A thesis submitted in fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Architecture

Methodology for malleable applications on distributed memory systems

Doctoral thesis by: Jimmy Aguilar Mena
Thesis Supervisor: Paul Carpenter
Thesis Tutor: Xavier Martorell

Departament d'Arquitectura de Computadors (DAC)
Universitat Politècnica de Catalunya (UPC)

Barcelona, 2022

Abstract

The dominant programming approach for scientific and industrial computing on clusters is MPI+X. While there are a variety of approaches within the node, denoted by the “X”, Message Passing Interface (MPI) is the standard for programming multiple nodes with distributed memory. This thesis argues that the OmpSs-2 tasking model can be extended beyond the node to naturally support distributed memory, with three benefits:

First, at small to medium scale the tasking model is a simpler and more productive alternative to MPI. It eliminates the need to distribute the data explicitly and convert all dependencies into explicit message passing. It also avoids the complexity of hybrid programming using MPI+X.

Second, the ability to offload parts of the computation among the nodes enables the runtime to automatically balance the loads in a full-scale MPI+X program. This approach does not require a cost model, and it is able to transparently balance the computational loads across the whole program on all its nodes.

Third, because the runtime handles all low-level aspects of data distribution and communication, it can change the resource allocation dynamically, in a way that is transparent to the application.

This thesis describes the design, development and evaluation of OmpSs-2@Cluster, a programming model and runtime system that extends the OmpSs-2 model to allow a virtually unmodified OmpSs-2 program to run across multiple distributed memory nodes. For well-balanced applications, it provides similar performance to MPI+OpenMP on up to 16 nodes, and it improves performance by up to 2× for irregular and unbalanced applications such as Cholesky factorization.

This work also extended OmpSs-2@Cluster for interoperability with MPI and Barcelona Supercomputing Center (BSC)’s state-of-the-art Dynamic Load Balancing (DLB) library in order to dynamically balance MPI+OmpSs-2 applications by transparently offloading tasks among nodes. This approach reduces the execution time of a microscale solid mechanics application by 46% on 64 nodes, and on a synthetic benchmark, it is within 10% of perfect load balancing on up to 8 nodes.

Finally, the runtime was extended to transparently support malleability, i.e. to dynamically change the number of computational nodes being used by the program. The only change to the application is to explicitly call an API function to request the addition or removal of nodes. The entire process, including data redistribution and interaction with the Resource Management System (RMS) is done by the runtime system. We also provide a checkpoint-restart API as an alternative malleability approach that saves the application state and restarts with a different number of nodes.

In summary, OmpSs-2@Cluster expands the OmpSs-2 programming model to encompass distributed memory clusters. It allows an existing OmpSs-2 program, with few, if any, changes to run across multiple nodes. OmpSs-2@Cluster supports transparent multi-node dynamic load balancing for MPI+OmpSs-2 programs and enables semi-transparent malleability for OmpSs-2@Cluster programs. The runtime system has a high level of stability and performance, and it opens several avenues for future work.

Keywords: OmpSs-2, Runtime, Distributed memory, MPI, Cluster.

Acknowledgements

I start by thanking God for everything, for the health and the strength to resist, and for those negligible coincidences I don't even know about, some of which brought me here in the first place and some others prevented me from perishing on this very long way.

To my family, the only real reason why all this sacrifice made some sense in the first place. You have been the motivation, support, and driving force since I have memory. Mima, Yaya, Michel... this would be much more difficult without you all always around.

To those that are not with us anymore, Pipo, Francisco, and the father Mario. The lord didn't give you time to see this from here. But without you this moment would never have come.

To my girlfriend and her family, your patience these five years is invaluable in goods of this world.

To my supervisor Paul, you deserves much more praise than myself for this work in all the senses... including the patience to overcome the countless setbacks, the unconditional availability and, of course, the human support.

To my previous professors and supervisors, specially to Ivan from ICTP; you risked a lot with me, I hope I made it worthwhile.

To the pre-defense committee and to the external reviewers of this thesis for all your comments suggestions and your time which helped to improve the manuscript.

To Barcelona Supercomputing Center, their employees, and specially those that are also friends. Thanks to all of you. I am very grateful for all the personal, professional and academic growth and for all the experiences. The way was hard, but there is no resurrection without Calvary.

The author

This research has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 671578 (ExaNoDe), 754337 (EuroEXA) and 955606 (DEEP-SEA). It is supported by the Spanish State Research Agency - Ministry of Science and Innovation (contract PID2019-107255GB-C21/MCIN/AEI/10.13039/501100011033 and Ramon y Cajal fellowship RYC2018-025628-I /MCIN/AEI/10.13039/501100011033 and by "ESF Investing in your future"), as well as by the Generalitat de Catalunya (2017-SGR-1414).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The dominant programming approach for scientific and industrial computing on clusters is MPI+X. While there are a variety of approaches within the node, denoted by the "X", such as OpenMP [137], OmpSs [64]/OmpSs-2 [21], OpenACC [136] and others, the *de facto* standard for programming multiple nodes with distributed memory is Message Passing Interface (MPI) [74, 75, 76]. MPI provides a high-performance and highly scalable interface that matches the system architecture of individual nodes connected by a high-speed network. It is, therefore, no surprise that almost all High-Performance Computing (HPC) applications communicate among nodes entirely via MPI.

The tasking approach of OpenMP [137, 138] and OmpSs [64]/OmpSs-2 [23, 22, 24] offers an open, clean and stable way to target a wide range of hardware on a node. Each task is a self-contained computation and multiple tasks can run concurrently on different cores of an Shared Memory Multiprocessor System (SMP), or they can be offloaded to accelerators such as a Graphical Processor Units (GPUs) or Field-Programmable Gate Arrays (FPGAs) [38]. Most current approaches, however, cannot run tasks on the resources (cores, GPUs, FPGAs, etc.) that belong to a different node. The first idea of this thesis is to address this limitation in OmpSs-2, extending its tasking model to naturally support distributed memory clusters. We design, develop and evaluate OmpSs-2@Cluster, an extension of OmpSs-2 that allows an unmodified (in many cases, to be explained later) OmpSs-2 program to execute efficiently on up to about 16 to 32 nodes, using all the resources on these nodes. It automatically offloads tasks among nodes and distributes taskloops among all cores on all nodes, with the details of data distribution, synchronization and communication handled by the runtime system.

OmpSs-2@Cluster is demonstrated and evaluated via an implementation using the Mercurium [18] source-to-source compiler and the Nanos6@Cluster [24] runtime system. At a small to medium scale, OmpSs-2@Cluster is an alternative to MPI or MPI+X. As it inherits the sequential semantics and common address space of OmpSs-2, the program can be much simpler than a distributed memory implementation using MPI or MPI+OmpSs-2. On the other hand, several runtime optimizations are needed to mitigate potential serialization or bottlenecks due to the sequential semantics. The OmpSs-2 model removes the need to write the MPI implementation in the first place, assuming it does not already exist, eliminating the need to explicitly distribute the data and convert all dependencies among computations into explicit message passing. It also avoids the complexity of Hybrid programming using MPI+X, which brings questions of correctness, deadlocks and differing abstractions. While these problems can be handled well by existing expert programmers, they increase development and maintenance time and cost. Pure OmpSs-2@Cluster can therefore improve programmer productivity.

OmpSs-2@Cluster is a complete new implementation that is not directly related to OmpSs-1@Cluster [43]. The design has been revisited from scratch, implemented in Nanos6 instead of Nanos++, and several design decisions are very different. In particular, there is no centralized directory cache. The weak accesses defined in OmpSs-2 allow tasks to be offloaded before the data is ready.

In addition to improving programmer productivity, the ability for the runtime to transparently offload tasks among nodes opens a number of additional opportunities that have so far been ignored. In particular, we make new contributions to the long-standing problems of load balancing and malleability.

Load balancing is one of the oldest and important sources of inefficiency in high-performance computing. Many applications take care of load balancing (see Section 2.2 for more details), either using partitioning or a second level of parallelism based on shared memory. The partitioning approach divides the problem among the MPI ranks; based on a cost model that predicts the load as a function of the partition. However, this cost model is not always accurate, and partitioning is costly, especially if it is reapplied periodically to address varying load imbalances. Alternatively, or in addition, if the MPI ranks can use a varying number of cores (or GPU resources, accelerator, etc.), then the resources on a node can be dynamically reassigned among the MPI ranks on a node. This requires a second level of parallelism that can utilize more than one core per MPI rank, e.g. OpenMP or OmpSs. It also requires a mechanism to assign the cores on a node, such as Dynamic Load Balancing (DLB) [81]. This approach takes care of load imbalance among the ranks on a node, but the total work on each node stays the same, with no way to mitigate any remaining load imbalance.

We therefore extend OmpSs-2@Cluster to provide transparent dynamic load balancing for existing MPI+OmpSs-2 programs. If the program is fully balanced, or any load imbalance can be addressed merely by shifting cores among the ranks on each node, then the execution will be similar to the existing approach using DLB. However, if the load imbalance is such that some nodes as a whole are more heavily loaded than others, then OmpSs-2@Cluster will transparently offload tasks among nodes to balance the loads, shifting the assignment of cores accordingly. Therefore, the above "shared memory" approach was extended to encompass the whole system. We show that this approach can mitigate load imbalance and significantly improve the performance of existing MPI+OmpSs-2 programs through experiments on up to 64 nodes.

The final topic is malleability, which is currently under active investigation in HPC.[1] Malleability is the ability for a running program to dynamically change its resources' utilization, e.g. by adding or removing nodes as needed. Malleability is similar to the concept of elasticity, which is well established in cloud computing.

MPI has supported basic malleability concepts since MPI-2 [74], which was published in 1997; however it is cumbersome to implement in the application and, even 20 years later, very few

[1]It is supported by the DEEP-SEA project [58], which is one of several ongoing projects that partially investigate malleability in HPC.

applications currently have any malleability support. Nonetheless, Malleability support can help improve the overall system utilization, but it requires support across the whole stack, ranging from applications, MPI library, to the batch scheduler.

We extended OmpSs-2@Cluster to support malleability for OmpSs-2@Cluster programs. A pure OmpSs-2@Cluster program, i.e. one that does not also include MPI, naturally supports malleability. The only change needed in the application is to call an Application Programming Interface (API) function to request the addition or removal of nodes to the computation. Since the virtual memory layout is independent of the number of nodes and the runtime takes care of all data distribution and task scheduling, the application does not need to do anything else. In the future, even the decisions to add or remove nodes may be moved to the runtime.

1.1 OmpSs-2

OmpSs-2 [23, 22, 24] is the second generation of the OmpSs programming model. It is open source and mainly used as a research platform to explore and demonstrate ideas that may be proposed for standardization in OpenMP. The name originally comes from two other programming models: OpenMP and Star SuperScalar (StarSs). The design principles of these two programming models constitute the fundamental ideas used to conceive the OmpSs and OmpSs-2 philosophy.

Like OpenMP, OmpSs-2 is based on directives that annotate a sequential program, and it enables parallelism in a dataflow way [144]. The model targets multi-cores and GPU/FPGA accelerators. This decomposition into tasks and data accesses is used by the source-to-source Mercurium [18] compiler to generate calls to the Nanos6 [19] runtime API. The runtime computes task dependencies and schedules and executes tasks, respecting the implied task dependency constraints and performing data transfers and synchronizations.

The OpenMP concept of data dependencies among tasks was first proven in OmpSs [134]. As tasks are encountered within the sequential program (or parent task), these tasks and their accesses are added to a Directed Acyclic Graph (DAG), which will be used to ensure that tasks are executed in a way that respects the ordering constraints. OmpSs-2 differs from OpenMP in the thread-pool execution model, targeting of heterogeneous architectures through native kernels and asynchronous parallelism as the main mechanism to express concurrency. Task accesses may be discrete (defined by start address) or regions with fragmentation [143]. OmpSs-2 extends the tasking model of OmpSs and OpenMP to improve task nesting and fine-grained dependencies across nesting levels [143, 7]. The improved nesting support enables effective application parallelization using a top-down methodology. Furthermore, the addition of weak dependencies (see Section 3.3) exposes more parallelism, allows better scheduling decisions and enables parallel instantiation of tasks with dependencies between them.

1.2 OmpSs-2@Cluster model for distributed systems

Chapter 3 introduces the OmpSs-2@Cluster programming model and its runtime implementation. Development of programming models for multi-core and many-core distributed systems is one of the main challenges in computing research. Any modern programming model should:[146] (1) be able to exploit the underlying platform, (2) have good programmability to enable programmer productivity, (3) take into account the underlying heterogeneous and hierarchical platforms, (4) enable the programmer to be unaware of part of that complexity.

No programming model will get used in practice if it cannot effectively exploit the platform (challenge 1). The large number of optimizations that were needed to realize our vision of transparent task offloading was the hardest and most time-consuming aspect of this research.

Some optimizations, such as dedicating a thread to receive control messages, processing some of these messages in parallel on idle cores, aggressively aggregating MPI messages to reduce overhead, data versioning, and so on are not new, but they took significant time to implement, and they are the cost of starting a new runtime implementation. Other optimizations, primarily the namespace (Section 3.6.1), which transparently allows direct propagation between two offloaded tasks on the same node, without any synchronization or critical-path messages to the original node are novel. All of these optimizations were necessary before we could make any publication with a compelling application or benchmark performance.

Programmer productivity and a clean abstraction were key considerations in this work (challenge 2). In traditional distributed memory programming, the developer must expose parallelism and manage the resources and decompose and express their computations for the machine. In an MPI+X program, the parallelism must be divided between the distributed-memory and shared-memory parts in a way that is performance-portable among machines with widely different configurations. In practice, the developer needs to deeply understand these complex systems, and much of this work is repetitive and time-consuming. Furthermore, to address efficient utilization of such heterogeneous resources, we need programming paradigms with the ability to express parallelism in flexible and productive manners [97]. A task-based programming model provides a unified way to handle more powerful and complex nodes, which can otherwise be hard to target in a performance-portable way. By handling low-level aspects in the runtime system, OmpSs-2@Cluster makes it possible to write clearer and more intuitive code that is not obscured by these low-level details.

OmpSs-2@Cluster provides a simple path to move an existing OmpSs-2 program from single node to small/medium-scale clusters. Any OmpSs-2 program with complete annotation of its dependencies may be a valid OmpSs-2@Cluster program (see: Section 3.3.2). In OmpSs-2@Cluster, there is no strict separation between the abstractions for distributed and shared memory. Execution starts with an annotated sequential program, as maintaining sequential semantics and exposing a common user-facing address space are non-negotiable and critical features that drive the productivity of the OmpSs-2 model. All ranks collaborating to execute an OmpSs-2@Cluster program are set up with the same virtual address space, so data is copied among nodes without changing its address. Conversely, the lack of address translation simplifies the runtime and provides no surprises to the user.

The Nanos6@Cluster runtime considers the underlying heterogeneous and hierarchical platform (challenge 3). It includes Non-uniform Memory Access (NUMA)-awareness and NUMA scheduling, inherited from the SMP support in Nanos6, and contains some small updates to encompass the OmpSs-2@Cluster programming model. However, cluster extension integration with the existing support in Nanos6 for GPUs and FPGAs has not been done yet. Such work is natural, as these parts of the runtime are mostly orthogonal and is anticipated to be done in future.

Finally, OmpSs-2@Cluster allows the programmer to be unaware of much of the system's complexity (challenge 4), including data distribution and message passing, which are abstracted by the programming model. The ability to offload tasks means that OmpSs-2@Cluster enables transparent dynamic load balancing and malleability, two complex capabilities that are difficult to realize using existing MPI or MPI+X approaches.

1.3 Dynamic load balancing

Load imbalance is one of the oldest and most important sources of inefficiency in high-performance computing. The issue is only getting worse with increasing application complexity, e.g. Adaptive Mesh Refinement (AMR), modern numerics, system complexity, e.g. Dynamic Voltage and Frequency Scaling (DVFS), complex memory hierarchies, thermal and power management [1], Operating System (OS) noise, and manufacturing process [100]. Load balancing is usually applied in the application [108, 133, 12], but it is time-consuming to implement, it obscures application logic and may need extensive code refactoring. It requires an expert in application analysis and may not be feasible in very dynamic codes. Automated load imbalance has been attacked from multiple points of view, and state-of-the-art solutions generally only target parts of the load imbalance problem, as explained in Section 2.2. Static approaches, such as global partitioning, are applied at fixed synchronization points that stop all other work [161, 120], and they rely on a cost model that may not be accurate. This task gets harder as the size of the clusters continues to increase.

Chapter 4 presents our solution for MPI+OmpSs-2 programs, which relieves applications from the load balancing burden and addresses all sources of load imbalance. Only minor and local changes are needed to be compatible with the model because no additional markup is required beyond the existing OmpSs-2 task annotations. The dynamic load balancing is done based on runtime measurements of the average number of busy cores in a way that is entirely transparent to the application. It leverages the malleability of OmpSs-2 in terms of dynamically changing numbers of cores, the core assignment capabilities of DLB and task offloading with work stealing from OmpSs-2@Cluster.

The key aspect is that each MPI rank visible to the application can directly offload work across a small number of helper processes on other nodes. We use the concept of an expander graph from graph theory, which is a graph with strong connectivity properties, so that load imbalance is not stuck in a local bottleneck. This approach limits the amount of point-to-point communication and state that needs to be maintained, and it provides a path to scalability to large numbers of nodes.

1.4 Malleability

Several studies [85, 156, 70] demonstrate a suboptimal use of resources in many of the TOP500 supercomputers [179]. Users often over-provision their jobs by requesting more resources than the application can efficiently utilize, or the application is only able to efficiently use all its allocated resources for a small portion of the total execution.

Although most traditional HPC applications allow the user to select the number of resources at submission time (they are *moldable*), this allocation cannot be modified once the application has started. The MPI paradigm means that once an application starts running, it exclusively occupies a fixed number of nodes. As a side effect, all other resources, including accelerators such as GPUs, that are attached to these nodes are also occupied during the entire execution of the application.

Chapter 5 presents our approach for malleability of OmpSs-2@Cluster programs. An alternative to full malleability is Checkpoint and Restart (C/R), which is a commonly used approach, but it stops and restarts the application, and it moves state through the Input Output (IO) hierarchy and filesystem. We introduce a C/R API for OmpSs-2@Cluster, implemented it inside the runtime using MPI-IO, and use C/R as the baseline for evaluation.

We explain how a pure OmpSs-2@Cluster program, i.e. one that does not interoperate with MPI, inherently supports malleability. In brief, this is because the virtual memory layout of the program is independent of the resource allocation and all data distribution and task scheduling is delegated to the runtime. We implement full transparent malleability support in Nanos6@Cluster, based on `MPI_Comm_spawn`. In-memory data migration and dynamic process management are handled within the runtime system. When nodes are removed, it may be necessary to migrate data to ensure that no data is lost, i.e. that at least one copy of the latest data is present on the remaining nodes. The remaining data can be eagerly redistributed among the nodes according to the new distribution, or it may be lazily redistributed. In most cases, the lazy redistribution scheme offers the greater potential for communication–computation overlap.

The introduction of MPI Sessions [95] into the MPI-5.0 standard makes this a good time to implement malleability in the runtime. Our current results are somewhat constrained by the peculiarities of the `MPI_Comm_spawn` interface since processes must be released back to the system in the same order and with no finer granularity than they were spawned. This introduces a tradeoff because processes may need to be spawned one by one at a much higher overhead than in a large group so that an unknown number of nodes can be released later. This issue will be resolved once the MPI libraries include support for MPI Sessions. Few changes will be needed to the low-level parts of the Nanos6@Cluster runtime to make use of this functionality and realize the full potential of our techniques.

1.5 Thesis contributions

This thesis presents the implementation of OmpSs-2@Cluster as a malleable task-based programming model and runtime system for distributed memory systems. The main contributions in this thesis include:

1. The first task offloading extension of the OmpSs-2 programming model for distributed memory systems that supports fine-grained tasks and with a complete decentralized execution graph and memory infrastructure.

2. The runtime implementation includes optimizations and implementation details to maintain compatibility with the OmpSs-2 programming model without sacrificing flexibility or performance. Such optimizations include from the runtime side: strategies for smarter access propagation that reduce auxiliary and redundant communication. From the user point of

view, we expose a new API and directives to provide extra control and flexibility and extend the programming model. This study produced a conference paper [4].

3. The integration of the runtime system with a normal MPI application to create a load balance strategy supported by the task offloading feature of OmpSs-2@Cluster. At the same time, the runtime system was successfully integrated with the DLB library to manage underlying resources, delivering everything in a completely transparent approach for the final user. This contribution produced another conference paper [5].

4. The runtime and programming model was extended to perform malleable operations with a semi-transparent C/R approach and a fully transparent Dynamic Process Management (DPM). This last contribution includes data migration and process creation strategies and policies to provide maximum flexibility and simplicity to the user. Additionally, this functionality implements smart integration with the Resource Management System (RMS) to exploit the dynamic capabilities when available.

1.6 Publications

This thesis has resulted in two international peer-reviewed publications as a first author:

- [4] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguade, and Jesus Labarta Mancho. "OmpSs-2@Cluster: Distributed Memory Execution of Nested OpenMP-style Tasks". In: *Euro-Par 2022: Parallel Processing.* Ed. by José Cano and Phil Trinder. Cham: Springer International Publishing, 2022, pp. 319–334. ISBN: 978-3-031-12597-3. DOI: `10.1007/978-3-031-12597-3_20`

  This paper presents much of the material of Chapter 3.

- [5] Jimmy Aguilar Mena, Omar Shaaban, Victor Lopez, Marta Garcia, Paul Carpenter, Eduard Ayguade, and Jesus Labarta. "Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB". in: *51st International Conference on Parallel Processing (ICPP).* 2022

  This paper presents the material of Chapter 4.

There are two additional publications as the second author:

- [86] Guido Giuntoli, Jimmy Aguilar, Mariano Vazquez, Sergio Oller, and Guillaume Houzeaux. "A FE 2 multi-scale implementation for modeling composite materials on distributed architectures". In: *Coupled Systems Mechanics* 8.2 (2019), p. 99. DOI: `10.12989/csm.2019.8.2.099`

  This paper presents the MicroPP benchmark, which was ported to OmpSs-2@Cluster for the evaluation in Chapter 4. Most of the optimizations and modifications implemented to make the code suitable for parallelization and to profile the application were introduced into the mainline development of MicroPP.

- [164] Omar Shaaban, Jimmy Aguilar Mena, Vicenç Beltran, Paul Carpenter, Eduard Ayguade, and Jesus Labarta Mancho. "Automatic aggregation of subtask accesses for nested OpenMP-style tasks". In: *IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2022

  This paper presents the auto keyword, which simplifies the development of programs using nested tasks and allows subtask memory allocation. This work was done in close collaboration with the programming model definition and runtime development.

1.7 Objective

The main objective of this research is to develop a runtime implementation of the OmpSs-2 directive task-based programming model for distributed memory systems with malleable capabilities. Such a runtime may simplify the development and porting of HPC parallel applications in small-medium size clusters and improve the resource utilization of the final applications by automatizing or simplifying access to common optimizations. Hence, allowing interoperability with other models, load balance and application dynamic resource management.

To achieve the objective, we have the following specific objectives:

- Develop a functional version of the Nanos6@Cluster runtime for distributed memory systems, using and following the OmpSs-2 programming model specifications to use as a baseline.
- Test and profile the first implementation with synthetic benchmarks to detect optimization opportunities in the runtime and limitations or peculiarities which may require modifications of the OmpSs-2 programming model for the cluster use case.
- Modify the Nanos6@Cluster runtime system to interoperate with external libraries and external MPI applications to integrate OmpSs-2@Cluster with the DLB library to explore load-balancing benefits in distributed memory systems.
- Add an API using MPI-IO to provide semi-transparent C/R capabilities to OmpSs-2@Cluster applications in order to perform on-disk process reconfiguration.
- Extend the OmpSs-2@Cluster and runtime system using the MPI DPM features to provide transparent malleability functionalities and integration with the RMS (Slurm) to the applications.

1.8 Document structure

The rest of this thesis is divided into five chapters.

Chapter 2 presents the related work and state-of-the-art relevant for the following chapters. The chapter is divided in three sections corresponding to the following chapters in the documents.

Chapter 3 develops the fundamental concepts of the OmpSs-2@Cluster programming model and the Nanos6@Cluster runtime implementation. It motivates and describes a number of crucial runtime optimizations and demonstrates scalability on up to about 16 to 32 nodes. It also describes the OmpSs-2-TestGen test generator program and automatic test case simplifier, which were crucial to obtaining high stability of the runtime.

Chapter 4 extends OmpSs-2@Cluster for interoperability with MPI to allow dynamic multi-node load balancing of MPI+OmpSs-2 programs. It starts several helper processes on each node, which can be used to execute offloaded tasks from the application's MPI ranks on other nodes. It uses Barcelona Supercomputing Center (BSC)'s DLB library to reassign cores among the application and helper ranks on each node, together with a work stealing task scheduler.

Chapter 5 develops application-transparent malleability for OmpSs-2@Cluster. Since an OmpSs-2@Cluster program is independent of the number of nodes on which it executes, the number of nodes can be updated dynamically, with all interactions with the batch scheduler, spawning and removing of nodes, and data redistribution handled transparently by the runtime.

Finally, Chapter 6 concludes the document and proposes some avenues for future work. We have built the core OmpSs-2@Cluster programming model and runtime implementation and proven its scalability and utility for dynamic load balance and malleability. All the codes and benchmarks developed over the course of this research are publicly available as open source in the hope that future researchers will build upon our work.

Chapter 2

State of the art and related work

This chapter covers the related work and state of the art for the three key topics of this thesis. Section 2.1 covers task-based programming for Shared Memory Multiprocessor Systems (SMPs) and distributed memories, and it is relevant for the whole thesis. Section 2.2 covers the state of the art for load balancing, which is relevant for Chapter 4. Finally, Section 2.3 covers the related work for malleability, which is relevant for Chapter 5.

2.1 Task-based programming models

2.1.1 Shared memory tasks

Task parallelism is supported in numerous frameworks for shared memory systems. Some have simple and explicit approaches, while others allow parallelization and ordering with more complex approaches of using specific programming schemes, e.g. C++-11 allows the mixing of `std::threads`, `std::futures`, `std::promises` and `std::mem_fn` to provide a primitive task-like behavior. Rust and some modern programming languages provide similar functionalities with native features.

Cilk is a task-based programming model for shared memory, based on tasks identified with the `spawn` keyword and synchronization to wait for spawned tasks using the `sync` statement [34]. It is perhaps the first well-known task-based programming model. Cilk++ extends the model to support parallel loops [118]. The Cilk approach influenced similar work like the research of Vandierendonck et al.; they present a unified parallel fork–join task based model with a Cilk-like syntax in many aspects [185].

OpenMP is a shared memory directive based approach that generates tasks from sequential code with a fork–join approach for parallel regions. It has become *de facto* standard for task programming in SMP systems since the inclusion of asynchronous tasks in version 3.0. Starting version 4.0 [137], the standard included support for data dependencies influenced by OmpSs and tested with Nanos++ [134]. Barcelona Supercomputing Center (BSC) also contributed to a later 4.5 version, which adds support for the `priority` clause to task and taskloop constructs (see: Section 3.2.1).

2.1.2 Distributed memory tasks

From the design point of view, there are three main characteristics of task-based programming models on distributed memory systems:

Granularity A typical data size or execution time of individual tasks.

Dependency graph Relationship among tasks, in terms of ordering and passing of data. These relationships are typically between tasks or between tasks and their dependencies.

Scheduling Allocation and scheduling of tasks to execute on compute resources, in order to minimize execution time, minimize data copies; i.e. maximize data locality, and avoid idle time, i.e. maximize load balance.

Other characteristics to consider: the level of language support, how to define or discover tasks, whether tasks can create their own subtasks and the relationships between tasks and subtasks of different tasks, the management of distributed memory, and keeping track of the location of data.

In the literature, there are multiple frameworks that implement different variants of task programming models. Table 2.1 has a brief characterization summary for the most well-known and cited ones:

CHAMELEON is a library for fine-grained load-balancing of MPI+X applications with tasks [116]. It can dynamically adapt to changing execution conditions, such as variability in node performance (Dynamic Voltage and Frequency Scaling (DVFS), memory hierarchies, power cap) or application load balance (adaptive mesh refinement, ADER-DG does extra work when the numerical solution is not physically admissible). It uses the OpenMP target offloading construct to define task data accesses, and data is always copied back to the parent after the execution of each task. Unlike OmpSs-2@Cluster, CHAMELEON is intended only to serve as a mechanism for dynamic load balancing; see Section 2.2.1 for a comparison in this context. It is not optimized for scalability with offloaded tasks and it has not been extended to support malleability.

DASH is a C++ template library that implements operations on C++ Standard Template Library (STL) containers on distributed memory using Partitioned Global Address Space (PGAS) and distributed tasking [78, 79]. Each process creates its local dependency graph in parallel. The dependencies on non-local memory are automatically resolved by the runtime system. Since there is no total ordering of the dependencies among nodes, the execution must be divided into phases so that non-local input dependencies in phase n access the latest result from phase $n-1$ or before. This increases complexity compared with the sequential programming model of OmpSs-2@Cluster.

DuctTeip is a distributed task-parallel C++ framework for hierarchical decomposition of programs using data structures such as dense matrices, sparse matrices and trees, abstracted through a data handle [191, 192]. The approach supports general task graphs and allows an efficient implementation of common application structures. It is a C++ framework, which exposes its own datatypes that must be employed by the application, which may be a significant effort for large existing applications. In contrast, OmpSs-2@Cluster supports C, C++ and potentially Fortran,[1] and it is designed for easy porting of existing applications using (multi-dimensional) arrays and pointers. DuctTeip also uses Message Passing Interface (MPI) for inter-node communication and pthreads for shared memory. It is implemented on top of the **SuperGlue** [177] shared memory library that supports data versioning. DuctTeip naturally supports multiple versions of the same data, which eliminates write-after-write dependencies. In a later work the authors of DuctTeip propose **TaskUniVerse** [190] as a general-purpose Application Programming Interface (API) that can serve as an abstraction layer for several tasking back-ends, such as cpuBLAS, SuperGlue, StarPU, DuctTeip or cuBLAS.

OmpSs-1@Cluster is a variant of OmpSs-1 for clusters of Graphical Processor Units (GPUs) [43]. It has a similar approach to OmpSs-2@Cluster, but task creation and submission are centralized, dependencies are only among sibling tasks, and address translation is needed for all task accesses. It uses a directory on one node to track all data location, rather than passing the location through the edges of a distributed dependency graph. It has only strong tasks, so it has limited ability to overlap execution with task creation overhead. It does not support interoperability with MPI or malleability.

COMP Superscalar (COMPSs) is a framework for the development of coarse-grained task-parallel applications, ensembles or workflows for distributed infrastructures, such as clusters, clouds and containerized platforms [173, 15, 11, 123]. It is the other major programming model, besides OmpSs/OmpSs-2, in the Star SuperScalar (StarSs) family. OmpSs and COMPSs both compute the task dependency graph from the task accesses, and they both provide sequential semantics. Whereas OmpSs supports C and Fortran, COMPSs targets Java, C/C++ and Python. The task granularity for COMPSs is much higher than that of OmpSs: approximately 1 second to 1 day, rather than roughly 0.1 ms to 10 ms for OmpSs. Both provide a single address space for computing dependencies, but whereas OmpSs computes dependencies based on regions of (virtual) memory, COMPSs computes dependencies on files or objects. COMPSs schedules tasks to run concurrently, respecting data dependencies across the available resources, which may be nodes in a cluster, cloud or grid. In cloud environments, COMPSs provide scalability and elasticity features with many similarities to the malleability support in Chapter 5.

Cluster Superscalar (ClusterSs) is a Java-based member of the StarSs family that targets the IBM's Asynchronous Partitioned Global Address Space (APGAS) runtime for clusters of SMPs [174]. Cluster Superscalar (StarSs) tasks are created asynchronously and assigned to the available resources using APGAS, which provides an efficient and portable communication layer based on one-sided communication. It has a strict division between the main node that generates tasks and the worker nodes that execute them. It has similar characteristics to COMPSs.

Pegasus is another workflow management system that uses a Directed Acyclic Graph (DAG) of tasks and dependencies between them. Deelman et al. [57] use **Chimera** [77] to build the abstract

[1]Fortran support is not implemented, but is anticipated in the design.

workflow based on the user-provided partial, logical workflow descriptions.

GPI-Space is a fault-tolerant execution platform for data-intensive applications [154, 30, 35]. It supports coarse-grained tasks that help decouple the domain user from the parallel execution of the problem (including fault tolerance). It is compared with Hadoop for MapReduce-style computations. It is built upon Global Address Space Programming Interface (GPI) to benefit from Remote Direct Memory Access (RDMA) and stores intermediate results in the Random Access Memory (RAM) instead of storage, which is what Hadoop does.

Charm++ is a C++-based object-oriented programming model and runtime for running migratable objects known as *chares* [112, 2, 141, 142]. It uses a message-driven runtime model in which methods result in sending a message to a chare, resulting in asynchronous function execution with some similarities to task execution. Whereas OmpSs-2, and by implication OmpSs-2@Cluster, build a dynamic dependency graph oriented to the data, Charm++ maintains a work pool of seeds for new chares and messages for existing chares. Each chare can only execute a single method at a time. Charm++ also include support for load balancing and malleability (see: Section 2.2.1 and Section 2.3.2). The OmpSs-2@Cluster approach is designed for easy porting of existing C, C++ or Fortran (in future) programs with tasks and dependencies computed from data accesses to (multi-dimensional) arrays and pointers.

PARTEE is an alternative implementation of the OmpSs programming model with improved performance for fine-grained nested tasks and irregular dependencies [140, 139]. It improves upon the earlier **BDDT** [181] task-parallel runtime by extending its block-based dynamic dependence analysis to run across multiple threads and to support nested parallelism. A version of PARTEE has been developed [28] that executes tasks across four nodes connected by the GSAS shared memory abstraction on top of UNIMEM hardware support [113]. Myrmics is a predecessor of PARTEE for a custom hardware architecture called Formic [125]. It has a hierarchical task dependency graph, which is distributed over 512 mini-nodes communicating via Direct Memory Access (DMA). It uses region-based memory allocation and has dynamic task footprints with regions.

PaRSEC is an efficient system for the distributed task execution [40, 39, 97]. The annotated source code is automatically translated into a parameterized DAG representation. The full graph is not stored or expanded in full, and information about tasks can be found through local queries on the DAG representation. Scheduling is Non-uniform Memory Access (NUMA)-aware and locality-based.

Event Driven Asynchronous Tasks (EDAT) is a library that provides a task-based model to abstract the low-level details while still keeping a realistic view of the distributed nature of the application [42]. Event Driven Asynchronous Tasks (EDAT) supports task nesting, and it uses the concept of events, which is equivalent but not the same as dependencies in other programming models. The programmer does not need to care about the low-level details of communications, such as sending and delivering events. Other distinctive concepts of EDAT are the transitory and persistent tasks and events. From the programmer's point of view, the model behaves like tasks consuming events, which are fired from a source to a target at any point in the application. The programmer's main role is to design parallelism and decide which aspects of the code interact with each other, but not manage how this actually happens at the low level.

Kokkos is a C++ library abstraction layer that enables programmers to express parallel patterns and data spaces abstractly, by relying on the Kokkos library [66, 67, 180]. It maps these to an underlying model such as OpenMP, Compute Unified Device Architecture (CUDA) or Heterogeneous Interface for Portability (HIP). This approach insulates the programmer from vendor-locked programming models by providing a path from common code to various backends increasing the portability. As an abstraction layer Kokkos does not require any compiler support, which, therefore places the burden of writing and maintaining backends on Kokkos itself, rather than sharing between the compiler community. Kokkos supports a kernel style of programming, where lambda functions are written to describe the execution of a single loop iteration (similar to the kernels of Open Computing Language (OpenCL)) and are executed over a range in parallel.

Raja and its associated libraries provide a similar approach to performance portable programming to leverage various programming models, and thus architectures, with a single-source codebase [29]. Raja targets loop-level parallelism by relying solely on standard C++11 language features for its external interface and common programming model extensions, such as OpenMP and CUDA, for its implementation. OmpSs-2@Cluster takes an alternative approach based on annotated C or C++ tasks and data accesses.

Wool is another library for nested independent task parallel programming [71]. It focuses on low overhead fine-grained tasks. Tasks can be created and spawned, and the SYNC operation blocks until the task has completed execution. It also supports parallel for loops.

XKaapi is a custom OpenMP runtime implementation for clusters of SMPs and NUMA architectures [84, 186]. The authors introduce several heuristics to use the dependency's information to modify the behavior of the application at several levels to control data and task placement; and the task stealing strategy, depending on the topology.

HPX is an implementation of the ParalleX [109] programming model, in which tasks are dynamically spawned and executed close to the data [110]. An Active Global Address Space (AGAS) is used to transparently manage global object data locality. HPX is locality-agnostic in that distributed parallelism capabilities are implicit in the programming model rather than explicitly exposed to the user. Synchronization of tasks is expressed using futures and continuations, which are also used to exchange data between tasks.

X10 is an object-oriented programming language designed by IBM for high-productivity programming of cluster computing systems [50, 187]. Similar to OmpSs-2@Cluster, X10 spawns asynchronous computations, but the programmer is responsible for describing the data across the PGAS memory. This is done using high-level programming language constructs to describe the data distribution, creating asynchronous tasks and synchronizing them. A central concept in X10 is a *place*, which is a location to place data and computation. Although OmpSs-2@Cluster has a mechanism to provide a data distribution hint in the `dmalloc` memory allocation, it is not necessary.

Fortress is a multi-purpose programming language with emphasis on mathematical syntax and large parallel systems [176, 187]. The core of the language provides built-in support for task and data parallelism with a syntax similar to Fortran. Task parallelism is managed using implicit threads and spawned (or explicit) threads, and it also supports parallel constructs similar to OpenMP's `parallel` and `parallel do` clauses. Fortress organizes memory locations in a tree of regions closely associated with the underlying structure of the system. Lower levels describe the local entities (shared memory), and higher levels represent distributed entities. Unlike OmpSs, these levels imply that the user needs to be aware of the underlying architecture and memory distribution.

Cascade High Productivity Language (Chapel) is another programming language focussed on the HPC community with task support and its own syntax [175, 187]. Chapel abstracts computational units as "locales", and task parallelism is mainly supported through parallel statements such as `forall`, which works similarly to OpenMP's `parallel for`.

X10, Fortress and Chapel, share with OmpSs-2@Cluster a similar approaches in the design of parallelism, task support, the use of partitioned global address space and the multi-threading support (which OmpSs-2@Cluster inherits from OmpSs-2@SMP). However, unlike OmpSs-2@Cluster, all of them attempt to be new programming languages, so they have a steeper learning curve for existing High-Performance Computing (HPC) developers and more effort to port applications.

ProActive is a parallel programming model and runtime for distributed-memory infrastructures, such as clusters, grids and clouds [46]. It supports parallelism for "active objects", which can run concurrently and remotely on other nodes. Each active object has its own control thread, and it serves incoming requests for executing methods. The default communication layer is Java Remote Method Invocation (RMI), which provides portability across distributed memory infrastructures at the cost of some performance. OmpSs-@Cluster is designed and optimized for performance and scalability.

StarPU is a C library and runtime system for executing applications on heterogeneous machines [14]. Some efforts [55] extend the GNU Compiler Collection (GCC) compiler suite to promote some concepts exported by the library as C language constructs. In **StarPU**, the programmer can define tasks to be executed asynchronously on the core of an SMP or offloaded to an accelerator. Similarly to OmpSs, the programmer specifies the direction so that the runtime discovers and enforces the dependencies between them.

StarPU-MPI is the multi-node extension of **StarPU** [13]. All processes create the same DAG of top-level tasks, and it uses an owner-compute model to determine which node executes the task.

The allocation of tasks to nodes is therefore done at task creation time. Whenever a dependency is between top-level tasks to be executed on different nodes, both the sender and receiver post the MPI call at task creation time. Posting the receives in advance removes the need for additional control messages to track satisfiability, but it implies a high memory consumption and some throttling mechanism, which limits the discovery of parallelism [163].

Legion is a parallel programming system with an Object Oriented Programming (OOP) syntax similar to C++ based on logical regions to describe the organization of data, with an emphasis on locality and task nesting via an object-oriented syntax [26]. Legion also enables explicit, programmer-controlled movement of data through the memory hierarchy and placement of tasks based on locality information via a mapping interface. Part of Legion's low-level runtime system uses Unified Parallel C (UPC)'s GASNet [37, 36, 117].

Programming model	*Languages*			*Task definition*		*Task discovery*		*Nested tasks*		*Address space*		*Dependencies*		*Depend. graph*			*Location tracking*		
	C	C++	Fortran	From sequential	Explicit	Static	Dynamic	Supported	Offloadable	Common	Different	Flexible	Early release	Distributed	Centralized	Duplicated	Via dependencies	Directory-based	Owner computes
OmpSs-2@Cluster	✓	✓		✓	✓		✓	✓	✓	✓		✓	✓	✓			✓		
OmpSs-1@Cluster	✓	✓		✓	✓		✓	✓	?		✓				✓			✓	
StarPU-MPI	✓	✓			✓		✓	✓			✓	?				✓			✓
DuctTeip		✓			✓		✓	✓			✓					✓		✓	
DASH		✓			✓		✓	✓		✓						✓			✓
PaRSEC	✓	?	✓		✓	✓	✓	?	?		✓					✓			✓
Charm++	✓	✓	✓		✓		✓		✓		✓								

Table 2.1: Comparison of distributed tasking models.

Table 2.1 summarizes the main frameworks for fine-grained distributed memory task parallelism. Most of them wrap or extend existing frameworks for shared memory tasking.

2.2 Dynamic load balancing

Even though load imbalance has been attacked from very different points of view (data partition, data redistribution, resource migration, etc.); these solutions generally only target the imbalance in one of its sources (input data imbalance, resource heterogeneity, resource sharing, etc.).

This is especially dramatic in MPI applications because data is not easily redistributed. In MPI based hybrid applications, the computational power loss due to load imbalance is multiplied by the number of threads used per MPI process. For these reasons, load imbalance in MPI applications is targeted in many research works.

There are many proposed solutions in the literature, most of them directly provided by the application itself, e.g., BT-MZ [108], discrete event simulation [133] or Monte Carlo simulations [12].

However, we consider a general solution that relieves applications from the load balancing burden. Moreover, it must be a solution that addresses all sources of load imbalance. In a general way, load balancing solutions can be divided into:

Before the computation redistribute the data to load balance it between the processes. These solutions can only be applied to certain data distribution (graph, mesh, etc.) and must be calculated before each execution for each different set of input data. The main limitation is that it cannot handle a dynamic load imbalance that changes during the execution. One of the best-known solutions to distribute the work before the execution is the mesh partitioner METIS [114]. Some studies also show that mesh partitioning strongly affects performance [166].

During the execution tries to load balance applications during the execution. Generally, it uses two kinds of approaches the ones that redistribute the data or the ones that redistribute the computational power. Repartitioning methods are also based on mesh partitioning that is applied periodically during the execution [161, 120].

The two main challenges of these methods are that it is not trivial to determine a load heuristic to predict the load and that the repartitioning process is time-consuming. Thus, users must apply it with caution.

2.2.1 Data redistribution to balance work

Redistributing the data is rigid in terms of the type of imbalance it is capable of solving, and it only applies to applications where data can be partitioned. Due to these two conditions, generally, the application needs modifications to be aware of the load balancing algorithm.

Charm++ To achieve load balance, the Charm++ runtime can migrate the chares before their execution, which is decided by the load balancing strategy. It automates dynamic load balancing for task-parallel as well as data-parallel applications via separate suites of load-balancing strategies. The capability can be triggered manually or automatically after a load imbalance is detected. Charm++ performs permanent migration, i.e. workpieces are migrated to a processing unit and will remain there until a potential new re-balancing step is executed. Some studies use Charm++ to develop and experiment with different load balance strategies, including hierarchical methods [195] or considering topology, communications and affinity [105, 106]. Our approach implements load balancing in an MPI+OmpSs-2@Cluster program, and it addresses issues of interoperability between the MPI and tasking models.

Adaptive MPI (AMPI) is an implementation of MPI that uses the load balancing capabilities of Charm++ [98, 33]. Balasubramaniam et al. also propose a library that dynamically balances MPI processes by predicting the load and migrating the data accordingly [16].

Yang et al. present a technique to use the server-client features of Asynchronous Dynamic Load Balancing (ADLB) and perform domain decomposition of irregular spatial data distributions in order to balance the application dynamically with this hybrid approach [189]. They take into account spatial join costs.

2.2.2 Hybrid programming

One of the most common approaches in this direction is to add a second level of parallelism based on shared memory. Several shared memory runtime systems implement dynamic work-sharing and work-stealing among threads to mitigate the effects of load imbalance in shared memory. Section 2.1 already mentions some of these frameworks and libraries, so here only extra remarks are added.

The Intel Threading Building Blocks [145], currently known as oneTBB, is a C++ template library that provides functions, interfaces, and classes to parallelize an application.

OpenMP is the most widely used shared memory programming model, and MPI+OpenMP, is the most widely used hybrid model for HPC applications [138]. A recent proposal, for *free agent threads* [122] improves OpenMP's malleability at the level of cores; i.e. by allowing the addition or removal of cores inside a parallel section. This approach improves the load balancing capabilities of MPI+OpenMP, but unlike OmpSs-2@Cluster, it can only balance the load independent on each node, so it cannot move work from one node to another.

Khaleghzadeh et al. consider the bi-objective optimization problem for performance and energy (e.g. on heterogeneous HPC platforms) to do workload redistribution [115]. However, Smith et al. [167], Henty [93] and Cappello et al. [45] shows that hybridizing can help improve load balance, but not in all situations. The effectiveness may depend on the code, level of imbalance, the communication patterns, and memory access patterns.

Dynamic Thread Balancing Dynamic Thread Balancing (DTB) is a library that intercepts the PMPI function calls of the MPI and shifts threads between the MPI processes residing on the same node with the OpenMP `omp_set_num_threads` call [168]. The busy waiting freed threads are put to sleep to avoid consuming compute cycles. In this regard, this approach is very similar to Dynamic Load Balancing (DLB) (see below). The algorithm needs several iterations to converge to a stable and optimal redistribution, and their approach is focused on SMPs with many Central Processor Units (CPUs).

The works done by Spiegel et al. [168] and Duran et al. [63] are similar to our approach in the use of information from previous iterations. They both aim to balance applications with two levels of parallelism by redistributing the computational power of the inner level. And they both do it at runtime without modifying the application. The first one balances MPI+OpenMP hybrid applications, and the second one balances OpenMP applications with nested parallelism. We extend these ideas to enable load balancing not just individually inside each node but across the whole application.

Asynchronous Dynamic Load Balancing ADLB is a master-slave infrastructure that works in a server-client fashion for MPI+X applications [124]. When a process requests work of a given type, ADLB can find it in the system either locally (that is, the server receiving the request has some work of that type) or remotely. Every server knows the status of work available on other servers by updating a status vector that circulates around the ring of servers. ADLB also considers memory use and balance.

FLEX-MPI is an extension to MPI that will redistribute the data to improve the load balance based on profiling information [127]. All these solutions need coarse-grained and rather persistent load imbalances to be efficient.

CHAMELEON allows fine-grained load balancing of task-parallel MPI+X applications [116]. They included "a node known to have lower energy efficiency" with a power cap and likwid-setFrequencies [91] to reduce the core frequency of a single node [116]. On the other hand for software imbalances, they 1) vary the work per rank in a dense matrix multiply and 2) use an Adaptive Mesh Refinement (AMR) application, and 3) use ADER-DG, which has load variability.

CHAMELEON is the closest approach to ours. In comparison with CHAMELEON, OmpSs-2@Cluster supports task dependencies, using the same data access specifications to describe task dependencies and data locality/copies. We execute the task in a context with the same virtual address space, simplifying the porting and debugging of programs that work on data structures with pointers. We use a small number of helpers per MPI rank in the program, organized as a sparse expander graph, to limit the amount of communication and coordination among MPI ranks. Whereas CHAMELEON is only reactive to fine-grained load imbalance, OmpSs-2@Cluster uses DLB's Lend When Idle (LeWI) module [80] to react to fine-grained load imbalance, as well as DLB's Dynamic Resource Ownership Management (DROM) module [56] to reserve cores to address coarse-grained load imbalance.

2.3 Malleability

Parallel applications can be classified into four categories:

Rigid Require a fixed allocation of processors. Once the number of processors is configured, the application cannot be executed on a smaller or larger number of processors.

Moldable Can run on a flexible number of processors. However, the allocation of processors remains fixed during the runtime of the application.

Evolving Can change the number of processors during execution. The change is triggered by the application itself.

Malleable Can change the number of processors during execution, and it is triggered by an external resource management system.

These categories respond to a scheduling perspective, considering who and when the application size is determined [72, 68, 92].

For the more dynamic classifications (evolving and malleable) several approaches have been tried for years. Based on prior work [102, 103, 104], two main reconfiguration approaches are clearly defined based on the data migration strategies:

On disk reconfiguration On-disk reconfiguration is based on the principle of saving the state of a job in a non-volatile memory device, in order to load it when required. (see: Section 2.3.1)

On memory reconfiguration Dynamic data redistribution mechanisms distribute the data, point-to-point or collectively, among processes without accessing the disk. (see: Section 2.3.2)

Fully malleable applications require some support and integration with the Resource Management System (RMS). Several studies demonstrate the benefits of scheduling algorithms which consider malleable jobs, using theoretical analysis, and simulation with job traces [73, 65, 99]. Other works have demonstrated the benefits of a malleable processor allocation technique based on a combination of moldability and folding techniques [182]. Finally, some other approaches tried adaptive scheduling policies in grids with **KOALA** multi-cluster scheduler and **DYNACO** [44] framework, the AppLeS [32] project, Satin [188] and GrADS+SRS [183].

2.3.1 Malleability with Checkpoint and Restart

The simplest approach for malleable applications is Checkpoint and Restart (C/R) which is also useful for some other purposes such as resilience and to support jobs that run for longer than the maximum time slot allowed by the RMS. The main difference between the other two use cases with malleability is that generally, they assume that the restart the parallel application will have a similar configuration than the checkpoint process (same number of nodes), which is not the case for malleability.

C/R has been used for more than 30 years for malleability purposes because it does not require reallocation or job resize support in the RMS or the MPI library. To perform a malleability operation using C/R, the application first saves its state data to the hard-drive (to perform a checkpoint), it then stops the execution and schedules another job, which restores the data state from the checkpoint and continues the computation (restart).

