



On the Design and Development of Programming Models for Exascale Systems

Marcos Maroñas

On the Design and Development of Programming Models for Exascale Systems

by Marcos Maroñas

A thesis submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science.

Copyright © 2016–2020 Marcos Maroñas.

All rights reserved.

First edition: Barcelona, October 2020

Trademarked names are used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended. All the trademarked names cited in this thesis are © of their respective owners.

Advisor:	Dr. Vicenç BELTRAN QUEROL
Co-Advisor:	Dr. Eduard AYGUADÉ PARRA
Department:	Department of Computer Architecture (DAC)
University:	Universitat Politècnica de Catalunya–BarcelonaTech (UPC)
Pre-dissertation Committee:	Dr. Alejandro DURAN GONZALEZ Dr. Xavier MARTORELL BOFILL Dr. Miquel MORETÓ PLANAS
Dissertation Committee:	Dr. Jose GRACIA CALVO Dr. Xavier MARTORELL BOFILL Dr. Alejandro RICO CARRO



Departament d'Arquitectura
de Computadors

UNIVERSITAT POLITÈCNICA DE CATALUNYA



Barcelona
Supercomputing
Center

Centro Nacional de Supercomputación

By *perseverance* the snail reached the ark.
Proverb, published by Charles Spurgeon.

A los míos.

Abstract

High Performance Computing (HPC) systems have been evolving over time to adapt to the scientific community requirements. We are currently approaching to the Exascale era. Exascale systems will incorporate a large number of nodes, each of them containing many computing resources. Besides that, not only the computing resources, but memory hierarchies are becoming more deep and complex. Overall, Exascale systems will present several challenges in terms of performance, programmability and fault tolerance.

Regarding programmability, the more complex a system architecture is, the more complex to properly exploit the system. The programmability is closely related to the performance, because the performance a system can deliver is useless if users are not able to write programs that obtain such performance. This stresses the importance of programming models as a tool to easily write programs that can reach the peak performance of the system. Finally, it is well known that more components lead to more errors. The combination of large executions with a low Mean Time To Failure (MTTF) may jeopardize application progress. Thus, all the efforts done to improve performance become pointless if applications hardly finish. To prevent that, we must apply fault tolerance.

The main goal of this thesis is to enable non-expert users to exploit complex Exascale systems. To that end, we have enhanced state-of-the-art parallel programming models to cope with three key Exascale challenges: programmability, performance and fault tolerance.

The first set of contributions focuses on the efficient management of modern multicore/manycore processors. We propose a new kind of task that combines the key advantages of tasks with the key advantages of worksharing techniques. The use of this new task type alleviates granularity issues, thereby enhancing performance in several scenarios. We also propose the introduction of dependences in the `taskloop` construct so that programmers can easily apply blocking techniques. Finally, we extend `taskloop` construct to support the creation of the new kind of tasks instead of regular tasks.

The second set of contributions focuses on the efficient management of modern memory hierarchies, focused on NUMA domains. By using the information that users provide in the dependences annotations, we build a system that tracks data location. Later, we use this information to take scheduling decisions that maximize data locality.

Our last set of contributions focuses on fault tolerance. We propose a programming model that provides application-level checkpoint/restart in an easy and portable way. Our programming model offers a set of compiler directives to abstract users from system-level nuances. Then, it leverages state-of-the-art libraries to deliver high performance and includes several redundancy schemes.

Resumen

Los supercomputadores han ido evolucionando a lo largo del tiempo para adaptarse a las necesidades de la comunidad científica. Actualmente, nos acercamos a la era *Exascale*. Los sistemas *Exascale* incorporarán un número de nodos enorme. Además, cada uno de esos nodos contendrá una gran cantidad de recursos computacionales. También la jerarquía de memoria se está volviendo más profunda y compleja. En conjunto, los sistemas *Exascale* plantearán varios desafíos en términos de rendimiento, programabilidad y tolerancia a fallos.

Respecto a la programabilidad, cuánto más compleja es la arquitectura de un sistema, más difícil es aprovechar sus recursos de forma adecuada. La programabilidad está íntimamente ligada al rendimiento, ya que por mucho rendimiento que un sistema pueda ofrecer, no sirve de nada si nadie es capaz de conseguir ese rendimiento porque es demasiado difícil de usar. Esto refuerza la importancia de los modelos de programación como herramientas para desarrollar programas que puedan aprovechar al máximo estos sistemas de forma sencilla. Por último, es bien sabido que tener más componentes conlleva más errores. La combinación de ejecuciones muy largas y un tiempo medio hasta el fallo (MTTF) bajo ponen en peligro el progreso de las aplicaciones. Así pues, todos los esfuerzos realizados para mejorar el rendimiento son nulos si las aplicaciones difícilmente terminan. Para evitar esto, debemos desarrollar tolerancia a fallos.

El objetivo principal de esta tesis es permitir que usuarios no expertos puedan aprovechar de forma óptima los complejos sistemas *Exascale*. Para ello, hemos mejorado algunos de los modelos de programación paralela más punteros para que puedan enfrentarse a tres desafíos clave de los sistemas *Exascale*: programabilidad, rendimiento y tolerancia a fallos. El primer conjunto de contribuciones de esta tesis se centra en la gestión eficiente de procesadores *multicore/many-core*. Proponemos un nuevo tipo de tarea que combina los puntos clave de las tareas con los de las técnicas de *worksharing*. Este nuevo tipo de tarea permite aliviar los problemas de granularidad, mejorando el rendimiento en algunos escenarios. También proponemos la introducción de dependencias en la directiva *taskloop*, de forma que los programadores puedan aplicar *blocking* de forma sencilla. Finalmente, extendemos la directiva *taskloop* para que pueda crear nuestro nuevo tipo de tareas, además de las tareas normales.

El segundo conjunto de contribuciones está enfocado a la gestión eficiente de jerarquías de memoria modernas, centrado en entornos NUMA. Usando la información de las dependencias que anota el usuario, hemos construido un sistema que guarda la ubicación de los datos. Después, con esa información, decidimos dónde ejecutar el trabajo para maximizar la localidad de datos.

El último conjunto de contribuciones se centra en tolerancia a fallos. Proponemos un modelo de programación que ofrece *checkpoint/restart* a nivel de aplicación, de forma sencilla y portable. Nuestro modelo ofrece una serie de directivas de compilador que permiten al usuario abstraerse de los detalles del sistema. Además, gestionamos librerías punteras en tolerancia a fallos para conseguir un alto rendimiento, incluyendo varios niveles y tipos de redundancia.

Contents

Abstract ix

Resumen x

List of Figures xvii

List of Tables xxi

List of Listings xxiv

Agradecimientos xxv

1 Introduction 1

- 1.1 Objectives 4
- 1.2 Efficient Management of Modern Multicore/Manycore Processors 5
- 1.3 Efficient Management of Memory Hierarchies (NUMA) 6
- 1.4 Fault Tolerance 7
- 1.5 Thesis Outline 8
- 1.6 Publications 8
- 1.7 Financial and Technical Support 9

2 Background 11

- 2.1 Programming Models 11
- 2.2 Fault Tolerance 22

3 Related Work 25

- 3.1 Efficient Management of Modern Multicore/Manycore Processors 25
- 3.2 Efficient Management of Memory Hierarchies (NUMA) 29
- 3.3 Fault Tolerance 31

4	Efficient Management of Modern Multicore/Manycore Processors	35
4.1	Introduction and Motivation	35
4.2	Background	37
4.3	Worksharing Tasks	43
4.4	Taskloop with Dependences	51
4.5	Combining Taskloop and Worksharing Tasks	55
4.6	Evaluation and Discussion	57
4.7	Conclusions	98
5	Efficient Management of Memory Hierarchies (NUMA)	101
5.1	Introduction and Motivation	101
5.2	Background	103
5.3	Nanos6 NUMA Allocation API	104
5.4	Nanos6 Data Tracking System	108
5.5	Nanos6 NUMA-Aware Scheduling System	111
5.6	Changes Required in the Application	113
5.7	Evaluation and Discussion	115
5.8	Opportunities at Scale	124
5.9	Conclusions	125
6	Fault Tolerance	127
6.1	Introduction and Motivation	127
6.2	Background	129
6.3	OpenCHK Model	131
6.4	Implementation Details	140
6.5	Evaluation and Discussion	145
6.6	Conclusions	153
7	Combining Contributions	155
8	Conclusions	165
8.1	Conclusions	165
9	Future Work	169
9.1	Efficient Management of Modern Multicore/Manycore Processors	169
9.2	Efficient Management of Memory Hierarchies (NUMA)	170
9.3	Fault Tolerance	171

9.4 Other Challenges 172

A Using CR in N-body Simulation 175

A.1 Full Example of N-body Simulation Kernel Using OpenCHK 176

A.2 Full Example of N-body Simulation Kernel Using FTI 178

A.3 Full Example of N-body Simulation Kernel Using VeloC 180

A.4 Full Example of N-body Simulation Kernel Using SCR 182

Bibliography 189

Nomenclature 203

List of Figures

1.1	Trends of processors in the last years. Figure extracted from [59]	2
1.2	CPU-memory gap in the last years. Figure extracted from [29]	3
2.1	Representation of Bulk Synchronous Parallel (BSP) paradigm.	12
2.2	Representation of Data-Flow paradigm.	13
4.1	Typical granularity chart.	39
4.2	Illustration of CPU occupation using different parallelism techniques.	42
4.3	Illustration of similar execution using OmpSs-2 worksharing tasks and OpenMP worksharings.	46
4.4	Comparison of HPCCG benchmark using region dependences with regular tasks and worksharing tasks.	48
4.5	Partition of work and dependences between tasks created using taskloop	51
4.6	Comparison of taskloop with dependences using regions and discrete dependency systems.	52
4.7	Illustration of the taskloop semantics in our implementation	53
4.8	Illustration of the taskloop for semantics	57
4.9	Granularity chart of different implementations of the N-body simulation.	60
4.10	Granularity chart of different implementations of the Stream benchmark.	62
4.11	Chunksize granularity of an N-body simulation and the Stream benchmark.	65
4.12	Pseudo strong scaling charts of the N-body simulation.	68
4.13	Pseudo strong scaling charts of the matmul benchmark.	69
4.14	Pseudo strong scaling charts of the Stream benchmark.	70
4.15	Pseudo strong scaling charts of the HPCCG benchmark.	71

- 4.16 Execution trace of the LULESH T version using a single granularity for all the tasks with a big problem size on 24 cores (1 NUMA socket) 75
- 4.17 Execution trace of the LULESH T version using a single granularity for all the tasks with a small problem size on 24 cores (1 NUMA socket) 76
- 4.18 Execution trace of the LULESH T version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket) 78
- 4.19 Granularity chart of LULESH 79
- 4.20 Execution trace of the LULESH TF version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket) 79
- 4.21 Updated granularity chart of LULESH 80
- 4.22 Execution trace of the updated LULESH TF version using multiple granularities for different task types on 24 cores (1 NUMA socket) 80
- 4.23 Results of the LULESH weak scaling experiment 81
- 4.24 Results of the LULESH strong scaling experiment 82
- 4.25 Execution trace of the LULESH MPI version showing the time performing computations 83
- 4.26 Execution trace of the LULESH MPI version showing the time performing communications 83
- 4.27 Structure of the HPCCG application 85
- 4.28 Execution trace of the HPCCG T version using a single granularity for all the tasks with a big problem size on 24 cores (1 NUMA socket) 85
- 4.29 Execution trace of the HPCCG T version using a single granularity for all the tasks with a small problem size on 24 cores (1 NUMA socket) 86
- 4.30 Execution trace of the LULESH T version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket) 87
- 4.31 Granularity chart of HPCCG 88
- 4.32 Granularity chart of HPCCG 88
- 4.33 Execution trace of the HPCCG TF version using a single granularity for all the tasks on 24 cores (1 NUMA socket) 89

4.34	Results of the HPCCG weak scaling experiment	89
4.35	Results of the HPCCG strong scaling experiment	90
4.36	Evaluation of taskloop using DOTPROD benchmark	93
4.37	Evaluation of taskloop using HEAT benchmark	93
4.38	Evaluation of taskloop using LULESH application	94
4.39	Evaluation of taskloop using MATMUL benchmark	94
4.40	Evaluation of taskloop using MULTISAXPY benchmark	95
4.41	Evaluation of taskloop using NBODY benchmark	95
4.42	Evaluation of taskloop using STREAM benchmark	96
4.43	Evaluation of taskloop using WAXPBY benchmark	96
5.1	Nanos6 directory tracking different memory regions.	109
5.2	Nanos6 directory tracking different memory regions in a more complex scenario.	109
5.3	Detailed analysis of different implementations of the Nanos6 Data Tracking System on different granularities of the Cholesky benchmark	110
5.4	Analysis of different implementations of the Nanos6 Data Tracking System	112
5.5	Changes required in the application to use the Nanos6 NUMA-aware system	114
5.6	Distances between NUMA nodes	116
5.7	Detailed analysis of Nanos6 NUMA-aware system using the Stream benchmark	118
5.8	Analysis of Nanos6 NUMA-aware system in platform A	121
5.9	Analysis of Nanos6 NUMA-aware system in platform B	122
6.1	Long-running applications hardly complete when MTBF is too small.	129
6.2	Comparison between a CP-dedicated threads scheme and a traditional scheme.	135
6.3	Overhead reduction with differential checkpoint for a certain scenario (2400 processes write 1 GB per process to the PFS). n_d corresponds to the ratio of dirty data blocks to protected data blocks.	136
6.4	Three-layer architecture.	141
6.5	Overhead introduced by OpenCHK compared to using native FTI/SCR/Ve-loC	150

7.1 Evaluation of the combination of all the thesis contributions in platform A 164

List of Tables

- 4.1 Average task time, and multiplication factor, for each of the task types of LULESH mini-app using a small problem size 77
- 4.2 Average task time for each of the task types of HPCCG mini-app using a small problem size 86
- 4.3 Programmability metrics to compare the use of the taskloop construct with the manual use of tasks 97

- 5.1 Number of lines of code changed to use the OmpSs-2 NUMA-aware system. 117

- 6.1 Cyclomatic complexity of a 2D heat simulation using OpenCHK (before and after compiler transformation), FTI, SCR, and VeloC. 144
- 6.2 Machine architecture 147
- 6.3 Software and versions used to perform the experiments 147
- 6.4 Number of lines of code required to perform application-level CR using FTI and OpenCHK. 151
- 6.5 Number of lines of code required to perform application-level CR using SCR and OpenCHK. 152
- 6.6 Number of lines of code required to perform application-level CR using VeloC and OpenCHK. 152

List of Listings

2.1	OmpSs-2 annotations format	16
4.1	OMP_F	38
4.2	Region deps	41
4.3	Comparison of a simple code using worksharing constructs and tasks with blocking	42
4.4	Simple code using the taskloop construct	43
4.5	OpenMP code to get similar behavior than a worksharing task	43
4.6	Code of Figure 4.3	44
4.7	Fortran example	44
4.8	Example of use of taskloop for construct	56
4.9	OMP_F(S/D/G)	58
4.10	OMP_T/OSS_T	58
4.11	OMP_TTL	59
4.12	OMP_TF(N)	59
4.13	OSS_TF(N)	59
5.1	Examples of nanos6 NUMA allocation method	105
5.2	Gauss-Seidel using representatives	106
5.3	Compute of NUMA score	112
5.4	Using Nanos6 NUMA API with C++ STL	114
6.1	Brief example of the serialization/deserialization process and the modification in the program flow	130
6.2	Use example of OpenCHK init and shutdown directives in C/C++ and Fortran	133
6.3	Use example of OpenCHK load directive in C/C++ and Fortran	133
6.4	Use example of OpenCHK store directive in C/C++ and Fortran	134
6.5	HDF5 checkpoint file structure using OpenCHK. The protected data consists of 2 scalars and one array	138

- 6.6 Code snippet to produce an HDF5 file with a structure similar to the one shown in Listing 6.5. On top using OpenCHK and below using native HDF5 routines 139
- 6.7 Use example of OpenCHK self-iterative data expressions 143
- 7.1 Task-based Stream benchmark 155
- 7.2 Stream benchmark using worksharing tasks 156
- 7.3 Stream benchmark using taskloop 158
- 7.4 Stream benchmark using taskloop for 159
- 7.5 Stream benchmark using taskloop for with NUMA support 160
- 7.6 Stream benchmark using taskloop for with NUMA support and OpenCHK 161

Agradecimientos

A pesar de tener un único autor, esta tesis tiene pedacitos de mucha gente, a la que quiero agradecer. Sin ellos, esto no habría sido posible.

Antes de nada, quiero mostrar mi profundo agradecimiento a Vicenç Beltran, uno de mis directores. Gracias por darme la oportunidad de unirme a tu equipo. Esta tesis es tan mía como tuya.

Quiero agradecer también a Eduard Ayguadé su desempeño como director de tesis.

Además de mis directores, Sergi Mateo ha sido una persona muy importante en el desarrollo de esta tesis. Gracias por haber dedicado interminables horas a ayudarme. Estoy seguro de que el resultado no sería el mismo sin ello.

Por supuesto, quiero agradecer a todos los compañeros del BSC con los que he compartido este camino: Alberto Miranda, Sandra Macià, Sandra Catalán, Xavier Teruel, Jaume Bosch, Julián Morillo, Juanjo Costa, Raúl Peñacoba, Roger Ferrer, Victor López, Antonio Filgueras, Lluç Álvarez, Ferran Pallarès, Marco D'Amico, Marta García, Guillermo Miranda, Javier Bueno, Jorge Bellón, Hugo Meyer, Carlos Álvarez. Seguro que olvido nombrar a alguien, pero este agradecimiento es para todos los que me habéis acompañado estos años. Me habéis ayudado a crecer tanto en el plano profesional como en el plano personal, gracias.

I would also like to thank the people I met during my internship in EPCC. Thank you Mark Bull, Mario Antonioletti, Catherine Inglis and Rosa Filgueria, for taking care of me and supporting my research.

Evidentemente, no puedo dejar de agradecer a mi familia que me haya apoyado durante estos años. No siempre ha sido un camino fácil, pero ellos siempre han estado a mi lado, para lo bueno y para lo malo. En especial, mi más profundo agradecimiento a mi madre, María del Carmen, y a mi hermano, Fran.

Aprovecho para expresar mi gratitud a mi gran amigo Javier. Un modelo y un gran apoyo. También a Luis, que nunca dudó de mí y siempre me animó cuando más falta hacía. Hago extensivo este agradecimiento a todo el resto de mis amistades.

Por último, pero no por ello menos importante, quiero agradecer a la persona que más cerca de mí ha estado durante la realización de esta tesis. Gracias por hacerlo siempre fácil, Neus.

Introduction

Historically, as observed by Dennard, the transistors can get smaller while keeping the power density constant. Reducing the area of transistors results in a reduction of circuit delays, thereby increasing operating frequency, while the power consumption remains unchanged. Moore predicted in his law that the number of transistors of a chip doubles about every two years, thereby, considering Dennard scaling, increasing the performance of the chip. This fact has been true during many years. However, at some point, the frequency stopped increasing, and the single thread performance started to grow much slowly. Thus, chip manufacturers introduced the multiprocessor architecture to keep increasing the processor performance.

Figure 1.1 shows the trend of the amount of transistors, the single thread performance, the frequency, the power and the number of logical cores of processors in the last years. The figure shows the facts explained previously.

Although parallelism was already popular to exploit distributed systems, it was not used in shared-memory environments. The introduction of multiprocessor architecture forced users to employ parallelism in shared-memory environments to adequately exploit the full power of the processors. Given the high-level of expertise that programmers require to optimally exploit multiprocessor architectures, shared-memory programming models born. The objective of such programming models was abstracting end-users with no expertise in computing (e.g. physicists, biologists, etc.) from the low-level details of machines, while delivering acceptable performance.

Over time, architectures became more and more complex with processors increasingly containing more and more cores, and overall performance considerably increasing. However, the memory performance did not scale as much as the CPU did. This problem is widely known as *Memory Wall* [120]. As a result, there is an increasing gap between processor and memory performance that manufactur-

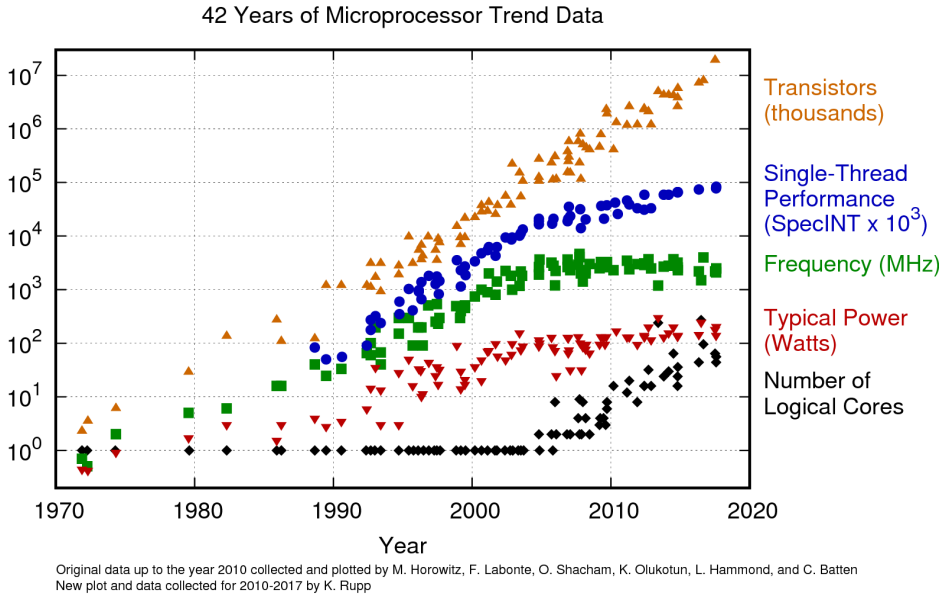


Figure 1.1: Trends of processors in the last years. Figure extracted from [59]

ers try to minimize by using several levels of cache memories, usually known as memory hierarchy. The mentioned gap can be observed in Figure 1.2.

The idea is to place faster but smaller memories (cache) close to the compute resources, to reduce the mentioned gap. Nowadays, it is quite frequent to observe up to three levels of cache, some of them shared between several compute resources, and the main memory. Some architectures goes beyond and add High Bandwidth Memories (HBM), which are faster than main memory, but slower than cache memories.

Nevertheless, the memory hierarchy also comes at a cost. Caches are more expensive, require more area in the chip layout, and consume more power per byte. Moreover, caches require advanced optimizations in applications to optimally exploit them. Otherwise, performance can be negatively affected.

Currently, data movement is a big bottleneck in applications. The easier way to program is moving data to where compute is running. One example of this is *Non-Uniform Memory Access* (NUMA). In NUMA environments, the whole system memory is addressed using a single address space that is shared to all the cores, to ease the programming. That way, cores from different sockets can access remote

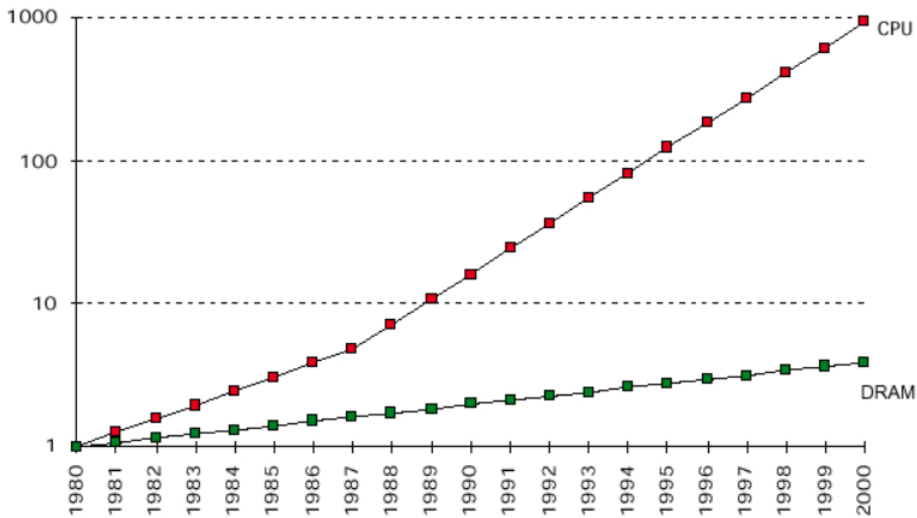


Figure 1.2: CPU-memory gap in the last years. Figure extracted from [29]

memories transparently. However, it is more expensive than accessing the local memory.

Newest architectures are incorporating on-chip NUMA. Apart from having different sockets that can be accessed using NUMA, they are introducing several memory controllers within the chip to reduce memory latencies and increase memory bandwidth. Those memory controllers work also as a NUMA domain, in the sense that all the cores within the chip can use them, but at different costs.

To sum up, a lot of time is wasted moving data from one location to another, downgrading performance. Not only performance is worse, but also power consumption.

Overall, the current complexity of HPC systems is already very high, and it keeps increasing as we approach to the Exascale era. We focus on three concrete factors of this increasing complexity. The first factor is the increase in the number of cores per node. The increase in the intranode parallelism causes that traditional approaches do not deliver as good performance as in previous systems. Additionally, as a natural consequence of the increase in the number of cores per node, newest processors are incorporating more memory controllers to reduce the memory latencies and maximize the memory bandwidth. These memory controllers behave in a NUMA

way: all the cores can access all the memory controllers, but at different costs. The proper management of the different memory channels is crucial to maximize the performance. Finally, there is an increase in the overall number of nodes of HPC systems. Such an increase in the number of nodes leads to an increased error rate, requiring fault tolerance mechanisms more than ever to guarantee the progress of applications.

1.1 Objectives

The main goal of this thesis is to enable non-expert users to exploit complex Exascale systems. To that end, we have enhanced state-of-the-art parallel programming models to cope with three key Exascale challenges: the increase in the number of cores per node, the increase of NUMA domains per processor, and the increase in the overall number of nodes. Our enhancements focus on improving performance, programmability and fault tolerance.

Performance is the most important, most of the efforts focus on improving performance. However, with programmability being a critical issue nowadays due to systems increasingly becoming more and more complex, we must find a compromise between performance and programmability. Usually, the focus is on performance, but it does not matter the performance a system can deliver, if nobody is able to get such performance because it is too hard to handle the system. State-of-the-art programming models still have room for improvement in some issues that will become critical in Exascale systems, namely efficient management of multi-core/manycore processors, and efficient management of memory hierarchies, especially NUMA domains.

Finally, fault tolerance is as important as the two previous challenges in Exascale. Exascale systems will incorporate an enormous amount of components. It is broadly known that the more components, the more likely to experience faults. Thus, if we are not able to adequately manage faults, it does not matter how fast our system is, because applications have a high possibility of not completing. Therefore, converting all our efforts in programmability and performance in meaningless efforts.

In this thesis, we leverage modern directive-based parallel programming models. These are easy to use, portable, and deliver good performance. Our contributions extend the programming models with novel features that are not present in the state-of-the-art.

1.2 Efficient Management of Modern Multicore/Manycore Processors

The first contribution of this thesis focuses on the efficient management of multicore/manycore modern processors. Two popular paradigms for managing parallelism are Bulk Synchronous Parallel (BSP) and Data-Flow.

BSP contains three main stages. In the first one, each of the cores performs concurrent local computation. Then, in the second one, there is data exchange and communication. Finally, the third step is a barrier where all the participants are synchronized.

