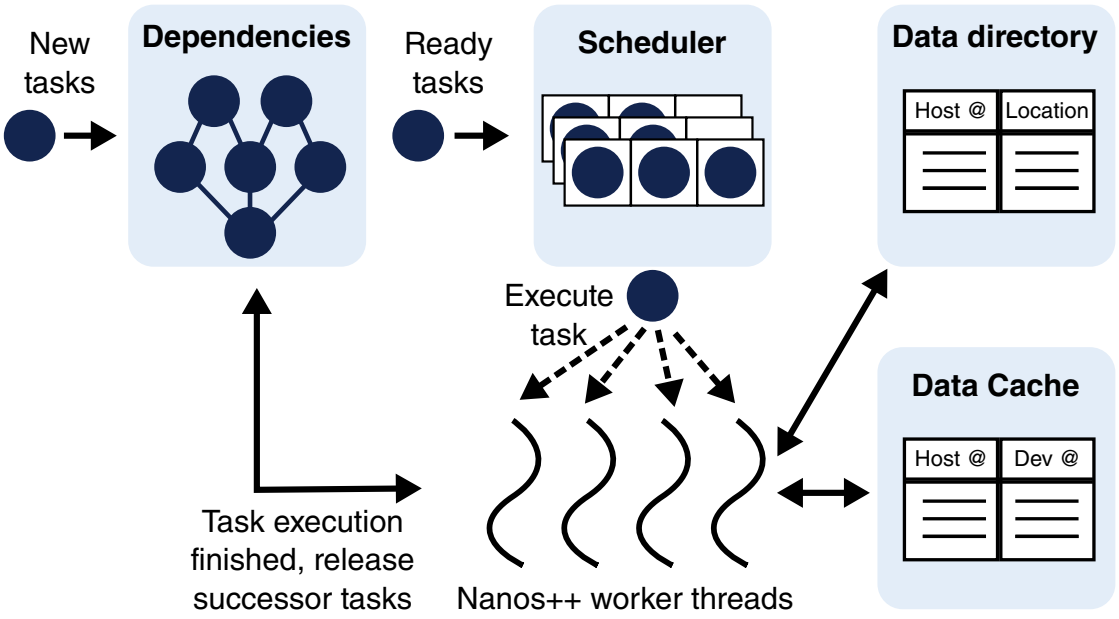


Figure 3.3: Nanos++ internal subsystems

Dependencies

The dependencies subsystem is in charge of deciding which work descriptors are ready to be executed. To do this it relies on the information provided by the programmer through the specific dependence clauses (Section 2.3) and the creation order of each individual task.

A directed acyclic graph is built dynamically as new work descriptors are created. The nodes of the graph are the work descriptors of the application and each link represents a dependency between them. Dependencies establish the predecessors and successors of a work descriptor. A work descriptor can not start its execution until all predecessors have been executed. When a work descriptor completes its execution, it notifies it to all his successors. Whenever all successors of a work descriptor have completed, it becomes ready for execution.

Each work descriptor has his own task graph to keep track of the dependencies between his child work descriptors. This means that a child task will have dependencies with other child tasks (its sisters) but will never interact with siblings of the parent.

The dependencies are created at run-time by the memory accesses that a task declares through the `in`, `out` and `inout` clauses. Nanos++ uses different algorithms to compute the dependencies based on these clauses. The most basic, named *plain dependencies* uses only the base addresses of the memory references that appear in them. A slightly more complex approach is implemented by the *cregions dependencies* algorithm, which uses the base address and the length of the data to build the dependencies between tasks. With this extra information it is capable of taking into account possible overlaps between different memory accesses, and compute the appropriate dependencies accordingly. Finally, the most complex algorithm, named *regions dependencies*, supports non-contiguous regions of memory, which allows the expression of dependencies based on complex accesses to data. This implementation is detailed in [86].

Scheduler

The scheduler mission is to assign work descriptors that are ready to be executed to threads that will execute them. It is composed of a common API that interacts with the other subsystems of the run-time and a part that defines the behavior of the subsystem. There are different implementations of the latter part that come in the form of plug-ins. These plug-ins can be selected at run-time and provide different scheduler behaviors. With this, the user can select the most appropriate strategy for his or her application.

The following scheduling policies were originally implemented in Nanos++:

Breadth first It tries to executes the program tasks in creation order.

Distributed breadth first It follows the same idea than the breadth first scheduling policy but it tries to execute the created tasks on the PE that created them. This strategy aims to exploit data and temporal locality when executing applications with multiple levels of parallelism.

Work First It executes tasks as soon as they are created, and defers the creator tasks for later execution.

Priority It executes the tasks in priority order. Priority is provided by the use of the clause `priority` (see Section 2.7.2)

Smart priority It is the same idea as the priority policy but it propagates the priority of child tasks to their parents.

Data directory

The data directory has the responsibility of knowing where the program data is located during the execution. This component is needed when the system has more than one address space, and keeping track of where, given a memory reference, its most recent value is stored. This functionality is required in order to provide the memory model specified by OmpSs.

The original Nanos++ design has some important limitations, the data references allowed by the data directory are limited to a single address with a length. This means that only contiguous regions of data can be tracked with the data directory. In addition, different memory regions should not overlap with each other.

For each memory reference tracked by the data directory, there is also a version number. The version number is used to keep the last produced value of a piece of data. It is needed because data may be located in several address spaces at the same time. This happens when different tasks reference the same piece of data as *input* data, since the tasks are not modifying the value of it, they can run at the same time and the data can be replicated safely. However as soon as a new value is produced, the rest of the copies must be invalidated. To implement this, the version number of a memory reference is incremented each time a new value is produced (by a task referencing the data in a *out* or *inout* clause). When the directory is asked for the location of a memory reference, only returns the location with the latest produced version.

Data cache

The data cache is closely related to the data directory since it also shares the responsibility of keeping the memory coherence during the execution. This component however manages the data that has to be transferred in and out of an individual address space different than the host address space. An instance of the data cache knows if, given a memory reference and a version number of it, the data is available in its managed address space. Whenever a task is about to be executed, if the data is not present on the execution address space, the data cache will issue the appropriate operations to transfer the missing data. As a consequence of this functionality, the data cache also maintains the association between addresses of the host address space and addresses of the managed address space.

This subsystem also manages the available space on the managed address spaces, and it does so transparently. When a task needs to be run on a given address space, the data cache may evict pieces of data that are not in use to free space for the required task data.

The name of this component, data cache, may be a bit misleading but it actually refers to the fact that Nanos++ manages the address spaces of accelerators as if they were cache memories of the host address space. This design choice abstracts even more the access to separate address spaces but also is in line with the OmpSs which hides the presence of these to the user.

The data cache can also be configured by the user, who can select between different behaviors. These behaviors specify what to do with the data that has been generated in the managed memory, which can affect the performance of applications. Nanos++ implements three different data cache policies:

Write-Through This policy forwards the newly produced data to the host memory as soon as the work descriptor that has produced it ends its execution. This behavior will issue transfer operations between the device memory and the host on almost each work descriptor, but it can effectively reduce the time of evicting a piece of data from the device, since the version on the host will always match the version of the device, thus a transfer will not be needed when doing an eviction. It can also be beneficial when multiple devices require the produced values of a single work descriptor, since the data will be available in the host instead of in a single device. Making the data available on the host memory may yield better performance on some applications.

Write-Back This policy keeps the data in the device memory as long as possible. Newly produced values are not transferred to the host until a work descriptor running on

PE that uses a different memory requests them. This has the important benefit of minimizing the amount of data transferred between the different address spaces, but on the other hand it can add some delays to the critical path of the application.

No-cache It is a basic strategy used mostly for debugging purposes where data is always released when a work descriptor finishes. This means that each time a work descriptor is executed, all its required data will have to be transferred since it will not be present on the memory space.

3.2.4 Execution flow

The execution flow of Nanos++ is divided in three different stages: initialization, program execution and finalization.

Initialization

During this stage Nanos++ sets up the internal data structures based on the resources available in the system and the configuration options provided by the user. PEs are created, threads are started and binded to PEs, and the different subsystems are also initialized. Initialization is done by the main thread of the application. Other threads created during this stage start in the idle status.

Program execution

When the initialization stage finishes the main thread starts executing the *main* method of the OmpSs application, which effectively marks the start of the user program. Program execution is composed of two different scenarios, *task issuing* and *task execution*. Both cases may appear multiple times during the execution since tasks may create new child tasks.

Task issuing Eventually the user program will call Nanos++ services to create tasks and will submit them for execution. If tasks have dependencies they will be processed by the dependencies subsystem, which will decide if they are ready for execution. Tasks without dependencies are considered to be ready as soon as they are created. Ready tasks are handed to the scheduler subsystem which is in charge of distributing them to idle threads.

Task execution An idle thread will actively request the scheduler a work descriptor to execute in its bounded PE. When it receives a new work descriptor the task execution procedure

starts. The first action that takes place is to prepare the task data. This step may use the data directory and the data cache if the system has separate address spaces. Data may be located on an address space distinct from the one where the task is about to be executed, and this will require move the data from its location. The new location of the data is also registered in the data directory, and, if the target address space is not the host address space, the corresponding data cache subsystem is also updated. Once this step is completed, the thread can start the execution of the task. When the execution completes, the thread proceeds to use the data cache subsystem to transfer the generated data back to the host address space–again, this only happens if the execution has taken place in a PE with a separate address space. Finally the last step is to notify the dependencies subsystem that the task has completed, this may release their successors which will become ready work descriptors and will be transferred to the scheduler.

Finalization

When the *main* method finishes the finalization stage starts. This stage waits until the created tasks finish their execution, and then it proceeds to join the created operating system threads, frees the device resources and exits the main thread.

3.3 Conclusions

The OmpSs run-time environment is composed by two components, the Mercurium C/C++/FORTRAN source-to-source compiler and the Nanos++ run-time library. Both components have been developed to be extensible in order to allow the inclusion of new features with ease.

The work developed for this thesis has been done using the Nanos++ library. The following chapters describe additions and changes needed to add a new system architecture and improve the support of several OmpSs features. Therefore this chapter has presented an overview of the internal Nanos++ design, with the goal of ease the comprehension of the upcoming content.

4

OmpSs for clusters of multi-cores

As discussed in chapter 1, cluster architectures have become ubiquitous in the High Performance Computing field, however, developing applications for them has proven to be a difficult task. New programming models have been created in order to ease this process, specially because users of these kind of systems can be experts on different scientific areas but without a deep knowledge in computer science and parallel programming.

OmpSs was conceived with the idea of being a high productivity programming model, which means offering a simple interface but also being able to compete with other tools in terms of performance. Originally OmpSs was available for SMP architectures and systems with GPUs, however given its features, cluster architectures could be also a reasonable target for it.

The support for private address spaces can be used in order to automatically manage the data movement between nodes of the cluster, freeing the user of having to think about it. In addition the users can develop parallel applications using a more step-by-step methodology, because OmpSs directives can be applied to sequential code.

This chapter discusses the design and implementation of the cluster support for OmpSs and shows a performance evaluation and a comparison against the MPI programming model, which constitutes the first contribution of the thesis. First section 4.1 defines the characteristics of the cluster architectures that we are going to target. Section 4.2 explains how the cluster support has been design and implemented, and finally section 4.4 shows an evaluation of the implementation of OmpSs for clusters and how it compares against the most

popular programming model for distributed systems: MPI.

4.1 Cluster architectures

By cluster architectures we refer to systems with multiple nodes connected through a communications network. Figure 4.1 depicts this definition. Each individual node must be formed by one or more CPUs and must be able to communicate with the rest of nodes in the cluster. Nodes may have different configurations, but during this document we will consider that all nodes of a cluster have the same amount of hardware resources.

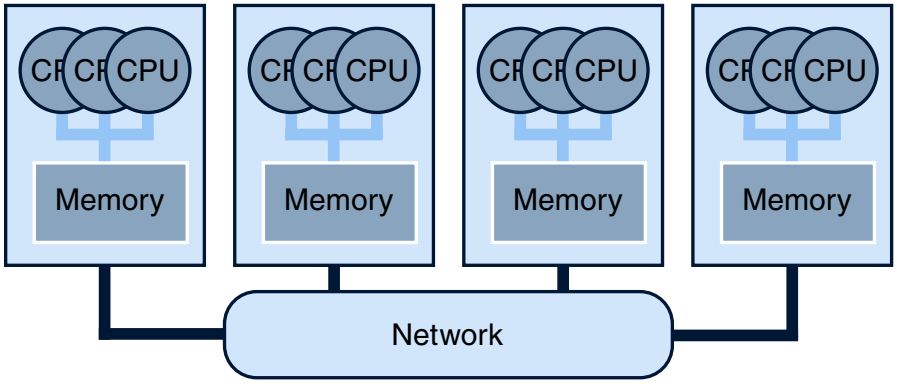


Figure 4.1: Cluster architecture

4.2 Nanos++ for clusters

Providing support for cluster environments required some changes and additions in Nanos++, however the majority of them fitted in the general design of the library.

The most significant change was to move from a single process execution to a multiple process execution. In order to keep the original Nanos++ design and also following the OmpSs execution model, we decided to follow a master-slave design, where one process acts a director of the execution (master) and the rest of processes follow the commands issued by the master (slaves).

A new *device* was implemented to provide Nanos++ with the implementations of the generic concepts that already were supported: processing elements, threads, methods to access the cluster remote memory.

The cluster support also exposed a new resource that was not required by other architectures: the communications network. It is used to transfer data and issuing commands

between the different nodes of the cluster, and some subsystems of Nanos++ required to interact with it to coordinate the execution on the cluster.

4.2.1 New components

Several components of the cluster environment match the basic elements of the Nanos++ design. The *cluster device* has been implemented to provide the specific routines and objects to interact with the Nanos++ design, more precisely, the *cluster device* provides the cluster specific processing element implementation, which represents a node of the cluster, and the mechanisms to expose the memory of each cluster node. Figure 4.2 summarizes the new hardware elements that are added by the cluster implementation. The *cluster device* also provides a specific *cluster thread* implementation in order to match the rest of the Nanos++ design. However it does not implement the *cluster work descriptor device data* as cluster specific work descriptors will not exist. The cluster implementation will handle work descriptors that target other architectures.

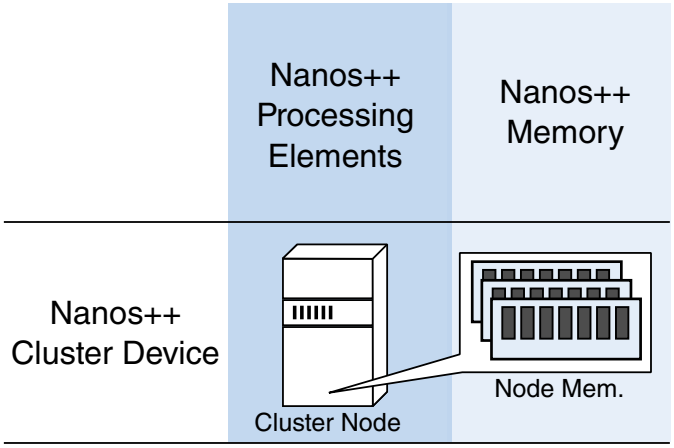


Figure 4.2: Nanos++ new hardware resources

Cluster node

In the cluster implementation, the concept of a Nanos++ processing element is represented by a *cluster node*. From the master point of view, a cluster node fulfills the responsibilities of a Nanos++ processing element: it can execute tasks. Also, a cluster node has its own private memory where the task data must be transferred before starting the execution.

There is an important difference between a cluster node and other implementations of

processing elements. For The SMP and GPU architectures, their respective processing elements can only execute work descriptors that target the specific hardware, this is, work descriptors that contain the appropriate type of device data. The cluster node must be able to execute work descriptors originally created to run on other architectures. This is needed in order to make the cluster support completely transparent to the programmer, since any task will be potentially capable of being run on a remote node. During this chapter the *cluster node* will only be capable of handling work descriptors that target the SMP architecture.

Cluster thread

Following the Nanos++ design, *cluster threads* are components that allow the execution of tasks on *cluster nodes*. The design also requires one thread per processing element which means that at least there are as many threads as remote nodes. These threads do not execute tasks themselves so they are helper threads. They are in charge of sending work descriptors to their associated nodes and notifying when these have completed their execution.

Originally the cluster threads were implemented using the same approach as the GPU threads, which was to actually use a SMP thread as a cluster thread. However this created one problem when running applications with a high amount of remote nodes. Creating too many SMP threads impaired the performance of the run-time since these are actual operating system threads. Creating too many O.S. threads is problematic because the O.S. ends up multiplexing them on a single core (cluster threads were binded to a specific CPU different from the ones assigned to the SMP worker threads), which is an expensive operation. The solution to this problem was to create the cluster threads as threads that were executed on top of a single SMP thread, and leave the responsibility of scheduling them to the Nanos++ run-time library. The scheduling of these threads was implemented using a round-robin schema where each thread could run and perform its operations until it yields for the next one. This has an important implication, operations done by the cluster threads must be non-blocking as they would prevent the rest of the threads of executing.

Cluster threads are created only on the master node of the execution. Slave nodes can not issue tasks for remote execution and thus they do not need to spawn cluster threads.

Node memory

Nanos++ supports devices with disjoint address spaces. The device specific code has to provide a specific methods to be able to transfer data from the host address space to the device address space, and the other way around. The memory consistency model required

by OmpSs is implemented by two generic subsystems, the *data directory* (section 3.2.3) and the *data cache* (section 3.2.3), the specific details of how it is organized in the cluster case are discussed in section 4.2.5.

The cluster specific device implements these methods using the *network* subsystem (see section 4.2.2). The *data directory* and the *data cache* did not required any modifications to support the cluster device.

4.2.2 Network

The network resource appeared in Nanos++ as a new subsystem. A generic interface was designed to interact with the rest of the subsystems independently of the network specific details.

The first implementation of the network subsystem was done using the GASNet communications library. GASNet offers active messages, a pattern that provides one sided communication that is commonly used in distributed memory programming. When a node receives an active message a handler is automatically invoked to process it. Typically active messages carry some associated data which is accessible by the handler. This schema can take advantage of techniques such as Remote DMA (RDMA), which usually provide better performance than traditional communications protocols.

An important benefit of using GASNet is that the library offers a common interface but it provides with implementations for different high-performance network devices. Implementations for generic network libraries, like UNIX sockets or MPI, are also available.

4.2.3 Master-slave design

We followed a master-slave pattern to implement the cluster support because it suited the thread-pool execution model of OmpSs. Figure 4.3 summarizes the chosen design. One node of the cluster will act as a master node and will orchestrates the execution of the OmpSs application. The rest of the nodes will act as a slave nodes where the master can send program tasks for remote execution.

The slave nodes act as a *daemons* that wait for incoming commands. These commands can instruct them to execute tasks, send or receive data, or send a synchronization message back to the master among other actions.

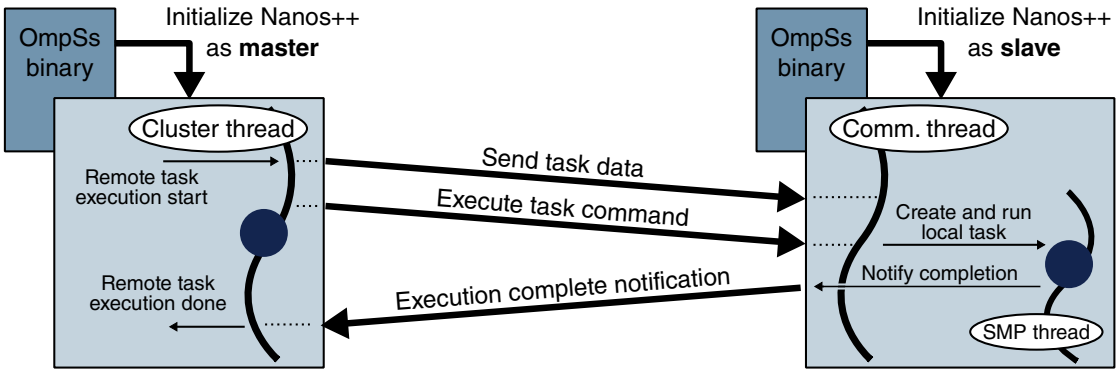


Figure 4.3: Master-slave pattern used to implement the Nanos++ cluster support

4.2.4 Task execution

The execution of tasks on the cluster follows the general Nanos++ flow described in section 3.2.4. Cluster threads are in charge of this responsibility. All operations are non-blocking but some actions require waiting for a response, in this cases, the wait is implemented by checking the status of the operation, and yielding to other cluster threads in order to not stall the execution. The detailed process for the cluster implementation follows these steps:

1. The first action a cluster thread does is to ask the scheduler subsystem for a task to execute. If there are ready tasks the scheduler will likely provide one, but if none is provided, the thread will yield to other threads and keep asking for work to do once the rest of the cluster threads yield the execution to it.

2. Once a thread has obtained a work descriptor to execute remotely, it has to issue the data movement operations to transfer the data that the task will use. This is done in cooperation with the Nanos++ directory (described in section 3.2.3) and data cache (described in section 3.2.3). The directory provides the information about the location of the required data. If it is not present on the target node, it will be transferred. The cache is in charge of assigning a memory chunk for the data and will end up invoking the appropriate cluster device operations which will also use the network subsystem to do the real transfers through the communications network. Sometimes data must be transferred from a slave node to another slave node, this process requires two operations, the first one to transfer the data from the current owner to the master node and the second to perform the final transfer from the master to the destination. In this case, the operation involves waiting for the incoming data and issue the final transfer.

3. The second action, imposed by Nanos++, consists in waiting for the operations to be completed. This is needed to ensure that when the tasks starts running, the data is totally transferred into the destination memory. However, in the cluster case this stage does not perform any wait because data transfer commands and task execution commands are synchronized on the destination.

4. The third action corresponds to the issue of the command to start the execution of the task. The master sends to the target slave node the device specific information to reconstruct the original work descriptor on the destination node. In the case of work descriptors that target the SMP architecture this information is composed by the address of the function to call and its arguments. In addition, each work descriptor also requires the memory references corresponding to the data that it will use. At this stage, these memory references are translated, using the data cache, to match the addresses of the destination node. In the original Nanos++ design this step waited until the execution of the work descriptor finished, blocking the launcher thread in the process. For the cluster architecture this was changed in order to avoid the blocking and allow the thread to continue doing other actions.

5. The last action happens when the master node receives the notification that the task has completed the execution and notifies the executor thread to process the event. The thread is in charge of issuing data transfer operations (the last decision depends on the cache subsystem) and notifying the completion of the task to the dependencies subsystem.

None of this steps block the execution of the current thread, because, as explained in section 4.2.1, all cluster threads are executed on top of a O.S. thread and their execution is explicitly multiplexed, which means that if one of them would perform a blocking operation, all of them will also be blocked.

4.2.5 Memory management

Figure 4.4 shows how the different Nanos++ subsystems are organized to manage the memory of the whole cluster. The master node is the responsible for keeping the memory coherent with the OmpSs memory consistency model but also for offering the OmpSs single address space view (section 2.5.1). The master node memory is what OmpSs considers the *host memory* or *host address space*, and it is the only address space exposed to the application. The memory of each slave node is treated as a private device memory and is managed by

the master node.