In order to achieve starting and stopping of the parallel applications, the total or partial state of the applications have to be checkpointed. The extensive diffusion of modern and faster Solid State Drive (SSD) in HPC facilities accompanied by better networks and more efficient parallel filesystems reduce the cost to choose this approach for malleability purposes [148]. Various studies [135, 69, 147] have surveyed several checkpointing strategies for sequential and parallel applications. Some strategies are in the kernel, while others are at the user level; but the common goal in all of them is to implement checkpoint and restore with the minimum runtime overhead for checkpointing.

Most checkpoint schemes and libraries are mainly designed for fault tolerance [27, 149] and not for malleability. Therefore they optimize Input Output (IO) and checkpoint operations to reach permanent storage while exploiting all available hardware features to reduce times, overheads and latency [61, 165, 135, 31]. A very common consequence of these optimizations is that they make it hard to recover the checkpoint with a different parallel application configuration (i.e. number of processes).

Checkpointing systems for parallel applications can be classified depending on the transparency to the user and the portability of the checkpoints.

Transparent Hides all details of checkpointing and restoration of saved states from the application. These approaches are generally not portable, e.g. DMTCP [10], CoCheck [169], NVRC [135]

Semi-transparent Generally hides most checkpoint details such as state and distributions, but they still require some modifications to the source code to specify the data or the moments to checkpoint, e.g. CLIP [51], SRS [184] and FTI [27]

Non-transparent involves the users to make modifications in their programs but are highly portable across systems, e.g. PLinda [107], DyRecT [87]

Some runtime systems already utilize a "checkpointing and restart" mechanism in order to resize the application, essentially spawning a new job every time a reassignment of resources is required [92, 98].

For malleability purposes, C/R has some advantages considering that its main drawback (IO time and latency) is being improved in modern architectures:

1. The checkpoint files are useful for multiple purposes, such as analyzing the results, checking application status or progress, tracing and porting or migrating the application to another system.
2. No support is needed from the RMS to reconfigure the application because the rest of the computation is scheduled as a new job.
3. The wallclock time and scheduling time are independent of the application and reconfiguration.
4. The approach is intuitively in the MPI paradigm.
5. Is it possible to implement C/R using simple MPI primitives without extra dependencies.
6. The application is portable across HPC facilities without special permissions or configuration.

The most basic, portable and extended approach for parallel portable C/R applications is the MPI-IO API introduced in the MPI standard since 1997 [74]. The goal of the MPI-IO interface is to provide a widely used standard for describing parallel IO operations within an MPI message-passing application [54, 89, 90].

However, there are still two disadvantages of C/R for malleability purposes. First, the applications need to be designed or modified to conditionally perform checkpoint and restart operations. Second, the checkpoint and restart files need to be formatted independently of the resource configuration and data distribution. The latter point is complex because many fault tolerance libraries, e.g. FTI [27] normally create a separate checkpoint file per process so that the data format is tightly coupled to the data distribution and the number of processes. It then becomes difficult or nearly impossible to reformat the data to restart with a different number of nodes or MPI ranks.

2.3.2 Malleability with dynamic process management

MPI applications can in principle be moldable since the introduction of Dynamic Process Management (DPM) in the MPI-2 standard. Multiple studies have attempted to provide rescalable applications [89, 90]. However, this feature has been notoriously cumbersome to use for application developers since it requires hand-coding for the redistribution of data across nodes. MPI does not provide explicit support for malleability or interaction with the resource manager, making the moldable and malleable MPI applications complex and scarce. In general, the functionality does not fit comfortably in the programming model, increasing code and software design complexity.

An interesting approach to implement malleability used by [126] in the Process Checkpointing and Migration (PCM) API simplified the data migration mixing a C/R approach with DPM. That work conceives malleability as split and merges operations supported using PCM calls added to application code. However, since MPI applications are processor-centric, and their scheme needs significant application code modification for performing data re-decomposition after resize. The main advantage of this schema is that C/R is generally simpler to implement than a complex generalized data migration schema with communications.

Cera et al. provide malleable MPI applications with dynamic CPUSETs mapping (specific to multicore machines) and dynamic MPI with the OAR resource manager [47]. This work investigates two techniques to provide malleable behavior on MPI applications and the impact of this support upon a resource manager. The second technique is more general to allow shrinking or growing using MPI process spawning primitives. Moreover, significant application programmer effort is necessary to perform data re-decomposition after resizing.

2.3.2.1 Frameworks and libraries with support for malleability

Charm++ The Charm++ allows migrating all or part of the chare objects' state to the disk using the Pack-UnPack (PUP) routines. With that, it is possible to perform checkpoint-restart strategies with some abstractions. However, the process is not totally automated, and the programmer needs to design part of the checkpoint and restart strategy. Besides that, Charm++ jobs also have some ability to shrink or expand their node footprint by dynamic migrating chares to different processors [111]. This approach is limited because in case of shrink, some residual processes are still left on the processors that are removed from the available processor pool. These residual processes carry out low-level processor-based tasks, such as forwarding messages for migrated objects to their new homes and spanning tree-based reductions. The residual processes raise severe inter-job interference and security concerns, making this approach impractical. Moreover, in this approach, for expansion, the job size is limited to the number of nodes where it was initially launched. Hence, for efficient and true resize, one needs to eliminate these residual agents.

Adaptive MPI (AMPI) combines over-decomposition and checkpoint-restart [98]. The main idea is to perform shared file-system-based checkpoint-restart and application re-launch. Its main drawback is slowness due to IO and re-launch costs. Prabhakaran et al. [150] combined AMPI with the workload manager Torque/Maui and extended the RMS to deal with malleable jobs, and they provided a communication layer between the Charm++ runtime and the Torque/Maui scheduler.

The most relevant efforts in job malleability aimed at adaptive workloads, ready to be adopted in production environments, integrate reconfiguration capabilities with an RMS which is aware of the cluster status.

ReSHAPE is a framework that includes a programming model and API, data redistribution algorithms, a runtime library, and a parallel scheduling and resource management system [170]. It is strongly centered on iterative applications, and some of its components are derived from the DQ/GEMS project [172]. The initial extension of DQ to support resizing was done in previously by Swaminathan [171].

Power-aware resource manager (PARM) uses over-provisioning, power capping, and job malleability to maximize job throughput under a strict power budget in over-provisioned facilities [160]. Regarding malleability, it mainly relies on the Charm++ runtime support.

Elastic MPI is an infrastructure and a set of API extensions for malleable execution of MPI applications based on Slurm and MPICH [53]. Using the functions provided in this API, an application is declared as malleable and the processes of the application check whether Slurm initiated a reconfiguration. MPICH has been enhanced with a new set of functions that replace and complement the standard implementation. In addition to being MPICH-dependent, As a drawback, this approach does not assist in data redistribution.

Dynamic Management of Resources (DMR) proposes an API with a mechanism to accomplish malleability on MPI applications in collaboration with Slurm [101, 103]. The work proposes a methodology for dynamic job reconfiguration driven by the programming model runtime in collaboration with the global resource manager and the library works like a communication layer between Nanos++ and Slurm that allowing moldable and malleable MPI applications. The DMR API relies on Slurm for managing and reallocating resources, and in the offload semantics of **OmpSs-1** [155] to handle processes and data redistribution automatically. Although the DMR API provides a highly usable interface, it requires a special effort to integrate the reconfiguration capabilities for irregular applications (e.g. applications implementing a consumer–producer scheme) because not every process features the same data structures.

In a later work, the same author presents a library for dynamic management of resources (**DMRlib**) [104]. Such a library exposes a minimalist set of semantics in an MPI-like syntax which eases malleability adoption for developers familiar with the MPI programming model to easily integrate malleability mechanisms in MPI applications by using the standard MPI communication routines and a reconfiguration trigger, which will determine the synchronization point for malleability actions. The job reconfiguration is encapsulated in a single call to DMRlib that abstracts the resource reallocation in Slurm, the process management (spawns and terminations), and the data redistribution among processes. Users can provide their own redistribution functions using any function implemented in the underlying MPI library. The implementation does not require any new functions or wrappers to the MPI standard.

DMRlib is the closest work to ours due to its integration with OmpSs-1, its interaction with the RMS and the use of `MPI_Comm_Spawn` to dynamically add processes. Unlike OmpSs-2@Cluster, the user must provide code to implement the data redistribution, although the process is simplified through a defined interface.

Iserte made a detailed study comparing some of these approaches, syntax and usability, including code examples and useful comments and implementation descriptions [102].

2.3.3 Other approaches

C/R strategies as explained in Section 2.3.1 attain low performance because of the cost of disk operations. Some applications go around this issue using in-memory C/R [194, 193], transferring the data among processes, point–to–point or collectively, without accessing the disk. The overhead of the reconfiguration provided by traditional C/R mechanisms compared with a dynamic redistribution of data is discussed by Lemarinier et al. [119] and Iserte [102].

EasyGrid is an EasyGrid Application Management System (AMS) aimed at adjusting automatically the scale of a running application [162]. For this reason the library provides a new set of functions that enables developers to: determine reconfiguration points, calculate the new grade of parallelism depending on the data gathered during the execution, and redistribute the data, and trigger reconfiguration.

Flex-MPI integrates three features: monitoring, load-balancing and data redistribution [127]. The User Level Failure Migration (ULFM) [119] is a non-standard MPI malleability extension combining the fault tolerance mechanism in ULFM `MPI_Comm_shrink` with the MPI-2 standard routine `MPI_Comm_spawn` to support dynamic reconfiguration.

Chapter 3

OmpSs-2@Cluster

3.1 Introduction

Combining two fundamentally different programming models is difficult to get right [158, 157]. The tasking approach of OpenMP and OmpSs offers an open, clean, and stable way to improve hardware utilization through asynchronous execution while targeting a wide range of hardware, from Shared Memory Multiprocessor Systems (SMPs), to Graphical Processor Units (GPUs), to Field-Programmable Gate Arrays (FPGAs). OmpSs-2@Cluster extends the same approach, of OmpSs-2 tasking to multiple nodes. We develop OmpSs-2@Cluster, which provides a simple path to move an OmpSs-2 program from single node to small- to medium-scale clusters. We also present the runtime techniques that allow overlapping of the construction of the distributed dependency graph, efficient concurrent enforcing of dependencies, data transfers among nodes, and task execution.

Several research groups are looking into tasks as a model for all scales ranging from single threads and accelerators to clusters of nodes, as outlined in Section 2.1. Our approach is unique, in that, in many cases, a functional multi-node version of an existing OmpSs-2 program can be obtained simply by changing the configuration file supplied to the runtime system. The meaning of the program is defined by the sequential semantics of the original program, which simplifies development and maintenance. All processes use the same virtual memory layout, which avoids address translation and allows direct use of existing data structures with pointers. Improvements beyond the first version can be made incrementally, based on observations from performance analysis. Some optimizations that are well-proven within a single node, such as task nesting to overlap task creation and execution [143] are a particular emphasis of OmpSs-2@Cluster, since they clearly have a greater impact when running across multiple nodes, due to the greater node-to-node latency and a larger total number of execution cores.

The program is executed in a distributed dataflow fashion, which is naturally asynchronous, with no risk of deadlock due to user error. In contrast, MPI+X programs often use a fork–join model due to the difficulty in overlapping computation and communication [158]. We show how well-balanced applications have similar performance to Message Passing Interface (MPI)+OpenMP on up to 16 nodes. For irregular and unbalanced applications like Cholesky factorization, we get a 2× performance improvement on 16 nodes, compared with a high-performance implementation using MPI+OpenMP tasks. All source code and examples are released open source [24].

3.2 Background

3.2.1 OmpSs-2 programming model

OmpSs-2 [23, 22, 24] is the second generation of the OmpSs programming model. It is open source and mainly used as a research platform to explore and demonstrate ideas that may be proposed for standardization in OpenMP such as the concept of data dependencies among tasks which was first proven in OmpSs. Like OpenMP, OmpSs-2 is based on directives that annotate a sequential program, and it enables parallelism in a dataflow way [144]. The model targets multi-cores and GPU/FPGA accelerators. This decomposition into tasks and data accesses is used by the source-to-source Mercurium [18] compiler to generate calls to the Nanos6 [19] runtime Application Programming Interface (API). The runtime computes task dependencies and schedules and executes tasks, respecting the implied task dependency constraints and performing data transfers and synchronizations.

OmpSs-2 differs from OpenMP in the thread-pool execution model, targeting heterogeneous architectures through native kernels, and asynchronous parallelism as the main mechanism to express concurrency. Task dependencies may be discrete (defined by start address), or regions with fragmentation [143].

At the beginning of the execution, the OmpSs-2 runtime system creates an initial pool of worker threads. The main function is wrapped in an implicit task, which is called the main task, and it is added to the queue of ready tasks. Then, one of the worker threads gets that task from the queue and starts executing it. Meanwhile, the rest of the threads wait for more ready tasks, which could be instantiated by the main task or other running tasks.

OmpSs-2 enable asynchronous parallelism using data-dependencies between the different tasks. When a new task is created, its dependencies are matched against those of existing tasks, and if found, the task becomes a successor of the corresponding tasks. Tasks are scheduled for execution as soon as all their predecessors in the graph have finished or at creation if they have no predecessors. The task that executes code to generate more tasks is the parent task of the newly created tasks, and consequently, the new tasks are their children. For this reason, the main task is the parent task of all user tasks and is at the top of the task hierarchy.

This version extends the tasking model of OmpSs and OpenMP to improve task nesting and fine-grained dependencies across nesting levels [143, 7]. The `depend` clause is extended with `weakin`, `weakout` and `weakinout` dependency types, which serve as a linking point between the dependency domains at different nesting levels without delaying task execution. They indicate that the task does not itself access the data, but its nested subtasks may do so. Any subtask that directly accesses data needs to include it in a `depend` clause in the non-weak variant. Any task that delegates accesses to a subtask must include the data in its `depend` clause in at least the weak variant.

The `taskwait` directive in OmpSs-2 has a similar semantic and description than the one in OpenMP. `taskwait` suspends the current task until all child tasks generated before the `taskwait` finalize. If the `taskwait` include dependencies the current task region is suspended only until all the child tasks with dependencies on the declared dependency region complete execution. Unlike OpenMP, OmpSs-2 tasks do not execute a `taskwait` at the end of their body because they delay the finalization of tasks, they hinder the discovery of parallelism and delay the release of all dependencies until all child tasks finish (even the dependencies not declared by any subtask).

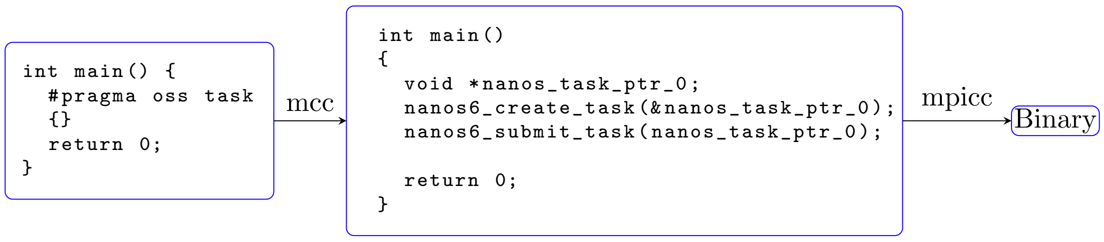
Early release is one of the distinctive features of the OmpSs-2 programming model [143]. OmpSs-2 tasks do not hold all their task accesses until all their subtasks finish. Instead, when the parent finishes, having created and submitted all its subtasks, it connects the last subtask for each access to the parent's successor. When the subtask completes, the data is passed directly to the successor task without additional synchronization via the parent.

Similarly to OpenMP, the taskloop construct specifies that the iterations of one or more associated loops will be executed in parallel using explicit tasks. The iterations are distributed across tasks generated by the construct and scheduled to be executed.

OmpSs-2 defines `task for` as a work-sharing task constructor with the ability to run concurrently on different threads, similarly to OpenMP `parallel for`. `task for` tasks do not force all threads to collaborate nor introduce any kind of barrier. The `task for` partitions the iteration space of a for-loop in chunks that will be spread among collaborator threads. Such chunks do not have task-associated overheads such as memory allocation or dependency management.

3.2.2 OmpSs-2 program build and execution

Figure 3.1 shows the schematic internal process to build any OmpSs-2 application. This process is the same for OmpSs-2@Cluster programs because the runtime library Nanos6@Cluster is the one linked with the MPI library, not the user code.



```
int main() {
  #pragma oss task
  {}
  return 0;
}
```

```
int main()
{
  void *nanos_task_ptr_0;
  nanos6_create_task(&nanos_task_ptr_0);
  nanos6_submit_task(nanos_task_ptr_0);

  return 0;
}
```

Figure 3.1: OmpSs-2@Cluster application build with Mercurium.

In Figure 3.1 the initial user source code with annotations is transformed by Mercurium into an intermediate source code with Nanos6 API calls. Any native compiler can then create the final executable from the transformed sources and link with Nanos6 and the MPI library.

Mercurium automatizes the compilation into a single step for the final user: `mcc source -o binary`. However, some options allow the user to access the intermediate files with the transformed code if needed. The binary created in Figure 3.1 can be executed in the same way as any MPI program.

3.3 OmpSs-2@Cluster programming model

OmpSs-2@Cluster extends the OmpSs-2 tasking approach, to multiple nodes. OmpSs-2@Cluster programs are executed using `mpirun` or `mpiexec` like any MPI application and the runtime will initialize the processes in the correct role automatically. Each MPI process will execute multi-threaded automatically using the same thread-pool model than OmpSs-2 on SMP explained in

Section 3.2.1. The tasks created at any nesting level can execute on any thread of any MPI process, dependencies, offloading, and data transfers are handled transparently by the runtime.

As a programming model, OmpSs-2@Cluster only has one new requirement for correctness (full dependency specification) and a few programming extensions to improve performance. The compiler only requires a minor revision to support these new features.

Table 3.1 summarizes the main features of OmpSs-2@Cluster as a programming model. The table includes some internal runtime optimizations but also features exposed to the user.

Feature	**Description**
Sequential semantics	Simplifies development, porting, and maintenance. Tasks can be defined at any nesting level and can be offloaded to any node.
Common address space	Simplifies porting of applications with complex data structures by supporting pointers and avoiding address translation.
Distributed dataflow execution	Task ordering and overlapping of data transfers with computation of another task are automated, reducing synchronizations and avoiding risk of deadlock.
Distributed memory allocation	Informs runtime that memory is only needed by subtasks, reducing synchronization and data transfers. Provides data distribution affinity hint.
Minimizing of data transfers	The `taskwait on` and `taskwait noflush` directives help minimize unnecessary data transfers.
Early, late or auto release of dependencies	A tradeoff between parallelism and overhead is exposed through control over the release of dependencies.
Cluster query API and scheduling hint	Optional ability to instruct the runtime to control detailed behavior and optimize decisions.

Table 3.1: Key features of OmpSs-2@Cluster.

Figure 3.2 shows a sliced matrix–matrix multiplication kernel using OmpSs-2@Cluster. Every task executes a matrix multiply of $ts \times dim$ where dim is the matrix dimension, and ts is a predefined task size. The figure includes some features and several optional annotations useful in the explanation below but not needed or redundant in a real matrix–matrix benchmark.

In Figure 3.2 the weak dependencies outer task is an optimization to allow subtask creation to be overlapped with task execution and namespace direct propagation (see: Section 3.6.1). In general, the program as a whole is executed in a *distributed dataflow* fashion, with data transfers and data consistency managed by the runtime system. Data location is passed through the distributed dependency graph.

3.3.1 Sequential semantics

Like OmpSs-2 on an SMP, tasks are defined by annotations to a program with *sequential semantics*, and all tasks, including offloadable tasks, can be nested and defined at any nesting level. OmpSs-2@Cluster assumes that all the tasks can be offloaded without any strong restriction. The runtime system decides dynamically what and where the tasks are offloaded, generally based on scheduling policies or constraints defined in the annotations. Tasks may be executed across all threads within the process (which is typically the whole node or a single socket), or offloaded to another process.

The example in Figure 3.2 shows how the same code can be executed in a sequential workflow and may be valid just by removing the annotations (`pragmas`).

In an OmpSs-2@Cluster, parallel application execution starts on the master process, which runs `main` as the first task. The tasks are dynamically discovered/created while the execution advances.

```
void matmul(const double *A, const double *B, double *C, int dim, int ts)
{
  int rowsPerNode = dim / nanos6_get_num_cluster_nodes();

  for(int i = 0; i < dim; i += rowsPerNode) {
    int targetNode = i / rowsPerNode;

    #pragma oss task weakin(A[i*dim; rowsPerNode*dim]) weakin(B[0; dim*dim]) \
       weakout(C[i*dim; rowsPerNode*dim]) node(targetNode) label("weakmatvec")
    {
      #pragma oss task for in(A[i*dim; rowsPerNode*dim]) in(B[0; dim*dim]) \
        out(C[i*dim; rowsPerNode*dim]) label("taskformatvec")
      for(int j = i; j < i + rowsPerNode; j += ts) {
        cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                    ts, dim, dim, 1.0, &A[j * dim], dim,
                    B, dim, 0.0, &C[j * dim], dim);
      }
    }
  }
}

int main()
{
  double *A = nanos6_dmalloc(dim * dim * sizeof(double), nanos6_equpart_distribution);
  double *B = nanos6_dmalloc(dim * dim * sizeof(double), nanos6_equpart_distribution);
  double *C = nanos6_dmalloc(dim * dim * sizeof(double), nanos6_equpart_distribution);

  (...)

  matmul(A, B, C, dim, ts);

  return 0;
}
```

Figure 3.2: OmpSs-2@Cluster dense matrix–matrix multiply, using a weak parent task to overlap task creation and execution. Every task executes a matrix multiply of $ts \times dim$ where dim is the matrix dimension and ts is the task size.

When those tasks execute, they can create more nested tasks whose dependencies are a subset of the parent's ones or memory allocated within the parent context before the sibling creation. This process continues until the task body completes the execution and the next task starts.

The example in Figure 3.2 offloads one task per node and then subdivides the work among the cores using a `task for`.

3.3.2 Full dependency specification:

In SMP systems accesses that are not needed to resolve dependencies may be omitted. It is also a common practice to specify only the dependency over the first element of an array instead of the whole array if there is no expected fragmentation or partial accesses in future tasks.

Offloadable tasks (see below) require a full dependency specification, i.e., `in`, `out`, and `inout` dependencies must specify all accesses, rather than just the constraints needed for the task ordering because the access annotations are also used to create data transfers.

This is the only reason that a valid OmpSs-2 program may not be a valid OmpSs-2@Cluster program; but it is not a new issue because a similar requirement exists for accelerators with separate memory spaces.

Figure 3.2 shows the annotations including the whole dependency regions.

3.3.3 Common address space

There is a *common address space* across cluster nodes, with data mapped to the same virtual address space on all nodes. As long as the task's accesses are described by dependencies, any data allocated on any node can be accessed at the same location on any other node. There is sufficient virtual address space on all modern 64-bit processors to support up to 65k cores with a typical 2 GB/core footprint. Almost any OmpSs-2 program can therefore be executed using OmpSs-2@Cluster and, conversely, if new features are ignored or implemented with a stub, any OmpSs-2@Cluster program is a valid OmpSs-2 program. This property minimizes porting effort and allows re-use of existing benchmarks.

The example in Figure 3.2 allocate distributed virtual memory from the common address space with the API function `nanos6_dmalloc`. The distributed and local memory concepts are explained in Section 3.5.2

3.3.4 Distributed dataflow execution

As derived from sequential semantics, OmpSs-2 provides the runtime with all the information to ensure task ordering and automate the overlapping of data transfers with computation. OmpSs-2 uses a data-oriented dependency approach in which the vertices of the dependency graph are task data accesses rather than tasks.

OmpSs-2@Cluster imposes the constraint to require complete region annotations in all the tasks. Nanos6@Cluster uses that information not only to construct the dependency graph and enforce task ordering but also to determine the data transfers that are required to execute a task on another node. The runtime starts the data transfers in the background concurrently with other tasks execution and computation, using non-blocking communications. On the other hand, most of the synchronization is managed by the runtime dependency system to not consume resources in any blocking status. This reduces synchronizations and avoids the risk of deadlock.

The tasks in the example Figure 3.2 contains the complete dependencies annotations for the two levels of parallelism. The nested tasks only contain sub-dependencies of the parent task's dependencies, and the runtime handles all the data transfers internally for the strong tasks, i.e. tasks without any weak accesses, when needed.

3.3.5 Distributed memory allocation and affinity hints

When using a distributed memory architecture, the data-oriented dependency approach becomes a fundamental characteristic in order to execute parallel applications efficiently and reduce the number of internal communications and data transfers. The memory access pattern is specific and different to every problem and a general purpose runtime should not make excessive assumptions without user hints.

The new distributed malloc, `nanos6_dmalloc`, is an alternative memory allocation primitive for large data structures manipulated on multiple nodes. The distributed memory can be released using `nanos6_dfree`. Distributed malloc has three important distinguishing characteristics, which are:

i. Since the data is intended to be manipulated by concurrent tasks on several nodes, it can be assumed that the data is not used by the enclosing task, only its subtasks or descendants, similarly to a weak dependency. In particular, the data is not copied back to the enclosing task on a `taskwait`.

ii. Since large allocations are infrequent and use significant virtual memory, it is efficient to centralize the memory allocator, as the overhead is tolerable, and it leads to more efficient use of the virtual memory.

iii. It is a convenient place to provide a data distribution hint.

Therefore, the data distribution hint is communicated to all nodes and is intended to help the scheduler improve load balance and data locality, using information from the programmer, if available. The hint does not mandate a particular data distribution, only the data affinity. Furthermore, depending on the chosen scheduling policy, the scheduler can account for the affinity and current location.

The node with the affinity for a distributed memory region is called **home node** for that region.

The main utility of the affinity hint is for scheduling purposes without requiring the user to specify a `node` clause. Some other use cases include Checkpoint and Restart (C/R) functionalities as will be explained in Section 5.3.2.2 and Section 5.4.2.1.

The example in Figure 3.2 passes the `nanos6_equpart_distribution` policy to `nanos6_dmalloc` in order to distribute the array with equal affinity portions on every node. Similar to a round-robin distribution.

In later work, not included in this thesis, Shaaban et al. [164] extends the OmpSs-2 tasking model to support an `output` (technically `inout`) of unknown memory allocated by the task. This is done using the new `auto` keyword, which allows a dataflow execution for OmpSs-2@Cluster programs involving memory allocation in offloaded tasks. This can be either normal memory allocation using `nanos6_lmalloc` or distributed memory allocation using `nanos6_dmalloc`.

3.3.6 Minimizing data transfers and early release

The main synchronization directive in the OmpSs-2 programming model is the `taskwait`. OmpSs-2@Cluster adds the `noflush` clause for taskwaits in order to separate synchronization from data dependency.

OmpSs-2@Cluster extends the behavior definition to synchronize within the task but also to transfer all the accesses in the local memory back to the task's execution node. Such behavior guarantees that the task can access the local variables (from the stack) and local memory after the `taskwait` in a more natural way. Even if the regions were written in a remote node like in the code example 3.3. The contents of the memory allocated by `nanos6_dmalloc` and weak dependencies are by default noflush so that tasks can wait for their child tasks without copying data that is not needed.

Figure 3.3 shows the basic use of the default `taskwait` mechanism. After the taskwait, the code can access all the stack variables in spite of their initialization having taken place on a different

```
// taskwait
#pragma oss task node(0) label("parent")
{
  int a, b;

  #pragma oss task out(a) node(1) label("A")
  {
    a = 1;
  }

  #pragma oss task out(b) node(2) label("B")
  {
    b = 1;
  }

  #pragma oss taskwait

    printf("a = %d; b = %d\n", a, b);
}
```

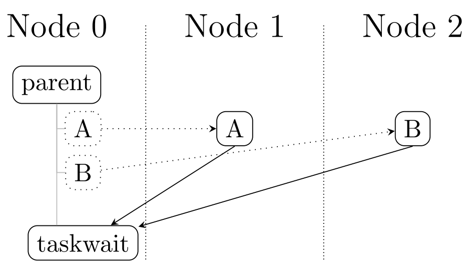


Figure 3.3: Code example using a `taskwait` to synchronize and update local variables within a task. The runtime performs one copy-back transfer for every local variable located or touched in a remote node.

node. In this case, the nested tasks require the `node` clause because the default scheduler is based on locality and will avoid offloading tasks with only local accesses to remote nodes.

While this behavior is common and consistent with the SMP programming model, it has some issues when applied to distributed memory system because any `taskwait` may create sometimes undesired data transfers.

For this reason, OmpSs-2@Cluster defines the other two synchronization alternatives in order to provide more flexibility to the programmer:

taskwait on When only a subset of locally-allocated data is needed by the enclosing task, the dependency and data transfer can be expressed using `taskwait on`. This variant reduces the synchronization contention to the tasks accessing some locally-allocated data. This is the more fine-grain synchronization method because it reduces the data transfers by coping back only the regions it applies to.

Using `taskwait on` instead of `taskwait` for the example in Figure 3.3 may save some communications by sacrificing the possibility of accessing the other variables after the taskwait.

taskwait noflush The last synchronization method is a small variation of the original method that only waits for execution but does not perform any copy. The synchronization extends to all subtasks in the scope but saves the latency of the copy-back. The `noflush` variant is also useful for timing parts of the execution.

In Figure 3.3 the `taskwait noflush` will not perform any data transfer, and the variables can not be accessed after the synchronization for reading purposes.

Early, late or auto release of dependencies On a single node, early release is beneficial because it allows tasks to begin earlier (see: Section 3.2.1). In Figure 3.4 task AB creates two subtasks, $A0$ and $B0$. Task $A0$ completes first, at which point access to `a` is released to the successor task $A1$, and task $A1$ becomes ready for execution. This happens before subtask $B0$ completes,

```
#pragma oss task out(a) out(b) label("AB")
{
  #pragma oss task out(a) label("A0")
  {  // A short task
    a = 1;
  }

  #pragma oss task out(b) label("B0")
  {   // Some long work
    b = 1;
  }
}
#pragma oss task out(a) label("A1")
{
  a = 1;
}
```

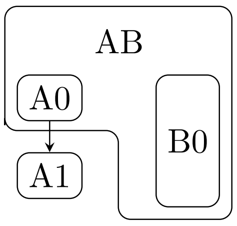


Figure 3.4: Example to illustrate early release.

at which point AB as a whole can be deleted. Early release is beneficial when the subtasks are unbalanced or have different execution times. However, early release has some drawbacks in SMP when the task granularity is very fine because it implies more stress in the dependency system. This problem is exacerbated in OmpSs-2@Cluster where a release may imply inter-node communication.

If the subtasks finish at roughly the same time as each other, compared with the overhead, it may be worth inhibiting early release. This is a problem-specific scenario for which it is difficult for the runtime to make the decision. The alternative, late release, can be forced using the OmpSs-2 `wait` clause, which adds an implicit `taskwait` after the completion of the task body. The `wait` clause is better than adding an explicit `taskwait` inside the task because it happens after the release of the stack. An explicit `taskwait` has the benefit that subtasks may have accesses on the stack, but the stack needs to stay allocated until all subtasks are complete. Parent tasks, especially those with weak dependencies, can execute and complete well before the subtasks that do the work are executed. So the memory needed by all the parent task stacks may be surprisingly high.

Autowait release is a third approach that mixes early and late release in a more elaborated schema. Offloaded tasks, by default, have an early release to successors on the same node, but all other dependencies have late release.

Figure 3.5 shows an example that illustrates the auto-release behavior. In the example the `init` tasks are all executed on the master node, but the “consumer” tasks are distributed across two nodes. The “consumer” scheduled to execute on the master node receives the early release of data accesses, while the “consumer”s on different ranks wait for the late release in order to reduce node-to-node communications.

We fully implemented early, late and auto-release of dependencies in Nanos6. We also added the `nowait` clause to the Mercurium compiler and Nanos6 API, which enables full early release even for offloaded tasks. There is also a runtime configuration variable that disables autowait.

3.3.7 Cluster query API

Nanos6 as a runtime library collects information about the system and environment. OmpSs-2@Cluster extends the API to provide information about the nodes and the runtime. In MPI applications, some of this information could be obtained through MPI functions; but OmpSs-2@Cluster

```
int *a = lmalloc(4 * sizeof(int))

#pragma oss task out(a[0;4]) node(1) label("init")
{
  for (int i = 0; i < 4; ++i) {
    #pragma oss task out(a[i]) label("I")
    initialize(a[i]);
  }
}

#pragma oss task weakin(a[0;2]) node(0) label("Weak0")
{
  #pragma oss task in(a[i]) node(0) label("A0")
  some_function(a[0]);

  #pragma oss task in(a[i]) node(0) label("A1")
  some_function(a[1]);
}

#pragma oss task weakin(a[2;2]) node(1) label("Weak1")
{
  #pragma oss task in(a[2]) node(1) label("A2")
  some_function(a[2]);

  #pragma oss task in(a[3]) node(1) label("A3")
  some_function(a[3]);
}
```

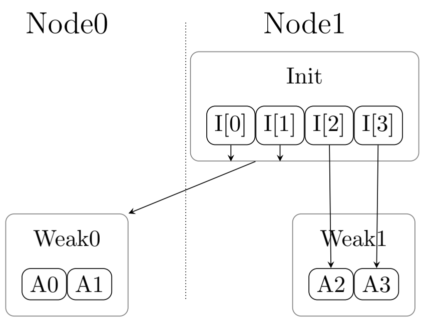


Figure 3.5: Example to illustrate auto release behavior.

uses a different internal communicator that can change on some conditions. Directly calling MPI with the default `MPI_COMM_WORLD` may return incorrect information.

Memory allocation functions are also API functions because they should use the common address space and may initiate internal communications to synchronize.

The example in Figure 3.2 uses two of the most common API functions: `nanos6_dmalloc` for memory allocation and `nanos6_get_num_cluster_nodes`, which gets the total number of MPI processes. The same example also uses the `nanos6_equpart_distribution` defined in the user API. The access to the API function does not require any special linking or header file because Mercurium automatically includes the `nanos6.h` header file, and it links with the Nanos6 runtime system.

3.3.8 Scheduling hint

The runtime contains multiple scheduling policies, which are implemented as internal modules. The user can define the default scheduling policy for the application with a configuration variable; but sometimes, this is not enough.

Some irregular applications require more precise control of the scheduling policies in portions of the code or in some specific tasks. The user may need to do that to create data distributions, not available in the allocation policies or to reduce data transfers for the rest of the execution if the programmer has extra knowledge about the memory access patterns.

OmpSs-2@Cluster adds the `node` clause to the task annotation for that purpose, as shown in the examples in Figure 3.3 and Figure 3.5. The node clause also accepts scheduling policies instead of node numbers to allow the use of different schedulers in the same application independently of the default scheduling policy. Some of the valid policy values are:

`nanos6_cluster_locality` : Execute the task on the node that is the current location for the greatest number of input/output bytes.

`nanos6_cluster_home` : Execute the task on the node that is the home node (see: Section 3.3.5) for the greatest number of input/output bytes.

`nanos6_cluster_random` : Execute the task on a random node. This policy has sometimes helped to expose bugs in the runtime.

`nanos6_cluster_no_offload` : Do not offload the task. This policy can be useful in limiting offloading and communications.

`nanos6_cluster_balance` : Work stealing scheduler that balances data locality and loads.

`nanos6_cluster_no_hint` : Use the default scheduling policy. This is normally not needed but useful to simplify giving a node hint programmatically.

3.4 Optimizing OmpSs-2@Cluster performance

As explained in Section 3.3, OmpSs-2@Cluster adds minimal modifications to the OmpSs-2 specifications. The sequential semantics (Section 3.3.1) in the programming model generally simplifies application development and portability and the Nanos6@Cluster runtime automates many optimizations to reduce communications, latency and data transfers (Section 3.6). Nevertheless, the developer should follow the below guidelines in order to obtain good performance:

Grouping of work Dividing the work into a sufficient number of tasks to keep all cores on all nodes busy may require a single execution thread to create a large number of tasks. Depending on the granularity of these tasks, the runtime overhead (for task creation, submission, dependency operations and offloading) may become significant. This overhead can already be an issue when using OmpSs-2 on an SMP, but it can be much worse for OmpSs-2@Cluster given the larger total number of cores. The overhead and latency of MPI communications for task offloading and management may also become significant. For this reason, it is advisable for the program to use two levels of parallelism: one across the nodes and one across the cores. Two levels of parallelism allow work to be created concurrently by multiple tasks, and since only the top-level tasks can be offloaded, fewer tasks require offloading. The benefit is reduced overhead and improved scalability.

Weak tasks A task can only be offloaded once it becomes ready. When using two levels of parallelism for the **grouping of work**, the top-level (offloaded) tasks create subtasks to perform computation; they do not access any of the data regions themselves. The accesses of the top-level tasks can therefore be weak accesses, which prevents these accesses from enforcing ordering among the top-level tasks. This allows top-level tasks to be offloaded well in advance, moving the offloading overhead off of the critical path of execution. Since multiple subtasks from different offloaded tasks will likely exist at the same time at a remote node, it becomes possible for the runtime to directly pass satisfiability among these tasks without involving the parent (see the "namespace" optimization in Section 3.6.1).

Taskloop The runtime distributes taskloops across all cores on all nodes, automatically applying two levels of parallelism for **grouping of work** and also using **weak tasks** at the top level. It is, therefore, beneficial to express the computation as a taskloop when possible. Taskloops that should only be distributed among the cores on a node should be marked as non-offloadable as described in Section 3.3.8.

Node clause The current scheduler is not always able to fully optimize load balance and data locality. A great body of work exists on scheduling policies, which can be incorporated into the runtime as future work (see Section 6.2.7). In the meantime, if the optimal assignment of tasks to a node is known to the programmer, then the `node` clause can be used to improve performance.

3.5 Nanos6 runtime implementation

3.5.1 Overview

All the OmpSs-2@Cluster processes contain an instance of the Nanos6 runtime, as shown in Figure 3.6.

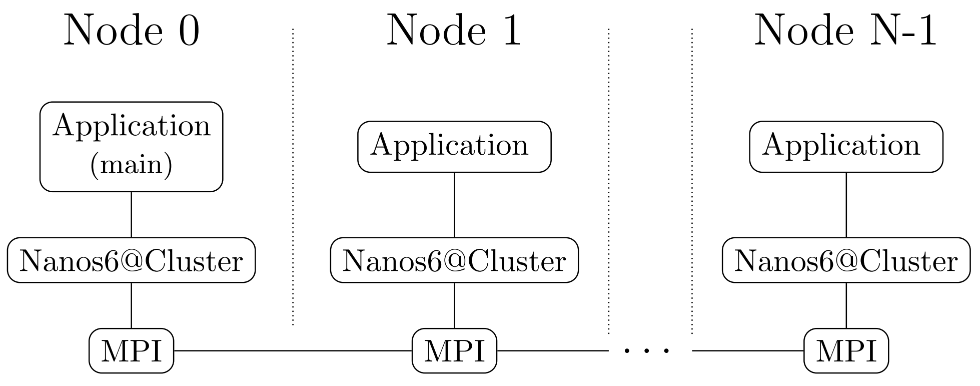


Figure 3.6: OmpSs-2@Cluster architecture. Each node is a peer, except that Node 0 runs the `main` task and performs distributed memory allocations.

There is one instance of the runtime system per node (or per socket). Each instance of the runtime schedules tasks across its cores in exactly the same way as Nanos6 on an SMP. In addition, they coordinate to execute tasks among all nodes. All nodes are peers; the only distinctions among them are that (a) Node 0 (master) executes the body of `main` as the first task, (b) Node 0 manages runtime operations that require internal collective synchronization, e.g. `nanos6_dmalloc` [1] and `nanos6_resize`, and (c) Node 0 initiates the shutdown process when `main` returns.

The processes communicate via point-to-point MPI, with a dedicated thread on each node to handle MPI control messages (see Section 3.6.4). Nanos6 uses an MPI communicator initialized as a copy of the `MPI_COMM_WORLD` at the beginning of the execution. Communication calls and synchronization are not directly exposed to the user; the runtime manages and controls all communication,

[1] nanos6_dmalloc can be called on any node, but the actual memory allocation is always done on node 0.

optimizing and minimizing it where possible. An OmpSs-2@Cluster program must not perform MPI calls in the user code. However, an extension to support MPI+OmpSs-2@Cluster is presented in Chapter 4.

On initialization, the runtime sets up the virtual address space (Section 3.5.2). The whole program is a hierarchy of tasks with dependencies, and its execution is conceptually separated into task offloading (Section 3.5.3), building the distributed dependency graph (Section 3.5.4), performing data transfers (Section 3.5.5), and executing tasks. All, of course, happen for multiple tasks concurrently.

3.5.2 Memory management

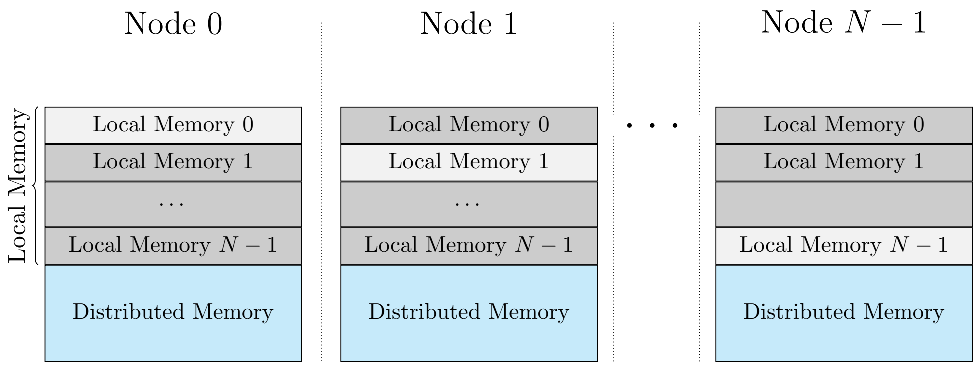


Figure 3.7: Runtime virtual memory map, which is common to all processes on all nodes.

During runtime initialization, all processes coordinate to map a common virtual memory region at the same address on all nodes. To do this, every process collects a list of the currently unused regions in its virtual address space and sends it to Node 0. Node 0 then chooses the largest contiguous region that is free on all nodes and broadcasts its starting address and size to all nodes. All processes map this same region into their address space using `mmap`.

The common virtual memory region is partitioned in the same way on all nodes, as shown in Figure 3.7. The *Local Memory* region is subdivided into one subregion per node. Each node controls memory allocation and freeing within its subregion so that it can allocate stacks and user data without coordination with other nodes. The *Distributed Memory* region is used for memory allocation via `nanos6_dmalloc` and `nanos6_dfree`, and it is managed by node 0.

The common address space allows any task executing on any node to access data from any of the regions in Figure 3.7, without address translation. Only virtual memory is mapped at the beginning of the execution. Physical memory will be allocated on demand when accessed by a task that executes on the node. The runtime does not currently release unused physical memory, as doing so has not proved necessary for our benchmarks so far. It is anticipated, however, that the runtime will be updated at some point to release unused physical memory using `madvise`.

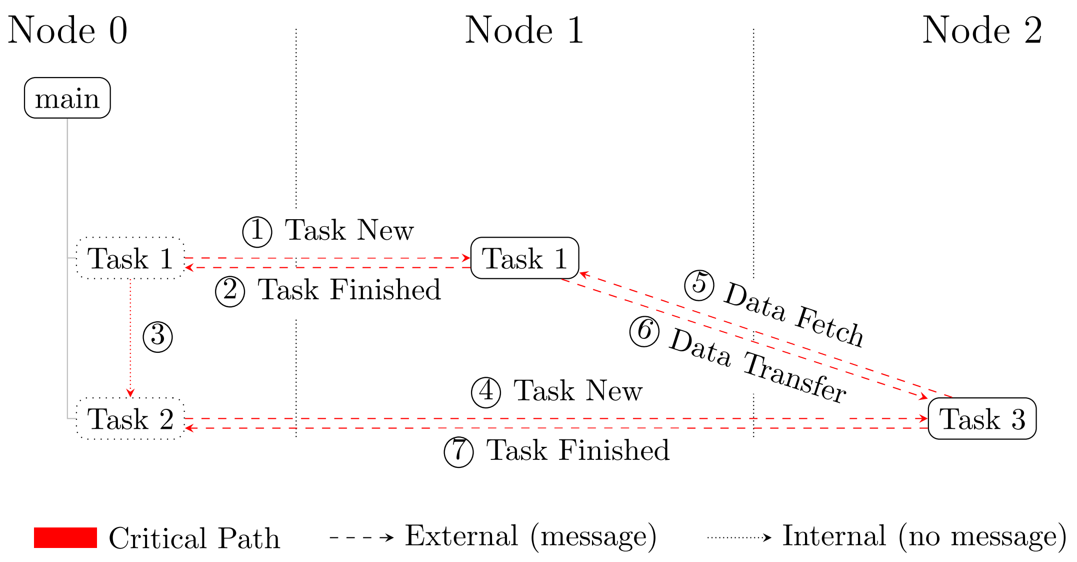
As explained in Section 3.3.3, modern 64-bit processors typically have a 48-bit virtual address space. With a typical 2 GB/core footprint and a 47-bit virtual address space available to the user space, it should be possible to support up to $2^{47}/2^{31} = 2^{16}$ cores, which is about 1,300 HPC nodes. In practice, the virtual address space will be fragmented into pieces that are all smaller than 2^{47}

bytes, but the potential number of nodes is still much larger than the goal of scaling to 16 to 32 nodes.

3.5.3 Task offloading

```
int main()
{
  int a = 0;

  #pragma oss task out(a) node(1) label("Task1")
  { }

  #pragma oss task inout(a) node(2) label("Task2")
  { }
}
```

(a) Code example



(b) Representation of task execution

Figure 3.8: Example task offloading and execution

Figure 3.8 shows a simple example with two offloaded tasks. The source code is shown in Figure 3.8a and the resulting execution is illustrated in Figure 3.8b. All tasks are created by their parent task on the node that executes the parent. In this example, Task 1 and Task 2 are both initially created on Node 0.

When a task becomes ready, the scheduler decides which node should execute the task. If the task is executed locally, it is passed to the local SMP scheduler in the same way as any OmpSs-2@SMP task. Otherwise, the task is offloaded to another node. All nodes can offload tasks to any node, including to Node 0.

If the task is offloaded, the *creation node*, on which the task was first created (Node 0), sends a Task New message to the *executor node*, on which the task will execute (Node 1 for Task 1 and Node 2 for Task 2). This message, shown as Step ① and Step ④, contains all information needed to recreate the task, including the addresses and sizes of all its accesses. The body of the original task is disabled so that the user code is not executed on the creation node. When the executor node receives the Task New message, it creates a copy of the task, known as a *proxy task*, which will execute the task body. Once created, the proxy task is submitted and passed to the executor's SMP scheduler. When the task completes execution, the executor node sends Data Release messages to release all of its data accesses, and it sends a Task Finished message to allow the task itself to be cleaned up. These messages are often all combined into a single message, Step ② and Step ⑦.

In either case, whether executed locally or offloaded, the data regions required by the task may not yet be present when the task is passed to the SMP scheduler. Data transfers, which are visible in the image, are explained in Section 3.5.5.

3.5.4 Distributed dependency graph

As described in Section 3.5.3, OmpSs-2@Cluster only offloads ready tasks. It may be, however, that an offloaded task has weak access on a data region, so it is ready immediately, but it will create subtasks with strong accesses, and these subtasks will have to wait for the data to become available. We still wish to offload the weak parent task when it becomes ready since doing so allows the parent to be offloaded and to create all its subtasks well in advance and off of the critical path.

The availability of data is indicated using additional messages known as Satisfiability messages. A Satisfiability message is sent by the parent task to its child whenever one of its data regions becomes satisfied. The message indicates whether the region is read satisfied (able to be read) or write satisfied (able to be written). When the access becomes read satisfied, the data's location is also sent. Any satisfiability information and data locations that are known when the task is offloaded are included in the Task New message. In the example shown in Figure 3.8, all satisfiability information is included in the Task New messages because the tasks have strong accesses, so when the tasks become ready, all of their accesses must already be fully satisfied. There are no separate Satisfiability messages.

3.5.5 Data transfers

When a task is scheduled to execute, the latest version of one or more of its data access regions may still be located on another node, in which case one or more data transfers will be needed before the task can execute. When the SMP scheduler schedules the task to execute, the runtime checks whether any data must be copied from any other node or whether there is an already-initiated but not complete data copy (see Section 3.6.2). In the first case, the runtime starts the copy process, and in either case, the runtime will reschedule the task once the copy finalizes. Even non-offloaded tasks on Node 0 may require data transfers since the latest data may be located on another node.

Data transfers may be configured to be either lazy or eager:

Lazy data transfers are only initiated when the task is scheduled to execute. In this case, a Data Fetch message is sent to the node that has the latest version of the data, and the task is blocked until all necessary data has been received. The example in Figure 3.8 has lazy data transfers, so Node 2 sends a Data Fetch message (Step ⑤) to Node 1 and Node 1 responds with an MPI Data Transfer (Step ⑥) containing the actual data.

Eager data transfers initiate the data transfers at the earliest moment, even for weak accesses, which is when Satisfiability is sent. The Satisfiability message includes the MPI tag for each eager data send. This is generally a good optimization, but it may happen that part of the weak accesses are not used by any strong subtask, and the data transfer is unnecessary. This situation happens in Cholesky factorization due to the pattern of the data accesses of the `dgemm` tasks.

When all subtasks are created in advance (i.e. the weak offloaded task does not execute any expensive computation itself or does not have any intermediate taskwait), this situation can be detected when a task with weak accesses completes, at which point all subtasks have been created, but no subtask accesses all or part of the weak access. In this case, a No Eager Send message is sent to the predecessor. In all cases, when a task completes, the data is not copied back to the parent (write-back) unless needed by a successor task or taskwait. The latest version of the data remains at the execution node until needed.

An alternative would be to exploit an existing Software Distributed Shared Memory (SDSM) library. Doing so would be simpler to implement than the approach using explicit data transfers. The two main disadvantages are that (a) SDSM generally initiates the data transfer when the data is needed by a task, at which point the core is occupied by the task, whereas the runtime data transfers are initiated before the task is scheduled to execute, and (b) SDSM generally works at a page-level granularity. Two tasks executing concurrently on different nodes may access disjoint memory regions involving the same memory page. This problem is known as false sharing and can dramatically impact performance.

3.6 Runtime optimizations

This section describes the runtime optimizations that were implemented. None of the optimizations change the programmer's model, although some only provide benefit for certain styles of program; e.g. the intra-node (namespace) and inter-node propagation only benefits offloaded weak tasks (see Section 3.4 on performance optimization). Apart from the inter-node passing of satisfiabilities (Section 3.6.6), which was not sufficiently useful to compensate for its complexity, quantitative results showing the benefit of each optimization are given in Section 3.9.6.