Data-Flow, in contrast, models executions as directed graphs. The computations are nodes of the graph, and the data are the edges. Computations using the same data are connected in the graph, and they cannot run concurrently. Consequently, there is no need for a global barrier that synchronizes all the cores. So, in other words, Data-Flow provides lightweight synchronization.

State-of-the-art programming models provide two incarnations of these paradigms: fork-join is an implementation of BSP, while task-based is an implementation of Data-Flow.

The fork-join execution model introduces few overhead and delivers very good performance when workloads are regular, structured and well-balanced. However, this model has a very rigid synchronization mechanism and does not fit the requirements of irregular, unstructured and imbalanced parallelism. This rigid synchronization mechanism is not good either for systems with massive amount of cores.

Task-based parallelism naturally copes with irregular, unstructured and imbalanced parallelism, due to its lightweight data-flow synchronization. Nevertheless, task management costs are a noticeable source of overhead, especially for fine granularities.

In terms of programmability, the fork-join approach is easier for the user. Usually, a single construct is enough to parallelize a full loop. In the case of tasking, we need to create several tasks to feed all the available cores. For this purpose, it is frequent to apply blocking techniques to loops, where each of the blocks is a task. This is not a difficult process, but it is more complex compared to the fork-join alternative.

We believe that tasking is the right way to program modern systems, specially considering the drawbacks of the fork-join models and the current trends in sys-

tem's architecture. For this reason, we try to enhance the tasking model to overcome current limitations. We contribute with two proposals in this topic: work-sharing tasks and taskloop with dependences.

Worksharing tasks are a new kind of task that combines the key advantages of tasks with the key advantages of worksharing techniques. They behave as a task in almost all the aspects, except that regular tasks run in a single thread while work-sharing tasks can run in several threads concurrently. Worksharing tasks uncover hidden parallelism leveraging worksharing techniques, resulting in a better usage of the computational resources. The use of this kind of tasks alleviates granularity issues, thereby enhancing performance in several scenarios.

The taskloop construct is a concept already present in the OpenMP standard. It distributes the iteration space of a loop into several tasks. Thus, programmers can easily apply blocking techniques. However, currently, the taskloop construct does not accept data dependences between the tasks created by the taskloop, limiting the applicability of the construct. We extend the taskloop construct to accept dependences based on the induction variable of the loop, so that tasks created by the taskloop can include dependences to other tasks, either created by the same taskloop or not. Finally, we extend the taskloop construct to support the creation of worksharing tasks instead of regular tasks.

With these contributions, we enhance the tasking model both in performance and programmability, becoming a good choice for Exascale system by addressing the challenge related to the increase in the number of cores per node.

1.3 Efficient Management of Memory Hierarchies (NUMA)

The second set of contributions of this thesis focuses on the efficient management of memory hierarchies, specifically NUMA domains. We already introduced the CPU-memory gap, and the idea of memory hierarchy to mitigate the effects of such a gap. We also mentioned the increasing complexity in programming such architectures to adequately exploit such memory hierarchies. There is a lot of work in the literature demonstrating that a good memory hierarchy management is crucial to obtain peak performance in modern systems, becoming data movement an important source of overhead.

We propose a mechanism to prevent data motion, by moving compute to where data is, rather than the other way around. Our proposal focuses on systems with Non-Uniform-Memory-Access (NUMA). Modern processor architectures frequently

include more than one socket in a single shared-memory address space. Notwithstanding, accessing from one socket to another has a penalty depending on the distance between them. This is frequently known as the NUMA effect.

Besides that, there is a new trend that incorporates several memory controllers inside a single socket. All the cores within the socket can access the different memory controllers, but at different costs, similarly to what happens in regular NUMA with different sockets. This is known as on-chip NUMA.

We propose a NUMA-aware system to mitigate the NUMA effect both in multi-socket architectures and on-chip NUMA architectures. Our solution includes an API to manage the memory allocation, a data tracking system which stores data location during the whole execution, and a scheduling system able to schedule tasks where it requires fewer data movements. Our proposal requires very low effort from the user side and is able to mitigate the NUMA effect, thereby improving performance.

With this contribution, we address the second challenge related to the increase of the memory controllers (NUMA domains) in modern systems.

1.4 Fault Tolerance

The last set of contributions of this thesis focuses on resilience. We mentioned that one of the factors that increase complexity in Exascale systems is the increase in the number of nodes. Increasing the number of nodes means increasing the total number of components in the system, increasing also the error rate. An increased error rate requires devoting efforts to develop fault tolerance mechanisms. We focus our work on application-level checkpoint/restart (CR). State-of-the-art application-level CR tools provide advanced I/O capabilities and several redundancy schemes. Nonetheless, there is still room for improvement in terms of programmability and portability, because end-users must manually serialize and deserialize application state using low-level APIs, modify the flow of the application to consider restarts, or rewrite CR code whenever the backend library changes

We designed and implemented OpenCHK, a programming model based on compiler directives and clauses that allows users to specify CR operations in a simple way. Our approach relies on state-of-the-art libraries to provide advanced I/O capabilities and redundancy schemes, and an additional level of abstraction using compiler directives. As a result, both programmability and portability improves.

With this final contribution, we address the last challenge related to the increase

of the overall number of nodes.

1.5 Thesis Outline

The structure of this thesis is as follows: Chapter 2 contains a high-level background to understand the contributions of this thesis; Chapter 3 reviews relevant related work; Chapter 4 details the contributions done related to the efficient management of multicore/manycore processors, including specific motivation and context; Chapter 5 explains the contributions related to the efficient management of memory hierarchies, focused on NUMA domains, also including specific motivation and context; after that, Chapter 6 describe the important of fault tolerance in Exascale systems, and presents the contributions done in this thesis related to this topic; and, finally, Chapter 8 summarizes the work done throughout this thesis, and provides concluding remarks, and Chapter 9 introduces possible future work directions.

1.6 Publications

Publications of this Thesis:

- **M. Maroñas**, S. Mateo, V. Beltran and E. Ayguadé, "A Directive-Based Approach to Perform Persistent Checkpoint/Restart," 2017 International Conference on High Performance Computing & Simulation (HPCS), Genoa, 2017, pp. 442-451, doi: 10.1109/HPCS.2017.72
- **M. Maroñas**, K. Sala, S. Mateo, E. Ayguadé and V. Beltran, "Worksharing Tasks: An Efficient Way to Exploit Irregular and Fine-Grained Loop Parallelism," 2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC), Hyderabad, India, 2019, pp. 383-394, doi: 10.1109/HiPC.2019.00053
- **M. Maroñas**, S. Mateo, K. Keller, L. Bautista-Gomez, E. Ayguadé, V. Beltran, Extending the OpenCHK Model with Advanced Checkpoint Features, Future Generation Computer Systems, Volume 112, 2020, Pages 738-750, ISSN 0167-739X, <https://doi.org/10.1016/j.future.2020.06.003>

- **M. Maroñas**, X. Teruel, M. Bull, E. Ayguadé and V. Beltran, "Evaluating Worksharing Tasks on Distributed Environments," 2020 IEEE Cluster Conference (CLUSTER)
- **M. Maroñas**, E. Ayguadé and V. Beltran, "Mitigating the NUMA Effect on Task-Based Runtime Systems," 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (Submitted pending acceptance)
- D. Alvarez, K. Sala, **M. Maroñas**, A. Roca and V. Beltran, "Advanced Synchronization Techniques for Task-based Runtime Systems," 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), Seoul, South Korea, 2021 (Accepted pending publication)

1.7 Financial and Technical Support

This work is supported by the European Union's Horizon 2020 Research and Innovation programme under grant agreement No. 754304 (DEEP-EST) and 823767 (PRACE), the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 610476 (DEEP-ER), the Ministry of Economy of Spain through the Severo Ochoa Center of Excellence Program (SEV-2015-0493), by the Spanish Ministry of Science and Innovation (contract PID2019-107255GB, TIN2015-65316-P, and TIN2012-34557) and by the Generalitat de Catalunya (2017-SGR-1481, 2014-SGR-1051, and 2014-SGR-1272). Part of this work has been performed under the Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in particular, the author gratefully acknowledges the support of Dr Mark Bull (EPCC) and the computer resources and technical support provided by EPCC.

Background

This chapter contains a background of parallel programming models, and fault tolerance techniques. Additionally, it explains the main pieces of software used in this thesis.

2.1 Programming Models

Programming models appeared to abstract users from low level details that were difficult to manage and highly tied to underlying characteristics of the system. There are different kinds of programming models pursuing different objectives. In this work, we focus on parallel programming models. Although parallel programming models existed before the multiprocessor architecture to exploit distributed systems, they became more prominent when the first multiprocessor architectures arrived. Multiprocessors require several threads to exploit the available compute resources they incorporate. Nevertheless, there are different ways of using thread-based programming. Operating Systems (OS) provide threading implementations, such as POSIX threads in Linux or Win32 threads in Windows OS. Many programming languages also include their own threading implementation, like C++, Python or Java. Each of these implementations provide different APIs, and usually, they are not compatible with others. Consequently, programmers have to write a different version for each of the possibilities, in case they want to test multiple options.

Programming models help users with this kind of issue. A programming model abstracts the user from low-level details, providing an API that works regardless of the underlying OS, language or architecture.

There are many different parallel programming models nowadays. Some of the most important are Intel TBB [65], OpenMP [84], CUDA [69] or MPI [117]. They can be classified in several different ways: shared or distributed memory, sup-

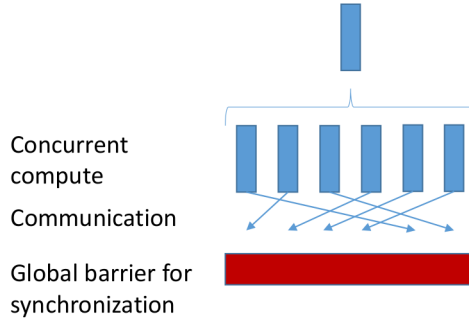


Figure 2.1: Representation of Bulk Synchronous Parallel (BSP) paradigm.

port for heterogeneity, target architecture, task-based, type of synchronization, directive-based or API-based, etc.

In this thesis, we contribute to two programming models: OmpSs-2 [14] and OpenCHK [15]. OmpSs-2 is an already existing programming model that we enhanced with novel features. In the case of OpenCHK, this is a new programming model we designed and developed from scratch. They aim for very different objectives: OmpSs-2 is a programming model devoted to parallelize programs, while OpenCHK is a programming model devoted to add fault tolerance to programs.

In the following sections, we explore the state of the art in programming models for parallelize programs. As well, we explore the different solutions and approaches available in the state of the art of fault tolerance.

2.1.1 Parallel Programming Models

As introduced, parallel programming models are tools that enable users to parallelize programs while abstracting them from low level details and system nuances. There are two main paradigms for parallelism: Bulk Synchronous Parallel (BSP), and Data-Flow.

Figure 2.1 shows a representation of the BSP paradigm. As we can observe there, there are three main stages. In the first stage, all the different threads perform a concurrent computation. After that, they can communicate with other threads. Finally, there is a barrier for synchronization where all the threads wait for the slowest. This is a very simple paradigm but implies very rigid synchronization.

Figure 2.2 shows a representation of the Data-Flow paradigm. In this paradigm,

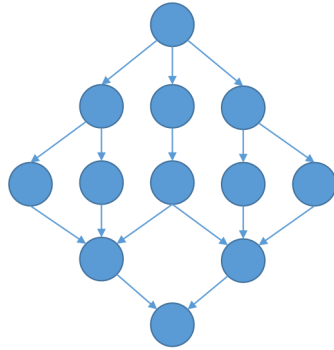


Figure 2.2: Representation of Data-Flow paradigm.

the execution can be modeled using a directed acyclic graph. The nodes of the graph represent the computations while the edges represent the data. When several different computations require the same data, the latest must wait for the previous to finish before starting. This is known as data-flow synchronization, and it is much more flexible than BSP. In this case, each computation can start as soon as its data is ready.

We can find incarnations of both BSP and Data-Flow paradigms in state-of-the-art programming models. The fork-join execution model is an implementation of the BSP paradigm, while the task-based execution model is an implementation of the Data-Flow paradigm. OpenMP, Intel TBB, CUDA, Cilk [67], Habanero [31], etc., support both execution models. Others, like OmpSs-2 are purely task-based.

2.1.2 Parallel Programming Models for Shared-Memory Environments

Despite the high number of existing parallel programming models for shared-memory environments, such as Intel TBB, OmpSs-2, and Cilk, OpenMP is the *de facto* standard in HPC. It is a directive-based programming model, that means users define actions through compiler annotations, also known as *pragmas*. Directive-based programming models are easy to use and allow incremental parallelization. Additionally, the compiler annotations can be easily ignored by compilers, enabling programmers to have multiple versions (serial and parallel) in a single code.

As introduced previously, OpenMP supports both fork-join and task-based parallelism. The most popular way of parallelizing programs using OpenMP is using worksharing constructs. They distribute the available work into several threads,

that run concurrently. At the end, all the threads wait until the last of them finished in what is known as a *barrier*.

In version 3.0, OpenMP introduced the task concept. A task is basically a portion of serial code that can be run asynchronously. Several tasks can run concurrently, achieving parallelism. Tasks can contain data annotations to define dependences. Given two sibling tasks, A and B, being B created after A, if task A annotates variable *X* as an output, and task B annotates the same variable either as input, output or both; task B depends on task A, and so, task B cannot run until task A has finished.

Worksharing constructs are frequently used over loops, and more generally, to exploit structured parallelism. They introduce very few overhead. However, the barrier at the end makes worksharings susceptible to load imbalance. Thus, they are not a good choice for unstructured parallelism.

Tasks provide more flexibility. The synchronization when using tasks (+ data dependences) is data-driven. Thus, threads can get new work as soon as there is more work available (unlike worksharings). Nevertheless, tasks introduce more overhead than worksharing constructs.

2.1.3 OmpSs-2

OmpSs-2 is a programming model designed and developed by the Programming Models group of the Barcelona Supercomputing Center (BSC). It is a directive-based programming model. Currently, OmpSs-2 is a purely task-based programming model, which means that the only way of partitioning work is using tasks.

OmpSs-2 is the second generation of the OmpSs programming model. Both generations are research-oriented. In fact, they acted as a forerunner of OpenMP in the sense that many features that are currently part of the OpenMP standard has been firstly proposed by OmpSs/OmpSs-2.

The OmpSs [19] parallel programming model is an integration of features from the different programming models of the StarSs family into a single programming model. Also, OmpSs born with the objective of extending OpenMP with new directives to support new features such as asynchronous parallelism and heterogeneity (devices like GPUs). In fact, the name *OmpSs* is a combination of the names of the above-mentioned programming models pointing out the relationship between them.

OmpSs inherits from OpenMP the philosophy to develop parallel programs: start from a sequential code and add compiler directives to uncover parallelism.

Therefore, OmpSs, like OpenMP, uses compiler directives. These compiler directives are just annotations in the code. The main advantage of these annotations is that they do not affect at all to the program semantics, while they allow the compiler to produce a parallel version of it. Using this philosophy, application developers can parallelize their applications incrementally: starting from a sequential version, they can add directives to different parts of the code obtaining parallelism on them.

On the other hand, OmpSs inherits from StarSs the possibility of targeting heterogeneous architectures (included also in OpenMP 4.0), and the use of tasks as the main mechanism to express parallelism. Nevertheless, OmpSs still support fork-join parallelism in the same way than OpenMP, using worksharing constructs.

OmpSs-2 is the natural evolution of OmpSs. At BSC we believe that task-based parallelism is the right choice to address several challenges of modern systems, so OmpSs-2 completely focuses on task-based parallelism. Thus, no support for fork-join parallelism is provided at all, so worksharing constructs are not present. In OmpSs-2, tasks are the only way of parallelizing applications.

Basics of OmpSs-2

As introduced, OmpSs-2 is a parallel programming model for shared-memory environments. It enables parallelization by means of a set of compiler directives and library routines. OmpSs-2 can be used with C/C++ and Fortran. As OmpSs did in the past, OmpSs-2 seeks to extend the OpenMP standard with new features that provide users a simple and productive way of exploiting modern HPC systems.

We already mentioned that OmpSs-2 is a purely task-based model, where the tasks are the elementary and unique unit of work. The synchronization of tasks can be done using data dependences or explicit synchronization points (i.e. `taskwait`). OmpSs-2 highlights are listed following:

- **Extended lifetime of task data environment.** Tasks *complete* when they finish their execution, and *deeply complete* when all their children tasks deeply completed or there are no childs, and the task completed. As a result, the data environment of a task (i.e. the variables captured at task creation) is only destroyed when a task is deeply completed.
- **Connection of nested dependence domains.** The dependences of a task are propagated to its children tasks. Thus, when a task finish, we replace their outgoing dependences by those generated by its children.

- **Dependencies' early release.** Dependencies that are not included in any un-completed children task are released as soon as the task completes. This default behavior can be modified using a clause that postpones the release until the task is deeply completed.
- **Weak dependencies.** OmpSs-2 distinguishes between strong and weak dependencies. The strong dependencies are the regular dependencies that specify that a task actually requires some data, while the weak dependencies specify that their children will require the data. As a result, a task that only contains weak dependencies is immediately ready.

Annotating OmpSs-2 Programs

A subtle but important difference of OmpSs-2 compared to its predecessor, and also OpenMP, is that the sentinel used in the pragma is not `omp` anymore. OmpSs-2 uses the `oss` sentinel. Listing 2.1 shows the format of OmpSs-2 annotations.

Listing 2.1: OmpSs-2 annotations format

```
#pragma oss directive [clause[ [,] clause] ...]
```

OmpSs-2 Execution Model

Unlike in OpenMP, in OmpSs-2 users do not require the `parallel` directive in OmpSs-2, because the runtime creates the threads at startup. These threads are later used to run the available tasks. In OmpSs-2, the user has no control of the pool of threads, which is managed by the runtime.

The OmpSs-2 runtime system creates an initial pool of worker threads at startup. The main function of the program becomes an implicit task called `main` task, and it is enqueued in the ready queue as any other regular task. Once a thread gets the task, it starts running the main function, while the rest of the threads keep waiting until other tasks are ready for execution.

Nesting in OmpSs-2

Nesting is defined as the ability to create multiple levels of parallelism, one within another. In OmpSs-2, tasks can create more tasks, enabling users to actually get multiple levels of parallelism that can help to improve performance. Nesting is

especially important for recursive algorithms, because it enables the parallelization of such algorithms.

OmpSs-2 Dependency Model

As usual in task-based programming models, in OmpSs-2 the tasks can contain data dependences between them. The runtime library uses the data annotations specified by the user, and the task creation order to produce execution order constraints between tasks. These are known as data dependences.

The data annotations enable users to specify which data is accessed within a task, and which kind of access it is (read, write or read-write). In the event of a task creation, the data annotations of the task are matched against the data annotations of other tasks. In case there is a dependency, either RaW (Read after Write), WaR (Write after Read) or WaW (Write after Write), the recently created task becomes a successor of the corresponding existing tasks, and its execution will be deferred until its successors finish. Otherwise, when there are no dependences, the task is immediately ready for execution.

Apart from the usual kind of data annotations (read, write, read-write), OmpSs-2 provides the commutative and concurrent data annotations. When several tasks have been annotated with the commutative clause over the same data, all of them are eligible to run, but just a single one can run at a time. Note that the commutative annotation forces tasks to wait for predecessors that use the same data with any other kind of data annotation. The concurrent annotation allows multiple tasks that have been annotated with a concurrent over the same data to run in parallel, while forcing them to wait for predecessors with any other kind of data annotation.

The OmpSs-2 programming model provides two different dependency systems. The first one enable users to define dependencies over regions of data, rather than a single element. The second dependency system is equivalent to the OpenMP one, where the dependences are defined only on the start address of the data specified. We usually refer to them as region dependences for the first one, and discrete dependences for the second one.

The region dependency system is very useful because it looks for partial overlapping of memory regions. This way, the user can annotate programs in a more natural way, specifying the whole region accessed by a task, and the runtime will ensure that tasks are executed in the proper order, respecting the dependences. However, in the case of discrete dependences, tasks accessing partially overlapped data may run concurrently, producing incorrect results. A common way of preventing this

problem is the use of sentinels. On the other hand, the discrete dependency system is faster than the region dependency system, that introduces non-negligible overhead, especially in fine granularities.

Two important features of the OmpSs-2 dependency model, already introduced, are the early release of dependences and the weak dependences.

Early Release of Dependences in OmpSs-2 This feature is closely related to task nesting, enabling users to speedup some specific applications. In OpenMP and OmpSs, the dependency domain of parent and children tasks can be connected by means of taskwaits. Placing a taskwait at the end of the parent task, we ensure that all the children tasks finished and so, the parent can release its dependences. Nevertheless, this may be deferring the release of dependences of some children that may have finished before. As well, it prevents the parent from finishing its own execution.

OmpSs-2 enable users to connect the dependency domain of parent and children tasks with no taskwaits. The runtime realizes that when a parent task finishes its own execution, it does not require the enforcement of its own dependences, and also that no further children tasks will be created. Consequently, all the dependences defined by the parent that are not defined by any other alive child task, can already be released, preserving only the dependences required by the alive children tasks

Weak Dependences This is an extension to the OmpSs and OpenMP dependency system to improve the integration of task nesting and dependences. OmpSs-2 provide two different kind of dependences: strong and weak. The strong dependences are the regular dependences already present in OpenMP, while the weak dependences are a new kind of dependency that enable users to specify data that is required by the children tasks rather than by the task with the annotation. Usually, parent tasks include dependences that are not required by them but by their children tasks, in order to connect the outer dependency domain with the inner dependency domain. In these cases, it is not required to defer the parent task execution, because it is not actually accessing data. As a result, tasks with only weak dependences become immediately ready.

Using weak dependences, it is simpler to setup mechanisms to create tasks concurrently, accelerating the task creation.

Reference Implementation of OmpSs-2

The reference implementation of OmpSs-2 uses the Mercurium compiler [11], and the Nanos6 runtime library [12].

Mercurium Compiler Mercurium is a source-to-source compiler infrastructure developed at BSC. It supports C/C++ and Fortran. Nowadays its main use is in combination with Nanos++ in order to implement OpenMP and OmpSs programming models, and Nanos6 in order to implement OmpSs-2 programming model. Withal, given its extensible nature, it also has been used to implement other programming models different than the mentioned ones, as well as performing different compiler transformations. Some of them are Cell SuperScalar (CellSs), Software Transactional Memory or Distributed Shared Memory.

Mercurium is designed using a plugin architecture. Each plugin of Mercurium represents a compiler phase. All the Mercurium plugins are developed in C++, and they are dynamically loaded according to the configuration chosen by the user.

Mercurium is in charge of processing OmpSs-2 directives and perform the appropriate transformations to convert directives into runtime library calls. Furthermore, Mercurium is able to restructure code depending on its target device (CPU or GPU, for instance). For that purpose, Mercurium includes a specific handler for each device. If required, it can even generate device-specific code in different files for each of the target devices. Mercurium also can invoke different device-specific compilers such as `nvcc` to compile CUDA code.

Nanos6 Nanos6 is a runtime library designed to be used in parallel environments. Nowadays, its main use is OmpSs-2. This library provides support for asynchronous task-based parallelism relying on data-dependences. Although Nanos6 main goal is to be used in research about parallel programming environments, it is a highly-optimized runtime that has been already used in several EU projects.

On [1], Alvarez, et.al, detail several of the optimizations applied in the runtime system, including a wait-free dependency system, a scheduling system using advanced locks with delegation techniques, optimizations in the memory management, and a lightweight instrumentation mechanism.

2.1.4 Parallel Programming Models for Distributed Environments

As with OpenMP in shared-memory environments, MPI has been a *de facto* standard in distributed environments. It is a library-based programming model, where users define actions by calling methods from the library. It enables users to pass messages between different processes using the network. As a result, programs can run in many different nodes concurrently.

During many years, programmers used to parallelize applications with a pure MPI implementation. The most frequent pattern is to decompose the data set of the application into several chunks, each of them run by a single MPI process. At some point, the processes communicate between them to exchange required information to continue the execution. Another option is replicating the whole data set in every single MPI process, but this is not a very popular choice because it requires vast amounts of memory.

Pure MPI parallelizations have been delivering very good performance during many years, but nowadays, due to the large core count that modern systems contain, it is not always the best option. Thus, we are transitioning to a hybrid approach where MPI is combined with other programming models. In this hybrid approach, MPI manages the inter-node parallelism, while another programming model manages the intra-node parallelism. Examples of such an approach are the combination of MPI+OpenMP or MPI+OmpSs-2. In the case of MPI+OpenMP, we can use both fork-join parallelization or task-based parallelization for the intra-node parallelism, while OmpSs-2 can only provide task-based parallelizations due to its purely task-based nature.

Concretely, MPI+tasking is becoming a very popular approach recently. It can combine the low latency and high throughput of the MPI library with the natural ability of tasks to deal with load imbalance, and overlap computation and communication phases, overall providing opportunities to speedup applications.

There are other programming models that provide parallelism in distributed environments, but they are less popular than MPI. Examples are Charm++, Legion or Chapel. In these programming models, the distributed memory management is implicit and managed by the runtime.

2.1.5 MPI+OmpSs-2 Interoperability

The programming models group at BSC has been working on MPI+OmpSs-2 interoperability the last years. The Task-Aware MPI (TAMPI) [17] library is fruit of

this work.

TAMPI is a library designed to improve the interoperability between task-based programming models and MPI. Placing MPI calls inside tasks may cause deadlocks due to the out-of-order execution of tasks. This library implements a cooperation mechanism between the tasking runtime and MPI library that ensures a deadlock-free and efficient execution.

TAMPI provides two different modes: blocking and non-blocking. In this thesis, we only use the non-blocking mode. This mode focuses on the use of non-blocking asynchronous MPI operations inside tasks. When using TAMPI non-blocking mode inside a task, the task binds its completion not only to the execution of its body, but also to the completion of all the MPI requests indicated. The task completion implies the release of its dependences, as well as freeing its data structures.

TAMPI offers two methods to describe which requests a task must wait for, both of them asynchronous and non-blocking: `TAMPI_Iwait` for a single request, and `TAMPI_Iwaitall` for multiple requests. So, a task that binds its completion to one or more MPI requests using the mentioned methods will not complete (and so release its dependences) until its body is run *and* the MPI requests have completed. If a task completes its body before one or more MPI requests finished, it will not release its dependences, and so, the successors cannot become ready for execution. However, the core that was running the unfinished task can proceed to execute other ready tasks, thus preventing the core to be idle waiting for the communication.

When the communication actually finishes, TAMPI notifies the task-based runtime system that the task is actually completed. After that, the runtime can release the dependences, and the successors become ready for execution. With this mechanism, progress is ensured. The effects are not only deadlock-free executions, but there are also possible improvements in performance, because the CPU utilization tends to be better.

In normal MPI programs, the programmer can try to overlap communication and computation by placing non-blocking communication calls as early as possible and the corresponding waits as late as possible, and hope that the MPI library progresses the communication while the intervening computation is being executed. By using tasks and TAMPI, the MPI calls do not require such careful placing, available computation can be discovered dynamically via the task dependency system, and reliance on the MPI progress engine is reduced.

2.2 Fault Tolerance

Large HPC systems can be affected by several different errors and faults, ranging from network problems to node shutdown, including power breaks. In the event of an error or fault, applications are aborted and its progress is lost. Given the increasing number of components in modern HPC systems, that leads to an increased error rate [38], it is important to introduce mechanisms that preserve the progress and correctness of applications in the presence of failures. Fault tolerance mechanisms provide such an ability.