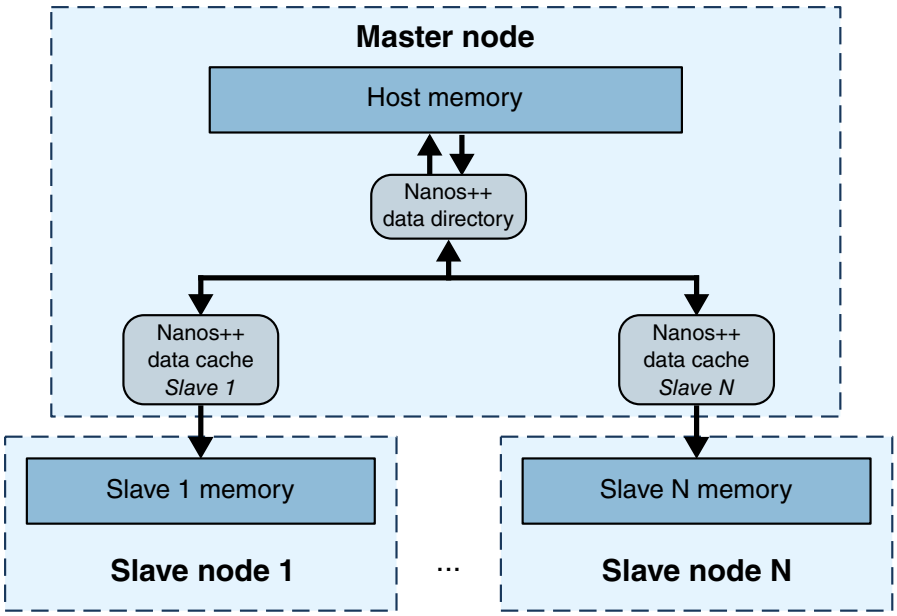


Figure 4.4: Nanos++ distributed memory management organization

The *data cache* component manages the operations that the master node has to do to transfer data to and from private memories. In the case of the cluster architecture, each node memory is considered a private memory. There is one instance for each address space present on the system other than the main node. These components can allocate memory chunks, free them and transfer data from their managed address spaces to the host address space and the other way around, and also keep the mapping of host memory addresses to their private memory addresses. These operations are implemented by the *cluster device*. Memory transfer operations are implemented using network transfers. Allocation and free operations are handled locally at the master node, since the devices performs a big allocation on each remote node when the run-time starts, this way there is no network latency when performing these operations.

A memory reference may have several copies of its contents on different address spaces of the system. To maintain the coherence of the memory, the master node uses the *data directory*. It contains the information of where the last produced values of a memory reference are located. With it, the system can determine which transfer operations must perform to execute a task in any device of the system. Also, each task execution updates the information of the data directory to reflect the newly produced data.

4.2.6 Optimizations

Running a task on a remote node can be a expensive operation depending on the amount of data that the task may need to transfer. The cost of moving the data can not be avoided but it can be mitigated using two strategies: exploiting the data locality of different tasks and overlapping communication with computation. The former is the basic concept of a specific scheduling policy, the latter is exploited by a mechanism of task pre-sending which is discussed in section 5.3.3.

The *data cache* provided a sub-optimal solution when data had to be transferred from one slave node to another slave node. The original behavior involved moving the data through the master node, which was an unnecessary step since each node has direct access to the rest through the network. The data cache was modified in order to avoid the transfer through the master.

Affinity scheduler

A new scheduling policy was developed to use when running applications on a cluster. The main idea is to execute tasks on nodes where the data they access is located, this strategy reduces the amount of transferred data during the execution. The scheduler also takes care of distributing the data and avoiding imbalance.

The scheduling process first classifies the tasks in two classes: *data producer* tasks and *computation* tasks.

Data producer tasks are these which will only write new data, that is, their data usage clauses only include `out` or `copy_out`. These kind of tasks typically appear at the beginning of applications and are in charge of generating large amounts of data that will be used later by other tasks. The scheduler distributes this tasks according to the amount of data that they generate, trying to balance also the amount of data generated on each node of the cluster.

Computation tasks are the rest of the tasks that are not data producers. Generally computation tasks read certain pieces of data and produce or update the results on a different location, so they are declared using several `in/copy_in` clauses and some `out/copy_out` or `inout/copy_inout` clauses. For each computation task, the scheduler assigns it to the node that contains the largest amount of data required. This is done computing an *affinity score* for each node on the cluster. The score is the amount of data, in bytes, available on the given node. The information of where the data is located is provided by the data directory subsystem (see Section 3.2.3). The scheduler assigns a task to the node with the highest affinity score.

Different nodes may tie for the highest affinity score. This may happen because data may be replicated on different nodes, or because data produced with a computation task is later used as input data of other tasks. The scheduler keeps a set of counters, one for each node, that keep track of the number of tied tasks that have been scheduled to each node. Whenever a task ties for two or more nodes the scheduler picks the node with the least number of tied tasks scheduled, and updates the counter. If several nodes also tied for the tied tasks counter value, the node with the lowest identifier is picked.

The assignation of tasks to threads is implemented using task queues. Each node has a specific queue where the corresponding cluster thread will check for new tasks to execute.

As a final mechanism to avoid imbalance between nodes, *task stealing* may be enabled by the user. Task stealing allows a cluster thread to pick tasks from queues of other threads. This provides an automatic balancing mechanism since when a node runs out of tasks to execute, its cluster thread will pick tasks originally scheduled for other threads. This mechanism, while probably disrupting the original purpose of the scheduler, will obtain a more efficient use of the parallel resources.

The aforementioned scheduling policy is used only when scheduling top level tasks. Top level tasks are these that are created by the main program, that is, not created by any other parallel task. Children tasks are always executed on the same node where they were created. This favors locality since the data accessed by children tasks is always a subset of the data accessed by the parent task, thus it makes sense to schedule the children tasks where the parent is running.

Slave-to-slave communication

As described in section 4.2.4, when a task is being prepared for execution, one of the first steps is to transfer the required data to the node where it will be executed. This process is driven by the *data cache* associated to the destination node. The initial implementation of the cache did not consider the case where the devices could transfer data between each other. This meant that when a slave node required data located on a different slave node, the data had to be first transferred to the master node and then from the master to the destination node.

For the cluster architecture, this process is too inefficient because the communications network allows each node to send and receive data from each other, thus we can simplify the process by just moving the data from the origin to the desired destination. The *data cache* was enhanced to support this scenario. The performance improvement of this optimization

is detailed in Section 5.5.4.

4.3 Impact in OmpSs

The implementation of the cluster support for Nanos++ has a minimal impact on the OmpSs programming model. No extra directives or clauses have been added to enable the cluster support. There is also no need to specify the target architecture when declaring tasks as is the case when targeting CUDA GPUs.

The cluster implementation does however add an extra requirement when declaring tasks. OmpSs assumes that tasks target the default architecture, SMP, when there is no a specific target declared. Typically SMP tasks do not contain data usage clauses because in a SMP environment there is only one address space. This assumption is not true in the case of the cluster implementation, and therefore the user must add the required clauses when the program is intended to be run on a cluster. This is not a big burden since the clause `copy_deps` can easily allows the re-usage of the dependencies causes as data usage clauses.

4.3.1 Cluster memory

The OmpSs programming model has an inherent drawback when it comes to dealing with a distributed memory architecture. As described in Section 2.5.1, OmpSs virtualizes the distributed nature of the memory and offers a single address space view. This virtualization is implemented by making the memory of the remote nodes replicas of the master node memory. This has an important consequence: all program data must fit within the master node, and thus, the memory available to the programmer is not the aggregation of the memory of the nodes but the memory of one single node. This limitation is serious because limits the amount of data that can be managed by OmpSs programs, but, in this thesis, we have chosen not to address it. Overcoming this issue would require drifting away from the original OmpSs specification and philosophy so this problem is left for future work.

4.4 Performance evaluation

The design of the Nanos++ cluster support offers clear productivity benefits since it allows OmpSs applications to run on a distributed environment almost without having to change the source code. However, as OmpSs targets high performance systems, the proposed design

must be able to deliver a reasonable performance when compared against other programming models.

4.4.1 Methodology and environment

In order to evaluate the Nanos++ cluster support we implemented a set of benchmarks and executed on a cluster of SMPs. For each application, two versions were implemented: one using OmpSs, and a second using MPI. The goal is to compare the performance of the OmpSs environment against the performance that can be achieved with MPI, which is the most widely used programming model when it comes to distributed architectures. The benchmarks were run in two versions of the MareNostrum supercomputer, MareNostrum2 and MareNostrum3:

MareNostrum2 The nodes of the first version were composed by two PowerPC 970MP @ 2.3GHz processors, each of them had 2 cores. There were 4 Gb of physical memory on each node, the operating system used was SLES 10 and the interconnection network of the cluster was based on Myrinet hardware along with the Myrinet Express driver. OmpSs was run using the MPI conduit for GASNet. Since there is no specific conduit available for the Myrinet Express driver, the MPI conduit allowed us to indirectly use Myrinet Express through the MPICH library that was installed on the system. All benchmarks were compiled using the Mercurium C/C++ compiler version 1.3.5.7 using the GCC compiler as the back-end compiler, -O3 optimization level was always used.

MareNostrum3 The latest version is composed by nodes containing two Intel® Sandy Bridge EP E5-2670/1600 processors. Each processor has eight cores, 20 Mb of cache memory and runs at 2.6 GHz. Each node has 32 Gb of memory available and the operating system used is SUSE Linux Enterprise Server 11. The interconnection network of the cluster is Infiniband FDR10. The OmpSs configuration for this cluster used the dedicated Infiniband GASNet conduit. Benchmarks were compiled using the Mercurium C/C++/FORTRAN compiler version 1.99.3 and Intel Compiler 14.0.2, and always using the -O3 optimization flag.

4.4.2 Benchmarks

We selected four applications with different computation patterns in order to see how OmpSs handled different situations. While not all applications may be classified as one of them, the

four applications offer a wide enough range of traits to evaluate the capabilities of the cluster implementation of OmpSs. A detailed description of each application follows:

Matrix Multiplication The benchmark performs a dense matrix multiplication of two square matrices. It represents a computation intensive application that uses a significant amount of data. The parallel tasks share data and compute common data, so synchronization and data communication occurs between the tasks execution. Each matrix is divided in blocks; in the MPI version this is used to tile the execution of the algorithm, in the OmpSs version tiling is also applied, however the algorithm is structured slightly different. The OmpSs version uses multi-level parallelism, the top level tasks are the ones that are distributed through the cluster and they are in charge of creating the bottom level task on each remote node. These bottom level tasks will perform the final computation. Using this schema the bottom level tasks benefit from better data locality, since their parent task already requested their needed data. In the MareNostrum2 experiments, the MPI version and the OmpSs version used a simple kernel in order to perform the matrix multiplication, and each matrix had a size of 32x32 blocks of 400x400 doubles on both versions. The OmpSs code had the matrices stored in blocks and, due to the limitations of OmpSs described in Section 6.1, an input matrix was transposed before doing the computation. This transposition is required to make parent tasks reference contiguous memory. The code was updated for the MareNostrum3 experiments. The contribution described in Chapter 6 made the transposition stage unnecessary. Also a Intel® Math Kernel Library (MKL) kernel was used in the OmpSs benchmark. The MPI version used was the ScaLAPACK call `pdgemm` from the Intel®, which performs a matrix multiplication and internally uses MPI to transfer the data among the nodes. The size of each matrix was 16384x16384 elements. In the OmpSs implementation, the computation was divided in blocks of 512x512 elements.

NAS EP As its name says (Embarrassingly Parallel) this application has a lot of computation that can be done in parallel and the parallel parts share almost no data among them. The EP benchmark generates pairs of Gaussian random deviates according to a specific scheme. A main loop distributes the computation among threads and a reduction is done to verify the results. The implementation for OmpSs divides the main loop of the benchmark in tasks, and implements the reduction manually. For the MareNostrum2 experiments, the OmpSs implementation was based in the C implementation of the 2.3 NAS Parallel Benchmarks. The MPI version comes from the original NPB v2.3 for FORTRAN [36]. The code was updated in the MareNostrum3 experiments, where we

used the FORTRAN code to implement the OmpSs version. Also the main loop was parallelized using two levels of nested tasks, that is, the main loop was divided in tasks and these tasks also divided their computation in more tasks. In both experiments the class C problem size was used.

STREAM STREAM is a benchmark [71] [73] that measures memory bandwidth for simple kernels, intended for use with large data sets, it is also part of the HPC Challenge benchmark suite [65]. The benchmark represents applications that may be sensitive to memory latency. The parallelism structure is fairly simple, with low communication and synchronization between the parallel parts. It performs four simple operations over three one dimensional arrays. The first operation copies the elements from one array to another, the second multiplies a scalar value to each element of one array and stores the result on a second one, the third operation performs a vector addition of two of the arrays and stores the result on the third array, and the four operation is also a vector addition but it scales the elements of one of the source arrays. The application is divided in four parts using barriers. During each part one of the kernels is executed over the arrays. The MPI version used was the reference implementation [72]. In the MareNostrum2 experiments each array was 500 Mb large in order to use the maximum memory that the GASNet configuration used could handle. In MareNostrum 3 the array size was incremented up to 2 Gb of memory per array, and it scaled with the number of nodes, as it happens with the MPI implementation. We also parallelized the MPI implementation with OpenMP to better exploit the multi-core nodes of MareNostrum3.

Sparse LU It computes a LU decomposition on a sparse matrix and can have empty blocks. Due to the sparseness, the total number of tasks generated is less than in a regular LU. This application generates irregular parallelism, the parallel tasks may be computationally intense but the amount of parallelism created is dependent on the data input. Task parallelism with dependencies can benefit from this situation as it can overlap multiple iterations at the same time whereas MPI needs barriers across different iterations. OmpSs provides fine grain synchronization between different parallel parts and adapts to any data input. The OmpSs implementation was produced using a MPI version developed at the BSC. In the MareNostrum2 experiments the matrix size used was 32x32 blocks of 400x400 doubles. In MareNostrum3, we used OpenMP to parallelize several parts of the MPI version to exploit the multi-core environment, and the matrix size was incremented to 64x64 blocks of 512x512 doubles.

4.4.3 Experiments

The selected applications were run with different configurations of numbers of nodes to obtain the speed-up of each application for both OmpSs and MPI. The same data size is used with the different hardware configurations for all the benchmarks. For all the experiments, we enabled the run-time optimizations discussed in Section 4.2.6.

In the MareNostrum2 experiments, the baseline result for computing the speed-up was obtained using the execution time of a sequential version of each benchmark. Also, only one worker thread per node is used.

The MareNostrum3 experiments report the performance achieved in terms of execution time (lower is better) except for the STREAM benchmark, which reports the performance in gigabytes per second (higher is better). In this environment we also show different configurations of worker threads per node, ranging from just one to the maximum available. The maximum number for OmpSs is 15 threads due to the presence of the communication thread whereas in MPI 16 threads are used in the maximum threads scenario.

In both environments we used from 1 to 32 nodes to run the experiments. We did not experiment with more nodes because of the restriction described in Section 4.3.1, which limits the amount of memory that the benchmarks can handle. Using more than 32 nodes results in too many hardware resources for the problem sizes used, and we believed that the results would not have been significant enough to be evaluated.

4.4.4 Results: MareNostrum2

Figure 4.5 shows the performance results of the benchmarks implemented in OmpSs and MPI and executed on the MareNostrum2 cluster. The performance results of each application are detailed in the following lines.

Matrix Multiply

The benchmark achieves a good performance on OmpSs, almost identical to the MPI version. This is somewhat expected since Matrix Multiply has a lot of data parallelism that can be exploited efficiently using either MPI or task parallelism. Figure 4.5a shows the results obtained for both OmpSs and MPI, perfect scalability is not achieved since communication and computation are not overlapped, but the scalability of the OmpSs code is on par with the MPI code.

NAS EP

The EP benchmark has almost no data sharing among tasks, so it fits well on the set of applications that can achieve a good performance on distributed environments. With OmpSs there is no exception and the results showed a perfect lineal speed-up, on par with the original MPI implementation. Figure 4.5b shows the results we obtained executing the class C of the benchmark.

STREAM

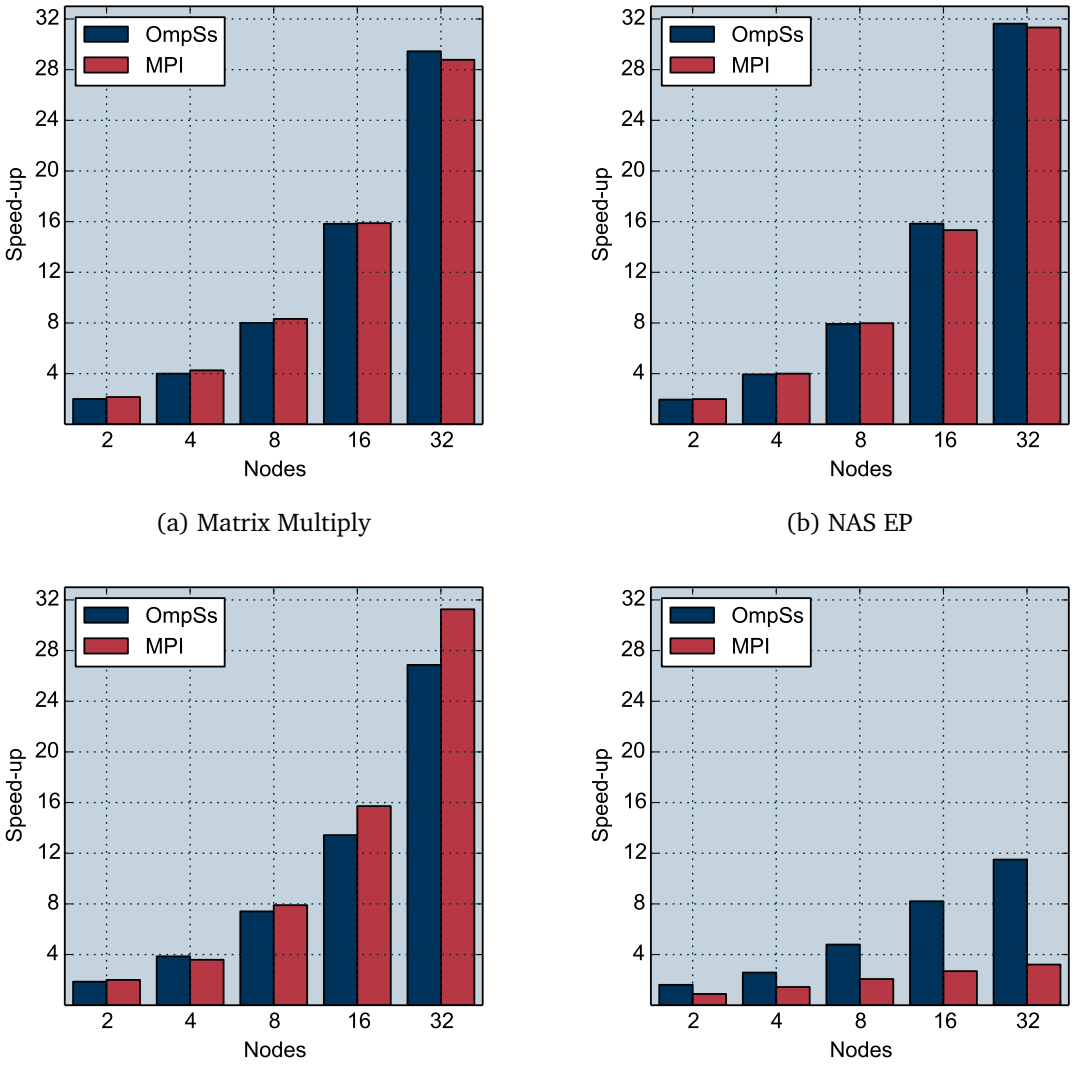
The results obtained by the STREAM benchmark are almost on par with MPI. As seen in figure 4.5c, the OmpSs implementation achieves almost the same scalability as the MPI version, only losing a little bit of performance due to the centralized initialization of the tasks. MPI outperforms OmpSs because of the SPMD model, since, besides the first synchronization done when initializing the MPI run-time, there is nothing to setup. On the other hand the creation of tasks in OmpSs takes some time, and it is proportional to the number of tasks, in addition, the distribution of these tasks also takes time proportional to the number of nodes of the execution.

Sparse LU

Sparse LU using OmpSs performs better than MPI since it exploits the advantages of having fine-grained tasks with dependencies. This is specially important in sparse matrices since data parallelism is lower than in non-sparse ones. The MPI version uses explicit synchronization between iterations of one of its loops, which limits the parallelism to the maximum that can be generated per iteration. By contrast, OmpSs can overlap the execution of different iterations and generate more parallelism. Figure 4.5d shows this effect, where the OmpSs application achieves higher scalability than MPI on any number of nodes.

4.4.5 Results: MareNostrum3

The following sections present the results of the MareNostrum3 experiments. For each benchmark five charts are shown, each one corresponds to a specific configuration of workers per node. The values go from 1 worker per node to the maximum, 15 for OmpSs and 16 for MPI. The difference is caused by the communication thread required by the Nanos++ library, which occupies a CPU of the system and limits the amount of worker threads that can be created. The horizontal axis of each chart correspond to the number of nodes used.



(a) Matrix Multiply

(b) NAS EP

(c) STREAM

(d) Sparse LU

Figure 4.5: Scalability comparison between OmpSs and MPI on MareNostrum2

The vertical axis represent the performance result of the application. The range of the vertical axis is scaled according to the number of worker threads, so the maximum value of the second chart, which corresponds to the case when 2 worker threads are used, is twice the maximum value of the first chart.

Matrix Multiply

Similarly to the results of MareNostrum2, this benchmark shows a good performance that can be compared to the one provided by the MPI implementation. Figure 4.6 shows the performance measurements and the performance achieved by OmpSs is slightly better when running with 8 and the maximum amount of workers per node. The OmpSs scalability is worse than MPI when going from 16 to 32 nodes. This is caused by the centralized execution model of Nanos++ which introduces some unbalance during the execution. In addition, the block size used also limits the amount of parallelism created, which is low in the case of 32 nodes and does not allow Nanos++ to benefit from the pre-send mechanism. The MPI version benefits from following a more rigid structure, and does not face these problems.

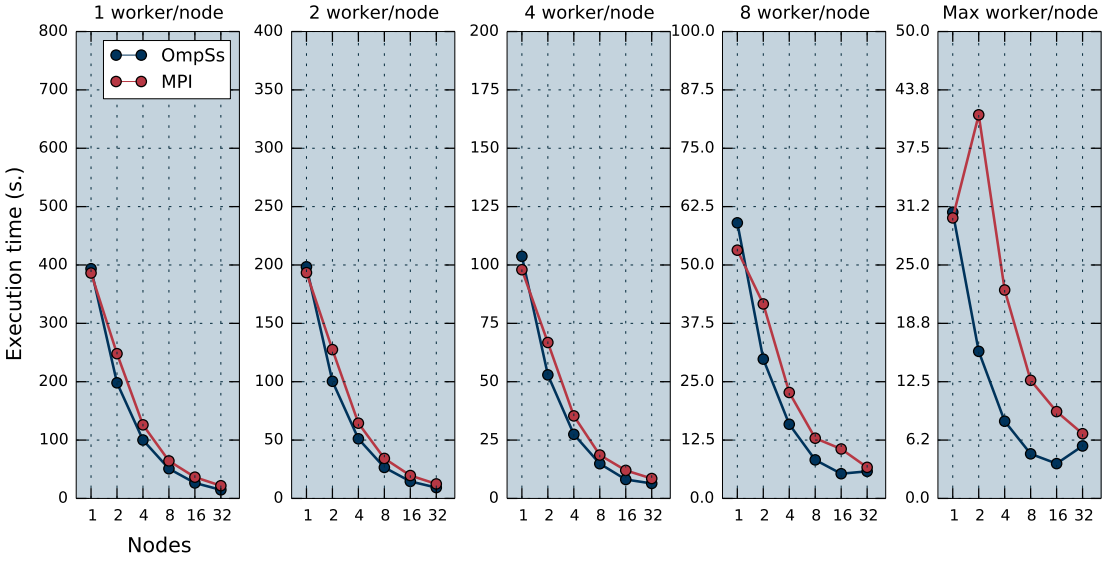


Figure 4.6: Matmul performance comparison between OmpSs and MPI on MareNostrum3

NAS EP

The EP also achieves a good performance on MareNostrum3 as it was the case on MareNostrum2. Figure 4.7 shows the performance results and, as expected, the OmpSs implementation matches the MPI version.