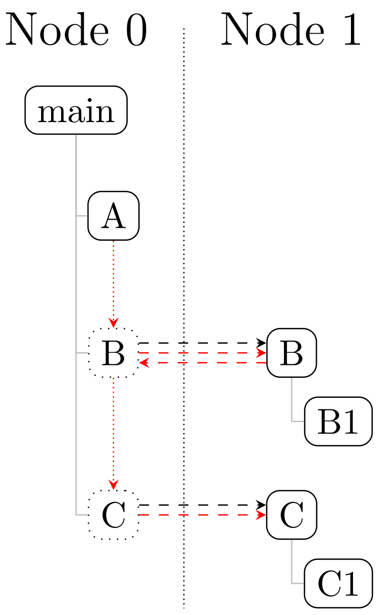
3.6.1 Intra-node propagation (namespace)

The initial OmpSs-2@Cluster implementation offloaded every task independently. This scheme caused excessive communication via the parent node. An example is shown in Figure 3.9. Figure 3.9a shows the source code and Figure 3.9c illustrates the baseline execution. Black arrows represent Task New messages, and red arrows show the propagation path followed by the access. Tasks B and C are both offloaded from Node 0 to Node 1. Nevertheless, when B1 completes, Node 1 sends a DataRelease message to Node 0, which in turn sends a Satisfiability message back to Node 1.

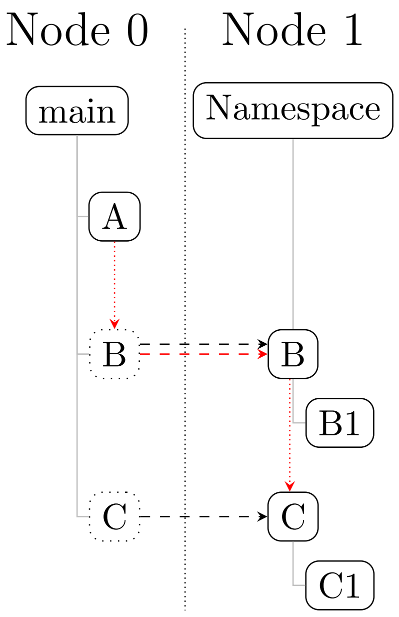
Figure 3.9c shows the effect of the "namespace" optimization, which optimizes the passing of satisfiability information on the same node. In this case, satisfiability is passed directly between B1 and C1 via their parents B and C, all of which occur on the same node, Node 1. Since there is no communication among the nodes, the latency is much lower.

```
int main()
{
  int a = 0;

  #pragma oss task out(a) node(nanos6_no_offload) label("A")
  { a = 1; }

  #pragma oss task weakinout(a) node(1) label("B")
  {
    #pragma oss task inout(a) node(nanos6_no_offload) label("B1")
    { a++; }
  }

  #pragma oss task weakinout(a) node(1) label("C")
  {
    #pragma oss task inout(a) node(nanos6_no_offload) label("C1")
    { a++; }
  }
}
```

(a) Simple code example with multi-node propagation of dependencies.



(b) Without namespace



(c) With namespace

Figure 3.9: Differences in access propagation with namespace. Red arrows represent the access propagation path.

An important consideration is how to ensure that satisfiability is propagated remotely in precisely the situations in which it is valid, taking account of potential race conditions. This is done using a global ID for each task, which is different among all tasks in the program, irrespective of the node on which the task was created. The global ID is an atomic 64-bit counter, which is initialized to contain the node ID in the top bits.

Each access identifies the task ID of the sibling task that last wrote to the memory region. This information is determined when the task is created and inserted into the dependency system. This information is sent to the executor task when the task is offloaded. If this global ID matches that of the last task offloaded to this node, then direct propagation of dependencies between these two tasks is enabled. Direct propagation is also enabled if the last task offloaded to the node only reads the data (an `in` access) and the global ID of its previous writer matches. Direct propagation is therefore not enabled if either (a) the region is modified by an intervening task executed on a different node or (b) a race condition causes tasks to be received and submitted out of order. In either case, the normal mechanism, involving DataRelease and Satisfiability messages, will be used.

Write-after-read accesses, i.e. `inout` following multiple `in` accesses, requires some synchronization to ensure that all concurrent reads complete before the task of performing the write operation can begin. Since concurrently reading tasks may be scheduled on multiple nodes, this synchronization is done explicitly via the *creation node*, being the node on which the parent task executes. Satisfiability is passed to the `inout` access once all the `in` accesses (one per node) have released the access back.

3.6.1.1 Namespace implementation

Each node (including Node 0), creates a single *namespace* task, which is the implied common parent of all tasks offloaded to that node. There is a namespace task on all nodes, including Node 0. Whenever a task receives a Task New message, the message is placed in a queue, and the namespace task is unblocked. When the namespace executes, it creates the offloaded task as a subtask in the normal way. The only special consideration is that the dependency system is aware that the child is an offloaded task, so propagation from its predecessor is only enabled if the global IDs match as described above. Once all pending children have been created, the namespace task is unblocked.

The namespace also process other messages like resize and finalization messages. These messages have order constraints and require that all the previous tasks finalize before being handled. Using the namespace task to handle them simplifies the implementation by not requiring condition variables, extra locking or message reordering to ensure correctness.

3.6.2 Dependency writeID and pending transfer

When tasks have large read-only dependency regions, it is common to have multiple copies of the same data in multiple executor nodes. Not keeping track of the data already existing in a node implies the execution of multiple redundant copies. This issue may affect multiple High-Performance Computing (HPC) applications that use transformation matrices, boundary conditions, experimental results, and so on. In general, any initialized data that is not intended to be updated frequently during the application execution.

As mentioned before: A key design choice of OmpSs-2@Cluster is that no cluster node builds the whole computation graph of the application. This complicates the region tracking because a node cannot be sure of how the data was updated since the last time it was copied. The versioning approach becomes complicated when there are not nesting or offload restrictions.

OmpSs-2@Cluster uses a simple 64-bit version number known as the WriteID to reduce redundant data transfers without doing strict tracking of all locations. The WriteID is related to the global task ID described in Section 3.6.1, but it is not the same. As described above, the global task ID is used to avoid communication when satisfiabilities are passed between offloaded tasks on the same node. It is calculated when the task is created and passed to the successor when it is created. The global ID only relates dependencies among sibling tasks, which is sufficient to avoid communication with Node 0 at the top level of the task hierarchy. In contrast, the WriteID is calculated when the task is scheduled and passed to the successor when it becomes read satisfied. It, therefore, relates dependencies among tasks at any level of the hierarchy. As such, it is able to avoid data transfers even among non-sibling tasks.

```c
int main()
{
  int *a = nanos6_dmalloc(N * sizeof(int));

  #pragma oss task out(a[0;N]) node(nanos6_no_offload) label("A")
  initialize(a)

  #pragma oss task in(a[0;N]) node(1) label("B")
  read(a)

  #pragma oss task in(a[0;N]) node(1) label("C")
  read(a)
}
```

Figure 3.10: Simple code for multiple read-only accesses on the same node.

Figure 3.10 shows a simple code where a task in node zero initializes some data, and two other tasks read it but execute on the same remote node.

In this example, when Task A finalizes, the access propagates directly to read-only tasks B and C. The location set to the region will be the master node because it is the current known data location and the intermediate task (B) is also read-only access.

When the two Task New messages arrive at the remote node, the location information will be the same, but the tasks are considered independent, and the runtime is unable to assume some association between their dependencies. The first task (i.e. B) will create a copy step for the data, and eventually, the transfer will be executed in the future with asynchronous communication. When the transfer completes, then B will register the region in the WriteID cache and future tasks willing to access the same version of the data can check before starting a new copy.

There is a corner case when C is generated early after B, and the copy is not finalized yet. OmpSs-2@Cluster considers this possibility because it was observed in some benchmarks. For that reason, the new task C also checks the list of pending transfers in order to detect if another task has already started an unfinished transfer for the same version of the data region. If that is the case, then C adds a hook to be notified when the transfer completes.

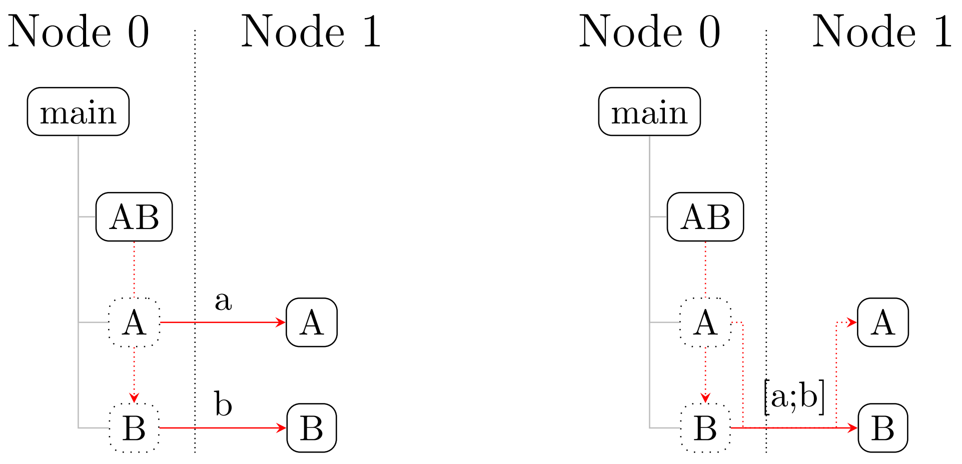
WriteID supports regions with fragmentation in the same way as the fragmented region dependency system. Due to concurrent access for different tasks, the tree requires protection with a lock. To reduce the amount of lock contention, there are some trees (currently 512) corresponding to different hashes of the WriteID. Most of the time , blocks with different writeIDs can be updated without locking contention.

3.6.3 Message aggregation

When a task finishes in OmpSs-2 releases accesses individually in the dependency system propagation to solve load imbalance and increase parallelism (see: Section 3.3.6). In OmpSs-2@Cluster that behavior may produce an excessive number messages between nodes; a problem exacerbated when the offloaded task has multiple irregular accesses and/or creates subtasks that fragment all of them locally or in different node locations.

```
int main()
{
  int a = 0, b = 0;

  #pragma oss task out(a) out(b) node(0) label("AB")
  { a = b = 1; }

  #pragma oss task weakin(a) node(1) label("A")
  {
  }

  #pragma oss task weakin(b) node(1) label("B")
  {
  }
}
```

(a) Simple code example where message grouping reduce communications.



(b) No message grouping. **(c) With message grouping.**

Figure 3.11: Execution with and without message grouping.

OmpSs-2@Cluster tries to reduce the severity of the problem in two ways already mentioned: with `auto` release and namespace propagation. But sometimes, data and tasks distribution inhibits the namespace propagation, and the problem is unsuitable for the `auto` release or the `wait` clause.

The current OmpSs-2@Cluster approach currently inhibits the immediate send of some messages when propagating in the dependency system and sends them at the end of the propagation in groups per node. This approach is completely compatible with the others with the extra benefit is that the target node receives more information in a single message as well and knows that they are related.

Figure 3.11 shows a simple example where message grouping takes place. Tasks A and B have only weak in dependencies on two independent variables a and b; so they are offloaded in advance while AB executes. The outputs (out dependencies) can propagate to the successors immediately when AB finalizes because A and B are totally independent from the runtime point of view. The accesses can propagate directly from AB to A and B. Without message grouping, Node 0 sends two independent satisfiability messages to the individual proxy tasks A and B and those messages are processed by Node 1 also individually.

The overhead introduced by the multiple MPI calls for tiny messages and the sequential processing on the receiving side introduce undesired overhead proportional to the number of fragments and successor tasks.

In more complex scenarios like nesting tasks and propagating release accesses and satisfiabilities through multiple nodes, the message grouping reduces the number of MPI calls by several orders of magnitude.

3.6.4 Dedicated leader thread

Complementary to reducing the number of MPI communications it is possible to improve performance by reducing delays and latency while sending, handling and replying to messages.

OmpSs-2@Cluster uses MPI non-blocking point-to-point communications during the application execution. Normally the messages are sent (`MPI_Isend`) and the associated request (`MPI_Request`) is enqueued in a list to check later. A separate polling service checks the requests and releases when the communications are complete. It also executes some completion hooks when needed (see: Section 3.6.2).

On the receiving part, there is another polling service checking (`MPI_Iprobe`) for incoming messages often. When a new message is arriving, the handler checks the size (MPI_Get_count), allocates memory and receives it (MPI_Recv).

The polling services infrastructures are part of the Nanos6@SMP infrastructure to execute different functions frequently without oversubscribing threads. Nanos6 contains two kinds of polling services: opportunistic and task based services. Either of them can execute only when there are available cores.

When a node is highly loaded with computation, the polling services may have to wait for a significant time before a core becomes free. In OmpSs-2@Cluster, these delays may become critical because a single node may stop the execution of parallel work on several remote nodes just by not replying to requests in a timely way. For example, in the matvec benchmark, the master node offloads all the work early , and it immediately starts doing its own computation. When the remote nodes fetch their tasks' input data, the master node may already have created enough work to occupy all its cores. If so, it will not respond until it has finished all computations in the iteration. This will delay the execution of the work on the other nodes until the master has finished working, which, depending on the exact timings, may effectively double the execution time per iteration (see Section 3.9.6).

To solve this OmpSs-2@Cluster reserves one core exclusively for the leader thread in the initialization. The leader thread will then execute the polling services reliably. For machines with fewer cores, it is possible to disable this behavior in order to use all the cores for computation, but the such decision is left to the opinion of the user.

3.6.5 Message handler helper tasks

Handling messages is a common source of inefficiency in OmpSs-2@Cluster. Despite all the previous optimizations the runtime system requires an important number of communications and control messages that sometimes become a bottleneck. As explained in Section 3.6.4, we can divide the message receiving into two basic steps: (1) MPI transfer and receive, (2) Nanos6 handle and react accordingly.

However, it is common for there to be a high flow of messages arriving to a rank, idle cores, and tasks created but not ready, waiting for some accesses locations or transfers. Some of those messages may contain information needed by the non-ready tasks in that node (satisfiability) or may request data from ready tasks in remote nodes (fetch data).

There are some messages that may be handled without order restrictions. These messages include satisfiability and data fetch; two of the most frequent ones. The handling of those messages may be slow in some cases because Satisfiability messages can create new communications, interaction with the dependency system and tasks scheduling and data fetch messages may start big data transfers.

OmpSs-2@Cluster faces this issue by adding helper polling tasks to the message handler. For this purpose, Nanos6 has a polling service infrastructure relying on internal task creation and blocking. Unlike threads, when a task blocks, it stops the execution by releasing the resources (core) to the runtime. To unblock a task, the runtime only needs to reschedule it as any other task, which is optimal to avoid over-subscription as well.

With this approach, the message handler polling service prioritizes the MPI message receive part to advance the deliveries and increase the message arrival throughput. At the same time, if the rank is using all the resources in computation, the workers are not interrupted to execute the polling task, but the leader thread (see: Section 3.6.4) will still process and handle the incoming messages sequentially.

Moreover, this enables to process and advance other operations like accesses propagation, message grouping or task creation from messages already received when one of the helper tasks (or the leader thread) block the MPI library or the network performing big transfers. The number of concurrent helper tasks needed or useful is different and strongly depends on the application (average number of messages over time, tasks granularity, load balance). For that reason OmpSs-2@Cluster allows the user to control that with a configuration variable.

3.6.6 Inter-node passing of satisfiabilities

Section 3.6.1 explained the "namespace" optimization, which reduced communication on the critical path to and from the creation node whenever two consecutive tasks execute on the same node. The same idea can be applied when the tasks are offloaded to different nodes. Unfortunately, the initial results were not promising, and the implementation is still at an experimental stage. The implementation is not yet on the main development branch , so this optimization is not included in the evaluation.

Figure 3.12 illustrates an example. Task 2 and Task 3 are offloaded to Node 1 and Node 2, respectively. When Task 3 is offloaded to Node 1, the global ID of its predecessor Task 2 is known. Moreover, even though both are weak tasks, they are almost always scheduled in order, so when Task 3 is scheduled; it is known on which node Task 2 is executed. This information is provided in Task 3's Task New message, along with the global ID of Task 2. Since Task 2 was executed on a different node, Node 2 sends a new Remote Connect message to Node 1.

```
int main()
{
  int a = 0;

  #pragma oss task weakout(a) node(1) label("Task1")
  { }

  #pragma oss task weakinout(a) node(1) label("Task2")
  { }

  #pragma oss task weakinout(a) node(2) label("Task3")
  { }
}
```

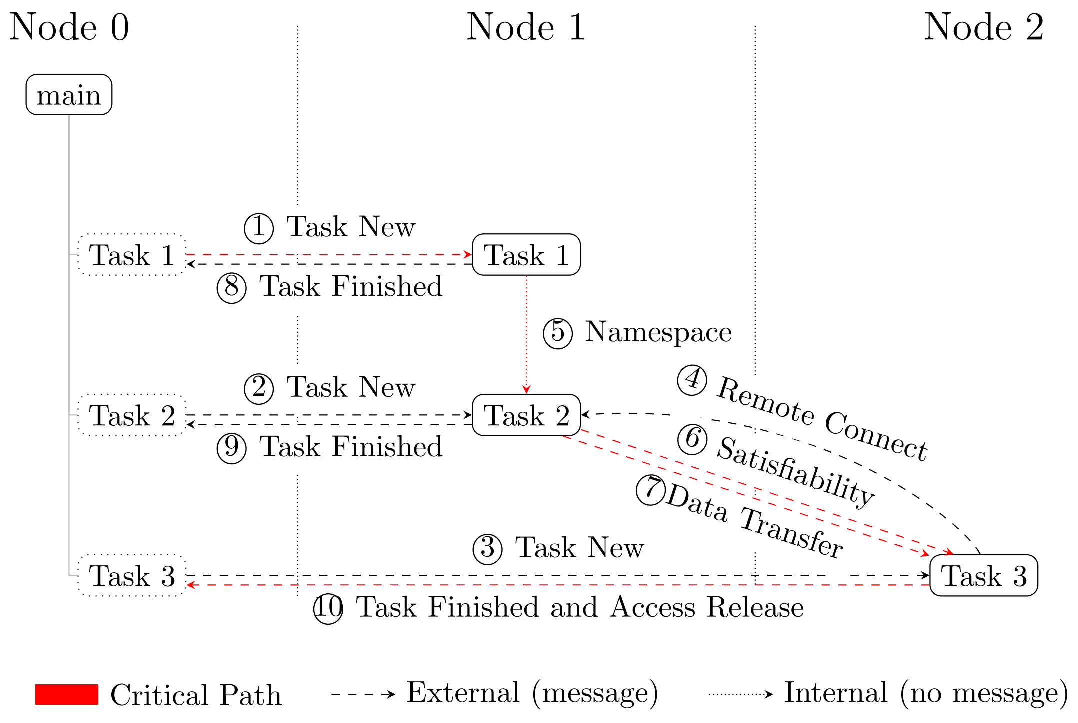


Figure 3.12: Inter-node direct propagation example.

When Task 2 completes, assuming that the Remote Connect message is received in time, it is known that the successor of the access (Task 3) is in a different node. The accesses are therefore released by sending a point-to-point Satisfiability message from the predecessor (Node 1) to the successor's *execution node* (Node 2). In addition, an eager data transfer is initiated if eager data transfers are enabled (see Section 3.5.5). If there is no Remote Connect message or it is not received in time, then the normal approach involving a Data Release message is used. Since weak tasks are offloaded well in advance, it is expected that this fallback mechanism is only used at the beginning of the program's execution.

With the use of weak tasks in the example, only the first Task New message is on the critical path. The other Task New messages are sent immediately after the creation of Task 2 and Task 3 because a weak tasks becomes ready immediately. Offloading of these tasks overlaps with the execution of Task 1.

3.7 Other features and tools

Instrumentation Nanos6 has six options for instrumentation:

ctf Generates a trace in the Common Trace Format (CTF) [60].

extrae Generates a Paraver trace using the Extrae library [178]. The user can control the desired events with a configuration variable to tradeoff the completeness of the trace file and trace file size.

graph Generates a number of encapsulated postscript (eps) files with a graphical representation of the dependency graph. This is intended to be used in small problems.

lint Flag error and bugs in the dependencies constructors with a linter-like fashion.

stats Outputs a summary of some runtime events at the end of the execution, during the runtime finalization. This includes the numbers and total sizes of all MPI messages sent and received by the runtime.

verbose Generates a verbose log of messages about the important events taking place in the runtime to the standard error output. The user can control the verbosity level via a configuration variable.

Every instrumentation is in a different library, and the runtime loader decides which one to use during the initialization based on a user config variable.

OmpSs-2@Cluster adds specialized events to the extrae, stats and verbose instrumentation options. We instrumented the MPI internal communications (send, receive, handle, data transfers), namespace status and propagation and offloaded task.

OmpSs-2-TestGen We developed a random test generator, known as OmpSs-2-TestGen, which generates random valid OmpSs-2 or OmpSs-2@Cluster programs as a way to discover corner cases in the runtime system. It is implemented in Python and able to generate test cases with zero to 1000 nested tasks, taskfors or taskloops, each of which have accesses of any supported access type (in, inout, out, commutative, concurrent, weakin, weakinout, and weakout) to random regions in known arrays. The program satisfies the nesting rules of OmpSs-2, and it automatically checks the correctness of the data passed between tasks. Since it was developed for OmpSs-2@Cluster, it assumes the fragmented region's dependency system. We used a script to generate, compile and test large numbers of random tests. We implemented a script that is able to progressively simplify a failing test case generated by OmpSs-2-TestGen into a minimal failing example. In total, we have 600 tests, the vast majority of which were once-failing test cases generated by OmpSs-2-TestGen.

Address sanitizer The runtime has been tested with Address sanitizer.

3.8 Benchmarks and methodology

3.8.1 Hardware and software platform

We evaluate OmpSs-2@Cluster on the general purpose partition of MareNostrum 4 [17]. Each node has two 24-core Intel Xeon Platinum 8160 Central Processor Units (CPUs) at 2.10 GHz, for a total of 48 cores per node. Each socket has a shared 32 MB L3 cache. The High Performance LINPACK (HPL) Rmax performance equates to 1.01 TF per socket. Communication uses Intel MPI 2018.4, which is the default and supported the implementation of MPI on MareNostrum, which fully exploits the 100 Gb/s Intel OmniPath network and HFI Silicon 100 series Peripheral Component Interface Express (PCIe) adaptor. The runtime and all the benchmarks were compiled with Intel Compiler 18.0.1, and all the kernels use the same code, and the same standard Basic Linear Algebra Subprograms (BLAS) functions from Intel Math Kernel Library (MKL) 2018.4.

All the benchmarks were executed in configurations of 2 processes per node (one per Non-uniform Memory Access (NUMA) node) from 1 to 32 nodes for a total of 64 MPI processes. We selected this configuration to maximize the number of processes and see the impact of the inter-process communications in the results. This is also a way to avoid the NUMA effect which is part of future work and not in the scope of this research.

Using 64 processes, it is enough to expose the different behavior of every benchmark implementation, and the runtime limitations and optimizations.

Every benchmark was executed at least 10 times to report the average and the standard deviation of the values and to calculate the standard error of the mean. We set the number of iterations to get execution times in the order of minutes and reduce statistical noise.

Every benchmark is shown in two different sizes to show the problem size influence in the results (either associated with task granularity or the proportion of communications/computation per node).

3.8.2 Benchmarks

In this work, we implement multiple variants of 4 basic benchmarks with different configurations from 1 to 32 nodes. We selected these benchmarks to represent common HPC applications and detect the most critical runtime issues that may affect them. The application in the list goes from less to the most complex in the same order they were implemented and tested.

For every benchmark, we implemented at least one equivalent MPI+OpenMP version to use as a baseline comparison. The MPI versions use the same algorithm and data decomposition as the OmpSs-2@Cluster version.

3.8.2.1 Matmul benchmark

matmul is a matrix–matrix multiplication: $A \times B = C$ where A, B and C are squared matrices.

This benchmark has multiple characteristics that make it the best candidate to start:

1. It has a complexity of $O(N^3)$ useful to make big tasks.

2. The total amount of data is big enough ($3 * N^2$), but most of it is in read-only (in) accesses. This was useful to detect redundant, unneeded data transfers.

3. The matrix A is divided into slices of rows coinciding with the data affinity(see: Section 3.3.5).

4. The matrix rows are contiguous in memory, so there is no fragmentation.

5. The number of floating point operations is well known.

6. The operations in each row group are totally independent of the others; then, there are no communications between the iterations among the execution.

7. Being a well-known and deterministic problem, it is easy to detect correctness errors using test programs to compare results with tested libraries.

8. It was simple to implement an equivalent MPI version with a similar execution pattern and data distribution to compare.

The results in Section 3.9.1 shows three OmpSs-2@Cluster versions:

Strong flat This is the simpler version; it implements a single level loop of strong tasks on every iteration that is offloaded when they become ready. OmpSs-2@Cluster offloads the tasks only when all the dependencies are satisfied (ready task). In In this benchmark, the access propagation still involves communication with the creator node, but the accesses between tasks can propagate directly and the only delay is associated with the communications between iterations which are much smaller than the computations.

Strong nested This is an optimization of the previous version because it divides the loop in two: a first-level loop creates one strong task per node; once offloaded, the task creates subtasks in the remote node saving the need to send multiple task new messages one by one. As the tasks are ready and with strong dependencies, they do not require later satisfiability messages. The offload latency may be also small because the tasks are offloaded in the same location the required data already is, so they will not start data transfers before executing. Finally, as the problem is well-balanced the grouping of tasks within a strong taskstrong parent task does not create any barrier effect.

Weak nested The key of this benchmark is that the outer loop creates weak instead of strong task. This allows the tasks to be offloaded in advance, almost at the beginning of the execution, so they are created in the nodes long before their execution. The offload and creation overhead should disappear, but the nodes still need to exchange some satisfiability messages and multiple task finish.

These benchmarks exposed the redundant data transfers between iteration fixed with the implementation of the WriteID (see: Section 3.6.2)

3.8.2.2 Matvec benchmark

matvec is a sequence of row cyclic matrix–vector multiplications without dependencies between iterations. Very similar to the previous benchmark.

After matmul, the next logical step was to use matvec because of the evident similitude between them in memory and communication pattern. Most of the characteristics of matmul apply to matvec as well, with only two important differences:

1. The algorithm complexity is N^2, so the tasks will be much more fine-grained compared to matmul.

2. The total amount of data is 3 times fewer data than matmul ($N^2 + 2n$) so the internal control messages will take prevalence over data transfers, if any.

The benchmark versions are equivalent to the ones with matmul in Section 3.8.2.1 and most of the explanation is valid as well, but considering a much smaller task granularity. With smaller tasks the communications between nodes become more important, especially when the number of tasks per node decreases.

Strong flat In spite of the implementation is very similar to the one used in Section 3.8.2.1, the task granularity completely change the expected behavior because the communications and latency may be in the same order as the task computation.

Strong nested The strong nested implementation in this case should behave as an improvement respecting the **Strong flat** version because it significantly reduces the number of messages. However, the latency of offloading one task per node between iterations and create the nested ones introduce extra overhead.

Weak nested When the task duration is in the order of the communications, the weak nested version should provide the best results because the application creates the work in advance, and most of the involved messages will be out of the critical path.

In this case, the expected results are different because the communications are more important and the offload delays more critical. In the end, we discovered that suppressing the offload+creation overhead was not enough to perform with these finer-grained tasks.

3.8.2.3 Jacobi benchmark

Jacobi's method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations. This algorithm is a stripped-down version of the Jacobi transformation method of matrix diagonalization.

If we have a square system of n linear equations: $Ax = y$ where A is a strictly diagonally dominant matrix and x and y are vectors with y known; then we can decompose $A = D + L + U$ and the solution for the system can be obtained iteratively: $x_i^{k+1} = D^{-1}(y - (L + U)x^k)$. We can then set $M = -D^{-1}(L+U)$ and $b = D^{-1}y$ and write the iterative solution as $x_i^{k+1} = Mx^k + b$ being M and b pre-computed in the initialization.

This problem is conceptually similar to matvec, but with the difference that the output in one iteration is the input in the next one. It has the same $O(N^2)$ complexity as matvec, but it has $(N-1)^2$ data transfers between iterations (see below). The objective is to measure the impact of data transfers and control messages to detect optimization opportunities for a problem with complexity and memory access known.

We only implemented versions that offload weak tasks to take advantage of the namespace propagation (see: Section 3.8.2.2 and Section 3.6.1) because we had the previous experience from matvec; a problem with a similar complexity and memory accesses pattern, but without data transfers between iterations.

One of the main advantages of the OmpSs-2 programming model is the possibility to execute different iterations at the same time in some conditions. In the previous benchmarks, we could observe such behavior in the traces, but Jacobi inhibits that feature because all the tasks in the iteration $k+1$ depend on every task in iteration k.

From the runtime point of view, in this benchmark, every task reads and writes a well-defined portion of the matrix and the result vectors; but the input vector needs to be shared in an all-to-all fashion between iterations. The MPI+OpenMP benchmark use `MPI_Allgather` between the iterations but in OmpSs-2@Cluster all communications are point-to-point.

Every released access needs to be shared with the $n-1$ other nodes; then each of them starts a fetch data for that access, and finally, every node shares its portion of the output vector with the others. This gives about $3N(N-1)$ communications between iterations.

Based on the results from matvec this benchmark is implemented in two weak versions only.

Weak nested Similar to the versions with the same name in matmul (Section 3.8.2.1) and matvec (Section 3.8.2.2).

Taskfor This version substituted the inner loop with an OmpSs-2 `task for` construct. The `task for` clause is very similar to the OpenMP `parallel for` from the semantic point of view. It is simpler to write and has the advantage that in Nanos6 it is implemented as a single task that uses multiple helper cores/threads; which means that the `task for` needs to satisfy all the dependencies to start and do not early release dependencies until all the helpers finalize.

3.8.2.4 Cholesky benchmark

The Cholesky factorization is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose, which is useful for efficient numerical solutions.

Given A and a Hermitian positive-definite matrix, the Cholesky algorithm obtains the L real lower triangular matrix with positive diagonal entries such that $LL^T = A$. Every Hermitian positive-definite matrix (and thus also every real-valued symmetric positive-definite matrix) has a unique Cholesky decomposition.[88]

From the runtime point of view the Cholesky factorization is a complex execution and dependencies pattern. This benchmark performs a higher number of smaller tasks compared with `matmul`, and it introduces load imbalance and irregular patterns. The simple version code uses strong tasks and only needs a few lines, while the optimized version uses `task for` and memory reordering optimizations to reduce fragmentation and data transfers.

This benchmark combines 4 BLAS functions interleaved between them, being `DGEMM` the most frequent and expensive. To distribute work, the code implements a fair data distribution by memory blocks with the same dimension of tasksize.

The MPI+OpenMP implementation was complex to implement in order to mix MPI communications with OpenMP tasks without blocking all the threads. MPI is not task aware and the threads can block when doing MPI blocking communications.

The OmpSs-2@Cluster benchmarks performed well compared with MPI since the beginning despite having a much simpler code and not using any optimization from the user's point of view. However, from the traces, we detected some optimization opportunities that we wanted to test in order to get the best possible results with the current runtime implementation and test some other OmpSs-2 advanced features mixed with the Cluster implementation.

In the results, we include three variants for this benchmark:

Strong , which is a simple iterative implementation with strong tasks offloaded from the master node. In this version, all the tasks finished with release accesses go to master.

Weak nested A more elaborated approach adding a nesting level of tasks with only weak accesses (weak tasks) to offload work in advance to take advantage of the namespace propagation (see: Section 3.6.1) when possible.

Optimized This is a very elaborated version with multiple optimizations to reduce data transfers and support messages. This version is based on the **Weak** one but implements multiple tricks to improve performance from the user side without needing runtime modifications. This version exposes how the user may be capable of improving the application with correct application analysis and giving more information to the runtime.

This benchmark tests the OmpSs-2@Cluster performance in a more complex application. The multiple kernels with different execution time introduce some imbalance between tasks; at the same time the dependencies between tasks are much more complex, and the fair data distribution requires a significant number of control messages and complex communication pattern.

The optimized version of the benchmark shows the performance boost a user may expect by giving more hints to the runtime and the effectiveness of some advanced features like task priority, eager send and the use of taskfors for coarse-grained work and reduce communications.

3.9 Results

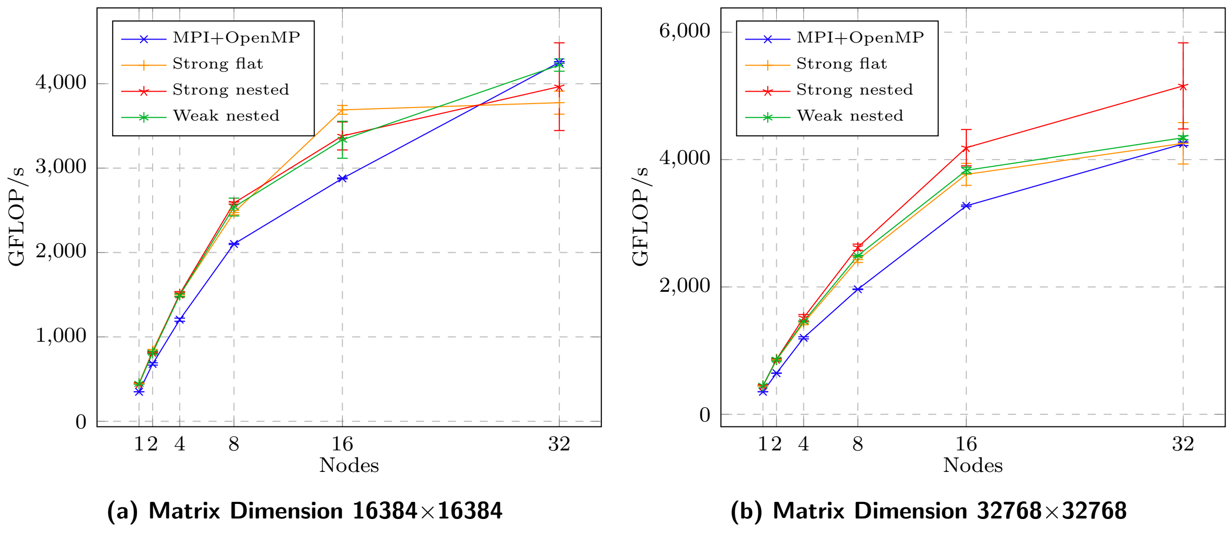
In all strong scalability graphs, the x-axis is the number of nodes, from 1 to 32, and the y-axis is the performance. To calculate the performance, the number of floating point operations of every benchmark is well known and can be divided by the total algorithm execution time to obtain the average number of Floating Point Operations (FLOP) per second. Dividing by 10^9 it gives Billion Floating Point Operations per Second (GFLOP/s).

3.9.1 Matmul

After the first tests, where the scalability was very bad, it was evident from the generic traces that there was a very high number of messages between nodes in spite of the benchmark do not need communications between iterations because its tasks are independent.

At that point, we needed some way to discriminate the MPI messages by OmpSs-2@Cluster message type; and we added specialized Extrae events to the Nanos6 instrumentation (see: Section 3.7).

With that information, we detected that most of the messages were redundant data transfers because the nodes forgot the information about the data already copied. This was solved with the WriteID and the pending transfer optimizations. (see Section 3.6.2)



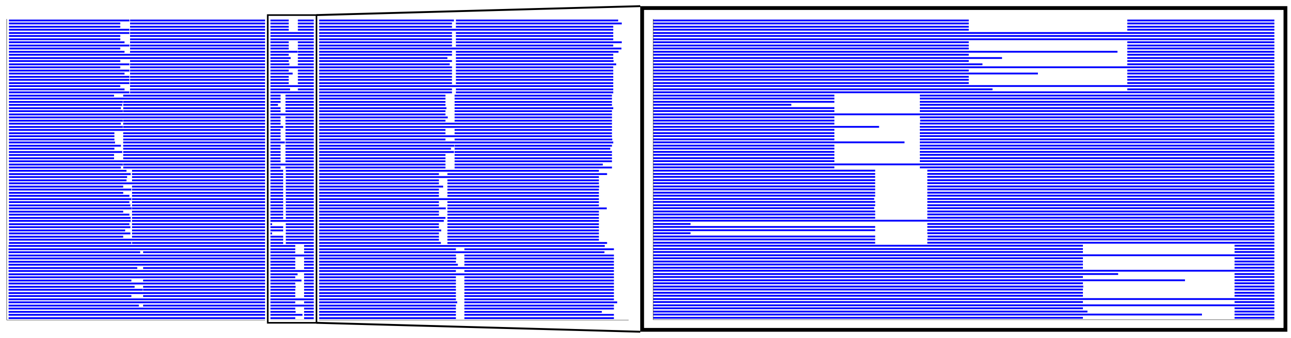
(a) Matrix Dimension 16384×16384 (b) Matrix Dimension 32768×32768

Figure 3.13: Matmul strong scalability.

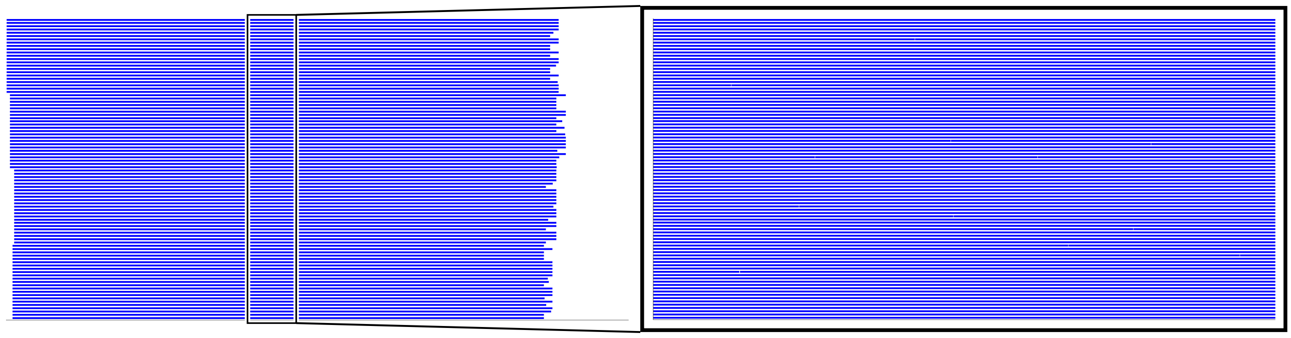
Figure 3.13 shows the scalability for matmul with the three benchmarks compared with an MPI+OpenMP tasks version.

As expected, the best results came from the nested strong version that minimizes the number of messages between nodes at the cost of some delay. For coarse-grain tasks, the delay seems to be negligible compared with the benefit of having less messages. We should remember that in the strong nested version the delay is very small because the WriteID avoids the redundant data transfers for read-only accesses; so the only cost is the offloading cost.

Figure 3.13a shows that the performance of the strong flat version grows faster than the other two for smaller problem sizes. The reason for this is that with smaller problems and many nodes, the number of tasks offloaded per node (Task New messages) is small. Offloading tasks individually seems to compensate the small imbalance that creates the strong tasks between iterations in the strong nested version and the larger number of messages in the weak nested one.



(a) MPI + OpenMP



(b) OmpSs-2@Cluster

Figure 3.14: Paraver traces showing synchronization for matmul: 16384×16384 matrix on two nodes (4 MPI processes) and 24 cores per process each.

Figure 3.14 shows the Paraver traces for 4 iterations of matmul with two 16384×16384 matrices in two nodes (4 processes) and 24 cores per process each. The time scales are normalized to the same time interval either in the full trace images and in the zoom ones.

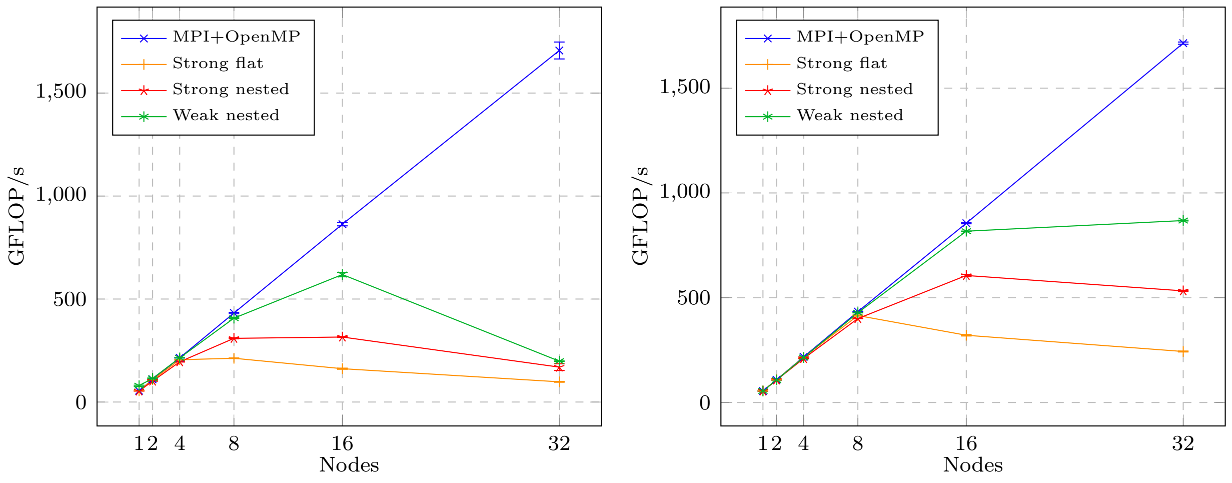
The MPI+OpenMP version (Figure 3.14a) shows that the fork-join approach of OpenMP introduce some small gaps between iterations due to the lack of early release in OpenMP. These gaps are only significant in matmul because the task granularity introduces some imbalance between threads when the number of tasks per iteration is not exactly divisible by the number of workers in the node, and the nest iteration needs to wait that all the previous work finishes before starting.

The OmpSs-2@Cluster strong-flat version (Figure 3.14b) does not show any gap because, in spite of there are communications between iterations, their number and overhead is very small compared to the iteration cost, and they guarantee that there is always work available during all the execution. This is the main reason why the best version for matmul inFigure 3.13 is the strong-flat OmpSs-2@Cluster one.

3.9.2 Matvec

Matvec exposed the cost of the release and satisfiability messages as well as the offload delay in more fine-grained tasks. Initially, we expected that the weak nested version could reduce the delay problem; but the number of release data accesses caused congestion in the master node that was very busy propagating the satisfiabilities for every message. The messages were sent to the master node just to return as satisfiabilities or task new.

The strong tasks does not become ready until all the dependencies are satisfied, which means that there was no alternative to send the messages back to the master. However, the weak messages were already offloaded, and the strong tasks created, waiting for dependencies released on the same node. To solve this, we implemented the namespace optimization. (see: Section 3.6.1).



(a) Matrix Dimension 32768×32768 **(b) Matrix Dimension 65536×65536**

Figure 3.15: Matvec strong scalability.

Figure 3.15 shows that on smaller problems (Figure 3.15a) only the weak version with namespace propagation scale close to the MPI version on to 8 nodes. In these smaller problem sizes, the communication impact starts noticing at 8 nodes, especially for the strong versions. In 32 nodes, the work per node is too small, and the communications increase so much that the execution is almost as slow as in 4 nodes.

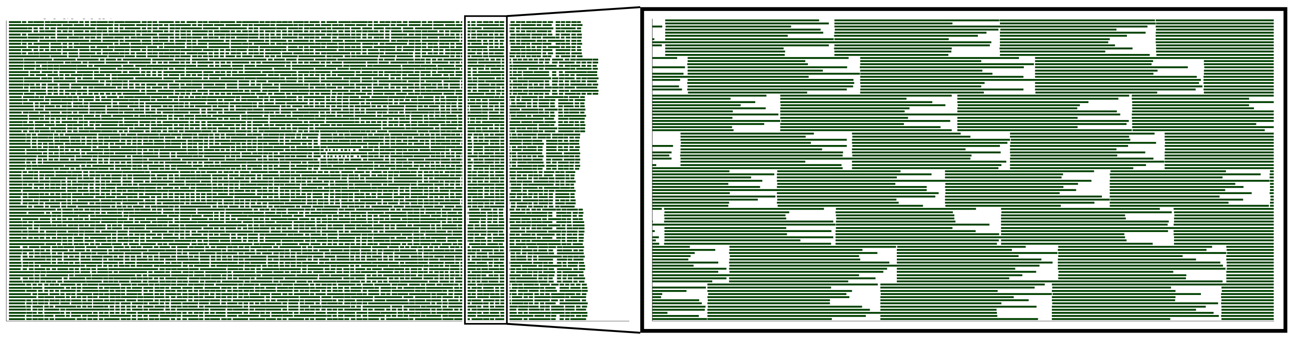
On the other hand, for larger problem sizes (Figure 3.15b) the three OmpSs-2@Cluster versions scale with similar performance to MPI up to 8 nodes (16 processes). In 16 nodes, the behavior is different, but in general, all of them perform worst than MPI which does not perform any communication between iterations.

The strong flat version slows down more than the others because of the cost and delay of offloading multiple strong tasks one by one between iterations outweighs the computations; this is more significant on 32 nodes.

The strong nested is in an intermediate situation because the number of tasknew messages is smaller, but they are sent after processing all the release access messages and satisfiabilities from the previous iteration, adding unneeded delays. Strong nested starts getting inefficient at 16 nodes where the scalability goes away from the linearity. Finally, it slows down on 32 nodes.

Finally, the nested weak version scales close to the MPI+OpenMP version up to 16 nodes (32 processes). This shows the benefit of the namespace propagation and early offload when compared with strong nested version. However, for 32 nodes, this benchmark slows down a bit because, like in the other two, the amount of work per iteration and node is very little.

Figure 3.16 shows shows Paraver traces for 100 matvec iterations on 8 processes (4 nodes) for a matrix dimension of $32768 times 32768$ and 12 cores per process. The MPI+OpenMP version (Figure 3.16a) shows gaps like the ones of Figure 3.14a, consequence of the OpenMP fork–join. In this case the gaps are much smaller because the task granularity is small and they are more evident


(a) MPI + OpenMP

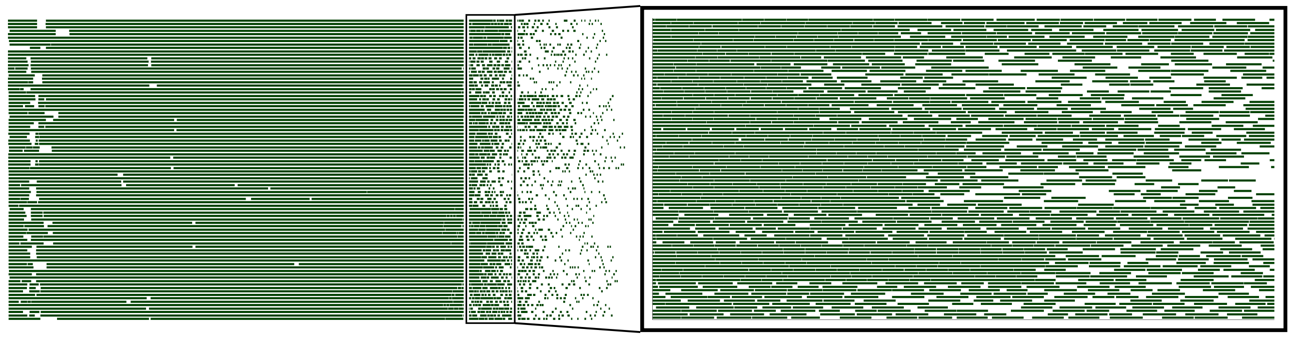

(b) OmpSs-2@Cluster

Figure 3.16: Paraver traces showing synchronization for matvec: 32768×32768 matrix on two nodes (4 MPI processes) and 12 cores per process each.

in the zoom. The OmpSs-2@Cluster version (Figure 3.16b) do not show these gaps in most of the execution, and they are only present at the beginning when the work is still being created and at the end when there is less work and the dependency system propagation evidences some overhead. However in this configuration the two implementations are similar in performance. If the system increases then the processes do not have enough work and most of the trace will look like the end of the trace; for that reason the scalability for 32768×32768 (Figure 3.15a) drops dramatically for more than 16 nodes (32 processes).

It worth to remember that, for this benchmark, the problem size needs to stay constant, so, when increasing the number of nodes the process start starving for work easier because the computation is not very intense. In real applications it may be possible and desirable to increase the problem size to use the extra memory available. In that case the weak scaling will not suffer of work starving issues so easily.

3.9.3 Jacobi

This is one of the harder benchmarks for OmpSs-2@Cluster because of the intensive number of communications, the fine-grained tasks, and the communications pattern. The expected number of communications between iterations for this benchmark is very high as explained in Section 3.8.2.3. But it exposed many optimizations opportunities as well.

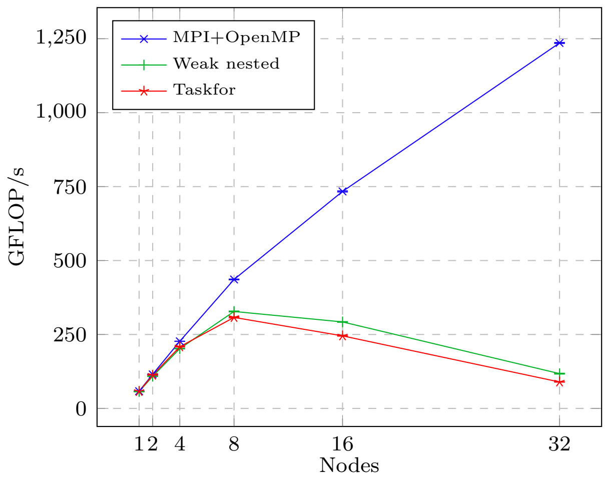
In the first executions, the nested weak version did not scale at all. The `task for` version was not ideal, but at least much faster than the Nested Weak version and scaled up to 4 nodes. On the other hand, looking at the MPI version, we get almost perfect scalability. That was the final sign that all the performance problem is associated with communication because the key difference is that the MPI version use a collective operation.

With the **Nested Weak** approach, the tasks released dependencies individually generating 4 data release messages + a task finish each. All these release messages were processed by the master creating a bottleneck in the propagation and a rainbow of satisfiability messages that saturated all the message handlers. To reduce this issue, we separated the message receiving an operation from the message handling and added helper tasks (see: Section 3.6.5). This gave an initial improvement in the order of 25% for larger task sizes and few nodes; clearly not enough for the number of messages. For smaller task sizes (more tasks and messages to handle), this approach did not benefit performance and actually stressed more the communication infrastructure and the dependency system. In fact, this optimization benefited more from the `task for` version that scaled up to 8 nodes for a 32768×32768 matrix. This was beneficial overall; but made things harder with the goal of bringing both curves closer.