HPC community has been doing a lot of research in fault tolerance techniques for modern HPC systems. Most popular strategies rely on rollback-recovery, which periodically save the process state while the application runs with no trouble. The saved state is usually known as a checkpoint. Then, in the event of a failure, the application can restart from one of the checkpoints rather than from the beginning. It is important to keep consistency between all the processes when recovering from a checkpoint.

Other implementations of rollback-recovery are based on message logging. They try to reduce the overhead in non-faulty executions, as well as preventing the restart on processes where no faults occurred.

Next, we provide more details about checkpoint-based and log-based approaches.

2.2.1 Checkpoint-Based Fault Tolerance

Checkpoint-based fault tolerance is also known as checkpoint/restart (CR). Fault tolerance techniques that rely on checkpoints must save global consistent states. In the case of MPI programs, we require to take checkpoints in every single process participating in the parallel execution. It is also important that all the processes have matching communication messages. In other words, all the messages received by receivers must have its corresponding emission from the senders. If there is a single process that is missing one communication, for instance, it sends a message that the other process already received before a restart, the execution result may be incorrect.

Checkpoint-based approaches may be classified on coordinated and non-coordinated approaches.

Coordinated Checkpoint-Based Fault Tolerance

In coordinated approaches, every single process take the checkpoint at the same moment. Usually, there is a global barrier after the checkpoint to ensure that every single process took the checkpoint at the same point of the execution. With this technique, we ensure that there are no mismatching communications, and we achieve a global consistent state very easily. If a failure occurs, all the processes stop and restart from the last correct checkpoint. On the other hand, coordinated checkpoints introduce non-negligible overhead during the synchronization of all the processes, especially with large processes count. Additionally, as all the processes store the checkpoints simultaneously, the storage and network can become a bottleneck.

In this thesis, we focus on coordinated checkpoint-based fault tolerance. There are several kinds of CR. They can be classified using different criteria: application-level or system-level, according to where it is implemented; persistent or diskless, depending on the method of storing data; and coordinated or non-coordinated, according to whether process coordination is required to create the checkpoints.

We target application-level CR because of its efficiency both in performance and disk space compared to other approaches. There are many different libraries providing application-level CR. Most of them provide single-level checkpointing, and few of them multi-level checkpointing. The difference is that multi-level checkpointing solutions are able to write checkpoints in different storage layers rather than using only the Parallel File System (PFS). Writing checkpoints only using the PFS easily introduce a lot of overhead. Using multi-level checkpointing, checkpoints can be written in RAM disks, local-node storage or SSDs, reducing the overhead introduced.

Non-Coordinated Checkpoint-Based Fault Tolerance

In non-coordinated approaches, each process is free to choose the best moment to take a checkpoint. Thus, there is no synchronization between different processes to take checkpoints. This is a great advantage in terms of overhead, because we do not need to spend time synchronizing the processes to take every checkpoint. Besides that, the possibility that all the processes take a checkpoint simultaneously decreases, so the pressure on the storage and network is lower. Also, with non-coordinated checkpoint, in the ideal case, only the faulty process has to recover, while the rest can keep running as if nothing occurred.

However, in this kind of approaches, it is much more difficult to recover a global consistent state. When recovering from a checkpoint in the faulty process, we may encounter lost or orphan messages. Lost messages are those sent by other processes to the faulty process, that were never received. Orphan messages are those sent by the faulty process more than once, before the fault and after the recovery. The existence of orphan and lost messages may result in incorrect executions. To prevent orphan and lost messages, we may end up applying rollback in non-faulty processes. In the worst-case this may cause a domino effect that brings us to the initial state. In order to avoid such domino effect, non-coordinated checkpoints are usually combined with message logging techniques [104].

2.2.2 Message Logging

Message logging rely on the assumption that replaying messages in the correct order leads to a global consistent state. In this approach, applications are a sequence of deterministic (compute) and non-deterministic events (communication). As said, it is frequently used in combination with non-coordinated checkpoint to prevent the domino effect. The most important benefit of message logging is the ability to rollback only the faulty process. Nevertheless, it introduces overhead on every single message and is usually more difficult to develop and use than simple coordinated checkpoint approaches.

Related Work

We present in this chapter the related work of this thesis. This chapter is subdivided to group specific related work of the three main contribution sets.

3.1 Efficient Management of Modern Multicore/Manycore Processors

Modern multicore/manycore processors are challenging traditional ways of exploiting parallelism. The two main paradigms are fork-join and task-based parallelism and both present some challenges to address related to the efficient management of modern multicore/manycore processors. In the case of fork-join parallelism, the synchronization is too rigid resulting in possible performance pitfalls, while in the case of task-based parallelism the granularity must be well-tuned to prevent either task management overheads or lack of parallelism.

The idea of hierarchical partitioning can help to overcome the challenges presented by modern multicore/manycore processors. This idea is broadly used in distributed environments to reduce overheads. Most applications firstly partition data using inter-node parallelism, spreading such data among different nodes. Then, the work is partitioned again using intra-node parallelism. There are several works in the literature proposing several techniques based on this idea, such as [44][33][90].

Additionally, with regard to programmability, fork-join parallelism provides a simple way of parallelizing loops. In contrast, task-based parallelism requires more effort from the user side who has to apply blocking techniques to prevent lack of parallelism.

As OpenMP is the standard for shared memory parallelism, we performed a thorough review of the OpenMP environment to search related work. This can be seen in Section 3.1.1. In addition, wider related work can be found in Section 3.1.2.

3.1.1 OpenMP related work

We already mentioned in Chapter 2 that OpenMP supports both loop-based parallelism and task-based parallelism. The most common way of using loop-based parallelism in OpenMP is by means of the worksharing constructs. In terms of programmability, worksharing constructs enable users to parallelize loops using a single construct. Thus, they are very simple to use. In terms of performance, worksharing constructs deliver good performance in the general case. Nevertheless, they contain an implicit barrier at the end of the worksharing region, introducing very rigid synchronization.

The task-based approach is a bit more complex in terms of programmability. It usually requires blocking techniques to uncover parallelism, which require some more code than a single construct. Regarding performance, tasks have a natural ability to deal with load imbalance, but they have associated costs that may introduce some overhead depending on the granularity. OpenMP provides also the `taskloop` construct, that distributes the iteration space of a loop into several tasks. There is the possibility of specifying a *grainsize* guaranteeing that each of the tasks created executes no less than *grainsize* iterations. Thus, the `taskloop` construct simplifies the use of task-based parallelism, enabling users to parallelize loops with a single construct. Nevertheless, OpenMP does not support dependences in the `taskloop` construct. As a result, users must rely on fork-join-like synchronization with explicit synchronization points. Consequently, dropping the data-flow execution model of task-based parallelism, and its benefits. By enabling the use of data dependences in the `taskloop` construct, we offer users the possibility of parallelizing loops in a single construct while keeping the benefits of the data-flow execution model. Additionally, the use of the `taskloop` construct, may reduce the tasking overhead because allocations could be optimized to be done as a whole, instead of one by one. However, the number of tasks that will be created and scheduled is still proportional to the problem size.

Related to the hierarchical partition of the work, OpenMP offers some possibilities too. In OpenMP we can use nesting techniques to reach hierarchical partition of the work, for instance, combining worksharing and tasking constructs.

One of the possibilities is using tasks to perform a first partition of the work, and then, each task contains a nested parallel region with a worksharing construct. The reason for using tasks in the first level of partitioning is the flexibility given by the data dependences. This implementation may increase the resource utilisation in some scenarios, boosting performance. However, we end up introducing a barrier

inside each task, at the end of the nested parallel region.

Barriers have been broadly treated in literature [78]. Currently, they are usually highly optimized so that they introduce only a few overhead in some situations, though, if the work is not perfectly balanced, the intrinsic rigidity of the fork-join model may lead to undesired waiting times. OpenMP introduced the `nowait` clause to palliate this issue. This clause omits the barrier at the end of a worksharing region. Nonetheless, this mechanism is not useful to avoid the barrier at the end of a parallel region inside a task. This is because the barrier is necessary to postpone the release of the task dependences until the work is completed.

OpenMP also provides different scheduling policies for the worksharing constructs, alleviating load imbalance problems. Still, they are not enough for many cases, and the rigidity of the fork-join model may lead to an underutilization of the resources.

It is possible to implement a different solution using task nesting. This is basically creating tasks inside tasks. So, a first partitioning is done using coarse-grained tasks with data dependences, which are then partitioned into fine-grained tasks without data dependences. The second level of partitioning reduces the overhead compared to a single level of partitioning where all the tasks have data dependences because the nested tasks do not pay the dependence management costs. In addition, using tasks improves load balance. However, tasking introduces other overheads associated with tasks management, such as dynamic allocations and task scheduling.

Another possible solution using OpenMP tasking model includes the `taskloop` construct. We could have a first partitioning using coarse-grained tasks with data dependences with `taskloop` constructs within. The overhead of the task creation could be reduced in the inner level thanks to the use of the `taskloop` construct, as we explained earlier. Still, this is very similar to the previous version using pure task nesting, so, although alleviated, we keep having the same problems.

To address the challenges that modern multicore/manycore processors present, we propose two contributions. Section 4.3 details worksharing tasks, our proposal based on the idea of hierarchical partitioning. Worksharing tasks leverage work-sharing techniques to mitigate granularity issues, while keeping the data-flow execution model of tasks. The second is focused on programmability. We introduce support for data dependences in the `taskloop` construct, to enable task-based parallelizations of a loop with data-driven synchronization using a single construct. This second contribution is explained in Section 4.4.

3.1.2 Non-OpenMP related work

There exist other works, such as StarPU [112] *Parallel tasks* [113]. A parallel task is a task that can be run in parallel by a set of CPUs, which might sound similar to our proposal of worksharing tasks. Nonetheless, these tasks are like the combination of an OpenMP task with a worksharing construct inside. Thus, it contains an implicit barrier at the end. Moreover, in many cases, only a single parallel task can run at a time. The reason for this is that many environments and libraries they leverage internally do not support concurrent calls to create several parallel regions without nesting.

Intel Cilk presents the `cilk_for` [55], which is used to parallelize loops. The body of the loop is converted into a function that is called recursively using a divide and conquer strategy for achieving better performance. However, there is a `cilk_sync` at the end of each iteration. Therefore, synchronization is quite rigid, similarly to OpenMP worksharings.

3.1.3 Hybrid MPI+X

In this contribution set, we also explore the ability of worksharing tasks to speedup applications at scale. For that purpose, we used the MPI+X approach. “MPI+X” has become the dominant paradigm for hybrid parallel programming. This approach is based on the Message Passing Interface (MPI) plus a second approach/-model leveraging the system- or the node- level capabilities of the HPC system. To improve interoperability in between these two components (the *MPI* and the *X*), the HPC community is actively exploring new opportunities and extensions. Some of them have been already incorporated into the MPI standard: levels of threading support (introduced in MPI 2.0 [74]), matched-probe operations or inter-process shared memory (both included in MPI 3.0 [75]). Some of these ideas have not yet been included into the standard, but they have proven their usability: Infini-Band GPU-to-GPU communication [118], or the Endpoints [37] extensions. And finally some of them directly impact on the *X* component: the collective offloading at clusters [94], implemented in `OmpSs`, for instance.

Programming the *X* component may follow, intentionally dismissing the data-parallelism, two different approaches: the fork-join model or the tasking model. The fork-join model efficiently manages the overhead of the computational phase parallelization, but it also imposes very strong restrictions with respect to the thread synchronization (in the join phase). The tasking model (with dependences) allows

the data-flow execution by means of taskifying the computation and communication phases and let the dependences to guide the execution. However, it adds a non-negligible overhead that directly impacts in the performance.

- In Section 4.6.1, we explore the behavior of worksharing tasks, which leverage the benefits of the fork-join model with respect to the low overhead, and remove the synchronization constraints imposed by OpenMP parallel regions. However, the study does not analyze any interoperability option to improve the communication behaviour between MPI processes.
- In [95, 96], the authors turn inside out the interoperability options between the OmpSs-2 programming model and the MPI library. They extend the functionalities of *blocking* and *non-blocking* services allowing the task-based runtime system to context switch when a MPI communication service is not ready yet. This approach minimizes the number of cycles a CPU begins to idle (when there are still other tasks to execute). However, the study is completely based on the pure tasking model, imposing overheads that make it impossible to work with very fine granularities.

In Section 4.6.2 we carried out a study combining both approaches, leveraging the strengths, and minimizing the weaknesses, of each one.

3.2 Efficient Management of Memory Hierarchies (NUMA)

NUMA-awareness is a topic that have been studied for long time. Some of the works done in this topic rely on the first touch policy such as the one of Al-Omairy et. al. [82], based on the NUMA support of the first generation OmpSs programming model [13]. In this work, the distribution of data relies on the distribution of initialization tasks. The authors assume a first touch policy, and based on this assumption, the runtime annotates the location of data. In other words, they assume that data is in the NUMA node that ran the first task (initialization task) that accesses a memory region. Users have two options for distributing initialization tasks. The first option is relying on the runtime, that is able to identify initialization tasks and schedules them in a round robin fashion across the different NUMA nodes. The second option is using an API call that provides a hint to the runtime specifying the NUMA node where the task should run. This must be done per task, as it only applies to the immediately next created task. Then, using the information

gathered when running initialization tasks, the runtime performs a NUMA-aware scheduling, with distance-aware work stealing.

However, as they assume first touch policy, if the user already does a manual data distribution, their assumption is not correct and their approach does not work. Additionally, they rely on a proper distribution of data done by the distribution of initialization tasks providing poor flexibility. As well, their API to provide hints is tedious, as hints must be provided per task. Besides that, their implementations keeps querying to a map, which may become a bottleneck, especially in fine granularities. Finally, the runtime library of OmpSs only steals work from adjacent nodes, which could be a problem if there is load imbalance.

Sánchez et. al. [97] applied graph partitioning techniques to perform NUMA-aware scheduling. They apply dynamic graph partitioning to the Task Dependency Graph (TDG), where tasks represent code pieces, and edges represent data dependences. As a result, tasks accessing the same data run in the same cores, reducing data motion. Nevertheless, this work also relies on the first touch policy, presenting the same drawbacks than the previous works: it provides poor flexibility to distribute data, and it does not work if the user does manual data distribution.

Others, like Tanaka and Tatebe [109] proposed static graph partitioning to schedule workflows and reduce data movement. Notwithstanding, workflows frequently use coarser granularities than shared-memory codes. As a result, a valid approach for workflows, becomes too expensive in our environment.

OpenMP also put the focus on memory management in its latest release (5.0). They included OpenMP allocators [83] to the standard. Although this is more focused on deep memory hierarchies, they also provide some features that could be used to perform data distribution in NUMA environments. However, the user can only provide hints to the runtime, which is the one actually deciding where to allocate data. In [108] the authors mention the idea of the OpenMP board to include support for distribution of allocations across NUMA domains. Additionally, OpenMP allocators are not combined with a locality-aware scheduling.

Other works based on OpenMP focused on locality-aware scheduling. The authors of [63] extend OpenMP to exploit task-to-data affinity. The authors introduce a new clause where the user must specify the data considered for affinity in each of the tasks, clearly separating data dependences and affinity. In our case, we use the information specified in the data dependences rather than adding a new specific clause. When a task becomes ready, the runtime checks the physical location of the data using the `move_pages` method from `libnuma`, and store the information in a map. In successive tasks accessing the same data, they query the map rather

than using `move_pages` again. The use of `move_pages` suggests that they do not offer an API to distribute the data, but only perform locality-aware scheduling relying on the data distribution performed by the OS. Thus, the only possible data distribution policies are first touch or interleaved. Also, they keep querying a map per memory access, which may become a bottleneck, especially in fine granularities. Finally, their NUMA-aware task stealing considers only distance, prioritizing the stealing from queues of threads residing in the same NUMA node. In our case, we consider both distance and load balance, because stealing a task considering only distance may lead to a domino effect in stealing, where a thread is stealing the last task of another, causing the next to steal also, increasing the probability of harming locality.

There are other approaches to minimize the NUMA effects provided by the Linux kernel such as *libnuma* [3] or *numactl* [4]. *libnuma* provides an extensive API to manage memory allocation and data movement, but the user is still in charge of doing a proper work scheduling based on the data location. *numactl* is a command that runs processes with a specific memory placement policy or NUMA scheduling. It provides an easier way of managing data distribution than *libnuma*, but still lacks support for schedule work based on data location.

To sum up, in literature there are different approaches trying to mitigate NUMA effects. However, none of them combine a flexible data distribution mechanism that enables multiple allocation policies, a highly optimized data tracking mechanism able to support fine granularities, and a NUMA-aware scheduler that considers data location to perform task scheduling with task stealing considering both distances and load balance. Our approach, explained in Chapter 5 combines all this features, mitigating NUMA effects in several scenarios.

3.3 Fault Tolerance

In this section, We describe different CR approaches focusing on persistent solutions. We discuss different kinds of checkpointing and examine some checkpointing tools, such as BLCR, FTI, SCR, and VeloC.

The CR technique consists in regularly storing application data and restoring it in case of error, thereby benefiting from previous work rather than restarting from scratch. For addressing soft errors, data can be saved in memory (non-persistent) [105], whereas for hard faults, data must be stored persistently in storage.

CR approaches can be organized using several criteria: application-level or system-

level, according to where it is implemented; persistent or diskless, depending on the method of storing data; and coordinated or non-coordinated, according to whether process coordination is required to create the checkpoints.

In coordinated checkpointing, the processes must coordinate to take their checkpoints building a global state. In other words, all the processes must create checkpoints simultaneously. This simplifies the recovery process because there is no problem with possible rollback propagations. Additionally, coordinated mechanisms only need one checkpoint for a successful recovery, reducing storage overhead. Non-coordinated checkpointing, in contrast, allows processes to create checkpoints at any moment. This is a great advantage because checkpoints can be created when it is most convenient, but, on a restart, a globally consistent state must be built by searching the whole set of saved checkpoints. Therefore, non-coordinated checkpointing may be affected by rollback propagation, ending up resuming from the beginning of the execution. Thus, overhead grows both in terms of performance and especially storage space, because each process must keep several checkpoints. CoCheck [102] is an example of coordinated checkpointing, while the work presented in [24] is non-coordinated.

With the objective of removing the main source of overhead, diskless checkpointing [88] [103] [106], [123] eliminated stable storage from checkpointing. However, non-persistent approaches are less resilient than their persistent counterparts, and they cannot tolerate complete system failures such as power outages. Furthermore, they increase memory, processor, and network overhead.

There are several persistent checkpointing solutions, providing either system-level or application-level checkpointing. The strongest point of system-level approaches, such as [42], [92], [41], [98], or [48] is the transparency: no changes in the application code are required. However, this comes at the cost of higher overhead in performance and disk space compared to application-level solutions.

There are some solutions that are halfway between system-level and application-level, such as that proposed by Bronevetsky et al. [25] for shared memory environments. The authors present it as an application-level approach, but the user cannot decide which data must be checkpointed nor the frequency of the checkpoints. In fact, the user can only place some calls to a given method indicating that a checkpoint must be taken. Then, the system saves the heap, call stack, and local and global variables. Given the low degree of freedom provided to the user, it cannot be considered a pure application-level solution. Apart from that, most applications do not need to save all the data stored by this approach for a successful restart, but only a subset of it. Thus, overhead is increased both in performance and disk space

usage.

There are also a variety of solutions at application-level [122], [40], [89], [36]. At application-level we can find solutions that are specific of an application or kind of application [57][57][30], and tools or libraries that can be applied to any kind of application. Some provide single level checkpointing while a few provide multi-level checkpointing. Applications that store all their checkpoints in the PFS may introduce a large amount of overhead [47], [99], [81], [91], [100]. Given the gap between the CPU and I/O performance, multi-level checkpointing [45], [115] becomes essential for reducing overhead. The key is using different—and faster—components than PFS, such as RAM disks, local node storage, or SSDs to write the checkpoints, and moving those checkpoints only when necessary, asynchronously and transparently. FTI [22], SCR [77] and VeloC [5] are multi-level CR solutions.

Those libraries overlap in their multi-level character and their multiple redundancy schemes, like partner checkpoints and erasure codes. However, they differ in the way these schemes are applied. In FTI and VeloC, the cluster topology is detected automatically and the appropriate partner nodes for the redundancy schemes are selected by the library. In contrast, SCR allows a slightly more flexible setup. Besides the standard groups NODE and WORLD, users or system administrators may define additional groups (e.g., all nodes that share a common power supply). This can be used to increase the likelihood of successful recoveries from the various redundancy levels.

VeloC was started as a project to combine FTI and SCR into a single framework. On the one hand, it offers a *memory-based mode* that is very similar to FTI. On the other hand, there is a *file-based mode* that behaves much like SCR. However, VeloC is still missing some features that FTI or SCR support, e.g., different checkpointing types (i.e., full checkpoint, differential checkpoint, etc.).

Nevertheless, current CR libraries require a considerable effort from the user side. As well, given that there is no standard library for CR, different clusters may provide different CR libraries, forcing users to write their applications to be compliant with different interfaces. Our approach, the OpenCHK model detailed in Chapter 6, reduces the burden of programmers, enhancing portability and programmability.

Efficient Management of Modern Multicore/Manycore Processors

In this chapter, we explore different approaches to enhance the management of massive amount of computational resources using task-based parallelism. Concretely, our enhancements focuses on performance and programmability. Exascale systems will include very large amounts of computational resources both at the internode and intranode levels. Thus, it is important to provide tools that are able to make the most of such resources while requiring low expertise from users.

4.1 Introduction and Motivation

The introduction of the first multiprocessor architectures led to the development of shared memory programming models. One of those is OpenMP, which became a *de facto* standard for parallelization on shared memory environments.

OpenMP [84], with its highly optimized fork-join execution model, is a good choice to exploit structured parallelism, especially when the number of cores is small. Worksharing constructs, like the well-known `omp for` construct, are good examples of how OpenMP can efficiently exploit structured parallelism. However, when the number of cores increase and the work distribution is not perfectly balanced, the rigid fork-join execution model can hinder performance.

The `omp for` construct accepts different scheduling policies that can mitigate load-balancing issues; and the `nowait` clause avoids the implicit barrier at the end of an `omp for`. Still, both techniques are only effective in a few specific situations. Moreover, the fork-join execution model is not well-suited for exploiting irregular, dynamic, or nested parallelism.

Task-based programming models were developed to overcome some of the above-mentioned limitations. The first tasking models were based solely on the tasks and `taskwait` primitives, which naturally support irregular, dynamic, and nested

parallelism. However, these tasking models are still based on the fork-join execution model. The big step forward came with the introduction of data dependences. Thus, replacing the rigid fork-join execution model by a more flexible data-flow execution model that relies on fine-grained synchronizations among tasks. Modern task-based programming models such as Cilk, OmpSs or OpenMP tasking model have evolved with advanced features to exploit nested parallelism [86], hardware accelerators [7][43][6], and seamless integration with message passing APIs such as MPI [96][95].

The flexibility of the data-flow execution model relies on the dynamic management of data-dependences among tasks. However, dependences management might introduce a non-negligible overhead depending on the granularity and number of tasks. Hence, finding the adequate task granularity becomes a key point to get good performance: too many fine-grained tasks will increase task overheads, but too few coarse-grained tasks will hinder the available parallelism. Yet, it is not always possible to reach the optimal granularity that is coarse enough to compensate for the overheads while opening sufficient parallelism. Moreover, the granularity is limited by the problem size per core. Thus, if the problem size per core is too small, the granularity might be suboptimal, hurting the performance.

We truly believe that data-flow model is the right choice for Exascale systems. These systems will contain multicore/manycore processors with high core count in each node. Therefore, using a model that relaxes the synchronization, such as data-flow, is crucial to adequately exploit the potential of the hardware. However, we are aware that tasking still presents some challenges, such as the recently mentioned granularity problem, that must be solved in order to make them a real alternative for a wide range of applications and problem sizes.

For those situations, it makes sense to combine both strategies—tasking and worksharing—in a way that we can palliate the drawbacks of each strategy while maximizing their strengths. To do so, we propose an enhanced combination of task data-flow execution (outer level, coarse-grained) with fork-join (inner level, fine-grained). The idea is to leverage the relaxed synchronization model of data-flow execution, with the the efficient fork-join model to exploit structured parallelism using a reduced number of cores. This innovative concept is the worksharing task.

Additionally, worksharing techniques are easier to apply compared to tasking. A single worksharing construct is enough to parallelize a loop. In contrast, using tasks, it requires more effort from the user. There must be at least a task per core, to feed all the cores and prevent lack of parallelism. A frequent technique applied to create enough tasks is blocking. This technique partitions a loop in several blocks,

and each of the blocks is a task. Although this is not a complex technique, it requires more effort than the worksharing alternative.

The OpenMP standard contains a directive able to distribute the iteration space of a loop into tasks, which, theoretically, enables users to parallelize a whole loop with a single construct using tasks. This is the `taskloop` construct. However, in practice, it is not useful for a single reason: it does not support data dependences. Thus, a `taskloop` creates a set of tasks that cannot have data dependences, and so, the synchronization must be done using explicit synchronization points (i.e. `taskwaits`). So, basically, we end up in a fork-join model but with increased overhead compared to worksharing constructs.

We propose adding support for data dependences to the `taskloop` construct. Our proposal enables users to use the induction variable of the loop to specify data dependences. Thus, each task created by the `taskloop` will register the data dependences specified by the user. If the induction variable is used to specify any dependence, each task will register the dependence using its own value of the induction variable. As a result, apply blocking is possible using a single construct, enhancing programmability.

Finally, we present the `taskloop for` construct. This construct is a combination of the worksharing tasks and the `taskloop` with dependences. It behaves like a regular `taskloop` but creating worksharing tasks instead of regular tasks. As a result, the enhancements of performance of the worksharing tasks and the programmability improvements of the `taskloop` with dependences join in a single construct.

4.2 Background

4.2.1 Tasking Performance Challenges

Structured parallelism can be found in most HPC applications. So, it is important to develop techniques that perform well for this kind of parallelism. Harris et al. [49], already explored the importance of properly supporting structured parallelism.

Nowadays, developers can use loop-based parallelism or task-based parallelism for coding their applications containing structured parallelism. Loop-based parallelism is quite simple to write, and it performs well in architectures with a low number of cores and applications with a small load imbalance. Despite this, it implies a rigid synchronization resulting in performance drops when facing many-core ar-

chitectures and imbalanced applications. Task-based parallelism allows a data-flow execution, which is more flexible than its loop-based counterpart. Additionally, it provides several key benefits, previously mentioned in Section 4.1. Thus, it usually performs well in many-core architectures and load imbalanced applications.

Still, an inherent problem of task programming is granularity choice. If task granularity is not adequately set, overhead may penalize overall performance. The overhead of tasks is caused by several different sources. The first one is the actual task creation, which usually implies costly dynamic memory allocations. Secondly, the computation of dependences between tasks, which involves the use of dynamic and irregular data-structures. Finally, the scheduling of the tasks across many cores can also become a bottleneck.

Task granularity and the number of created tasks are inversely proportional. Consequently, a given problem can be solved either by using many fine-grained tasks or a few coarse-grained ones. Thus, finding an adequate granularity is a key point to optimally exploit resources when using tasks [80], alleviating the aforementioned overheads, but still creating enough parallelism to maximize resource utilization.

A typical granularity chart is shown in Figure 4.1. The x-axis varies the granularity of tasks, while the y-axis represents performance. The chart presents the results of the synthetic benchmark shown in Code 4.1. There are three different series representing different problem sizes. The chart also contains coloured parts which represent different chart (not application) phases. Note that from $X=256$ to $X=1K$, phases 1 and 3 are merged. This is because it is Phase 1 for $PS=16K$, but Phase 3 for $PS=128K$ and $PS=1M$.