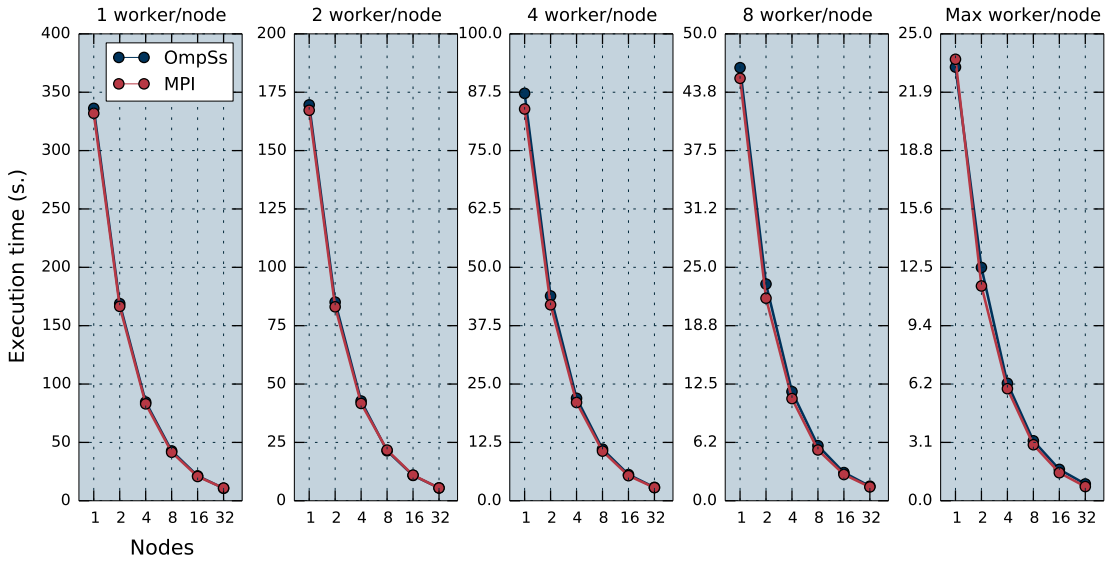


Figure 4.7: NAS EP performance comparison between OmpSs and MPI on MareNostrum3

STREAM

Figure 4.8 shows the performance achieved by the STREAM benchmark. The scalability is comparable in both implementations. The benchmark does not perfectly scale when increasing the number of worker threads, but this is expected since the benchmark is bounded by the memory bandwidth of the system. The performance loss of the MPI version with 1 and 2 worker threads is difficult to explain without knowing the internal details, but experiments without the OpenMP parallelization achieved a similar performance than the OmpSs version.

Sparse LU

The Sparse LU results shown in Figure 4.9 match the performance previously observed in MareNostrum2. The OmpSs version is capable of achieving a better performance thanks to the task-based parallelism. The OmpSs version manages to scale well until 64 worker threads, regardless the number of nodes, are used. From that point, there is no more parallelism to be exploited. The MPI version manages to scale well when increasing the number of worker threads on each node, but the structure of the application limits the performance that can be achieved by adding more nodes.

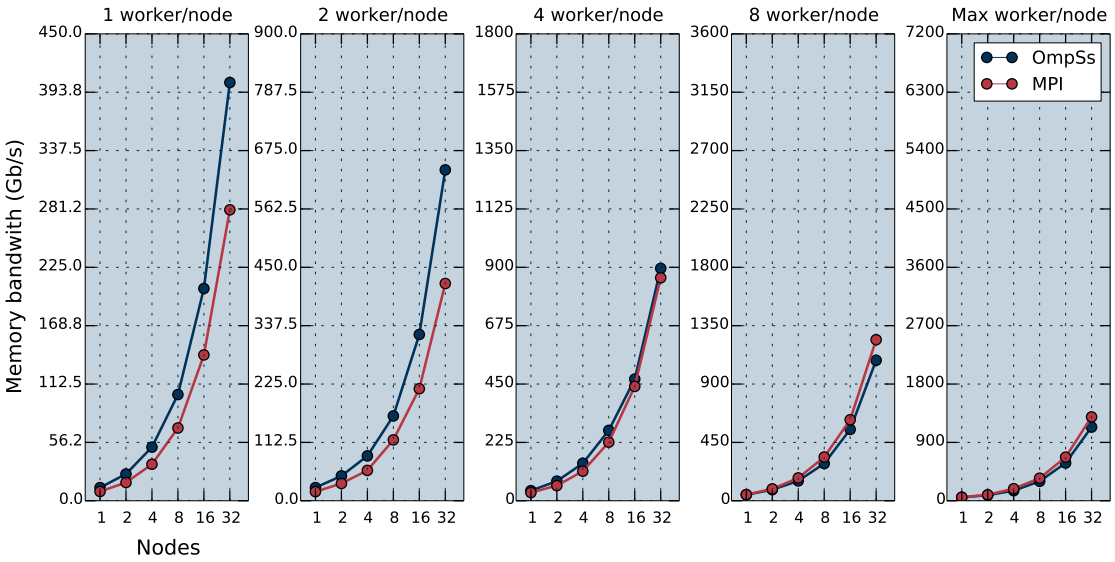


Figure 4.8: STREAM performance comparison between OmpSs and MPI on MareNostrum3

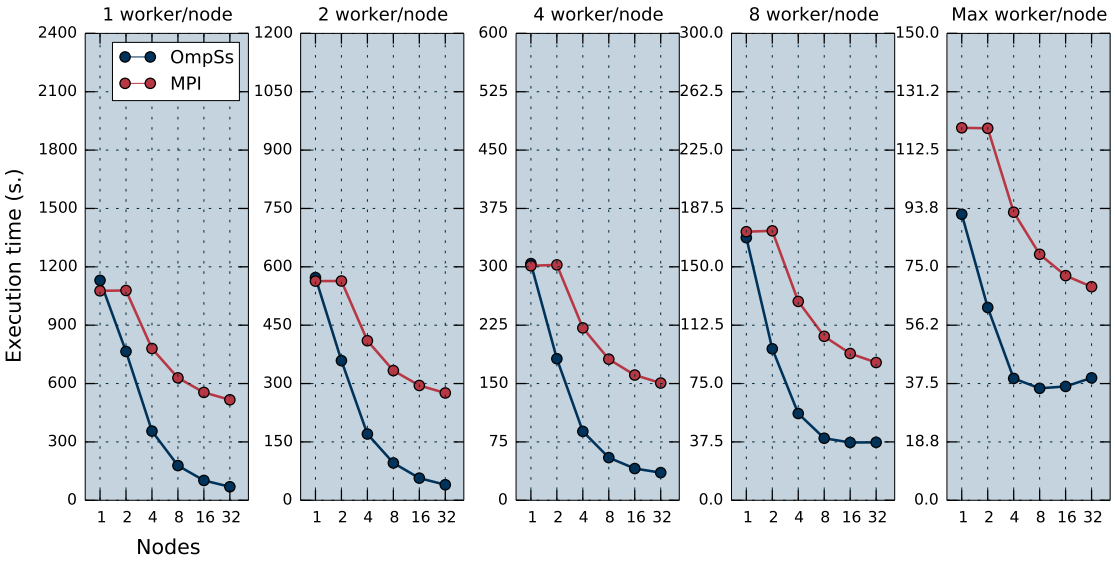


Figure 4.9: SparseLU performance comparison between OmpSs and MPI on MareNostrum3

4.5 Conclusions

This chapter presents the effort to provide support for cluster architectures to the OmpSs programming model. The implemented solution expands the original OmpSs run-time library, Nanos++, to include this functionality. This approach has the benefit that the OmpSs

compiler, Mercurium, does not require any changes, since it already targets the Nanos++ API.

The implementation of the cluster support has been easily done thanks to the generic nature of the Nanos++ design, which has allowed us to easily accommodate the new components that model a distributed environment.

In order to match the execution flow of OmpSs applications, the cluster support uses a master-slave design. The first node (master) starts running the OmpSs application, which eventually will create tasks. The master will instruct some of these tasks to be executed in the remote nodes (slaves).

The network subsystem has been developed to enable the usage of communication networks. It is required by the cluster components to be able to run on distributed environments. The nodes need to send commands and responses among them to coordinate the execution, but also application data must be transferred.

We have developed a set of benchmarks to measure the performance of the system. It is composed by four applications: Matmul, EP, STREAM and SparseLU. Two versions of each application have been implemented, one version using OmpSs and the other using MPI, to compare the performance achieved by both programming models.

The results show that the performance achieved by OmpSs is on par with the performance obtained by MPI and even in some cases OmpSs can outperform MPI. The measured benefits are thanks to the asynchronous parallelism of OmpSs tasks and the fine grain synchronization that can be implemented using the dependencies mechanism. With them, OmpSs allows extracting parallelism more efficiently than using MPI. The proposed design, along with the new developed scheduler, has succeeded at delivering an optimal performance despite being an initial design with no optimizations implemented.

The results of this contribution have been published in [17].

5

OmpSs for clusters of GPUs

The addition of accelerators to High Performance Computing systems has boosted the performance and efficiency of these machines in terms of cost and energy. As already seen Figure 1.1, GPUs have become an important component in modern supercomputers.

These new components have however increased the difficulty of writing applications for these systems. Accelerators are usually programmed using very explicit APIs, which expose a lot of the underlying hardware design to the application developer. Currently there are two programming models that can be used to program GPUs: CUDA and OpenCL. Both of them expose the architectural elements of GPUs. Data must be transferred explicitly to the private GPU memory and kernels have to be issued through command queues. In addition, the code that will be executed on a GPU device must be coded using a specific language other than C or C++, which have to be used to code the CPU part of the applications. Programmers must have a deep understanding of the details of these architectures when dealing with these programming models to be able to write correct programs and to make them obtain a reasonable performance. On top of that, CUDA and OpenCL do not target distributed systems, they must be used in conjunction with other programming models that support them.

OmpSs already targets GPU systems using the CUDA programming model. It offers mechanisms that hide a lot of the aforementioned complexity. OmpSs handles automatically the data movement between the host memory and the GPU memory, leaving to the programmer the only responsibility of writing the code that will be run on the GPU. Nanos++ also pro-

vides techniques to boost the performance of GPU applications. Automatic overlap of data transfers and kernel executions can be enabled without having to modify the source code of the application.

In the same way that has been described in chapter 4, it makes sense to use OmpSs to target distributed systems with GPUs. The mechanisms that OmpSs provides and Nanos++ implements to hide the system complexity can be also applied to this kind of architectures. This chapter covers the implementation of the required support in Nanos++ to handle clusters with GPUs. Section 5.1 defines some characteristics and limitations of the architectures that are being targeted. A brief description of the original GPU support of Nanos++ is presented in Section 5.2. Section 5.3 describes the design and implementation of the changes made to Nanos++ to support clusters of GPUs. The impact in OmpSs of the support for clusters of GPUs are analyzed in Section 5.4. Section 5.5 shows an evaluation using a set of benchmarks implemented in OmpSs and MPI with CUDA shows the performance of Nanos++ and a comparison of both programming models. Finally Section 5.6 tries to compare the productivity of OmpSs against the rest of the programming models used in the performance evaluation. Section 5.7 summarizes the conclusions of this chapter.

5.1 Clusters with GPUs

The architecture this chapter refers to corresponds fundamentally to the defined in the previous chapter (section 4.1) with the addition that GPUs may be present on any node of the cluster. Figure 5.1 illustrates this definition. GPUs have its own private memory space that can be only accessed by local CPUs.

There are no restrictions on the number of GPUs per node that may be supported and the number may not be uniform across the nodes, so the situation where a node has one GPU but the rest has more than one is supported. Also the GPUs may not be the same hardware model, but they must be supported by the CUDA run-time environment. Even with the possibility of allowing flexible configurations, we have worked with the assumption that all nodes of a cluster of GPUs were uniform, that is, they contain the same number of GPUs and CPUs.

5.2 Nanos++ support for GPUs

The implementation of the GPU support in Nanos++ follows a similar structure that the cluster support, as it is based on the generic Nanos++ components described in section 3.2.2.

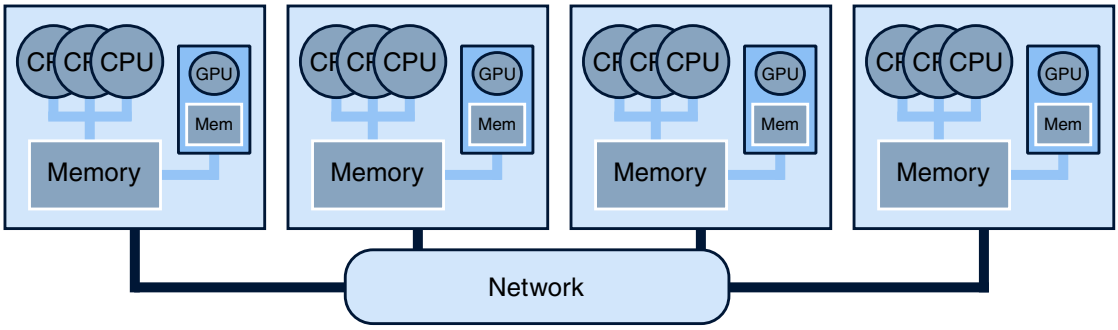


Figure 5.1: Cluster GPU architecture

A processing element is created For each GPU on the system. Each GPU processing element is managed by a SMP thread. These threads can manage the execution of tasks that need to be run on GPU devices. The execution of this kind of tasks follows the usual path. First data is copied to the GPU memory using the data cache and directory. When the data is ready the task execution starts. For the GPU case, the thread will actively wait until the task completes before starting the execution of any other task. As there is one SMP thread per GPU, this wait does not block the execution of tasks in the rest of GPUs. Finally the thread will trigger the transfer of output data if the cache subsystem requires it.

There are two optimizations implemented that boost the performance of OmpSs applications using GPUs. They are the task prefetch mechanism and the overlap of data inputs and outputs. All of them aim to maximize the overlap of kernel execution with data transfers, and they must be used in conjunction to maximize performance. Task prefetch selects, before starting the execution of a task on a GPU, which task will be executed immediately after. With this information, the overlap of inputs can start the data transfers of this next to be executed task. In a similar way the overlap of outputs can always start the transfer of the last executed task, operation that will be overlapped with each kernel execution.

5.3 Nanos++ for clusters of GPUs

The design of Nanos++ has not required many changes in order to support clusters of GPUs. Clusters and GPUs were already supported independently so the implementation changes were focused to make both devices operate together.

5.3.1 Cluster thread

An important change was to adapt the cluster threads (section 4.2.1) to handle tasks marked as both SMP and GPU. With this, GPU tasks created in the master node could be sent to be executed on remote nodes.

5.3.2 Hierarchical address space organization

The most relevant aspect of the GPU cluster architecture is the presence of address spaces corresponding to different kinds of devices. Because of the OmpSs design principles, these address spaces remain hidden to the programmer, who only cares about program data.

From the design point of view, the memory organization can be seen as a hierarchy, where the memories of each node stand at the same level and the GPU memories a level below, each one depending on the memory of its local node. Data may be transferred among memories at the same level. The communication network is the channel used among node memories whereas the PCI Express bus is used among GPUs on the same node. Data can be moved between levels (from the node memory to the GPU memory) using only the PCI Express bus and only among memories located on the same physical node.

The design of Nanos++ to handle this architecture also exploits this hierarchical structure, however the master node defines a new top level that holds the host OmpSs memory space. Remote nodes and GPUs located on the master node form the level below the master main memory. These two levels are managed entirely by the master node, this is, the master node commands any data movement operation among them. The bottom level of this hierarchy is formed by the remote GPUs, which are managed by the slave nodes. Slave nodes do not have total control about their main memory, as this is managed by the master, but they manage the GPUs memory. This design structure is depicted in figure 5.2.

Slave nodes, despite being controlled by the master, have to perform critical actions to keep the coherency of the data when GPUs are present on the nodes. These actions occur when the master node issues a command to a slave node. There are two possible scenarios:

Sending data to a slave node Whenever a slave nodes receives a piece of data associated to a specific memory reference, this data is placed on the main memory of the node. Then, if there are any GPU devices on the node, Nanos++ must check if there were old versions of the same memory reference present. If there are, then the data present on the GPU memory must be invalidated. By doing this, the most recent version, which has been received through the network, will become the only valid version of the data

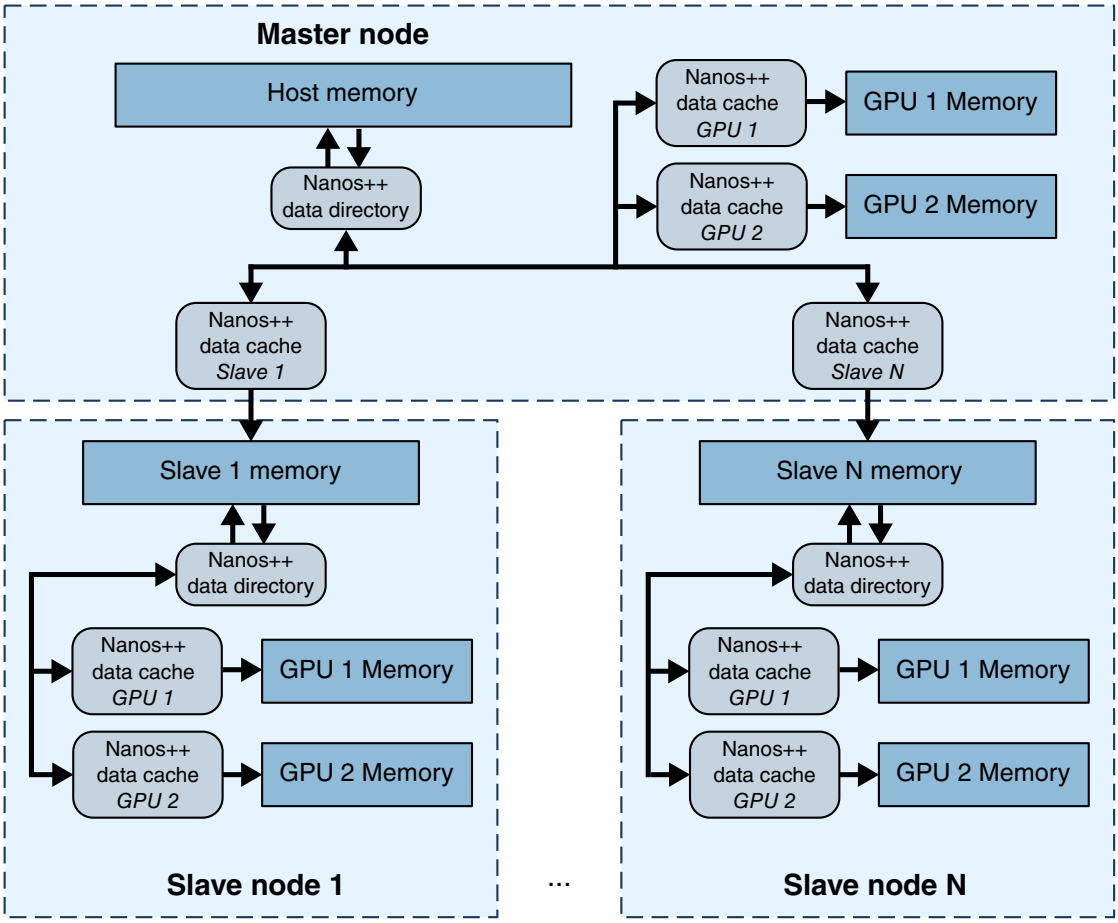


Figure 5.2: Hierarchical memory management structure

for that memory reference. This situation occurs when a piece of data has become replicated by being referenced as read only data by different tasks. At some point a task running on another node may produce new values of this data. Whenever a copy of the new values is received on a slave node, Nanos++ can assume that any references present on GPUs are old and should be discarded. This invalidation schema, which only invalidates data on demand, is more efficient than issuing invalidation commands to all nodes of the system as soon as new data is produced.

Requesting data from a slave node Because the master node is only conscious of the main memory of remote nodes, when it requests data to a slave node, that node must update their main memory to ensure that the proper values are sent back. It may happen that the newest version of a data that has been requested is located in a GPU memory, in

this situation the slave node must copy the data from the GPU memory to the main memory of the node. This scenario is common since tasks tend to produce new data, that may be needed by other nodes.

5.3.3 Optimizations

Two optimizations were implemented to improve the performance of applications on GPU clusters. The data forwarding mechanism starts transferring data to GPU memories as soon as it is produced by non GPU tasks. The task pre-send mechanism tries to overlap the communication with computation to reduce the amount of time that tasks wait for data.

Data forwarding

The data forwarding mechanism aims to reduce the time that GPU tasks have to wait for their data to be available on the GPU memory. The strategy is to send the data produced by SMP tasks to the GPU memory as soon as possible, so when a GPU task requires the same data, it will have been already transferred.

This technique tries to shorten the critical path of the application by advancing the transfer of data from main memory from the main memory to the GPU. Without this technique, Nanos++ issues the data transfers when a task is about to be executed, which means that this operation is in the critical path. The data forwarding mechanism tries to exploit the asynchronous nature of GPU data transfers by starting the transfers as soon as the SMP tasks finish their execution. Thanks to being asynchronous the critical path of Nanos++ is not increased significantly, and the actual transfer operations will be performed in parallel with other Nanos++ actions.

This strategy is totally speculative and some applications may end transferring more data than what would have been expected without using it. However the cost of transferring data is not high as the movement is controlled by DMA engines which do not affect to the CPU or GPU processors. The mechanism can be counter-effective when SMP tasks generate data that is not directly needed by GPU tasks, but this scenario is uncommon. Also when the mechanism sends useless data, the cost of replacing the data is negligible, so the performance is not degraded. Lastly, the data forwarding is only enabled when there is only one GPU per cluster node.

Task pre-send

A cluster-level task pre-sending mechanism was implemented to allow the overlapping of communication and computation. Exploiting this strategy can help to increase the performance of applications since data transfers can be run in parallel while all CPUs and GPUs perform computation, this can reduce the time spent by CPUs and GPUs waiting for data.

The mechanism implemented tries to exploit the overlapping possibilities when executing several tasks on a given node. Nanos++ controls the number of running tasks on each node. Originally Nanos++ sent as many tasks as available CPUs were present in the node. The pre-send mechanism overrides this and keeps sending tasks to the remote node even if the number of running tasks surpasses the number of available CPUs or GPUs. With this schema Nanos++ can start sending the data of the tasks that will be executed on a node while all the CPUs in it are busy, with the hope that when these pre-sent tasks start the execution, their required data is already available. This schema reduces the idle time of CPUs on the remote nodes and removes data transfers from the critical execution path, reducing the total execution time of the application.