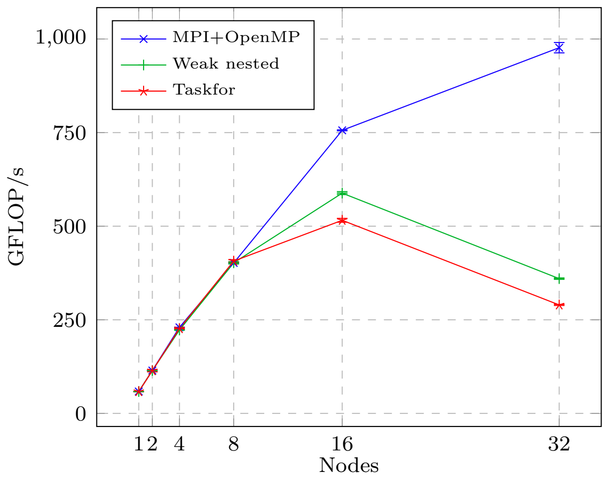
The solution passed then to reduce the number of messages. We first implemented the Message aggregation optimization (see: Section 3.6.3). This reduced the number of release messages by a factor of 4. Then we also grouped the task finish message in the same package when possible, getting an extra bit of improvement. Similarly, in the dependency system, we grouped the satisfiability messages created in propagation by grouping them per remote node. Considering that in this benchmark, every access needs to propagate to all the tasks in the next iteration, this improved performance by a factor of 100 on 8 nodes.

At this point, we had improved the performance significantly comparing with the initial benchmarks; but the taskfor version still was faster. We had grouped the satisfiabilities per task but did not inhibit the early release of the accesses of subtask within a weak task. The `task for` version did that by design, and we considered doing the same for all tasks.

However, we reconsidered the approach because inhibiting all the early release could affect the namespace propagation in applications with unbalanced tasks/subtasks or with less regular work; then the auto release came out (see: Section 3.3.6).



(a) Matrix Dimension 32768×32768

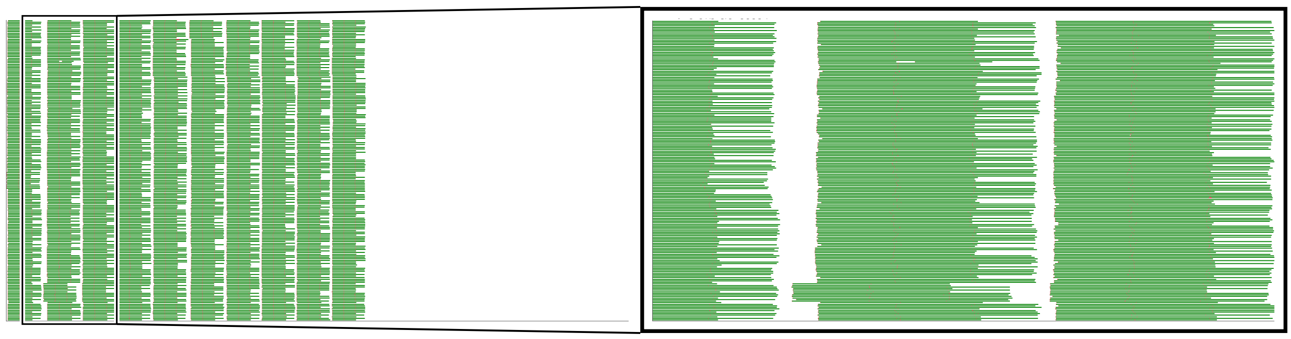


(b) Matrix Dimension 65536×65536

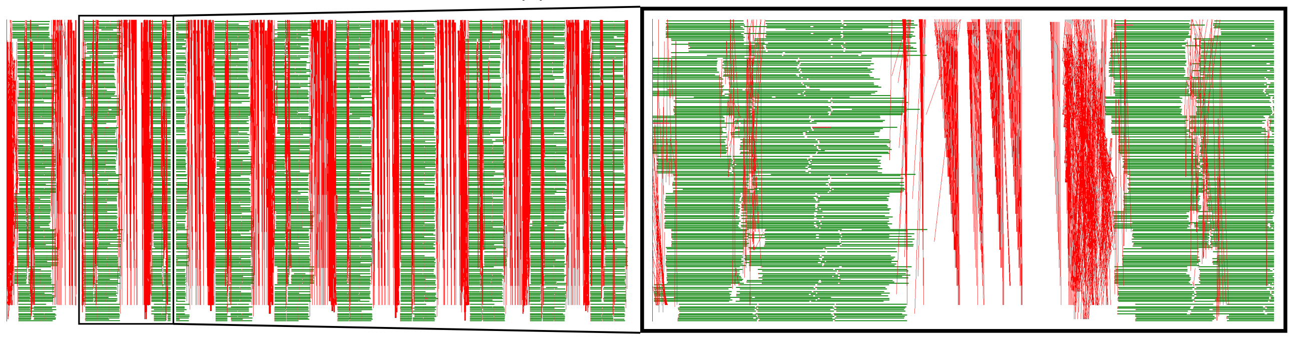
Figure 3.17: Jacobi strong scalability.

Figure 3.17 shows the final performance graphs for the Jacobi benchmark after all the previous optimizations. After all the optimizations' efforts to improve the communications, the performance of the two OmpSs-2@Cluster benchmarks is very similar, which was the initial objective. For 32 nodes (64 processes), the communication cost is so important that it slows down the total execution time below the time of 8 nodes' execution.

Considering that the benchmark is very similar to matvec and the main difference between them is the communication; it is fair to compare these results with Figure 3.15.



(a) MPI+OpenMP



(b) OmpSs-2@Cluster

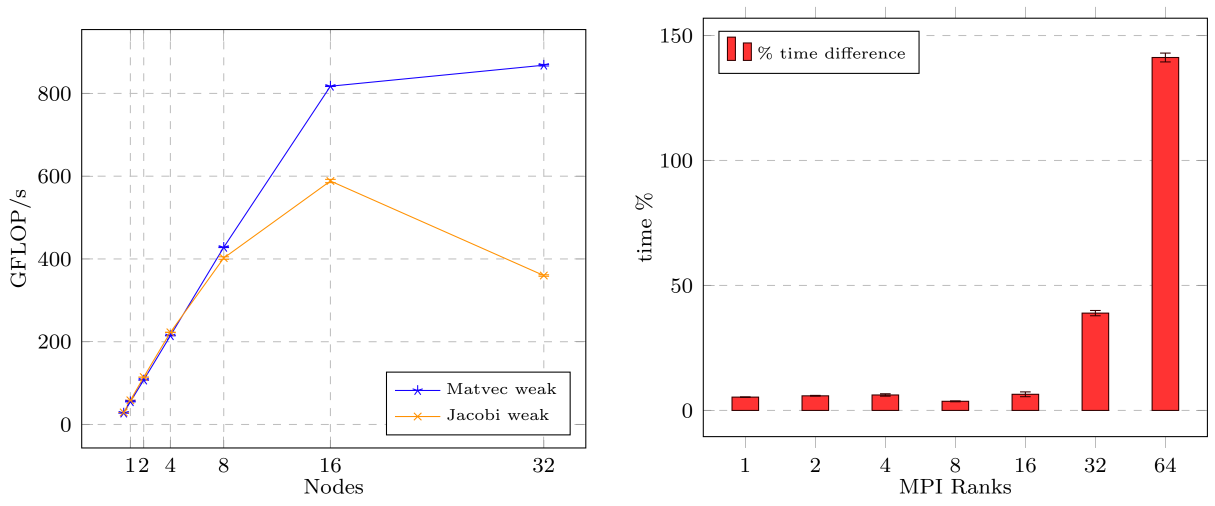
Figure 3.18: Paraver traces showing synchronization for 10 Jacobi iterations : 32768×32768 matrix on 8 nodes (16 MPI processes) and 12 cores per process each.

Figure 3.18 shows Paraver traces for 10 Jacobi iterations with a 32768×32768 matrix on 16 processes with 12 cores each. This trace uses a slightly bigger number of nodes then the other in order to make more evident the overhead introduced by the all-to-all communications. Figure 3.18a shows that the overhead introduced by the MPI collective operation is much smaller, and that is the main reason why in this case, the scalability drops when increasing the number of nodes.

The overhead introduced by OmpSs-2@Cluster between iterations is in the same order as the time needed for computation in every node in this configuration, even using only 12 cores per MPI process and 16 processes (8 nodes). For bigger number of nodes the communication time increases, and the computation time decreases exacerbating and exposing the problem even more.

On the other hand, for this configuration with only 12 cores per process, the overhead between iterations in OmpSs-2@Cluster (Figure 3.18b) is already in the same order than the time needed for computation. For a bigger number of nodes the communications increase and the computation time decreases because there will be more processes to communicate and less work for every process; exacerbating and exposing the problem even more.

3.9.4 Jacobi vs Matvec performance comparison



(a) Performance **(b) Relative Time Difference**

Figure 3.19: Matvec vs Jacobi: strong scalability comparison of the weak nested versions on 65536×65536 problem size.

Figure 3.19 compare the deviation of Jacobi vs Matvec benchmarks respecting the number of processes. This is a fair comparison because they use very similar kernels and the only difference between them is that matvec has no inter-process communication, whereas Jacobi has all-to-all communications.

We use the number of processes and no nodes because the communications are related to the number of processes involved in the application.

In Figure 3.19a the performance of the two benchmarks is almost the same up to 16 processes with an accuracy of 95%. By increasing the number of processes, the curves start diverging because the Jacobi communications between nodes grow quadratically with the number of nodes, and the work per node decreases with the inverse of that number. From Figure 3.19b we see that for 64 processes the extra communications overhead is about 1.5 times larger in Jacobi than Matvec.

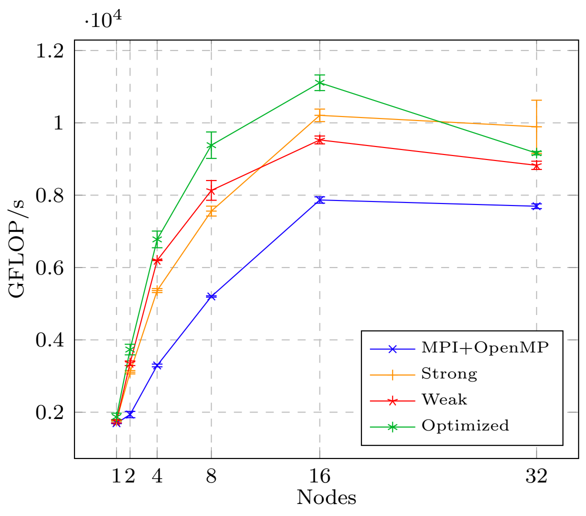
The Jacobi benchmark is still under study because it exposes more optimization opportunities under development considering.

3.9.5 Cholesky

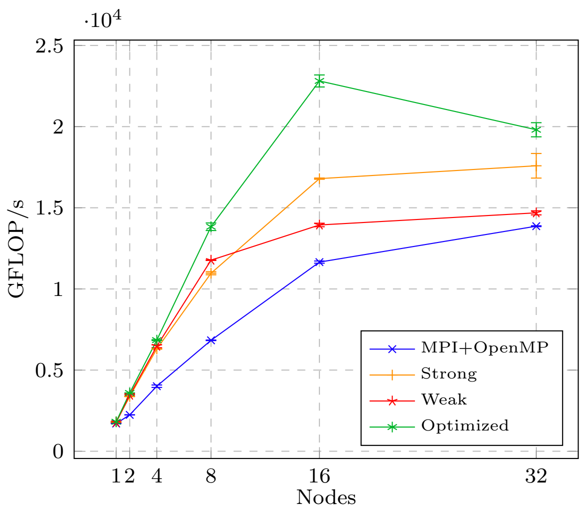
The most striking detail in Figure 3.20 is that all the OmpSs-2@Cluster variants are significantly faster than the MPI+OpenMP version.

The main reason for this is that the MPI version require blocking calls to ensure order and communication completion. OmpSs-2@Cluster does not have this problem. Without dependencies between nodes, the MPI version can only use some blocking calls to synchronize the nodes and ensure a correct operation order. In the MPI benchmark, we only use non-blocking communications, and call `MPI_Wait` selectively when the application will use some data to ensure the transfer already finished.

This schema creates a deadlock at the beginning of the execution because many threads wait



(a) Matrix Dimension 32768×32768



(b) Matrix Dimension 65536×65536

Figure 3.20: Cholesky strong scalability.

for transfers and blocks, leaving the application without the resources to start the sends. The most efficient alternative to solve this issue implements multiple sentinels to ensure that not all the threads block at the same time. The sentinel approach is more efficient because it also behaves as a throttle for communications and prevents blocking many threads on communications for future work. The MPI code implementing all these optimizations is 4 times longer than the simpler OmpSs-2@Cluster version.

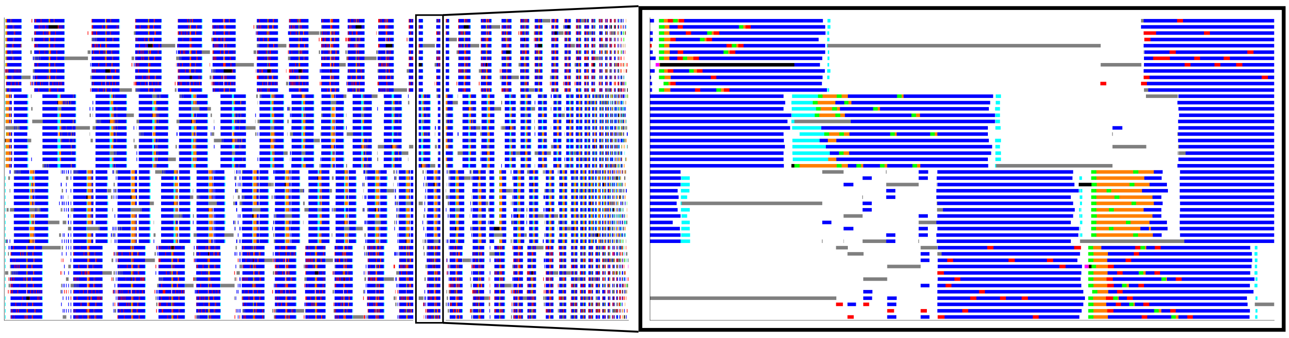
The same Figure 3.20b shows that the strong and the weak versions are very close, up to 8 nodes, but the weak version gets slower on 16 nodes. We found that tasks created in the default weak algorithm creates successors on multiple nodes and inhibits the namespace propagation.

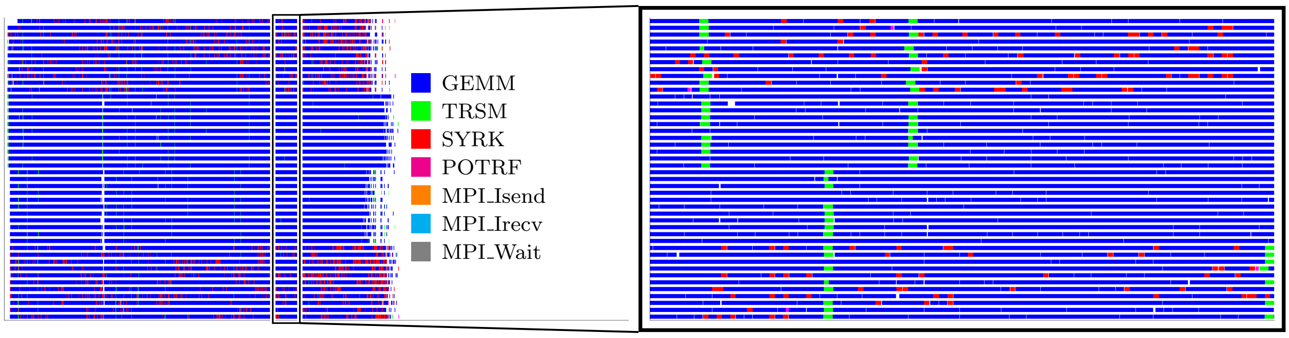
Although the strong version has offloading overhead, in this case, it is a bit beneficial because it creates a throttle and priority effect delaying the operations at the end of the loop that generally is less critical. The offload overhead, in this case, is less important because (1) the fair distribution reduces the number of transfers; (2) the number and granularity of tasks are generally big enough.

The optimized version was finally the best because it solves the order issue by using the OmpSs-2 priority clause and reorders the loops to increase namespace propagation.

To understand this in more detail, Figure 3.21 compares Paraver/Extrae traces for cholesky with MPI+OpenMP vs. optimized OmpSs-2@Cluster. To show an intelligible trace, it is a small example of a 16384×16384 matrix on four nodes, with 12 cores per node. Both traces show a time-lapse of 1160 ms since the algorithm started; the zoomed regions are 50 ms. The traces show the BLAS kernels and MPI communications (only non-negligible in the MPI version).

We see that the OmpSs-2@Cluster version has almost 100% utilization, but synchronization among tasks and MPI communication causes the MPI+OpenMP version to have utilization of only about 50%. As MPI is not task-aware, there are limitations on how MPI calls may be used with OpenMP Tasks to avoid deadlock. Otherwise, tasks in all threads may try to execute blocking MPI calls, occupying all threads even though other tasks may be ready, leading to a deadlock. Resolving this needs synchronization, such as serializing waits or limiting the number of send or receive tasks


(a) MPI + OpenMP



(b) OmpSs-2@Cluster

Figure 3.21: Paraver traces showing synchronization for cholesky: 16384×16384 matrix on four nodes, 12 cores per node.

in every step with artificial dependencies. Figure 3.21a shows how the waits (gray) stop parallelism between iterations.

On the other hand, Figure 3.21b does not show any waits because OmpSs-2@Cluster does not use any MPI blocking function. Tasks not satisfied are not ready, so they remain in the dependency system; while ready tasks with pending data transfers are re-scheduled when they can execute. This approach allows the runtime to concurrently execute tasks from multiple iterations and keeps the workers busy while transfers occur. The priority clause is advantageous to prioritize the scheduling of critical-path tasks, similarly to OmpSs-2 on SMP, but it is more important for OmpSs-2@Cluster due to the network latency.

There is a trade-off in the Cholesky benchmarks involving task size. Smaller tasks achieve higher parallelism and better load balance over most of the execution because there is enough work to keep all the cores busy. But the amount of communication and the number of dependency system operations increases with the number of tasks, i.e. with the inverse of the task size. On the other hand, larger task sizes reduce the total amount of communications and the number of dependency operations. Additionally, the BLAS kernels are applied to larger submatrices, which is more efficient. But as Cholesky decomposition proceeds through the computation, the amount of parallelism decreases and load imbalance appears towards the end of the execution. The load imbalance is visible towards the end of the trace in Figure 3.21b, and it is due to the fair data distribution. It may be improved through a better scheduler as part of future work (see Section 6.2.7). In each case, we chose the best block size for each execution, which is a compromise between the two extremes. We use the largest problem size that fits into the physical memory of one node, and weak scaling results would, of course show better scalability.

3.9.6 Performance impact of the multiple optimizations

This section compares the influence and impact of the main 5 optimizations described in Section 3.6 for the optimized version of the benchmarks described in Section 3.8.2. The impact is different for every benchmark depending on the problem type, data flow, task granularity and application regularity.

For these benchmarks, all the optimizations implemented in the runtime were disabled and added in the same order they were implemented one after the other. The effect of every optimization is not lineal respect to the others because some optimization enhances others (i.e. writeID and namespace) while some others address similar problems with different approaches (i.e. Message aggregation and Message handler helpers).

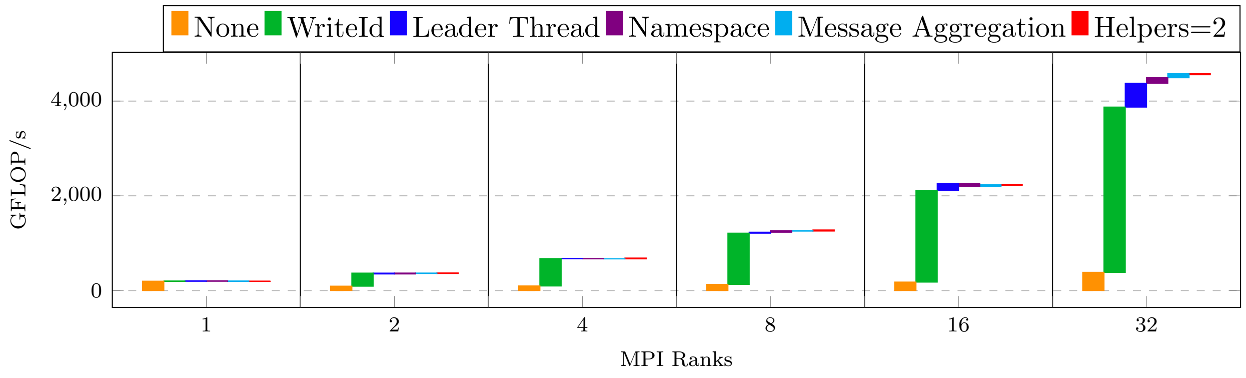


Figure 3.22: Performance impact of the different optimizations in matmul benchmark. Matrix dimension 16384×16384.

Figure 3.22 shows that writeID is the most important optimization for matmul. As there are not communications between iterations and the tasks are coarse-grained, then the main source of inefficiency exposed in matmul is the redundant data transfers addressed and solved with writeID optimization as explained in Section 3.6.2. Most of the other optimizations have a very limited impact on performance for matmul because they solve delays and latency issues associated with communications not exposed by matmul as explained in Section 3.9.1.

On the other hand, matvec is more influenced by communication optimizations as shown in Figure 3.23.

- Without optimizations (yellow), the scalability is negative.
- Up to 8 nodes the WriteID (green) and namespace (violet) are the most important optimizations. This is expected because the namespace optimization addresses the main new issue added by matvec, which is the small task granularity.
- For 32 nodes, the Leader Thread (blue) and the Message Handler Helpers (red) become more important because the amount of work per iteration and node is very small, and handling the control messages becomes more important.

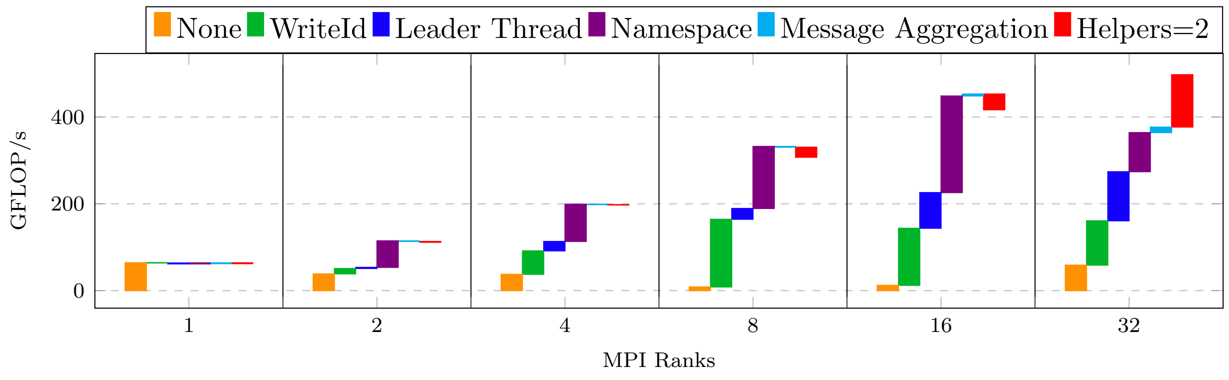


Figure 3.23: Performance impact of the different optimizations in matvec benchmark. Matrix Dimension 32768×32768

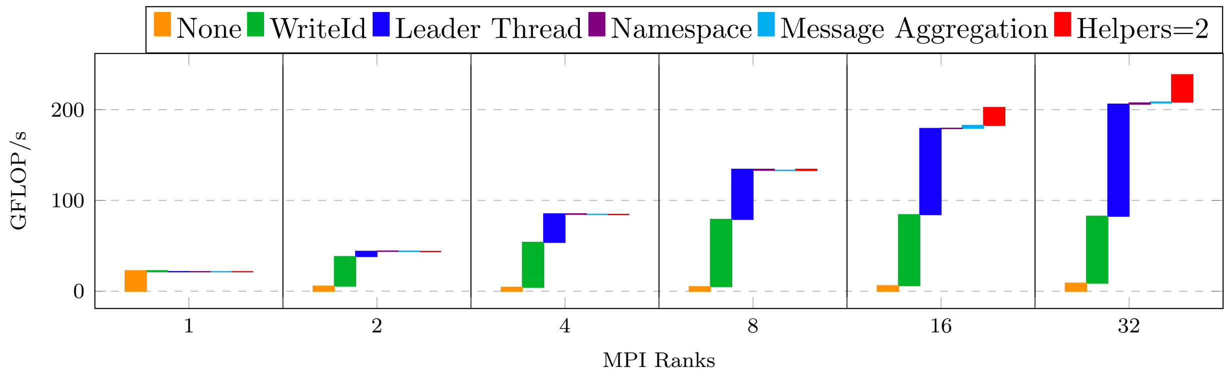


Figure 3.24: Performance impact of the different optimizations in Jacobi benchmark. Matrix Dimension 32768×32768

On the other hand, Jacobi is specially benefited by the WriteId and the Leader thread for any number of nodes according to Figure 3.24. This is expected because Jacobi is especially intense in communications, as explained in Section 3.9.3. An interesting detail is that Figure 3.24 does not show any important improvement of using Message aggregation for Jacobi even though it is very intense in communications; the main reason for this is that Jacobi uses the `wait` clause to inhibit the early release (see: Section 3.3.6) which allows OmpSs-2@Cluster unfragment the release access before sending the messages.

Finally Figure 3.25 show some improvements with all the optimizations except for the helper threads. While Cholesky perform multiple communications during the execution, those are not so intense respecting to the order of the computations and the helper threads may start some communications to satisfy work in remote nodes while more critical work is pending in the local one. The such bad impact is actually reduced on 32 processes mainly because the work per node is smaller, and the impact of temporarily using some threads become negligible.

3.9.7 Performance impact of namespace propagation

As explained in Section 3.6.1, the namespace propagation benefits only the tasks offloaded in advance generally weak tasks, by reducing latency and unnecessary messages.

Figure 3.26 shows impact of removing the namespace propagation and keep all the other opti-

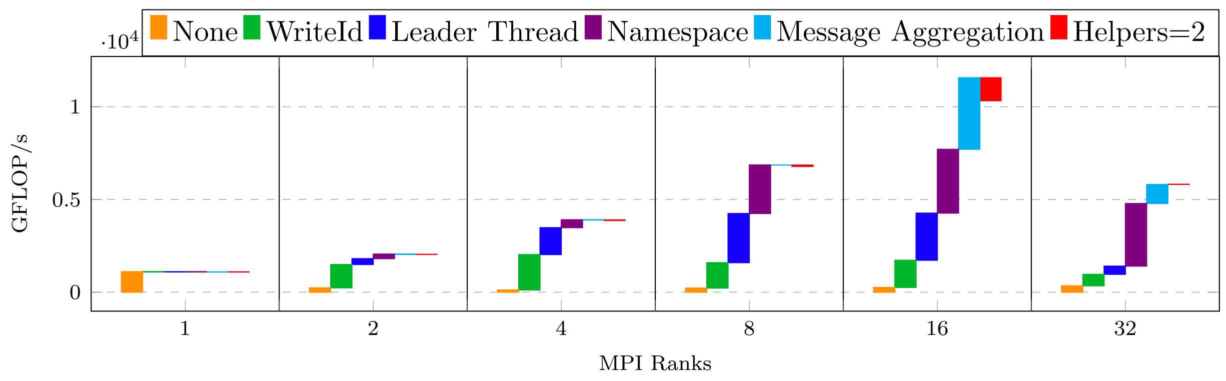
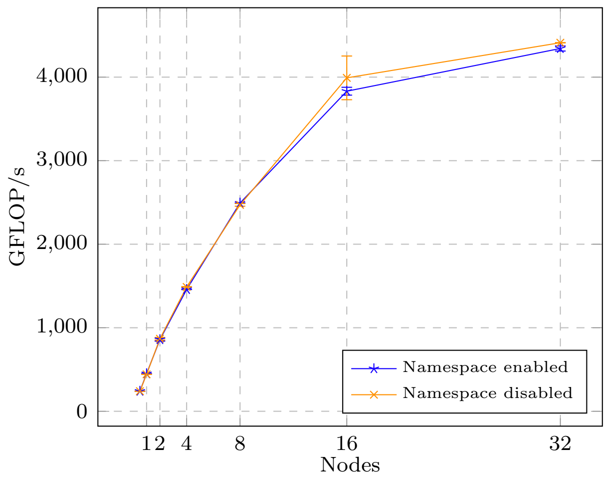
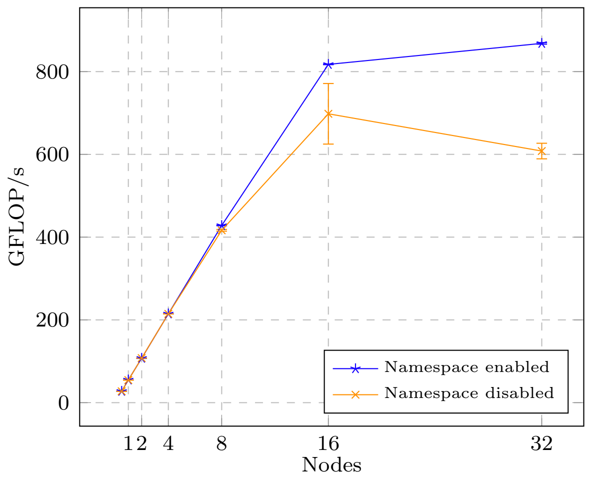


Figure 3.25: Performance impact of the different optimizations in Cholesky benchmark. Matrix Dimension 32768×32768



(a) Matmul 32768×32768



(b) Matvec 65536 × 65536

Figure 3.26: Namespace impact on strong scalability for matmul and matvec weak nested versions.

mizations enabled in matmul (see: Figure 3.13b) and matvec (see Figure 3.15b).

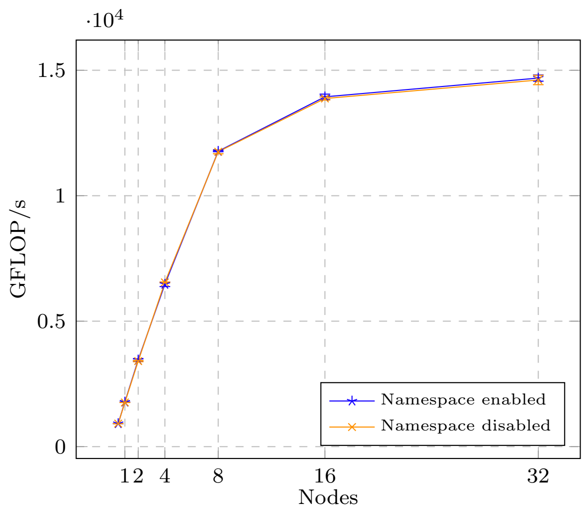
In matmul (Figure 3.26b), there is no appreciable difference because the task size is much larger than the rest of the overhead, even with 32 nodes (64 processes).

On the contrary, matvec (Figure 3.26b) shows a benefit of 25% in performance on 32 nodes. The difference is negligible with less than 8 nodes because the relation between computation and overhead is not critical, and the runtime is capable to overlap both.

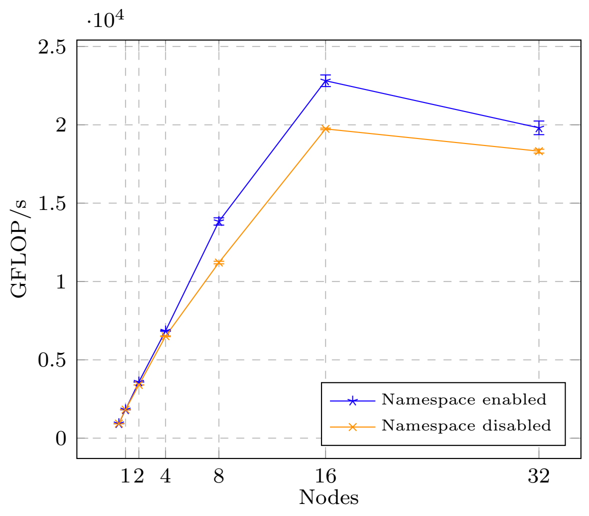
This section does not include a similar comparison for the Jacobi benchmark because the namespace propagation is negligible compared with the rest of the communication overhead. And the results in Figure 3.24 prove that Jacobi does not benefit from namespace propagation.

Similar to Figure 3.26; Figure 3.27 compares the namespace performance benefit but this time for the Cholesky versions with weak tasks. Figure 3.27a shows that the weak version of the cholesky benchmark is unable to take a significant advantage of the namespace propagation in spite of having weak tasks.

On the other hand, the optimized version (Figure 3.27b) gets a 15% of improvements on 16 nodes from namespace propagation by just modifying and reordering the initial code.



(a) Weak



(b) Optimized

Figure 3.27: Namespace impact on strong scalability for the Cholesky benchmarks versions with weak tasks

3.9.8 Performance impact of the wait clause

Jacobi benchmarks do not significantly benefit from the namespace propagation because of the intrinsic all-to-all propagation between iterations and the fragmentation of the output (see: Section 3.8.2.3 and Section 3.9.3). However, once the user knows the application communication pattern, it is possible to provide extra hints to the runtime to optimize the application execution (see: Section 3.3.6)

The wait clause is one of those optimizations added by OmpSs-2@Cluster under the user's control. In this section, we study the direct impact of the `wait` clause in the performance for the Jacobi benchmark.

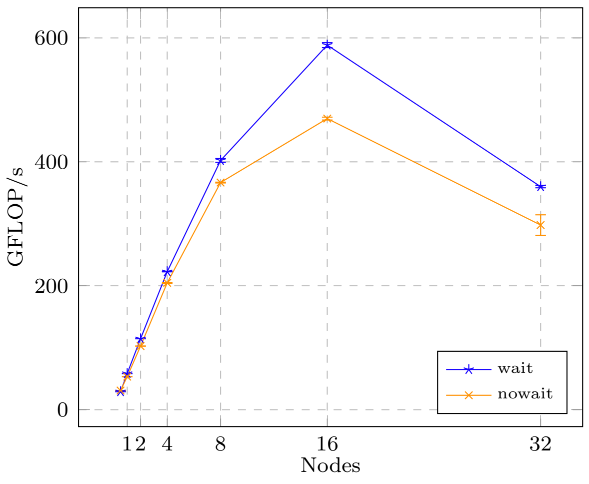


Figure 3.28: Wait clause impact on strong scalability for Jacobi weak nested benchmark.

Figure 3.28 shows the impact of adding a wait clause in the outer task level of the Jacobi weak nested benchmark. In this case, the wait clause only reduces some of the messages involved in the release, data accesses from nested tasks, but many of the other support messages (satisfiability, send data raw) are still taking place.

As expected, for a small number of processes (< 8), there is no significant difference in performance, but for more than 8 nodes, the wait clause gives from 10% (8 nodes) up to 25% performance improvement on 32 nodes.

3.10 Conclusions

This chapter presented OmpSs-2@Cluster, a programming model and runtime system, which provides efficient support for hierarchical tasking from distributed memory to threads.

We describe the OmpSs-2@Cluster programming model, its differences and contributions to OmpSs-2 in Section 3.3 and provide details about the Nanos6 implementation in Section 3.5. The Section 3.6 explains in detail the optimizations required and implemented with special emphasis in the namespace optimization.

Section 3.8.2 describes the 4 benchmarks: matmul, matvec, Jacobi and Cholesky and their comparative characteristics and reason for electing each.

Finally, the results in Section 3.9 show that performance of OmpSs-2@Cluster is competitive with MPI+OpenMP for regular and well-balanced applications. The main limitations to scalability are the use of point-to-point data communications in the runtime (for Jacobi, in Section 3.9.3) and offloading/satisfiability overheads for fine-grained tasks on more than about 16 or 32 nodes. For irregular or unbalanced applications it may be significantly better without increasing the code complexity or sacrificing the programmer productivity. The section also discusses the effectiveness of the main automated optimizations implemented in the runtime, and one under the user control and consideration (`wait` and `nowait` clauses).

This work opens future work to leverage this model for (a) resiliency, (b) scalability, (c) intelligent multi-node load balancing, and (d) malleability. With this aim, all source code and examples are available open source [24].

Chapter 4

Multi-node dynamic load balancing

4.1 Introduction

As explained in Section 1.3, load imbalance is a long standing and significant source of inefficiency in high performance computing. Load imbalance is made worse by increasing application complexity, e.g. Adaptive Mesh Refinement (AMR) and modern numerics, and system complexity, e.g. Dynamic Voltage and Frequency Scaling (DVFS), complex memory hierarchies, thermal and power management [1], Operating System (OS) noise, and manufacturing process [100]. Load balancing is usually applied in the application [108, 133, 12], but it is time-consuming to implement, it obscures application logic and it may need extensive code refactoring. Load balancing requires an expert in application analysis and it may not be feasible in very dynamic codes. Static approaches, such as global partitioning, are applied at fixed synchronization points that stop all other work [161, 120], and they rely on a cost model that may not be accurate. This task gets harder as the size of the clusters continues to increase.

This chapter presents an automated approach for load balancing Message Passing Interface (MPI) + OmpSs-2 [23] programs that relieves the application from the burden of balancing the load across nodes. We extend the work in the previous chapter, which enables OmpSs-2 tasks to be offloaded to other nodes, to employ Dynamic Load Balancing (DLB) [82] mechanisms and handle fine-grained and coarse-grained load imbalance.

The motivations to use an approach oriented to hybrid MPI+OmpSs-2 applications are:

Existing applications There are multiple hybrid MPI+OmpSs-2 applications and benchmarks developed for OmpSs-2@Shared Memory Multiprocessor System (SMP). Those applications can benefit from this approach with a minimal effort, while porting them to a pure OmpSs-2@Cluster version may be time consuming and break some of their backward compatibility. Moreover, porting hybrid MPI+OpenMP applications to use OmpSs-2 tasks requires programmer effort. However, these changes are generally small compared to the effort of removing all the MPI infrastructure if it already exists. That may end up in a complete new implementation for the application. Due to the similarity in the annotation approach and syntax, some applications like Alya [8] support OpenMP and OmpSs at the same time for shared memory parallelism with the same code base. Those applications can also benefit from this approach keeping support for OpenMP but without maintaining different code bases.

Large number of nodes The current OmpSs-2@Cluster implementation is scalable to a small number of nodes. Although future work will be aimed at optimizing and improving the runtime for larger systems, some OmpSs-2@Cluster applications will have difficulties to scale in a large number of nodes when compared with MPI applications optimized in the user code level (i.e. using collective operations).

An MPI+OmpSs-2 program requires only minor and local changes to be compatible with our model because no additional markup is required beyond the existing annotation of the task accesses. The single mechanism of task accesses is used to compute dependencies for parallel task execution, for data locality on the node, and for data transfers and locality optimizations on multiple nodes.

We leverage the malleability of OmpSs-2 and OpenMP in terms of their ability to adjust to dynamically varying numbers of cores. Our approach uses DLB as the underlying mechanism; the Lend When Idle (LeWI) module [80] reacts to fine-grained load imbalance and DLB's Dynamic Resource Ownership Management (DROM) module addresses coarse-grained load imbalance by reserving cores on other nodes that can be reclaimed on demand.

A key aspect of this work is the use of an expander graph (defined in Section 4.4.2), in which each MPI rank in the application directly offloads work to a small number of other nodes. This approach limits the amount of point-to-point communication and state that needs to be maintained, and it provides a path to scalability to large numbers of nodes.

We evaluate application-level imbalance using a micro-scale solid mechanics application with up to 64 nodes, and reduces the execution time by 46% compared with an execution using the same number of nodes and state-of-the-art DLB (which performs load balance separately on each node). This is only 7% above the theoretical time-to-solution with perfect load balancing. We evaluate system-level imbalance by executing an n-body Barnes–Hut simulation with two ranks per node on up to 16 nodes, one of which has a reduced clock frequency of 1.8 GHz, while the rest run at 3.0 GHz. We also use a synthetic benchmark to increase confidence that our approach works for a range of scenarios that may not all be encountered in the applications. The synthetic benchmark performs a sweep of the load imbalance, from perfectly load balanced to the extreme where all work is on one node.

DLB reduces the execution time by 16% and our approach reduces the execution time by a further 20%, with respect to the same baseline. This is an example where our dynamic approach operates in addition to the application's own load balancing technique, in this case Orthogonal Recursive Bisection (ORB), whose cost model does not adapt to varying node performance. The synthetic benchmark shows that our approach provides good load balancing across a wide range of application imbalance, within 10% of perfect load balancing for an imbalance of up to 2.0 on 8 nodes.

4.2 Background

4.2.1 Dynamic Load Balancing (DLB) fundamentals

DLB [82, 81] is a library to enable dynamic load balancing among multiple processes on the same node, from either the same or different applications. DLB can balance applications with two levels of parallelism where the inner level is malleable and use the malleability of the inner level to balance the outer level. It exploits the malleability features of OmpSs-2 and OpenMP; i.e., the ability to dynamically adapt to varying resources at runtime, in this case the number of cores. It is compatible with MPI, OpenMP and OmpSs.

DLB is a flexible solution transparent to the applications and the programmer does not need to modify the application. Since version 3.0 DLB includes three independent and complementary modules:

Lend When Idle (LeWI) The main idea of the LeWI algorithm is to lend the OpenMP threads (Central Processor Units (CPUs)) of an MPI process while it is waiting in a blocking call to another MPI process running in the same node that is still doing computation [80].

When the MPI process that has lent the CPUs gets out of the blocking call it will recover its CPUs and the process that was using them will be notified to stop using the lent threads

Under the control of the runtime (either Nanos6 or OpenMP), LeWI provides the mechanism to respond to a fine-grained imbalance. It provides a straightforward Application Programming Interface (API) to lend cores when they would otherwise be idle and to borrow them, when needed, by a different process on the node. Crucially, the lender may reclaim the cores as soon as they are needed again.

Dynamic Resource Ownership Management (DROM) is an interface that provides efficient malleability with no effort for program developers [56]. The running application is enabled to adapt the number of threads to the number of assigned computing resources in a completely transparent way to the user through the integration of DROM with standard programming models, such as OpenMP/OmpSs, and MPI.

DROM enables coarser-grained load balancing by providing an API to change the semi-permanent ownership of cores among the processes on the node. Ownership of cores in proportion to the average load provides the ability for processes to reclaim the right number of cores when they are needed.

Tracking Application Live Performance (TALP) is a portable, extensible, lightweight, and scalable tool for parallel performance measurement [121]. Tracking Application Live Performance (TALP) implements the well-defined and established Performance Optimisation and Productivity Centre of Excellence (POP) metrics [151] and offers an API to consult them during the execution.

The API can be used by the application or other resource managers or job schedulers. The basis of the implementation intercepts the MPI calls and accounts for the time spent by each MPI process doing useful computation or communication. It also takes into account the number of threads that are being used in case the applications use an Hybrid programming model.

In this work we use the DROM and LeWI modules to perform coarse and fine-grain load balancing.

4.3 Programming model

4.3.1 OmpSs-2@Cluster and MPI interoperability

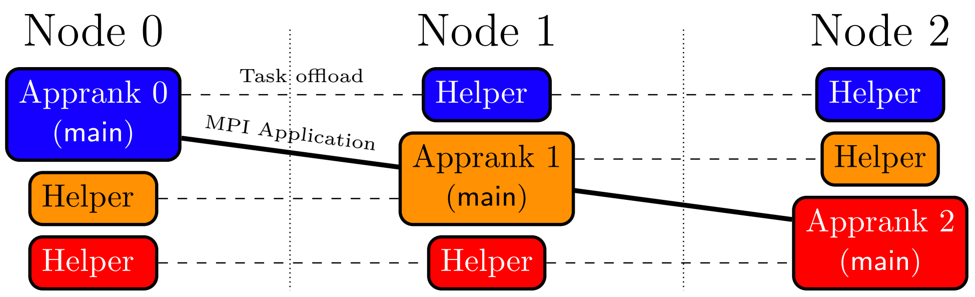


Figure 4.1: Architecture of MPI+OmpSs-2@Cluster. Application ranks (appranks) communicate via MPI and helper ranks on some other nodes can execute tasks of heavily loaded appranks. The main function runs as tasks on the appranks.

We extended the OmpSs-2@Cluster programming model to support interoperability with MPI so it can be used for load balancing of MPI + OmpSs-2 programs. The overall approach is illustrated in Figure 4.1. As for any MPI program, the application's main function executes in Single Program Multiple Data (SPMD) fashion across the compute nodes. The MPI ranks visible to the application are known as appranks (or application ranks), and they communicate in the normal way using MPI (solid line in Figure 4.1). Unlike regular MPI, however, each apprank is supported by a few helper ranks on other nodes (connected with dashed lines). All ranks including helper ranks are full instances of Nanos6 that execute tasks across a variable number of cores. The runtime instances will coordinate to balance the loads, offloading tasks from the apprank to a helper rank, as needed. Offloadable tasks are defined using regular OmpSs-2 task annotations. Figure 4.2 shows an example n-body kernel that is a slightly simplified extract from the n-body application used in the evaluation. There is a single offloadable task that calculates the forces on a number of bodies. The `pragma` annotation defines the task together with its inputs and outputs.

The application is compiled and executed in the same way as any MPI+OmpSs-2 program (see: Section 3.2.2). The runtime only needs an extra configuration parameter to enable task offloading.

Two superficial changes are required in the application source code to support MPI+OmpSs-2@Cluster. Firstly, all occurrences of MPI_COMM_WORLD should be replaced with a call to the relevant runtime API call, nanos6_app_communicator(), to obtain the communicator to be used by the application. Secondly, since the runtime requires MPI communication before and after the execution of the main function, the application should not itself call MPI_Init() or MPI_Finalize(), as these calls are made by the runtime. Making these changes in the application, rather than through a custom `mpi.h` passed to the application ensures portability across all systems. It maintains the OmpSs-2 property that code without OmpSs-2 pragmas, which may use `mpi.h`, may be compiled in the normal way with the host compiler.

```
for (int j = 0; j < num_bodies; j += block_size) {
  int n = MIN(block_size, num_bodies-j);

  #pragma oss task in(bodies[j;n]) in(cells[0;num_cells]) out(forces[j;n])
  for (int k=j; k<j+n; k++) {
    // Compute forces[k]
  }
}
```

Figure 4.2: Example OmpSs-2@Cluster code for n-body kernel.

MPI calls on the application's communicator are valid, so long as it is not done in an offloadable task, or a descendant (child, grandchild, etc.) of an offloadable task. This is consistent with the normal way of developing MPI+OpenMP or MPI+OmpSs-2 programs, which use tasks to perform compute operations, but not communication, due to the risk of deadlock. The intention is to avoid having to intercept MPI calls to manage MPI communication among offloaded tasks.

We maintain the property of regular OmpSs-2@Cluster that tasks are always executed in a process with the same virtual address structure as their apprank Chapter 3. The different appranks have isolated virtual address spaces, so that different objects on different appranks may be allocated to the same virtual address, even if they are both accessed by tasks executed (within different appranks) on the same physical compute node. This simplifies the porting of applications that use global or static data or that use libraries that do so. It allows the programmer to not have to worry about placing global and static data at different addresses, which could otherwise cause bugs that are difficult to track down.

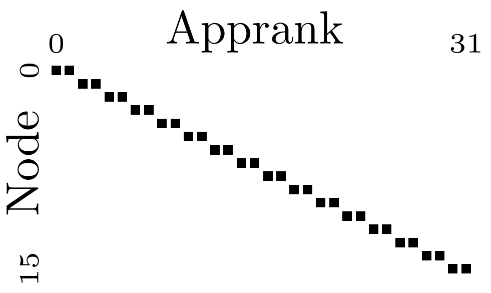
4.4 Runtime implementation

4.4.1 Overall architecture

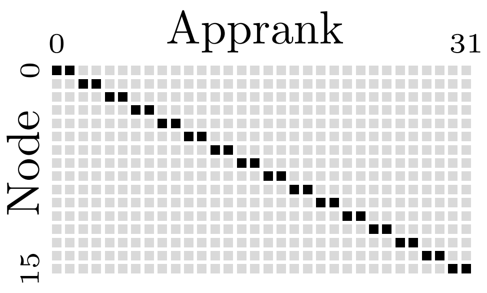
In Figure 4.1, each apprank and helper rank is a process with multiple threads that can execute tasks. If the application is well-balanced, then only the appranks are involved in the computation, in the same way as any MPI+OmpSs-2 program. In this case, the helper ranks will remain idle. Using DLB, multiple appranks on the same node can respond to small load imbalances, by shifting cores to the appranks with more load. At some point, however, it will become necessary to break the confinement of a single node and offload work to helper ranks on other nodes.

Offloading uses the existing task offload mechanism described in Chapter 3. No additional support is needed for MPI+OmpSs-2 because MPI communication is not supported in offloadable tasks or their descendants. Most programs do not need to perform MPI communication in tasks, and it would be difficult to implement the matching of MPI communication among tasks offloaded to different nodes.

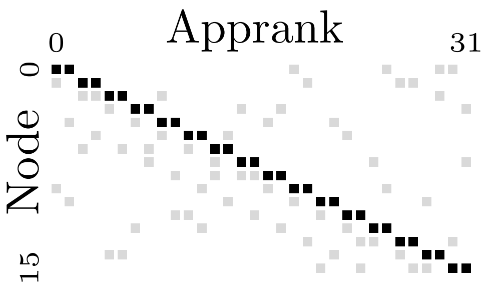
The offloading mechanism does not merely spread the load of a single heavily loaded apprank or node. It is designed to avoid any bottlenecks, where a local load imbalance can get "stuck" within a single node or a small group of nodes. At the same time, we wish to minimize the number of helper ranks in the system, since each helper rank implies point-to-point communication and state and scheduling complexity.



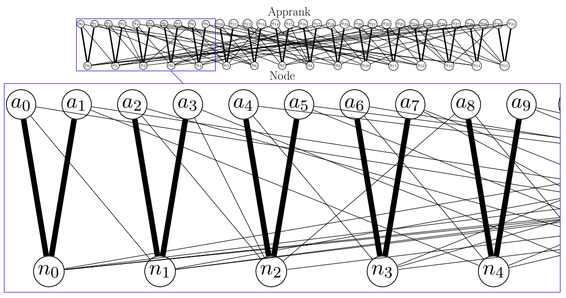
(a) DLB: load balancing on each node.



(b) Offloading any apprank to any node.