When the problem size is 1M and 128K, there are three well-differentiated phases. In the first phase, we can see how the performance is low because there are too many very fine-grained tasks and the overheads of creation and management of that amount of small tasks are too costly. In the second phase, performance grows until reaching peak performance. Finally, in the third phase, performance decreases again because there is not enough parallelism (i.e., not enough tasks are being created to feed all the CPUs).

Listing 4.1: OMP_F

```
for(size_t block = 0; block < NUM_BLOCKS; block++) {
    size_t start = block*TSIZE;
    size_t size = start+TSIZE > N ? N-start : TSIZE;
    #pragma omp task inout(a[start;size]) priority(block)
```

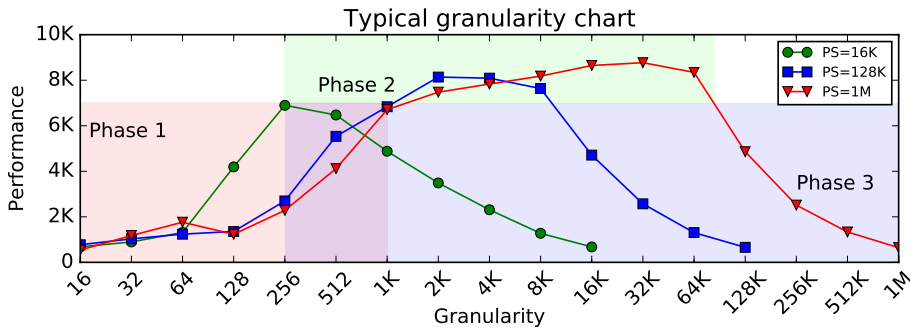



Figure 4.1: Typical granularity chart.

```

for(size_t j2=start; j2 < start+size; j2++) {
    a[j2] += b[j2]*c[j2];
}
}

```

Typically, a good granularity allows having, at least, one work unit per core to occupy all the resources. Ideally, having more than one work unit per core is better to mitigate potential load imbalance. Additionally, it is important setting a granularity coarse enough to alleviate task management overheads. However, there is a crucial factor that can limit the granularity choice: the problem size per core. The problem size per core is the result of dividing the total problem size by the number of available cores. In consequence, the problem size per core only depends on the total problem size and the number of cores available in our system.

In an ideal case, like the problem sizes of 1M and 128K in Figure 4.1, the granularity can grow until the overhead is not a problem. At that point, the problem size per core is big enough to create sufficient tasks—of a granularity that is not affected by the overhead—to feed all the resources. This happens in the second phase when peak performance is reached.

In contrast, if the problem size per core is not big enough, the developer must decide between a finer granularity that is still affected by the overhead but creates sufficient parallelism, or a coarser-granularity that is less affected by the overhead but causes a lack of parallelism. When this happens, the second phase of the typical granularity chart does not appear, being unable to reach peak performance. This phenomenon occurs in Figure 4.1 when the problem size is 16K.

Daily, developers are involved in situations where the problem size per core is

not optimal, jeopardizing the use of tasks in their applications.

Strong scaling in distributed environments

This is a common scenario in HPC environments. Strong scaling starts from a given problem size per core, and make it smaller either by augmenting the number of resources or by decreasing the total problem size. As we have seen, reducing the problem size per core while maintaining the granularity of the tasks can lead to insufficient work.

Many-core architectures

Increasingly, architectures have more and more cores. This trend directly affects the problem size per core, which becomes reduced because the same problem size is divided among more resources. Thus, setting an adequate granularity becomes harder or even impossible, leading us to either increased overhead or lack of parallelism.

Applications requiring different granularities

Many applications rely on different kernels to perform a computation, and each of them may require a different task granularity to achieve optimal performance. Finding an adequate granularity that fits all the different algorithms may be impossible. For this case, it is especially important to have a broad set of granularities where peak, or at least acceptable, performance is reached because if all the kernels have several granularities that reach peak performance, it is easier to find a granularity that performs well for all the kernels than it would be if there is a single granularity getting peak performance for each kernel.

Additionally, it may happen that an application with different kernels must share the same granularity. The reason is that the data partitioning may implicitly set the task granularity. When this happens, it is especially important having a wide set of granularities performing well in all the kernels. This way, it is easier to find a coincidence across all the sets.

Apart from this, granularity issues may prevent runtime libraries from developing sophisticated and smart policies. Those policies may introduce few overhead per task but could provide benefits in terms of programmability and performance. However, if a program contains a huge number of tasks, the aforementioned small overhead per task, rapidly becomes unaffordable. A good example is the support

for region dependences. This kind of dependences enables users to annotate their codes with the whole memory regions a task actually access. Then, the runtime library computes the dependences with all the partial overlappings, actually preventing any task that shares even a single byte to execute until the current task finishes. In Listing 4.2, using region dependences, the second task depends on the first task, while it does not when using discrete dependences (e.g., OpenMP dependences) because those only consider the start address. Region dependences are very useful to simplify codes, but they come at a cost. The computation of the dependences is more expensive compared to discrete dependencies. In consequence, if the number of tasks is huge, the overhead may become excessive.

Listing 4.2: Region deps

```
// a[0;8] means from 0 (included) to 8 (not included)
#pragma omp task inout(a[0;8])
task_body();

// a[2;6] means from 2 (included) to 6 (not included)
#pragma omp task inout(a[2;6])
task_body();
```

To sum up, task-based parallelism offers several key benefits that developers want to keep. Notwithstanding, there are currently several difficulties or problems when programming structured parallelism with tasks. (1) Granularity is critical: for that purpose, a thorough and time-costly analysis must be done in order to choose it adequately; (2) adequate granularity does not always exist: some scenarios may force developers to choose either overhead or lack of parallelism; and (3) runtime libraries cannot develop sophisticated tasking management policies: those could jeopardize the performance in programs with a very large task number. These challenges are addressed in Section 4.3.

4.2.2 Tasking Programmability Challenges

Compared to using worksharing techniques, tasks are more complicated to use. If we simply replace worksharing constructs by task constructs, there is very few parallelism, and most of the cores are idle. This is because a worksharing construct distributes the work among all the available cores, that run concurrently. In contrast, a task is a piece of code that runs only in a single core. Figure 4.2 shows such a problem. The figure also shows one possible solution, which is the use of blocking.

```

#pragma omp parallel for
for (i = 0; i < 20; i++)
    {...}

#pragma omp task
for (i = 0; i < 20; i++)
    {...}

for (i = 0; i < 20; i+=5)
    #pragma oss task
    for (ii = 0; ii < 5; ii++)
        {...}

```

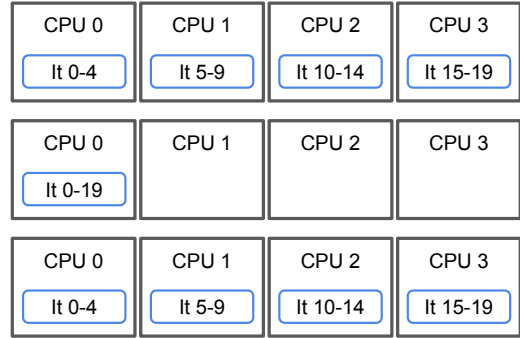


Figure 4.2: Illustration of CPU occupation using different parallelism techniques.

Listing 4.3 shows a real code using worksharing constructs and task with blocking. It is possible to see that applying blocking techniques is simple, but also that it requires more effort than using worksharing constructs. For a single loop, work-sharing constructs require only three lines of code, while tasks with blockings require five lines of code.

Listing 4.3: Comparison of a simple code using worksharing constructs and tasks with blocking

```

// WORKSHARING
#pragma omp for
for(size_t j = 0; j < N; j++)
    b[j] = scalar*c[j];

// TASKS WITH BLOCKING
for(size_t j = 0; j < N; j+=BS) {
    size_t size = j+BSIZE > N ? N-j : BSIZE;
    #pragma oss task in(c[j;size]) out(b[j;size]) label(scale)
        priority(block)
    for(size_t j2=j; j2 < j+size; j2++)
        b[j2] = scalar*c[j2];
}

```

The OpenMP standard already provides a construct known as taskloop that distributes work into several tasks. This construct is the natural replacement of work-sharing constructs to use tasks. Listing 4.4 shows the example of Listing 4.3 using the taskloop construct. Notwithstanding, the tasks created using the taskloop

construct cannot have data dependences, so they can only be synchronized using explicit synchronization points (i.e. taskwaits). As a result, we have a fork-join pattern with its rigid synchronization. So, we moved from worksharing constructs to tasks to benefit from a more lightweight data-flow synchronization, but the impossibility of using data dependences when using the taskloop construct prevents us from obtaining all its benefits.

Listing 4.4: Simple code using the taskloop construct

```
#pragma oss taskloop chunksize(BSIZE)
for(size_t j = 0; j < N; j++)
    b[j] = scalar*c[j];
#pragma oss taskwait
```

In summary, tasks require more effort from users than worksharing constructs. However, tasks provide key benefits that fit the requirements of Exascale systems better than worksharing constructs. The taskloop construct enables programmers to use tasks with a similar effort than the effort required by worksharing constructs. Nevertheless, it does not support data dependences, and this prevent users from getting the key benefits of tasking.

The challenges presented here are addressed in Section 4.4.

4.3 Worksharing Tasks

The concept of worksharing task is similar to insert a worksharing construct within a task construct in OpenMP. Listing 4.5 shows an example. Nevertheless, this OpenMP version presents several issues that worksharing tasks are able to mitigate. Firstly, this requires nested parallelism, which is not always well managed in OpenMP, becoming a potential source of overhead. Besides that, in OpenMP, every single thread that is part of the team is forced to enter the worksharing region, even if there is no work to do. Finally, there is an implicit barrier at the end of the worksharing region where all the threads must wait for the rest. In the following sections, we will explain how worksharing tasks are able to overcome or alleviate this issues. Concretely, we detail the syntax and semantics of our proposal, its constraints and how it fits into the tasking model. Also, we discuss its applicability and utility.

Listing 4.5: OpenMP code to get similar behavior than a worksharing task

```
#pragma omp task
{
    #pragma omp parallel for
    for (...) {...}
}
```

4.3.1 Syntax

We propose a new clause for the task construct. This is the `for` clause for C/C++ and the `do for` for Fortran.

A task `for`—or worksharing task—accepts all the clauses accepted by a regular task except the `final` clause because `task for` is always `final`. Note that being `final` means that no tasks can be created inside the context of a worksharing task. Additionally, it accepts the `chunksize(integer-expr)` clause. The `integer-expr` specified as a `chunksize` sets the minimum chunk of iterations that each worker is going to execute when it requests work to the worksharing task, except for the last chunk that might contain fewer iterations. If not set, the default value is defined in Equation 4.1. This default value causes that each collaborator has at least one chunk to run.

$$\text{chunksize} = \text{task_size} / \text{number_of_collaborators} \quad (4.1)$$

The `for` clause can only be applied to a task that is immediately succeeded by a loop statement. Codes 4.6 and 4.7 contain examples of code using the new clause.

Listing 4.6: Code of Figure 4.3

```
for(int i = 0; i < 2; i++) {
    #pragma oss task for [inout(a)]
    for(...) {...}
    #pragma oss task for [inout(b)]
    for(...) {...}
    #pragma oss task for [inout(c)]
    for(...) {...}
}
```

Listing 4.7: Fortran example

```
!$oss task do
do i=0, N
```

```
    call do_work();  
end do
```

4.3.2 Semantics

A worksharing task behaves like a regular task in almost everything. The main difference is illustrated in Figure 4.3 whose code is shown in Code 4.6. Regular tasks are executed entirely by a single worker concurrently, while a task `for` may be executed collaboratively by several workers, as a worksharing construct. Nevertheless, one can see in Figure 4.3 that it does not imply any synchronization or barrier at all. A worksharing task is like a regular task in this sense, and the synchronization is done through data dependences or explicit synchronization points. Note that the data dependences of the worksharing tasks are released when the last chunk is finished by the thread that runs that last chunk. This can be seen in Figure 4.3, represented by the small yellow piece at the end of the last chunk of each worksharing task.

As a worksharing construct, the iteration space of the `for`-loop is partitioned in chunks of *chunksize* size. The key point is that these chunks do not have the usual overheads associated with a task—such as memory allocation and dependences management. To run a chunk, a thread only needs the boundaries of that chunk and the data environment, much like worksharing constructs. So, in summary, a worksharing task can be run in parallel by multiple threads, better amortizing the task management overheads.

Usually, programmers use coarse granularities to overcome tasking overheads. Using tasks, coarse granularities may limit parallelism, causing some resources to be idle as in the bottom part of Figure 4.3. In contrast, using coarse-grained worksharing tasks, the work is split again into several fine-grained chunks that can be run concurrently by several workers. Thus, preventing resources from becoming idle and maximizing resource utilization, as shown in the top part of Figure 4.3.

Regarding chunk distribution, a worksharing task is highly flexible. The only guarantee is that work is partitioned in chunks of *chunksize* size and it is executed at most by N collaborators of the same *team*.

A worksharing task creates a *worksharing region* that is executed by a *team* of workers. One important property of worksharing regions is illustrated in Figure 4.3: up to N threads **may** collaborate on the completion of the work, but they are not forced to do so. This behavior happens with *TF A0* and *TF B0* which are

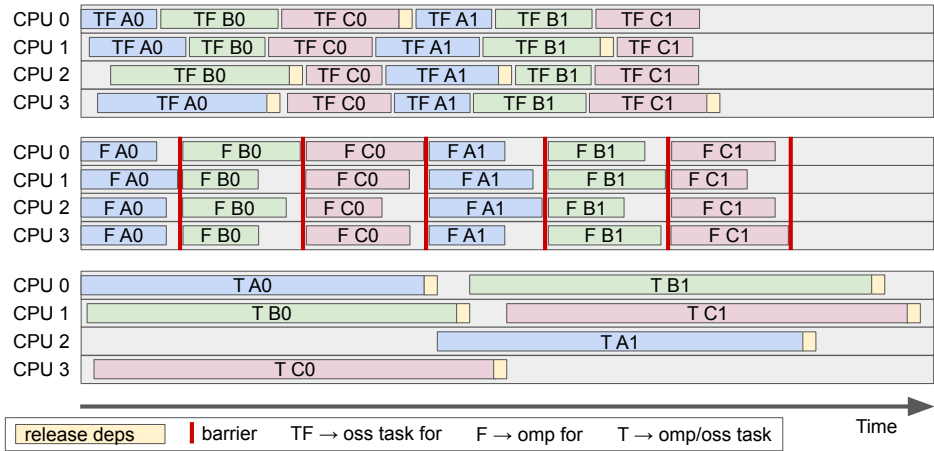


Figure 4.3: Illustration of similar execution using OmpSs-2 worksharing tasks and OpenMP worksharings.

run only by three threads while their team contains four threads. This happens because all the work has been assigned prior to the arrival of the last thread, so the last thread just goes ahead and gets more work.

A further key feature can be observed in Figure 4.3. CPUs can leave the worksharing region before the actual completion of the whole worksharing task. CPU0 finishes its *TF A0* chunk while CPU1 and CPU3 are still completing their chunks. However, instead of waiting as a regular worksharing does in the middle part of the figure, it moves forward to *TF B0*. In other words, worksharing regions do not contain implicit barriers at the end. This behavior is equivalent to set a `nowait` clause in OpenMP worksharing constructs.

This feature is especially important because it permits the pipelining of different worksharing regions. This behavior can be observed in Figure 4.3. For instance, when CPU1 finishes its *TF A0* chunk, there is no remaining work in *TF A0*. Hence, it leaves that worksharing region and joins *TF B0*. However, *TF A0* was still in execution by CPU3.

In summary, worksharing tasks implicitly alleviate the effects of a possible load imbalance through the ability of collaborators to leave a worksharing region when there is no remaining work. Thus, threads can just go forward and get more work instead of becoming idle waiting at a barrier. Worksharing tasks also palliate the granularity issues by allowing the use of coarse granularities that are partitioned anew at an additional level of parallelism. So, task management overheads are min-

imized, and resource utilization is maximized.

4.3.3 Integration in OmpSs-2

The concept of worksharing task is completely integrated into the model since at all levels it is like a task, except that it may be executed by several workers instead of by a single one. For that reason, it can interact with regular tasks without further problem using regular mechanisms: data dependences and explicit synchronization points regarding synchronization; and data-sharings for managing how the data is shared across different pieces of work.

4.3.4 Applicability

Worksharing tasks applicability is as wide as OpenMP worksharings. If the iterations in a loop are independent, then worksharing tasks can be applied. Worksharing tasks are especially useful to deal with applications containing multiple kernels especially if those present different patterns (regular/irregular). Worksharing tasks enables users to program using a pure data-flow model while efficiently exploiting structured parallelism.

4.3.5 Utility

Worksharing tasks mitigate or solve the problems presented in Section 4.2. Firstly, worksharing tasks enlarge the set of granularities that deliver good performance. In scenarios where only a few tasks are created and if these are not enough to keep all the resources busy, the use of worksharing tasks mitigate the lack of parallelism. Thus, providing several extra granularities that still work well compared to regular tasks, overall, easing the granularity choice.

Furthermore, as we already discussed in Section 4.2, there are scenarios when a good granularity does not exist and developers incur either on overhead or lack of parallelism. For that scenarios, worksharing tasks are especially useful because developers can reduce overhead by setting coarser granularities, without fearing a lack of parallelism. Given that worksharing tasks split the work among a whole team of collaborators, the total number of tasks required to keep all the resources busy is reduced from the total number of cores to the total number of teams. Hence, offering a solution to scenarios where tasks are unable to perform well.

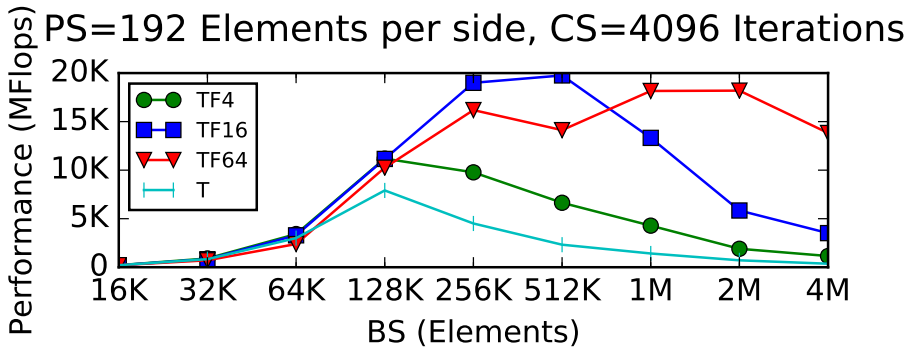


Figure 4.4: Comparison of HPCCG benchmark using region dependences with regular tasks and worksharing tasks.

Finally, since worksharing tasks are able to reduce the number of tasks by making them coarser without any significant performance loss, runtime libraries can develop sophisticated mechanisms to deal with task management. One example can be seen in Figure 4.4, where region dependences are in use. They are not suitable to be used with regular tasks given its low performance. However, they become suitable when combined with worksharing tasks.

4.3.6 Implementation

We have implemented the concept of worksharing tasks in the OmpSs-2 programming model, which relies on the Mercurium source-to-source compiler [11] and the Nanos6 runtime library [12]. In this section, we detail the extensions performed in both components to support worksharing tasks.

Mercurium compiler

We have extended the Mercurium compiler to support the new `for` clause applied to the task construct. Though, as this combination of task `for` can only be applied to loop statements, Mercurium is also in charge of checking so. In the same line, Mercurium also checks that final clause is not applied to a task `for` since it is not valid.

Given that a worksharing task may be executed by several different threads, each of them should have a correct data environment to avoid possible errors in the com-

putations. Sometimes, this may imply firstprivate or private data-sharings. Mercurium has the responsibility of providing a valid data duplication method to the runtime if firstprivate or private data-sharings have been set by the user. Otherwise, the runtime may not know how to duplicate data.

Nanos6 runtime library

Regarding the runtime library, firstly, we have extended the work descriptor of a task to include some extra information that permits splitting and spreading of the work. Basically, we add information about the boundaries of the loop and the chunksize. This information is taken at task creation time.

Then, when the worksharing task becomes ready, it is enqueued as the rest of the tasks. Eventually, the task is scheduled for execution. At this point, worksharing tasks follow a different path from regular tasks. Regular tasks are assigned to a worker thread, and it is in charge of executing the task and release its dependences if any. Worksharing tasks are also initially assigned to a worker thread but, instead of executing the whole task itself, it shares the task with its team.

Currently, in our implementation, the maximum size of the teams is defined at the beginning of the execution and remain unchanged until the end. Moreover, all the teams have the same maximum size. Currently, the default teams include all the cores of a NUMA node to prevent NUMA effects, but this can be easily changed at runtime using an environment variable. Nevertheless, as said, the size of the teams is the same for all the teams, and during the whole execution. We realize that teams with different sizes or even teams that can vary the size during the execution could benefit in some scenarios. This is a future work line that remains open.

The way a worksharing task is actually executed also differs from regular tasks. While for regular tasks we simply assign a work descriptor with its respective data environment to a thread and it just runs; worksharing tasks need some further steps. First of all, as several workers may collaborate to do the work, each of them needs its own work descriptor and data environment to avoid interferences caused by the concurrency.

Each CPU has a preallocated extended work descriptor. When this CPU receives a chunk, the preallocated extended work descriptor is filled with the actual information of the chunk that has been assigned. This represents the control information for running its part of the worksharing task.

Regarding the data environment, given that we do not know in advance how many collaborators will be—neither which of them—, each worker triggers the

duplication of the data environment in a lazy way. The worker triggers the data duplication when it has received work to do, and it has filled its preallocated work descriptor with the control information. Using the data duplication method provided by the compiler, the runtime duplicates the data environment and assigns it to the preallocated work descriptor of the thread.

Once a CPU already has the work descriptor and the data environment, it can start running its part of the work. The assignment of work from a worksharing task is done on a first-come-first-serve basis. It is guaranteed that a worker never receives fewer iterations than those specified in the chunksize clause, except the last chunk if there are not enough iterations to complete a chunk. However, it may receive several chunks.

In the current implementation, the chunk scheduling policy is very similar to the guided scheduling policy of OpenMP since the number of assigned chunks is proportional to the number of unassigned chunks divided by the number of collaborators. Note that no matter how many chunks a collaborator receives, it performs the work descriptor filling and the data environment duplication only once per work request. After terminating the assigned chunks, a thread checks if it is the last. If so, the worksharing task has finished all the chunks, and as a result, it has finished as a whole. Data dependences, if any, are released at this moment.

Otherwise, when a worker finishes its assigned chunks but the whole worksharing task has not finished there exist two possibilities: (1) all the work has been assigned and other collaborators are still running; (2) there is still work to be assigned. In (1), the worker that finishes its chunks just leaves the team and tries to get new work. In (2), the worker requests more chunks to the current worksharing task.

It is also important to highlight that assigning chunks to a worker and finishing those chunks imply some overheads that regular tasks do not have. Even though we have tuned our implementation to allow fine-grained chunks, setting an adequate chunksize is important for the proper exploitation of worksharing tasks as shown in Section 4.6.1. Furthermore, the process of requesting work crosses the scheduler path. So, it has some associated locks that may be taken into account when setting the chunksize.

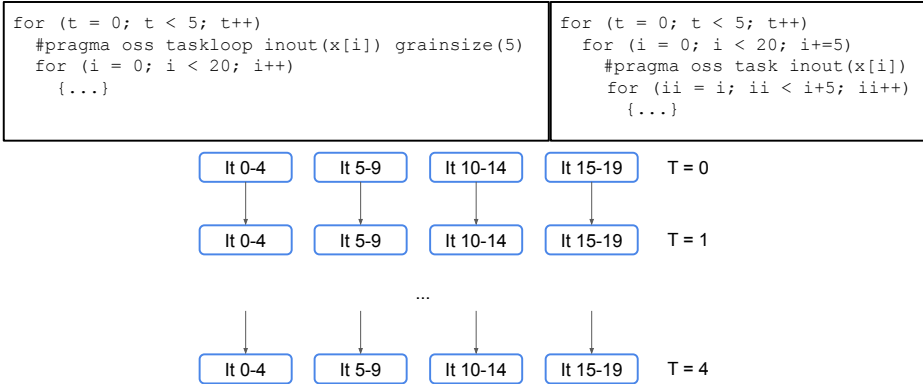


Figure 4.5: Partition of work and dependences between tasks created using taskloop

4.4 Taskloop with Dependences

In this section, we detail the syntax of our proposal to support data dependences in the taskloop construct. We also provide implementation details.

4.4.1 Syntax

We propose the use of the induction variable to define data dependences in tasks created by the taskloop construct. By using the induction variable, each task will contain its own dependences based on the partition of work they received. Figure 4.5 shows an example. There is an outer loop and an inner loop parallelized using the enhanced taskloop construct with data dependences. As we can see, the data dependences contain the induction variable. In this case, it means that each of the tasks register a dependence over the i -th element of x . As each of the tasks receive a part of the iteration space, each of the tasks will have different values for i , thereby allowing them to run concurrently, but defining a dependence with tasks of the next and previous t iteration that work over the same data. Figure 4.5 also includes the code to get equivalent behavior using regular tasks.

With this mechanism, expressivity is enhanced and the taskloop construct becomes usable in many real-world examples while keeping the key benefits of tasking.

Finally, we would like to point out that the mechanism to express the granularity of the tasks created by a taskloop construct is the grainsize clause, as shown in

```

#pragma omp taskloop inout(x[i]) grainsize(5) label( blue)
for (i = 0; i < 20; i++) {...}
#pragma omp taskloop inout(x[i]) grainsize(5) label( green)
for (i = 1; i < 21; i++) {...}

```

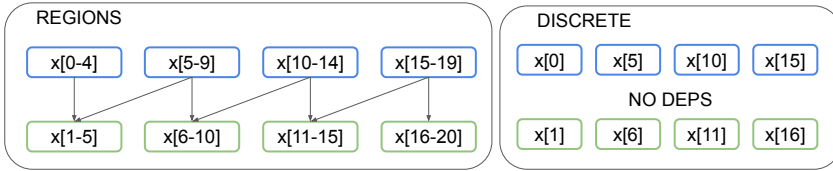


Figure 4.6: Comparison of taskloop with dependences using regions and discrete dependency systems.

Figure 4.5. If no grainsize is provided, the default value is one.

Integration with Different Dependency Systems

The OmpSs-2 programming model supports two different dependency systems: discrete and regions. We already introduced the difference between both of them in Section 4.2. Briefly explained, discrete dependences only register the start address of the data specified by the user, while dependences based on regions register the whole memory region annotated by the user. In Listing 4.2, using region dependences, the second task depends on the first task, while it does not when using discrete dependences because those only consider the start address.

We designed the taskloop to work with both dependency systems. For the discrete dependency system, if the induction variable is used, it is replaced in each of the tasks by the value of the induction variable in the first iteration run by the task. In the case of regions dependences, if the induction variable is used, the dependence is registered over the whole region comprised in the chunk assigned to the task, rather than only the first element. This is shown in Figure 4.6. Using regions, although tasks of the first and second loop do not start in the same iteration, there are dependences between tasks; while using discrete, as tasks of the first and second loop do not start in the same iteration, there are no dependences.