The amount of tasks that are pre-sent is a parameter that can be set by the user, and can be defined for both types of tasks, SMP and GPU.

5.4 Impact in OmpSs

No changes have been introduced in OmpSs in order to support the changes described in this chapter. GPU applications developed with OmpSs can run transparently on a cluster environment without modifications.

5.5 Performance evaluation

In the field of High Performance Computing, the performance that the programming model can deliver is a critical aspect of it. The following sections cover the description of the environment, the benchmarks used, the experiments done and the results obtained.

5.5.1 Methodology and environment

Using the same approach as in the previous chapter, we have measured the performance using a set of benchmarks developed in OmpSs and we have compared it against the performance of the same applications implemented using MPI and CUDA. MPI is considered

the most relevant programming model when it comes to distributed programming in high performance environments, and CUDA is the official programming model when developing applications for NVIDIA® GPUs.

The experiments have been run in a GPU cluster environment. Each node had two Intel Xeon E5620 processors with four cores each and one GTX 480 GPU with 1.5 GB of memory and peak performance in single precision 1.35 TFLOPS and a peak memory bandwidth of 177.4 GB/s. The total system memory of each node was 25 GB. The nodes were interconnected with a QDR Infiniband network with a bandwidth peak of 8 Gbits/s. OmpSs was compiled to use the native Infiniband conduit for GASNet.

All the codes where compiled with our Mercurium compiler with optimization $-O3$ level. GCC version 4.3.4 and CUDA version 3.2 were used as back-end for the SMP and GPU parts respectively.

5.5.2 Benchmarks

The applications selected were Matrix Multiplication, the STREAM benchmark, a Perlin Noise generator and a N-Body simulator. The details of each application are detailed in the following lines:

Matrix Multiplication A representative benchmark for linear algebra codes, it performs a matrix multiplication between two source matrices and stores the result on a third matrix. Each matrix contained 12288x12288 single precision floating point elements. The computation is performed using the CUBLAS[80] sgemm call and it is divided in blocks of 1024x1024 floats each. For the MPI+CUDA version, we implemented a Summa algorithm as shown by Geijn and Watts [47] and it also used the CUBLAS kernel. While the MPI+CUDA implementation did not implement any techniques that could improve its performance, we believe it is representative to compare it in order to illustrate the difficulty of exploiting the whole potential of CUDA. For the OmpSs version, we developed three versions of the benchmark, each of them initialized the data using a different schema. The most basic version, named *seq* in the results, performed the initialization in the sequential part of the application. The second version, named *smp*, parallelized the initialization process with tasks that were executed on CPUs. Finally the third version, named *gpu*, also parallelized this part, but the tasks were executed on GPUs. The three different versions allowed us to measure the impact of the data forwarding mechanism (Section 5.3.3). The mechanism was exploited by the *smp* version of the benchmark, where the data generated by the tasks was au-

tomatically sent to the GPUs, the *seq* version did not use it due to the absence of the initialization tasks, also the *gpu* version was already generating the data in the memory of the GPUs. The performance is measured using the GFLOPS achieved.

STREAM The benchmark measures the performance of the memory. Both OmpSs and MPI+CUDA implementations were developed from the original source code. The OmpSs version was adapted from the OpenMP STREAM with new handmade CUDA kernels. The MPI+CUDA version was based on the original MPI with the same handmade kernels used in the OmpSs version. The application allocated 768MB per GPU in each version. Data was initialized in the GPU memory. The performance is reported in GBytes/s.

Perlin noise The benchmark is an image filter that generates noise to provide improved realism in computer generated images. It is representative of image processing applications. We have used an image of 1024 x 1024 pixels. The benchmark measures the MPixel/s that the implementation can process.

N-Body The N-Body simulation computes the gravitational interaction of a system of different bodies. The CUDA kernel comes from a set of NVIDIA® examples. After each iteration of the system the data from the previous round must be distributed to all GPUs. We have simulated 10 iterations of a system with 20000 bodies. The benchmark measures the number of particles processed each second.

5.5.3 Experiments

The aforementioned benchmarks were executed with different number of nodes to measure the scalability of both programming models. The data size used was the same for all configurations. The results were obtained using the Nanos++ parameters that delivered the best performance. The baseline to compute the speed-up was obtained by running sequential versions of the same applications.

We used from 1 to 8 nodes for each application. Using a higher amount of nodes would have required to use bigger problem sizes. In our case, we set the problem size to fit on a single GPU in order to avoid data transfers caused by filling the GPU memory. These transfers are part of the invalidation mechanism, which is not evaluated in this work.

A deeper analysis was done for the Matrix Multiplication benchmark. We tested different Nanos++ parameters in order to see the impact of the available options and techniques that were implemented in the run-time. These options include the usage of slave-to-slave data transfers (see Section 4.2.6), the amount of tasks that we allow to be present to each node

and the three initialization types implemented.

5.5.4 Results

Matrix Multiply

Figure 5.3 shows the performance obtained with the Matrix Multiplication application with different configuration parameters. The X axis contain the different number of nodes used by each execution, for each number of nodes, there may be two groups depending on if the transfers between slave nodes was enabled or not (*MtoS* means that no slave-to-slave transfers were done, *StoS* otherwise). Finally the last grouping describes the initialization type that was done before computing the matmul, *seq* means that all the data initialization was done sequentially on the master node, *smp* means that the initialization was parallelized using OmpSs tasks that were executed on the CPUs of the cluster, and *gpu* means that the initialization parallelized with OmpSs tasks and done by the GPUs of the system. Also, for each setup we present the performance achieved by using three different values of the task pre-send mechanism.

The results show that slave-to-slave transfers are a must to achieve a proper scalability since they greatly reduce the data that must be moved around the network. Initializing the data in parallel also turns out to be a critical factor, since it reduces the amount of data transfers during the execution because data is then produced in the remote nodes or GPUs. The *smp* initialization provides in general better results than *gpu* approach since moving data to remote nodes is more expensive when the data is available only on the GPUs. In the *gpu* case, moving data to a remote node requires first a transfer from GPU memory to the host memory, and then the final transfer through the network. In the *smp* version, the data forwarding mechanism (section 5.3.3) automatically transfers the data to the GPU and it also overlaps this movement with other computation.

The pre-send technique also provides a significant impact in performance. It helps to improve scalability, specially when increasing the number of nodes. Pre-send must be used along with slave-to-slave transfers in order to not overload the master with network operations that disrupt the overall execution. Finally, figure 5.4a shows the performance of the best setup of OmpSs it is compared against the Matrix Multiplication implemented using MPI+CUDA. While the MPI obtains better performance with 1 and 2 nodes, the techniques implemented by our run-time outperform the MPI+CUDA version.

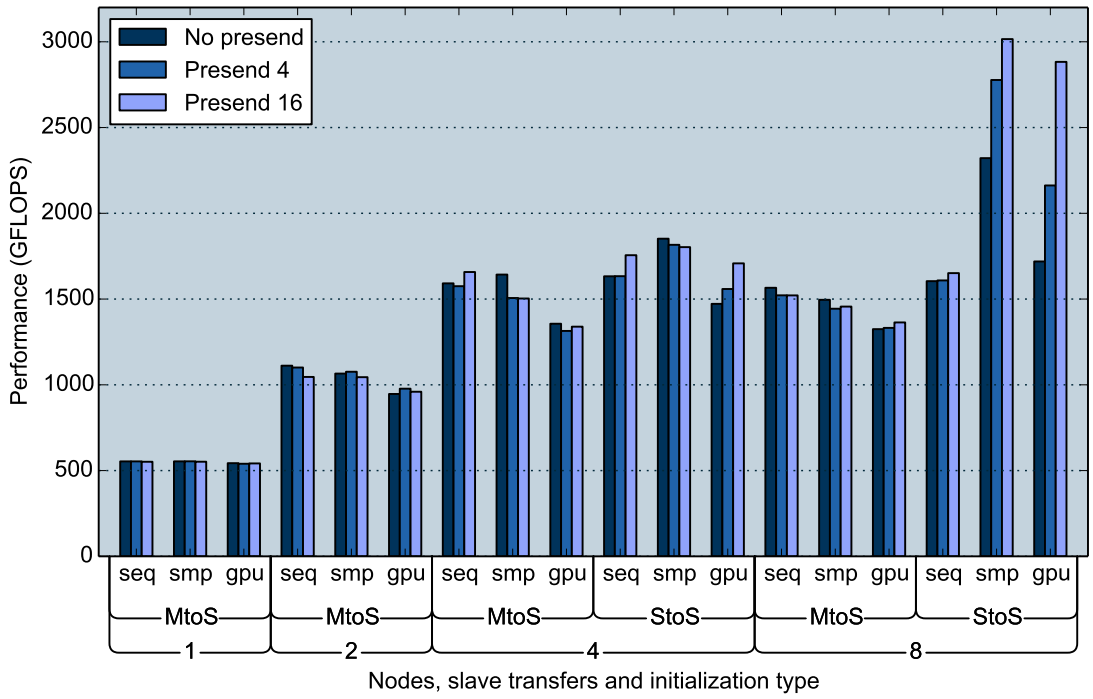


Figure 5.3: Matmul performance results on the GPU cluster environment

STREAM

The STREAM benchmark results are shown in figure 5.4b. The application scales perfectly since there are no data transfers among the nodes of the cluster, thus it achieves a good performance using MPI+CUDA and OmpSs.

Perlin noise

This benchmark also achieves a good performance with all configurations, as can be seen in Figure 5.4c. The data used by the benchmark can be divided evenly among the GPUs of the cluster, which allows for a regular parallelization with good scalability results. The algorithm iterates over the same data so the scheduler can easily send each task to the appropriate node and GPU. With this, the execution does not require sending data among the nodes except when distributing the data on the initialization stage. The MPI version also does not require communication between iterations and can achieve a perfect scalability.

N-Body

Figure 5.4d shows the performance achieved on the cluster by the N-Body application. The benchmark performs a series of iterations where all data produced by a task on a GPU must be sent to the rest of the GPUs. This all-to-all pattern is better handled by the asynchronous nature of OmpSs, which can overlap the execution of different iterations. Also, this benchmark benefits from the *write-through* data cache policy of Nanos++ (see Section 3.2.3), since updating the main memory of the node with the results computed by the GPU fastens the all-to-all transfers. The MPI+CUDA code follows a more rigid approach and have to synchronize after each iteration. This causes the OmpSs performance to be better with 4 and 8 nodes.

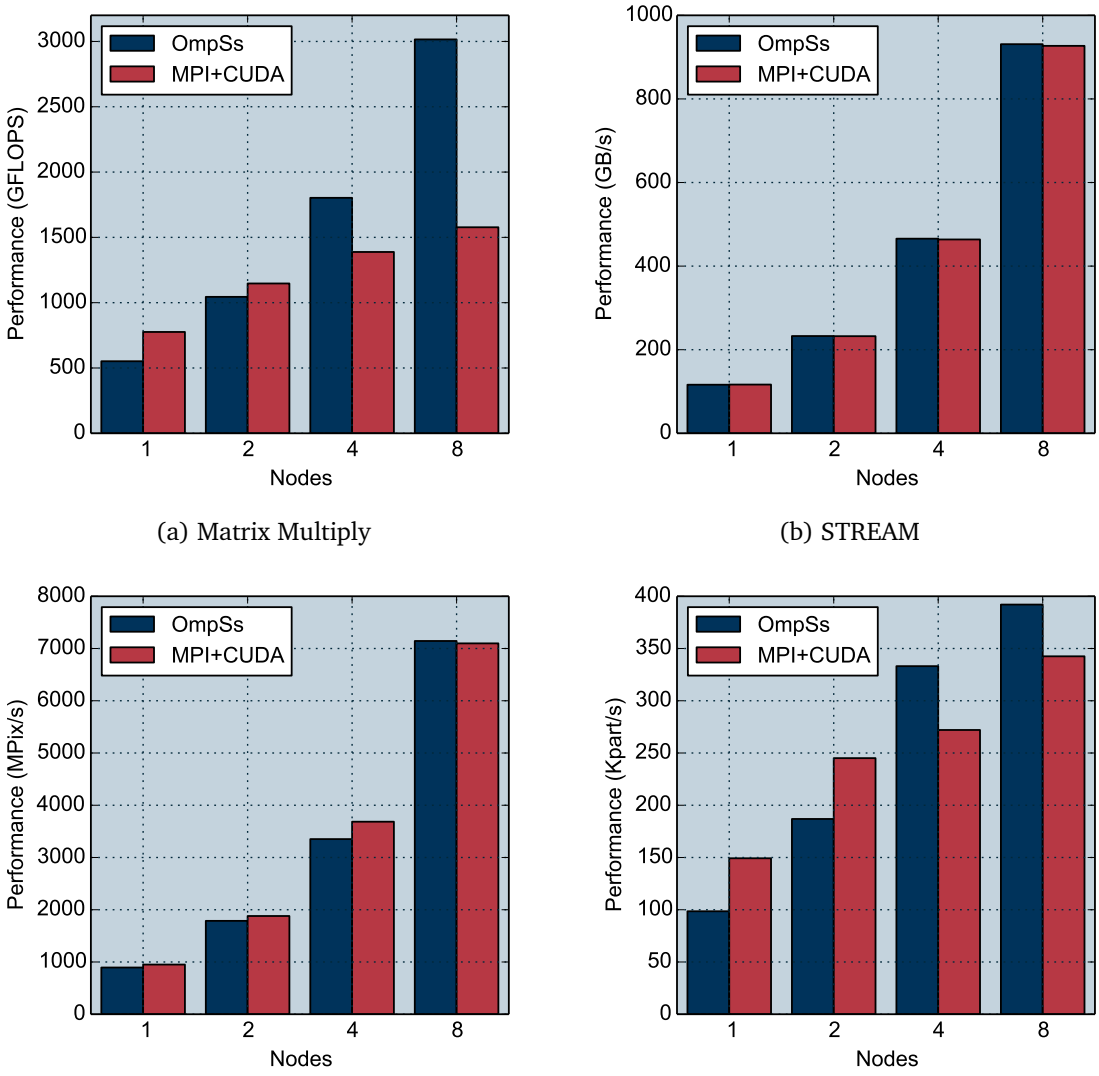
5.6 Productivity evaluation

Productivity is key for programmers to effectively exploit the resources available in current systems. Evaluating the productivity of a programming model is very complex, involving development times and effort spend in the work. However, we propose a simple metric to try to model the productivity of OmpSs based on the number of code lines that each implementation contains.

After developing the benchmarks in CUDA, MPI+CUDA and OmpSs, we have counted the number of useful lines of code that result in each version, and computed the percentage of variation between them. Table 5.5 shows these results. For each benchmark and version, the number of lines is shown, and the percentage increase with respect the serial version is indicated in parenthesis.

As it can be observed, the common trend is that the CUDA version adds some lines of code, and the MPI+CUDA version even more. By contrast, the increase in the number of lines is lower when using OmpSs. CUDA versions add lines of code to initialize the GPU device, allocate memory on it, copy the data between the host memory and the GPU, prepare the kernel parameters, and invoke the kernel. On top of this scheme, the MPI versions always add new lines to communicate messages between the nodes, and barriers when synchronization points are needed.

When using OmpSs the additional lines of code come from the use of the compiler directives. Two lines of code are usually added per each task declaration. Additional lines may be needed if the task uses many memory references. It must be noted that the OmpSs parallelization can also be done incrementally, which eases the parallelization process, since



(a) Matrix Multiply

(b) STREAM

(c) Perlin noise

(d) N-body

Figure 5.4: Scalability comparison between OmpSs and MPI+CUDA

the programmer can work on small parts of the application independently. MPI or CUDA are more intrusive in that aspect, and changing a small part of the application may require big changes.

Benchmark	Serial	CUDA	MPI+CUDA	OmpSs+CUDA
Matmul	643	683(+6.2%)	696(+9.2%)	677(+5.2%)
STREAM	378	485(+28%)	496(+31%)	420(+11%)
Perlin	562	761(+35%)	788(+40%)	632(+12%)
N-Body	888	908(2.2%)	1049(+18%)	908(2.2%)

Figure 5.5: Comparison of total number of lines in Serial, CUDA, MPI+CUDA and OmpSs+CUDA versions of the benchmarks (in parenthesis, the percentage of increment, with respect to the Serial version)

5.7 Conclusions

Extending the cluster support of Nanos++ to handle clusters of GPUs was a obvious step given the increasing popularity of these systems in HPC environments, but also because OmpSs already provided support for single-node multi-GPU systems.

A few changes in the run-time were needed in order to support this new architecture. The most relevant were caused by the addition of the memory of the GPUs, which increased the complexity of the memory management components of Nanos++. The memory of the system could be seen as a hierarchy were the top level is formed by the main memory of each node, and, for each node, the level below corresponds to the memories of the local GPUs. This structure is used by Nanos++ to manage the coherence of the whole memory. The master manages the memory of the slave nodes and its local GPUs, and the slaves manage the memory of their local GPUs.

In addition to these changes, two optimizations were implemented to improve the performance of applications. The data forwarding mechanism tries to send the data produced by the CPUs to the memory of the local GPU, with the hope that this data will be required by tasks executed on that device. The benefit of this is that data transfers are started sooner, with respect of the normal execution path, which issues the data movement when the task begins its execution. Therefore the critical path of the application becomes shorter. While this approach is completely speculative, the worst case scenario does not usually add a performance penalty, since discarding the pre-sent data does not require any costly operation.

The task pre-send mechanism also tries to overlap communication with computation but at the network level. This feature is particularly critical in order to distribute data and tasks more efficiently through the cluster, since GPUs can process some tasks way faster than a CPU

would, thus they may require more data to achieve optimal performance. The mechanism can be configured by the programmer in order to adapt it to the needs of any application.

An evaluation of the productivity that OmpSs provides when developing applications for clusters of GPUs is also detailed in this chapter. We compared the number of lines required to develop a series of benchmarks in OmpSs with respect to developing them using MPI and CUDA, the most widely used programming models for these systems. For all benchmarks, OmpSs is the programming model that requires less additional lines of code with respect to the serial version. Also, with OmpSs the same code can be used on a multi-GPU system or on a cluster-GPU system without modifications.

Lastly we have presented a performance evaluation where we measured the scalability of several applications executed on a cluster-GPU system. We tested different execution parameters to measure the impact of our optimizations. We also compared the performance of the OmpSs applications against the achieved by same applications implemented using MPI and CUDA. The overall results show that OmpSs can match and even improve the performance of these applications, while offering an easier way to develop applications for these systems.

The contributions of this chapter have been published in [18].

6

Regions of data in OmpSs

One of the critical aspects of developing applications for distributed environments is distributing and managing the program data efficiently. Some applications may require complex distribution and/or access patterns in order to obtain good performance, therefore programming models need to offer tools to ease this part of the development process.

Explicit programming models hardly offer mechanisms to ease the task of the programmer. The services that are usually offered are meant to explicitly transfer data between memories, which do not actually reduce the complexity of the problem. Data must be distributed manually and the access code is then tied to the selected distribution. A clear example of these explicit programming models is MPI, which includes many different types of calls in its API to send messages between nodes, but leaves the responsibility of arranging and moving the data to the programmer.

By contrast, implicit programming models include some features that allow the specification of complex partitioning schemes. They can be used to distribute complex data structures among different nodes, typically multi-dimensional arrays. Also the programmer can easily set the strategy to do the distribution of the data. Examples of programming models with this kind of features can be the Unified Parallel C (UPC) [28] or X10 [23].

OmpSs does not offer specific mechanisms to partition the data as it offers a global address space and therefore such concept can not be made explicit. Nonetheless the partitioning can be done automatically by Nanos++ thanks to the task data specifications, as described in section 3.2.3. The OmpSs syntax allows to easily define complex access patterns

to sub-regions of arrays (section 2.5.2). However, the original implementation of Nanos++ did not fully support this syntax. As a result, some applications could not be programmed using OmpSs, and some others had sub-optimal code in order to work correctly.

This chapter presents the design and implementation of a new mechanism that handles the memory references in the data directory and cache subsystems. This new design fully supports the OmpSs syntax specification, and it allows complex specifications of data. We usually refer to these complex specification as *data regions* or *regions*, since they allow to specify sub-regions of multi-dimensional arrays. Providing this support improves the programmability of OmpSs and allows the implementation of a much wider range of applications. Section 6.1 describes the limits of the original implementation. Section 6.2 shows the importance of providing this support in OmpSs. Section 6.3 describes the implementation of the regions support in Nanos++, how regions are internally represented and how the overlaps are computed. Section 6.4 discusses some effects that the region support caused in the cluster support. Finally Section 6.5 presents an evaluation of the implementation presented during this chapter using applications that require regions to be properly written in OmpSs. We also evaluate the performance of the system against versions of the same applications implemented using MPI. Section 6.6 summarizes the conclusions of the chapter.

6.1 Limitations of Nanos++

The original Nanos++ design did not fully support the memory reference syntax of OmpSs. Memory references are used by two important features: the task data dependencies and the task data usage.

6.1.1 Dependencies

Originally the dependence subsystem used only the base address of the memory reference when doing the dependence analysis. This decision caused that dependencies based on array sections were not computed as the specification stated. It was not possible to specify a task that worked on a certain column of a matrix since they are not stored contiguously when using the C programming language (in the case of FORTRAN, the rows of a matrix are not contiguous in memory). These limitations sometimes could be overcome by reshaping the data manually, or by adding explicit synchronization directives.

Pérez et al. [86] proposed an improved implementation to manage regions for tracking dependencies between tasks. They implemented a new dependence tracking system that

detected overlapping situations and thus offered much more control than the original. This proposal has some limitations, because it uses a very compact representation but only capable of containing precise information under certain circumstances. The base address of the data regions must be aligned to the size of the region, and the region size must be a power of two. If these conditions are not met, then dependencies are not precisely handled and the execution may be sub-optimal, this, however, does not cause an incorrect execution in any case.

6.1.2 Data directory and cache

Similarly to the dependencies subsystem, the data directory and cache only used the base address and size of specified objects. In addition, it was incorrect if memory references of two different tasks overlapped with other. There was no way to specify non-contiguous data, and two references with the same base address could not be specified of different size.

6.2 Impact of data regions in OmpSs

Offering regions support is important because it increases the flexibility of the OmpSs programming model and allows more efficient and simple implementation of applications.