(c) MPI + OmpSs-2@Cluster: each apprank can offload to few nodes.



(d) Bipartite graph representation for 4.3c.

Figure 4.3: Work spreading of 32 appranks on 16 nodes. Offloading tasks to a few other nodes allows the work to be spread across all nodes used by the application.

4.4.2 Spreading of work

Figure 4.3a represents the mapping of appranks (application ranks) to nodes when an application executes with 32 appranks on 16 nodes, two appranks per node. Assuming that DLB is enabled, the load can be balanced among the two appranks (black squares) on the same node, but any load imbalance is confined to a node. Figure 4.3b is the opposite extreme, where each apprank executes its main function on the same rank as before (black square). It can still balance the load with the other apprank on the same node, but it can also offload tasks to any other node (grey squares in the same column).

This configuration provides the maximum ability to spread work, but it requires a lot of state for point-to-point communication and for each rank to keep track of where to send the work. Changing behaviour of the different appranks in response to each other causes a lot of variability in the execution. Finally, Figure 4.3c is an example following the philosophy of this chapter: each apprank executes tasks across a small number of nodes, providing the ability to spread work while limiting the amount of state and variability.

Static work spreading: The simplest approach is to allow each apprank to directly offload tasks to a few other nodes that have been chosen before the application begins execution. A large body of work exists on *expander graphs*, which have the properties we need, so we first translate the problem into the language of graph theory. Rather than arranging the appranks and nodes into a grid, we define a bipartite graph of appranks and nodes and draw an edge between an apprank and a node if the apprank can execute tasks on that node.

Figure 4.3d shows the graph representation of the scenario from Figure 4.3c at the top, with a zoom on part of it at the bottom. There are two types of vertices. Each of the vertices drawn across the top of the graph represents an apprank, i.e. an instance of `main`. Each of the vertices drawn across the bottom represents a node. The thicker solid edges identify the original processes running the appranks, i.e. it indicates which node executes `main` for that apprank. The thinner lines indicate the presence of a helper rank that can execute offloaded tasks from a given apprank on a given node. Overall, each edge corresponds to a process that executes an instance of Nanos6: it executes tasks from a given apprank on a given node. In Figure 4.3d, each apprank has degree three (one apprank and two helper processes) and each node has degree six (the two appranks and four helper processes on that node).

There are several definitions of an expander graph in the literature [96], but for our purposes we define a bipartite expander graph as a bipartite graph for which $|N(A)| \geq (1+\epsilon)|A|$ for some large $\epsilon > 0$, for every subset A of at most half of one of the partitions of the graph [62]. In our context, A is a subset of the appranks and $N(A)$ is the set of all nodes that are adjacent to at least one apprank of A. $|A|$ is the number of appranks under consideration and $|N(A)|$ is the number of nodes over which the work of A can be divided.

The definition means that the work belonging to each subset of the appranks can, in principle, be spread across a good number of nodes, which grows with the number of appranks. This achieves our aim of avoiding bottlenecks on the ability to spread local load imbalances. The definition is a strict one that applies to every subset, not merely a probabilistic one, so no matter where the load imbalances arise, the work can be spread out across a good number of nodes.

It is well-known that a large randomly-chosen graph is an expander graph with high probability [96, 41, 62]. We add the constraints that each apprank has the same number of incident edges, as do the nodes (it is *bipartite biregular*), and generate a random graph according to these constraints. Small graphs are generated using a heuristic-based search or known-optimal solution. For small graphs up to about 32 nodes, we also run some checks to avoid bad graphs, i.e., with limited connectivity, by calculating the vertex isoperimetric number (the minimal value of $1+\epsilon$ in the above equation). Each graph is stored for future executions so that it is only created once.

The number of edges per apprank is a user-provided parameter, known as the *offloading degree.* An offloading degree of one corresponds to the baseline without task offloading. An offloading degree of two means that each apprank can execute tasks on its main node and one additional node. Since the offloading degree is known in advance, as is the assignment of appranks and helper ranks to nodes, the initialization of all Nanos6 instances is done at initialization time. Our results on up to 64 nodes are generally insensitive to the offloading degree, so long as it is large enough to accommodate the application's level of imbalance and to provide enough connectivity given the number of nodes. It can be set as recommended in Section 4.6.3. We have not investigated larger numbers of nodes, but it is known that the graph diameter of a regular graph increases with the number of vertices, so it may be necessary to increase the offloading degree. [52]

Dynamic work spreading: The static approach has a parameter (the offloading degree), which must be provided by the user. Depending on the value of this parameter, a fixed number of helper ranks is created at the beginning of the execution. These helper ranks require at least one core each, leaving fewer resources for the actual computation, even if the helper ranks turn out to not be required. The need to use at least one core per process is part of the design of DLB, which does not support oversubscription of cores [83, Section 5.2] Moreover, the optimal bipartite graph depends on the application and input data. It could also take account of specific communication latencies and thereby depend on the physical topology of the nodes allocated to the application.

A better approach may therefore be to grow the expander graph dynamically. This would allow the execution to adapt to the program and system characteristics, and it would remove the offloading degree parameter. Doing so is compatible with all the contributions in this chapter, and is a natural extension of our work. The main change to the runtime would be to extend it to support dynamic process spawning. Our experience, however, discussed in Section 4.6.3, shows that the benefit would likely not be sufficient to compensate for the extra implementation and evaluation complexity.

4.4.3 Fine-grained load balancing via LeWI

At any time, a fine-grained load imbalance may appear among the workers on a node. LeWI is the mechanism that allows a worker process to lend otherwise idle cores to another process on the same node that can make use of them. As the borrower process completes tasks more quickly, it can be scheduled more tasks from the `main` (or other) process, as explained in Section 4.4.5.

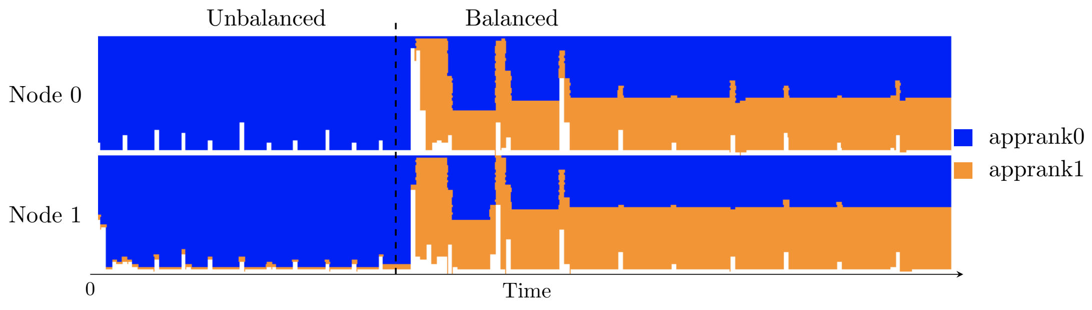
4.4.4 Coarse-grained load balancing via DROM

At any time, each CPU core is owned by one of the appranks or helper ranks executing on that node. At the beginning of the execution, each helper rank owns one core (the minimum possible with DLB), and ownership of the remaining cores is divided equally among the appranks on the node. On MareNostrum 4, for example, which has 48 cores per node, each helper rank of Figure 4.3c starts with one owned core and each apprank starts with 22 owned cores. Ownership of cores is updated dynamically as the execution progresses. We propose two approaches for doing so, a local convergence approach and a global solver approach, both described below.

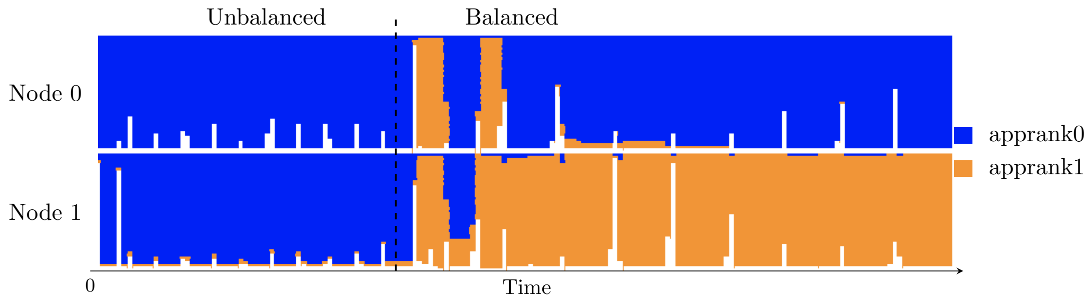
4.4.4.1 Local convergence approach

As the program executes, each apprank's task scheduler (Section 4.4.5) tries to balance the load by exploiting all cores, on all nodes, that are assigned to the apprank. At the same time, the local convergence approach tries to balance the load-per-core among the workers running on each node. To do so, each worker measures its average number of busy cores, i.e., the average number of cores executing tasks or runtime code except the idle loop. The workers on a node coordinate to ensure that all cores have an owner and that the number of cores owned by each worker is proportional to its average number of busy cores. These two processes, one local to the apprank and one local to the node, together try to keep all cores in the system equally busy.

This approach is simple to implement and understand, it has no global communication and low overhead, and it does a good job of balancing the loads among the appranks. It is not guaranteed, however, to minimize the amount of task offloading. Figure 4.4a shows an example with two



(a) Local convergence approach.



(b) Global solver approach.

Figure 4.4: Coarse-grained load balancing: The local approach balances the load but does not minimize task offloading, as both appranks of the balanced kernel execute tasks across both nodes. The global approach balances the load and minimizes task offloading at the cost of global communication.

appranks on two nodes. The x-axis is time and the y-axis shows the number of cores executing for Apprank 0, in blue, and Apprank 1, in orange. Node 0 is shown in the top half of the trace and Node 1 is shown in the bottom half of the trace. The first half of the execution has an unbalanced load: both appranks have 100 tasks per core, but each task of Apprank 0 has duration 50 ms whereas each task of Apprank 1 has duration 0 ms. Almost all computation is therefore on Apprank 0. We see that the local approach almost immediately starts executing the tasks of Apprank 0 on both nodes, making full use of the computational resources.

The second half of the execution, however, has an equally balanced load across the two appranks. Both appranks still have 100 tasks per core, but now all tasks on both appranks have duration 50 ms. Since before this point, both nodes have almost all cores owned by Apprank 0, the first expensive tasks from Apprank 1 must wait for cores to become available. Once the work of Apprank 0 is complete, the tasks of Apprank 1 are scheduled across both nodes. Both nodes see the same pattern of load, and they react in the same way, by increasing the number of cores owned by Apprank 1, to converge towards equal ownership.

The outcome is that Apprank 0 offloads half of its tasks from Node 0 to Node 1 and Apprank 1 offloads half of its tasks from Node 1 to Node 0. It is clearly unnecessary to offload tasks when the load is balanced, and Figure 4.4b shows the optimal approach. It starts in the same way as Figure 4.4a, but once the load becomes balanced, there is no unnecessary offloading of tasks. It was obtained using the global solver approach, described in the next section.

4.4.4.2 Global solver approach

The global approach employs an external solver to determine how many cores should be allocated to each worker process. Similarly to the local approach, it uses the number of busy cores as an estimate of the amount of work, but the work is summed across all workers in the apprank. It then uses a linear program formulation to minimize the value of:

$$\max_{\text{Apprank } a} \frac{\text{Total work on } a}{\text{Total cores on } a}, \tag{4.1}$$

subject to the constraints that each worker owns at least one core, the sum of owned cores on each node is at most the number of physical cores, and that each apprank may only own cores that it is adjacent to in the bipartite expander graph, e.g., in Figure 4.3d. This is an Integer Linear Program (ILP), i.e. a linear program whose variables must all be integers, since the numbers of owned cores must be integers. In general an ILP is NP-complete, but it is sufficient to solve the continuous linear problem and round the solution to an integer number of owned cores per worker that sums to the total number of physical cores. This is always a valid solution, but it may not be the perfectly optimal solution when it is unclear which direction to round to integer. A heuristic counts the cores on the apprank itself as being marginally faster, in order to prefer to not offload unless necessary. We did not find the precise value of this incentive to be critical, as the solver tends to minimize offloading once it has an incentive to do so, even when the magnitude of this numerical incentive is small. We use a value of one part in 10^{-6}, i.e. the "Total work on a" in the numerator of Equation 4.1 is the sum of the non-offloaded work and $1 + 10^{-6}$ times the offloaded work. Since the number of busy cores includes runtime overheads, the global policy is able to converge to a solution that provides additional resources, if necessary, for runtime execution overheads.

The global policy has the advantage that it will always find an optimal solution that balances the load and minimizes task offloading. Its disadvantages are that it requires periodic global communication, and it centralizes the work of determining the core allocation onto a single node. The implementation has a separate Python process using CVXOPT [9], and it executes the global solver every two seconds. We always run the solver on the first node, which in many cases happens to be the highest loaded node. But it could of course be migrated to the least loaded node. The time to solve the global allocation problems for the 32-node experiments in Section 4.6 is approximately 19 ms. Running the solver every two seconds gives an overhead of about 1%.

Since the time to solve the linear program grows approximately quadratically with the size of the graph, larger graphs than 32 nodes should be partitioned and solved in parts on multiple nodes. These 32-node groups are very likely to contain heavily and lightly loaded nodes and allow almost complete load balancing. It is a significant improvement above the existing DLB approach, which only supports load balancing among the processes on a single node.

4.4.5 Task scheduling

To maintain load balance, the scheduler makes a tentative scheduling decision whenever a task becomes ready. If the best node, according to data locality, currently has fewer than two tasks per-core, then the task is scheduled to that node immediately. Otherwise, it there is an alternative node with fewer than two tasks per core, the task will be immediately scheduled to that node instead. Two tasks per core allows one task to be executing and another to have the data transfer (if any) initiated in advance and be blocked ready to execute as soon as the executing task finishes. If all nodes already have at least two tasks per core, then the newly ready task is held in a queue, and will be stolen as tasks complete.

When calculating the number of tasks per core, the number of cores is the number owned via DROM. It does not take account of any short-term lending or borrowing of cores via LeWI. This is because the lent cores are not really gone, as they can easily be reclaimed if needed, while borrowed cores may have to be returned at any moment. Offloading a task is a "final" decision because once a task has been offloaded to a node, it cannot be recalled, and it cannot be rescheduled or migrated to another node. By not taking temporary cores for granted, we ensure that there are always sufficient cores to execute the offloaded tasks in a timely manner.

4.5 Methodology

4.5.1 Quantifying imbalance

We quantify the application's load imbalance using the imbalance:

$$\mathit{Imbalance} = \frac{\text{Maximum}_{apprank} load}{\text{Average}_{apprank} load} \geq 1. \tag{4.2}$$

This metric is dimensionless, and it directly relates the maximum load, which gives a lower bound on the length of the critical path, to the average load, which estimates the length of the critical path with perfect load balance. It is preferable to other metrics, such as the standard deviation of loads, that have no direct connection to the problem to be solved. As formulated the load imbalance ignores any imbalance among the cores due to task scheduling. An imbalance of 1.0 is perfect load balance, whereas an imbalance of 2.0 indicates that the critical path has roughly twice the length that it would have with a perfect load balance. The maximum possible value for the imbalance is the number of appranks, which would correspond to the case where all the work is on one apprank.

4.5.2 Benchmarks

We use two application programs, Alya MicroPP and n-body. Alya MicroPP is a 3D finite element library for microscale solid mechanics in composite materials [86]. The code has unbalanced execution due to the mix of linear and non-linear finite elements. The n-body code [25] is a parallel implementation of Barnes–Hut [159] using Orthogonal Recursive Bisection to equalize the work across the ranks. Both applications are implemented in C++ with parallelization using MPI and OpenMP/OmpSs-2.

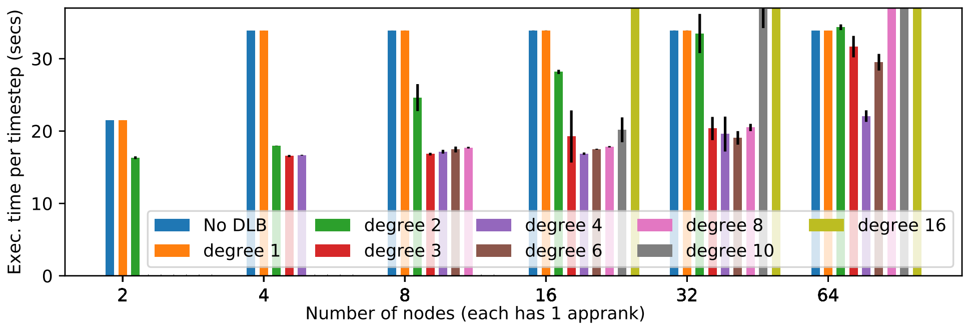
We also use a synthetic benchmark to increase confidence that our approach works for a range of scenarios beyond those found in the applications. The synthetic benchmark has a configurable imbalance (Equation 4.2). Each iteration of the program has 100 tasks per core, of average duration 50 ms. The task durations are different on the different appranks to meet the target imbalance. The execution time of the tasks on the worst-case rank is 50 ms multiplied by the target imbalance. The other execution times are uniformly distributed over the space of values respecting the constraints.

4.5.3 Hardware platform

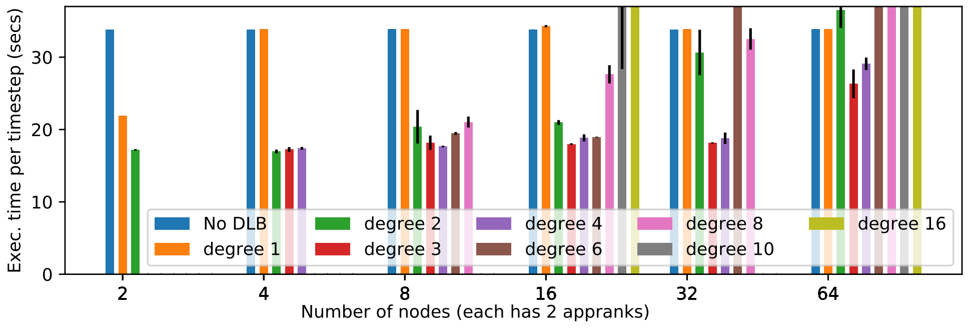
Most experiments were performed on up to 64 nodes of the general-purpose block of the MareNostrum 4 supercomputer [17]. MareNostrum 4 comprises 3456 compute nodes, each with two 24-core Intel Xeon Platinum sockets. We use normal memory capacity nodes, which have 96 GB physical memory (2 GB per core). The interconnect is 100 Gb Intel Omni-Path with a full-fat tree. The experiments with a slow node were performed on Nord3 [20], which has a newer version of Slurm that supports heterogeneous allocations, as needed to run different nodes of the same job at different clock frequencies. Nord3 has 756 compute nodes, each with two 8-core Intel E5-2670 SandyBridge sockets at 3.0 GHz (normal) or 1.8 GHz (slow).

4.6 Results

4.6.1 Application performance



(a) MicroPP on MareNostrum 4 (1 apprank per node)



(b) MicroPP on MareNostrum 4 (2 appranks per node)

Figure 4.5: MicroPP application performance with global allocation policy.

Figure 4.5 shows the results for MicroPP on two to 64 nodes of MareNostrum 4. Figure 4.5a has one apprank per node and Figure 4.5b has two appranks per node, i.e. 4 to 128 appranks. The baseline result (blue) is without task offloading or DLB. When there is just one apprank per node, single-node DLB makes no difference, as expected. When there are two appranks per node, the benefit from DLB alone (degree one) is also generally small, because some heavily loaded ranks share a node. All baseline and degree one results have no helper ranks, so all cores are used for computation.

Enabling task offloading via OmpSs-2@Cluster, however, makes a large improvement to performance in all situations. Assuming a moderate offloading degree of four, the time-to-solution is reduced by 49% on 4 nodes and 47% on 32 nodes, compared with DLB. Both are close to the perfect load balancing. Offload to a single extra node (degree two) has good results for small numbers of nodes, but as the number of nodes increases the limited graph connectivity becomes a constraint on the ability to balance the load. An offloading degree of three or four provides good results in all situations. Increasing the offloading degree to an excessive value, of eight or more, starts to affect performance, justifying the design decision to use few helper ranks per apprank.

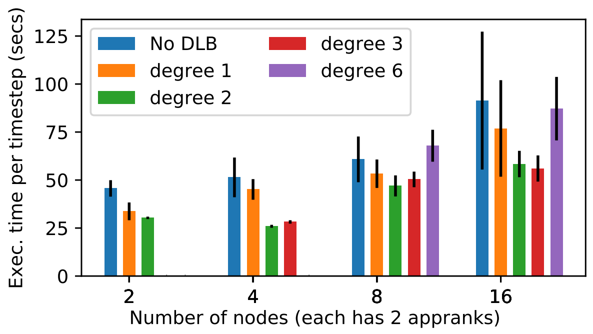


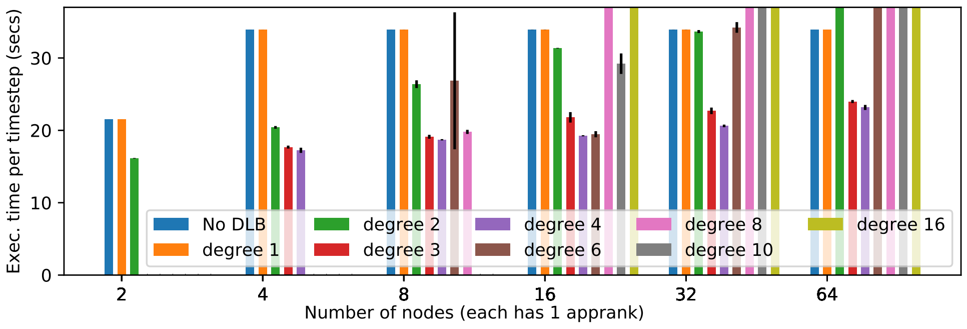
Figure 4.6: n-body on Nord3 with one slow node (2 appranks per node) and global allocation policy.

Figure 4.6 shows the results for n-body on Nord3 with one slow node. n-body is in itself a balanced application, as it uses ORB each timestep to rebalance the work. In this example with a slow node, however, ORB does not perform well. Here, on 16 nodes and two appranks per node, we see a 16% improvement when employing single-node DLB and a further 20% improvement (with respect to the same baseline) when enabling task offloading with degree 3. The single apprank per node results are not shown as the baseline performance was much worse, likely due to an issue with scheduling tasks across the two Non-uniform Memory Access (NUMA) nodes. Nord3 has an older CPU architecture, with 16 cores per node, so with two appranks per node, the offloading degree should be at most four. From both applications, the conclusion is that the global core allocation achieves excellent improvements in load balance, with an offloading degree of four.

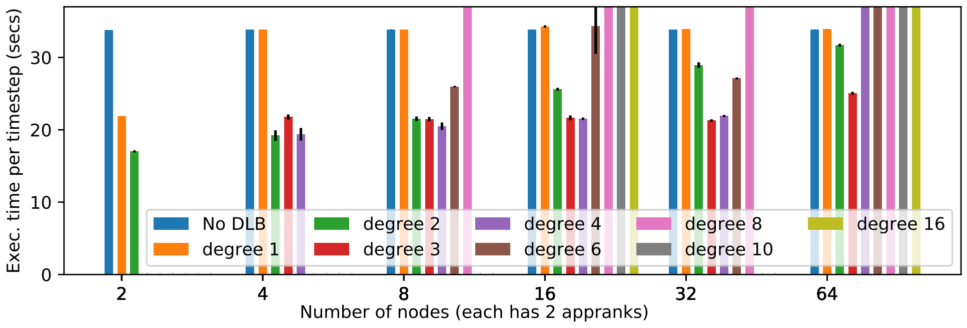
4.6.2 Local core allocation

The local policy has about 10% higher optimal execution time for MicroPP on 32 nodes than the global policy, and it is generally more sensitive to the offloading degree (number of nodes among which each apprank can execute tasks).

Figures 4.7 and 4.8 show the results using the local allocation policy. Overall, local allocation performs slightly worse than global allocation due to the issue with unnecessary task offloading (see: Section 4.4.4.1). We see that the local policy provides similar results for small numbers of nodes: reducing the time for two appranks per node on 4 nodes by 43%. But the local policy tends to offload too many tasks, and on 32 nodes the improvement is only 38%, compared with



(a) MicroPP on MareNostrum 4 (1 apprank per node)



(b) MicroPP on MareNostrum 4 (2 appranks per node)

Figure 4.7: MicroPP and n-body performance with local allocation policy.

47% found above for the global policy. It also tends to be more sensitive to a well-tuned offloading degree, with execution time rising for an offloading degree above 4 for MicroPP.

4.6.3 Sensitivity analysis on kernel imbalance

Figure 4.9 is a sweep of the execution time, as a function of the imbalance, for the synthetic test program (see: Section 4.5.2). The x-axis is the imbalance, defined in Equation 4.2, and the y-axis is the execution time per iteration, in seconds. In all three subplots, the baseline MPI+OmpSs-2 program with single-node DLB is indicated with the blue line. Since there is one apprank per node, there is no benefit from single-node DLB, so the case without DLB is not shown.

When the offloading degree is 1, there is no automatic load balancing. The execution time is dominated by the most heavily loaded node, which is the same as the most heavily loaded apprank. If the imbalance on the x-axis is I, then using a simple model, the average amount of work per apprank is 5 seconds but the amount of work on the most heavily loaded apprank is $5I$ seconds. The bottleneck is the most heavily loaded node, so the overall execution time should be proportional to the imbalance, which is what is seen in all four subplots. When the offloading degree is 2, the most heavily loaded node can offload tasks to one other node. The runtime can get close to a perfect load balance until the imbalance reaches 2. At that point, the work from the most heavily loaded apprank can be spread across 2 nodes, each of which will have at least $5I/2$ units of work, being half the work assigned to the worst case apprank. This assumes that the runtime has moved the rest of the work away from these two nodes to get perfect load balancing given the known offloading degree. The ideal outcome is therefore a flat execution time of 5 seconds until

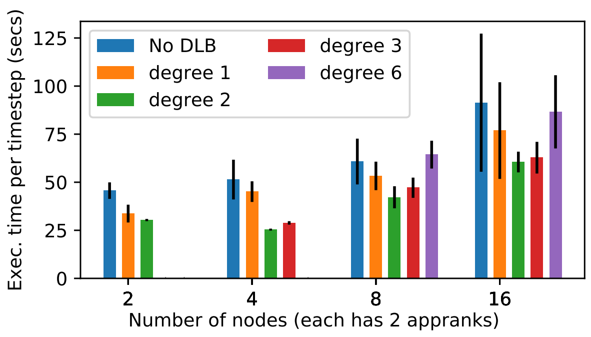


Figure 4.8: n**-body on Nord3 with one slow node (2 appranks per node) and local allocation policy.**

the imbalance reaches 2, at which point the execution time increases at half the slope that was seen for an offloading degree of 1. A similar argument indicates that the best possible outcome for an offloading degree of d is a flat execution time until the imbalance reaches d, at which point the execution time starts increasing at a rate of $1/d$ times that of an offloading degree of 1.

Figure 4.9 show that, on four to 64 nodes, an offloading degree of four (red) provides the best results across the whole range of application imbalance from 1.0 to 4.0. This is similar to MicroPP, where an offloading degree of four also generally provided the best results. For small numbers of nodes, up to about eight nodes, it is sufficient for the offloading degree to be at least as large as the imbalance. This is clearly seen in Figure 4.9a, on four nodes, where an offloading degree of 2 is sufficient for up to an imbalance of 2.0 and an offloading degree of 3 is sufficient for an imbalance up to 3.0. But there is no penalty for a moderately higher offloading degree.

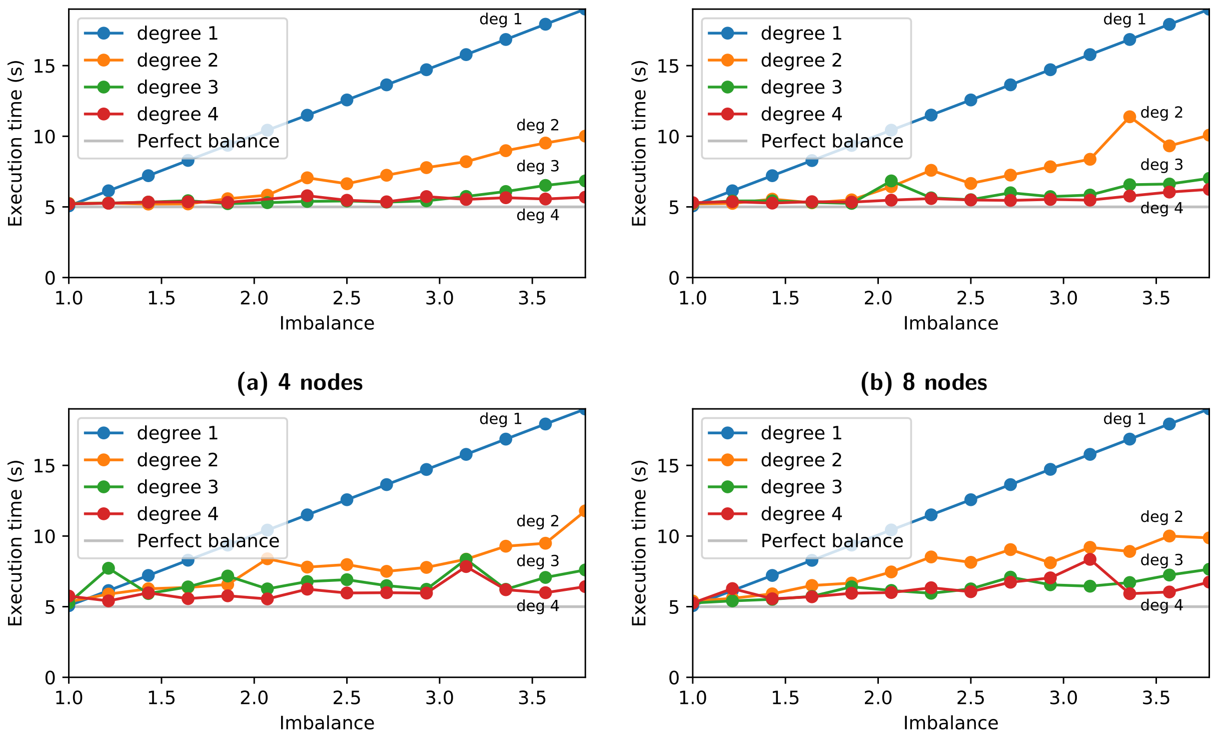
For larger numbers of nodes, the graph connectivity becomes an issue. On 64 nodes, an offloading degree of 4 provides the most dependable results even for small levels of imbalance, and is within 20% of optimal for imbalances in the range 1.0 to 2.0. The conclusion is that for up to 64 nodes, an offloading degree of four is sufficient, which is dramatically lower than full connectivity (an offloading degree of 64). The fact that there is no benefit for smaller offloading degrees when the imbalance is small supports our claim that a static expander graph is sufficient (see: Figure 4.4.2).

4.6.4 Fine-grain (LeWI) and coarse-grain (DROM)

In order to understand the role of LeWI and DROM, Figure 4.10 shows Paraver/Extrae execution traces for MicroPP with and without LeWI and DROM, with four appranks on four nodes. The outcome is similar for larger numbers of nodes, but this example gives more intelligible traces. The x-axis is time, and all timelines have the same scale, so the length of the trace is proportional to execution time. The y-axis indicates the number of cores for each apprank (colours), busy executing tasks or non-idle-loop runtime code (left-hand traces) or owned (right-hand traces). The four nodes are shown from top to bottom.

Figure 4.10a shows the original MPI+OmpSs-2 trace, in which there is a clear imbalance among the nodes, due to the greater amount of work done by Apprank 0. In fact, the imbalance as defined above equals 2.0 We see in Figure 4.10b that each apprank owns the cores on its node.

Figure 4.10c employs LeWI, but not DROM. In this case, once Apprank 1 finishes an iteration, LeWI reacts to the fine-grained load imbalance by allowing tasks from Apprank 0 to be offloaded



(a) 4 nodes **(b) 8 nodes**

(c) 32 nodes **(d) 64 nodes**

Figure 4.9: Execution time of synthetic application, with one apprank per node, using LeWI and DROM, as a function of the imbalance. An offloading degree of four provides consistently good results on up to 64 nodes.

to Node 1, and Apprank 0 executes its tasks across both nodes.

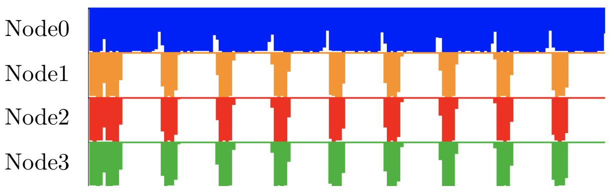
The use of remote cores when temporarily borrowing cores is well under 100% due to the issue described in Section 4.4.5. It is important not to be too aggressive when making use of borrowed cores, because the cores could be reclaimed at any moment. In fact, Apprank 1 reclaims its cores at the beginning of the next iteration.

Figure 4.10d shows the same static assignment of core ownership, since LeWI does not change the ownership of cores. Overall, the imbalance is reduced to 1.33 and the execution time is reduced to 83% of that of the baseline.

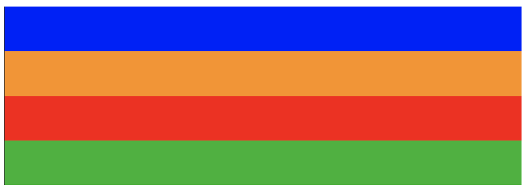
Figure 4.10e employs DROM rather than LeWI. DROM updates the ownership of cores to ensure load balance and high utilization in the later iterations. After a few iterations, almost all cores on Node 0 and Node 1 are consistently used by Apprank 0. In order to let this happen, Apprank 1 shifts its work to Node 2. The trace shown in Figure 4.10e uses the global core allocation policy, but the same effect occurs with the local policy. We see that the imbalance is reduced to 1.15 and the execution time is reduced to 65% of that of the baseline execution. Figure 4.10f shows how the core ownership converges to this optimal result.

Finally, Figure 4.10g employs both LeWI and DROM. LeWI is active in the first iteration, by reacting to the load imbalance immediately. But DROM quickly adjusts to the steady-state imbalance, ensuring an optimal load balance in later iterations. This shows how LeWI and DROM complement each other and together give the best execution.

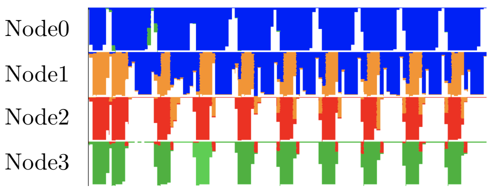
Surprisingly, in this example, the execution using both LeWI and DROM has a higher imbalance,



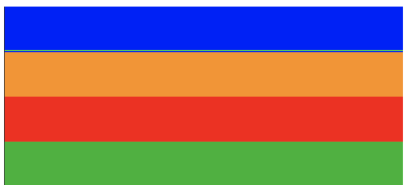
(a) No offload: MicroPP timeline of busy cores (Original imbalance=2.0; time: 79.8 s: 100%)



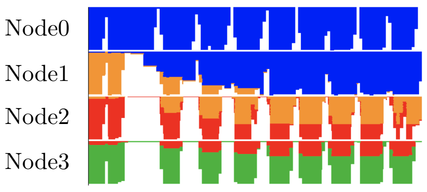
(b) No offload: MicroPP timeline of owned cores



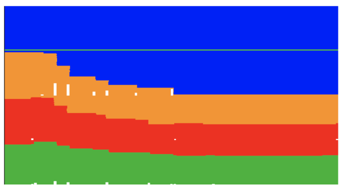
(c) Using LeWI but not DROM: busy cores (Imbalance=1.33; time: 66.6 s, 83%)



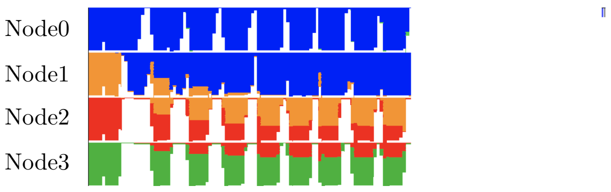
(d) Using LeWI but not DROM: owned cores



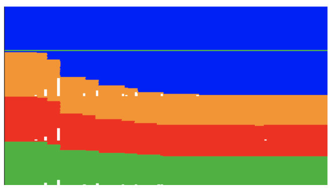
(e) Using DROM but not LeWI: busy cores (Imbalance=1.15; time: 51.6 s, 65%)



(f) Using DROM but not LeWI: owned cores



(g) Using LeWI and DROM: busy cores (Imbalance among nodes is now 1.22; time: 49.9 s: 63%)



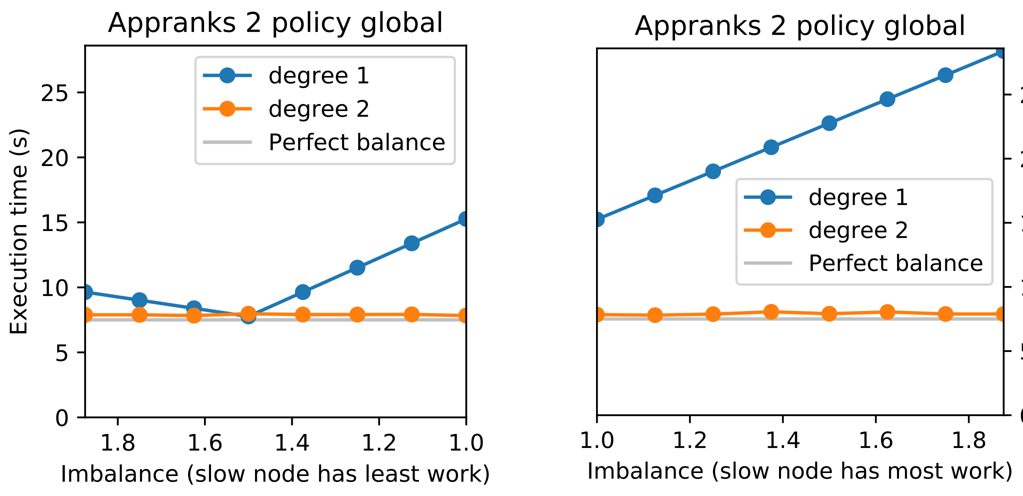
(h) Using LeWI and DROM: owned cores

Figure 4.10: Paraver/Extrae traces of Alya MicroPP on four nodes with degree two. LeWI allows Apprank 0 to borrow cores on Node 1 when they would otherwise be idle. DROM adapts the long-term ownership of cores to address the steady load imbalance.

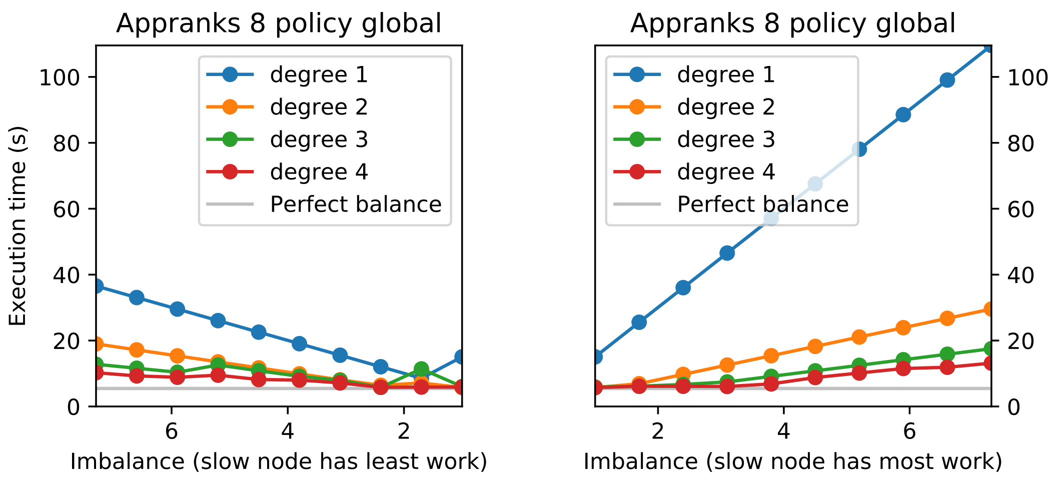
of 1.22, than using only DROM, for which the imbalance was 1.15. This is because the imbalance is not a perfect measure. In Figure 4.10e, the most heavily loaded node happens to be node 1. In Figure 4.10g, the execution is similar except that more tasks are offloaded to node 1 near the beginning of the execution, when node 1 would otherwise be mostly idle. Overall, the effect is to decrease the execution time, even though, for this example, more work is done on the overall most heavily loaded node.

4.6.5 Sensitivity analysis with slow node

Figure 4.11 shows a sweep of the execution time as a function of the imbalance, for a synthetic test case with one slow node that is three times slower than the other nodes. Being a synthetic example unlike n-body, it is not actually a slow node, just emulated by the task durations. The y-axis is the execution time per iteration, and the x-axis is the imbalance. Increasing imbalance to



(a) 2 nodes



(b) 8 nodes

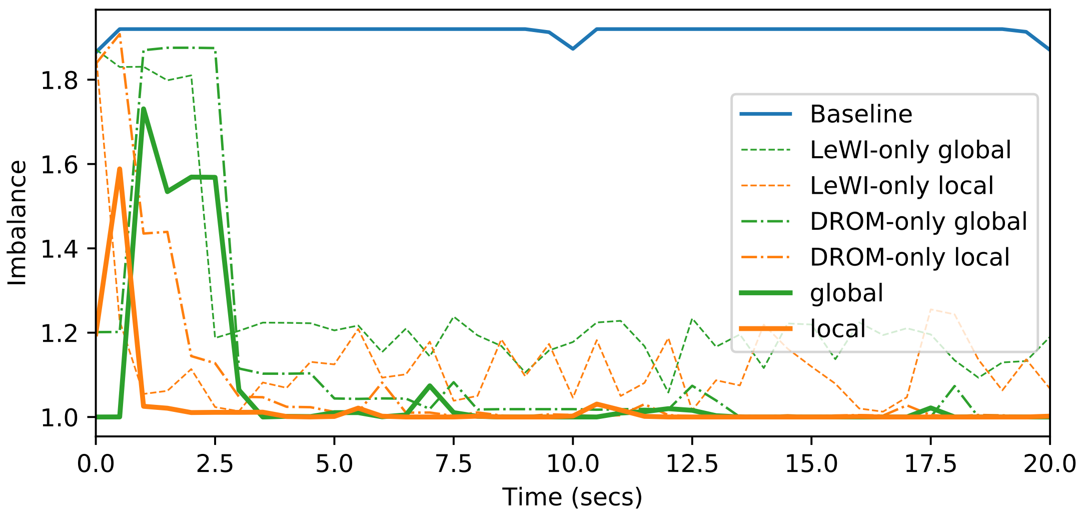
Figure 4.11: Synthetic test program with one emulated slow node (3 times slower than the other nodes), showing that tasks are offloaded to balance the loads.

the left indicates the case where the slow node has the *least* work and increasing imbalance to the right indicates that the slow node has the *most* work.

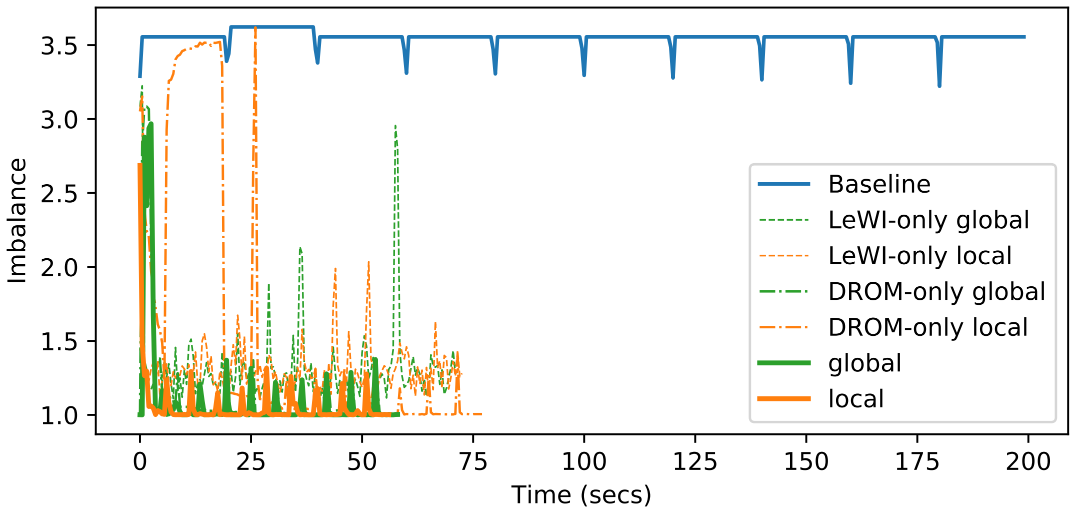
Figure 4.11a has two nodes, and a degree of 2 has almost flat execution time per iteration, close to the optimal (grey line) across the whole range of imbalance. With eight nodes, we see a similar story to before. When the slow node has the most work (to the right), the execution time is close to flat, as long as the offloading degree is a little higher than the imbalance. As before, we see that on offloading degree of four provides the best and most consistent results, and it is able to handle the imbalance up to an imbalance of 4.0.

4.6.6 Convergence with synthetic benchmark

Figure 4.12 shows time series plots for the synthetic benchmark: two nodes with an imbalance of 2.0 and four nodes with an imbalance of 4.0. The x-axis is time and the y-axis is the imbalance



(a) 2 nodes, imbalance = 2.0 (degree two)



(b) 4 nodes, imbalance = 4.0 (degree four)

Figure 4.12: Convergence of imbalance among nodes for synthetic benchmark. Local converges faster than global when LeWI is enabled. DROM is essential to reduce the node imbalance to close to 1.0.

among the nodes, given by (Maximum$_{node}$ *load*)/ (Average$_{node}$ *load*). The current load is the total average number of busy cores (see: Section 4.4.4), meaning that the imbalance among nodes is updated more frequently than application-level measurements.

We see a similar picture in both subplots of Figure 4.12. Both the local and global policies, when using DROM (with or without LeWI) are able to reduce the imbalance among the nodes to close to 1.0. Using LeWI but not DROM, the imbalance fluctuates around 1.2 in both scenarios, which is consistent with the behaviour observed in the MicroPP traces (see: Section 4.6.4).

The local policy converges quicker than the global policy, as it operates continuously whereas the global policy is updated every two seconds. We also see that LeWI helps to accelerate the speed of convergence of the local policy, in both scenarios, since cores are allocated to a helper rank in proportion to the average number of busy cores. By offloading tasks in reaction to a fine-grained load imbalance, LeWI helps to accelerate core usage. LeWI does not accelerate the speed of convergence of the global policy, as the solver responds to the total work on the apprank, but LeWI does reduce the peak near the beginning of the application, in Figure 4.12b. Overall we again see that the combination of LeWI and DROM provides the best results.

4.7 Conclusion

Load balancing has been an important concern of high-performance computing for a long time.

We introduce an automatic and transparent load balancing mechanism for MPI+OmpSs-2 programs, which relieves the application programmer from the burden of load balancing. We keep the programming model as simple as possible, with only minor local changes being needed for existing MPI+OmpSs-2 programs.

Our method uses OmpSs-2@Cluster to offload tasks to other nodes, and it employs DLB as the underlying mechanism to allocate the resources (cores) on each node. We leverage the DROM module of DLB to reserve cores on other nodes and thereby address coarse-grained load imbalances, and use the LeWI module to react to fine-grained imbalances.

Our system uses a sparse expander graph to minimize the amount of point-to-point communication and state. We show 46% reduction in time-to-solution for MicroPP solid mechanics on 32 nodes and 20% reduction beyond DLB for n-body on 16 nodes, when one node is slow. We perform a sensitivity analysis on imbalance, and obtain results within 10% of perfect load balancing for an imbalance of up to 2.0 on 8 nodes.

Chapter 5

Malleability

5.1 Introduction

An important issue regarding the usage of a supercomputer is the efficient utilization of its available resources. Several studies [85, 156, 70] show a suboptimal utilization of resources in many of the TOP500 supercomputers [179]. In general, the users over-provision their jobs, i.e., they request more resources than the application can efficiently utilize or only use a small portion of the total execution time.

This reduces the overall throughput of the system since applications are queued waiting for resources that are needlessly reserved by that problematic application.

There are new kinds of popular application domains, such as real-time applications or interactive applications that impose new challenges to the batch schedulers [48, 49, 6]. Real-time applications should be scheduled for execution as soon as possible, whereas interactive applications often require modifying the allocation of resources at runtime. Although traditional High-Performance Computing (HPC) applications are *moldable* by design, i.e., we can select the number of resources at submission time, they are not *malleable*, in that this allocation cannot be modified once the application has started (see classification in page 20). As a result, it is hard for the batch scheduler to accommodate the needs of the more dynamic sorts of applications that seem to attract a lot of attention within the HPC community.

HPC applications are normally inflexible. The classical/legacy Message Passing Interface (MPI) paradigm means that once an application starts running, it exclusively occupies a fixed number of nodes. As a side effect, it means that all other resources like accelerators or Graphical Processor Units (GPUs) on those nodes are also occupied during the entire execution of the application. This may not be considered an issue if all parts of the job scale perfectly, for the fixed problem size, since the application will always be able to make effective use of its resources. In practice, however, jobs may underutilize their allocated resources for all or much of their execution time.

Malleability is the ability to dynamically change the data size and number of computational entities in an application. It can be used by middleware to autonomously reconfigure an application in response to dynamic changes in resource availability in an architecture-aware manner, allowing applications to optimize the use of multiple processors and diverse memory hierarchies in heterogeneous environments [59].

Iserte et al. [104] define malleable workloads as a combination of four components working together:

1. User applications with support for on-the-fly scale-up/down (process malleability).
2. Parallel Distributed Runtime (PDR) responsible for re-scaling the jobs.
3. Resource Management System (RMS) with the necessary logic to reallocate resources considering the cluster status.
4. Communication mechanism that allows 1 and 2.

This chapter explains how the OmpSs-2@Cluster programming model inherently supports the development of transparent malleable parallel applications. An OmpSs-2@Cluster application naturally supports process malleability (Actor: 1) because the virtual memory layout is independent of the resource allocation and all data distribution and task scheduling are delegated to the runtime. Nanos6 performs the role of the malleable PDR, which interacts with Slurm (Actor: 3) to allocate or release the resources and MPI to execute the new processes (Actor: 4).

In OmpSs-2@Cluster We implement support for two reconfiguration mechanisms optimized at the runtime level.

1. A semi-transparent Checkpoint and Restart (C/R) mechanism implemented using MPI-IO Application Programming Interface (API). This is a baseline implementation for the purposes of comparison.
2. Transparent malleability with automatic in-memory data migration based on the internal OmpSs-2@Cluster data locality, affinity and communication.