4.4.2 Implementation

Our proposal is done in the OmpSs-2 programming model, built on top of the Mercurium compiler and the Nanos6 runtime library. Following, we detail the extensions done in both components to support dependences in the taskloop construct.

```

#pragma oss taskloop inout(x[i]) grainsize(5)
for (i = 0; i < 20; i++) {...}
#pragma oss taskloop inout(x[i]) grainsize(5)
for (i = 0; i < 20; i++) {...}

```

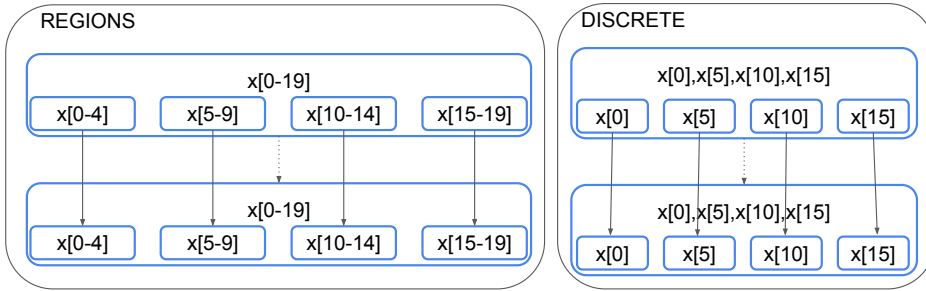
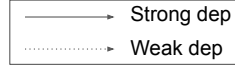


Figure 4.7: Illustration of the taskloop semantics in our implementation

We also conceptually explain our implementation.

Semantics

The taskloop construct is a convenient syntactic sugar to ease the use of tasks. It can be implemented just by applying an automatic blocking technique in the compiler side, similar to a manual blocking done by the end-user. However, our implementation is a bit different. We defined the taskloop as an entity, being a special kind of task that creates more tasks. Thus, the relationship between the taskloop and the tasks created by itself is a parent-children relationship, so we will refer to the taskloop entity as the parent, and the tasks created by itself as the children.

Figure 4.7 is an illustration of the behavior of our implementation of the taskloop construct with dependences. In the figure there are two equal taskloops one after the other. Therefore, we have two parents with four children each of them. In our implementation, the parent registers a weak dependence, and each of the children registers a strong dependence. As a reminder (more details about weak dependences can be found in Section 2.1.3), weak dependences are a special type of dependences available in the OmpSs-2 programming model. A task with a weak dependence means that the task itself is not going to perform any action over the data, but one or more of its nested subtasks will do so. Weak dependences do not imply a real dependency, and so, they not do defer the execution of tasks. They serve as a link between the dependency domains of different nesting levels. In short, this makes that a parent is scheduled for execution as soon as it is created,

but the children honor the dependencies, waiting until predecessors complete to start running.

The taskloop constructs of Figure 4.7 specify a dependence over $x[i]$. As stated in Section 4.4.1, our design is prepared to work with two different dependency systems. In the case of regions dependences, the weak dependence of the parent is registered using the whole range of the induction variable, and the strong dependences of the childs using the whole range of its assigned chunk, as shown in the figure. For the discrete dependences case, each of the childs register the dependence using only the first iteration of its assigned chunk. As well, the parent registers a weak dependence for each of the children, using only the first iteration of each of the chunks assigned to the children. This behavior is also shown in Figure 4.7.

We chose this implementation because of three main reasons: (1) this way, several taskloop entities can run concurrently, enabling users to have multiple creators rather than a single one; (2) we give more control to the runtime, that can apply further optimizations; and (3) we can easily identify taskloops when using our instrumentation libraries.

Mercurium compiler

The Mercurium compiler has been extended to support the use of data dependences in the taskloop construct. Mercurium is a source to source compiler, meaning that it receives code as an input, and generates code as an output. Mercurium creates a function to register dependences per each task construct found in the user code. To support the use of the induction variable in the taskloop dependences, Mercurium has to accept a new parameter in the functions used to register dependences. Given that in our implementation it is the runtime who partitions the work and assigns iterations to the tasks, Mercurium must receive the information of the assigned iterations to replace the induction variable by its real value.

Additionally, in the same line, Mercurium creates a function per task type including the user code that the task has to run. In this case, it also has to receive an additional parameter: the iterations that each task has to run.

Finally, when creating a taskloop entity, Mercurium has to enable some flags to let the runtime system now that this is not a regular task, but a taskloop.

Nanos6 runtime library

In the runtime system the first step is to extend the work descriptor of a task to include the iteration space of the loop, and the grainsize specified by the user, if any.

Then, after the `taskloop` is created, it registers the dependences. In the case of regions dependences, it simply passes to the compiler the whole iteration space. In contrast, in the discrete dependences case, it must mimic the partitioning of the loop, to register the dependences using only the first iteration of each task. Once the dependences of the parent are registered, it is immediately scheduled for execution.

When the parent starts running, it creates the children assigning part of the iteration space to each of them, and registering strong dependences in their part of the data. Depending on whether the predecessors have already finished or not, the children remain pending or are scheduled for execution. Eventually, the children will run only the iterations assigned by the parent.

We would like to point out that our current implementation focuses on programmability. Therefore, we are trying to provide a simpler way of using tasks that introduces no significant overhead compared to using other techniques such as manual blocking. Nevertheless, the `taskloop` construct provides the opportunity to apply further optimizations that cannot be applied in the case of manual blocking. Such optimizations could include a single allocation for all the children tasks, instead of allocating space for each of the children tasks individually; or the application of throttle policies to mitigate memory overuse when there are too many tasks in flight.

4.5 Combining Taskloop and Worksharing Tasks

In previous sections, we described the worksharing tasks and the `taskloop` with dependences. We introduced some challenges that current task-based programming models present, and how our contributions are able to mitigate or overcome such challenges. Specifically, worksharing tasks mitigate granularity problems enhancing performance across several different workloads, systems and scenarios; while `taskloop` with dependences eases the development of parallel programs using tasks, introducing no significant overhead. The enhancements of both contributions are demonstrated in our evaluation, in Section 4.6.

In this section, we present the combination of both contributions in a single construct that joins the enhancements in performance and productivity of the two previous contributions. Such a combination is the `taskloop for` construct. It is basically a `taskloop` with dependences support that creates worksharing tasks rather than regular tasks.

4.5.1 Syntax

In this section we introduce the syntax of the `taskloop for` construct. Listing 4.8 shows a brief example of how to use this new construct and which clauses accepts. A `taskloop for` accepts two clauses, apart from the dependences annotations. The first one is `grainsize`, that defines the number of iterations that each of the children tasks (except the last one) will run. The second one is `chunksize`, that specifies the minimum number of iterations that each worker of the worksharing tasks will run, except for the last chunk that might contain fewer iterations. If not set, the default value for the `grainsize` is 1, while the default value for the `chunksize` is defined in Equation 4.2. The `taskloop for` construct can only be applied to a loop statement.

$$chunksize = task_size / number_of_collaborators \quad (4.2)$$

Listing 4.8: Example of use of `taskloop for` construct

```
#pragma oss taskloop for [grainsize(gs)] [chunksize(cs)] [deps]
for(int i = 0; i < 1024; i++) {...}
```

4.5.2 Semantics

The `taskloop for` construct partitions a loop in two different levels. Firstly, the iteration space is partitioned into worksharing tasks of `grainsize`, which are later partitioned again into chunks of `chunksize`. Figure 4.8 shows an example of such a partitioning.

4.5.3 Implementation

Considering that we already extended the Mercurium compiler and the Nanos6 runtime system to support the `taskloop` construct with dependences and the work-sharing tasks, the modifications required to support the `taskloop for` construct are

```
#pragma oss taskloop for inout(x[i]) grainsize(4) chunksize(1)
for (i = 0; i < 20; i++) {...}
```

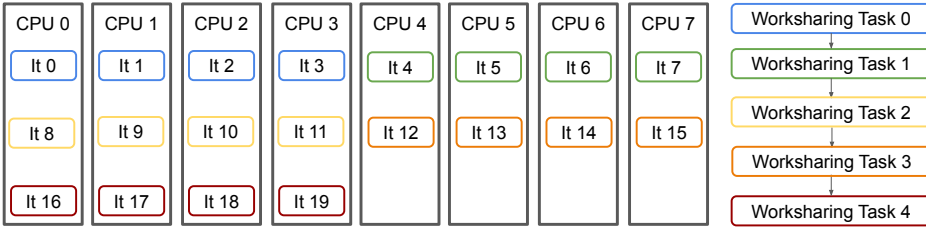


Figure 4.8: Illustration of the taskloop for semantics

very few. From the compiler side, we only need to enable an additional flag to let the runtime know that it must create worksharing tasks instead of regular tasks.

4.6 Evaluation and Discussion

4.6.1 Worksharing Tasks - Shared Memory Environment

In this section, we provide an evaluation of our proposal, as well as a discussion of the results. First of all, we introduce the environments and platforms in which the experiments were conducted. Following, the benchmarks used and the different implementations developed are described. Then, for each experiment, we detail the methodology followed along with the results and discussion about them.

We include three different experiments. The first one is a granularity analysis on a many-core system. The objective is to show how the traditional ways of exploiting parallelism may easily suffer from a lack of parallelism on many-core architectures. The second experiment is a chunksize granularity analysis which aims to stress the importance of an adequate chunksize. Finally, the third experiment is a pseudo strong scaling experiment to illustrate some scenarios where the problem size per core prevents setting a good task granularity.

We have used the OmpSs-2 discrete dependency system to make comparisons with OpenMP fairer.

Environment

The experiments were carried out on two different platforms. The first platform is composed of nodes with 2 sockets Intel Xeon Platinum 8160 2.1GHz 24-core and

96GB of main memory [21]. The second platform is composed of nodes with 1 socket Intel Xeon Phi CPU 7230 1.3GHz 64-core and 96GB of main memory plus 16GB of high bandwidth memory [20]. As well, the second platform can run up to 4 threads per core, reaching up to 256 threads. In our experiments, we did use all the threads for two reasons. Firstly, because we checked the performance and compute-bound benchmarks were obtaining noticeable benefit from using all the threads, while memory-bound benchmarks were obtaining slightly (2%-3%) worse performance using all the threads than using a single thread per core, so it is worth to use all the threads. Secondly, as we want to show scenarios where traditional approaches suffer from lack of parallelism, the higher amount of threads is helpful for this purpose.

Regarding the software, we used the Mercurium compiler (v2.3.0), the Nanos6 runtime library, the gcc and gfortran compilers (v7.2.0), and the Intel compilers (v17.0.4).

We would like to highlight that all the experiments have been run using the interleaving policy offered by the numactl command, spreading the data evenly across all the available NUMA nodes, in order to minimize the NUMA effect.

Benchmarks

We have considered four different benchmarks for the evaluation: the High Performance Computing Conjugate Gradient (HPCCG) [76], the matrix multiplication kernel (MATMUL), the N-body simulation and the Stream benchmark [114]. HPCCG and the Stream benchmark were selected as representants of memory-bounded benchmarks while MATMUL and the N-body simulation represent computed-bounded workload.

Listing 4.9: OMP_F(S/D/G)

```
#pragma omp for \  
schedule( \  
[static/dynamic/guided],TS)  
for(i=0; i<PS; i++)  
    do_work(i);
```

Listing 4.10: OMP_T/OSS_T

```
for(i=0; i<PS; i+=TS)  
    #pragma [omp/oss] task \  
    depend(inout: i)  
    for(j=i; j<i+TS; j++)  
        do_work(j);
```

Listing 4.11: OMP_TTL

```

for(i=0; i<PS; i+=TS)
  #pragma omp task \
  depend(inout: i)
  {
    #pragma omp taskloop \
    grainsize(cs)
    for(j=i; j<i+TS; j++)
      do_work(j);
  }

```

Listing 4.12: OMP_TF(N)

```

for(i=0; i<PS; i+=TS)
  #pragma omp task \
  depend(inout: i)
  {
    #pragma omp parallel \
    for schedule(guided,cs)
    for(j=i; j<i+TS; j++)
      do_work(j);
  }

```

Listing 4.13: OSS_TF(N)

```

for(i=0; i<PS; i+=TS)
  #pragma oss task for chunksize(cs) inout(i)
  for(j=i; j<i+TS; j++)
    do_work(j);

```

For each of them, we have developed six different versions. Code 4.9 implements a version using OpenMP parallel for with the static (OMP_F(S)), dynamic (OMP_F(D)) or guided (OMP_F(G)) scheduler. Code 4.10 shows a version using tasks in both OpenMP and OmpSs-2. This is a blocked version where each task computes a block of TS size. Code 4.11 is a version using the OpenMP taskloop. However, as taskloops do not accept data dependences, there is a first decomposition using tasks with data dependences. Then, inside each task, the block of TS size is partitioned anew using a taskloop. Code 4.12 is quite similar to the previous code just replacing the taskloop inside the tasks by a parallel for with guided scheduling, to make it similar to our worksharing tasks. Finally, Code 4.13 illustrates an implementation done with worksharing tasks.

The N in codes 4.12 and 4.13 indicates the number of threads used in each work-sharing construct and the maximum number of collaborators in a worksharing task, respectively.

As a final remark, all the OpenMP implementations have been compiled and run with Intel OpenMP.

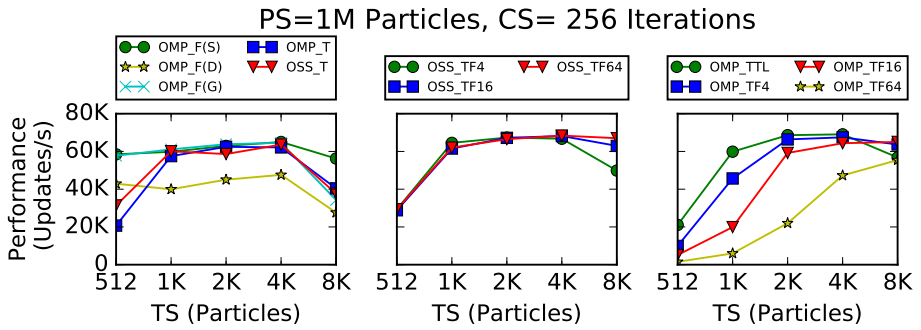


Figure 4.9: Granularity chart of different implementations of the N-body simulation.

Granularity evaluation in many-core architecture

This subsection is devoted to performing a deep evaluation of our proposal in a many-core architecture such as the Intel KNL. So, the experiments were conducted on the second platform.

In this section, we analyze the behavior of a compute-bound benchmark, the N-body simulation; and a memory-bound benchmark, the Stream benchmark.

In this experiment, we wish to show how the traditional ways of exploiting parallelism—worksharings and tasks—suffer from a lack of parallelism when the granularity is coarse. In that scenario, the versions using nested levels of parallelism—OSS_TF, OMP_TTL and OMP_TF—can perform better because they allow higher resource utilization.

The results presented were obtained by averaging the execution times of 5 different runs per version.

N-body simulation Left chart of Figure 4.9 compares how OMP_T, several OMP_F, and OSS_T versions perform with different granularities. The x-axis determines the size of the blocks. The y-axis represents performance. For the OMP_F version, TS means the chunksize specified in the `schedule(static,TS)` clause.

OMP_F(S) and OMP_F(G) implementations perform well almost across the whole set of granularities but the last one. This happens because the heaviest computational loop contains as many iterations as the number of blocks. Thus, when the block size is 8K, there is a lack of parallelism because there is only work for $(1M/8K = 128)$ threads, so the other 128 are idle, and performance falls. Note

that these versions performs well even when using quite small block sizes, where tasks suffer. This is because worksharing constructs introduce a few overhead in comparison with tasks. Notwithstanding, dynamic scheduler is performing quite badly. The overhead is introduced by the dynamic handling of chunks. It also happens with guided scheduling, but dynamic uses exactly the chunksize set by the user, while guided uses it as a minimum, and so may get bigger chunks, reducing the overall number of chunks and consequently the overhead.

The `OMP_T` and `OSS_T` versions start far from the worksharing because of the overhead introduced by tasks where the granularity is too fine. Then, they get peak performance until the last granularity when the performance falls for the same exact reason than `OMP_F`: there is not enough parallelism.

The second and third chart of Figure 4.9 exhibit the results for `OSS_TF(N)`, and `OMP_TF(N)` and `OMP_TTL`, respectively. There, one can see an important difference with respect to the previous versions. The difference is that performance does not fall for the biggest granularity when N is big enough. This means that these implementations are able to prevent the lack of parallelism when the granularities are too coarse. As all these implementations are using a nested level of parallelism, the lack of parallelism in the outer level is alleviated by using the idle resources in the inner level. Consequently, peak performance is maintained for a broader set of granularities than traditional implementations do.

That being said, there are some other interesting points in the second and third charts of Figure 4.9. Firstly, it is possible to observe divergences among the distinct OpenMP series in the third chart. `OMP_TTL` adds no extra overhead compared with using only tasks, in the lower granularities. Then, it gives a small boost to the performance in the peak granularities. Finally, for the coarser granularities, it starts falling, but the drop is less pronounced than the drop in the `OMP_T` version. In contrast, all the `OMP_TF` versions are introducing extra overhead comparing with `OMP_T`. Note that this extra overhead becomes bigger as N grows. However, they are also able to provide a small increase in the peak, like `OMP_TTL`.

On the other hand, the `OSS_TF` versions, shown in the second chart of Figure 4.9, are not introducing further overhead with respect to `OSS_T`, even with the biggest N , while they are also introducing a small improvement in the peak performance.

Stream benchmark The topmost chart of Figure 4.10 compares the performance of `OMP_T`, `OMP_F`, and `OSS_T` versions using different granularities. The x-axis determines the size of the blocks. The y-axis represents memory bandwidth. For the

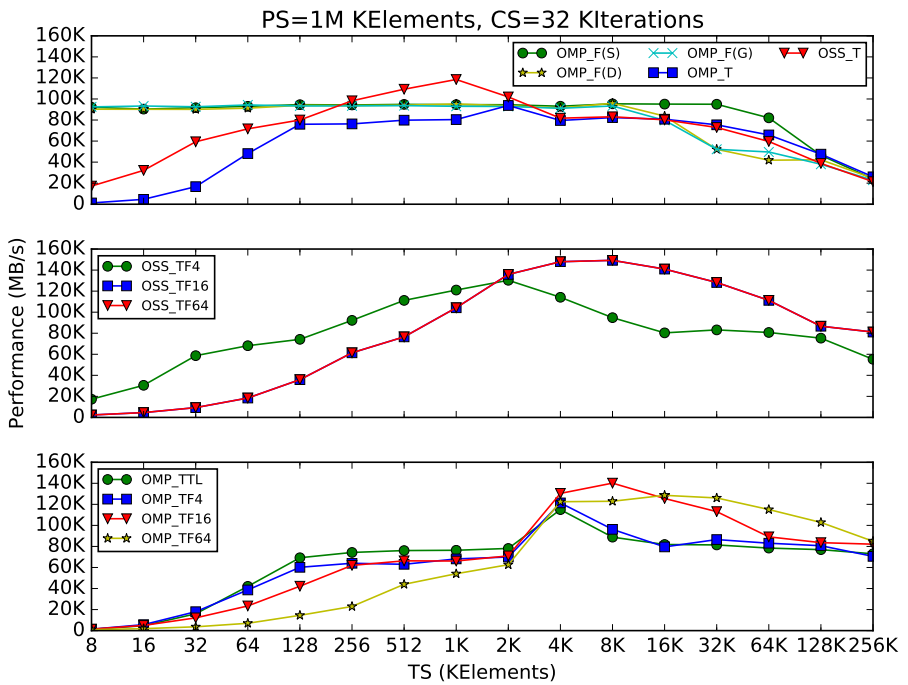


Figure 4.10: Granularity chart of different implementations of the Stream benchmark.

OMP_F version, TS means the chunksize specified in the `schedule(static, TS)` clause.

The main difference between the Stream benchmark and the N-body simulation is the weight of the computation, which is much lighter for the Stream benchmark. Therefore, it needs bigger granularities to hide the overhead of tasks. In the topmost chart of Figure 4.10, the first granularity in which tasks get good performance is 3072 Kbytes, while for the N-body simulation, it was 44 Kbytes, almost 70x more. Given that we need bigger granularities, it is more likely to end up in a granularity that constraints parallelism.

Looking at the topmost chart of Figure 4.10, it is interesting to point out that the OMP_F versions performs well even for the lowest granularity, as happened in the N-body simulation, confirming that it introduces very small overhead. All the OMP_F versions perform very similarly except for the biggest granularities. The reason for that remains in a small change in the source code: the version with static scheduler can use the `nowait` clause. Hence, the OMP_F(S) version only waits once at the end of the four loops while the rest waits four times, one per each loop. So, when granularities are fine, each thread runs several chunks and load balancing problems can be solved by the dynamic/guided scheduler. However, with the biggest granularities, very few chunks, or even a single one, are run by each thread, so load balance problems make a significant difference, happening 4 times against only 1.

Apart from this, in the same chart is possible to see the tasking versions outperforming all OMP_F versions at some specific granularities. The reason for this is that tasking versions are able to exploit some data locality due to the immediate successor mechanism of the scheduler. With this mechanism, when a task finishes, if some other successor task becomes ready due to the data dependences release of the finished task, the successor is bypassed to the same CPU to exploit data locality. Finally, for the biggest granularities, there is a performance drop in all the versions since there is insufficient parallelism given that few tasks are created. For instance, for the biggest granularity, only $(1M/256K) = 4$ tasks are created, so 252 threads are idle.

The center chart and the bottom chart of Figure 4.10 exhibit the results for OSS_TF(N), and OMP_TF(N) and OMP_TTL, respectively. Again, like for the N-body case, there exist important dissimilarities comparing these versions with the ones in the topmost chart of Figure 4.10. The main one is that the biggest granularities are not falling so much. Again, the reason for this is that the additional level of parallelism introduced in these implementations palliates the lack of parallelism in the outer level. So, we end up having a wider set of granularities reaching good performance.

Interestingly, in the versions shown in the second and third chart of Figure 4.10

there is a considerable speedup in comparison with its tasking counterparts. For OmpSs-2, second chart, OSS_TF64 gets a 1.25x against OSS_T. For OpenMP, the third chart, OMP_TF16 gets a 1.5x speedup against OMP_T. The reason for this is that they are able to better exploit the memory hierarchy. For instance, when the block size is 8 KElements, using N=16, there are at most 16 tasks running concurrently, that means 3GB. In contrast, tasks imply N=1, so that means 48GB. The high bandwidth memory of the KNL, which is acting as an L3, has 16GB of capacity. Then, for N=16, the whole dataset fits in L3, while for N=1, it does not.

Unlike with the N-body case, the OSS_TF(N) versions, shown in the second chart of Figure 4.10, do introduce further overhead with respect to OSS_T when N starts growing. The same happens with their OpenMP counterparts, shown in the third chart, and in fact, OMP_TF(N) versions are introducing much more overhead than OSS_TF(N). Anew, OMP_TTL does not introduce extra overhead comparing to OMP_T.

Overall, we would like to highlight how the set of granularities achieving peak performance becomes wider as N increases. This is a consequence of better resource exploitation. When N is small and TS is big, few tasks are created. If the number of created tasks is smaller than the number of concurrent teams, it is guaranteed that several resources will do nothing because some teams never get a task, hindering performance. When N grows, there are fewer teams—with many more collaborators—, and so it becomes more difficult for a team to get no tasks. Thus, it is unlikely that any of the resources remain idle. However, as the team size increases, the contention inside it also increases and may threaten performance. Overall, it looks like the best option is to use a big N, but still allow several concurrent teams.

The goal of this experiment is to show that traditional ways of exploiting parallelism suffer from a lack of parallelism when using coarse granularities. We did show that lack of parallelism on traditional approaches. However worksharing tasks still perform well in scenarios where traditional approaches do not. We can conclude that worksharing tasks offer a wider range of granularities delivering good performance making granularity choice easier and not so critical, especially when using large teams.

From this experiment, we can also conclude that the number of collaborators (N) is important for achieving good performance. Users must take into account several considerations for choosing it. The first one is the **number of worksharing tasks**. As happens with regular tasks, the best performance is achieved when all the resources are busy. Therefore, if there are many tasks, N can be lower, since the teams will be still busy. Oppositely, if there are only a few tasks, N must be bigger,

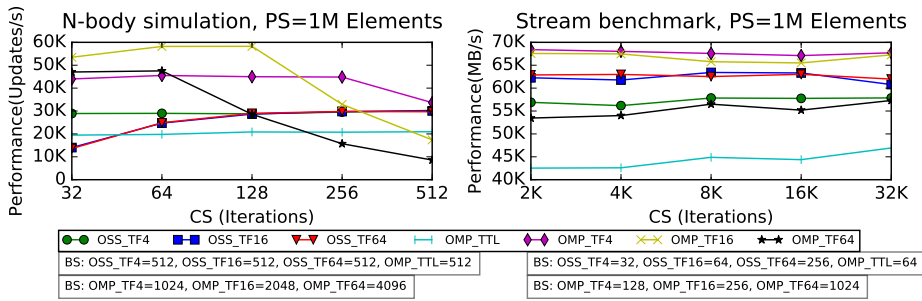


Figure 4.11: Chunksize granularity of an N-body simulation and the Stream benchmark.

so that the total number of teams is reduced, and they can be occupied with such a low number of tasks. The second one is **lock contention**. Each team contains a lock which is shared among all the collaborators. Although it is optimized, more collaborators introduce more contention into the lock. Thus, using a single group with all the available cores may result in performance degradation. The last one is **hardware layout**. We do not recommend going beyond a CPU socket when setting up teams. As a general recommendation, we suggest using one or two teams per socket. In fact, the default value of our implementation is one team per socket.

Chunksize granularity

The objective of this experiment is to show that the chunksize may affect the performance as much as the task granularity. Thus, it must be considered and adequately tuned.

Figure 4.11 show an analysis for the N-body simulation and the Stream benchmark of different chunksizes for a fixed problem size. The block size may vary for different versions, but it is the same for all the different chunksizes of a version. In consequence, series cannot be compared against others, only points within the same series can be compared. The block size is a point in the first phase of a typical granularity chart, where tasking overheads still hinder performance.

In the charts, the x-axis stands for chunksize (CS) in number of iterations, while the y-axis shows the performance for the N-body simulation, and the memory bandwidth for the Stream benchmark. The chunksize must be lower or equal than the TS, because a single chunk cannot do more iterations than those in the whole block.

Looking at the right chart of Figure 4.11, it can be seen that the chunksize does not have any effect on the Stream benchmark. This happens because the limiting resource in this benchmark is the memory. Thus, waiting to acquire some locks, or letting some resources become idle, wasting CPU time, is not so critical like in compute-bound benchmarks.

The chunksize is crucial in the N-body simulation, as can be seen in the left chart of Figure 4.11. Regarding worksharing tasks, for medium and large values of N, an adequate chunksize provides +2x of speedup compared with a bad chunksize. The reason for this is that an excessively small chunksize may imply many more requests to the scheduler, augmenting the contention on the scheduler locks. Making it too large does not affect, because even if there are not enough chunks in a task for to feed all the workers, new work can start.

In contrast, the left chart of Figure 4.11 shows the opposite behavior for OpenMP. It almost does not matter how small the chunksize is. The OpenMP guided policy assigns chunks dynamically. The actual chunk size is proportional to the number of unassigned chunks divided by the number of threads in the team, with the constraint that it can never be lower than the value set by the user. So, usually, big chunks are assigned at the beginning. Then, they become smaller and smaller until the last iterations where the restriction appears. So, when a user sets a chunksize too small, it only affects a few chunks at the end of the execution, and so it does not make a big difference. However, it is affected if the chunksize becomes too big since it cannot feed all the cores and some of them may wait in the barrier until the rest finish.