To demonstrate this, we can observe the differences between two different implementations of a simple matrix multiplication algorithm. Figure 6.1 contains the implementation without using the proposed data region support. In this implementation the arrays have been declared as a 3-dimensional arrays in order to have contiguous data blocks of `BS`×`BS` elements. Given this data layout, the references in the `in` and `inout` clauses refer to contiguous blocks of data (this requirement is needed without the proposed mechanism). In comparison, figure 6.2 shows the code that takes advantage of the proposed feature. It has some clear benefits over the code in figure 6.1. First, the array declaration is simpler since forcing the layout to have contiguous blocks is no longer required. The references in the `in` and `inout` clauses now refer to data that is not contiguous in memory (a block of `BS`×`BS` elements is scattered since the array declaration has a leading dimension of `N` elements). In addition, multi-level parallelism can be used now to implement the matrix multiplication, since the programmer can reference complete rows and columns of the different arrays, which is needed to specify the top level tasks (lines 7-9, in the code of figure 6.2). While these examples show statically allocated objects, the proposed mechanism also works with dynamically allocated data.

```
1  double A[NBLOCKS][NBLOCKS][BS BS];
2  double B[NBLOCKS][NBLOCKS][BS BS];
3  double C[NBLOCKS][NBLOCKS][BS BS];
4
5  //matmul main loop
6  for ( i=0; i < NBLOCKS; i++ )
7     for ( j = 0; j < NBLOCKS; j++ )
8        for ( k = 0; k < NBLOCKS; k++ )
9           #pragma omp target device (smp) copy_deps
10          #pragma omp task in(A[i][k], \
11                              B[k][j]),\
12                          inout(C[i][j])
13          matmul_blk(A[i][k], B[k][j], C[i][j], BS);
```

Figure 6.1: DGEMM OmpSs implementation without regions support

```
1  double A[N][N], B[N][N], C[N][N];
2
3  //matmul main loop
4  for ( i=0; i < N; i += BS )
5     for ( j = 0; j < N; j += BS )
6        #pragma omp target device (smp) copy_deps
7        #pragma omp task input(A[i;BS][0;N], \
8                               B[0;N][j;BS]),\
9                         inout(C[i;BS][j;BS])
10       for ( k = 0; k < N; k += BS )
11          #pragma omp target device (smp) copy_deps
12          #pragma omp task in(A[i;BS][k;BS], \
13                              B[k;BS][j;BS]),\
14                           inout(C[i;BS][j;BS])
15          matmul_blk(&A[i][k], &B[k][j], &C[i][j], BS, N);
```

Figure 6.2: DGEMM OmpSs implementation, with regions and multi-level parallelism

Another example can be seen in figure 6.3, in this case it corresponds to a FFT1D implementation using a Cooley–Tukey algorithm, to allow the data to be distributed on a two dimensional array, which splits the main problem by performing a FFT1D on each row of this distribution. These sub-FFT1D computations can run on parallel, however, some extra transpositions are done to achieve the final result. The data region support provides an easy way to implement the transposition using tasks that work on different blocks of the matrix. In addition keeping a proper data layout is required by the `fft1d_row` call.

It also must be noted that these examples are capable of being executed on cluster environments, with many different configurations of nodes and accelerators. Being able of

```
1  double ( A)[N][N];
2  size_t BS = getBlockSize();
3  A = malloc( N N sizeof(double) );
4
5  // first transpose
6  for ( i=0; i < N; i += BS )
7     for ( j = 0; j < N; j += BS )
8        #pragma omp target device (smp) copy_deps
9        #pragma omp task inout( A[i;BS][j;BS], \
10                               A[j;BS][i;BS])
11       transpose_blk(A[i][j], A[j][i], BS, N);
12
13 // first row fft
14 for ( i=0; i < N; i += BS )
15    #pragma omp target device (smp) copy_deps
16    #pragma omp task inout( A[i;BS][0;N])
17    for ( j = i; j < i + BS; j++ )
18       fft1d_row(A[j][0], N);
19
20 // transpose and twiddle
21 for ( i=0; i < N; i += BS )
22    for ( j = 0; j < N; j += BS )
23       #pragma omp target device (smp) copy_deps
24       #pragma omp task inout( A[i;BS][j;BS], \
25                               A[j;BS][i;BS])
26       transpose_tw_blk(A[i][j], A[j][i], BS, N);
27
28 // second row fft
29 for ( i=0; i < N; i += BS )
30    #pragma omp target device (smp) copy_deps
31    #pragma omp task inout( A[i;BS][0;N])
32    for ( j = i; j < i + BS; j++ )
33       fft1d_row ( A[j][0], N);
34
35 // last transpose
36 for ( i=0; i < N; i += BS )
37    for ( j = 0; j < N; j += BS )
38       #pragma omp target device (smp) copy_deps
39       #pragma omp task inout( A[i;BS][j;BS], \
40                               A[j;BS][i;BS])
41       transpose_blk(A[i][j], A[j][i], BS, N);
```

Figure 6.3: FFT1D OmpSs implementation

referencing data with this expressiveness is desirable because the run-time environment will be able to adapt the data distribution of the application according to the actual hardware configuration.

6.3 Precise regions of data in Nanos++

Supporting precise regions in Nanos++ created two important challenges:

1. Selecting an internal representation that allows to compute region overlaps fast enough to not impair the performance of applications.

2. Adapt the memory-related Nanos++ subsystems to handle regions of data instead of simply chunks of contiguous memory.

6.3.1 Representing regions

According to the OmpSs specification, regions are defined using a syntax similar to the one used to declare and access array types in the C programming language. A region is defined using a base object, which we will refer to as a *program-object*. This object will typically be an array with an arbitrary number of dimensions. Array subscripts are used to specify which elements of each dimension are covered by the region. For each dimension, two integer values define the first element and the number of elements covered on that dimension. These values must always represent a range where all of its elements fall within the limits of the corresponding dimension. We will refer to these pair of values as *dimension-access*. Figure 6.4 shows an example of code which defines a region. A two dimensional array is used as *program-object*, therefore a set of two *dimension-access* elements must appear in the region definition.

Regions can be dynamically defined using *dimension-access* containing program variables. This provides a lot of flexibility to the programmer, who can defer until the execution of the program the computation of the region size and shape. Also, it is legal that different regions overlap with each other. By overlap we refer to the fact that different regions access the same elements of a *program-object*.

How to internally represent regions is a critical design decision. The run-time library will need to perform operations between many different regions. For example, the run-time library may need to check if two or more regions overlap with each other. In addition, these operations will be part of the critical path of the whole execution, therefore the representation must allow a fast implementation of these.

In order to achieve a simple design, we have set a few limitations to our system. These limitations are:

```
1 double M[8][8];
2
3 void compute() {
4    #pragma omp task inout(M[0;6][0;4])
5    {
6       for (int j=0; j<6; i++) {
7          for (int i=0; i<4; i++) {
8             process_elem(M[j][i]);
9          }
10      }
11   }
12 }
```

Figure 6.4: A task defining a region. The *program-object* of the region is the array `M`. It is a two dimensional array so two *dimension-access* elements are needed to declare the region, `[0;6]` defines the range of accessed elements of the outer dimension and `[0;4]` defines the range of accessed elements of the inner dimension

- Regions are associated always to a single *program object*. Regions covering more than one object are not supported.
- All regions that refer to the same object must be consistent with each other, that means that all of them must provide the same number of dimensions of the given object and access valid elements of these dimensions.

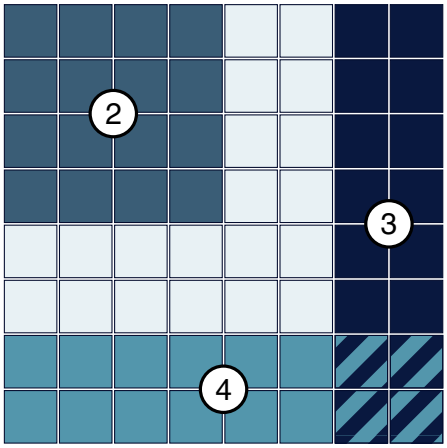
These limitations have a very low impact to the programmer, since both describe situations that generally would be considered programming errors. In addition, the run-time can detect these cases and emit an error message to inform the user.

Region identification

Program objects are registered at run-time and are identified by their base address. Each region is associated to a single *program object*. A region is defined by a set of *dimension-access* objects, one for each dimension of its referred object. In order to have a simple way to identify regions an integer value, unique within its *program object* is assigned to each region. We will refer to this value as the *region identifier*. Figure 6.5 shows a sample code that contains three tasks and each one declares an access to a region of a program object (figure 6.5a) and the layout of the regions defined on the program object with the internal identifiers (figure 6.5b).

```
1  double M[8][8];
2
3  void compute() {
4     #pragma omp task inout(M[0;4][0;4])
5     {
6        ...
7     }
8     #pragma omp task inout(M[0;8][6;2])
9     {
10       ...
11    }
12    #pragma omp task inout(M[6;2][0;8])
13    {
14       ...
15    }
16 }
```

(a) A function that creates three tasks, and each one defines a region using the program object M.



(b) The representation of the regions defined by the function on the left, with their corresponding internal identifiers.

Figure 6.5: Regions are assigned an internal identifier which is incremented as new regions are declared. Region number 2 corresponds to the first declared region on the code, number three is the second region and finally region number 4 is the last region declared. Regions 3 and 4 overlap since four elements are accessed by both regions.

The generation of the identifier is done using a dictionary implemented using a *trie* (or *digital tree*). Each node of the tree contains a value of each of the *dimension-access* values (first element and number of elements covered). With this, a region information is stored as the path from the leaf to the root of the tree. The leaf node contains the *region identifier*. Figure 6.6 illustrates the data structure that results from registering the regions defined in the previous example (figure 6.5). The first two levels of the tree contain the range defined by the dimension-access of the outer dimension whereas the two following levels contain the information of the outer dimension. This data structure can be used with any number of dimensions, the tree will be created with the appropriate number of depth levels to contain the complete region information. The usage of this tree eases the implementation of the mechanism to check the existence of a given region and also makes this process efficient. The cost of doing this operation scales with the number of dimensions, not with the number of registered regions.

The *region identifier* provides a simple way to identify a region within an object, and

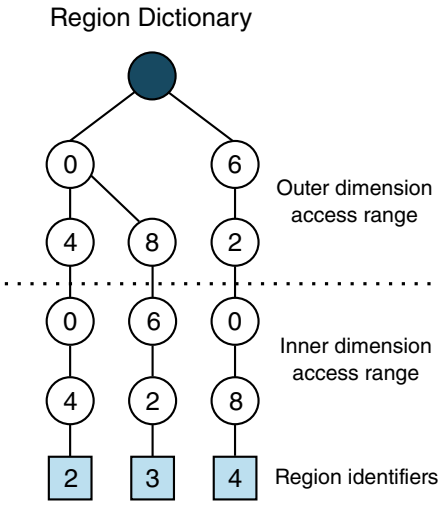


Figure 6.6: Representation of the internal tree that holds the region information and the leaf nodes corresponding to the region identifiers. Starting from a region identifier node, the information of the region can be retrieved by navigating to each parent node until reaching the root.

it also is used to access the region meta-data, which is stored using an array of meta-data entries. The region meta-data contains the location information of the related data, that is, in which memory address spaces (GPUs, nodes) is stored.

Region overlaps

Regions may overlap. To detect which regions overlap with another region, we maintain an *intersection map*. An *intersection map* is actually composed by a set of maps, one for each dimension. Each of these contains the information about how regions cover the elements of the given dimension. They map *dimension-access* elements to sets of *region identifiers*. These maps can be seen as a projection of the present regions onto a one dimensional space. Figure 6.7 depicts the contents of the intersection map corresponding to the regions declared on the given matrix.

To compute the set of regions that overlap with a given region, we must first obtain the sets of *region identifiers* using each *dimension-access* object of the region. The intersection of all obtained sets is the set of overlapping region identifiers.

Example: assuming the state described by figure 6.7, a new region, `M[0;2][5;2]`, is registered and the overlap detection process goes as follows. The first dimension, `[5;2]`, is used to access the *Dimension 0* map, which returns the set `{3,4}`. Then the second dimen-

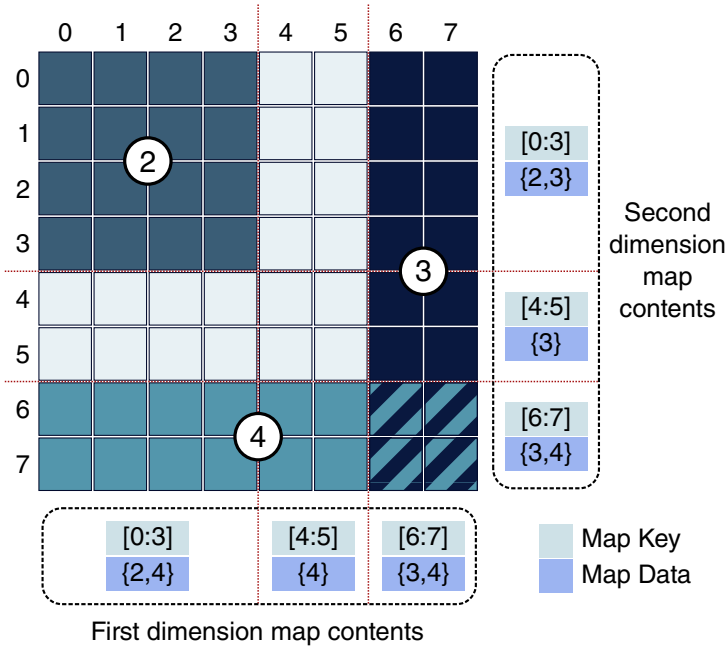


Figure 6.7: Intersection maps

sion, `[0;2]`, is used on the *Dimension 1* map and the result is `{2,3}`. The intersection of both result sets is `{3}`, meaning that only region 3 overlaps with the new region.

In terms of cost, in order to compute the intersections of a region, the proposed mechanism has to perform $D \times \log_2 n$ operations, where D is the number of dimensions of the object, and n is the number of entries in the maps of the *intersection map*. We expect the number of dimensions will be a low number, rarely being higher than 5. With this, the cost scales well when having a high number of entries in the maps. Regarding the memory consumption, the number of entries is generally proportional to the number of registered regions.

Distributed region information

Region meta-data is generally accessed by the run-time when it starts the execution of a task since it requires to know where its needed data is located. Since the described design is a centralized one, it can suffer from contention problems when running with several threads at the same time, since all of them will try to access to the *intersection maps* if they access the same *program-objects*.

To mitigate this contention, we exploit the dependence subsystem of OmpSs. When a task depends on another one, it is likely that the regions that were accessed by the preceding

task will be also accessed by the successor task. We transfer the regions meta-data of a task to its successors hoping that they will be able to use it instead of having to request it to the centralized data structures.

Memoizing overlaps

A very common operation that the OmpSs run-time has to perform is to compute if two given regions overlap, and which is the resulting region of this overlap, if it exists. Because regions are always given unique identifiers we can apply memoization in order to perform the overlap check and store the result to avoid computing the same values in the future.

6.3.2 Region based memory management

The main subsystem of Nanos++ that had to be adapted was the memory management part. Moving from a management based on contiguous memory, to a one based on regions with potentially non-contiguous data caused important changes in the way Nanos++ allocates and transfers data among the different address spaces of the system.

The allocation of regions brings a few questions. When it comes to allocating strided regions of data, the actual allocated size will always be bigger than the used size by the application, because the shape of the data must remain the same. This can be a problem when we are working with accelerators with limited memory, but there is nothing that can be done from the run-time library to solve this, unless other techniques like data *reshaping* are implemented. For the case of this work targeting the cluster environment, this is not a problem because each node has plenty of memory available, and the decision here has been to allocate whole objects in remote nodes. This reduces the costs of allocating data on demand, which can potentially cause re-allocations with severe performance penalties. An additional policy is implemented to target devices with more limited memory amounts, with it, Nanos++ will only allocate the minimum chunk required to fit the referenced data.

Data movement operations of strided data were originally implemented on a very naive way. For each region that had to be moved, a list of contiguous data chunks was generated, and then a movement operation was issued for each chunk. While this approach fulfills the requirements of correctness, its performance may vary depending on the architecture system that is the responsible for moving the data. The following section describes how this was optimized for the cluster case.

6.4 Optimizing for clusters

Supporting regions of non contiguous data added extra challenges to the implementation of the Nanos++ devices. Originally devices had a method to transfer contiguous data from main memory to their private address spaces, and while this could be used to implement non contiguous transfers by reducing them to its contiguous components, this could also be very costly, specially when interacting with the device resulted in performing system calls to the operating system. To handle this we extended the Nanos++ device interface and implemented a mechanism to improve the performance for the cluster device.

In addition, the affinity scheduler implemented in Chapter 4 was extended with new options to increase the performance of applications. The goal of these options was to achieve a better usage of the network resources.

6.4.1 Data packing for non-contiguous data transfers

The main problem of the first implementation was the number of data movement operations that were generated for strided regions. In the cluster architecture these operations were translated to network messages. While sending a network message is an expensive operation by itself, some regions caused hundreds or thousands of these calls, which resulted in a poor application performance.

To overcome this problem, we implemented a simple mechanism to minimize the number of network messages that were needed to transfer regions of strided data. The idea is to use a temporal buffer where the strided data is placed contiguously prior to be sent through the network. Once the data arrives to its destination, the run-time library can reverse the operation and place it with its original shape. Figure 6.8 shows a sample transfer using the described mechanism.

An important implementation detail that had to be taken into account when we were implementing this mechanism is to control which thread is responsible for packing and unpacking the data. In Nanos++ there is a thread in each node which is responsible of monitoring the network for incoming messages, this is a requirement of the underlying communications library used (GASNet [15]). Delegating the task of unpacking the data to the thread which is actually selected to execute a parallel task was needed to keep a good network latency, if not, the communication thread could spend too much time dealing with this packing mechanism, stalling the execution of the whole application. In the case of data packing, the problem is more complex because it usually is the communication thread the one starting the transfers

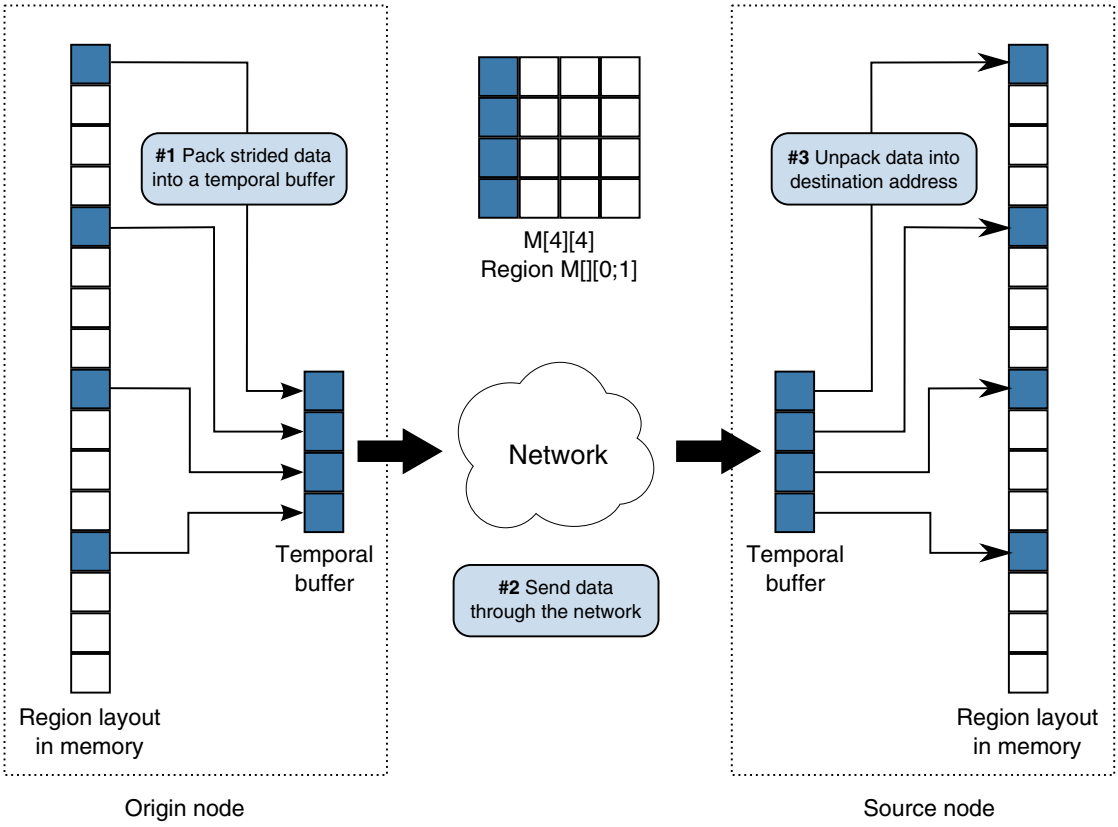


Figure 6.8: Region transfer with data packing

of data, so these tasks can not be delegated to another thread as easily as the unpacking.

As will be seen in Section 6.5, this technique greatly improved the performance of applications in the cluster environment, specially those that require an efficient usage of the network bandwidth.

While this technique is available for cluster environments, its implementation is generic enough to be applied on other architectures like GPUs. However, the effectiveness on other systems is yet to be evaluated.

6.4.2 Affinity scheduler improvements

During the development of the region support the scheduling strategy presented in Section 4.2.6 was improved to provide a better performance when running applications sensitive to network performance.

Two mechanisms were implemented to achieve this goal:

1. The capability of SMP worker to process network messages when they are waiting for incoming data.

2. Prioritize the execution of tasks depending on the data transfers they will issue.

The first mechanism boosts the performance of the network usage by the master node by enabling extra threads to issue messages. When SMP worker threads request remote data, they have to wait until the data is ready to resume the execution. This wait is used to issue small control messages that are critical to keep the activity of remote nodes. Small messages include requests to start the execution of tasks and requests to send data from a remote node to a second remote node. This increases the throughput of the master node and also increases the overall activity and parallelism of the execution, since more tasks can be started in less time.

The second mechanism tries to achieve an execution order of tasks that uses the network resources more efficiently. The original affinity strategy respected the creation order when executing the tasks that had to be executed on a given node–that is two tasks could be executed out of order only because they were executed on different nodes. In a cluster system, the creation order of tasks can have some properties that are not desirable when it comes to execution. Two or more tasks created consecutively are likely to access the same data. In a cluster system this may result in data transfers from a single node to the rest. If this tasks are also executed close in time, this may create a bottleneck in the node that holds the shared data. The implemented strategy tries to schedule tasks that require data from different nodes with the hopes of distributing the network traffic evenly among all nodes.

6.5 Performance evaluation

This section covers the evaluation experiments done to measure the performance of the OmpSs applications using the presented mechanism on a cluster of multi-core processors.

6.5.1 Methodology and environment

To evaluate our environment we selected a set of OmpSs applications and measured their scalability with our run-time environment. These applications exploit the support for non-contiguous regions of data. We also compare them against well known MPI implementations of the same applications. We run the experiments in two hardware systems:

Minotauro It is cluster with nodes consisting of two Intel® Xeon® E5649 @ 2.53GHz multi-core processors with six cores each. Each node also has 24 GB of memory divided in two NUMA banks. Each node has two Infiniband QDR network interface controllers connected to a non-blocking network. All OmpSs benchmarks were compiled using the Mercurium C/C++ source-to-source compiler, using GCC 4.4.4 as a the final C compiler, and using -O3 optimization level. Benchmark kernels are compiled with the Intel® C Compiler (ICC) version 12.0.4, also with -O3 optimization level enabled, or invoked from the Intel® Math Kernel Library (MKL) version 10.3.4. The MPI distribution is provided by the hardware vendor, but it is derived from the Open MPI version 1.1.11.1.

MareNostrum3 Cluster of nodes containing two Intel® Sandy Bridge EP E5-2670/1600 processors. Each processor has eight cores, 20 Mb of cache memory and runs at 2.6 GHz. Each node has 32 Gb of memory available distributed in two NUMA banks and the operating system used is SUSE Linux Enterprise Server 11. The interconnection network of the cluster is Infiniband FDR10. In this system, the kernels were compiled using the Intel® C Compiler (ICC) version 13.0.1 or invoked using the Intel® Math Kernel Library with the same version. Other parts of the benchmarks are compiled using the Mercurium C/C++ source-to-source compiler along with GCC 4.7.2 as the native compiler. The optimization level -O3 was always used with all compilers. The MPI distribution used was OpenMPI version 1.8.1.