We focus on the underlying support for malleability, including interaction with the MPI library and Resource Management System (RMS) and data redistribution. Intelligent mechanisms to decide when to spawn or release nodes can build upon the work in this chapter. Such future work should optimize resource utilization or minimize wallclock time. A cost model may be required in order to compare the overheads of spawning nodes with the performance benefit and potentially the cost of additional resources.

This chapter starts by describing in detail (Section 5.2) the MPI features used for the in both cases with a brief description of the Slurm features for the integration. The next sections describe our implementations of C/R (Section 5.3) and Dynamic Process Management (DPM) (Section 5.4) and Slurm integration (Section 5.5) on Nanos6. The main contributions of this work are the different policies to create processes and migrate data with a transparent interface to automate and abstract the most complex part of data migrations, Input Output (IO) operations and RMS interaction while fitting comfortably in the programming model, optimizing the process and requiring minimal modifications in the user code.

5.2 MPI features and requirements

The current implementation of OmpSs-2@Cluster uses MPI for communication (see Section 3.5.1). In the future, it may be possible to use other communication interfaces and libraries in order to improve performance or use more advanced communication patterns. However, it is simple and cleaner to use the default MPI features when possible to establish a baseline to improve in the future.

We developed two malleability approaches that rely on pure MPI features without external dependencies or extra configurations. Consequently, our implementation has the same constraints and advantages that the underlying MPI functionality and performance will also depend on the optimization in the MPI implementation.

5.2.1 MPI characteristics for Checkpoint and Restart

The actual checkpointing API relies on the collective parallel IO basic functions based on OmpSs-2@Cluster memory regions and creates the checkpoints based on the memory data and agnostic to the data types in the memory region.

For distributed memory regions (see Section 3.3.5 and Section 3.5.2), it is more efficient to perform the checkpointing based on the distributed allocation policy. Which allows using collective `MPI_File_open` in a synchronous way and then read/write the data with `MPI_File_read_at` or `MPI_-File_write_at`. Depending on the data distribution, it is possible to use some optimized versions of these functions to obtain some performance improvement in the IO operation.

We use the generalized `MPI_BYTE` data type for the IO operation because the C/R interface does not require the user to provide any hint about the underlying data type. However, as the MPI interface only accepts integers and `MPI_BYTE` is a single-byte data type, data transfers need to be split into units of less than 2^{31} bytes, i.e. of 2 GB. In the initial benchmarks, we did not observe any performance penalty related to this choice.

Figure 5.1 shows the basic schema to perform checkpoint and restart with MPI. The example performs this to spawn a new process and pass from 2 processes to 3. The `MPI_File_open` and `MPI_File_close` are collective operations. The example in the graph use the MPI API functions: `MPI_File_write_at` and `MPI_File_read_at`; these functions can be substituted by other alternatives optimized for specific use cases depending on the distribution and layout of the data to checkpoint and restart. But these in the example are the most general because they work even without the layout information.

One of the costly parts of the C/R process is the process finalization and re-initialization. This is not intended to be very different from spawning a new process (or group of processes) from MPI; because the `MPI_Comm_Spawn` function generally uses the same functionalities as `mpirun` or `srun`.

An important detail from MPI-IO is that the collective operations require that all the processes involved in the communicator call it at the same time and in the same order. This may be an issue in highly parallel systems with `MPI_THREAD_MULTIPLE` thread support level. OmpSs-2@Cluster is optimized to minimize overhead and overlap data transfers with IO operations. The implementation is detailed in Section 5.3.2.

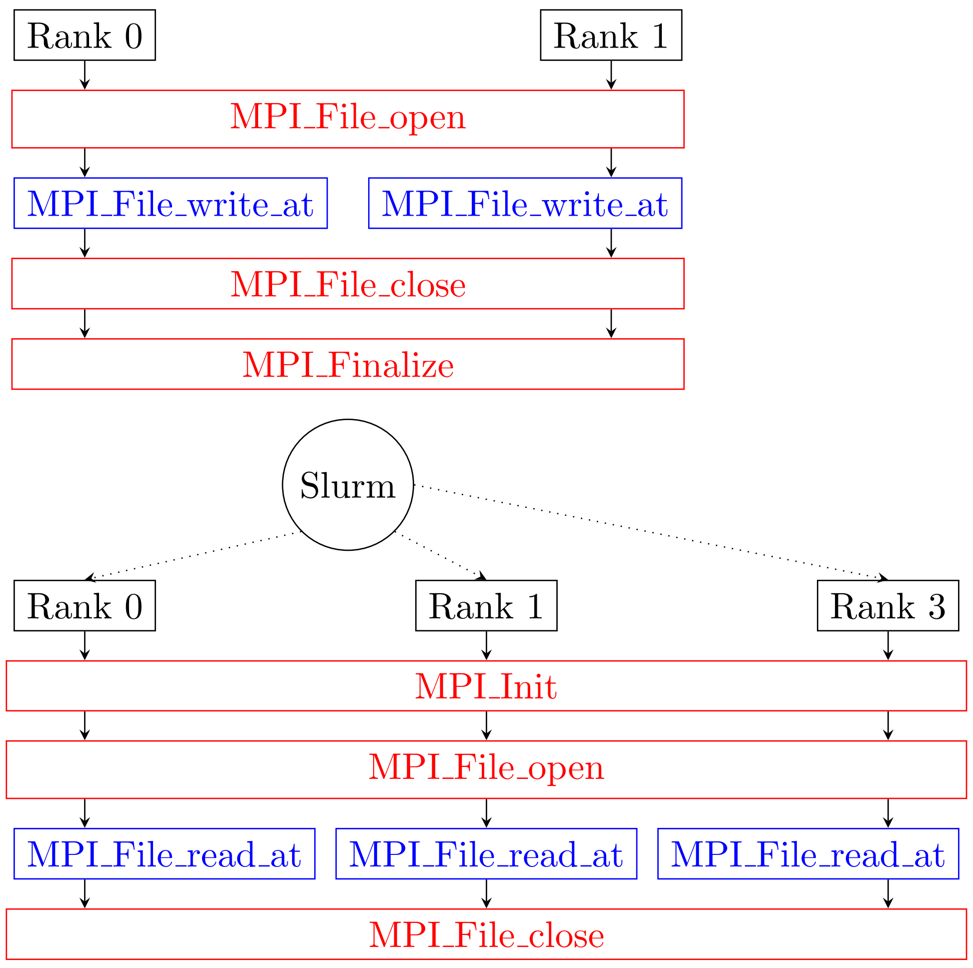


Figure 5.1: OmpSs-2@Cluster checkpoint and restart.

5.2.2 MPI features for Dynamic Process Manager

MPI introduced DPM in MPI-2 standard with the addition of the functions `MPI_Comm_spawn` and `MPI_Comm_spawn_multiple`. Such functions create another `MPI_COMM_WORLD` parallel to the existing one. And an extra communicator to send messages between the "old" and "new" groups of processes. DPM is the key feature to perform in-memory process reconfiguration (see Section 2.3)

Figure 5.2 shows the basic schema and the inner working of DPM on MPI with `MPI_Comm_spawn` and `MPI_Comm_merge`. The key details in Figure 5.2 are:

- Initially, all the processes in the initial communicator (ranks 0,1 and 2) call `MPI_Comm_spawn`, which is a blocking collective operation.
- The `MPI_Comm_spawn` function communicates with the RMS to start the new processes. Generally, this is implemented in `MPI_Comm_spawn` internally using the same mechanism that started the initial processes (`srun`, `mpirun` or `mpiexec`).
- The RMS (Slurm in the picture) starts the new processes which collectively call `MPI_Init` and create a new `MPI_COMM_WORLD` where they have new indices starting from zero.
- The `MPI_Comm_spawn` does not return until the `MPI_Init` call in the new processes complete.

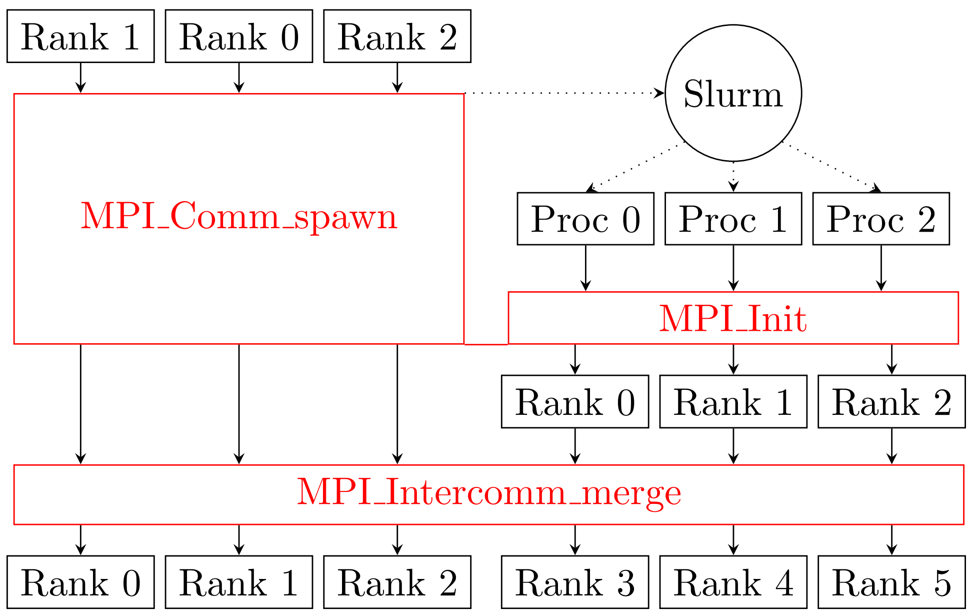


Figure 5.2: OmpSs-2@Cluster spawn and merge.

- When `MPI_Comm_spawn` and `MPI_Init` complete there are two independent “worlds” (the new and old groups have separate `MPI_COMM_WORLD`s. These worlds are connected with an inter-communicator, but they are otherwise totally independent.

- All processes the two communicator call `MPI_Intercomm_merge` to create a new communicator where the processes from the two communicators will have sequential ranks.

The main advantage of the `MPI_Comm_spawn` API is its flexibility to execute different applications. Most MPI implementations have some integration with the RMS to create the new processes in the right hosts according to the system policy. The function also receives an `MPI_Info` object with hints about the process creation.

By definition `MPI_Comm_spawn` is a collective blocking operation. The creation of new processes does not start until all the ranks in the parent communicator call the function and do not return until all the processes in the new communicator finalize the call to `MPI_Init`. The new processes are created in a new `MPI_COMM_WORLD` parallel and totally independent of the “parent” one. The function provides a new inter-communicator to send messages between the “old” and “new” group of processes.

Regrettably, the MPI specifications do not specify any standard or portable method to interact with the RMS. `MPI_Comm_spawn` accepts an `MPI_Info` object as an argument to provide some job creation hints, but the values and formats are implementation specific.

Although only the master node executes the `main` function (as a task), OmpSs-2@Cluster is not a master–slave system. All the worker processes can independently offload work to any other node, perform direct data transfers or propagate dependencies. This simple all-to-all communication schema requires transparent communication among the ranks that `MPI_Comm_spawn` does not provide alone.

Generally, all the MPI libraries communicate with the RMS as some point to create the new processes respecting the application policies. When the application is in a cluster with Slurm as the RMS the most common approach is to make `MPI_Comm_spawn` use `srun` as a backend.

The MPI specification also provides the `MPI_Intercomm_merge` call to create a new intra-communicator merging two independent intra-communicators connected through an inter-communicator. This function is also a collective operation for all the processes involved. `MPI_Intercomm_merge` is generally not a cheap function in terms of performance, but compared to `MPI_Comm_spawn`, its cost is negligible.

Malleability requires not only adding new processes and hosts to the application but also remove them.

One of the well-defined characteristics in the MPI standard is the immutability of the `MPI_COMM_WORLD` as a global and constant variable existing in all the MPI processes. It is not possible to apply `MPI_Comm_split` or any modification to `MPI_COMM_WORLD`. Furthermore, `MPI_Finalize` is a collective operation over the `MPI_COMM_WORLD` communicator. This means that all the processes created in a call to `MPI_Comm_spawn` need to call `MPI_Finalize` together in order to release the resources to the system and remove them from the parallel application.

To release spawned processes and return from `MPI_Finalize` it is also necessary to disconnect the inter-communicator created by `MPI_Comm_split`. `MPI_Comm_disconnect` is a collective operation that waits for all pending communication to complete internally, deallocates the communicator object, and sets the handle to `MPI_COMM_NULL`. `MPI_Comm_disconnect` acts like a release of the processes from the parent side.

The new MPI-4 standard introduces the sessions model to add more flexibility and dynamism to DPM and solve some issues mentioned above with `MPI_Init` and `MPI_Finalize`. However, in spite of the MPI sessions concept has been under development for some years [95, 94] it was introduced in the MPI standard recently [76] and is not fully supported, experimental or under development in the MPI libraries/implementations available at the moment.

5.2.3 Slurm features and requirements

Slurm is an open-source, fault-tolerant, and highly scalable cluster management and job scheduling system for large and small Linux clusters. Slurm requires no kernel modifications for its operation and is relatively self-contained [132].

Slurm has a centralized manager, `slurmctld`, to monitor resources and work. There may also be a backup manager to assume those responsibilities in the event of failure. Each compute server (node) has a `slurmd` daemon, which can be compared to a remote shell: It waits for work, executes that work, returns status, and waits for more work. The `slurmd` daemons provide fault-tolerant and hierarchical communications.

There are several alternatives to interact with Slurm in order to inform, collect information, request or release resources.

- The simplest and most well-known way to interact with Slurm is through the commands such as `scontrol`, `srun` and `sinfo` [131].
- There is an optional Slurm REST API Daemon (slurmrestd) useful to interact with Slurm

through a Representational State Transfer (REST) API [130].

- The C API exposed through the `slurm.h` header file and `libslurm` library [128].
- Pyslurm is a non-official python interface for Slurm that uses `libslurm` as a backend [153].

In general Slurm is self-contained, but many features and components are optional. Optional plugins can be used for accounting, advanced reservation, gang scheduling (time-sharing for parallel jobs), backfill scheduling, topology optimized resource selection, resource limits by user or bank account, and sophisticated multifactor job prioritization algorithms. There is also an API to develop Slurm plugins [129, 102].

To create interaction between Nanos6 and Slurm the most useful alternative is to use the C API. The key function, in this case, is `slurm_update_job` which allows changing a job parameter, including its resources.

When the application requires more resources, a new dependent job needs to be submitted with `slurm_allocate_resources`. This function will fail or return an error if the job can't be submitted or if any dependency is incorrect (i.e. job expansion is not permitted). The functions provide a handler that can be checked with `slurm_allocation_lookup` to determine if the resources were finally granted.

It is important to keep in mind that many actions are generally restricted in production clusters, the user/partition permissions or system hardware. Frequently those variables are out of the user and application control, and the application needs to do all possible checks in advance to avoid hangs or undesired errors. To perform these checks, Slurm sets some information in the environment when a job starts, and those environment variables can be used together with API functions.

For malleability purposes, there is a special configuration option that needs to be enabled in order to expand the jobs: `permit_job_expansion`. Such an option was added since version 19.05 when running job expansion was disabled by default; it is disabled in most of the systems around. There is no restriction or special requirement to release resources from a running job.

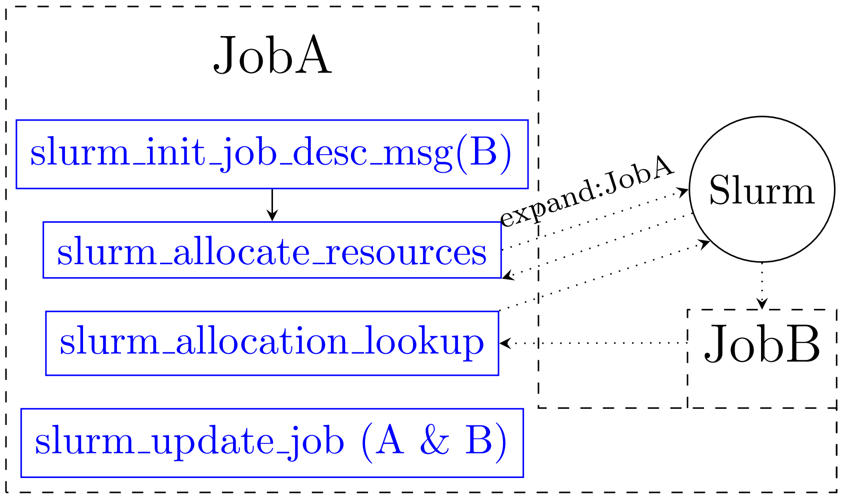


Figure 5.3: OmpSs-2@Cluster spawn scheme using Slurm.

The schema in Figure 5.3 shows the basic steps to extend a running Slurm job with more resources (nodes).

- `slurm_init_job_desc_msg` creates a new job to submit a with an extra `expand` dependency.
- `slurm_allocate_resources` submits the allocation request, but it is just a non-blocking call. When the resources are available immediate allocation can occur. In that case `slurm_allocate_resources` already returns all the information about the new job and `slurm_allocation_lookup` is unneeded.
- The queue waiting time (or immediate allocation) is decided by Slurm based on resource availability and the system configuration. The application's programmer is responsible for creating strategies to handle all situations according to their needs.
- `slurm_allocation_lookup` only checks the allocation status, and it is also a non-blocking call that can be used in a loop with minimal cost and without blocking a thread which is perfect for a polling task.
- There is another useful API function `slurm_job_will_run` which determines if a job would be immediately initiated if submitted at the moment. We don't use this function for the moment, but it is the simpler and more efficient way to implement a sort of "try to spawn" policy for dynamic malleability in high availability clusters.
- The Slurm API also exposes some features to collect information about the system, partitions and current allocations, but some of them are restricted to specially authorized users.

5.3 OmpSs-2@Cluster Checkpoint/Restart (C/R)

We added a straightforward semi-transparent checkpoint and restart interface to OmpSs-2@Cluster. It is currently supported by MPI-IO, but the design is not specific to MPI or MPI-IO, and backends for other communication or IO libraries could be implemented. It could also potentially be extended into a complete runtime implementation of C/R for resilience (see: Section 6.2.5).

Due to the highly parallel nature of the programming model OmpSs-2@Cluster relies mostly on MPI point-to-point operations. Coordinating and finding opportunities to use collective operations may improve performance in some application patterns; it is currently an open research field.

The simplest and more natural approach to create a checkpoint for distributed memory in OmpSs-2@Cluster is to rely on tasks and the dependency system to execute the IO operations. Such approach fits perfectly in the programming model, guarantee correctness, load balance in the IO operation but also ensures there are no deadlocks in the synchronization.

The mechanism used in this section is also a proof of concept on how to coordinate collective operations in OmpSs-2@Cluster without expensive taskwaits or blocking workflows.

5.3.1 OmpSs-2@Cluster C/R API

Figure 5.4 shows a simplified OmpSs-2@Cluster application using the new C/R interface. The program follows a similar pattern as that used with common C/R libraries such as FTI [27]. The application data is allocated and used in the same way as it would be without C/R. In this case, a single array is allocated by the call to `nanos6_dmalloc`. Its original data distribution follows the equipartition data distribution affinity hint, but, as for any OmpSs-2@Cluster application, the data distribution may change over the course of the execution depending on the scheduling of tasks (see: Section 3.3.8).

```c
int main(int argc, char* argv[])
{
  size_t size = 1024;
  double *A = nanos6_dmalloc(size * sizeof(double),
      nanos6_equpart_distribution);

  // Get restart from the environment
  // restart = ...
  if (restart != 0) {
    nanos6_CR(A, size * sizeof(double), restart, ID, false);
  } else {
    #pragma oss task out(A[0;size])
    initialize(A, size);
  }

  // User code ...

  // Decide whether to checkpoint
  // checkpoint = ...

  if (checkpoint != 0) {
    nanos6_CR(A, size * sizeof(double), checkpoint, ID, true);
  }

  #pragma oss taskwait
  nanos6_dfree(A, size * sizeof(double));

  return 0;
}
```

Figure 5.4: OmpSs-2@Cluster application code to perform C/R.

Most of the underlying complexity in data redistribution is then hidden from the final user in order to keep the programming model and API simple and transparent. The runtime takes care of all the low-level detail using most of the existing infrastructure. But the current implementation is not yet fully transparent because the final code needs some modifications in order to use it.

For fault tolerance purposes, it makes more sense to provide a fully transparent interface for the final user, but this approach has some limitations in OmpSs-2@Cluster and is not part of the topic of this research.

The semi-transparent approach is simpler to implement but powerful enough to provide a flexible infrastructure for different backends maintaining a full abstraction within the programming model requirements.

- The memory allocation is completely independent of the C/R core. This has the flexibility that the only requirement to allow restarting is the array size.
- Although the allocation policy is useful internally to save the data, the final checkpoint format is independent of it, so the restart can be executed even if the restart array was allocated with a different policy.
- The API as now is oriented to memory regions similar to the malloc interface.

- The policy used to recover the data is the one used in the target array allocation. This is consistent with the locality policies but also avoids the need to save extra metadata from the original objects.
- There is no taskwait before or between the C/R API calls. As explained in Figure 5.7 the implementation internally uses tasks for the IO operations and the runtime system can assert the correctness.
- The `restart` and `checkpoint` are just numbers to identify the data to recover and save.
- The variable id is an internal identifier for a variable to match the save and recover operation.

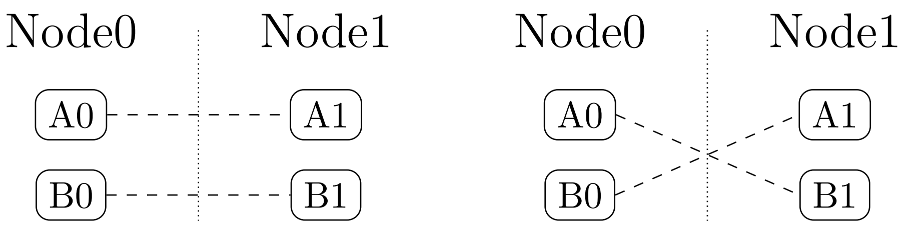
5.3.2 C/R implementation with tasks

5.3.2.1 Collective operations with tasks

The first problem when using collective operations with `MPI_THREAD_MULTIPLE` is the deadlock risk, which may occur when MPI calls are inside OmpSs-2@Cluster tasks.

```
{
  #pragma oss task in(A[0]) node(0) label("A0")
  MPI_Collective(A[1]);

  #pragma oss task in(A[1]) node(1) label("A1")
  MPI_Collective(A[1]);
}

{
  #pragma oss task in(B[0]) node(0) label("B0")
  MPI_Collective(B[0]);

  #pragma oss task in(B[1]) node(1) label("B1")
  MPI_Collective(B[1]);
}
```

(a) Source code



(b) Correct matching of collectives **(c) Incorrect matching of collectives**

Figure 5.5: Example correctness issue with offloaded OmpSs-2 tasks.

Figure 5.5 shows a simple example where collective calls within offloaded tasks may match incorrectly, depending on the order in which tasks are scheduled. There is nothing in Figure 5.5a to ensure that the collectives in tasks A0 and A1 match as the first operation and the collectives in tasks B0 and B1 match as the second operation, which is the intended behaviour illustrated in Figure 5.5b. Depending on the order in which the tasks are scheduled, the collectives in tasks A0 and B1 and in tasks B0 and A1 may incorrectly match, as illustrated in Figure 5.5c.

- Generally, the collective operations in MPI are thread-safe, which means that they may be safely used by multiple threads without any user-provided thread locks; however, thread-safe enforcement does not guarantee the execution for the routines and becomes fragile in highly parallel environments like OmpSs-2.

- Not all the MPI collective operations are guaranteed to be thread-safe by the standard (i.e `MPI_File_open`) and a correct application will require extra protections from the runtime side.

- The MPI internal locking is not enough when the same task is intended to execute multiple MPI collective operations.

```
{
  #pragma oss task in(A[0]) inout(S0) node(0) label("A0")
  MPI_Collective(A[0]);

  #pragma oss task in(A[1]) inout(S1) node(1) label("A1")
  MPI_Collective(A[1]);
}

{
  #pragma oss task in(B[0]) inout(S0) node(0) label("B0")
  MPI_Collective(B[0]);

  #pragma oss task in(B[1]) inout(S1) node(1) label("B1")
  MPI_Collective(B[1]);
}
```

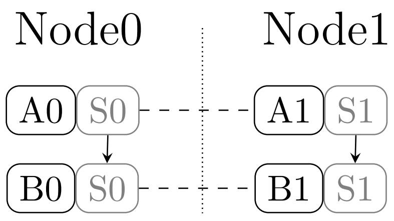


Figure 5.6: Solution with sentinel and strong tasks for the issue in Figure 5.5.

There is a simple solution for this issue to enforce the order of the tasks either locally and remotely without taskwait or blocking threads. We can use a sentinel for the tasks but specific to every node as shown in Figure 5.6. Such a sentinel may be known at the main task level and allocated in the OmpSs-2@Cluster known memory (either local or global) in order to be added as an OmpSs-2@Cluster task dependency.

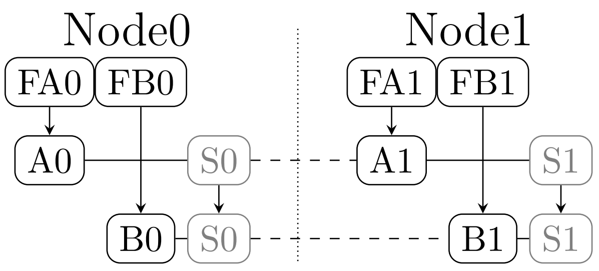
The solution in Figure 5.6 solves the correctness issue serializing the tasks locally with the sentinel and enforcing the execution order thanks to the OmpSs-2 semantic. All the other tasks with different dependencies may continue executing in parallel and the task generation is not stopped in any moment.

This approach works correctly when there are few and small dependencies or when the collective operation will be executed in the same node the data actually is. But the tasks `B1` and `B2` are not ready until the sentinels are released because the dependencies are strong. This means that the task offloading for B1 but also, the data transfers for B0 and B1 are unnecessarily delayed waiting for the sentinels on the main node and potentially adding an important overhead for big data transfers.

When the tasks need to access and copy a significant amount of data before executing the collective operation; it is more efficient to offload and perform the copies as soon as possible and enforce the execution order only for the collective call itself to reduce excessive serialization. If there is some region overlapping, we rely on the dependency system for correctness without extra mechanisms.

```
#pragma weakin(A[0]) weakinout(S0) node(0)
{
  #pragma oss task in(A[0]) node(nanos6_no_offload) label("FA0")
  {}
  #pragma oss task in(A[0]) inout(S0) node(nanos6_no_offload) label("A0")
  MPI_Collective(A[1]);
}
#pragma weakin(A[1]) weakinout(S1) node(1)
{
  #pragma oss task in(A[1]) node(nanos6_no_offload) label("FA1")
  {}
  #pragma oss task in(A[1]) inout(S1) node(nanos6_no_offload) label("A1")
  MPI_Collective(A[1]);
}

#pragma weakin(B[0]) weakinout(S0) node(0)
{
  #pragma oss task in(B[0]) node(nanos6_no_offload) label("FB0")
  {}
  #pragma oss task in(B[0]) inout(S0) node(nanos6_no_offload) label("B0")
  MPI_Collective(B[0]);
}

#pragma weakin(B[1]) weakinout(S1) node(1)
{
  #pragma oss task in(B[1]) node(nanos6_no_offload) label("FB1")
  {}
  #pragma oss task in(B[1]) inout(S1) node(nanos6_no_offload) label("B1")
  MPI_Collective(B[1]);
}
```

(a) Code solution with weak, strong and sentinel.



(b) Task representation.

Figure 5.7: Sentinel solution for the issue in Figure 5.5 using weak and strong tasks.

The schema shown in Figure 5.7 presents several advantages compared to the one presented in Figure 5.6

- The weak tasks become ready immediately and can be offloaded in advance with no delay.
- The fetch tasks F[A-B][0-1] do not have a dependence on the sentinel, so they become ready immediately when the data locations are available.
- Like in Figure 5.6 the call order for the collective MPI functions in every node is determined by the sentinels.
- Only the tasks in the same node will have strong dependencies on the node's sentinel, which avoids unnecessary data transfers between operations.
- As the weak tasks are offloaded in advance; the sentinels can propagate directly within the node's namespace, saving all communications and the corresponding latency.

- For read operations that do not require previous data transfers, most of this optimization is valid, but the fetch tasks are unneeded.

5.3.2.2 Data locality and IO operation balance.

The parallel IO MPI operation is sensitive to the IO operation balance. When using collective write or read functions, the MPI library has extra information to optimize re-balancing the IO and transferring internally; but such optimizations are implementation specific and using only collective MPI-IO sacrifices flexibility for a general purpose solutions like ours.

In OmpSs-2@Cluster the data (memory content) stays in the place where it was generated or modified by the last tasks which "touched" it. This approach is useful to minimize unnecessary data transfers during the execution but may produce some imbalance in the data locations.

The affinity and distribution policy in the distributed memory allocation API is only a hint generally used for scheduling purposes; but resulted in being useful for IO purposes on distributed regions.

The distributed memory policies, in general, assign a more or less balanced amount of data to every node. This characteristic is not a strong requirement, but it is the most common use case to increase load balancing without needing a node clause on every task. The distribution policy also reduces the risk of running out of physical memory in a node by associating a virtual memory region with a node where it can be touched and used with preference.

Relying on the distributed memory policy is then a good candidate for a default IO policy for C/R. The final implementation will join the algorithm described in Figure 5.7 with the steps in Figure 5.1.

However, more specialized policies can be implemented relatively easily with the current infrastructure in the future.

5.4 Dynamic process management on Nanos6

Our work has endowed Nanos6 to dynamically manage the number of processes in use transparently for the user. The implementation decouples the DPM from Slurm integration in order to use it in different scenarios and restrictive clusters where some characteristics are disabled (see: Section 5.2.3)

Unlike the C/R strategy, DPM requires different implementations when adding or removing processes. This section dedicates a part to the process of addition (Section 5.4.1) and another to the process of deletion (see: Section 5.4.2).

5.4.1 Adding processes in OmpSs-2@Cluster

Nanos6 internally uses an abstraction where every MPI rank is considered a "compute" place ClusterPlace. The runtime determines the initial number of ClusterPlaces and its configuration (number of cores, host, memory) by the arguments passed to `srun` or `mpirun`; the environment variables set by MPI and the RMS and some information exchanged between all the MPI ranks during the initialization. When the RMS API integration is enabled, more advanced, specific and precise information can be collected (see Section 5.2.3 and Section 5.5).

That information is used internally and updated during the resizing or system changes because the OmpSs-2@Cluster resize policies (see Section 5.4.1.3) do not rely on the default Slurm or MPI distribution policies to create and set the new processes:

1. Depending on the configuration, the RMS may allow processes over-subscription, which generally is undesirable in OmpSs-2@Cluster impacting performance negatively.

2. `MPI_Comm_spawn` generally relies on `srun` or `mpirun` by default (see: Section 5.2.2). Either of those distributes the processes with different policies generally not compatible with malleability. (i.e. using any distribution similar to round–robin per host with 4 MPI processes per host require removing at least 13 ClusterPlaces from the application in order to release a single host; leaving a system with few compute places and many unused resources.)

5.4.1.1 Limit number of processes

As explained in Section 5.2.2 the current DPM implementation in OmpSs-2@Cluster is constrained by some limitations in the MPI standard. The immutability of `MPI_COMM_WORLD` imposes limits to the minimum number, the granularity and distribution of new processes.

Since the initialization moment, the user defines the minimum and the maximum number of MPI ranks the application can use. It is the number of MPI processes at the beginning of execution when the runtime started (i.e. the `-N` argument of `srun`). In other words, the processes that conformed to the initial `MPI_COMM_WORLD` cannot be removed later.[1]

On the other hand, the common address space allocation (see Section 3.3.3 and Section 3.5.2) limits the maximum number of processes. In the same way MPI and the RMS influences these limits and distributions.

During the initialization, a collective "negotiation" takes place between all the initial processes to establish the common address space. For that reason, every spawned process receives the allocation information from the master node in the startup; because it was not part of that initial negotiation. The new process rejects to initialize as part of the application if the virtual memory region is not available to map. This later situation is possible but almost nonexistent because the initial process considers multiple variables and race conditions.

The runtime assigns contiguous local virtual memory regions in the same order, the ClusterPlaces are in the processes list (see Section 5.4.1). Which keeps the local memory infrastructure and ownership checks are simpler, more robust and efficient.

[1]From the runtime point of view, it is actually possible to remove some initial processes from the list of ComputePlaces and instruct them to finalize, but all of them will be effectively waiting in the implicit barrier in `MPI_Finalize`. A potential use case for that may be when using multiple processes per node, and we need to release memory to be used by other processes in the same nodes. This use case would require more strict control on process distribution and creation by the initial `srun`.

5.4.1.2 Communicator update

As explained in Section 5.2.2 the result of `MPI_Comm_spawn` is another `MPI_COMM_WORLD` and an inter-communicator.

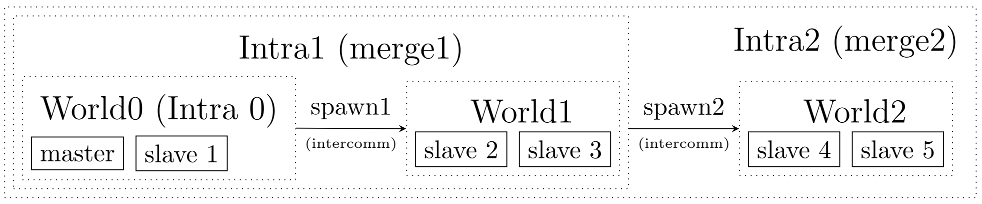


Figure 5.8: Communication update process in Nanos6 for OmpSs-2@Cluster. Every `MPI_Comm_spawn` is followed by an `MPI_Comm_merge`.

As Figure 5.8 shows, every `MPI_Comm_spawn` is followed by an `MPI_Comm_merge`. This constructs an onion-like structure where every new intra-communicator only extends the last intra-communicator to include the new spawned world. This onion-like schema exposes some of the characteristics that will direct the shrink process later: The outer layers can be removed without affecting the inner ones; this means that the last processes added will be the first to be removed (see Section 5.4.2).

In this example, the spawn is executed in two steps, either with two user spawns of size two or a single user spawn of 4 and granularity 2. The resulting underlying granularity is the same.

Another advantage of this schema is that the previous communicators are cached in order to simplify the reverse process and provide the runtime with all the information of the latest spawn size, and some other properties.

It could be possible to revert the merge process with other methods like `MPI_Comm_split`, but that approach ignores the granularity of the underlying `MPI_COMM_WORLD` merged in previous resize operations. Without such granularity information, the runtime does not have control over how many processes are effectively removed in incomplete spawn steps or which are waiting in the `MPI_Finalize` without releasing the resources. (see Section 5.2.2)

With the spawn–merge approach, the all-to-all communication pattern is saved from the inter-communicator world-to-world pattern emerging from the `MPI_Comm_spawn` default behavior. As explained in Section 5.2.2, `MPI_Comm_spawn` is a collective operation and needs to be called by all the ranks of the parent communicator.

At the end of the merge, the processes in the parent communicator preserve their rank while the new ones will have the consecutive values starting from the initial communicator size.

5.4.1.3 Spawn policies

A single call to nanos6_cluster_resize may result in one or more calls to MPI_Comm_spawn depending on the granularity. We implemented three different spawning policies to control the spawning granularity:

byOne : Processes are spawned one-by-one

byHost : Processes are spawned per host

byGroup : All processes are spawned at once

The **byOne** policy is the most flexible approach because it provides the smallest possible granularity. With this policy every spawn–merge step is executed individually for each new process. This policy provides a fully malleable application without restrictions, and it is especially useful when using multiple processes per node to have fully fine-grained control of the ranks in the parallel application. On the other hand, the **byOne** policy executes the maximum possible number of calls to `MPI_Comm_spawn` which makes the resize process time-consuming for a high number of new processes.

Figure 5.9 in Section 5.4.1.5 describes how this policy works internally.

The **byGroup** policy is the completely opposite of **byOne**. In this case, the spawned processes are created in a single `MPI_Comm_spawn`. This policy is much faster compared with the previous policy but sacrifices all the underlying granularity. In case of a later process removal, all the new processes created in a **byGroup** need to be removed together. Another disadvantage is that the policy completely relies on the system (RMS+MPI) decision for the processes' distribution and affinity.

The last policy is **byHost** which spawns the new processes by steps like **byOne**; but using subgroups in order to reduce the number of `MPI_Comm_spawn`. This policy coincided exactly with **byOne** when using only one process per node. The number of processes per node is one of the values that Nanos6 attempts to calculate more hardly based on the environment provided by RMS and MPI and in the worst case, it is assumed to be one to avoid over-subscription.

5.4.1.4 Data migration

Unlike most MPI applications where the simplest approach is to migrate and initialize the data just after creating the new processes; OmpSs-2@Cluster does not need any data transfers or migration when new processes come into the applications. This saves all the complexity of memory redistribution and migration while avoiding unneeded or redundant transfers.

The data migration during the expansion process is completely transparent to the user and consistent with the programming model. The user only needs to be concerned about task offloading locations and scheduling policies.

Data transfers happen lazily on demand: when a task is offloaded in one of the new nodes, it will perform the copies and data transfers in the steps previous to the execution. After that, the data remains in that node until another task uses it in the same node or starts a transfer to a different one. The lazy transfer schema does not add any significant overhead more than the `MPI_Comm_spawn` collective operation.

5.4.1.5 Spawn process on Nanos6

Figure 5.9 shows the spawn process from 3 to 5 ranks using the **byOne** (see Section 5.4.1.3))

1. When the resize is executed by the user code, only one API call is needed to specify the number of new processes. The resize function can only be used from the main code.

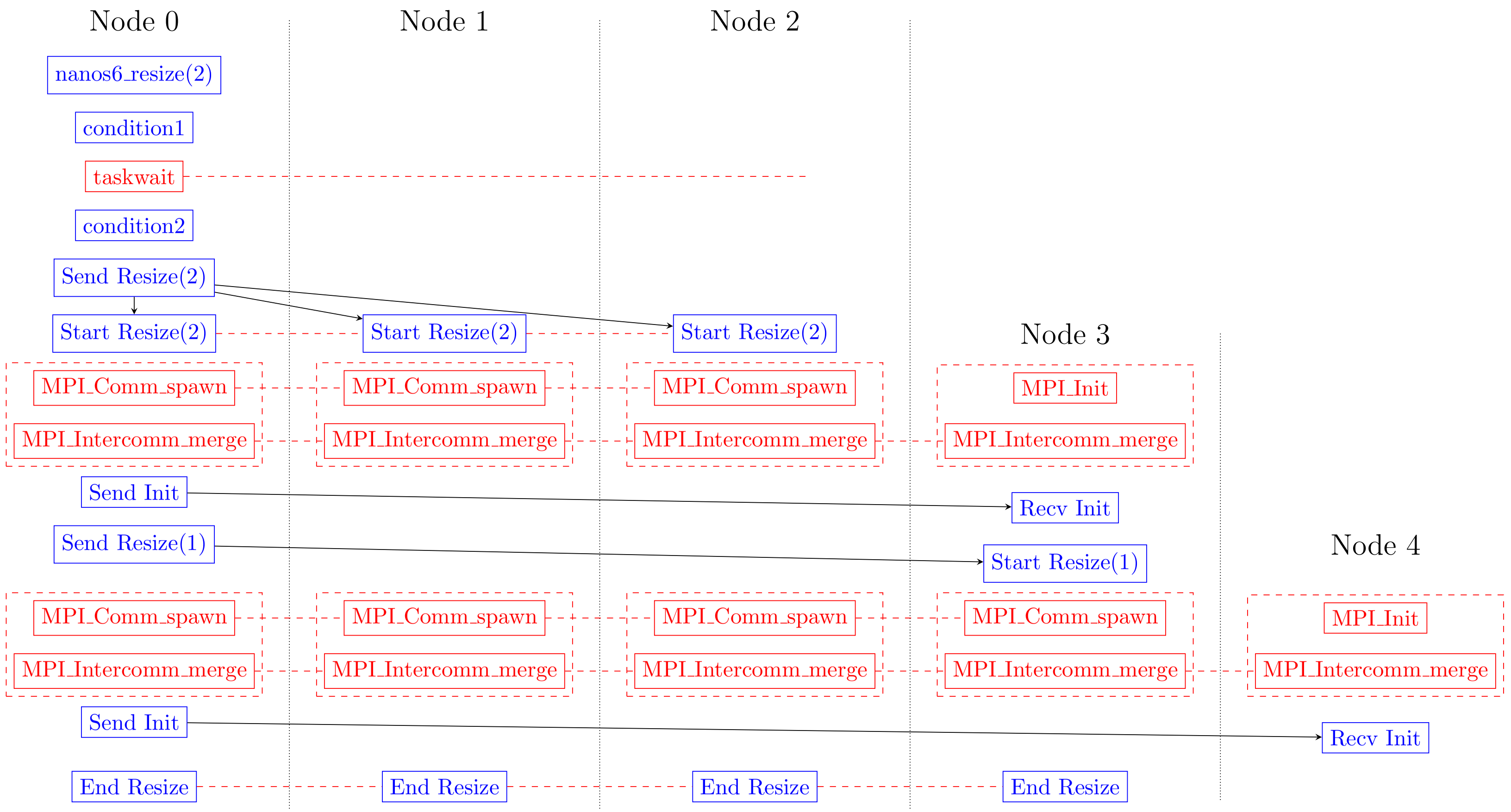


Figure 5.9: OmpSs-2@Cluster resize, adding two ranks with the stepped byOne policy in two spawns. The runtime requires two steps and performs internal intermediate communications with the new process on every step. The master node directs the process and the resulting underlying granularity will be one.

2. The resize call contains an implicit taskwait before starting the spawning process. This is necessary because in the resize process, the communicators will be replaced, so pending communications or polling services interfere and cause problems.
3. There are two conditions around the taskwait used by the RMS API for the flexible policies (see Section 5.5).
4. All the spawn process is managed by the main node. Once the spawn decision is taken, it is responsible for broadcasting the resize information with all the existing ranks.
5. Every `MPI_Comm_spawn` is followed by an `MPI_Intercomm_merge` involving all the processes in the parent communicator and in the new child communicator as well.
6. The main rank sends the initialization information to the new ranks in a very early step. Such information is the first a new process should receive because it contains information to initialize the runtime.
7. On stepped spawn policies like **byHost** and **byOne** the master rank also sends the resize information to the newly created processes before continuing with the rest of the steps. This is not needed for the very last step.
8. The end resize is a final stage where the runtime is collectively set into normal status. At that point some final remarks take place in order to make the new processes real ClusterPlaces like any other and set the runtime to the normal status. (i.e. share dmalloc information with the new ranks, restart the polling services, update system information about the processes in every host etc.)

5.4.2 Removing processes in OmpSs-2@Cluster

To remove processes OmpSs-2@Cluster has some constraints determined by the underlying MPI (see: Section 5.2.2 for more details). The most important from the user point of view is that only the processes added in a previous resize step (see Section 5.4.1) can be removed from the parallel application. This is actually an implementation choice to keep the runtime startup process simpler and efficient.[2]

From the implementation point of view, the process removal is simpler and, in many aspects, symmetric to process addition. The only significant difference is that the runtime needs to perform a data migration before removing some processes.

5.4.2.1 Data migration and locality

Data migration is a crucial step before removing the processes to avoid losing data. When the `main` task performs `nanos6_resize`, the first step is an implicit taskwait, which waits for all tasks to complete. Following the taskwait, the locations of the latest versions of all accessible data in the program are known in the access structures of the `main` task.

[2]These constraints could change in the future when most MPI implementation include the new `MPI_Session` feature and the backend MPI management migrate.

The data migration step is done immediately after the taskwait, following one of two possible approaches:

Eager : migrate all data immediately to the new node on which the data has an affinity.

Lazy : only migrate data from nodes that are about to be removed.

The **eager** method migrates all the data from its current location to the new home node. Depending on the amount of data and the application, this may be time-consuming and perform unnecessary data transfers that will be reverted by the data transfers for future tasks. Nevertheless, reorganizing the data may be desirable for automated locality scheduling.

On the other hand, the **lazy** policy minimizes data transfers by moving only data that is currently located on ranks that will be removed. The data is moved to the new home node. This minimizes transfers, but the resulting data location may be unbalanced or not totally well distributed considering the new number of ranks.

The list of data transfers is included in the "Resize" message that is sent to each node, allowing the source and destination of each data transfer to initiate two-sided point-to-point communication with MPI.

Home rank update The global memory allocation assigns a home rank to every distributed memory region as explained in Section 3.3.5. The general distribution of data is provided by the user, but the internal details are handled and hidden by the runtime. The runtime tracks these regions and policies with their initial distributions, but when a resize takes place, the runtime needs to reassign all the home ranks to include the new processes in order to improve the data and work balance. On the contrary, when removing ranks, the home redistribution is actually not an improvement but a correctness requirement.

5.4.2.2 process removing and granularity

Removing processes works using a stack like Last In First Out (LIFO) approach as explained in Section 5.4.1.2 together with the process granularity issue. Figure 5.8 on p.105 shows the source of such granularity and explains how the process removal and runtime updates work. With the same Figure, it is possible to explain the reverse process with opposite steps.

1. All the processes collectively call `MPI_Comm_free` over the intra-communicator **Intra2**.

   This step creates two groups of processes:

   Group0 The older group of processes that existed before the last spawn (master,slave[1-3]) inside the previous intra-communicator **Intra1**.

   Group1 The last group of merged processes (slave[4-5]) inside to their initial `MPI_COMM_WORLD`.

2. The two groups of processes disconnect the inter-communicator between them.

3. The Group1 goes to `MPI_Finalize`.

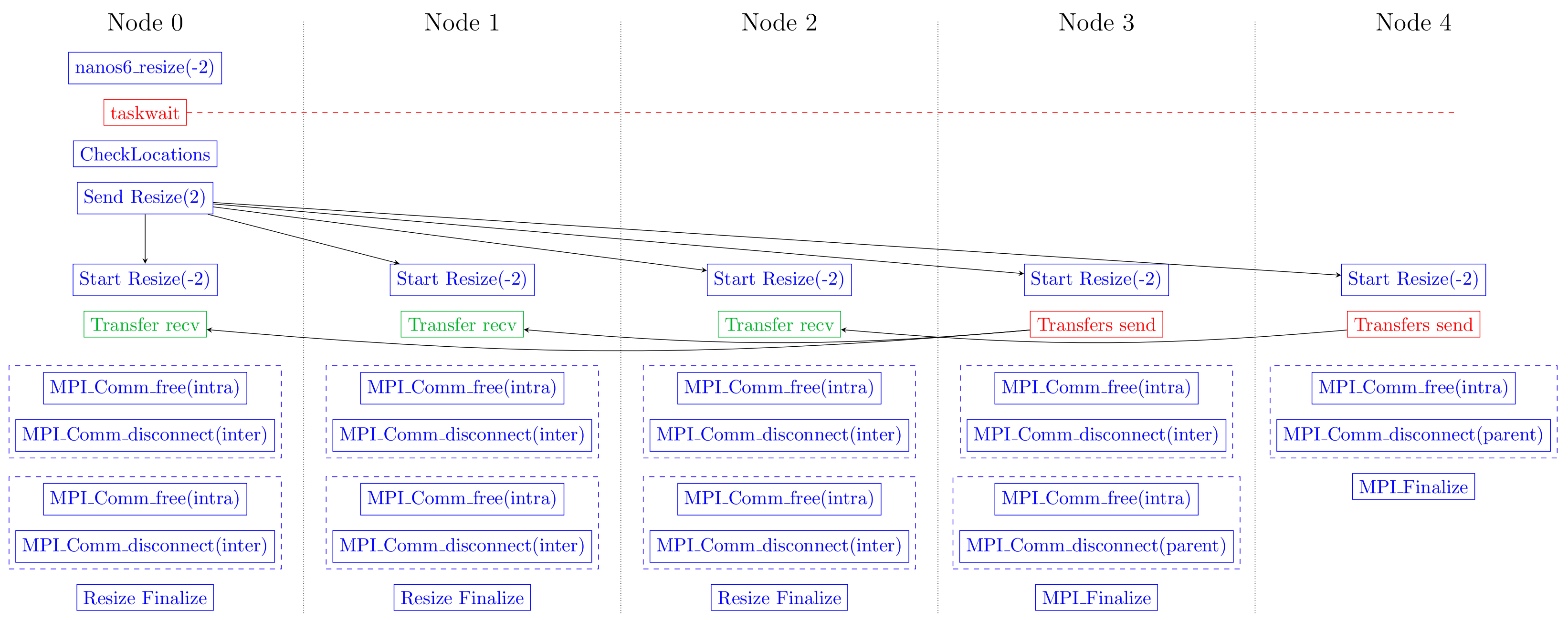


Figure 5.10: OmpSs-2@Cluster resize removing two ranks. The process takes advantage (and requires) an underlying granularity that allows releasing the removing nodes.

4. The Group0 repeats from step1 up to the desired number of released resources.

Figure 5.10 shows the internal steps to revert the resize example from Figure 5.9 in Section 5.4.1.5.

5.5 Nanos6–Slurm integration

The Slurm integration API is an optional module in nanos6 because not all the clusters use Slurm and most of the system with Slurm does not allow extending jobs in the configuration. The malleability feature is also disabled by default unless the user specifies a maximum number of ranks bigger than the initial world size. Malleability can be enabled to be used in over-provisioned jobs, and for that reason, it is not strongly dependent on the Slurm API availability.

5.5.1 Job expansion using Slurm

Section 5.2.3 explains the method to extend Slurm jobs using the Slurm API. Our runtime simplifies the process to the user using the method described in Figure 5.3. As explained, there are multiple situations that the user must consider allocating new jobs and extending the application.

As a general-purpose programming model, OmpSs-2@Cluster needs to provide multiple policies to execute job allocations in order to expose a simpler interface to the user but without sacrificing too much flexibility. The allocation policies decide when the application needs to request extra nodes and how strongly it should wait for it.

To reduce the Slurm dependency we use the `condition1` and `condition2` in Figure 5.9 to localize the Slurm API calls associated with these policies. The key of these two conditions is the taskwait between them.

Taskwaits are expensive and create excessive overhead because they stop the job creation and consequently the early task offloading, data copies and namespace propagation. For that reason, the taskwaits should be avoided when they are not necessary. However, to spawn new processes, the taskwait is important to collectively replace the communicator without pending communications or polling services interfering.

We based our implementations and the options available on the possible combinations form the schema in Figure 5.3.