We have evidenced that chunksize may be important in some applications; while completely nimium in others. As a general recommendation, we suggest using $CS = TS/N$ so that each collaborator in the team has at least one chunk to execute. Nonetheless, having at least one chunk per collaborator is not really important if we have several ready tasks at the same time, because in that case, collaborators can get new tasks. In contrast, when there are only a few ready tasks, it is important to have as many chunks as collaborators or they will probably remain idle. Furthermore, the cost of the computation is also important. Heavier computations can work well with lower chunksizes while lighter computations will require bigger chunksizes to palliate the overheads.

Pseudo Strong scaling

In strong scaling experiments, there is a fixed problem size and the number of computing resources increases, resulting in a smaller problem size per core at each new point. Our pseudo strong-scaling experiment consists in fixing a given number of resources and decreasing the problem size, obtaining a smaller problem size per core at each new point of the experiment like in regular strong scaling experiments. The goal is to illustrate that there exist scenarios where the problem size per core prevents the possibility of setting an adequate granularity. In these scenarios, either task management overheads—if the granularity is too fine— or lack of parallelism—if the granularity is too coarse— hinders performance. Thus, by using nested levels of parallelism that allow the use of coarse-grained tasks that are then split into several chunks, performance improves. For this experiment, we have used all the benchmarks presented in Section 4.6.1.

The results of the experiment are presented in two charts per benchmark, one per platform. In these charts, in the x-axis, there are different problem sizes. The left y-axis represents performance while the right y-axis stands for work units per hardware thread. The charts show four different series (bars) for each problem size. Those series are six different implementations, `OMP_F(S)`, `OMP_F(D)`, `OMP_F(G)`, `OMP_T`, `OSS_TF` and, finally, the one obtaining best performance between `OMP_TF(N)` and `OMP_TTL`. For each of the bars, there is also a circle pointing out the number of work units per hardware thread for that specific configuration. Finally, there is a horizontal line which corresponds to 1 work unit per hardware thread. Thus, it is easy to see when there is at least work for all the resources (above the line) and when there is not (below the line).

For all the versions, we have explored the whole set of combinations for each of the parameters (TS, CS and N if applicable), and selected the best configuration.

Figure 4.12 shows the results for the N-body simulation. In the first platform, left chart of Figure 4.12, all the implementations perform very similarly for the three biggest problem sizes, with all `OMP_F` versions standing out a bit for the biggest problem size. Then, performance decreases for all the versions. For the biggest problem sizes, all `OMP_F` versions deliver similar performance. Then, `OMP_F(S)` outperforms its dynamic and guided counterparts. Dynamic and guided schedulers introduce more overhead than static. They are worth if the application is highly imbalanced, but this is not the case. Hence, they are introducing overhead but not getting any benefit, hurting performance. It is not significant in the biggest problem sizes because the long execution time amortizes the overhead, but it pops up when the

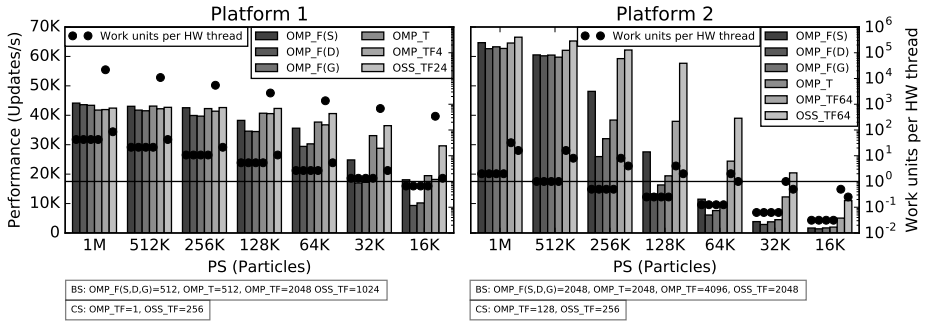


Figure 4.12: Pseudo strong scaling charts of the N-body simulation.

execution shortens.

Regarding OMP_F and OMP_T versions, except for the lowest problem size, there is at least one work unit for each hardware thread, so only the lowest granularity has a lack of parallelism. From 32K to 128K, there is a load balancing problem. There are, respectively, 5.33, 2.67, and 1.33 work units per thread for 128K, 64K, and 32K. This means that some threads are performing more work than others, and those others are just idle wasting resources.

OMP_TF version has more than enough parallelism when considering both levels of parallelism, but the nested parallel regions are introducing a lot of overhead, and that hurts performance. Additionally, for the lowest problem size, there is not enough parallelism in the first level to feed all the resources, so that even having enough work units when considering combined parallelism, those work units are concentrated in only half of the resources, remaining the rest idle.

In contrast, it can be seen how OSS_TF is able to maintain the performance much better than the other versions, reaching up to a 1.5x speedup for the lowest size against the best competitor. Note that the problem size is reduced by up to 64x, but OSS_TF performance is still 70% of the peak performance while the rest are below 50%. The main reason is that even for the lowest size, we reach high levels of hardware resources occupancy thanks to having very few (concretely 2) teams with a high amount of CPUs. This allows not only the parallelism to be maximized but also to improve load balancing.

In the second platform, the right chart of Figure 4.12, the behavior is very similar but accentuated because of the large number of cores available. The performance of OMP_F and OMP_T falls very quickly because of the lack of parallelism. In contrast,

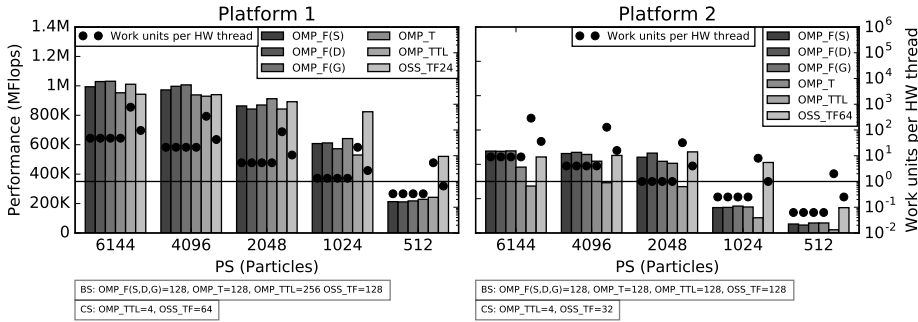


Figure 4.13: Pseudo strong scaling charts of the matmul benchmark.

both `OMP_TF` and `OSS_TF` are able to maintain acceptable performance even when there is not enough parallelism in the first level, thanks to its nested level of parallelism. Nevertheless, `OSS_TF` outperforms its OpenMP equivalent, becoming the difference between them bigger as the problem size per core decreases. `OSS_TF` is able to get up to 2x speedup compared with `OMP_TF` and more than 5x compared with `OMP_T` and `OMP_F`.

Figure 4.13 shows the results for the MATMUL benchmark.

Regarding the results of the first platform, displayed in the left chart of Figure 4.13 it is possible to observe one more time the performance reduction as the problem size becomes smaller. The reasons are load balancing, like for the N-body simulation, for `PS=1024`; and the lack of parallelism for `PS=512`. Yet, `OSS_TF` keeps performance better than the other versions, achieving up to a 2x speedup against the best OpenMP version.

The nature of the chart of the second platform, shown in the right chart of Figure 4.13, is similar to the previous, but in this case, the main problem is actually the lack of parallelism given the large number of available resources. In this platform, nonetheless, `OSS_TF` is able to reach up to a 2.7x speedup versus the best OpenMP version.

The results of the Stream benchmark, available in Figure 4.14, are different than the previous. In the previous benchmarks, there was a trend where lowering the problem size led to a performance drop, especially in the OpenMP versions.

In the first platform, the topmost chart of Figure 4.14, this does not happen, or at least, the drop is not as large. The main reason is that the limiting resource in this benchmark is the memory bandwidth instead of the CPU. Thus, even without

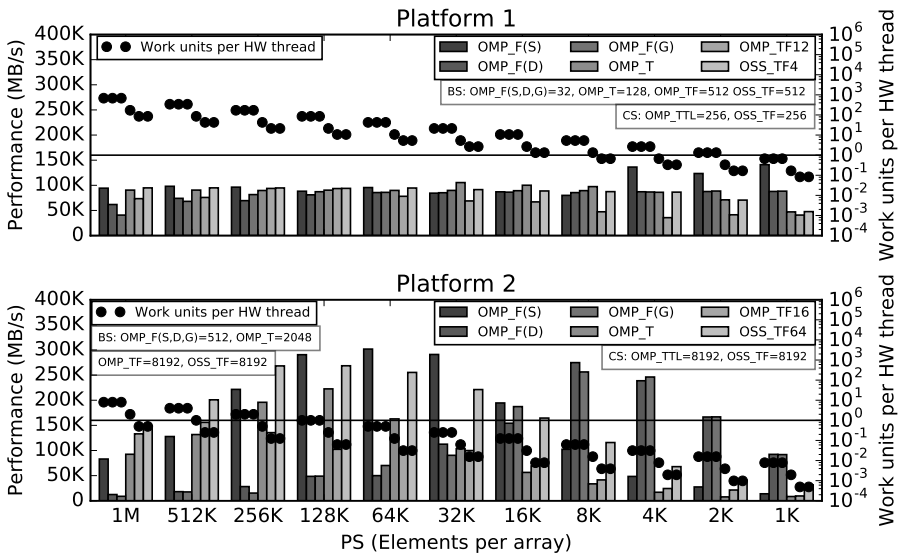


Figure 4.14: Pseudo strong scaling charts of the Stream benchmark.

using all the resources, peak performance can be achieved. For this reason, decreasing the problem size, leading to a lack of parallelism, is not so important in this benchmark.

That being said, we can see how the OMP_F versions are even increasing its performance, especially OMP_F(S). The increase stems from the data locality exploitation given that in the lowest sizes, the whole problem or a large part of it fits in the caches. This effect is seen in none of the other versions mainly due to two reasons. The first one is the pollution of the caches caused by the runtime libraries. The second one is that the static scheduling of the OMP_F(S), combined with the nowait clause, allows that a CPU executes the same elements of each loop, maximizing data reuse. In contrast, task-based versions, although they have immediate successor policy which favors locality, is not so perfect as the OMP_F(S) one. Regarding the dynamic and guided versions, they need to run the whole loop, iterating over the whole data arrays before moving forward to the next loop, preventing them from any kind of data reuse.

In the second platform, the bottom chart of Figure 4.14, the effect of insufficient parallelism is notable, like in the previous benchmarks. It is caused by the large

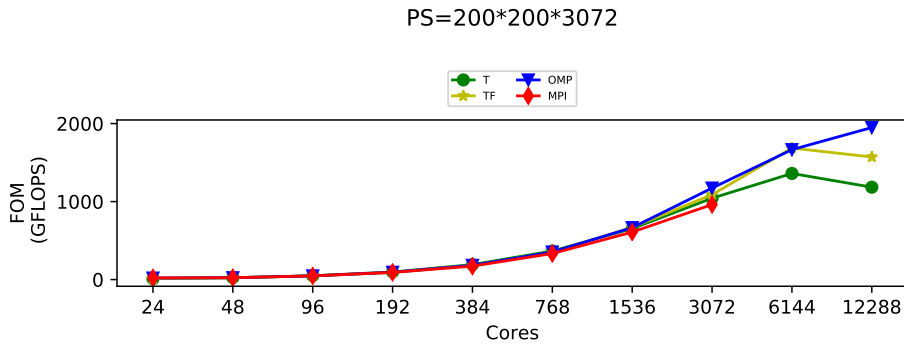


Figure 4.15: Pseudo strong scaling charts of the HPCCG benchmark.

number of cores available in this platform, which needs a bigger value of problem size to keep the problem size per core able to perform decently. In this platform, there are four versions that stand out. Again, the reason is data locality. This platform incorporates a 16GB high bandwidth memory used as L3. The problem sizes where we get the best performance are those where the whole data set fits in cache while there is enough parallelism.

Figure 4.15 introduces the results obtained in the HPCCG benchmark. In this benchmark, each chart only contains two series instead of the four mentioned previously. The reason is that the HPCCG benchmark contains several reductions. The Intel OpenMP compiler and runtime we have used do not support task reductions. Therefore, all the versions using tasks (OMP_T, OMP_TF and OMP_TTL) perform poorly.

The results of the first platform are given in the left chart of Figure 4.15. In that chart, it is possible to see again, like in the previous benchmark, how the performance of all OMP_F versions drops when the problem size per core decreases. In contrast, the performance of OSS_TF remains very similar until the lowest problem size where it finally drops. Despite the drop, OSS_TF gets more than 9x speedup compared to OMP_F for that problem size. The reason for the drop, one more time, is the lack of parallelism. However, it is possible to see some of the circles below the line while the performance is still peak. This happens for the same reason than the Stream benchmark. HPCCG is also memory-bound, so it does not need to occupy all the cores to reach peak performance. There are also some differences depending on the scheduler for the OMP_F versions. Dynamic and guided seems to

perform slightly better. The reason is load imbalance.

In the second platform, the right chart of Figure 4.15, the trends are similar. OMP_F performance is deteriorated by the lack of parallelism. However, the OSS_TF performance in this platform, falls faster because there are many more resources available, and even OSS_TF is not able to exploit enough of them when the problem size is reduced. Still, OSS_TF outperforms OMP_F by up to 1.65x.

We have demonstrated that when performing strong scaling experiments we can easily get into scenarios where the problem size per core prevents traditional ways of exploiting parallelism to get good performance. At the same time, we have shown how worksharing tasks mitigate the lack of parallelism issue being able to perform well across several benchmarks on two different platforms, even in scenarios where traditional approaches suffer.

4.6.2 Worksharing Tasks - Distributed Environment

Our distributed environment evaluation uses two well known mini-apps: **High Performance Computing Conjugate Gradients (HPCCG)** [76] and **Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)** [66]. Both of them are well known proxy apps used to assess the performance of software and hardware.

HPCCG is a simple conjugate gradient benchmark code for a 3D chimney domain. It is part of the Mantevo project [50]. This mini-app presents very well structured parallelism, and it is memory-bound.

LULESH solves the Sedov blast wave problem for one material in three dimensions [53]. It is one of the five challenge problems in the DARPA UHPC program. It is designed to test different tuning techniques and programming models. LULESH contains different phases and includes load imbalance issues.

The reference versions for both apps are the pure MPI and the OpenMP+MPI versions available in the public repositories of each of them. Starting from these versions, we developed two new versions using regular tasks and worksharing tasks. We would like to clarify that the version using regular tasks uses OmpSs-2, as does the worksharing tasks version. Also, both of them make use of TAMPI, while the OpenMP+MPI and the pure MPI versions do not.

Next we will detail the methodology followed in the evaluation. After that, we present, analyze and discuss the results from the two mini-apps.

Methodology

As already introduced, our evaluation involves four different versions. The names that appear in the legends of figures are as follows:

- **MPI.** The version using only MPI.
- **OMP.** The version using OpenMP worksharing loops and MPI.
- **T.** The version using regular tasks of OmpSs-2 and MPI. It includes TAMPI.
- **TF.** The version using worksharing tasks of OmpSs-2 and MPI. It includes TAMPI.

For each of the applications, we perform a granularity analysis in a single NUMA node. This is because in the strong and weak scaling experiments we use one MPI process per NUMA node. The objective is to determine the best granularity to be used in the weak and strong scaling scenarios.

It may happen that a single granularity does not fit all the tasks. We support our analysis with execution traces. In the traces, we show the weights of each task so that it is possible to see if granularities are adequate for some tasks but not for others. When this happens, we define a reference granularity and then we apply multiplication factors for those tasks that need it. For instance, suppose there are two tasks: task A and task B. The granularity is 10 iterations. This granularity is good for task A, but it is too fine for task B and we spend as much time creating the task as executing it. Therefore, we apply a multiplication factor of 2 for task B, so its granularity is now 20 iterations. The points of the granularity analysis will always refer to the reference granularity.

In addition, we motivate the decision of using worksharing tasks rather than regular tasks using execution traces. For obvious reasons, task granularity analysis does not apply to the MPI version. The OMP version uses only worksharing loops with static scheduling, so the granularity analysis does not affect it either.

Once the optimal granularity is determined, we move forward to our second analysis: scalability analysis in a distributed environment. This analysis incorporates two scenarios: weak scaling and strong scaling. Weak scaling starts from 1 node and a given problem size. At each new point, the number of nodes is increased and so is the problem size. Strong scaling also starts from 1 node and a given problem size. However, in this scenario, the problem size is fixed for all the points, while the number of nodes is increased at each new point. As a last remark,

in this experiment, we use one MPI rank per core for the pure MPI version, and one MPI rank per NUMA node (to favour data locality) for the hybrid versions.

Regarding the execution environment, all the experiments were carried out on Marenostrom 4. A node of Marenostrom 4 is composed of 2 sockets Intel Xeon Platinum 8160 2.1GHz 24-core and 96GB of main memory [21]. Regarding the software, we used the Mercurium compiler [11] (v2.3.0), the Nanos6 runtime [12] (2020-05-15), the gcc and gfortran compilers (v7.2.0), and the Intel compilers (v17.0.4). Regarding MPI, we used the Intel implementation (v17.0.4). We also use the Intel implementation of OpenMP. Finally, we used Extrae [10] (v3.7.1) to obtain the execution traces, and Paraver [16] to visualize them.

Finally, we would like to highlight that each of the results is an average of 5 executions. We did not observe any significant variation between different executions (standard deviation <5%), so we think 5 executions is sufficient.

LULESH

LULESH is a quite big and complex mini-app. It contains two different main phases, one devoted to perform operations over elements, and the other one devoted to perform operations over nodes. There are several kernels, with very different computational costs. This application has a high degree of parallelism across all the execution, if correctly annotated with data dependences. Also, by design, it presents load imbalance problems. These two facts, that can be seen in Figure 4.16, make LULESH a very suitable application for tasking.

Figure 4.16 shows an execution trace of LULESH. The execution trace shows what is being executed in each of the threads (y-axis) over time (x-axis). The white color means that no task is being executed, and each of the other colors represent a different task type. In this trace, we show a single iteration of the main loop: from the red tasks to the next red tasks, which are already part of the next iteration. The cyan tasks are the last tasks belonging to the first phase, and the purple tasks mark the start of the second phase. The arrows on the top of the trace show approximately the duration of each phase.

When using MPI, there are four type of communications in the application. The position of each of the them is approximately indicated in Figure 4.16. At the beginning of an iteration, there is a *MPI_Allreduce*. Then there are two more point-to-point communications before the small orange tasks, and after the cyan tasks. Finally, the fourth and last communication, also point-to-point, can be found in the white space between the green and the dark orange tasks.

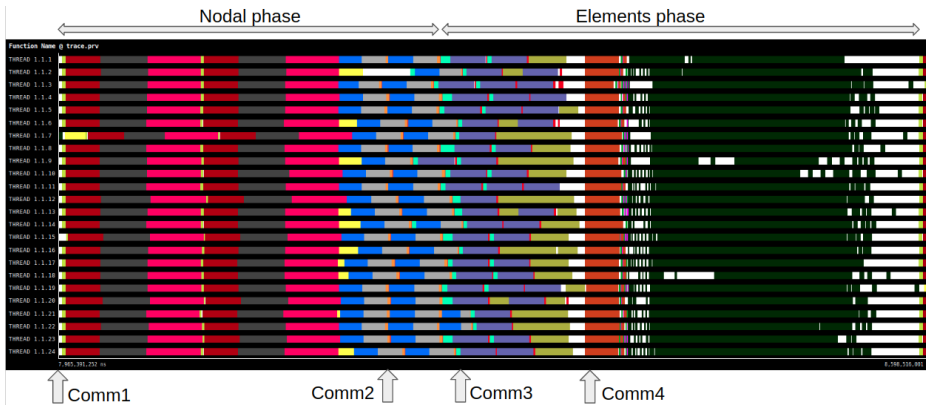


Figure 4.16: Execution trace of the LULESH T version using a single granularity for all the tasks with a big problem size on 24 cores (1 NUMA socket)

The OpenMP+MPI implementation uses worksharing loops (i.e. `parallel for`). Usually, there is a `parallel for` per loop. In some cases, the developers use the `nowait` clause that eliminates the implicit barrier at the end of a worksharing loop. As a side effect, this means that static scheduling is mandatory. Given that LULESH presents load imbalance issues, the use of worksharing with static scheduling may hinder performance. In contrast, tasks has an inherent ability to deal with load imbalance.

Moving from OpenMP worksharing to tasks requires replacing each of the work-sharing loops by several tasks. The most important consideration is to avoid the use of `taskwait`, and rely on the data dependences to achieve lightweight synchronization. Once we have the implementation with tasks using data dependences, we need to determine how many different granularities there should be in the mini-app. For that purpose, we use execution traces.

The trace of Figure 4.16 is an execution using a large problem size, which displays good behavior throughout. Figure 4.17 is the same trace but using a smaller problem size. In this case, to have the same number of tasks, they must be more fine-grained. In fact, they are too fine-grained, and the task management overhead becomes too much. Specifically, in Figure 4.17 it is possible to see the white color dominating the trace, meaning that most of the time the cores are idle. Looking at the complete trace, it is possible to see the producer core (core 0, purple color) creating tasks during the entire execution. The consumers cores execute tasks faster

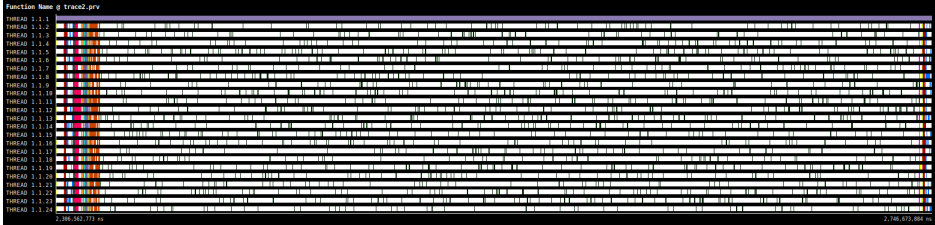


Figure 4.17: Execution trace of the LULESH T version using a single granularity for all the tasks with a small problem size on 24 cores (1 NUMA socket)

than the producer core can create them.


















The scenario with a small problem size is important considering the strong scaling experiment. Given that we set a fixed problem size and then we increase the number of processors, at each new step the problem size per process will decrease, ending up in scenarios such as the one in Figure 4.17 or with even smaller problem sizes.

One possible solution to this problem is using different granularities for different task types. Not all the task types have the same weight, even with the same granularity. Thus, for tasks with low weights, it may be better to use bigger granularities. Around a hundred microseconds is the minimum time we consider a task must last to be worth paying the management costs. Table 4.1 presents the average time a task of a given type requires to complete. In the table, each of the tasks types has a small square with the same color the task has in the execution trace of Figure 4.17. There are several task types whose average time that does not reach this threshold. Thus, we should change the granularity of several task types.

We increased the granularity of each task type to reach the 100 microsecond threshold. It is important to highlight that the increment of the granularity is not always directly proportional to the increment of time. That gave us the multiplication factors shown in Table 4.1 for each of the task types.

When we implement this, there is a $\sim 12x$ speedup in the execution time. Figure 4.18 shows how an iteration looks like using the listed factors. It is possible to see that the duration of the iteration is ~ 440 ms in Figure 4.17 and ~ 36 ms in Figure 4.18. Even so, there is still a lot of white color, meaning cores are running no tasks. In some regions of the execution trace this may be caused by dependences: tasks are waiting for their predecessors to finish so they can start. However, there are other regions where we know for sure there are no dependences between tasks,

Table 4.1: Average task time, and multiplication factor, for each of the task types of LULESH mini-app using a small problem size

Task type	Average	Factor
CalcLagrangeElements 	22.25 us	16
CalcVelocityAndPositionForNodes 	46.07 us	4
check_eosv_vc 	13.10 us	16
InitStressTermsForElems 	13.98 us	24
UpdateVolumeForElems 	5.12 us	48
CalcHourglassControlForElems 	361.32 us	1
CalcTimeConstraintsForElems 	81.99 us	2
CalcFBHourglassForceForElems 	615.50 us	1
EvalEOSForElems 	24.98 us	16
CalcFBH_collect 	175.45 us	1
CalcMonotonicQRegionForElems 	16.50 us	48
IntegrateStressTermsForElems 	386.66 us	1
CalcMonotonicQGradientsForElems 	212.95 us	1
IntegrateStressTermsForElems_collect 	93.06 us	2
CalcKinematicsForElems 	402.48 us	1
ApplyAccelerationBoundaryConditionsForNodes 	3.86 us	96
CalcAccelerationForNodes 	21.69 us	16

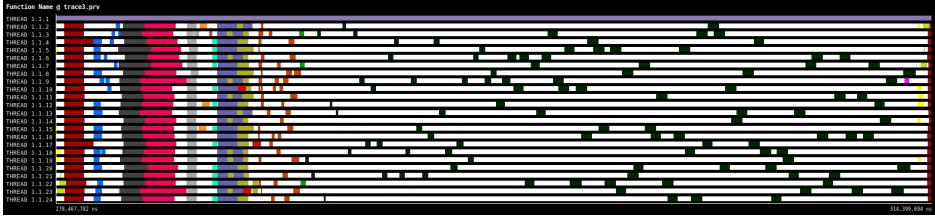


Figure 4.18: Execution trace of the LULESH T version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket)

for instance in the big region with dark green tasks. Thus, we conclude that the problem is that the thread creating tasks is not creating them fast enough. In other words, we are creating too many tasks, so again we need to increase the granularity.

Figure 4.19 shows the granularity chart using a problem size of 50 elements per dimension. In the x-axis we show the number of created tasks/worksharing tasks per each of the different task types. The y-axis shows the figure of merit (FOM) of the mini-app. The T version reaches the peak performance with a reference granularity of 48 tasks per type, and the TF version reaches the peak performance using a reference granularity of 24 worksharing tasks per type. Considering that a NUMA node of the machine has 24 cores, there is 1 worksharing task per core in the TF version, and 2 tasks per core in the T version. A significant performance difference exists between using regular tasks and worksharing tasks. Regarding the implementation, the difference between the two versions is using a single worksharing task, rather than multiple tasks, in several portions of code. All the methods listed previously, where we apply a multiplication factor, use a single worksharing task, except *CalcVelocityAndPositionForNodes*, *CalcLagrangeElements*, *InitStressTermsForElems* and *CalcMonotonicQRegionForElems*. With such a simple change, the peak performance shows a speedup of $\sim 2.1x$.

The main reason for such a big impact in performance is the significant reduction in terms of the number of created tasks. Fewer tasks means fewer overheads, and, usually, less parallelism. Notwithstanding, thanks to the use of worksharing tasks, creating fewer tasks does not affect parallelism. The number of created tasks is reduced to such an extent that the main thread is able to create enough work for ten iterations in the time taken to execute five iterations. In other words, for the regular tasks version, the main thread keeps creating tasks during the whole execution, while using worksharing tasks it requires only $\sim 50\%$ of the execution

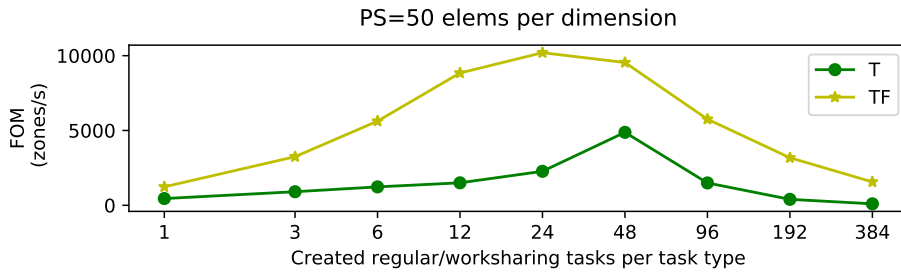


Figure 4.19: Granularity chart of LULESH

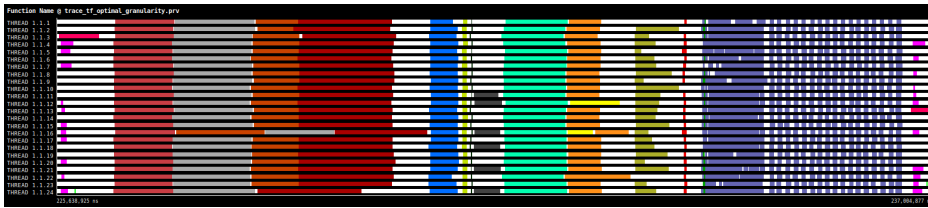


Figure 4.20: Execution trace of the LULESH TF version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket)

time. Additionally, this means that the core devoted completely to creating tasks in the regular tasks version, can contribute to running some of the tasks in the worksharing tasks version.