6.5.2 Benchmarks

The applications selected have been a double precision matrix multiplication (DGEMM), the PTRANS benchmark and a one dimensional Fast Fourier Transform (FFT1D). All of them are part of the HPC Challenge (HPCC) suite. A detailed description of each benchmark follows:

DGEMM The benchmark performs a matrix multiplication of two input matrices and places the results on a third matrix. Two different OmpSs implementations have been tested. The first one is the original OmpSs approach, it uses a single level of parallelism to perform the computation. The schema used corresponds to the code shown in figure 6.1, and it does not require the regions of data support designed in this chapter. The second one uses multi-level parallelism to improve locality and improve the distribution and creation of the computation tasks. The multi-level version exploits the region support since the top level tasks require the specification of non-contiguous regions. The schema followed to implement this version can be seen in figure 6.2. We

have compared both versions of OmpSs with the purpose of showing the performance benefits that can be achieved by using non-contiguous data regions. The parallelization is implemented by dividing the main algorithm in tasks that perform the matrix multiplication of blocks of 512 × 512 elements.

The MPI implementation used has been the ScaLAPACK [13] one provided by the Intel Math Kernel Library (MKL). The OmpSs versions have also used the Intel MKL. All implementations use a matrix size of 16384 × 16384 for each matrix.

PTRANS The benchmark measures the rate of transfer for larges arrays of data by implementing a parallel matrix transpose. It exercises the communications of the cluster heavily on a realistic problem where pairs of processors communicate with each other simultaneously. The arrays data used has also been two-dimensional arrays of 16384 × 16384 elements.

Following the same parallelization schema as the one used in the DGEMM experiments, the parallelization of the PTRANS is implemented using tasks that process sub-blocks 512 × 512 elements. We have tested the OmpSs version of this benchmark with two Nanos++ configurations, depending on whether the optimization technique of packing the strided data prior to send it over the network was enabled or not (described in Section 6.4.1). This shows how effective this optimization can be on applications that stress the usage of the network.

The MPI implementation of this benchmark comes from the HPC Challenge Benchmark suite [66], [38].

FFT1D This application measures the floating point rate of execution of the double precision complex one-dimensional Discrete Fourier Transform (DFT). The OmpSs implementation uses the Cooley-Tukey algorithm to implement the one-dimensional DFT. The data is distributed in a two-dimensional array of 16384 × 16384 complex double precision elements. The first step of the algorithm is to perform a in-place transposition of the data, after this, a FFT1D round is applied to each of the 16384 rows of the data. The next step is to transpose again the data and to apply a *twiddle* factor, to follow with a second round of FFT1D on each row. Finally, a last in-place transpose obtains the final result. The parallelization of the transpose and the twiddle+transpose are also implemented using tasks that operate on sub-blocks of the matrix, the dimension of these blocks is also 512. The row-FFT1D processes are parallelized by creating tasks that process blocks of 512 rows of the main matrix, each of these tasks also create

more parallel tasks to perform the final computation. This structure can be seen in figure 6.3. For this application, we have also evaluated the impact of the data-packing optimization implemented in Nanos++, to demonstrate its impact on a more complex application.

As for the selected MPI implementation, we have used the one provided by the HPC Challenge benchmark suite [66], [38], which is a FFT1D algorithm specifically tailored for MPI. We have also used the FFT1D kernel in our OmpSs implementation.

6.5.3 Experiments

For each one of the selected applications, we have performed tests with different configurations on the number of nodes and threads used on each node. In both systems, we used the specific GASNet Infiniband conduit.

The evaluation in the Minotauro cluster also includes an evaluation of the optimization discussed in Section 6.4.1. Each application is run with it enabled and disabled.

The MareNostrum3 experiments also include an evaluation of the effects of the techniques implemented in the affinity scheduler presented in Section 6.4.2. In this case, only the FFT1D is used to showcase the benefits of it.

We used from 1 to 32 nodes in both environments. Again, the maximum value is a consequence of the limitation in the maximum memory that can be handled by an OmpSs application (described in Section 4.3.1). Using more than 32 nodes would have required bigger data sets in order to achieve meaningful results.

6.5.4 Results: Minotauro

The following lines present the results of the evaluation in Minotauro. For each application, a figure illustrates the performance measurements obtained with the OmpSs versions and the MPI implementation.

Each figure is divided in different charts, one for each configuration of threads per node used. The leftmost chart depicts the scenario where executions use one thread per node, after this first chart, the value of threads per node is doubled until reaching the maximum value supported by the system. The range of the Y axes of each chart is doubled to represent the increase of resources used during the experiments. Each chart X axes represent the number of nodes and the Y axes the performance of the benchmark. The goal of scaling the range of the Y axes of each chart is to facilitate the visual representation of the speed-up of different configurations.

The number of worker threads when using the maximum value is actually different in the MPI version than the OmpSs versions. This is because Nanos++ uses a dedicated thread to manage communications and then limits the maximum number of worker threads to $N-1$ on a system with N available cores. The MPI implementations do not suffer from this restriction. As a result, in Minotauro, the experiments where the maximum worker threads per node are used, the OmpSs versions use 11 worker threads and the MPI implementation uses 12 worker threads.

DGEMM

Figure 6.9 shows the performance obtained by the DGEMM benchmark. We have implemented two versions of this benchmark, the one labeled "OmpSs 1L" refers to the original OmpSs implementation (the schema used by corresponds to the source code shown in figure 6.1) which can be coded without the region support described in this chapter. The second version, labeled "OmpSs ML" is the implementation using multi-level parallelism enabled by the data region specifications (same schema as the one shown in figure 6.2). The ScaLAPACK results are under the "MPI" label. In the MPI executions the threads per value is set through the `OMP_NUM_THREADS` environment variable, as the ScaLAPACK implementation uses OpenMP internally.

The charts show how the version implemented using multi-level parallelism is capable of achieving the same scalability as the ScaLAPACK version, even being able to obtain a slightly better performance due to other available optimizations of Nanos++ [17], [18] (task presend, affinity scheduler) that increase the overlapping of communication and computation. The simple OmpSs version performs well with a low number of nodes, but it fails to scale properly with 16 or more nodes, this is caused by the fact that in this version, task creation is centralized on the master node, which creates a bottleneck when distributing the tasks among the rest of the threads of the cluster. This is why the problem is more noticeable with an increasing number of nodes and threads. The multi-level version is capable of delegating part of the task creation to the remote nodes, which reduces the overhead of task distribution from the master.

PTRANS

The PTRANS results are shown in Figure 6.10. In this case we have tested a single OmpSs implementation of the benchmark but we have run it with two Nanos++ configurations, the first without enabling the technique of packing the strided data prior to send it through the

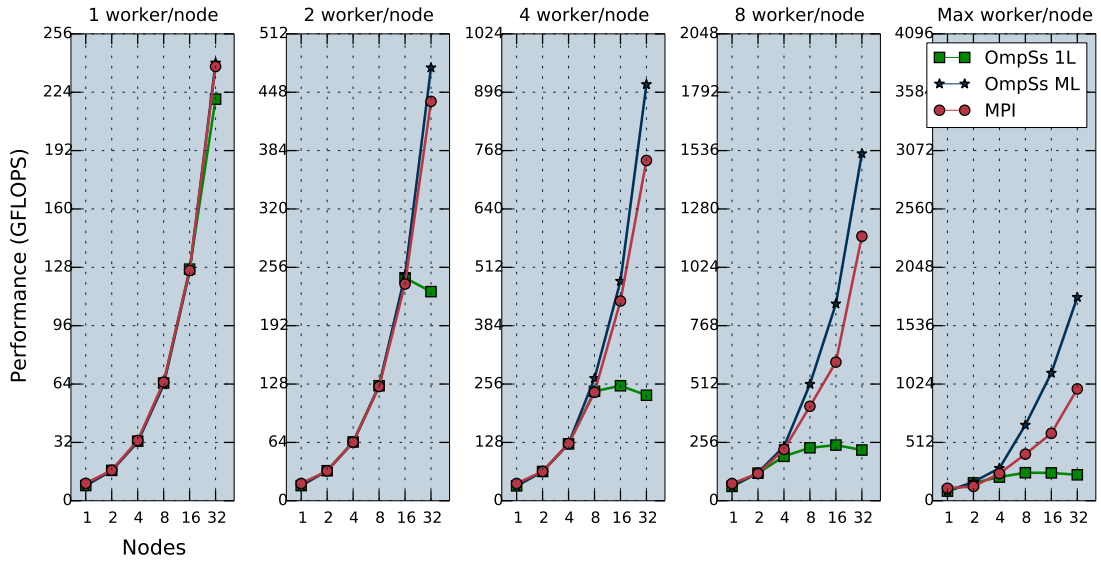


Figure 6.9: DGEMM performance comparison, OmpSs against MPI running on Minotauro.

network, labeled as "OmpSs NP", and the second with this technique enabled, labeled as "OmpSs". The "MPI" label in these charts refers to the HPCC implementation of the benchmark. It must be noted that this benchmark is not implemented using OpenMP, so to enable the usage of more than one core per node, we executed it spawning the appropriate number of MPI tasks instead of threads.

The results show how effective is the packing mechanism in this application. It greatly improves the performance of the OmpSs implementation in every situation and it actually makes OmpSs to be at the same level as the HPCC version. The reasons for this are that the PTRANS performance relies on achieving a good network usage. Without the packing technique this is not possible because strided accesses are transformed in one network message per contiguous data, which leads to a very high number of messages. By packing the data the run-time library reduces this number of messages and communications are more efficient, and the execution time is significantly reduced.

The comparison of MPI and OmpSs is a bit unfavorable to OmpSs, which shows a worse scalability than MPI when increasing the number of nodes. It must be noted that the OmpSs version is more efficient code than the MPI version, due to using a more efficient transposition kernel. This is clearly seen when executing with one thread per node, where the corresponding chart shows a clear improvement of OmpSs over the MPI code.

This benefit is however diluted when we increase the number of threads as the communication become a more important factor in order to achieve better performance. The MPI

implementation, while being a more rigid programming model, is capable of getting more performance because the communication pattern of PTRANS is well balanced. OmpSs, on the other hand, struggles to achieve the same scalability because of the dynamic approach of creating parallelism on demand can create sub-optimal communications among the nodes of the cluster. Also, creation of tasks is centralized in the master, which also impairs the performance scalability. All in all, the OmpSs version achieves a satisfactory scalability.

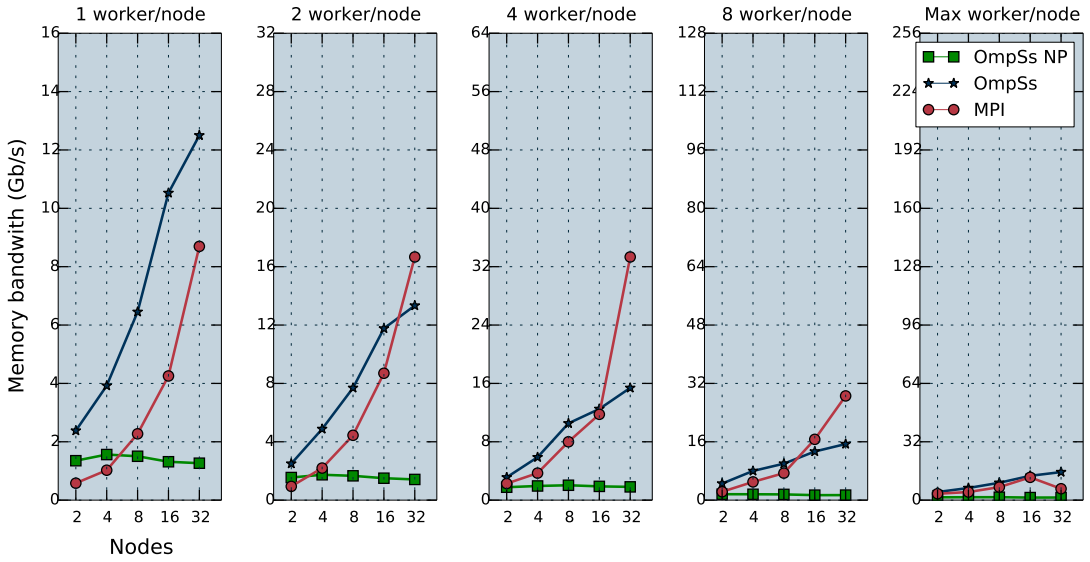


Figure 6.10: PTRANS performance comparison, OmpSs against MPI running on Minotauro.

FFT1D

Figure 6.11 shows the performance results of the FFT1D benchmark. As with the previous benchmark, we have executed the OmpSs version with the packing technique disabled and enabled to measure its impact on the FFT1D benchmark. The packing enabled data series is labeled with the "OmpSs" title, and the packing disabled is referred by the "OmpSs NP". The MPI version of the FFT1D benchmark provided by the HPC Challenge suite is implemented using MPI and OpenMP so in this case we can control the number of threads created on each node using the same way as in the DGEMM benchmark.

FFT1D is a demanding benchmark since combines some phases of high bandwidth requirements with phases of intense computation. Because of this, we see the same behavior of PTRANS when it comes to the effect of the packing optimization. Enabling the technique boosts performance in almost all cases and it is absolutely necessary to achieve a proper

scalability with a large number of nodes and threads per node.

In this benchmark, the MPI performance is comparable to the OmpSs performance. First it must be noted that the OmpSs code also benefits from a better transposition kernel than the MPI implementation. However, this benchmark also suffers from the problems exposed by the PTRANS benchmark, since during the execution of the FFT1D algorithm there are three transpositions of a two dimensional array. Still OmpSs is capable of hiding the overhead of managing these stages by overlapping them with the computation stages. MPI can not do this because has to synchronize explicitly between them. So in this scenario the dynamic approach of OmpSs is capable of overcoming some of its inherent problems.

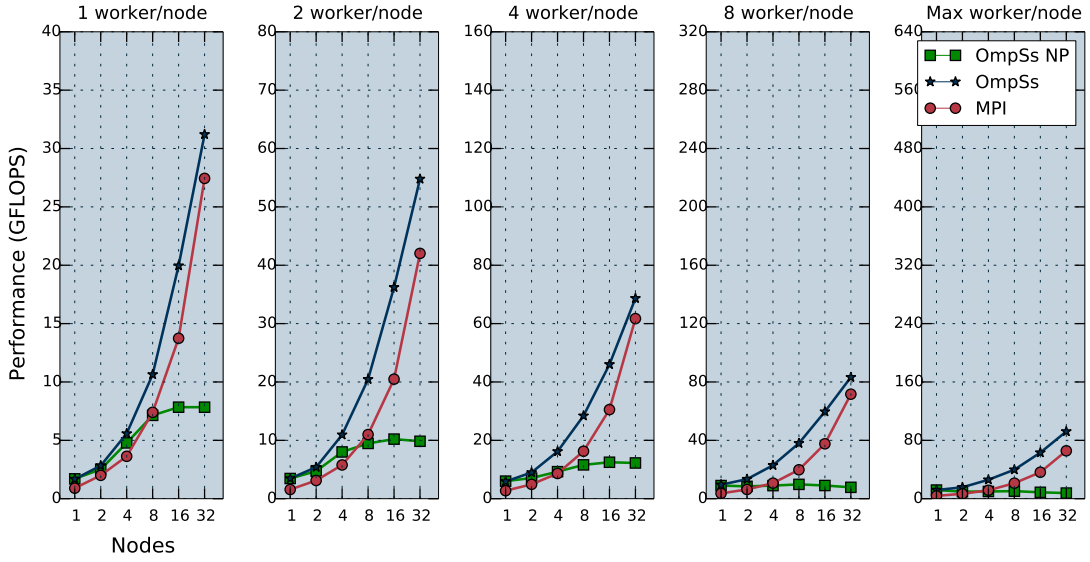


Figure 6.11: FFT1D performance comparison, OmpSs against MPI running on Minotauro.

6.5.5 Results: MareNostrum3

As done in Section 6.5.4 the results of the MareNostrum3 evaluation are presented individually for each application.

In this scenario, the maximum number of workers per node goes up to 16, which means that the OmpSs experiments actually use 15 whereas the MPI implementations use 16.

These experiments compare the performance of the OmpSs versions against the MPI implementations. In the case of the FFT1D benchmark, we also present an extra set of experiments to evaluate the benefits of the scheduling improvements described in Section 6.4.2.

DGEMM

The results of the DGEMM evaluation in MareNostrum3, located in Figure 6.12, show a similar behavior than the results obtained in Minotauro, except that OmpSs struggles to scale from 16 to 32 nodes when using 8 workers per node or more. This fact can be attributed to the limits of the problem size and the increased CPU performance of the system, which reduces the opportunity of overlapping communication with computation. It must be noted than even without scaling perfectly, OmpSs achieves a similar performance than the MPI implementations on this configurations.

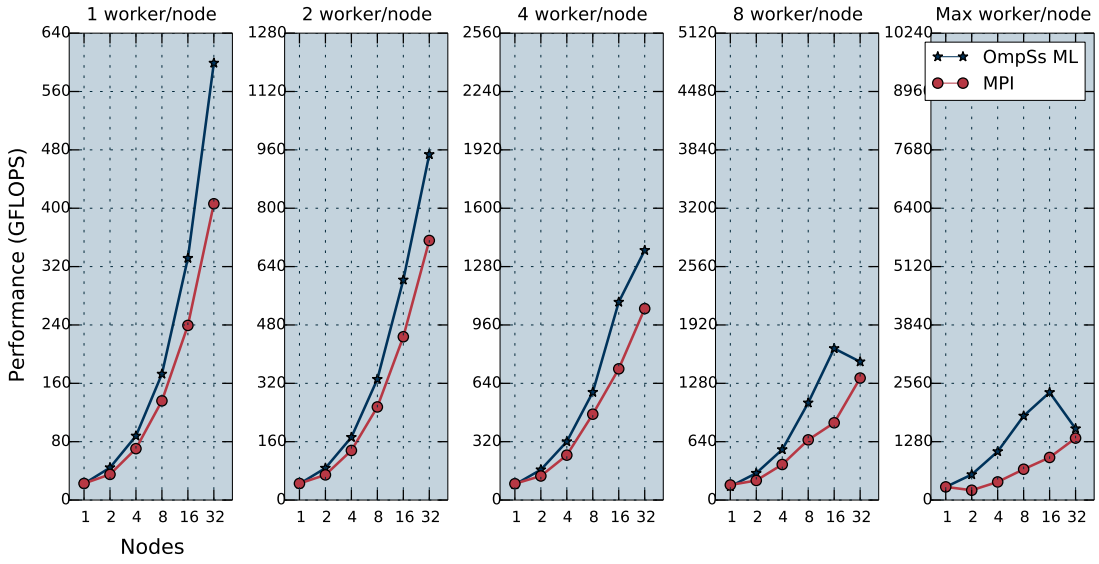


Figure 6.12: DGEMM performance comparison, OmpSs against MPI running on MareNostrum3.

PTRANS

The PTRANS benchmark performance is shown in Figure 6.13. The application suffers the same problems observed in Minotauro but amplified. The OmpSs version is capable of achieving some scalability. But the performance with 32 nodes is generally poor due to the cost of distributing the parallel tasks. In addition, the MPI performance is better in this system, which creates a bigger gap in the comparison against OmpSs.

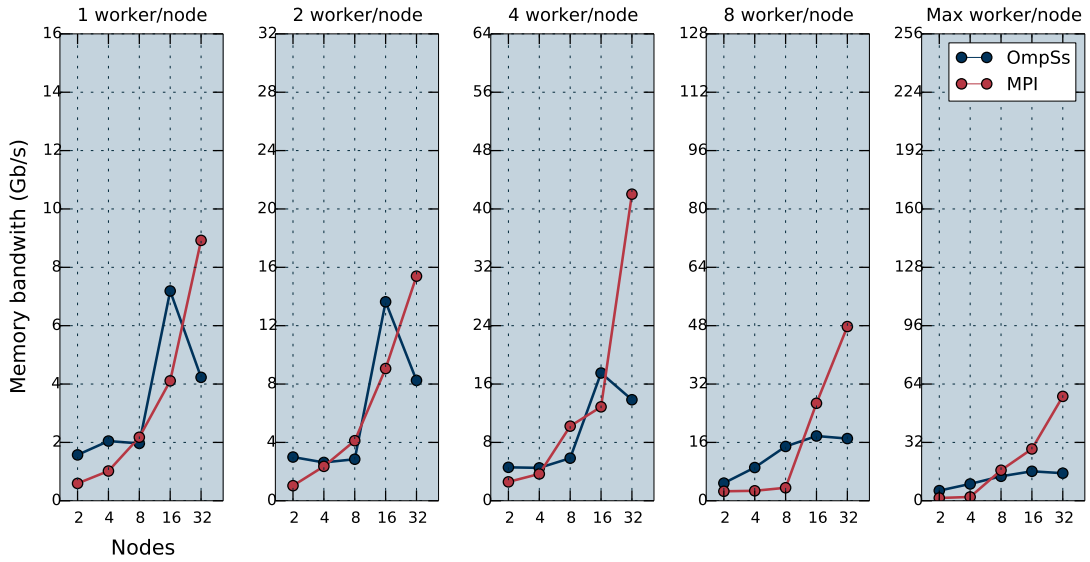


Figure 6.13: PTRANS performance comparison, OmpSs against MPI running on MareNostrum3.

FFT1D

Figure 6.14 shows the performance results of the FFT1D benchmark in MareNostrum3. Overall the results are comparable to the ones obtained in Minotauro. OmpSs version, with the benefit of the improved transposition kernel, is capable of achieving a better performance than the MPI implementation. The performance gap closes when running with 32 nodes, which is somewhat expected due to the cost of distributing the data being higher than in other configurations.

In addition to the OmpSs vs MPI comparison, Figure 6.15 offers an evaluation of the options affinity scheduler enhancements implemented and previously described in Section 6.4.2. Three different data series are shown. The data labeled with "no affinity" corresponds to the execution of the application with the default Nanos++ scheduling algorithm. The data labeled with "affinity" corresponds to the performance results while using the affinity scheduling with no extra options. Finally, the data labeled "affinity w/network opt." shows the results with the optimizations enabled.