5.5.1.1 Policies to allocate more resources from the RMS

Our implementation provides OmpSs-2@Cluster with two policies to allocate resources through the Slurm API. Each of them has different targets depending on the application and the availability of resources.

Timeout policy The application attempts to allocate the resources assuming some high availability, but not immediate, so it adds a timeout to wait for the allocation before resigning. If the allocation takes place before the timeout expires, then the taskwait is effectively added and the resize takes place after it.

The timed policy is the most extended approach for general purposes applications [101, 103, 104]. Our implementation takes the idea from the DMRlib implementation and from the Slurm API function `slurm_allocate_resources_blocking`.

The timeout values control all the policy:

0 Check for immediate allocations only. When the immediate allocation is not possible, the resize is cancelled.

> 0 Gives a grace period in seconds before cancelling the resize operation.

< 0 Undefined at the moment.

(Semi) Blocking policy The application attempts to resize existing taskwaits. This is a research experimental policy that assumes less availability of resources than the previous one.

When the resize function is executed, it checks if some extra nodes are necessary and attempt to allocate them in `condition1`. The taskwait is unconditionally added in this case. Once the taskwait is crossed, the runtime checks if the resources were already granted (condition2) and execute the resize. This policy also includes a timeout that produces important changes.

0 If the resources were not granted, then the allocation request and resize operation are cancelled immediately.

> 0 Gives a grace period before cancelling the resize operation. This timeout is intended to be higher than the ones in the Timeout policy.

-1 Waits indefinitely for the allocation before continuing. This is useful when the application cannot continue without the extra resources.

As OmpSs-2@Cluster applications generally offload many tasks in advance to exploit namespace propagation and overlap computation with data transfers; it is common that the main arrives to the taskwait long before the previous tasks finish their execution. This policy takes advantage of that to give more time and opportunity to receive the resources from Slurm. The main disadvantage of this policy is the requirement of the taskwait.

5.5.2 User interface

We extended the OmpSs-2@Cluster API to add two essential resize functions in order to perform malleability. This API modification was designed to be simple for the user but also to fit properly in the OmpSs-2@Cluster abstraction schema.

`nanos6_cluster_resize` This is the simple and probably what a real application will use in most cases where real malleability will rely on the configured policy and simple code.

This function just accepts an integer argument with the resized size. The number specifies the number of processes to create (when the argument is positive) or remove (when the argument is negative). This function takes the process creation policy (see: Section 5.4.1.3) from the configuration variable.

`nanos6_cluster_resize_with_policy` This is an advanced version where the user can specify the policy to create processes or transfer data. This version is especially useful for testing and is not supposed to keep compatibility with different versions of Nanos6; because it should provide all the options and features added to malleability in the future.

5.6 Benchmarks and methodology

5.6.1 Hardware and software platform

To execute the benchmarks, we use MareNostrum 4; the same hardware architecture described in Section 3.8.1.

For malleability benchmarks, we use MPICH and IMPI to compare the results. We measured the spawn times in the system using different configurations.

5.6.2 Slurm environment

MareNostrum restricts many permissions to normal users, some requirements to do malleability are not available as a production system (see: Section 5.2.3). For that reason we need a more flexible and controlled environment to test and profile the Nanos6–Slurm integration API.

Our method creates a nested Slurm environment inside a submitted MareNostrum 4 job. In that environment, we reserve a group of nodes and manage them with the nested Slurm as a mini-cluster without restrictions or permission problems. We can assert that the submitted jobs are allocated immediately for the spawn and shrink benchmarks in order to not have noise from lack of resources availability or other jobs. We use the release Slurm version 21 from the official repositories and the configuration to reproduce the nested slurm environment is available in our GitHub repository [3].

It is important to clarify that it is possible to use a normal cluster partition without our setup for the malleability benchmarks. In a pre-allocated over-provisioned job the OmpSs-2@Cluster application ca spawn and shrink processes in the nodes of the job; the slurm integration is optional and independent of the DPM (see: Section 5.5). When the runtime detects that the Slurm configuration restricts some malleability requirement it assumes it is in an over-provisioned job.

All the benchmarks reported here use our nested setup. Nevertheless, we execute some tests outside the setup to compare and the results shows no significant difference.

5.6.3 Benchmarks

5.6.3.1 Spawn benchmarks

For the initial tests of the malleability implementations, we reuse part of the codes from Section 3.8.2. The malleability operation itself is independent of the algorithm and data distribution because it does not perform any eager data transfer; it follows the OmpSs-2@Cluster approach to move data only when needed by some tasks.

This approach is simpler to implement and has the main advantage that it does not add extra overhead associated with the transfers. However, it is expected that the first tasks offloaded to the new processes require some transfers like any other task. The expected overhead may be proportional to the size of the total number of data transfers involved.

We selected the matvec benchmark because it is a very well-balanced problem. In practice, the real cost of spawn may be in the order of a few seconds. For unbalanced or irregular applications, the time spent on the taskwait may be longer and application specific.

For the spawn benchmarks, we modified one of the matrix–vector multiplication benchmarks used in Section 3.8.2 and added `nanos6_cluster_resize` calls every some predetermined number of iterations. We measure the time needed for every iteration, and we compare the three approaches.

In the benchmarks, we measure the time needed to spawn new processes using the three spawn policies described in Section 5.4.1.3.

The spawn benchmark performs a process resize every 4 matvec iterations for a 65536×65536 matrix. On every step, we duplicate the number of processes from 1 up to 32 with a distribution of 1 process per node in order to maximize any network effect.

5.6.3.2 Shrink benchmarks

We use two processes to measure the cost of the shrink process operation with the two transfer policies.

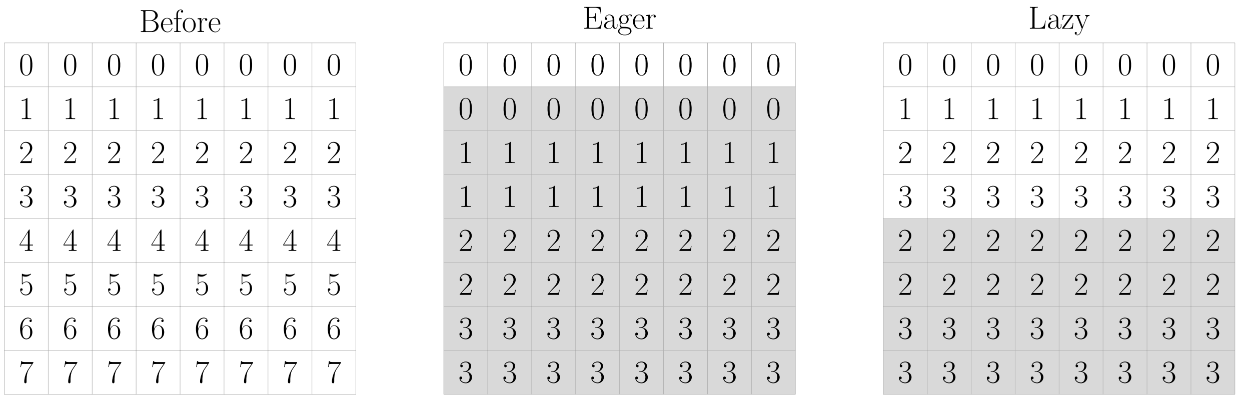
Shrinking processes is much faster than spawning in the MPI side because we do not need to split the communicators; we just release the merged intra-communicator and disconnect the inter-communicators (see Figure 5.8 and Section 5.4.1.2).

On the other hand shrinking requires executing data transfer to save the data that is located in some exiting nodes (see Section 5.4.2.1). In the benchmarks, we compare the data migration policies for two of the benchmarks with a very different memory distributions: matvec by slices and Cholesky with fair distribution

Matvec shrink benchmark Matvec implementation is explained in Section 3.8.2.2. It has a very regular memory and data distribution. In a fixed-size application, there is not even data migration for that benchmark because all the tasks are executed in the same home node of the access region. However, on application resize, the memory home nodes needs some update, and consequently, some transfers may be executed.

Figure 5.11 shows an example memory layout representation for the memory redistribution in a matrix of 8×8 from 8 processes to 4. The numbers represent the initial and final home node for every element in the matrix. Depending on the transfer policy, the number of transfers may vary. With the lazy transfer, only two data transfers are needed: $rank2 \rightarrow A[2;:] \rightarrow rank0$ and $rank3 \rightarrow A[3;:] \rightarrow rank0$. With the eager policy an extra transfer is needed: $rank1 \rightarrow A[1;:] \rightarrow rank0$.

Cholesky resize benchmark The cholesky benchmark uses a more complex "fair" distribution in order to balance the work on every node (see Section 3.8.2.4). Such distribution is specific to this problem and implemented at the user level. After the data initialization, all the memory regions are in their "fair" location instead of the home nodes. After a shrink resizes, the regions that need


Before

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5
6	6	6	6	6	6	6	6
7	7	7	7	7	7	7	7

Eager

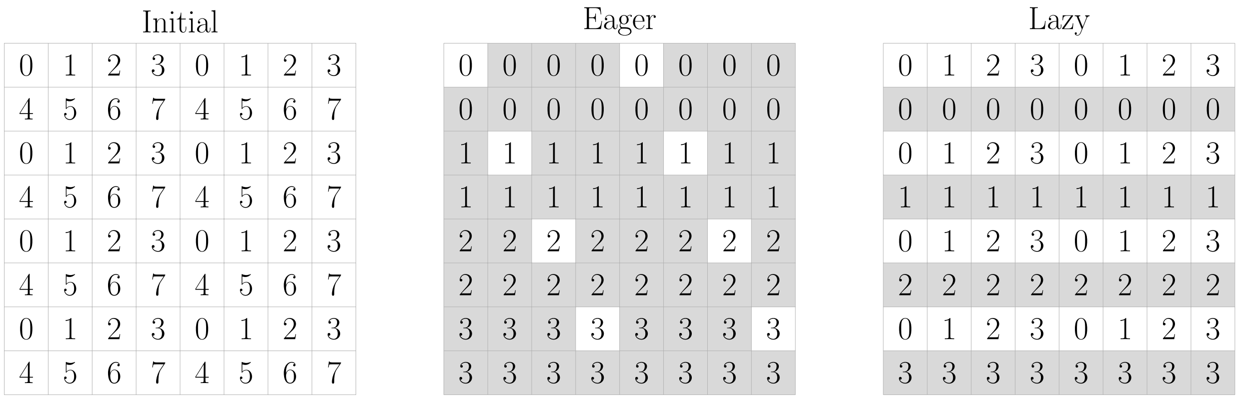
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3

Lazy

0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3

Figure 5.11: Matvec memory redistribution from 8 ranks to 4. The numbers represent the data location. The transferred regions are colored in gray.

migration are not in their home node but need to be migrated to the new homes for the nodes that are exiting.


Initial

0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7
0	1	2	3	0	1	2	3
4	5	6	7	4	5	6	7

Eager

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3
3	3	3	3	3	3	3	3

Lazy

0	1	2	3	0	1	2	3
0	0	0	0	0	0	0	0
0	1	2	3	0	1	2	3
1	1	1	1	1	1	1	1
0	1	2	3	0	1	2	3
2	2	2	2	2	2	2	2
0	1	2	3	0	1	2	3
3	3	3	3	3	3	3	3

Figure 5.12: Cholesky memory redistribution from 8 ranks to 4. The numbers represent the data location. The transferred regions are colored in gray.

Comparing Figure 5.12 with Figure 5.11 we get some details:

1. The total amount of data to transfer with the same policy is the same in the two benchmarks.

2. The fair distribution (Cholesky) requires more individual transfers because the initial data is spread in smaller regions among the nodes.

3. The equpart distribution (matvec) requires fewer transfers because the entire rows (contiguous in memory) are in the same node requiring a single communication to migrate.

4. The final distribution with the eager policy is always the same but requires a more intense communication pattern and update more data.

5. The final distribution after a lazy transfer is unbalanced for matvec but not for Cholesky.

6. With the lazy policy, only half of the "surviving" ranks are involved and receive data with equpart opposite to the fair distribution where all the nodes are involved.

5.7 Results

5.7.1 Spawn benchmarks

5.7.1.1 Checkpoint and restart for spawning

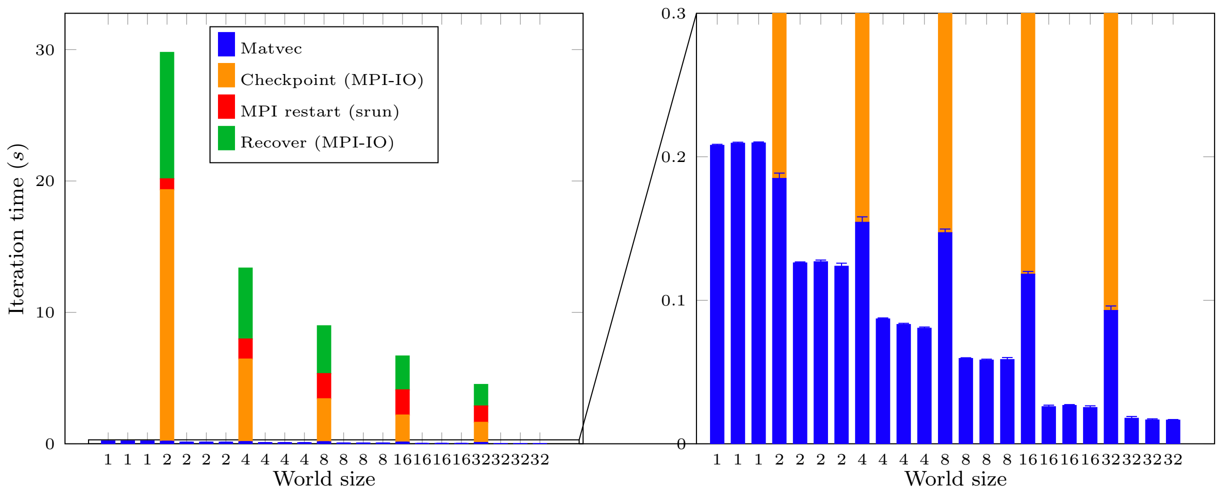


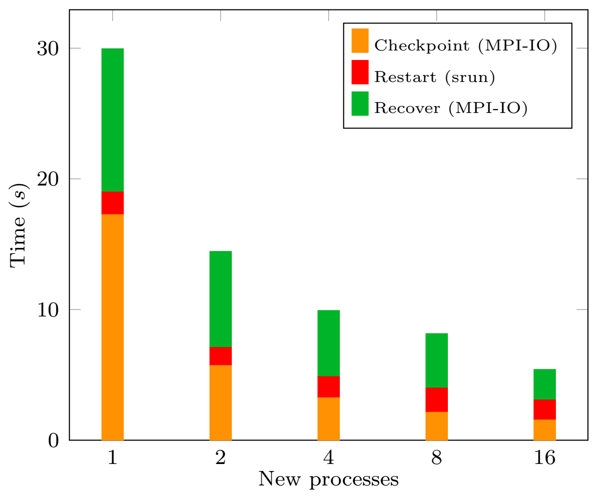
Figure 5.13: Matvec time per iteration for process duplication every 4 iterations in a 65536×65536 matrix with C/R.

Figure 5.13 shows a timeline of the time per iteration for the matvec benchmark, with a matrix of size 65,536 and one process per node, using the C/R approach for malleability (See Section 2.3). The x-axis covers the time, measured in terms of the iteration number, and each iteration is labelled with the current total number of nodes. The y-axis is the wallclock time for the iteration, subdivided into the time for the algorithm (matvec), and the times for checkpoint, rescheduling with `srun`, and recovery. All times are the average for that iteration, over 20 runs. The program starts with 1 process and doubles the number of processes every four iterations. The time to perform C/R is included as the first part of the first iteration with the updated number of nodes.

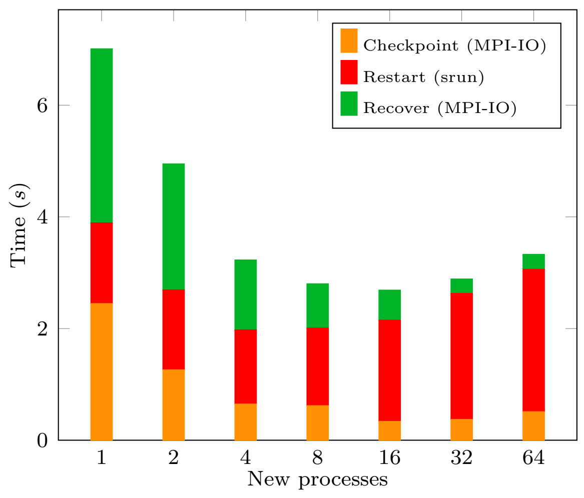
The left-hand plot of Figure 5.13 shows that the malleability operations take much longer than the application, but this is because they are happening far more often than would typically be the case—merely for the intelligibility of the plots. More importantly, they show that the majority of the time, for malleability is due to the IO operations, with the process restart being much faster (in this case where there are a sufficient number of idle nodes). The right-hand plot of Figure 5.13 shows a zoom, on the y-axis. We see that the additional processes are correctly being used by the runtime, although scalability is not perfect.

The first iteration after the restart is a bit slower than the others because, in the beginning, the matrix and the input vector are distributed in their home node, but the input vector is needed in all the processes and tasks in order to execute the computation. Therefore, all the nodes need to perform some small all-to-all transfers in the first iteration after the restart.

Figure 5.14 shows the times for C/R for the checkpoint and restart process, when doubling the number of processes. The x-axis is the number of spawned processes, e.g. "1 new process" corresponds to a transition from 1 to 2 processes. Figure 5.14a is extracted from Figure 5.13.



(a) Matrix 65536×65536, 1 process per node



(b) Matrix 32768×32768, 4 processes per node

Figure 5.14: Checkpoint and restart time duplicating the number of processes for matvec and MPI-IO backend with IntelMPI.

Figure 5.14b is the result for a smaller matrix of size 32768, with 4 processes per node. The results are again the average of over 20 runs.

The time needed for IO operations increases proportionally to the amount of data and decreases with the number of readers/writers and nodes involved in the operation.

Processes	Checkpoint (s)	sem	Init (s)	sem	Restart (s)	sem
1	17.25	$3 \cdot 10^{-1}$	0.33	$6 \cdot 10^{-3}$	21.12	$3 \cdot 10^{-1}$
2	5.71	$5 \cdot 10^{-2}$	0.87	$2 \cdot 10^{-1}$	10.96	$3 \cdot 10^{-1}$
4	3.24	$3 \cdot 10^{-2}$	0.69	$9 \cdot 10^{-2}$	7.35	$3 \cdot 10^{-1}$
8	2.13	$9 \cdot 10^{-2}$	0.81	$1 \cdot 10^{-1}$	5.06	$2 \cdot 10^{-1}$
16	1.54	$8 \cdot 10^{-2}$	0.93	$2 \cdot 10^{-1}$	4.16	$2 \cdot 10^{-1}$
32	2.22	$1 \cdot 10^{-1}$	0.76	$6 \cdot 10^{-2}$	2.34	$1 \cdot 10^{-1}$

Table 5.1: Time for the different phases for C/R in MareNostrum with one process per node for matvec on a matrix of 65536×65536 and Intel MPI 2017.4

Table 5.1 contains the main times for the different phases in a normal C/R operation. **Checkpoint** is the time to save the data to disc with the OmpSs-2@Cluster C/R API; **Init** includes the time to start the new parallel application, including the runtime initialization and **Restart** is the time to recover the checkpoint from the storage to the parallel application memory. In each case, the average and Standard error of the mean (sem), as an estimate of the standard deviation, are given. As expected, the parallel IO times decrease with the number of nodes while the Init time remains more or less constant independently of the number of processes.

5.7.1.2 Dynamic process management for spawning

Figure 5.15 shows a timeline of the time per iteration, similarly to Figure 5.13, but this time using the DPM support and the **byGroup** policy. Note the different scales on the y-axis. Regarding time, it is faster than the C/R approach because it saves the costly IO operations.

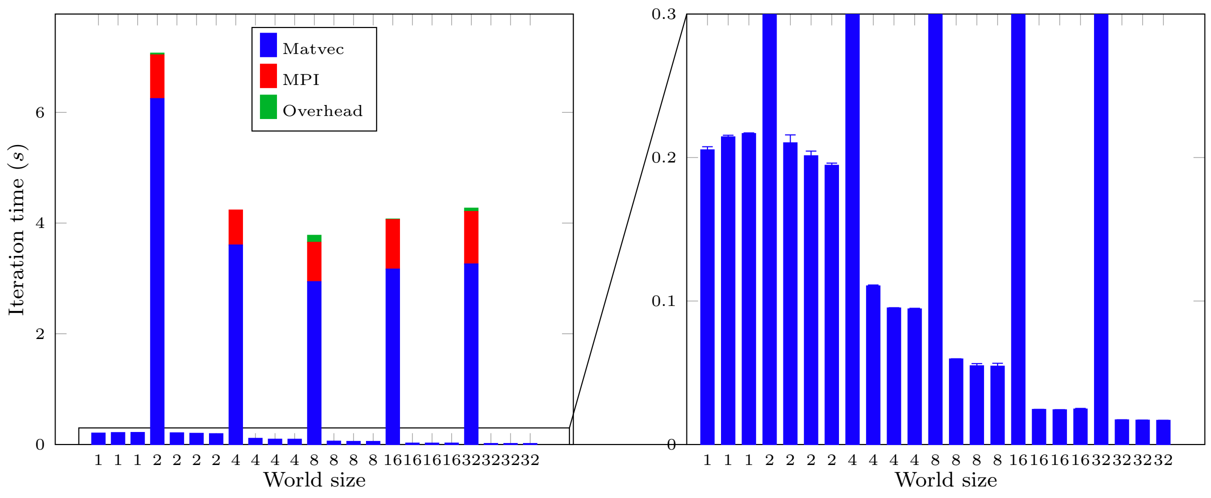


Figure 5.15: Matvec time per iteration for process duplication every 4 iterations in a 65536×65536 matrix with DPM using the byGroup policy.

The group policy is the most optimistic approach because it expands all the jobs in a single call. This is the behaviour we expect to reach once the MPI Sessions API is supported and stable in the different MPI implementations. Our approach is ready to go to use this feature as soon as MPI sessions arrive. In principle, that will avoid the granularity considerations explained in Section 5.4.1.1 and Section 5.4.1.2.

However, with the current standard, we require to use the stepped `MPI_Comm_spawn` approach from the **byOne** or **byHost** policies to keep the flexibility to remove processes.

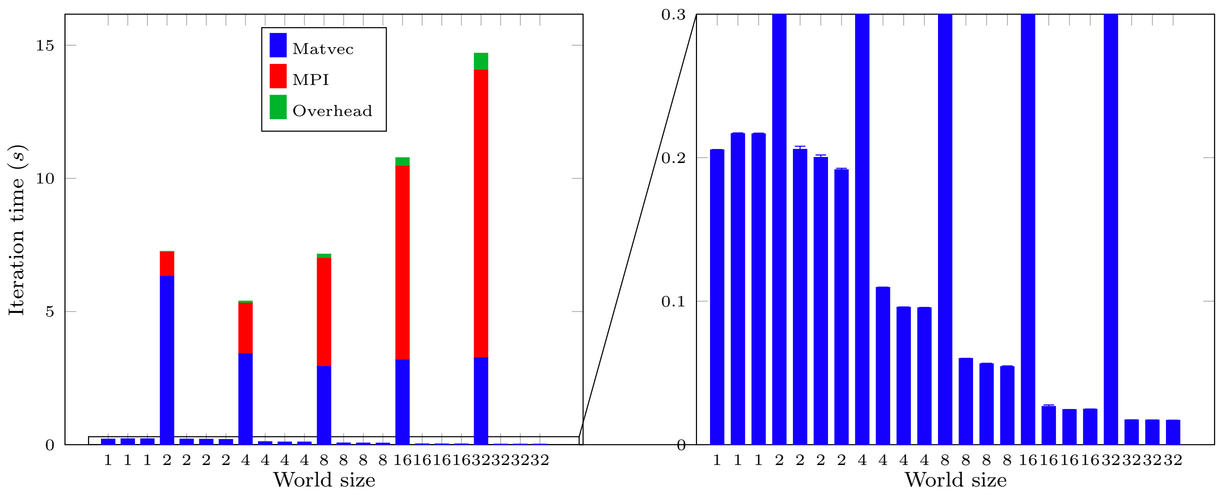


Figure 5.16: Matvec time per iteration for process duplication every 4 iterations in a 65536×65536 matrix with DPM using the byOne. With one process per node the byOne and byHost policies are equivalent.

Figure 5.16 shows how the time per iteration evolves when the number of nodes increases by a factor of two every 4 matvec iterations. The time between resizes is very short compared with the resize itself. This is equivalent to Figure 5.15 In real use cases, the applications are not intended to change the size so often, like in the example, because the resize process needs some seconds to complete in all cases. As expected, the time per iteration in both cases decreases by about half every time the resources duplicate.

Opposite to Figure 5.15 the results in Figure 5.16 are the worst case assumption. Figure 5.16 shows that the time consumed by `MPI_Comm_spawn` and `MPI_Comm_merge` grows linearly with the number of steps executed; this is expected when using the **byOne** policy.

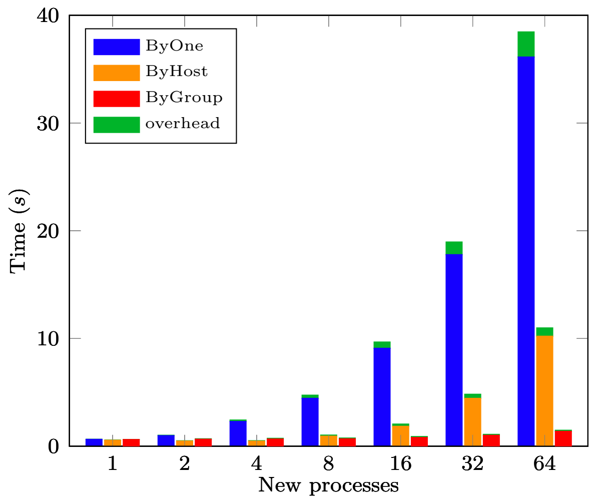
Another important detail from Figure 5.16 and Figure 5.15 is that the iteration just after the resize takes much longer than the others with the same size. The first tasks offloaded to the new ranks need to complete some data transfers before starting the computation. We do not represent such transfer time individually because it is part of the tasks overhead, but also because they take place asynchronously after the resize process finish.

After the first iteration, the other tasks are much faster because the WriteID optimization (see: Section 3.6.2) go into action to avoid redundant transfers. The taskwait needed to time the iterations to inhibit the namespace propagation. For this problem size, the transfers add overheads in the order of some seconds.

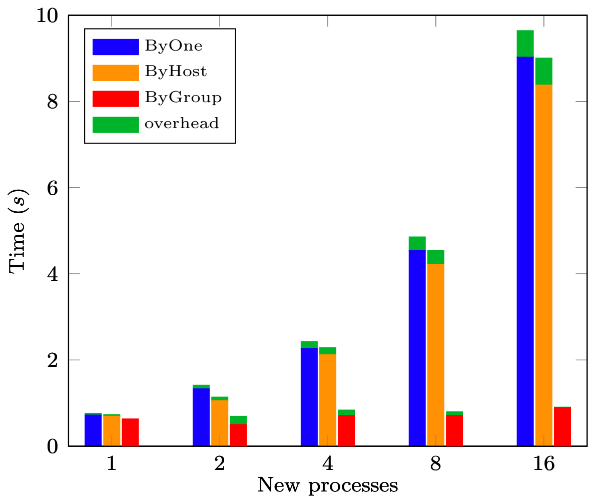
Considering that the matrix has 65536×65536 double precision elements. In the best case; when the number of processes goes from 16 to 32; every new rank will request at least 1 GB from the old ones. Besides that, most of the old ranks will request some data from others in order to equally redistribute part of the work and data.

Comparing Figure 5.16 with Figure 5.13, we see that the checkpoint and restart approach also takes some seconds to complete a resize. The iteration times are in the same order and the behaviour of the time per iteration is very similar.

Comparing the DPM with C/R in the two cases, the resize time is in the order of some seconds. For a few processes C/R is slower than DPM with the worst policy. Nevertheless, an important difference is that the C/R time decreases with the number of processes. This is because the parallel IO operations to checkpoint and restart the data are faster with more processes, especially in configurations with only one process per node.



(a) Matrix 32768×32768, 4 processes per node



(b) Matrix 65536×65536, 1 process per node

Figure 5.17: Spawn time duplicating the number of processes with the 3 policies.

Figure 5.17 Compares the time to duplicate the number of processes using the 3 spawn policies

(see: Section 5.4.1.3). Figure 5.17a shows the case when the application has 4 processes per node and Figure 5.17b a single process per node. Unlike Figure 5.16, this graph does not take into account the overhead in the first task after the spawn because that is specific to every problem. The bars include the total times consumed by `MPI_Comm_spawn` and `MPI_Comm_merge` in every policy, while the green overhead is the extra cost of the whole process, including the init messages and synchronizations as shown in Figure 5.9. The spawn time is independent of the problem itself.

The **byGroup** policy time stays constant independently of the number of processes spawned in this policy, there is almost not overhead because no intermediate communications take place. The only possible source of overhead in this policy is the synchronization point in the resize finalization function (see: Figure 5.9).

In the other extreme, the **byOne** policy takes > 35 seconds to spawn 64 new processes on 16 new nodes. With this policy `MPI_Comm_spawn` is called 64 times, consuming 94% of the total time.

Finally, the **byHost** policy in Figure 5.17 is 4 times faster than **byOne** because it only calls `MPI_Comm_spawn` once per host, spawning 4 processes at the time.

Figure 5.17b shows the same benchmark as Figure 5.17a but running one process per node. As expected, in this case the **byOne** policy is almost the same as **byHost** (see: Section 5.4.1.3).

Spawn size	MPI Time (s)	sem	Total Time (s)	sem	MPI %
1	0.66	$2 \cdot 10^{-2}$	0.7	$2 \cdot 10^{-2}$	94.87
2	1.03	$5 \cdot 10^{-2}$	1.11	$5 \cdot 10^{-2}$	93.29
4	2	$6 \cdot 10^{-2}$	2.15	$6 \cdot 10^{-2}$	92.8
8	4.09	$1 \cdot 10^{-1}$	4.41	$1 \cdot 10^{-1}$	92.78
16	8.06	$2 \cdot 10^{-1}$	8.67	$2 \cdot 10^{-1}$	92.92

Table 5.2: Spawn time with ByHost policy and 1 process per node. More than 92% of the time is spent in `MPI_Comm_spawn`.

In Table 5.2 contain the data for the **byHost** entries in Figure 5.17b. The table shows that more than 92% of the spawn time is consumed by the `MPI_Comm_spawn`. We expect that this will be improved once the new sessions features will be available and optimized in the MPI libraries.

Unlike DPM, the full time depends on the total data to checkpoint and the initial number of processes. The approximated time to do a C/R malleable operation may be calculated with the table by adding the checkpoint time for the initial size with the srun and restart time for the final size. $t(a->b)_{CR} = t_{checkpoint}(a) + t_{srun}(b) + t_{restart}(b)$.

5.7.2 Shrink

5.7.2.1 Checkpoint and restart for shrinking

When using C/R to reduce the number of nodes, the time to perform the resize remains relatively constant. The time per iteration is in the same order as the results in Figure 5.13, from a few seconds to almost half a minute, scaling inversely with the number of nodes. This is expected because the C/R mechanism is the same independently of the reconfiguration type (spawn or shrink).

Most of the comments and explanations made for Figure 5.13 in Section 5.7.1.1 are also valid for Figure 5.18

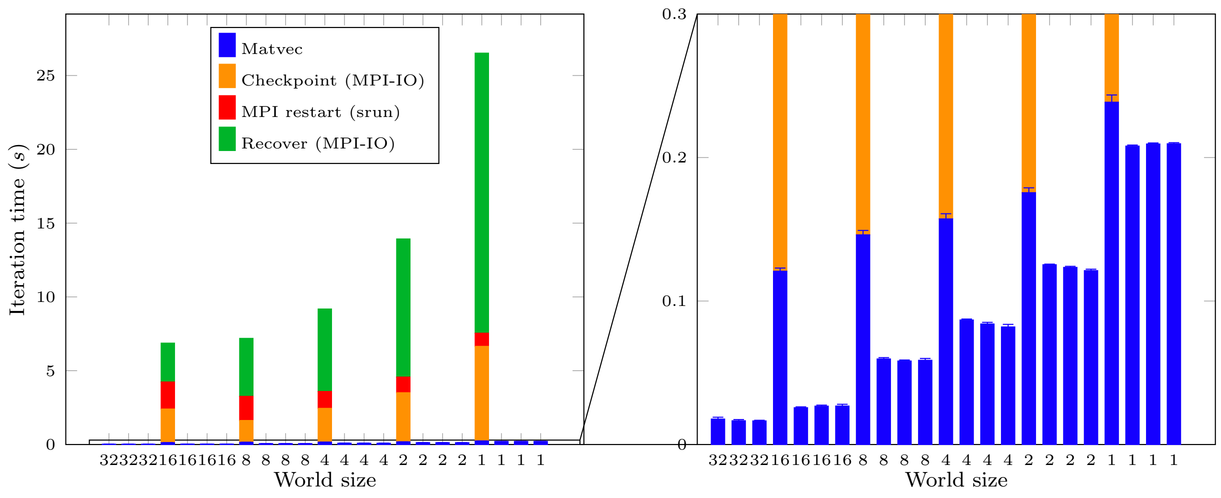
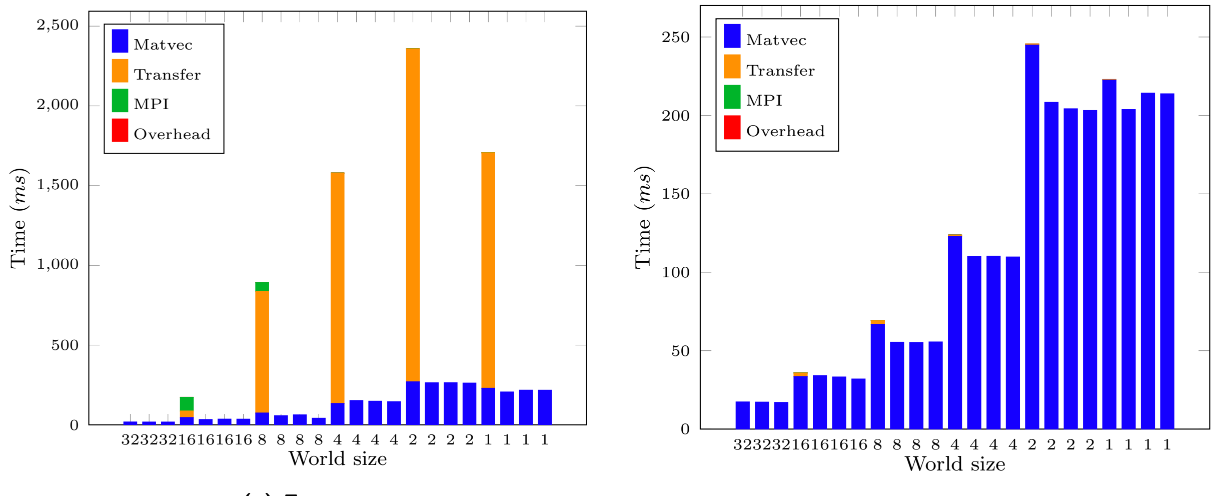


Figure 5.18: Matvec time per iteration for process halving every 4 iterations in a 65536×65536 matrix with C/R.

5.7.2.2 Dynamic process management for shrinking



(a) Eager **(b) Lazy**

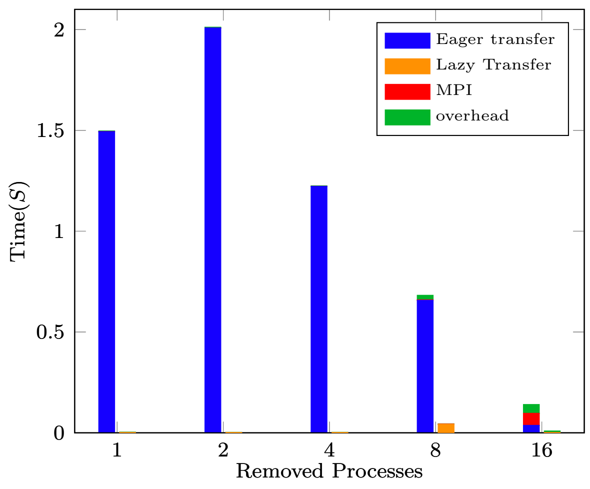
Figure 5.19: Matvec time per iteration for process halving every 4 iterations in a 65536×65536 matrix with DPM.

Figure 5.19 shows a timeline of the time per iteration. The axes are the same as Figure 5.16, but, as indicated on the x-axis, the program starts with 32 nodes and successively halves the number of nodes until just one node is left. The zoom image is unneeded in this case because the shrink times are much faster than the spawn, and the overhead is minimal. Because there are no blocking collective operations involved.

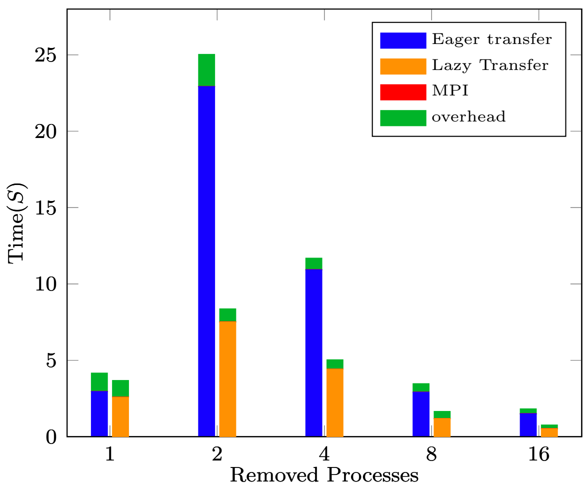
Figure 5.19a shows how the time required for the data transfers is in the order of the seconds, even using the optimized transfer approach and de-fragmenting transfers. Unlike Figure 5.16, when shrinking the spent time goes almost exclusively to data transfers.

In Figure 5.19, we see that with the lazy policy, there are no data transfers for the shrink operation as the inputs are already duplicated on multiple nodes, and there is always at least one copy among the nodes that will remain. The exiting nodes will only distribute back any data that they have exclusively and is not duplicated (i.e. the output vector). Those transfers will also go to different nodes in parallel.

As the Section 5.6.3.2 explains, the times dependent on the application data distribution and execution and depending on the problem size and data fragmentation, and spreading the results may be very different.



(a) matvec



(b) Cholesky

Figure 5.20: Shrink time halving the number of processes of matvec and MPI-IO backend with IntelMPI.

Figure 5.20 compares the time needed to remove half of the nodes for matvec and Cholesky. The Section 5.6.3.2 explains the difference between the two benchmarks especially associated with the initial and final data distribution in every case.

The data transfer time is more than 10 times larger for Cholesky than for matvec for the same problem size and policy. The Section 5.6.3.2 explains that the total amount of data to migrate is also the same in the two benchmarks. The key difference is that the Cholesky has smaller and non-contiguous regions to transfer from each node.

This comparison exposes the impact of the memory locations and data distribution in the shrink times. The results also show that even the lazy policy can take several seconds to complete when the data is very spread out in different nodes.

5.7.3 Conclusion

Malleability is the ability of an application to change its resource allocation while it is running, which makes a more efficient use of resources and improves system utilization.

OmpSs-2@Cluster applications naturally support malleability because the virtual memory layout is independent of the resource allocation and all data distribution and task scheduling is delegated to the runtime. We exploit this feature by extending Nanos6 to dynamically add and remove computational resources transparently, automatically taking care of resource allocation through Slurm, starting/stopping processes using MPI, data migration and updating of data affinity. We also define and implement a C/R API as a baseline approach for malleability. Our results show that the malleability approach with dynamic processes is promising, but its full potential will only be realized once MPI sessions is well supported in the MPI implementations. The design and implementation are ready, and this line of the research will continue in future work.

Chapter 6

Conclusions and Future work

6.1 Conclusions

We developed OmpSs-2@Cluster, an extension of the OmpSs-2 programming model for distributed memory systems. Our model simplifies the porting of HPC applications due to its sequential semantics and common address space. The runtime system automates many aspects, such as task ordering and data transfers, and it includes many performance optimizations. The results in Chapter 3 show that the approach is effective for small to medium numbers of nodes, even for relatively fine task granularities and significant numbers of communications. For larger numbers of nodes, there are still multiple optimization opportunities to explore. The results in Chapter 4 proved the effectiveness and value of breaking the confinement to a node when using Dynamic Load Balancing (DLB) to solve dynamic load imbalance problems, taking advantage of the offload capabilities of OmpSs-2@Cluster. Finally, we added malleability features to the runtime and programming model Application Programming Interface (API) in order to dynamically manage the cluster computing resources.

The first version of the runtime was developed following the OmpSs-2 programming specifications, with special emphasis on the two main new features of OmpSs-2 beyond OmpSs-1: weak dependencies and early release. The implementation was tested and profiled with multiple unit tests created to test specific features and with hundreds of randomly-generated tests (Section 3.7).

In Chapter 3, we developed the fundamental concepts of OmpSs-2@Cluster and profiled it with multiple implementations of four basic synthetic benchmarks (Section 3.8) on up to 32 nodes. We show that a pure OmpSs-2@Cluster program can scale to about 16 or 32 nodes, depending on the communication pattern and the task granularity, meaning that a small- to medium-scale program can be written using a single task-based programming model. We motivated and described a number of runtime optimizations discovered through experience that was needed to achieve this level of scalability. We also described how the OmpSs-2-TestGen program and automated test program simplification were crucial to achieving high stability of the runtime. This work was published at EuroPar 2022 [4]. In short, we established that OmpSs-2@Cluster is a viable alternative to Message Passing Interface (MPI) for small or medium-scale clusters.

The true value of OmpSs-2@Cluster, however, is not as an alternative to MPI for existing programs, but as a way to automate aspects that are currently difficult to achieve using the MPI model, such as dynamic load balancing, malleability and resilience.

In Chapter 4, we modified OmpSs-2@Cluster for interoperability with MPI for the purposes of multi-node dynamic load balance. An existing MPI+OmpSs-2 program can be executed using our Nanos6@Cluster runtime system. If the load is balanced across the nodes, then the execution will be as normal for an MPI+OmpSs-2 program. However, if there is a load imbalance, then the runtime system will dynamically offload parts of the computation among nodes in order to balance the load. This work was published at ICPP 2022 [5].

In Chapter 5, we took a different approach by extending the runtime to transparently support malleability. An OmpSs-2@Cluster program is written in a way that is essentially independent of the number of nodes, and scheduling and data distribution are handled transparently by the runtime. All implementation details can be hidden in the runtime system, with only a minimal API to control malleability exposed to the application. We also added an API to semi-transparently Checkpoint and Restart (C/R) a running OmpSs-2@Cluster program. The work in Chapter 5 will be submitted to an international peer-reviewed conference or journal.

In summary, this thesis has developed a stable high-performance offloading extension of OmpSs-2 and demonstrated its utility as an alternative to MPI at small to medium scale, as a way to enable transparent multi-node load balancing and as a way to enable semi-transparent malleability.

6.2 Future work

6.2.1 Larger applications and comparison with state of the art

OmpSs-2@Cluster has so far been evaluated using four benchmarks (matvec, matmul, Jacobi and Cholesky) and four mini-applications (MiniFE, MicroPP, InfOli and DMRG). [1] Future work should extend the evaluation to include larger applications, potentially exposing the need for additional programming model support or runtime optimizations. With larger applications, it will be possible to detect common application execution, code and memory access patterns that the runtime system could optimize either automatically or with extra hints in the code.

We also plan to perform a quantitative comparison with other frameworks and programming models apart from MPI+X, in order to learn from their experience and optimizations and determine which techniques are applicable to the OmpSs-2@Cluster programming model and runtime. The initial frameworks already in consideration are Charm++ [112, 2, 141, 142], due to its relevance and support in the three sections of this thesis; CHAMELEON [116], considering its similarity to the approach presented in Chapter 4 and its support for other alternative load balancing strategies; and DMRLib [104] for the malleability features, considering that it already covers MPI, OmpSs and integration with Slurm.

6.2.2 Multi-node dynamic load balancing using a dynamic graph

The multi-node dynamic load balancing approach executes offloaded tasks in a separate process per apprank (application rank). This ensures that the appranks have isolated virtual address

[1]MiniFE, InfOli and DMRG are not used in this thesis.

spaces so that objects on different appranks may be allocated at the same virtual address without interference. We believe this is the right approach, as it simplifies the porting of existing programs.

The current approach uses a fixed number of helper ranks per node, which is a parameter (the offloading degree) that must be provided by the user before the execution. Each helper rank requires at least one core, so there is a tradeoff between wasted cores for unnecessary helper ranks (if the application is balanced) and the ability to mitigate load imbalance (if highly imbalanced). We generally found that an offloading degree of 3 to 4 was optimal on MareNostrum 4. A better approach would be to start with no helper ranks and create helper ranks dynamically if they are needed. This would allow the execution to adapt to the program and system characteristics, and it would remove the offloading degree parameter. Doing so is a planned and natural extension of the work in this thesis. An important precursor has already been implemented in the runtime for Chapter 5: the ability to dynamically spawn (and remove) processes.

6.2.3 Combining MPI, dynamic load balancing and malleability

The contributions of Chapter 4 (multi-node dynamic load balancing) and Chapter 5 (malleability) are currently entirely separate and incompatible. In fact, they are currently implemented in different branches of the Nanos6@Cluster runtime. Research is planned to understand how to combine these different approaches so that an MPI+OmpSs-2 program can dynamically adapt to a varying number of nodes, maintaining a fixed number of application-visible MPI ranks, while dynamically adding and removing nodes that are used to execute offloaded tasks.

6.2.4 Irregular programs

OmpSs-2@Cluster is the first programming model that seriously extends a flexible task offloading mechanism among nodes, with sequential semantics, a common address space, and nested tasks with weak and strong dependencies. As such, there are questions related to irregular data structures, support for C++-style containers, returning memory allocated by subtasks, and so on, that could be addressed in future work.

6.2.5 Resilience

While Chapter 5 developed a C/R API, no resilience features have been developed in the runtime. Proper support for resilience could be an interesting avenue for future work. The runtime could automate decisions on when to checkpoint, and it should make use of a mature and highly efficient semi-transparent or non-transparent C/R library such as FTI [27], CLIP [51] or SRS [184]. With careful design, knowledge of the task graph and the lack of any communication other than via the tasks may allow the runtime to restart only the failing nodes [152].

6.2.6 Parallel IO

Questions of parallel Input Output (IO) have not yet been investigated in detail. The implementation of C/R API uses MPI-IO internally in the runtime, but it is not clear whether the same model should be recommended to the user as the right way to perform parallel IO in an OmpSs-2@Cluster application.

6.2.7 Scheduling and data distribution

Due to the huge body of existing work, this thesis has not concentrated on scheduling and data distribution policies. Future work could leverage this body of work to improve these aspects of the runtime.

6.2.8 Summary

In summary, by building a stable and high-performance task offloading programming model and runtime system, this thesis has built the fundamentals for a multi-node programming model that addresses scalability, dynamic load balance and malleability. We hope that the ideas in this thesis will be expanded upon to improve the performance, performance portability and usage of future High-Performance Computing (HPC) applications.

Glossary

apprank Application rank: MPI rank that runs the `main` function in an MPI+OmpSs-2 program

ClusterPlace Compute place abstraction for MPI processes in Nanos6

CPUSET Linux kernel mechanism to assign a set of Central Processor Units (CPUs) and Memory Nodes to a set of processes

evolving Application that can change the number of processors during execution, with changes triggered by the application itself

expander graph Sparse graph with strong connectivity properties, where every subset of the vertices (or edges) is collectively adjacent to a "large" number of vertices (or edges).

Extrae Barcelona Supercomputing Center (BSC)'s package for generation of Paraver trace files for a post-mortem analysis

home node In OmpSs-2@Cluster, the node for which a region of memory has affinity (see Section 3.3.5). It is a hint for scheduling and the node to which data that would be lost by a malleability operation gets copied; the home node has no special responsibilities in terms of tracking data location, copy back, etc

Hybrid programming Mixing different programming models, frameworks and tools in the same application

Integer Linear Program Program where the variables are integer values, and the objective function and equations are linear.

malleable Application that can change the number of processors during execution, with changes triggered by an external resource management system

Mercurium BSC's source-to-source compiler, mainly used with Nanos++ or Nanos6 to implement OpenMP and OmpSs/glsOmpSs-2

moldable Application that can run on a flexible number of processors. However, the allocation of processors remains fixed during the runtime of the application

MPI+X General term for Hybrid programming involving MPI together with another programming model such as OmpSs or OpenMP, denoted X

MPI-IO Low-level-interface for MPI parallel IO

namespace Namespace task, present on every OmpSs-2@Cluster node, including the first that is the implied parent of all offloaded tasks, facilitating task-to-task dependencies among offloaded tasks

Nanos++ BSC's Runtime system used to support OmpSs-1

Nanos6 BSC's Runtime system used to support OmpSs-2

Nanos6@Cluster BSC's fork of Nanos6 that is the runtime system used to support OmpSs-2@Cluster

offloading degree Total number of worker processes per apprank

OmpSs BSC's extension of OpenMP with new directives to support asynchronous parallelism and heterogeneity

OmpSs-2 BSC's extension of OmpSs to support task nesting and fine-grained dependencies across nesting levels

OmpSs-2-TestGen Random test generator for OmpSs-2, which was used to discover corner cases in Nanos6@Cluster

OmpSs-2@Cluster BSC's extension of OmpSs-2 to support task offloading with transparent multi-node dynamic load balancing and malleability

OpenMP Open Multiprocessing programming model for shared memory systems

Paraver BSC's flexible parallel program trace visualization and analysis tool

PMPI Message Passing Interface (MPI) Profiling Interface

rigid Application that requires a fixed allocation of processors. Once the number of processors is configured, the application cannot be executed on a smaller or larger number of processors

Slurm Slurm workload manager, formerly known as Simple Linux Utility for Resource Management, used on many supercomputers and clusters

strong access An access that is not weak, i.e. it defines a region of memory that is only by the task itself.

strong task Task with one or more strong accesses

taskfor In OmpSs-2@Cluster, work sharing task where a for loop body is executed by multiple threads within the same task, similar to OpenMP Parallel for constructor.

taskloop In OmpSs-2@Cluster, annotation to decompose a for loop into multiple independent tasks.

weak access A access that defines a region of memory that is only accessed by subtasks. Such an access is necessary as a linking point between dependency domains, but it does not enforce task ordering or in itself require data transfers.

weak task Task that has only weak accesses

Acronyms

ADLB Asynchronous Dynamic Load Balancing

AGAS Active Global Address Space

AMPI Adaptive MPI

AMR Adaptive Mesh Refinement

AMS EasyGrid Application Management System

APGAS Asynchronous Partitioned Global Address Space

API Application Programming Interface

BLAS Basic Linear Algebra Subprograms

BSC Barcelona Supercomputing Center

C/R Checkpoint and Restart

Chapel Cascade High Productivity Language

CPU Central Processor Unit

CTF Common Trace Format

CUDA Compute Unified Device Architecture

DAG Directed Acyclic Graph

DLB Dynamic Load Balancing

DMA Direct Memory Access

DMR Dynamic Management of Resources

DPM Dynamic Process Management

DROM Dynamic Resource Ownership Management

DTB	Dynamic Thread Balancing
DVFS	Dynamic Voltage and Frequency Scaling
EDAT	Event Driven Asynchronous Tasks
FLOP	Floating Point Operations
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GFLOP/s	Billion Floating Point Operations per Second
GPI	Global Address Space Programming Interface
GPU	Graphical Processor Unit
HIP	Heterogeneous Interface for Portability
HPC	High-Performance Computing
HPL	High Performance LINPACK
ILP	Integer Linear Program
IO	Input Output
LeWI	Lend When Idle
LIFO	Last In First Out
MKL	Math Kernel Library
MPI	Message Passing Interface
NUMA	Non-uniform Memory Access
OOP	Object Oriented Programming
OpenCL	Open Computing Language
ORB	Orthogonal Recursive Bisection
OS	Operating System
PARM	Power-aware resource manager
PCIe	Peripheral Component Interface Express

PCM	Process Checkpointing and Migration
PDR	Parallel Distributed Runtime
PGAS	Partitioned Global Address Space
POP	Performance Optimisation and Productivity Centre of Excellence
PUP	Pack-UnPack
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
REST	Representational State Transfer
RMI	Remote Method Invocation
RMS	Resource Management System
SDSM	Software Distributed Shared Memory
sem	Standard error of the mean
slurmrestd	Slurm REST API Daemon
SMP	Shared Memory Multiprocessor System
SPMD	Single Program Multiple Data
SSD	Solid State Drive
StarSs	Cluster Superscalar
StarSs	Star SuperScalar
STL	Standard Template Library
TALP	Tracking Application Live Performance
ULFM	User Level Failure Migration
UPC	Unified Parallel C

Bibliography

[1] Bilge Acun, Phil Miller, and Laxmikant V Kale. “Variation among processors under Turbo Boost in HPC systems”. In: *Proceedings of the 2016 International Conference on Supercomputing*. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–12. ISBN: 9781450343619. DOI: 10.1145/2925426.2926289 (cit. on pp. 5, 71).