Figure 4.20 shows an iteration of the TF version. In this case, the granularity used is the optimal one based on Figure 4.19. We can see there that an iteration takes ~ 12 ms to complete. Even so, there are some problems that should be addressed. In particular, there are some regions suffering a lack of parallelism and others suffering load imbalance. Considering that the granularity used causes the creation of only 1 task per type per core, and that the introduction of multiplication factors reduces the number of created tasks, this makes sense. We can see that all the regions suffering a lack of parallelism or load imbalance are using regular tasks. Thus, a logical step forward is to replace those regular tasks by worksharing tasks. By doing so, each of the worksharing tasks can be executed by several cores, thus preventing lack of parallelism. Also, given that worksharing tasks allow threads to move forward when there is no remaining work, load imbalance should be also improved.

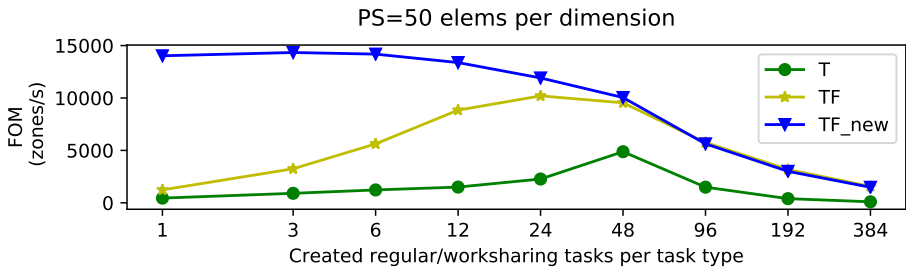


Figure 4.21: Updated granularity chart of LULESH

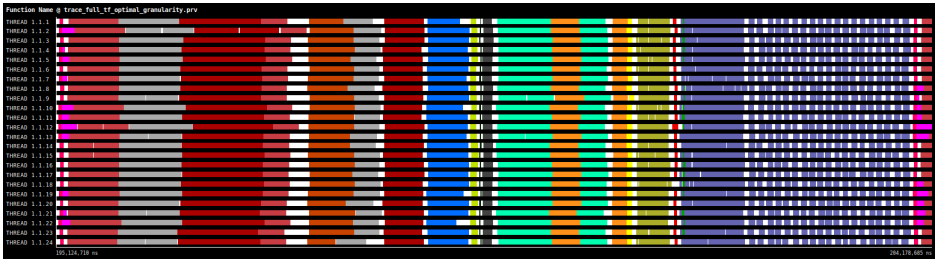


Figure 4.22: Execution trace of the updated LULESH TF version using multiple granularities for different task types on 24 cores (1 NUMA socket)

As a consequence of replacing all the regular tasks by worksharing tasks, it is likely that the optimal granularity is lower than the one used previously. Thus, we repeated the granularity analysis keeping the same problem size (50 elements per dimension), which is shown in Figure 4.21. The peak performance in this new version occurs when using a reference granularity of three worksharing tasks per task type. This is because the fewer tasks created, the fewer overheads are introduced. Worksharing tasks enables us to create a lower amount of tasks per task type, while keeping enough work to feed all the cores.

Figure 4.22 shows an iteration of the updated TF version (i.e. using worksharing tasks wherever possible). The trace was obtained using the optimal granularity: 3 worksharing task per task type. Now an iteration only takes ~ 9 ms to complete. So, to sum up, we started with a version where an iteration required ~ 440 ms to complete, and ended up with a version where an iteration requires only ~ 9 ms to complete.

After detailing the modifications done in the implementation, and selecting an

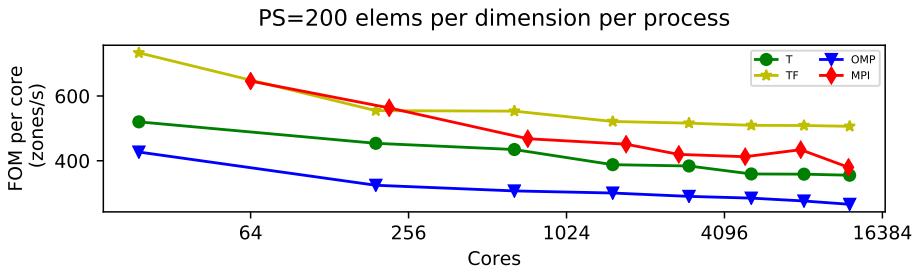


Figure 4.23: Results of the LULESH weak scaling experiment

adequate granularity, we are able to move forward and perform the experiments in a distributed environment. Figure 4.23 shows the results of the weak scaling experiment, and Figure 4.24 shows the results of the strong scaling experiment. This application restricts the number of MPI processes used to be a cube of an integer number. That introduces difficulties in the comparison between the MPI version and the rest because the total number of cores used does not match. For this reason, we decided to use the FOM/core instead of the raw FOM metric.

Figure 4.23 shows in the x-axis the number of total cores used and in the y-axis the FOM per core of the application. For this experiment, we used a problem size of 200 elements per dimension per process. The OMP version is the worst across all the different scenarios, while the TF is the best across all the different scenarios. The MPI version begins close to the TF version but then it falls. Regarding the T version, it is in between the OMP and the MPI version until the last point where it obtains almost the same performance than the MPI version. Overall TF version is able to reach speedups of up to $\sim 1.4x$, $\sim 1.9x$, and $\sim 1.3x$ compared to the T, OMP, and MPI versions, respectively.

Figure 4.24 shows in the x-axis the number of total cores used and in the y-axis the FOM per core of the application. For this experiment, we used a problem size of 300 elements per dimension. The TF and the MPI versions perform very similarly across all the scenarios. Similarly, the T and the OMP versions behave very much alike. However, there is a significant difference between the two groups, reaching up to $\sim 2.8x$ speedup.

The reason for the performance improvement of the TF version compared to the T version is that the number of tasks is drastically reduced. Consequently, there is a drastic reduction of overhead. In addition, by creating fewer tasks, the creator

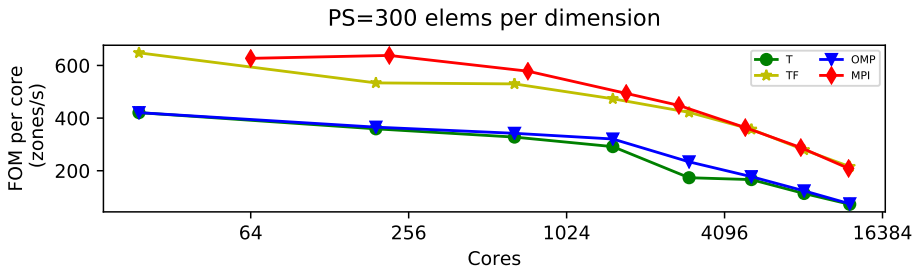


Figure 4.24: Results of the LULESH strong scaling experiment

can finish the creation earlier. Therefore, the cores rarely have to wait because the creator cannot create fast enough. Finally, this fact also means that the creator can start running tasks after finishing the creation.

Compared to the OMP version, the tasking versions introduces two key advantages: (1) the flexibility given by the data-flow execution model, and (2) the overlapping of computation and communication phases given by the TAMPI library. The OMP version is implemented using worksharing. At the end of each worksharing loop there is a barrier, where all the threads must wait until all of them have finished. This rigidity in the synchronization may introduce a significant performance penalty, especially if there is load imbalance, which is the case here. Also, the communication in the OMP version is always done outside parallel regions. Thus, when the data is required, threads in this version are idle while waiting for the communication to complete. In contrast, the T and the TF versions are able to keep progressing running other ready tasks.

Compared to the MPI version, the tasking versions introduce the key advantage of the overlapping of computation and communication phases. Figures 4.25 and 4.26 show the time performing computations and communications respectively. These figures evidence the amount of time that MPI version wastes in communications. In contrast, the tasking versions can keep progressing thanks to TAMPI. However, for the T version this is not enough due to the task management overheads. The TF version is able to reduce these overheads, and so, is able to outperform the MPI version consistently in the weak scaling experiment, reaching up to $\sim 1.3x$ speedup in the scenario with more cores. Regarding the strong scaling experiment, the TF version is competitive with the pure MPI version, and even obtains a slight speedup of 1.03x.

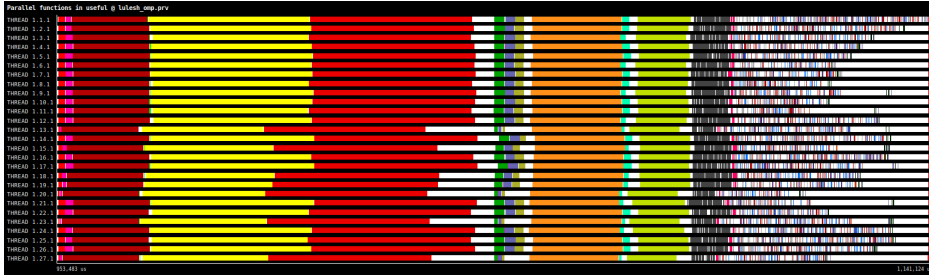


Figure 4.25: Execution trace of the LULESH MPI version showing the time performing computations

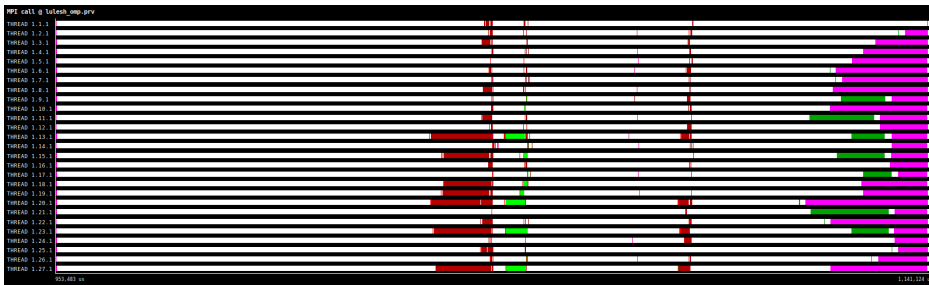


Figure 4.26: Execution trace of the LULESH MPI version showing the time performing communications

HPCCG

HPCCG is a simpler application compared to LULESH. It contains a single phase relying on three different kernels: `ddot`, `waxpby` and `sparseMV`. This application is very well suited for OpenMP worksharing loops, given its fork-join pattern shown in Figure 4.27. For each iteration, there is a `ddot` kernel that can run in parallel. After that, it computes the residual and the alpha, using at most two cores. This closes the parallelism because alpha is required by the following tasks. Then, parallelism is open again to run the `waxpby`, `sparseMV`, and `ddot` kernels. Following this comes the compute of beta, which closes parallelism again because the following tasks need beta. Finally, the `waxpby` kernels can run in parallel again.

When using MPI, there are three different communications. There is an `MPI_Allreduce` after each of the two `ddot` kernels, and one point to point communication before the `sparseMV`. The result of the reductions is required to compute beta, `normr`, and alpha. Thus, the overlapping of computation and communication is only possible in the point to point communication.

This structure is not well suited for tasks because it forces all the cores to wait twice per iteration, as if there was a `taskwait`. As a consequence, there is not much benefit from using data dependences because there are implicit barriers imposed by the application structure.

The reference OpenMP+MPI implementation simply uses a `parallel` for in each of the three kernels. In the tasking version we replaced each of the `parallel` for by a set of tasks with the required data dependences. Even in an application with a fork-join pattern is important to avoid the use of `taskwait`s. A `taskwait` implies that no more tasks will be created until all the already created tasks finish. If we use data dependences, the tasks are already created, and so, as soon as the data is ready, they can run. By using `taskwait`, when the data is ready (all the previous tasks finished), it starts creating tasks (one by one) again, increasing the overall waiting time.

We again start our analysis by determining how many different granularities there should be in the mini-app. For that purpose, we use the execution trace shown in Figure 4.28. The execution trace shows what is being executed in each of the threads (y-axis) over time (x-axis). The white color means no task is being executed and each of the other colors represent a different task type. In this trace, we show a single iteration of the main loop: from the blue and green tasks to the next blue and green tasks. Actually, in this application, the iterations are overlapped. The green tasks and some of the blue tasks of the beginning belong to the previous

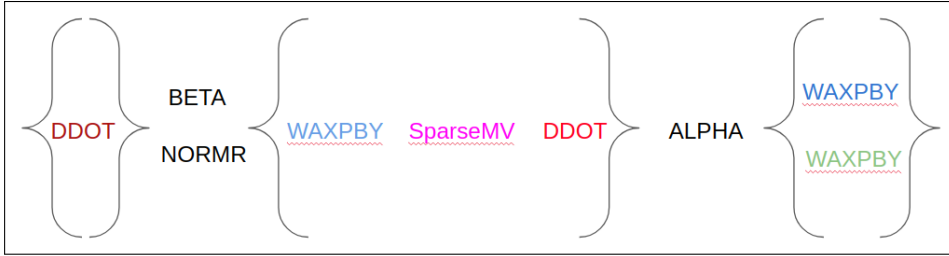


Figure 4.27: Structure of the HPCCG application

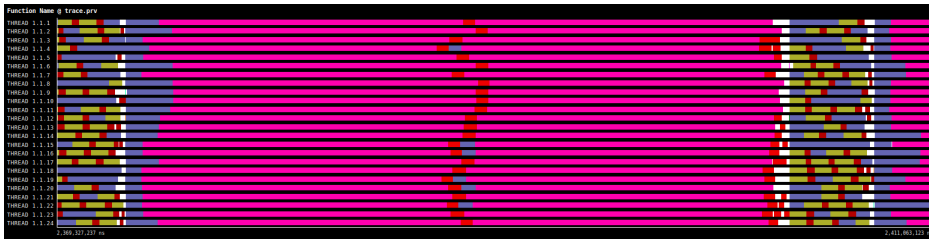


Figure 4.28: Execution trace of the HPCCG T version using a single granularity for all the tasks with a big problem size on 24 cores (1 NUMA socket)

iteration, and the interleaved dark red tasks and some other blue tasks belong to the iteration we show. At the end, some of the blue tasks overlapped with the pink tasks belong to the iteration we show, while the pink tasks, some other blue tasks and the dark red tasks belong to the next iteration.

The trace of Figure 4.28 shows very good behavior. The execution was performed using a big problem size. In contrast, Figure 4.29 was performed using a much smaller problem size. It also shows a single iteration with the overlapping with the previous and following iterations. In this case, to have the same amount of tasks, they must be more fine-grained. In fact, they are too fine-grained, and the task management overhead becomes too much. Specifically, in Figure 4.29 it is possible to see the white color dominating the trace, meaning that most of the time the cores are idle. Looking at the complete trace, it is possible to see core 0 creating tasks during the entire execution. The other cores execute tasks faster than core 0 can create them, and that is the reason to see so much white color in the trace.

The scenario with a small problem size is important, considering the strong scaling experiment. Given that we set a fixed problem size and then we increase the

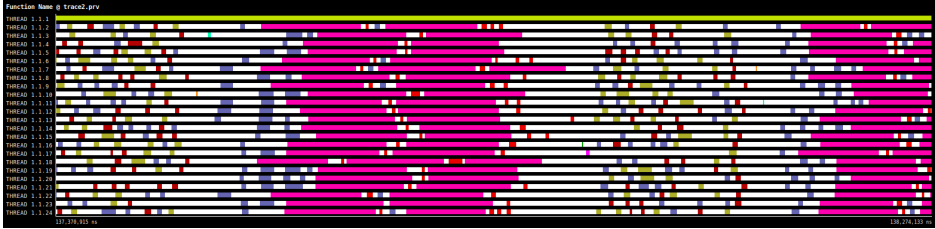


Figure 4.29: Execution trace of the HPCCG T version using a single granularity for all the tasks with a small problem size on 24 cores (1 NUMA socket)

Table 4.2: Average task time for each of the task types of HPCCG mini-app using a small problem size

Task type	Average	Factor	Task type	Average	Factor
HPCSparseMV ■	79.25 us	1	ddot_xx ■	4.45 us	4
ddot_xy ■	3.73 us	4	waxpby_beta ■	6.25 us	4
waxpby_negative_beta ■	6.48 us	4			

number of processors, at each new step the problem size per process will decrease, ending up in scenarios such as the one in Figure 4.29 or with even smaller problem sizes.

We want to determine how many different granularities we need in this mini-app. Looking at Figure 4.29 we see a big difference between the pink tasks and the rest. Table 4.2 presents the average time a task of a given type requires to complete. Also, each task type has a small square with the same color it has in the trace of Figure 4.29. Recall that around 100 microseconds is the minimum time we consider a task must last to be worth paying the management costs. None of the task types reaches the given threshold. Accordingly, we should use a higher granularity. However, then, lack of parallelism may appear. Apart from that, *HPCSparseMV* takes much more time to complete than the rest. Consequently, we apply a multiplication factor of 4 to all the task types except *HPCSparseMV* as we show in Table 4.2. We do this to balance the different task times.

After that, there is a $\sim 1.2x$ speedup in the execution time. Figure 4.30 shows what an iteration looks like using the listed factors. Still, there is a lot of white color, meaning cores are running no tasks. This is caused by lack of parallelism: there are not enough tasks to feed all the cores. In consequence, we need to decrease

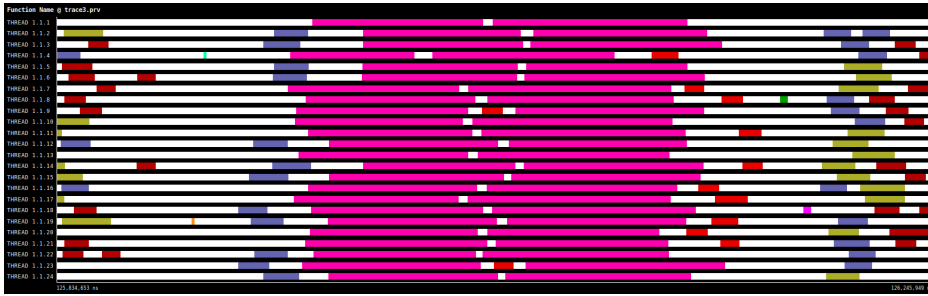


Figure 4.30: Execution trace of the LULESH T version using multiple granularities for different task types with a small problem size on 24 cores (1 NUMA socket)

the granularity. However, if we do so, task management overheads may hinder performance. Worksharing tasks offer us the possibility of keeping this granularity, but not be affected by task management overheads, and increasing parallelism due to its internal partitioning of work.

Figure 4.31 shows the granularity chart using a problem size of 50 elements per dimension. In the x-axis we show the number of tasks/worksharing tasks created per each of the different task types. The y-axis shows the figure of merit (FOM) of the mini-app. The T version reaches its peak performance using a reference granularity of 48 tasks per type, and the TF version reaches the peak performance using a reference granularity of 1 worksharing task per type. Regarding the implementation, we simply replaced all the regular tasks by worksharing tasks. Figure 4.31 shows an interesting point. On the right part, when more fine-grained tasks are created, regular tasks outperform worksharing tasks. The problem is that worksharing tasks partition the loop iteration space into as many chunks as cores. In this case, the tasks are already fine-grained, and then they are partitioned even more. As a result, cores end up running really small chunks (< 5 us). Worksharing tasks contain an internal mechanism of synchronization which is very lightweight, but not enough to perform well with such small chunks.

Nevertheless, worksharing tasks offer a mechanism to mitigate this effect. A user can set the minimum chunksize. We have selected a minimum chunksize based on the point where the performance of the TF version starts to be worse than the performance of the T version in Figure 4.31. We repeated the granularity analysis, keeping the same problem size (50 elements per dimension), including this new version. The results of the new analysis are shown in Figure 4.32. After setting the

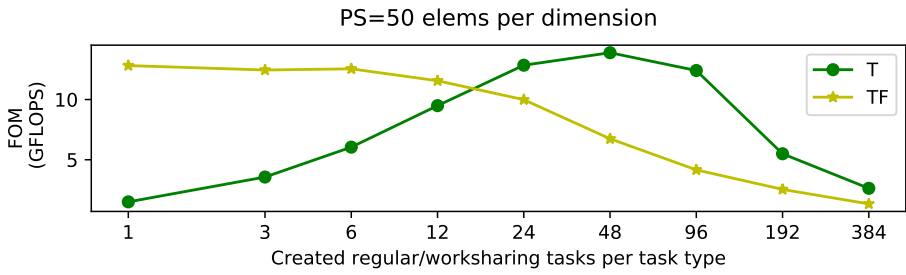


Figure 4.31: Granularity chart of HPCCG

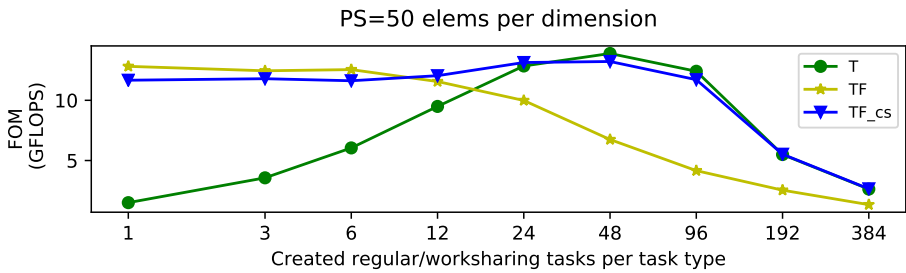


Figure 4.32: Granularity chart of HPCCG

minimum chunksize, both versions behave very similarly in the right-most part, while the TF version keeps very good performance in the left-most part.

In this mini-app, unlike LULESH, there is no big difference in the peak performance between the T version and the TF version using the optimal granularity. In HPCCG, there are not so many tasks and the creator can create tasks rapidly enough for the rest, which was the main issue in the LULESH. Figure 4.33 shows an iteration of the TF version using the optimal granularity. There, it can be seen the main problem that HPCCG presents. In the left part, between the dark red tasks and the blue tasks, there are two very small tasks. This also happens in the right part between the red tasks and the blue tasks. Those small tasks require the data computed by all the previous ones, and the following tasks require the data computed by those two small tasks. Basically, those tasks close the parallelism and open it again, following a fork-join pattern. Apart from that, there are still some other regions with cores running no tasks. The cause is the dependences between different tasks. The runtime requires some time to release the dependences of a

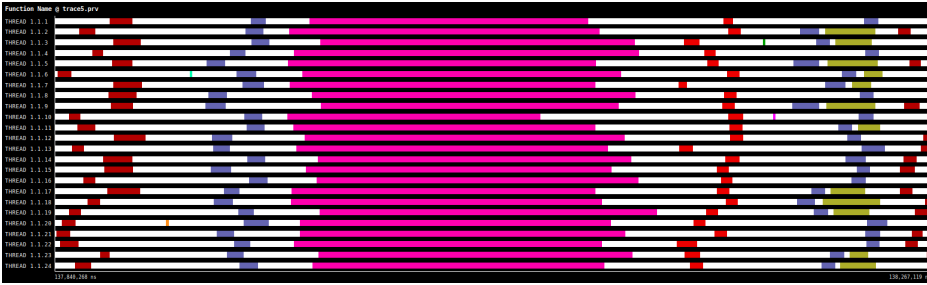


Figure 4.33: Execution trace of the HPCCG TF version using a single granularity for all the tasks on 24 cores (1 NUMA socket)

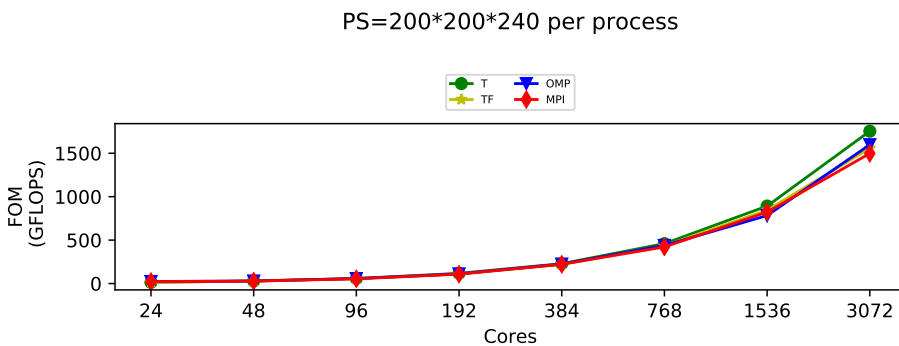


Figure 4.34: Results of the HPCCG weak scaling experiment

task and schedule the new ready tasks. Again, with higher granularities, this effect would be mitigated, but the low problem size prevents us from increasing granularity.

After detailing the modifications done in the implementation, and selecting an adequate granularity, we are able to move forward and perform the experiments in a distributed environment. Figure 4.34 shows the results of the weak scaling experiment using a problem size of $200 \times 200 \times 240$ per process, and Figure 4.35 shows the results of the strong scaling experiment using a problem size of $200 \times 200 \times 3072$.

In the weak scaling scenario shown in Figure 4.34 all the versions perform very similarly. The T version stands out over the rest, followed by the TF version. The reason for this is the load imbalance introduced by the sparseMV kernel. Different regions of the sparse matrix have different number of non-zeros. Tasks deal better

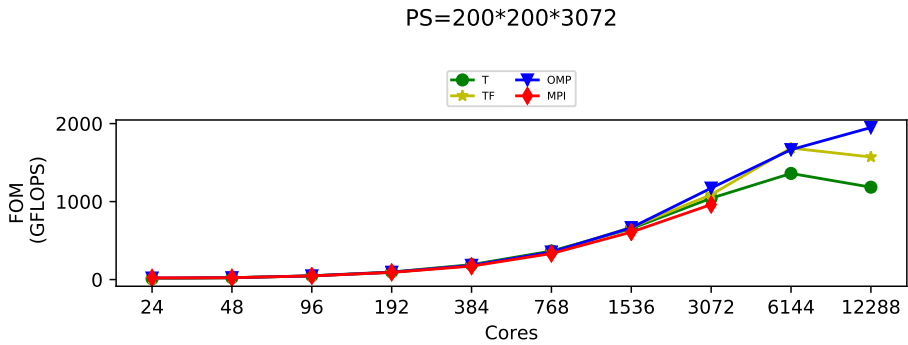


Figure 4.35: Results of the HPCCG strong scaling experiment

with load imbalance, and are able to get computation and communication overlapping. Overall, it gives the tasking versions a small boost in performance of up to 1.14x and 1.08x for the T and the TF versions, respectively. As a final remark, we were not able to scale to more cores in this scenario because of some overflow problems in the application indexes.