The results clearly show how important is caring about the network usage to achieve a good performance. This is particularly important in applications like the FFT1D, which mix stages of high network performance requirements with stages of computation intensive code. If the former stages are not well handled the execution may suffer from unbalance

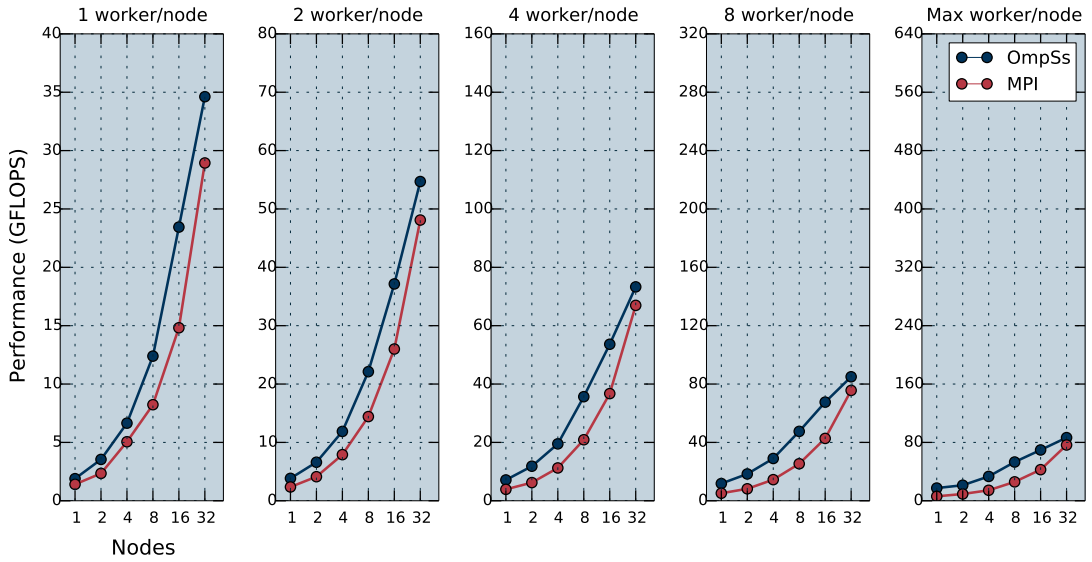


Figure 6.14: FFT1D performance comparison, OmpSs against MPI running on MareNostrum3.

and could end up affecting the latter. An important conclusion that can be extracted is that is important to use extra threads to fully exploit the bandwidth offered by the network. The affinity scheduling without the optimizations performs significantly worse when increasing the number of threads per node. With the optimizations, Nanos++ can increase the throughput of the master node and meet the increased requirements of the configuration–it has more threads to help with network activity but it has to feed more remote threads with tasks and data. In conclusion, the network optimizations have been critical to achieve a good performance with FFT1D.

6.6 Conclusions

This chapter has described the design and implementation of the support for managing regions of data in the OmpSs programming model on systems with disjoint address spaces. This allows the programmers to specify complex regions of data on their programs with a simple and flexible syntax, and the underlying run-time system will handle its movement among the memory of the different devices or nodes. We believe this feature is of high importance nowadays due to the increasing complexity of the HPC hardware, where machines with accelerators and/or distributed memory are ubiquitous. This makes desirable the automatic management of data that OmpSs provides.

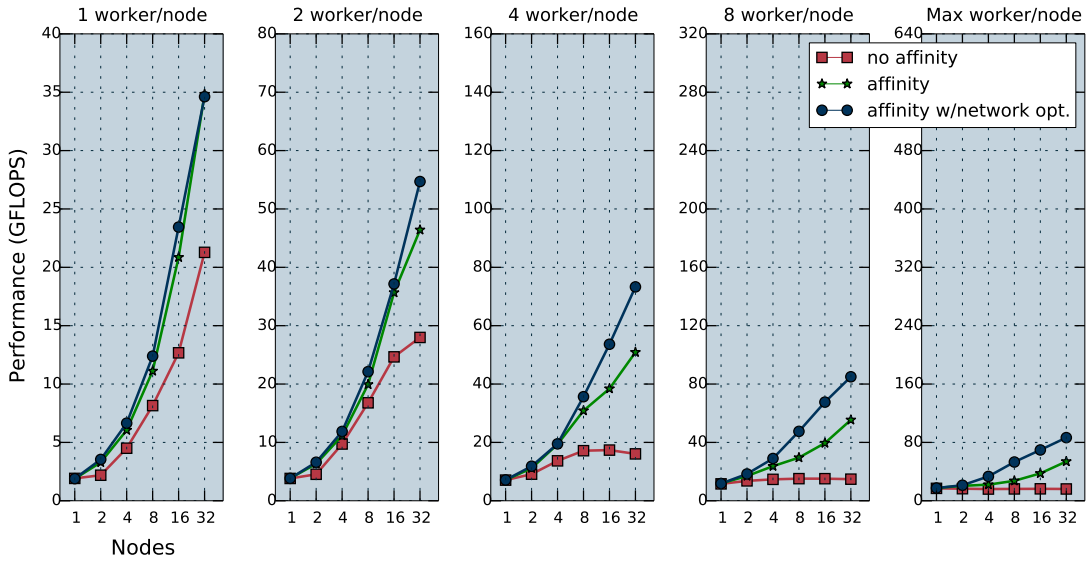


Figure 6.15: OmpSs FFT1D performance with different execution options running on MareNostrum3.

Following the OmpSs philosophy, the programmer responsibility is just to specify which data will be used by each task, and the run-time system will provide the needed mechanisms to ensure the correct execution, regardless of the available hardware configuration. In addition the run-time environment may found optimization opportunities to be able to achieve the better possible performance.

The solution that we have designed and implemented has allowed us to evaluate the effectiveness of having such mechanism implemented. Not only we have been able to implement more complex benchmarks but also we have made more efficient versions of others.

The evaluation has shown good performance results on a set of different benchmarks. We also have compared the performance of the OmpSs versions with the same applications but programmed using well-known implementations of MPI, and MPI+OpenMP. The results show that OmpSs can achieve a better performance in some cases and overall scalability is quite on par. It must be noted that this good performance would have not been possible without the optimizations implemented.

The results of this contribution have been published in [16].

7

Related work

Research on programming models for distributed systems has always been very active due to the increase usage of such systems. Nowadays there is a wide spectrum of solutions that tries to fulfill the necessities of all users. This chapter is divided in two big sections. The first one reviews the most significant efforts to develop programming models for distributed systems. The second section covers a more recent family of programming models, the ones that target accelerators and heterogeneous architectures.

7.1 Programming models for distributed systems

During beginning of the nineties, the first version of the Message Passing Interface (MPI) [75] was officially released. Since then, MPI has become widely used because of its portability and the fact that enhancements to the network or system technology would not made it obsolete. MPI is a collection of library routines that can be used to write message passing programs; when using MPI, the programmer has to explicitly introduce the desired calls, and thus introduces into the application the decisions of how the application will communicate data between the different computation nodes; usually a Single Program Multiple Data pattern is followed when developing MPI applications. This explicitness of MPI has both benefits and drawbacks. The big benefit of it is that well designed distributed applications usually achieve the best performance compared to other tools, while the big disadvantage is that reaching this well design is not an easy task.

Since the emergence of distributed systems there have been a lot of projects with the mission of easing the task of writing programs for them. There has also been a high diversity on the principles to address this issue, from compiler-only based approaches, to run-time libraries that offer a virtual shared memory environment to completely new programming models.

7.1.1 Translation to MPI

Many projects have had OpenMP as a reference programming model due to its simple concept of annotating a sequential code in order to allow the compiler to generate a parallel version. An example of this is Basumallik et al.[10], who presented another approach that aimed to translate OpenMP to MPI, focusing on parallel loops. It has achieved a good performance when running several OpenMP benchmarks, but it also has to rely on a run-time system to handle irregular array accesses.

7.1.2 Software distributed shared memory systems

Software distributed shared memory (SDSM) systems are another approach to program distributed systems by providing a virtual shared memory environment. The weak point of the SDSM systems is the mechanism to keep the memory coherent among the distributed memory. This tends to cause significant performance penalties and a common decision is to implement a relaxed memory consistency model in order to achieve a better performance.

JIAJIA [100] is an example of SDSM system that uses a lock-based protocol for scope consistency, but it does not have explicit OpenMP support. HAP [62] is another SDSM that relaxes the memory model by using a Lazy Release Consistency. Brazos [41] also uses a scope consistency memory model and supports multi-threading to take advantage of SMP servers.

SDSMs that have been conceived to run OpenMP applications include the OpenMP translator [102], the SCASH system [67] and ParADE [103]. The OpenMP translator allows applications to run applications on top of TreadMarks [20], its main task is dealing with the limitations that the TreadMarks system has due to his implementation (limited shared area and relaxed consistency). The SCASH system uses a release consistency memory model with a multiple writers protocol to avoid false sharing, it requires the Omni compiler [68] to transform the applications to run on it. ParADE also uses the Omni compiler to run OpenMP applications, it is based on a lazy release consistency memory model with home migration and uses MPI as a communication system.

Also, Intel has included SDSM support for OpenMP in its compiler suite. The run-time, called Cluster OpenMP [54], is based on the TreadMarks system. It adds the new directive *shareable* to specify data that will be shared through the SDSM mechanism, thus it requires some modifications in the source code of the applications to be executed with it. The compiler manages the dynamic memory allocation and the implications of having a lazy release consistency.

However, limiting the memory coherence is not really a must when designing a SDSM system, and everything-shared SDSMs with restrictive memory models do exist. An example of such systems is NanosDSM [31]. NanosDSM is an everything shared SDSM, meaning that the whole space address (global data, stacks, code) is shared. The system also provides sequential consistency as the memory consistency model, implying that memory operations are visible to other threads at the same time they are executed. Both features might seem as a performance killer for many applications, and of course the system has to pay some performance penalty for them, but there are optimization techniques that have been applied successfully on some programs, achieving a performance as good as other SDSMs with the restrictions seen above. Furthermore, both of these features ease the porting of applications to this system, since the vision of the architecture provided is almost identical to a hardware shared memory environment. NanosDSM and the Mercurium C/C++ compiler are capable of running standard OpenMP applications on cluster environments. Despite of application performance being limited by the algorithm nature, run-time optimizations can provide a reasonably good solution for some cases, as shown in [19]

7.1.3 Partitioned global address space

Partitioned Global Address Space (PGAS) programming models are also an effort to virtualize the memory of the system and offer a view of it that resembles a shared memory system. Annotations are provided in order to specify how to distribute the data across the nodes of the system, and special statements may be used to access to non-local data. The compiler is the responsible to generate the appropriate run-time calls to transfer the data. With this, the responsibility of moving data is left to the compiler and the underlying run-time system, which eases the development of the applications [45, 90, 104].

High Performance FORTRAN was an extension to the FORTRAN language that provided a PGAS-like environment that allows the declaration of shared arrays that are distributed dynamically at run-time.

A similar concept follows the Unified Parallel C (UPC) [29], an extension to the C pro-

gramming language that implements similar features. Shared variables can be specified with the usage of the `shared` keyword. This data will be distributed over the memory of the system and the compiler will handle accesses to these variables by adding the appropriate run-time calls. Special statements like `upc_forall` mimic the usual language statements but allow computation over shared data.

Following the same principle, Co-Array FORTRAN [79], a more modern FORTRAN extension than HPF, offers an improved PGAS environment using FORTRAN 95 as its base language. Titanium [105] is another similar effort but applied to the Java programming language.

Following the trend of exploiting asynchronous parallelism in order to allow expressing irregular parallelism. A few PGAS environments have emerged with features that support it, they have been denominated Asynchronous Partitioned Global Address Space languages (APGAS). X10 [22] is an example of APGAS for the java language. Chapel [21] is another example of APGAS in the form of a complete standalone language.

Similar to OmpSs, XcalableMP (XMP)[78], is a directive-based programming model for distributed environments. It also focuses on productivity and offers a global-view model to enable the parallelization of sequential code with a few directives [77].

7.2 Programming models for heterogeneous systems

GPUs had irrupted in the recent years in the scene for HPC. Due to the popularity of NVIDIA GPUs, CUDA have almost become a de-facto standard for programming GPUs. CUDA [81] is an extension to C++ and it is based on *kernels* that are run n times in parallel by n threads. However, the programmer is not only responsible of writing the application code and computational kernels, but also of performing memory transfers from host memory to device memory. Tools to better map the algorithms to the memory hierarchy have been proposed [98]. They advocate that programmers should provide straight-forward implementations of the application kernels using only global memory and that tools like CUDA-lite will do the transformations automatically to exploit local memories.

An alternative to program accelerators that can also be used to program general purpose multi-cores is the new standard OpenCL [58]. Although the portability is a strong aspect of OpenCL, it offers a too low-level API to the programmer, exposing her to explicitly manage the data and threads. For example, it is responsibility of the programmer to build the program executable or to move data between the cores and accelerators.

With regard the deployment of applications in clusters of GPUs, most of approaches

follow a combination of MPI between nodes and CUDA inside the node [50, 84, 56]. There are very few approaches that consider alternative programming models for clusters of GPUs. One of them is rCUDA [39, 40], which implements an almost compatible CUDA API that allows CUDA programs to run on a distributed environment with little modifications. This project, however, focuses on virtualizing the view of the GPUs with the purpose of optimizing their usage by allowing devices to be accessed by different processes. From the same research group, CU2rCU [91, 92] is a tool that eases the adaptation of CUDA programs to work with rCUDA.

The cudaMPI and glMPI[63] provide a MPI-like message passing interface that enables to communicate data stored on the GPU cards of a cluster. While cudaMPI extends MPI to work with clusters of GPUs using CUDA, glMPI does the same for OpenGL. cudaMPI's programming model is CPU centric, in a way that CPU codes initiate each communication operation and the GPUs only provide the communicated data. This allows communication operations to be performed massively on large blocks of data.

An implementation of HPL benchmark for a heterogeneous cluster of CPUs and GPUs is presented by Fatica[44]. This implementation is based on a host library that intercepts the calls to DGEMM and DTRSM and is able to execute them simultaneously on both GPUs and CPU cores. The application parallelizes across the cluster nodes with MPI and inside the node, the matrix is divided in two: one part is processed by the CPU cores and the other by the GPU. To make the adequate split, they statically take into account the process time and the transfer time required to send the data to the GPU.

XKaapi [46] offers an API for C, C++ and FORTRAN that allows the creation of parallel tasks that follow a similar semantics than the tasks offered by OmpSs. Tasks are executed following the data-flow of the application, in the same way OmpSs data dependencies operate. In addition, they also offer multi-implementation of tasks on CPUs or GPUs.

Grasso et al. [48] have developed libWater, a library that extends the OpenCL programming model and raising the abstraction level of it. It is intended to use along with tools to generate code and can support multi-GPU and distributed environments. libWater organizes the application commands using a directed acyclic graph, similarly to the dependence graph of OmpSs, and uses it for dynamic analysis and optimizations.

7.2.1 Translation to CUDA/OpenCL

Lee et al. [64] propose OpenMPC. The proposal is based on an OpenMP-to-CUDA translation system, which performs a source-to-source conversion of a standard OpenMP program to a

CUDA program and applies various optimizations to achieve high performance. This system has been built on top of the Cetus compiler infrastructure. The compiler interprets OpenMP semantics under the CUDA programming model and identifies kernel regions (code sections to be executed on a GPU) and transforms eligible kernel regions into CUDA kernel functions and inserts necessary memory transfer code to move data between CPU and GPU. OpenMPC does not support OpenMP tasks, the construct that allows irregular parallelism, and also it does not consider producing code for clusters of GPUs.

Another project, CU2CL [69], tries to achieve a similar goal but translating from CUDA to OpenCL. The main motivation to do this is that OpenCL is an open standard that can target multiple devices whereas CUDA only works with NVIDIA® GPUs. With it, CUDA codes would be able to run on OpenCL-based systems.

7.2.2 Annotation-based programming models

New computer architecture designs based on heterogeneous multi-cores have raised the question about their programmability. Programming models based on annotations on a sequential code had also appeared to ease the burden of heterogeneous parallel programming. Some of them also implement asynchronous parallelism in order to allow the implementation of complex applications.

OpenACC[82, 101] is an standard similar to OpenMP that targets heterogeneous CPU/GPU systems. It has been created by Cray, CAPS, NVIDIA and PGI. OpenACC defines directives to define computation kernels, to move data to and from the accelerators, and to synchronize the execution. It has been used to target CUDA and OpenCL devices [93].

HiCUDA [51] proposes a set of directives giving hints to the compiler about regions of code that can be exploited in the GPUs, and data directionality. Mint [99] implements a translator to transform stencil computations expressed in C, into CUDA code, to run on a GPU. According to the publication, Mint extensions would be needed to handle a multi-GPU environment.

The CAPS HMPP [37] toolkit is a set of compiler directives, tools and software run-time that supports parallel programming in C and FORTRAN. HMPP works based on *codelets* that define functions that will be run in a hardware accelerator. These codelets can either be hand–written for a specific architecture or be generated by some code generator.

Offload [30] is a programming model for offloading portions of C++ applications to run on accelerators. Code to be offloaded is wrapped in an *offload* block, indicating that the code should be compiled for an accelerator, and executed asynchronously as a separate

thread. Call graphs rooted at an offload block are automatically identified and compiled for the accelerator. Data movement between host and accelerator memories is also handled automatically.

The PGI Accelerator Compilers [89] and the Cray OpenMP Accelerator compilers [1] provide support for NVIDIA GPUs. Both compiler systems recognize regions of code annotated with a special pragma, and they outline the code to be run on GPUs. Data directionality clauses are also incorporated in both approaches. The latter proposals, HiCUDA, CAPS HMPP, Offload, and the PGI compilers do not support clusters. Instead, they target SMP nodes.

The Sequoia [61] [43] alternative focuses on the mapping of the application kernels onto the appropriate engines to exploit the memory hierarchy. Sequoia also supports some forms of irregular parallelism [11], using a form of tasks similar but more restricted than those defined by OpenMP 3.0. Sequoia also allows complex data specifications, but it is its compiler the responsible to generate the appropriate code to handle the data transfers, not the run-time environment. With this, the user has to specify the underlying hardware architecture at compile time. Sequoia targets a variety of systems including CUDA GPUs, clusters and SMP CPUs.

With the objective of tackling GPUs, an extension to UPC[24, 106] with hierarchical data distribution is presented. The approach extends the semantics of upc_forall to support multi-level work distribution. This work also presents features based on compiler analysis such as affinity-aware loop tiling and the implementation in the run-time of a unified data management on each UPC thread to optimize data transfers between CPU and GPU.

StarPU [3, 4] is a similar project to OmpSs. While not being based on annotations, it offers compiler extensions to target heterogeneous systems with accelerators [32]. StarPU focuses also in offering a productive environment [57] based in the specification of parallel tasks with the specification of their data usage. It also takes advantage of this information to guide the execution of the application [2]. StarPU can also be integrated with MPI [5] to target clusters with GPUs.

XcalableMP has also included support for accelerators in clusters through an extension named XcalableACC [76]. It extends the features that allow automatic data mapping and distribution of XMP to work with accelerators.

8

Conclusions and Future Work

This chapter summarizes the contents of this thesis and highlights the main contributions derived from it. Finally it presents different ideas about how this research can be continued.

Chapter 1 shows how the field of High Performance Computing has evolved towards more complex hardware architectures. Distributed memory systems became the most popular type of supercomputer more than a decade ago (see Figure 1.1). Nowadays the addition of accelerators, like GPUs or FPGAs, as a co-processors to boost the systems' performance represents another increase in the complexity of the systems' architecture. This increased complexity has impacted the way applications are programmed. Software developers face new challenges when porting or developing applications in order to take advantage of the new components and memory hierarchies.

The belief that programming models have the mission to ease the software development process has motivation the work presented in this thesis. We selected OmpSs, a programming model developed by the Programming Models group at the Barcelona Supercomputing Center, as the start point for our research. OmpSs is an annotation-based programming model based in OpenMP and StarSs, and tries to target modern HPC architectures. When this work began, OmpSs supported SMP architectures with GPUs. Chapter 2 offers an overview of the OmpSs programming model. Also, Chapter 3 presents the OmpSs run-time environment and describes the original design of Nanos++, the run-time library that implements the OmpSs programming model.

Chapter 4 presents the first contribution of this thesis, which is the implementation and

evaluation of the cluster architecture support for OmpSs. This feature allows OmpSs applications to be executed on a cluster system with minimal changes. It also means that a shared memory programming model can be used to target distributed memory architecture. Two optimizations techniques are also described as a part of the contribution. Both the *affinity* scheduler (Section 4.2.6) and the slave-to-slave communication mechanism (Section 4.2.6) try to minimize the network traffic during the execution of applications in order to reduce the total execution time. The performance scalability of OmpSs on a cluster is evaluated using a set of benchmarks on two different systems. In addition, the OmpSs performance is compared against an implementation of the same benchmarks using MPI, the most used programming model for distributed systems. The results show that OmpSs can deliver a performance comparable to MPI for systems with up to 32 nodes, with some applications doing even better thanks to the features of OmpSs.

The second contribution is described in Chapter 5. The cluster support is extended to handle clusters of SMPs with multi-GPUs. This enables OmpSs GPU applications to run on a cluster transparently. Additional optimizations are also implemented to keep up with the performance requirements of the GPUs. A task pre-send strategy (Section 5.3.3) enables the overlap of communication with computation on the cluster, and a speculative data forwarding mechanism (Section 5.3.3) advances data transfers to the GPUs in order to shorten the critical path of the applications. We used a set of benchmarks to evaluate the cluster-gpu support and the optimizations developed on a cluster with GPUs. We also compared the performance against the same applications implemented using MPI and CUDA, the reference programming model for GPUs. The results obtained by OmpSs are on par with the MPI and CUDA versions and generally OmpSs achieves a good performance with a system of up to 8 nodes with one GPU per node. We also studied the improvement that OmpSs offers in terms of programmability, generally the OmpSs implementations require fewer lines of code to produce the same application.

In Chapter 6 we discuss the last contribution of this thesis, the support for regions of non-contiguous data. We demonstrate that this feature is important in order to allow the implementation complex applications in OmpSs. We show examples of an application where the data structures had to be specifically designed to match the lack of support for non-contiguous memory regions, and even then, a more efficient version with nested parallelism could not be implemented. The internal design of this feature is also described in this chapter. A specific set of data structures was needed to provide a compact representation of the memory regions, and an efficient mechanism to compute region overlaps. The addition of the region support raised some performance issues when running applications on a cluster.

Transferring non-contiguous data can be expensive if the network interface only provides calls to transfer contiguous data. A specific optimization to pack and unpack non-contiguous data allowed the run-time to overcome this limitation and provide a decent performance. Also, the affinity scheduling policy was enhanced to improve the behavior of applications sensitive to network performance. It made worker threads help with the process of small messages whenever they were waiting for incoming data, and the scheduling algorithm prioritized the execution of tasks to balance the network activity among all the nodes of the cluster. The performance of OmpSs with the changes introduced by this contribution is evaluated during this chapter. We developed a set of representative applications that benefited from or required the non-contiguous region support and run them on two different systems. A comparison against the MPI versions is also included along with evaluations of the different optimizations implemented. The results show that the optimizations are critical in order to achieve a performance on par with the MPI versions. Overall the OmpSs performance is good, and the cost of supporting the non-contiguous data regions seems negligible.

Finally, Chapter 7 reviews the most relevant projects with similar goals as this thesis. Since the advent of distributed systems MPI has been a reference programming model for clusters. However many researchers consider it to be too cumbersome for developing programs. PGAS languages and SDSM systems are an example of ideas that have tried to ease this task. More recently, with the apparition of heterogeneous systems, a new layer of complexity has been added to software development for HPC systems. A new wave of programming models has appeared to make those devices as transparent as possible.

Regarding the future research directions, there is much work that can be done to keep pursuing the goals proposed by this thesis. One of the first questions that still remains is how well OmpSs can do in clusters with more than 32 nodes. However, tackling this question is not a matter of just testing how the current implementation performs. In order to use more resources we need bigger applications that require them, and OmpSs has a limitation regarding the memory an application can use. When running on a cluster, the single address space view (Section 2.5.1) provided by OmpSs replicates the address space of the master node in the remote nodes which effectively reduces the total memory usable to the memory available on a single node. This is a clear disadvantage when compared against MPI, which SPMD execution model lets the programmer aggregate the memory of all nodes of the cluster automatically. Solving this issue may require some deep changes to the OmpSs execution model but we believe it is something important that needs to be addressed.