[2] Bilge Acun et al. “Parallel Programming with Migratable Objects: Charm++ in Practice”. In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 647–658. DOI: 10.1109/SC.2014.58 (cit. on pp. 14, 124).

[3] Jimmy Aguilar Mena. *Nested slurm configuration*. 2022. URL: https://github.com/Ergus/SlurmInSlurm (visited on 08/31/2022) (cit. on p. 113).

[4] Jimmy Aguilar Mena et al. “OmpSs-2@Cluster: Distributed Memory Execution of Nested OpenMP-style Tasks”. In: *Euro-Par 2022: Parallel Processing*. Ed. by José Cano and Phil Trinder. Cham: Springer International Publishing, 2022, pp. 319–334. ISBN: 978-3-031-12597-3. DOI: 10.1007/978-3-031-12597-3_20 (cit. on pp. 7, 123).

[5] Jimmy Aguilar Mena et al. “Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB”. In: *51st International Conference on Parallel Processing (ICPP)*. 2022 (cit. on pp. 7, 124).

[6] Jose I. Aliaga et al. “A Survey on Malleability Solutions for High-Performance Distributed Computing”. In: *Applied Sciences* 12.10 (2022). ISSN: 2076-3417. DOI: 10.3390/app12105231 (cit. on p. 91).

[7] David Álvarez et al. “Advanced Synchronization Techniques for Task-Based Runtime Systems”. In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: Association for Computing Machinery, 2021, pp. 334–347. ISBN: 9781450382946. DOI: 10.1145/3437801.3441601 (cit. on pp. 3, 28).

[8] *Alya*. 2018. URL: https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational (visited on 01/10/2018) (cit. on p. 71).

[9] Martin Andersen, Joachim Dahl, and Lieven Vandenberghe. *CVXOPT: Python software for convex optimization*. 2022. URL: https://cvxopt.org/ (visited on 04/12/2022) (cit. on p. 80).

[10] Jason Ansel, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent checkpointing for cluster computations and the desktop". In: *2009 IEEE International Symposium on Parallel and Distributed Processing*. 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161063 (cit. on p. 21).

[11] Victor Anton et al. "Transparent Execution of Task-Based Parallel Applications in Docker with COMP Superscalar". In: *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. 2017, pp. 463–467. DOI: 10.1109/PDP.2017.26 (cit. on p. 13).

[12] Humayun Arafat et al. "Load Balancing of Dynamical Nucleation Theory Monte Carlo Simulations through Resource Sharing Barriers". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 2012, pp. 285–295. DOI: 10.1109/IPDPS.2012.35 (cit. on pp. 5, 17, 71).

[13] Cédric Augonnet et al. "StarPU-MPI: Task programming over clusters of machines enhanced with accelerators". In: *European MPI Users' Group Meeting*. Springer Berlin Heidelberg, 2012, pp. 298–299. ISBN: 978-3-642-33518-1. DOI: 10.1007/978-3-642-33518-1_40 (cit. on p. 16).

[14] Cédric Augonnet et al. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *CCPE - Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631 (cit. on p. 16).

[15] Rosa M. Badia et al. "COMP Superscalar, an interoperable programming framework". In: *SoftwareX* 3-4 (2015), pp. 32–36. ISSN: 2352-7110. DOI: 10.1016/j.softx.2015.10.004 (cit. on p. 13).

[16] M. Balasubramaniam et al. "A novel dynamic load balancing library for cluster computing". In: *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*. 2004, pp. 346–353. DOI: 10.1109/ISPDC.2004.5 (cit. on p. 18).

[17] Barcelona Supercomputing Center. *MareNostrum 4 (2017) System Architecture*. 2017. URL: https://www.bsc.es/marenostrum/marenostrum/technical-information (visited on 08/29/2022) (cit. on pp. 51, 82).

[18] Barcelona Supercomputing Center. *Mercurium*. 2021. URL: https://pm.bsc.es/mcxx (visited on 12/15/2021) (cit. on pp. 1, 3, 28).

[19] Barcelona Supercomputing Center. *Nanos6*. 2021. URL: https://github.com/bsc-pm/nanos6 (visited on 12/15/2021) (cit. on pp. 3, 28).

[20] Barcelona Supercomputing Center. *Nord III User's Guide*. 2021. URL: https://www.bsc.es/support/Nord3-ug.pdf (visited on 08/29/2022) (cit. on p. 82).

[21] Barcelona Supercomputing Center. *OmpSs-2 Programming Model*. 2021. URL: https://pm.bsc.es/ompss-2 (visited on 08/30/2022) (cit. on p. 1).

[22] Barcelona Supercomputing Center. *OmpSs-2 Releases.* 2021. URL: https://github.com/bsc-pm/ompss-releases (visited on 12/15/2021) (cit. on pp. 1, 3, 28).

[23] Barcelona Supercomputing Center. *OmpSs-2 Specification.* 2021. URL: https://pm.bsc.es/ftp/ompss-2/doc/spec/ (visited on 12/15/2021) (cit. on pp. 1, 3, 28, 71).

[24] Barcelona Supercomputing Center. *OmpSs-2@Cluster Releases.* 2022. URL: https://github.com/bsc-pm/ompss-2-cluster-releases (visited on 03/01/2022) (cit. on pp. 1, 3, 27, 28, 69).

[25] Patrik Barkman. *Parallel Barnes–Hut algorithm.* 2019. URL: https://github.com/barkm/n-body (visited on 04/12/2022) (cit. on p. 81).

[26] Michael Bauer et al. "Legion: Expressing locality and independence with logical regions". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 2012, pp. 1–11. DOI: 10.1109/SC.2012.71 (cit. on p. 17).

[27] Leonardo Bautista-Gomez et al. "FTI: High Performance Fault Tolerance Interface for Hybrid Systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis.* SC '11. New York, NY, USA: Association for Computing Machinery, 2011. ISBN: 9781450307710. DOI: 10.1145/2063384.2063427 (cit. on pp. 21, 22, 98, 125).

[28] Tobias Becker et al. *EuroEXA D3.2: Software Stack Release 2.* Tech. rep. July 2020 (cit. on p. 14).

[29] David A Beckingsale et al. "RAJA: Portable performance for large-scale scientific applications". In: *IEEE/ACM international workshop on performance, portability and productivity in HPC (P3HPC).* 2019. DOI: 10.1109/P3HPC49587.2019.00012 (cit. on p. 15).

[30] Dominik Bendle et al. "Integration-by-parts reductions of Feynman integrals using Singular and GPI-Space". In: *Journal of High Energy Physics* 2020.2 (Feb. 2020), p. 79. ISSN: 1029-8479. DOI: 10.1007/JHEP02(2020)079 (cit. on p. 14).

[31] John Bent et al. "PLFS: a checkpoint filesystem for parallel applications". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* 2009, pp. 1–12. DOI: 10.1145/1654059.1654081 (cit. on p. 21).

[32] F. Berman et al. "Adaptive computing on the Grid using AppLeS". In: *IEEE Transactions on Parallel and Distributed Systems* 14.4 (2003), pp. 369–382. DOI: 10.1109/TPDS.2003.1195409 (cit. on p. 21).

[33] Milind Bhandarkar et al. "Adaptive Load Balancing for MPI Programs". In: *Computational Science - ICCS 2001.* Ed. by Vassil N. Alexandrov et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 108–117. ISBN: 978-3-540-45718-3. DOI: 10.1007/3-540-45718-6_13 (cit. on p. 18).

[34] Robert Blumofe et al. "Cilk: An Efficient Multithreaded Runtime System". In: *Journal of Parallel and Distributed Computing* 37 (Feb. 1999). DOI: 10.1006/jpdc.1996.0107 (cit. on p. 11).

[35] Janko Böhm and Anne Frühbis-Krüger. "Massively Parallel Computations in Algebraic Geometry". In: *Proceedings of the 2021 on International Symposium on Symbolic and Algebraic Computation*. ISSAC '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 11–14. ISBN: 9781450383820. DOI: `10.1145/3452143.3465510` (cit. on p. 14).

[36] Dan Bonachea and Paul H. Hargrove. "GASNet-EX: A High-Performance, Portable Communication Library for Exascale". In: *Languages and Compilers for Parallel Computing*. Ed. by Mary Hall and Hari Sundar. Cham: Springer International Publishing, 2019, pp. 138–158. ISBN: 978-3-030-34627-0. DOI: `10.1007/978-3-030-34627-0_11` (cit. on p. 17).

[37] Dan Bonachea and Jaein Jeong. "Gasnet: A portable high-performance communication layer for global address-space languages". In: *CS258 Parallel Computer Architecture Project, Spring* 31 (2002) (cit. on p. 17).

[38] Jaume Bosch et al. "Application acceleration on FPGAs with OmpSs@FPGA". In: *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2018, pp. 70–77 (cit. on p. 1).

[39] George Bosilca et al. "DAGuE: A generic distributed DAG engine for High Performance Computing". In: vol. 38. May 2011, pp. 1151–1158. DOI: `10.1016/j.parco.2011.10.003` (cit. on p. 14).

[40] George Bosilca et al. "PaRSEC: Exploiting heterogeneity to enhance scalability". In: *Computing in Science & Engineering* 15.6 (2013), pp. 36–45. DOI: `10.1109/MCSE.2013.98` (cit. on p. 14).

[41] Gerandy Brito, Ioana Dumitriu, and Kameron Decker Harris. *Spectral gap in random bipartite biregular graphs and applications*. 2018. DOI: `10.48550/ARXIV.1804.07808`. URL: `https://arxiv.org/abs/1804.07808` (cit. on p. 77).

[42] Nick Brown, Oliver Thomson Brown, and J. Mark Bull. "Driving asynchronous distributed tasks with events". In: *ArXiv* abs/2010.13432 (2020). URL: `https://arxiv.org/abs/2010.13432` (cit. on p. 14).

[43] J. Bueno et al. "Productive Programming of GPU Clusters with OmpSs". In: *IEEE 26th International Parallel and Distributed Processing Symposium*. May 2012. DOI: `10.1109/IPDPS.2012.58` (cit. on pp. 2, 13).

[44] Jeremy B Buisson et al. "Scheduling malleable applications in multicluster systems". In: *2007 IEEE International Conference on Cluster Computing*. 2007, pp. 372–381. DOI: `10.1109/CLUSTR.2007.4629252` (cit. on p. 21).

[45] F. Cappello and D. Etiemble. "MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks". In: *SC '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. 2000, pp. 12–12. DOI: `10.1109/SC.2000.10001` (cit. on p. 19).

[46] Denis Caromel, Wilfried Klauser, and Julien Vayssiere. "Towards seamless computing and metacomputing in Java". In: *Concurrency: practice and experience* 10.11-13 (1998), pp. 1043–1061 (cit. on p. 16).

[47] Márcia C. Cera et al. "Supporting Malleability in Parallel Architectures with Dynamic CPUSETs Mapping and Dynamic MPI". In: *Distributed Computing and Networking: 11th International Conference, ICDCN 2010, Kolkata, India, January 3-6, 2010. Proceedings.* Ed. by Krishna Kant et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 242–257. ISBN: 978-3-642-11322-2. DOI: `10.1007/978-3-642-11322-2_26` (cit. on p. 23).

[48] Mohak Chadha. "Adaptive Resource-Aware Batch Scheduling for HPC systems". Masterarbeit. Munich: Technische Universität München, 2020 (cit. on p. 91).

[49] Mohak Chadha, Jophin John, and Michael Gerndt. "Extending SLURM for Dynamic Resource-Aware Adaptive Batch Scheduling". In: *2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC).* 2020, pp. 223–232. DOI: `10.1109/HiPC50609.2020.00036` (cit. on p. 91).

[50] Philippe 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. New York, NY, USA: Association for Computing Machinery, 2005, pp. 519–538. ISBN: 1595930310. DOI: `10.1145/1094811.1094852` (cit. on p. 15).

[51] Yuqun Chen, James S. Plank, and Kai Li. "CLIP: A Checkpointing Tool for Message-Passing Parallel Programs". In: *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing.* New York, NY, USA: Association for Computing Machinery, 1997, pp. 1–11. ISBN: 0897919858. DOI: `10.1145/509593.509626` (cit. on pp. 22, 125).

[52] Fan RK Chung. "Diameters and eigenvalues". In: *Journal of the American Mathematical Society* 2.2 (1989), pp. 187–196. DOI: `10.1090/S0894-0347-1989-0965008-X` (cit. on p. 77).

[53] Isaías Comprés et al. "Infrastructure and API Extensions for Elastic Execution of MPI Applications". In: *Proceedings of the 23rd European MPI Users' Group Meeting.* EuroMPI 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 82–97. ISBN: 9781450342346. DOI: `10.1145/2966884.2966917` (cit. on p. 24).

[54] Peter Corbett et al. "Overview of the MPI-IO Parallel I/O Interface". In: *Input/Output in Parallel and Distributed Computer Systems.* Ed. by Ravi Jain, John Werth, and James C. Browne. Boston, MA: Springer US, 1996, pp. 127–146. ISBN: 978-1-4613-1401-1. DOI: `10.1007/978-1-4613-1401-1_5` (cit. on p. 22).

[55] Ludovic Courtès. "C Language Extensions for Hybrid CPU/GPU Programming with StarPU". In: (Apr. 2013). DOI: `10.48550/ARXIV.1304.0878` (cit. on p. 16).

[56] Marco D'Amico et al. "DROM: Enabling Efficient and Effortless Malleability for Resource Managers". In: *Proceedings of the 47th International Conference on Parallel Processing Companion.* ICPP '18. Eugene, OR, USA: Association for Computing Machinery, 2018. ISBN: 9781450365239. DOI: `10.1145/3229710.3229752` (cit. on pp. 20, 73).

[57] Ewa Deelman et al. “Pegasus: A Framework for Mapping Complex Scientific Workflows onto Distributed Systems”. In: *Scientific Programming* 13.3 (Jan. 2005), pp. 219–237. DOI: 10.1155/2005/128026 (cit. on p. 13).

[58] *DEEP - Software for exascale architectures. Horizon 2020 grant agreement number 955606.* 2022. URL: https://www.deep-projects.eu/ (visited on 09/04/2022) (cit. on p. 2).

[59] Travis Desell, Kaoutar El Maghraoui, and Carlos A. Varela. “Malleable applications for scalable high performance computing”. In: *Cluster Computing* 10.3 (Sept. 2007), pp. 323–337. ISSN: 1573-7543. DOI: 10.1007/s10586-007-0032-9. URL: https://doi.org/10.1007/s10586-007-0032-9 (cit. on p. 91).

[60] Mathieu Desnoyers. *The Common Trace Format.* URL: https://diamon.org/ctf/ (visited on 08/23/2022) (cit. on p. 50).

[61] Sheng Di et al. “Optimization of Multi-level Checkpoint Model for Large Scale HPC Applications”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium.* 2014, pp. 1181–1190. DOI: 10.1109/IPDPS.2014.122 (cit. on p. 21).

[62] Vida Dujmović, Anastasios Sidiropoulos, and David R. Wood. “Layouts of Expander Graphs”. In: (2015). DOI: 10.48550/ARXIV.1501.05020. URL: https://arxiv.org/abs/1501.05020 (cit. on p. 77).

[63] Alejandro Duran, Marc Gonzàlez, and Julita Corbalán. “Automatic Thread Distribution for Nested Parallelism in OpenMP”. In: *Proceedings of the 19th Annual International Conference on Supercomputing.* ICS ’05. Cambridge, Massachusetts: Association for Computing Machinery, 2005, pp. 121–130. ISBN: 1595931678. DOI: 10.1145/1088149.1088166 (cit. on p. 19).

[64] Alejandro Duran et al. “Ompss: a proposal for programming heterogeneous multi-core architectures”. In: *Parallel processing letters* 21.02 (2011), pp. 173–193 (cit. on p. 1).

[65] Richard Dutton and Weizhen Mao. “Online scheduling of malleable parallel jobs”. In: *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems* (Jan. 2007) (cit. on p. 21).

[66] H. Carter Edwards and Christian R. Trott. “Kokkos: Enabling Performance Portability Across Manycore Architectures”. In: *2013 Extreme Scaling Workshop (xsw 2013).* 2013, pp. 18–24. DOI: 10.1109/XSW.2013.7 (cit. on p. 15).

[67] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3202–3216. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.07.003 (cit. on p. 15).

[68] K. El Maghraoui et al. “Malleable iterative MPI applications”. In: *Concurrency and Computation: Practice and Experience* 21.3 (2009), pp. 393–413. ISSN: 1532-0634. DOI: 10.1002/cpe.1362. URL: http://dx.doi.org/10.1002/cpe.1362 (cit. on p. 20).

[69] E. N. Elnozahy et al. “A Survey of Rollback-Recovery Protocols in Message-Passing Systems”. In: *ACM Comput. Surv.* 34.3 (Sept. 2002), pp. 375–408. ISSN: 0360-0300. DOI: 10.1145/568522.568525 (cit. on p. 21).

[70] M. Etinski et al. “Optimizing job performance under a given power constraint in HPC centers”. In: *International Conference on Green Computing.* Aug. 2010, pp. 257–267. DOI: 10.1109/GREENCOMP.2010.5598303 (cit. on pp. 5, 91).

[71] Karl-Filip Faxén. *Wool user’s guide.* Tech. rep. Technical report, Swedish Institute of Computer Science, 2009 (cit. on p. 15).

[72] Dror G. Feitelson and Larry Rudolph. “Toward convergence in job schedulers for parallel supercomputers”. In: *Job Scheduling Strategies for Parallel Processing.* Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–26. ISBN: 978-3-540-70710-3. DOI: 10.1007/BFb0022284 (cit. on p. 20).

[73] Dror G. Feitelson et al. “Theory and practice in parallel job scheduling”. In: *Job Scheduling Strategies for Parallel Processing.* Ed. by Dror G. Feitelson and Larry Rudolph. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 1–34. ISBN: 978-3-540-69599-8. DOI: 10.1007/3-540-63574-2_14 (cit. on p. 21).

[74] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0.* July 1997 (cit. on pp. 1, 2, 22).

[75] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 3.0.* Sept. 2012 (cit. on p. 1).

[76] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0.* June 2021 (cit. on pp. 1, 96).

[77] I. Foster et al. “Chimera: a virtual data system for representing, querying, and automating data derivation”. In: *Proceedings 14th International Conference on Scientific and Statistical Database Management.* 2002, pp. 37–46. DOI: 10.1109/SSDM.2002.1029704 (cit. on p. 13).

[78] Karl Fuerlinger, Tobias Fuchs, and Roger Kowalewski. “DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms”. In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS).* 2016, pp. 983–990. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0140 (cit. on p. 12).

[79] Karl Fürlinger et al. “DASH: Distributed Data Structures and Parallel Algorithms in a Global Address Space”. In: *Software for Exascale Computing-SPPEXA 2016-2019.* Springer International Publishing, July 2020, pp. 103–142. ISBN: 978-3-030-47956-5. DOI: 10.1007/978-3-030-47956-5_6 (cit. on p. 12).

[80] Marta Garcia, Julita Corbalan, and Jesus Labarta. “LeWI: A Runtime Balancing Algorithm for Nested Parallelism”. In: *2009 International Conference on Parallel Processing.* 2009, pp. 526–533. DOI: 10.1109/ICPP.2009.56 (cit. on pp. 20, 72, 73).

[81] Marta Garcia, Jesus Labarta, and Julita Corbalan. “Hints to improve automatic load balancing with LeWI for hybrid applications”. In: *Journal of Parallel and Distributed Computing* 74.9 (2014), pp. 2781–2794. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2014.05.004 (cit. on pp. 2, 73).

[82] Marta Garcia et al. “A Dynamic Load Balancing Approach with SMPSuperscalar and MPI”. In: *Facing the Multicore - Challenge II: Aspects of New Paradigms and Technologies in Parallel Computing*. Ed. by Rainer Keller, David Kramer, and Jan-Philipp Weiss. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 10–23. ISBN: 978-3-642-30397-5. DOI: 10.1007/978-3-642-30397-5_2 (cit. on pp. 71, 73).

[83] Marta Garcia Gasulla, Julita Corbalán González, and Jesús José Labarta Mancho. “Dynamic load balancing for hybrid applications”. PhD thesis. Universitat Politecnica de Catalunya, 2017 (cit. on p. 78).

[84] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. “Kaapi: A thread scheduling runtime system for data flow computations on cluster of multi-processors”. In: *Proceedings of the 2007 international workshop on Parallel symbolic computation*. 2007, pp. 15–23 (cit. on p. 15).

[85] Neha Gholkar, Frank Mueller, and Barry Rountree. “Power Tuning HPC Jobs on Power-Constrained Systems”. In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT 16. Haifa, Israel: ACM, 2016, pp. 179–191. ISBN: 978-1-4503-4121-9. DOI: 10.1145/2967938.2967961 (cit. on pp. 5, 91).

[86] Guido Giuntoli et al. “A FE 2 multi-scale implementation for modeling composite materials on distributed architectures”. In: *Coupled Systems Mechanics* 8.2 (2019), p. 99. DOI: 10.12989/csm.2019.8.2.099 (cit. on pp. 7, 81).

[87] Etienne Godard, Sanjeev Setia, and Elizabeth White. “DyRecT: Software Support for Adaptive Parallelism on NOWs”. In: *Parallel and Distributed Processing*. Ed. by José Rolim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 1168–1175. ISBN: 978-3-540-45591-2. DOI: 10.1007/3-540-45591-4_161 (cit. on p. 22).

[88] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)* USA: Johns Hopkins University Press, 1996. ISBN: 0801854148 (cit. on p. 54).

[89] W. Gropp and E. Lusk. “Dynamic process management in an MPI setting”. In: *Proceedings.Seventh IEEE Symposium on Parallel and Distributed Processing*. Oct. 1995, pp. 530–533. DOI: 10.1109/SPDP.1995.530729 (cit. on p. 22).

[90] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. Cambridge, MA, USA: MIT Press, 1999. ISBN: 0262571331 (cit. on p. 22).

[91] Thomas Gruber et al. *LIKWID*. Version v5.2.2. Aug. 2022. DOI: 10.5281/zenodo.6980692 (cit. on p. 20).

[92] A. Gupta et al. “Towards realizing the potential of malleable jobs”. In: *2014 21st International Conference on High Performance Computing (HiPC)*. Dec. 2014, pp. 1–10. DOI: 10.1109/HiPC.2014.7116905 (cit. on pp. 20, 22).

[93] D.S. Henty. “Performance of Hybrid Message-Passing and Shared-Memory Parallelism for Discrete Element Modeling”. In: *SC ’00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. 2000, pp. 10–10. DOI: 10.1109/SC.2000.10005 (cit. on p. 19).

[94] Nathan Hjelm et al. “MPI Sessions: Evaluation of an Implementation in Open MPI”. In: *2019 IEEE International Conference on Cluster Computing (CLUSTER)*. 2019, pp. 1–11. DOI: 10.1109/CLUSTER.2019.8891002 (cit. on p. 96).

[95] Daniel Holmes et al. “MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale”. In: *Proceedings of the 23rd European MPI Users’ Group Meeting*. EuroMPI 2016. Edinburgh, United Kingdom: Association for Computing Machinery, 2016, pp. 121–129. ISBN: 9781450342346. DOI: 10.1145/2966884.2966915 (cit. on pp. 6, 96).

[96] Shlomo Hoory, Nathan Linial, and Avi Wigderson. “Expander graphs and their applications”. In: *Bulletin of the American Mathematical Society* 43.4 (2006), pp. 439–561. DOI: 10.1090/S0273-0979-06-01126-8 (cit. on p. 77).

[97] Reazul Hoque et al. “Dynamic task discovery in PaRSEC: a data-flow task-based runtime”. In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. Denver, Colorado, Nov. 2017, pp. 1–8. DOI: 10.1145/3148226.3148233 (cit. on pp. 4, 14).

[98] Chao Huang, Orion Lawlor, and L. V. Kalé. “Adaptive MPI”. In: *Languages and Compilers for Parallel Computing: 16th International Workshop, LCPC 2003, College Station, TX, USA, October 2-4, 2003. Revised Papers*. Ed. by Lawrence Rauchwerger. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 306–322. ISBN: 978-3-540-24644-2. DOI: 10.1007/978-3-540-24644-2_20 (cit. on pp. 18, 22, 23).

[99] J. Hungershofer. “On the combined scheduling of malleable and rigid jobs”. In: *16th Symposium on Computer Architecture and High Performance Computing*. 2004, pp. 206–213. DOI: 10.1109/SBAC-PAD.2004.27 (cit. on p. 21).

[100] Yuichi Inadomi et al. “Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing”. In: *SC ’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807638 (cit. on pp. 5, 71).

[101] S. Iserte et al. “Efficient Scalable Computing through Flexible Applications and Adaptive Workloads”. In: *2017 46th International Conference on Parallel Processing Workshops (ICPPW)*. Aug. 2017, pp. 180–189. DOI: 10.1109/ICPPW.2017.36 (cit. on pp. 24, 112).

[102] Sergio Iserte. “High-throughput computation through efficient resource management”. PhD thesis. Universitat Jaume I, 2018 (cit. on pp. 20, 25, 97).

[103] Sergio Iserte et al. “DMR API: Improving cluster productivity by turning applications into malleable”. In: *Parallel Computing* 78 (2018), pp. 54–66. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2018.07.006. URL: https://www.sciencedirect.com/science/article/pii/S0167819118302229 (cit. on pp. 20, 24, 112).

[104] Sergio Iserte et al. “DMRlib: Easy-Coding and Efficient Resource Management for Job Malleability”. In: *IEEE Transactions on Computers* 70.9 (2021), pp. 1443–1457. DOI: 10.1109/TC.2020.3022933 (cit. on pp. 20, 24, 92, 112, 124).

[105] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. “Topology and Affinity Aware Hierarchical and Distributed Load-Balancing in Charm++”. In: *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*. 2016, pp. 63–72. DOI: 10.1109/COMHPC.2016.012 (cit. on p. 18).

[106] Emmanuel Jeannot et al. “Communication and topology-aware load balancing in Charm++ with TreeMatch”. In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702666 (cit. on p. 18).

[107] Karpjoo Jeong and D. Shasha. “PLinda 2.0: a transactional/checkpointing approach to fault tolerant Linda”. In: *Proceedings of IEEE 13th Symposium on Reliable Distributed Systems*. 1994, pp. 96–105. DOI: 10.1109/RELDIS.1994.336905 (cit. on p. 22).

[108] Haoqiang Jin and Rob F. Van der Wijngaart. “Performance characteristics of the multi-zone NAS parallel benchmarks”. In: *Journal of Parallel and Distributed Computing* 66.5 (2006). IPDPS ’04 Special Issue, pp. 674–685. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2005.06.016 (cit. on pp. 5, 17, 71).

[109] Hartmut Kaiser, Maciek Brodowicz, and Thomas Sterling. “ParalleX an advanced parallel execution model for scaling-impaired applications”. In: *2009 International Conference on Parallel Processing Workshops*. Oct. 2009. DOI: 10.1109/ICPPW.2009.14 (cit. on p. 15).

[110] Hartmut Kaiser et al. “HPX: A task based programming model in a global address space”. In: *8th International Conference on Partitioned Global Address Space Programming Models*. 2014. DOI: 10.13140/2.1.2635.5204 (cit. on p. 15).

[111] L. V. Kale, S. Kumar, and J. DeSouza. “A Malleable-Job System for Timeshared Parallel Machines”. In: *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*. May 2002, pp. 230–230. DOI: 10.1109/CCGRID.2002.1017131 (cit. on p. 23).

[112] Laxmikant Kale and Sanjeev Krishnan. “CHARM++: A portable concurrent object oriented system based on C++”. In: *ACM Sigplan Notes* 28 (Oct. 1995). DOI: 10.1145/167962.165874 (cit. on pp. 14, 124).

[113] Nikolaos D Kallimanis, Manolis Marazakis, and Nikolaos Chrysos. “GSAS: A Fast Shared Memory Abstraction with Minimal Hardware Support”. In: () (cit. on p. 14).

[114] George Karypis and Vipin Kumar. *METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0*. Tech. rep. 1995 (cit. on p. 18).

[115] Hamidreza Khaleghzadeh et al. “Bi-Objective Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms for Performance and Energy Through Workload Distribution”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 543–560. DOI: 10.1109/TPDS.2020.3027338 (cit. on p. 19).

[116] Jannis Klinkenberg et al. "CHAMELEON: Reactive load balancing for hybrid MPI+OpenMP task-parallel applications". In: *Journal of Parallel and Distributed Computing* 138 (Dec. 2019). DOI: `10.1016/j.jpdc.2019.12.005` (cit. on pp. 12, 20, 124).

[117] Berkeley Lab. *GASNet*. 2022. URL: `https://gasnet.lbl.gov/` (visited on 08/31/2022) (cit. on p. 17).

[118] Charles E Leiserson. "The Cilk++ concurrency platform". In: *The Journal of Supercomputing* 51.3 (2010), pp. 244–257. DOI: `10.1007/s11227-010-0405-3` (cit. on p. 11).

[119] Pierre Lemarinier et al. "Architecting Malleable MPI Applications for Priority-Driven Adaptive Scheduling". In: *Proceedings of the 23rd European MPI Users' Group Meeting*. EuroMPI 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 74–81. ISBN: 9781450342346. DOI: `10.1145/2966884.2966907` (cit. on p. 25).

[120] Hui Liu. "Dynamic Load Balancing on Adaptive Unstructured Meshes". In: *2008 10th IEEE International Conference on High Performance Computing and Communications*. 2008, pp. 870–875. DOI: `10.1109/HPCC.2008.12` (cit. on pp. 5, 18, 71).

[121] Victor Lopez, Guillem Ramirez Miranda, and Marta Garcia-Gasulla. "TALP: A Lightweight Tool to Unveil Parallel Efficiency of Large-Scale Executions". In: *Proceedings of the 2021 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy*. PERMAVOST '21. Virtual Event, Sweden: Association for Computing Machinery, 2021, pp. 3–10. ISBN: 9781450383875. DOI: `10.1145/3452412.3462753` (cit. on p. 73).

[122] Victor Lopez et al. "An OpenMP Free Agent Threads Implementation". In: *OpenMP: Enabling Massive Node-Level Parallelism*. Ed. by Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg. Cham: Springer International Publishing, 2021, pp. 211–225. ISBN: 978-3-030-85262-7. DOI: `10.1007/978-3-030-85262-7_15` (cit. on p. 19).

[123] Francesc Lordan et al. "ServiceSs: An interoperable programming framework for the cloud". In: *Journal of grid computing* 12.1 (2014). DOI: `10.1007/s10723-013-9272-5` (cit. on p. 13).

[124] Ewing L Lusk, Steve C Pieper, Ralph M Butler, et al. "More scalability, less pain: A simple programming model and its implementation for extreme computing". In: *SciDAC Review* 17.1 (2010), pp. 30–37 (cit. on p. 19).

[125] Spyros Lyberis. "Myrmics: A scalable runtime system for global address spaces". PhD thesis. University of Crete, 2013 (cit. on p. 14).

[126] K. El Maghraoui et al. "Dynamic Malleability in Iterative MPI Applications". In: *Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07)*. May 2007, pp. 591–598. DOI: `10.1109/CCGRID.2007.45` (cit. on p. 23).

[127] Gonzalo Martín et al. "FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems". In: *Euro-Par 2013 Parallel Processing*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 138–149. ISBN: 978-3-642-40047-6. DOI: `10.1007/978-3-642-40047-6_16` (cit. on pp. 20, 25).

[128] Sched MD. *Slurm APIs.* URL: `https://slurm.schedmd.com/api.html` (visited on 08/06/2022) (cit. on p. 97).

[129] Sched MD. *Slurm APIs.* URL: `https://slurm.schedmd.com/plugins.html` (visited on 08/06/2022) (cit. on p. 97).

[130] Sched MD. *Slurm REST API.* URL: `https://slurm.schedmd.com/rest.html` (visited on 08/06/2022) (cit. on p. 97).

[131] Sched MD. *Slurm Tutorials.* URL: `https://slurm.schedmd.com/tutorials.html` (visited on 08/06/2022) (cit. on p. 96).

[132] Sched MD. *Slurm Workload Manager.* URL: `https://slurm.schedmd.com/overview.html` (visited on 08/06/2022) (cit. on p. 96).

[133] Sina Meraji and Carl Tropper. "Optimizing Techniques for Parallel Digital Logic Simulation". In: *IEEE Transactions on Parallel and Distributed Systems* 23.6 (2012), pp. 1135–1146. DOI: `10.1109/TPDS.2011.246` (cit. on pp. 5, 17, 71).

[134] BSC Programming Models. *Influence in OpenMP.* URL: `https://pm.bsc.es/ftp/ompss-2/doc/spec/introduction/openmp.html` (visited on 08/20/2022) (cit. on pp. 3, 11).

[135] Akira Nukada, Hiroyuki Takizawa, and Satoshi Matsuoka. "NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA". In: *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum.* 2011, pp. 104–113. DOI: `10.1109/IPDPS.2011.131` (cit. on p. 21).

[136] OpenAcc-Standard.org. *The OpenACC Application Programming Interface Version 3.2.* Nov. 2021. URL: `https://www.openacc.org/sites/default/files/inline-images/Specification/OpenACC-3.2-final.pdf` (visited on 09/01/2022) (cit. on p. 1).

[137] OpenMP Architecture Review Board. *OpenMP 4.0 Complete specifications.* July 2013 (cit. on pp. 1, 11).

[138] OpenMP Architecture Review Board. *OpenMP Application Programming Interface, Version 5.2.* Accessed: 2022-04-19. Nov. 2021. URL: `https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf` (cit. on pp. 1, 19).

[139] Nikolaos Papakonstantinou. "Combining Recursively Parallel Runtimes with Blocked-based Dependence Analysis". MA thesis. 2015 (cit. on p. 14).

[140] Nikolaos Papakonstantinou, Foivos Zakkak, and Polyvios Pratikakis. "Hierarchical Parallel Dynamic Dependence Analysis for Recursively Task-Parallel Programs". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS).* May 2016. DOI: `10.1109/IPDPS.2016.53` (cit. on p. 14).

[141] University of Illinois Parallel Programming Lab Dept of Computer Science. *Charm++ Documentation.* URL: `https://charm.readthedocs.io/en/latest/index.html` (visited on 02/01/2022) (cit. on pp. 14, 124).

[142] University of Illinois Parallel Programming Lab Dept of Computer Science. *Charm++ parallel programming framework.* URL: `https://charmplusplus.org/` (visited on 02/01/2022) (cit. on pp. 14, 124).

[143] J. M. Perez et al. "Improving the Integration of Task Nesting and Dependencies in OpenMP". In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* May 2017, pp. 809–818. DOI: `10.1109/IPDPS.2017.69` (cit. on pp. 3, 27–29).

[144] Josep Pérez, Rosa M. Badia, and Jesús Labarta. "A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures". In: *Proceedings - IEEE International Conference on Cluster Computing, ICCC.* Sept. 2008, pp. 142–151. DOI: `10.1109/CLUSTR.2008.4663765` (cit. on pp. 3, 28).

[145] Chuck Pheatt. "Intel Threading Building Blocks". In: *J. Comput. Sci. Coll.* 23.4 (Apr. 2008), p. 298. ISSN: 1937-4771 (cit. on p. 19).

[146] Judit Planas et al. "Hierarchical Task-Based Programming With StarSs". In: *The International Journal of High Performance Computing Applications* 23.3 (2009), pp. 284–299. DOI: `10.1177/1094342009106195` (cit. on p. 3).

[147] James S. Plank. "An Overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance". In: 1997 (cit. on p. 21).

[148] Jonas Posner, Lukas Reitz, and Claudia Fohry. "A Comparison of Application-Level Fault Tolerance Schemes for Task Pools". In: *Future Generation Computer Systems* 105 (2020), pp. 119–134. ISSN: 0167-739X. DOI: `https://doi.org/10.1016/j.future.2019.11.031`. URL: `https://www.sciencedirect.com/science/article/pii/S0167739X19308295` (cit. on p. 21).

[149] Suraj Prabhakaran, Marcel Neumann, and Felix Wolf. "Efficient Fault Tolerance Through Dynamic Node Replacement". In: *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).* 2018, pp. 163–172. DOI: `10.1109/CCGRID.2018.00031` (cit. on p. 21).

[150] Suraj Prabhakaran et al. "A Batch System with Efficient Adaptive Scheduling for Malleable and Evolving Applications". In: *2015 IEEE International Parallel and Distributed Processing Symposium.* 2015, pp. 429–438. DOI: `10.1109/IPDPS.2015.34` (cit. on p. 23).

[151] POP project. *POP Standard Metrics for Parallel Performance Analysis.* 2022. URL: `https://pop-coe.eu/node/69` (visited on 08/31/2022) (cit. on p. 73).

[152] Petar Radojkovic et al. *Towards resilient EU HPC systems: A blueprint.* Tech. rep. `https://resilienthpc.eu/`. European HPC resilience initiative, 2020 (cit. on p. 125).

[153] Mark Roberts and Giovanni Torres. *PySlurm: Slurm Interface to python.* URL: `https://pyslurm.github.io/` (visited on 08/06/2022) (cit. on p. 97).

[154] Tiberiu Rotaru, Mirko Rahn, and Franz-Josef Pfreundt. "MapReduce in GPI-Space". In: *Euro-Par 2013: Parallel Processing Workshops.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 43–52. ISBN: 978-3-642-54420-0. DOI: `10.1007/978-3-642-54420-0_5` (cit. on p. 14).

[155] Florentino Sainz et al. “Collective Offload for Heterogeneous Clusters”. In: *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*. 2015, pp. 376–385. DOI: `10.1109/HiPC.2015.20` (cit. on p. 24).

[156] R. Sakamoto et al. “Production Hardware Overprovisioning: Real-World Performance Optimization Using an Extensible Power-Aware Resource Management Framework”. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2017, pp. 957–966. DOI: `10.1109/IPDPS.2017.107` (cit. on pp. 5, 91).

[157] Kevin Sala, Sandra Macià, and Vicenç Beltran. “Combining One-Sided Communications with Task-Based Programming Models”. In: *2021 IEEE International Conference on Cluster Computing (CLUSTER)*. 2021, pp. 528–541. DOI: `10.1109/Cluster48925.2021.00024` (cit. on p. 27).

[158] Kevin Sala et al. “Integrating blocking and non-blocking MPI primitives with task-based programming models”. In: *Parallel Computing* 85 (2019), pp. 153–166. DOI: `10.1016/j.parco.2018.12.008` (cit. on p. 27).

[159] John K Salmon. “Parallel hierarchical N-body methods”. PhD thesis. California Institute of Technology, 1991 (cit. on p. 81).

[160] Osman Sarood et al. “Maximizing Throughput of Overprovisioned HPC Data Centers Under a Strict Power Budget”. In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 807–818. DOI: `10.1109/SC.2014.71` (cit. on p. 24).

[161] K. Schloegel, G. Karypis, and V. Kumar. “Dynamic Repartitioning of Adaptively Refined Meshes”. In: *SC ’98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 1998, pp. 29–29. DOI: `10.1109/SC.1998.10025` (cit. on pp. 5, 18, 71).

[162] Alexandre C. Sena et al. “Autonomic Malleability in Iterative MPI Applications”. In: *2013 25th International Symposium on Computer Architecture and High Performance Computing*. 2013, pp. 192–199. DOI: `10.1109/SBAC-PAD.2013.4` (cit. on p. 25).

[163] Marc Sergent et al. “Controlling the memory subscription of distributed applications with a task-based runtime system”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 318–327. DOI: `10.1109/IPDPSW.2016.105` (cit. on p. 17).

[164] Omar Shaaban et al. “Automatic aggregation of subtask accesses for nested OpenMP-style tasks”. In: *IEEE 34th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2022 (cit. on pp. 8, 33).

[165] Faisal Shahzad et al. “An Evaluation of Different I/O Techniques for Checkpoint/Restart”. In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. 2013, pp. 1708–1716. DOI: `10.1109/IPDPSW.2013.145` (cit. on p. 21).

[166] Zhi Shang. “Impact of mesh partitioning methods in CFD for large scale parallel computing”. In: *Computers & Fluids* 103 (2014), pp. 1–5. ISSN: 0045-7930. DOI: `10.1016/j.compfluid.2014.07.016` (cit. on p. 18).

[167] Lorna Smith and Mark Bull. “Development of mixed mode MPI/OpenMP applications”. In: *Scientific Programming* 9.2, 3 (2001), pp. 83–98 (cit. on p. 19).

[168] Alexander Spiegel, Dieter an Mey, and Christian Bischof. “Hybrid Parallelization of CFD Applications with Dynamic Thread Balancing”. In: *Applied Parallel Computing. State of the Art in Scientific Computing.* Ed. by Jack Dongarra, Kaj Madsen, and Jerzy Waśniewski. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 433–441. ISBN: 978-3-540-33498-9. DOI: 10.1007/11558958_51 (cit. on p. 19).

[169] G. Stellner. “CoCheck: checkpointing and process migration for MPI”. In: *Proceedings of International Conference on Parallel Processing.* 1996, pp. 526–531. DOI: 10.1109/IPPS.1996.508106 (cit. on p. 21).

[170] Rajesh Sudarsan and Calvin J. Ribbens. “ReSHAPE: A Framework for Dynamic Resizing and Scheduling of Homogeneous Applications in a Parallel Environment”. In: *2007 International Conference on Parallel Processing (ICPP 2007).* 2007, pp. 44–44. DOI: 10.1109/ICPP.2007.73 (cit. on p. 24).

[171] G. Swaminathan. “A scheduling framework for dynamically resizable parallel applications”. MA thesis. Virginia Tech, 2004. URL: http://hdl.handle.net/10919/41130 (cit. on p. 24).

[172] Sriram Tadepalli, Calvin J Ribbens, and S. Varadarajan. “GEMS: A job management system for fault tolerant grid computing”. In: *High Performance Computing Symposium 2004,* San Diego, CA, USA: Soc. for Modeling and Simulation Internat, 2004, pp. 59–66. URL: http://hdl.handle.net/10919/9661 (cit. on p. 24).

[173] Enric Tejedor and Rosa M. Badia. “COMP Superscalar: Bringing GRID Superscalar and GCM Together”. In: *2008 Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGRID).* 2008, pp. 185–193. DOI: 10.1109/CCGRID.2008.104 (cit. on p. 13).

[174] Enric Tejedor et al. “A high-productivity task-based programming model for clusters”. In: *Concurrency and Computation: Practice and Experience* 24.18 (2012), pp. 2421–2448. DOI: 10.1002/cpe.2831 (cit. on p. 13).

[175] *The Chapel Parallel Programming Language.* 2022. URL: https://chapel-lang.org/ (visited on 09/04/2022) (cit. on p. 16).

[176] *The Fortress Language Specification.* 2022. URL: https://homes.luddy.indiana.edu/samth/fortress-spec.pdf (visited on 09/04/2022) (cit. on p. 16).

[177] Martin Tillenius. “SuperGlue: A shared memory framework using data versioning for dependency-aware task-based parallelization”. In: *SIAM Journal on Scientific Computing* 37.6 (2015). DOI: 10.1137/140989716 (cit. on p. 13).

[178] BSC Tools. *Extrae Home.* URL: https://tools.bsc.es/extrae (visited on 08/23/2022) (cit. on p. 50).

[179] TOP500.org. *Top 500 The List.* 2022. URL: https://www.top500.org/lists/top500/2022/06/ (visited on 08/29/2022) (cit. on pp. 5, 91).

[180] Christian R. Trott et al. “Kokkos 3: Programming Model Extensions for the Exascale Era”. In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022). DOI: 10.1109/TPDS.2021.3097283 (cit. on p. 15).

[181] George Tzenakis et al. “BDDT: Block-Level Dynamic Dependence Analysisfor Deterministic Task-Based Parallelism”. In: *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* Vol. 47. PPoPP ’12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 301–302. ISBN: 9781450311601. DOI: 10.1145/2145816.2145864 (cit. on p. 14).

[182] Gladys Utrera, Julita Corbalan, and Jesus Labarta. “Implementing Malleability on MPI Jobs”. In: *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques.* PACT ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 215–224. ISBN: 0-7695-2229-7. DOI: 10.1109/PACT.2004.18. URL: https://doi.org/10.1109/PACT.2004.18 (cit. on p. 21).

[183] Sathish Vadhiyar and Jack Dongarra. “Self Adaptivity in Grid Computing”. In: *Concurrency and Computation: Practice and Experience* 17 (Feb. 2005). DOI: 10.1002/cpe.927 (cit. on p. 21).

[184] Sathish S. Vadhiyar and Jack J. Dongarra. “SRS: A framework for developing malleable and migratable parallel applications for distributed systems”. In: *Parallel Processing Letters* 13.02 (2003), pp. 291–312. DOI: 10.1142/S0129626403001288. eprint: http://www.worldscientific.com/doi/pdf/10.1142/S0129626403001288. URL: http://www.worldscientific.com/doi/abs/10.1142/S0129626403001288 (cit. on pp. 22, 125).

[185] Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. “A Unified Scheduler for Recursive and Task Dataflow Parallelism”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques.* 2011, pp. 1–11. DOI: 10.1109/PACT.2011.7 (cit. on p. 11).

[186] Philippe Virouleau et al. “Using data dependencies to improve task-based scheduling strategies on NUMA architectures”. In: *European Conference on Parallel Processing.* Springer. 2016, pp. 531–544. DOI: 10.1007/978-3-319-43659-3_39 (cit. on p. 15).

[187] Michele Weiland. “Chapel, Fortress and X10: novel languages for HPC”. In: *EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706* 1 (2007) (cit. on pp. 15, 16).

[188] Gosia Wrzesinska, Jason Maassen, and Henri E. Bal. “Self-Adaptive Applications on the Grid”. In: *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* PPoPP ’07. San Jose, California, USA: Association for Computing Machinery, 2007, pp. 121–129. ISBN: 9781595936028. DOI: 10.1145/1229428.1229449 (cit. on p. 21).

[189] Jie Yang, Anmol Paudel, and Satish Puri. “Spatial Data Decomposition and Load Balancing on HPC Platforms”. In: *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning).* PEARC ’19. Chicago, IL, USA: Association for Computing Machinery, 2019. ISBN: 9781450372275. DOI: 10.1145/3332186.3333266 (cit. on p. 18).

[190] Afshin Zafari. “TaskUniVerse: A Task-Based Unified Interface for Versatile Parallel Execution”. In: *Parallel Processing and Applied Mathematics.* Ed. by Roman Wyrzykowski et al. Cham: Springer International Publishing, 2018, pp. 169–184. ISBN: 978-3-319-78024-5. DOI: 10.1007/978-3-319-78024-5_16 (cit. on p. 13).

[191] Afshin Zafari, Elisabeth Larsson, and Martin Tillenius. “DuctTeip: An efficient programming model for distributed task-based parallel computing”. In: *Parallel Computing* (2019). DOI: 10.1016/j.parco.2019.102582 (cit. on p. 12).

[192] Afshin Zafari, Martin Tillenius, and Elisabeth Larsson. “Programming models based on data versioning for dependency-aware task-based parallelisation”. In: *15th International Conference on Computational Science and Engineering.* 2012, pp. 275–280. DOI: 10.1109/ICCSE.2012.45 (cit. on p. 12).

[193] Gengbin Zheng, Xiang Ni, and Laxmikant V. Kalé. “A scalable double in-memory checkpoint and restart scheme towards exascale”. In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012).* 2012, pp. 1–6. DOI: 10.1109/DSNW.2012.6264677 (cit. on p. 25).

[194] Gengbin Zheng, Lixia Shi, and L.V. Kale. “FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI”. In: *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No.04EX935).* 2004, pp. 93–103. DOI: 10.1109/CLUSTR.2004.1392606 (cit. on p. 25).

[195] Gengbin Zheng et al. “Hierarchical Load Balancing for Charm++ Applications on Large Supercomputers”. In: *2010 39th International Conference on Parallel Processing Workshops.* 2010, pp. 436–444. DOI: 10.1109/ICPPW.2010.65 (cit. on p. 18).