In addition, it would be interesting to investigate different design choices for the distributed version of Nanos++. The centralized model where a master orchestrates the ex-

ecution of OmpSs applications may not scale beyond a certain amount of resources. The performance benefits of using nested parallelism are a proof of this. The parallelism creation is more efficient in the nested case because the amount of tasks processed and distributed by the master is much lower than in the non-nested scenario. A hierarchical architecture of Nanos++ could outperform the current centralized design when running applications on a higher number of nodes.

Last but not least, we think the Nanos++ run-time should be more aware of the underlying network architecture. Modeling the network usage is important in order to achieve a good performance on a cluster, as has been seen by the positive effect of the network optimizations described in Section 6.4.2). Using the network model to guide the execution of applications can greatly improve the performance achieved. While the current optimizations follow this direction, we believe that they are quite simple and can be developed further.

Bibliography

[1] Alistair Hart and Harvey Richardson and Alan Gray and Karthee Sivalingham. *Directive-based programming for GPUs, accelerators and HPC*. Dec. 2010. URL: www.many-core.group.cam.ac.uk/ukgpucc2/talks/Hart.pdf.

[2] Cédric Augonnet et al. "Data-Aware Task Scheduling on Multi-Accelerator based Platforms". In: *The 16th International Conference on Parallel and Distributed Systems (ICPADS)*. Shanghai, China, Dec. 2010. DOI: 10.1109/ICPADS.2010.129. URL: http://hal.inria.fr/inria-00523937.

[3] Cédric Augonnet et al. "Exploiting the Cell/BE architecture with the StarPU unified runtime system". In: *SAMOS Workshop - International Workshop on Systems, Architectures, Modeling, and Simulation*. Vol. 5657. Lecture Notes in Computer Science. Samos, Greece, July 2009. DOI: 10.1007/978-3-642-03138-0_36. URL: http://hal.inria.fr/inria-00378705.

[4] Cédric Augonnet et al. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631. URL: http://hal.inria.fr/inria-00550877.

[5] Cédric Augonnet et al. "StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators". In: *EuroMPI 2012*. Ed. by Siegfried Benkner Jesper Larsson Träff and Jack Dongarra. Vol. 7490. LNCS. Poster Session. Springer, Sept. 2012. URL: http://hal.inria.fr/hal-00725477.

[6] Eduard Ayguade et al. "Exploiting multiple levels of parallelism in openmp: A case study". In: *Parallel Processing, 1999. Proceedings. 1999 International Conference on*.

IEEE. 1999, pp. 172–180.

[7] Eduard Ayguadé et al. “An experimental evaluation of the new OpenMP tasking model”. In: *Languages and Compilers for Parallel Computing*. Springer Berlin Heidelberg, 2008, pp. 63–77.

[8] Eduard Ayguadé et al. “An extension of the StarSs programming model for platforms with multiple GPUs”. In: *Euro-Par 2009 Parallel Processing*. Springer, 2009, pp. 851–862.

[9] Eduard Ayguadé et al. “The design of OpenMP tasks”. In: *Parallel and Distributed Systems, IEEE Transactions on* 20.3 (2009), pp. 404–418.

[10] A. Basumallik and R. Eigenmann. “Towards automatic translation of OpenMP to MPI”. In: *Proceedings of the 19th annual international conference on Supercomputing*. ICS ’05. Cambridge, Massachusetts: ACM, 2005, pp. 189–198. ISBN: 1-59593-167-8. DOI: http://doi.acm.org/10.1145/1088149.1088174. URL: http://doi.acm.org/10.1145/1088149.1088174.

[11] Michael Bauer et al. “Programming the memory hierarchy revisited: supporting irregular parallelism in sequoia”. In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. PPoPP ’11. San Antonio, TX, USA: ACM, 2011, pp. 13–24. ISBN: 978-1-4503-0119-0. DOI: http://doi.acm.org/10.1145/1941553.1941558. URL: http://doi.acm.org/10.1145/1941553.1941558.

[12] Pieter Bellens. “Programming Model and Run-Time Optimizations for the Cell/B.E.” phdPhD Thesis. Universitat Politècnica de Catalunya (UPC), 2012.

[13] L. S. Blackford et al. *ScaLAPACK Users’ Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997. ISBN: 0-89871-397-8 (paperback).

[14] Robert D. Blumofe et al. “Cilk: an efficient multithreaded runtime system”. In: *SIGPLAN Not.* 30.8 (1995), pp. 207–216. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/209937.209958.

[15] D. Bonachea. *GASNet Specification, v1.8*. Tech. rep. U.C. Berkeley, 2006.

[16] Javier Bueno et al. “Implementing OmpSs support for regions of data in architectures with multiple address spaces”. In: *Proceedings of the 27th international ACM conference on International conference on supercomputing*. ACM. 2013, pp. 359–368.

[17] J. Bueno et al. “Productive cluster programming with OmpSs”. In: *Euro-Par 2011 Parallel Processing* (2011), pp. 555–566.

[18] J. Bueno et al. “Productive Programming of GPU Clusters with OmpSs”. In: *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE. 2012, pp. 557–568.

[19] J. Bueno et al. “Reducing data access latency in SDSM systems using runtime optimizations”. In: *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*. ACM. 2010, pp. 160–173.

[20] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, W. Zwaenepoel. “TreadMarks: Shared Memory Computing on Networks of Workstations”. In: *IEEE COmputer 29 (2)* (1996), pp. 18–28.

[21] B.L. Chamberlain, D. Callahan, and H.P. Zima. “Parallel Programmability and the Chapel Language”. In: *Int. J. High Perform. Comput. Appl.* 21 (3 Aug. 2007), pp. 291–312. ISSN: 1094-3420. DOI: 10.1177/1094342007078442. URL: http://portal.acm.org/citation.cfm?id=1286120.1286123.

[22] Philippe Charles et al. “X10: an object-oriented approach to non-uniform cluster computing”. In: *SIGPLAN Not.* 40 (Oct. 2005), pp. 519–538. ISSN: 0362-1340. URL: http://dx.doi.org/10.1145/1094811.1094852.

[23] P. Charles et al. “X10: an object-oriented approach to non-uniform cluster computing”. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA ’05. San Diego, CA, USA: ACM, 2005, pp. 519–538. ISBN: 1-59593-031-0. DOI: http://doi.acm.org/10.1145/1094811.1094852. URL: http://doi.acm.org/10.1145/1094811.1094852.

[24] Li Chen et al. “Unified Parallel C for GPU Clusters: Language Extensions and Compiler Implementation”. In: *Languages and Compilers for Parallel Computing*. Ed. by Keith Cooper, John Mellor-Crummey, and Vivek Sarkar. Vol. 6548. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, pp. 151–165.

[25] T. Chen et al. “Cell Broadband Engine Architecture and Its First Implementation: A Performance View”. In: *IBM J. Res. Dev.* 51.5 (Sept. 2007), pp. 559–572. ISSN: 0018-8646. DOI: 10.1147/rd.515.0559. URL: http://dx.doi.org/10.1147/rd.515.0559.

[26] George Chrysos. “Intelő Xeon Phi Coprocessor-the Architecture”. In: *Intel Whitepaper* (2014).

[27] Jan Ciesko et al. “Programmable and Scalable Reductions on Clusters.” In: *IPDPS*. 2013, pp. 560–568.

[28] UPC Consortium. *UPC Language Specifications V1.2*. May 2005.

[29] UPC Consortium. *UPC Specifications, v1.2*. Tech. rep. Lawrence Berkeley National Lab Tech Report LBNL-59208, 2005.

[30] Pete Cooper et al. “Offload – Automating Code Migration to Heterogeneous Multi-core Systems”. In: *Lecture Notes in Computer Science, HiPEAC Conference 2010*. 2010, pp. 307–321. ISBN: 978-3-642-11514-1. DOI: 10.1007/978-3-642-11515-8_25.

[31] J. J. Costa et al. “Running OpenMP applications efficiently on an everything-shared SDSM”. In: *In Proc. of IPDPS 04*. 2004, pp. 35–42.

[32] Ludovic Courtès. *C Language Extensions for Hybrid CPU/GPU Programming with StarPU*. Research Report RR-8278. INRIA, Apr. 2013, p. 25. URL: http://hal.inria.fr/hal-00807033.

[33] Stephen Craven and Peter Athanas. “Examining the Viability of FPGA Supercomputing”. In: *EURASIP J. Embedded Syst.* 2007.1 (Jan. 2007), pp. 13–13. ISSN: 1687-3955. DOI: 10.1155/2007/93652. URL: http://dx.doi.org/10.1155/2007/93652.

[34] David E. Culler et al. “Parallel Programming in Split-C”. In: *Proceedings of the Supercomputing ’93 Conference*. Portland, OR, Nov. 1993, pp. 262–273. URL: http://www.cs.cmu.edu/~seth/papers/culler-sc93.pdf.

[35] L. Dagum and R. Menon. “OpenMP: An industry standard API for shared-memory programming”. In: *IEEE International Conference on Computational Science and Engineering*. 1998.

[36] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. “The NAS Parallel Benchmarks 2.0”. In: (1995).

[37] Romain Dolbeau, Stéphane Bihan, and François Bodin. “HMPP: A Hybrid Multi-core Parallel Programming Environment”. In: *Workshop on General Processing Using GPUs*. 2006.

[38] Jack J. Dongarra, I. High, and Productivity Computing Systems. *Overview of the HPC Challenge Benchmark Suite*. URL: http://icl.cs.utk.edu/hpcc/.

[39] J. Duato et al. “Enabling CUDA acceleration within virtual machines using rCUDA”. In: *High Performance Computing (HiPC), 2011 18th International Conference on*. Dec. 2011, pp. 1–10. DOI: 10.1109/HiPC.2011.6152718.

[40] J. Duato et al. “rCUDA: Reducing the number of GPU-based accelerators in high performance clusters”. In: *High Performance Computing and Simulation (HPCS), 2010 International Conference on*. June 2010, pp. 224–231. DOI: 10.1109/HPCS.2010.5547126.

[41] E. Speight, J. K. Bennett. “Brazos: A Third Generation DSM System”. In: *Proc. of the USENIX Windows NT Workshop*. 1997.

[42] Zhe Fan et al. “GPU cluster for high performance computing”. In: *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society. 2004, p. 47.

[43] Kayvon Fatahalian et al. “Sequoia: programming the memory hierarchy”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. SC ’06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: http://doi.acm.org/10.1145/1188455.1188543. URL: http://doi.acm.org/10.1145/1188455.1188543.

[44] Massimiliano Fatica. “Accelerating linpack with CUDA on heterogenous clusters”. In: *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. GPGPU-2. Washington, D.C.: ACM, 2009, pp. 46–51. ISBN: 978-1-60558-517-8. DOI: http://doi.acm.org/10.1145/1513895.1513901. URL: http://doi.acm.org/10.1145/1513895.1513901.

[45] Francois Cantonnet, Yiyi Yao, Mohamed M. Zahran, and Tarek A. El-ghazawi. “Productivity Analysis of the UPC Language”. In: *International Parallel and Distributed Processing Symposium/International Parallel Processing Symposium*. 2004.

[46] Thierry Gautier et al. “XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures”. In: *27th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. Boston, Massachusetts, United States, May 2013. URL: https://hal.inria.fr/hal-00799904.

[47] Robert A. Van De Geijn and Jerrell Watts. *Summa: Scalable universal matrix multiplication algorithm*. Tech. rep. 1997.

[48] Ivan Grasso et al. “LibWater: Heterogeneous Distributed Computing Made Easy”. In: *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*. ICS ’13. Eugene, Oregon, USA: ACM, 2013, pp. 161–172. ISBN:

978-1-4503-2130-3. DOI: 10.1145/2464996.2465008. URL: http://doi.acm.org/10.1145/2464996.2465008.

[49] Michael Gschwind. “Chip Multiprocessing and the Cell Broadband Engine”. In: *Proceedings of the 3rd Conference on Computing Frontiers*. CF ’06. Ischia, Italy: ACM, 2006, pp. 1–8. ISBN: 1-59593-302-6. DOI: 10.1145/1128022.1128023. URL: http://doi.acm.org/10.1145/1128022.1128023.

[50] Tsuyoshi Hamada and Keigo Nitadori. “190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs”. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–9. ISBN: 978-1-4244-7559-9. DOI: http://dx.doi.org/10.1109/SC.2010.1. URL: http://dx.doi.org/10.1109/SC.2010.1.

[51] Tianyi David Han and Tarek S. Abdelrahman. “hiCUDA: High-Level GPGPU Programming”. In: *IEEE Transactions on Parallel and Distributed Systems* 22 (2011), pp. 78–90. ISSN: 1045-9219. DOI: http://doi.ieeecomputersociety.org/10.1109/TPDS.2010.62.

[52] Mark Harris. “Many-core GPU Computing with NVIDIA CUDA”. In: *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ICS ’08. Island of Kos, Greece: ACM, 2008, pp. 1–1. ISBN: 978-1-60558-158-3. DOI: 10.1145/1375527.1375528. URL: http://doi.acm.org/10.1145/1375527.1375528.

[53] Martin C Herbordt et al. “Achieving high performance with FPGA-based computing”. In: *Computer* 40.3 (2007), p. 50.

[54] Jay P. Hoeflinger. “Extending OpenMP to Clusters”. In: (2006).

[55] D. Jimenez-Gonzalez, X. Martorell, and A Ramirez. “Performance Analysis of Cell Broadband Engine for High Memory Bandwidth Applications”. In: *Performance Analysis of Systems Software, 2007. ISPASS 2007. IEEE International Symposium on*. Apr. 2007, pp. 210–219. DOI: 10.1109/ISPASS.2007.363751.

[56] N.P. Karunadasa and D.N. D. N. Ranasinghe. “Accelerating High Performance Applications with CUDA and MPI”. In: *Proceedings of the Fourth International Conference on Industrial and Information Systems (ICIIS 2009)*. Sri Lanka, Dec. 28, 2009.

[57] Christoph Kessler et al. “Programmability and Performance Portability Aspects of Heterogeneous Multi-/Manycore Systems”. Anglais. In: *Design, Automation and Test in Europe (DATE)*. Dresden, Allemagne, Mar. 2012. URL: http://hal.inria.fr/hal-00776610.

[58] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.2*. Nov. 15, 2011. URL: http://khronos.org/registry/cl/specs/opencl-1.2.pdf.

[59] Volodymyr V Kindratenko et al. “GPU clusters for high-performance computing”. In: *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*. IEEE. 2009, pp. 1–8.

[60] David Kirk. “NVIDIA Cuda Software and Gpu Parallel Computing Architecture”. In: *Proceedings of the 6th International Symposium on Memory Management*. ISMM '07. Montreal, Quebec, Canada: ACM, 2007, pp. 103–104. ISBN: 978-1-59593-893-0. DOI: 10.1145/1296907.1296909. URL: http://doi.acm.org/10.1145/1296907.1296909.

[61] Timothy J. Knight et al. “Compilation for Explicitly Managed Memory Hierarchies”. In: *Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2007.

[62] L. Whately, R. Pinto, R. Bianchini, C. L. Amorim. “Adaptative Techniques for Home-Based Software DSMs”. In: *13th Symposium on Computer Architecture and High Performance Computing* (2001).

[63] Orion S. Lawlor. “Message passing for GPGPU clusters: CudaMPI”. In: *Proceedings of the IEEE International Conference on Cluster Computing*. 2009, pp. 1–8. DOI: 10.1109/CLUSTR.2009.5289129.

[64] Seyong Lee and Rudolf Eigenmann. “OpenMPC: Extended OpenMP Programming and Tuning for GPUs”. In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. ISBN: 978-1-4244-7559-9. DOI: http://dx.doi.org/10.1109/SC.2010.36. URL: http://dx.doi.org/10.1109/SC.2010.36.

[65] Piotr R Luszczek et al. “The HPC Challenge (HPCC) benchmark suite”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0. DOI: http://doi.acm.org/10.1145/1188455.1188677. URL: http://doi.acm.org/10.1145/1188455.1188677.

[66] Piotr Luszczek et al. *Introduction to the HPC Challenge Benchmark Suite*. Tech. rep. 2005.

[67] M. Hess, G. Jost, M. Müller, R. Rühle. "Experiences using OpenMP based on Compiler Directed Software DSM on a PC Cluster". In: *Workshop on OpenMP Applications and Tools (WOMPAT'02*. 2002.

[68] M. Sato, S. Satoh, K. Kusano, Y. Tanaka. "Design of an OpenMP Compiler for an SMP Cluster". In: *EWOMP'99*. 1999, pp. 32–39.

[69] G. Martinez, M. Gardner, and Wu-chun Feng. "CU2CL: A CUDA-to-OpenCL Translator for Multi- and Many-Core Architectures". In: *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*. Dec. 2011, pp. 300–307. DOI: 10.1109/ICPADS.2011.48.

[70] Xavier Martorell et al. "Thread fork/join techniques for multi-level parallelism exploitation in NUMA multiprocessors". In: *Proceedings of the 13th international conference on Supercomputing*. ACM. 1999, pp. 294–301.

[71] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25.

[72] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. URL: http://www.cs.virginia.edu/stream/.

[73] John D. McCalpin. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Tech. rep. A continually updated technical report. Charlottesville, Virginia: University of Virginia, 1991-2007. URL: http://www.cs.virginia.edu/stream/.

[74] The TOP500 Editors: Hans Meuer et al. *TOP500 Supercomputer Sites*. June 2014. URL: http://www.top500.org/lists/2014/06/.

[75] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.2*. available at: http://www.mpi-forum.org. Sept. 2009.

[76] Masahiro Nakao et al. "XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters". In: *Proceedings of the First Workshop on Accelerator Programming Using Directives*. WACCPD '14. New Orleans, Louisiana: IEEE Press, 2014, pp. 27–36. ISBN: 978-1-4799-7023-0. DOI: 10.1109/WACCPD.2014.6. URL: http://dx.doi.org/10.1109/WACCPD.2014.6.

[77] Masahiro Nakao et al. “XcalableMP Implementation and Performance of NAS Parallel Benchmarks”. In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. PGAS ’10. New York, New York, USA: ACM, 2010, 11:1–11:10. ISBN: 978-1-4503-0461-0. DOI: 10.1145/2020373.2020384. URL: http://doi.acm.org/10.1145/2020373.2020384.

[78] M. Nakao et al. “Productivity and Performance of Global-View Programming with XcalableMP PGAS Language”. In: *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*. May 2012, pp. 402–409. DOI: 10.1109/CCGrid.2012.118.

[79] R Numwich and J Reid. *Co-Array Fortran for parallel programming*. Tech. rep. 1998.

[80] NVIDIA. *CUDA cuBLAS Library v6.5*. Tech. rep. Aug. 2014.

[81] *NVIDIA CUDA Compute Unified Device Architecture Version 2.0*. NVIDIA Corporation. 2008.

[82] OpenACC-Standard.org. “The OpenACC Application Programming Interface Version 2.0”. In: 2013.

[83] OpenMP ARB. *OpenMP Application Program Interface, v. 3.0*. May 2008.

[84] S. J. Pennycook et al. “Performance analysis of a hybrid MPI/CUDA implementation of the NASLU benchmark”. In: *SIGMETRICS Perform. Eval. Rev.* 38 (4 Mar. 2011), pp. 23–29. ISSN: 0163-5999. DOI: http://doi.acm.org/10.1145/1964218.1964223. URL: http://doi.acm.org/10.1145/1964218.1964223.

[85] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. “A Dependency-aware Task-based Programming Environment for Multi-core Architectures”. In: *IEEE Int. Conference on Cluster Computing* (Sept. 2008), pp. 142–151. ISSN: 1552-5244. DOI: 10.1109/CLUSTR.2008.4663765.

[86] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. “Handling task dependencies under strided and aliased references”. In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS ’10. Tsukuba, Ibaraki, Japan: ACM, 2010, pp. 263–274. ISBN: 978-1-4503-0018-6. DOI: 10.1145/1810085.1810122. URL: http://doi.acm.org/10.1145/1810085.1810122.

[87] Josep M. Perez et al. *TECHNICAL REPORT 03/2007 A Flexible and Portable Programming Model for SMP and Multi-cores BSC-UPC*. 2007.

[88] Judit Planas et al. *Self-Adaptive OmpSs Tasks in Heterogeneous Environments*. 2013. DOI: 10.1109/IPDPS.2013.53.

[89] Portland Group Inc. *PGI Accelerator Compilers*. Sept. 2011.

[90] Rajesh Nishtala and George Almasi. "Performance without pain = productivity: Data layout and collective communication in UPC". In: *In Principles and Practices of Parallel Programming (PPoPP)*. 2008.

[91] Carlos Reano et al. *CU2rCU: a CUDA-to-rCUDA Converter*. 2013.

[92] C. Reano et al. "CU2rCU: Towards the complete rCUDA remote GPU virtualization and sharing solution". In: *High Performance Computing (HiPC), 2012 19th International Conference on*. Dec. 2012, pp. 1–10. DOI: 10.1109/HiPC.2012.6507485.

[93] Ruymán Reyes et al. "accULL: An OpenACC implementation with CUDA and OpenCL support". In: *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 871–882.

[94] Alexander Stepanov and Meng Lee. *The standard template library*. Vol. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304, 1995.

[95] Yoshizumi Tanaka et al. "Performance evaluation of OpenMP applications with nested parallelism". In: *Languages, Compilers, and Run-Time Systems for Scalable Computers*. Springer, 2000, pp. 100–112.

[96] Xavier Teruel et al. "Extending Nanos RTL for OpenMP Task Support". In: (2007).

[97] Xavier Teruel et al. "Support for OpenMP tasks in Nanos v4". In: *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*. IBM Corp. 2007, pp. 256–259.

[98] Sain-Zee Ueng et al. "CUDA-lite: Reducing GPU Programming Complexity". In: *In Languages and Compilers for Parallel Computing (LCPC) 21st Annual Workshop*. Aug. 2008.

[99] Didem Unat, Xing Cai, and Scott B. Baden. "Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C". In: *Proceedings of the 25th ACM International Conference on Supercomputing*. ICS '11. Tucson, Arizona, USA: ACM, 2011, pp. 214–224. ISBN: 978-1-4503-0102-2. DOI: http://doi.acm.org/10.1145/1995896.1995932. URL: http://doi.acm.org/10.1145/1995896.1995932.

[100] W. Hu, W. Shi, Z. Tang. "JIAJIA: An SVM System Bbased on A New Cache Coherence Protocol". In: *Proceedings of the High Performance Computing and Networking (HPCN'99)*. Springer, Amsterdam, Netherlands, 1999, pp. 463–472.

[101] Sandra Wienke et al. “OpenACCfirst experiences with real-world applications”. In: *Euro-Par 2012 Parallel Processing*. Springer, 2012, pp. 859–870.

[102] Y. C. Hu, H. Lu, A. L. Cox, W. Zwaenepoel. “OpenMP for Networks of SMPs”. In: *Journal of Parallel and Distributed Computing 60 (12)* (2000), pp. 1512–1530.

[103] Y. Kee, J. Kim, S. Ha. “ParADE: An OpenMP Programming Environment for SMP Cluster Systems”. In: *Supercomputing 2003 (SC’03)*. 2003.

[104] Katherine A. Yelick et al. “Productivity and performance using partitioned global address space languages”. In: *PASCO*. ACM, 2007, pp. 24–32.

[105] Katherine A. Yelick et al. “Titanium: A High-performance Java Dialect”. In: *Concurrency - Practice and Experience* 10.11-13 (1998), pp. 825–836.

[106] Yili Zheng et al. “Extending unified parallel C for GPU computing”. In: *SIAM Conference on Parallel Processing for Scientific Computing (SIAMPP)*. 2010.