In the strong scaling scenario, all the versions perform very similarly until 3072 cores. Then, the T version cannot scale as much as OMP and TF. Regarding the MPI version, it cannot scale further because the problem cannot be split in more MPI processes. The TF version is competitive until 6144 cores, where it stops scaling, while the OMP version keeps scaling up until the end (12288 cores). With such a number of cores, the problem size per process becomes very small and most of it fits in cache. The OMP version is able to exploit data locality thanks to the static scheduling of the worksharing loops. However, the TF cannot do so because of two reasons: worksharing tasks do not guarantee that the same core executes always the same iterations, and the structures of the runtime pollute the cache.

All in all, the TF version is able to keep very competitive performance until a large number of cores compared to OpenMP, in an application perfectly suited for the fork-join execution. Compared to the T version, this experiment evidences the problems of task granularity that regular tasks suffer, and that worksharing tasks can mitigate, obtaining up to 1.33x speedup.

4.6.3 Taskloop with Dependences

In this section, we wish to demonstrate that the taskloop with dependences provides an enhancement in programmability when using tasks, while introducing no significant overhead compared to a manual equivalent implementation. For that purpose, our evaluation will focus in both programmability and performance.

Regarding programmability, we used several different metrics to compare the different implementations: Source Lines of Code (SLOC), Development Estimate Effort (DEE) and Cyclomatic Complexity (CC). It is important to highlight that for the SLOC metric, we only consider the code related to the parallelization. In terms of performance, we compare the different implementations to demonstrate that using the taskloop construct do not add any significant overhead.

Environment

The experiments were carried out on the Marenstrum 4 supercomputer. It is composed of nodes with 2 sockets Intel Xeon Platinum 8160 2.1GHz 24-core and 96GB of main memory.

Regarding the software, we used the Mercurium compiler (v2.3.0), the Nanos6 runtime library, the gcc and gfortran compilers (v7.2.0), and the Intel compilers (v17.0.4).

Methodology

As previously introduced, we focus our evaluation in two different aspects: performance and programmability. Our experiments will use two different versions of each application/benchmark:

- **T.** Version using regular tasks. It requires manual blocking.
- **TL.** Version using the taskloop construct with dependences.

Regarding performance, for each of the benchmarks/applications, we select two different problem sizes, one small-medium size, and one big size. For each of the problem sizes, we try several block sizes to show that the differences between the T and the TL are small or even non-existent in several different scenarios.

All the experiments ran using the interleaving policy offered by the numactl command, spreading the data evenly across all the available NUMA nodes, in order to minimize the NUMA effect.

The results shown in the figures are averages of five different executions. We decided to use only five executions because the variability across different executions was very small.

Related to programmability, we count the SLOC required to parallelize the baseline code for each of the versions, and use the SLOCCount tool and the Lizard tool to retrieve the DEE and CC respectively.

Performance Evaluation

In this section, we evaluate several applications/benchmarks to demonstrate that the use of the `taskloop` construct does not introduce overhead compared to a manual alternative. All the figures show the Figure of Merit (FOM) of the application in the y-axis, and different task granularities in the x-axis. All of them have four series: one using the T version with a small-medium problem size, one using the TL version with a small-medium problem size, one using the T version with a big problem size, and one using the TL version with a big problem size. We would like to highlight that the T versions use a naive approach where a single core creates all the tasks.

Figure 4.36 shows the results of the dot product benchmark. In this case, we repeat the dotprod kernel a given number of iterations to make the execution longer. For both problem sizes, the TL version performs better than the T version in the small task sizes. The T version has only a single core creating tasks. When the granularity is small, a single creator cannot create rapidly enough to feed all the cores. In contrast, the TL version may have several cores creating tasks, speeding up the creation, and increasing the overall performance. The TL version may have several cores producing tasks because each iteration of the kernel is a `taskloop` instance, that can be running in different cores concurrently, while the T version has a single core creating all the iterations sequentially. Finally, from $TS=8192$, all the versions perform very similarly.

Figure 4.37 shows the results of the Heat benchmark, where we can see no differences between the T and the TL versions. In this benchmark, a single core creating tasks is enough because the tasks are more time-consuming than in the previous benchmark even in the smallest granularities.

Figure 4.38 shows the results of the LULESH application. Note that this figure reports the number of created tasks rather than the task size in the x-axis. The number of created tasks is inversely proportional to the task size: the more created tasks, the smaller task size. In this figure, we can see small differences between the

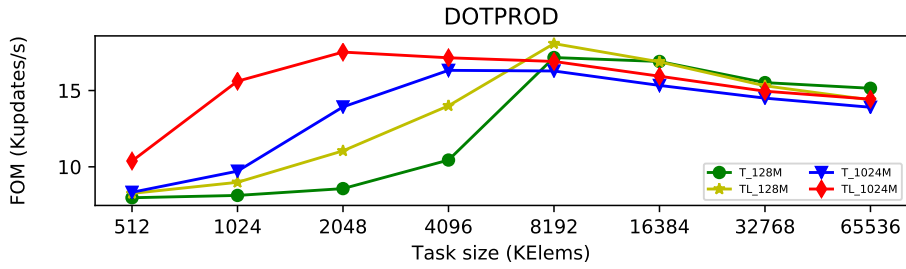


Figure 4.36: Evaluation of taskloop using DOTPROD benchmark

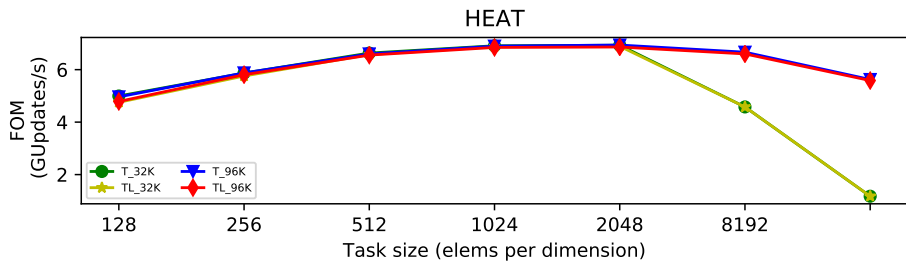


Figure 4.37: Evaluation of taskloop using HEAT benchmark

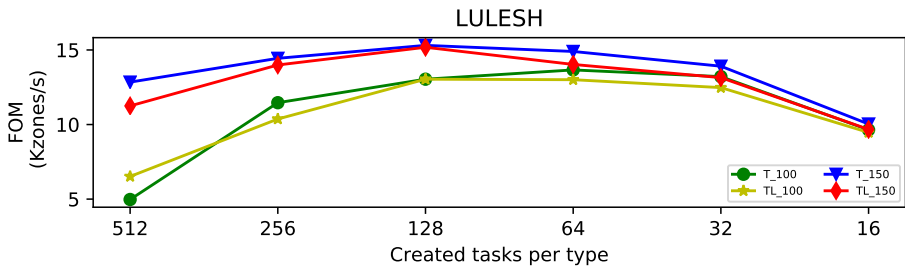


Figure 4.38: Evaluation of taskloop using LULESH application

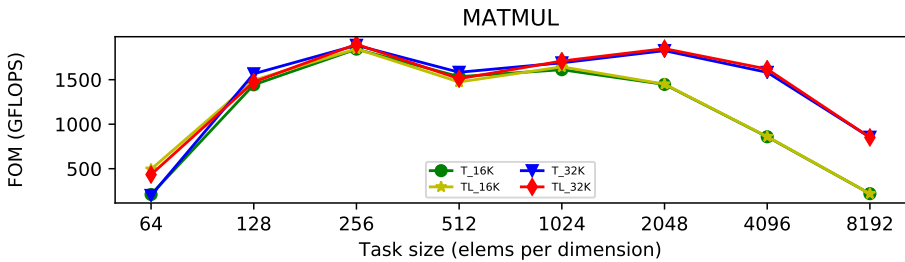


Figure 4.39: Evaluation of taskloop using MATMUL benchmark

T and TL versions for both problem sizes for the same reason than in the previous benchmark.

Figure 4.39 shows the results of the Matrix Multiplication benchmark. For this benchmark, again, all the versions perform very similarly except in the smallest granularity, where the TL versions outperform the T versions for both problem sizes. Like with the Dot product benchmark, the TL version has several taskloops than can run in parallel, and so, create tasks in parallel. In contrast, the T version has a single core creating all the tasks, and it is not able to create rapidly enough to feed all the cores.

Figure 4.40 shows the results of the Multisaxpy benchmark. Multisaxpy benchmark is a saxpy kernel repeated a given number of iterations. As with several of the previous benchmarks, there are almost no differences between the different versions.

Figure 4.41 shows the results of the N-body benchmark. For this benchmark, we see again a difference in the smallest granularity, where the TL version outper-

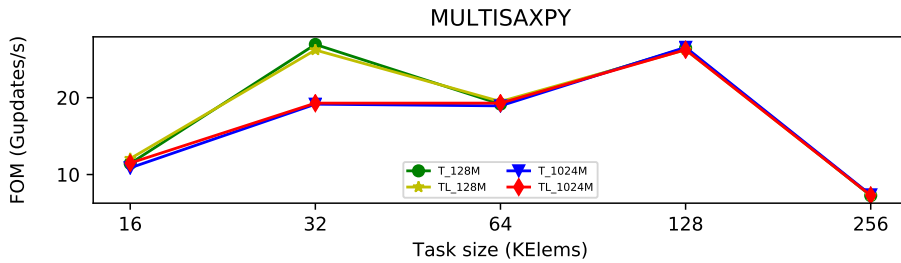


Figure 4.40: Evaluation of taskloop using MULTISAXPY benchmark

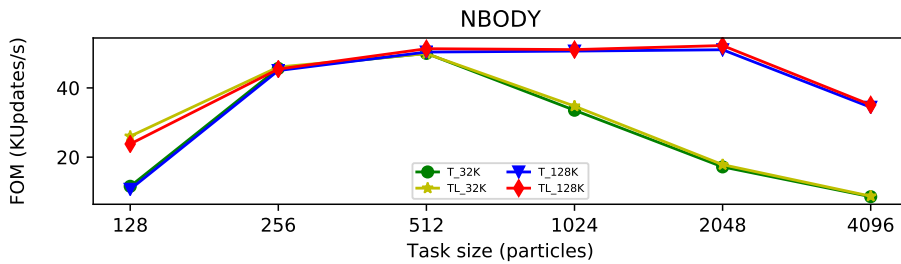


Figure 4.41: Evaluation of taskloop using NBODY benchmark

form the T version for both problem sizes. Like previously, this is because there are several taskloops that can be creating tasks concurrently in the TL version, while there is a single core creating tasks sequentially in the T version, and it is not quick enough to feed all the cores.

Figure 4.42 shows the results of the Stream benchmark. In this benchmark, there are some differences between the T and TL versions. Firstly, in the smallest granularity, the TL version outperforms the T version in both problem sizes. Like in some previous benchmarks, this is because the TL version has multiple taskloops that can create concurrently rather than a single one, and speeding up the creation improves the overall performance. Then, when TS=64, for the small problem size the T version outperforms the TL version, and the other way around for the big problem size. Our runtime system has an immediate successor mechanism to exploit data locality between successor tasks. In this case, this mechanism is making the difference. We repeated the experiment with no immediate successor, and the results for both versions were very similar. For the big problem size, the TL version

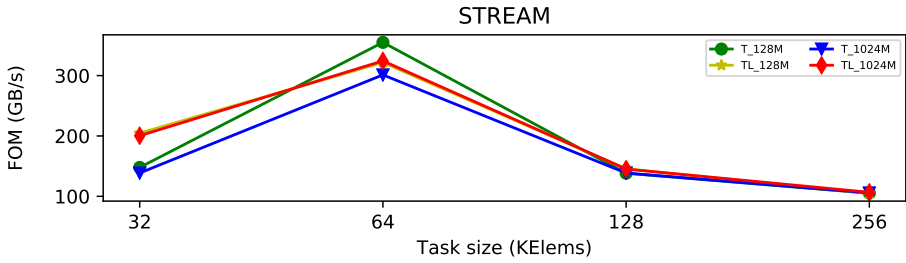


Figure 4.42: Evaluation of taskloop using STREAM benchmark

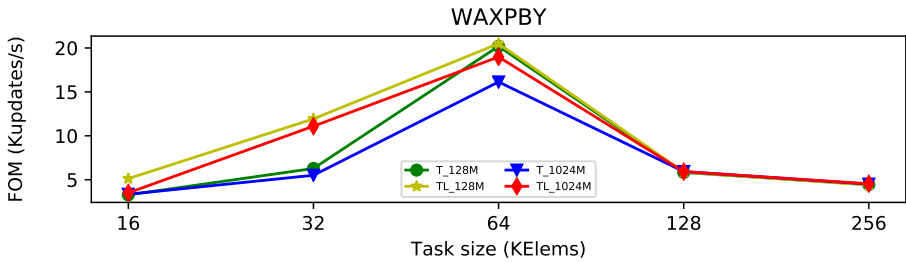


Figure 4.43: Evaluation of taskloop using WAXPBY benchmark

is able to find more immediate successor tasks than the T version, and the other way around for the small problem size.

Finally, Figure 4.43 shows the results of the Waxpby benchmark. This benchmark computes the $w = \alpha * x + \beta * y$ equation a given number of iterations. The TL version outperforms the T version in several granularities for both problem sizes. For the smallest granularities, the key again is speeding up the creation of tasks. In this case, for the big problem size, the TL version also delivers better performance in the granularity obtaining peak performance. Like in the Stream benchmark, this is caused by the immediate successor mechanism. In this specific scenario, the TL version is able to find more immediate successor tasks than the T version.

Overall, we see that there are few differences between the T and TL versions, with the TL versions generally outperforming the T versions in fine granularities, thanks to the use of multiple creators. Thus, we can conclude that the TL is not only introducing very few overhead, but it is able to enhance performance in some specific scenarios.

Table 4.3: Programmability metrics to compare the use of the `taskloop` construct with the manual use of tasks

	T			TL		
	SLOC	DEE	CC	SLOC	DEE	CC
DOTPROD	7	0.07	5	3	0.05	2.5
HEAT	12	0.09	5	12	0.09	5
LULESH	503	6.44	8.2	437	6.32	7.8
MATMUL	13	0.15	5	13	0.15	5
MULTISAXPY	5	0.06	2.5	2	0.03	2
NBODY	10	0.29	2.6	10	0.29	2.6
STREAM	25	0.37	13.5	7	0.3	9.5
WAXPBY	10	0.16	4	4	0.14	2.2

Programmability Evaluation

Table 4.3 shows the different programmability metrics evaluated in this analysis for different benchmarks. The DEE is a metric that tries to estimate the effort that a developer must spend to write a given code. In this case, it is measured in *person-months*. The size of the code affects the DEE. The CC metric is higher when a code can take more different paths. For instance, adding an `if` increases the CC.

In Table 4.3, some benchmarks show no difference between the T version and the TL version. As previously explained, a frequent way of parallelize an application with tasks is using blocking, thereby converting a single loop into two loops, one to iterate over blocks, and one to iterate over the elements of each block. The `taskloop` construct prevents users from requiring this twofold loop structure in some cases, saving some lines of code. In the case of the Heat, Matmul, and Nbody benchmarks, the data layout in our implementations is blocked, so we cannot eliminate the twofold loop. Thus, there is no real improvement in programmability for these benchmarks. In contrast, the Stream benchmark and the LULESH app show improvements in all the different metrics, reaching up to a 3.57x reduction of code lines for the Stream benchmark.

4.7 Conclusions

In this chapter we propose two contributions to overcome limitations of the tasking model. On the one hand, focused on performance, a new concept called worksharing tasks that leverage the flexibility of tasks and the efficiency of worksharing techniques to exploit structured parallelism. Our proposal introduces the new `for` clause—`do` clause in Fortran—to the task construct. On the other hand, focused on productivity, we add support for data dependences to the `taskloop` construct, enabling users to fully parallelize loops with tasks using a single construct.

A worksharing task is like a regular task that encompasses a `for` loop. The key difference is that this `for` loop can be run by several workers using worksharing techniques that have been adapted to avoid any fork-join synchronization to preserve the fine-grained data-flow execution model of regular tasks.

In general, task-based programming models require at least one task per core to achieve the best performance. This fact provides an upper bound on the task granularity, which proportionally increases with the problem size but proportionally decreases with the number of cores. Thus, a small problem size combined with a large number of cores limits task granularity and impacts performance.

Using too fine-grained tasks, the overheads related to task management hinder performance; while using too coarse-grained tasks, the number of tasks is not enough to fully exploit all cores. The lower bound of task granularity that reaches peak performance is determined by the efficiency of the runtime system to handle tasks, while the upper bound of task granularity is determined by the problem size per core.

Worksharing tasks overcome the requirement of one task per core to achieve high resource utilization by allowing a small number of coarse-grained worksharing tasks that are partitioned into several fine-grained chunks. Worksharing tasks allow us to increase task granularity up to 64x without limiting the available parallelism. Hence, a small number of worksharing tasks can efficiently exploit a many-core processor.

Our evaluation shows that worksharing tasks not only outperform traditional tasks and worksharing techniques, but also advanced combinations of both techniques. Worksharing tasks get up to 9x speedup against the most performant OpenMP alternative in some scenarios. Moreover, the use of worksharing tasks increases the range of granularities that reach peak performance. Finally, our proposal does not add any coding complexity over the traditional task-based implementation.

The `taskloop` construct is a directive that distributes the iteration space of a loop into several tasks. This gives a boost to productivity when using the tasking model. However, the current implementation of this construct does not cover the vast majority of cases, because it is missing data dependences support. Therefore, users are forced to use explicit synchronization points, like in the fork-join model. As a result, users get a fork-join like structure, with increased overhead compared to worksharing constructs.

By providing support for data dependences to the `taskloop` construct, we enable users to utilize this directive in the majority of cases. Thus, they are able to fully convert their loops into tasks with a single directive, maximizing productivity, while keeping the main benefit of tasks, a lightweight synchronization based on data dependences.

Our evaluation shows that `taskloop` with dependences delivers as much performance as its manual counterpart, but with a reduced number of lines of code. The number of lines of source code required to parallelize a code using `taskloop` with dependences is 3.57x times smaller than its manual counterpart.

Finally, we provide a single construct that combines the two previous contributions: the `taskloop for` construct. This behaves like a regular `taskloop` but creating worksharing tasks rather than regular tasks. With this construct, we enable users to benefit from the performance enhancements of the worksharing tasks while keeping the improvements in programmability introduced by the `taskloop` with dependences.

Efficient Management of Memory Hierachies (NUMA)

In this chapter, we propose enhancements in the handling of memory hierarchies focusing on NUMA domains. Increasingly, the memory hierarchies of HPC systems becomes more complex, and considering the performance gap between memory and CPU, moving data is too expensive. We propose techniques that can track the location of data, and move compute rather than data.

5.1 Introduction and Motivation

Dennard scaling is a law that states with every new technology generation, transistors can reduce their area to the half, while keeping their power density constant, and increasing the frequency. Overall, every new generation, transistors gets faster while power consumption remains unchanged.

This fact has been true during many years, and processors has been increasing their performance thanks to this. However, it came to an end around 2006. As a result, chip manufacturers focused on multiprocessor and multicore architectures to keep increasing the performance of processors. Following this trend, processors have been incorporating more and more computing units, and, from several years ago, it is frequent to find architectures with several sockets in a single shared-memory space.

The idea of joining several sockets in a single address space simplifies the programming of such architectures, but it comes at a cost: the NUMA (non-uniform memory access) effect. The NUMA effect is a noticeable source of overhead if the accesses are not correctly managed. We experienced the NUMA effect in some of the experiments performed in Chapter 4. Concretely, we mentioned in Section 4.3.6 that we took some design decisions (the size of teams for worksharing tasks) to prevent NUMA effects. As well, in Section 4.6.2, we motivate the decision of using 1 MPI rank per NUMA node in the NUMA effect.

Additionally, newest trends in processor architectures are introducing on-chip NUMA. In this, apart from having different sockets in a single memory address space, each of the sockets include several memory controllers to improve memory latencies and bandwidth. All the cores within the socket are able to access all the memory controllers, but at different costs, similarly to the regular NUMA effect. Some examples of this trend are the Intel Knights Landing processor (KNL) [101], the IBM Power9 processor [93], the Fujitsu Post-K processor (A64FX) [64], and the AMD Zen 2 (ROME) processor [107].

The NUMA library, or *libnuma*, offers an API to handle the NUMA policy in the Linux kernel. This library provides a set of methods that enable expert users to perform a proper setup to eliminate or minimize NUMA effects. Nevertheless, it requires some expertise, reducing the share of users that can successfully utilize it. Additionally, *libnuma* provides methods to handle the allocation of data, to place data where it is best, or functions to move data, but the user is still in charge of doing a proper work scheduling to reduce data movement or remote accesses from one socket to others.

Linux also provides the *numactl* functionality. This is a command that enables users to run a process with a specific NUMA scheduling or memory placement policy. It is easier to use than *libnuma*, but still insufficient, leaving the work scheduling to the user.

In this chapter, we propose a new approach that improves data locality on NUMA systems. Our solution is based on the OmpSs-2 programming model, a directive-based task-based parallel programming model, similar to OpenMP. We first provide a simple API to properly allocate memory in NUMA systems using different policies. Then, using the information provided by the user to specify dependences between tasks, combined with the information collected when application's data is allocated, the runtime library is able to perform a NUMA-aware work scheduling.

Our contribution includes:

1. a simple and flexible API to distribute memory among different NUMA sockets in several possible ways (e.g., equally partitioned among all the nodes, block-cyclic, etc.), that can be used in C++ STL
2. a data-tracking system to know exactly where is the application's data
3. a NUMA-aware scheduling system that takes into account data locality and system load to minimize data motion and load imbalance.

5.2 Background

Architectures with several sockets in a single shared-memory address space are ubiquitous in modern HPC systems. Thus, the importance of adequately managing these architectures is crucial to maximize the performance of such systems.

One frequent problem of multi-socket systems is the NUMA effect. In this kind of systems, each socket has its own memory, but the compute units can access all the memories, regardless of the socket they belong to. Obviously, in the physical layout of the processor, some memories are closer than others, and so accessing remote memories is more expensive than accessing the local memory.

The default NUMA policy in Linux is *first-touch*. With this policy, when memory is allocated, it is not actually placed until it is either read or written for the first time. This policy places the memory in the NUMA node of the compute unit that firstly touched data. This policy can easily lead to suboptimal distribution of data, negatively affecting application performance. Thus, the first step to minimize the NUMA effect is a proper distribution of data.

As we explained earlier in this document, a very common way of programming shared-memory environments is using parallel programming models, such as OpenMP. OpenMP provides two main ways of exploiting parallelism: using worksharing constructs and using tasking constructs. Next, we provide a brief recall about those strategies.

Worksharing constructs distribute the work among all the available threads, and wait at the end until all the threads have finished. This technique introduces very few overhead, but implies a very rigid synchronization, that makes it a bad option for imbalanced or irregular workloads.

Tasking constructs provide a more lightweight synchronization by using data dependences. A task can run as soon as all of its data dependences are satisfied. Thus, it is less frequent to waste time waiting for other threads to finish their work, as happens with worksharing constructs. However, task management introduces more overhead than worksharing constructs, especially when using small granularities. Notwithstanding, the issue of granularity has already been covered in literature [80][72].

For the purposes of this work, tasking is specially the right choice because of the extremely valuable information that data dependences provide to the runtime system. As users already has to annotate the data that is read or written in a task, we can use that information to perform a good scheduling, sending tasks to com-

pute units in the right NUMA socket, which is the one with a biggest share of data accessed in a task.

As said, tasks can run as soon as all of its dependences are satisfied. In fact, when all the dependences of a task are satisfied, the task becomes *ready*. At this point, the runtime system schedules it for execution, and the task will run as soon as a compute unit gets it from the ready queue. If there is no specific scheduling policy, the task may run in a compute unit of the wrong NUMA socket, experiencing an important performance penalty for accessing to remote nodes. This is the second key point to minimize NUMA effect, a proper work scheduling.

To sum up, there are two main challenges to minimize the NUMA effect, as demonstrated in [27][26]:

1. A proper distribution of data.
2. A proper work scheduling based on data location.

Both are equally important, because if a single one fails, it could jeopardize the performance of the application.

5.3 Nanos6 NUMA Allocation API

We already stated in previous sections the importance of a proper distribution of data to minimize NUMA effects. In this section, we present the Nanos6 NUMA allocation API. This is a simple API that offers a lot of possibilities to users. Contrasting with other contributions of the thesis where we used directives, we provide here an API for two reasons. Firstly, an API gives users the possibility of distributing their data over the different NUMA domains without requiring a compiler. Besides that, in our other contributions, the use of directives actually provides a benefit, usually in terms of programmability requiring less effort from the user side. In this case, the user would have to provide the same parameters to the compiler than it does directly to the runtime. The compiler would not be adding any real value, only replacing the API call by a directive, but the rest would remain the same.

Next, we introduce the methods included in the API, and detail their behavior.

`nanos6_numa_alloc_block_interleave(size, bitmask, block_size)`: This method serves to allocate data. Users should replace their regular allocation function (e.g., `malloc`, `mmap`, `new`, etc.) by this one. It allocates *size* bytes, interleaving blocks of

block_size among the nodes specified in *bitmask*. This very simple function offers a lot of possibilities. For instance, in Listing 5.1 we show (1) how to spread data among all the available NUMA nodes, (2) how to allocate all the data in a single NUMA node, (3) how to distribute data using a block-cyclic policy among all the available NUMA nodes.

nanos6_numa_alloc_sentinels(size, bitmask, block_size): This method serves to allocate sentinels or representatives. A detailed use case for this method is shown in Section 5.3.1. Briefly, it allocates *size* bytes, but rather than actually interleaving blocks of *block_size* among the nodes specified in *bitmask*, it allocates everything in a single node but annotates it as if it was distributed.

nanos6_numa_free(ptr): Users can invoke this method to release memory. *ptr* is the pointer to free, and it must be a pointer returned by the nanos6 NUMA allocation function. Apart from the memory release, it also removes the associated directory information.

nanos6_bitmask_clearall(bitmask): Sets all bits of *bitmask* to 0.

nanos6_bitmask_clearbit(bitmask, n): Sets the *n*-th bit of *bitmask* to 0.

nanos6_bitmask_setall(bitmask): Sets to 1 the *N* least significant bits of *bitmask*, being *N* the available number of NUMA nodes in the machine.

nanos6_bitmask_setbit(bitmask, n): Sets the *n*-th bit of *bitmask* to 1.

nanos6_bitmask_set_wildcard(bitmask, wildcard): Sets *bitmask* to the corresponding value depending on the wildcard. Supported wildcards are `NUMA_ALL`, which represents all the nodes available in the system; `NUMA_ALL_ACTIVE`, which represents the nodes where we have **all** the cores assigned; and `NUMA_ANY_ACTIVE`, which represents the nodes where we have any of the cores assigned..

nanos6_bitmask_isbitset(bitmask, n): Returns the value of the *n*-th bit of *bitmask*.

nanos6_count_setbits(bitmask): Returns the amount of enabled bits in *bitmask*.

The default value of the bitmask is unspecified. Thus, users must use the bitmask manipulation methods to ensure a correct value.

Listing 5.1: Examples of nanos6 NUMA allocation method

```
nanos6_bitmask_t bitmask;
nanos6_bitmask_set_wildcard(&bitmask, NUMA_ALL);
int NUMA_nodes = nanos6_count_setbits(&bitmask);
size_t size = 4096*sizeof(int);

// Case 1: Distribute data among all the NUMA nodes
```

```

size_t block_size = size/NUMA_nodes;
void *A = nanos6_numa_alloc_block_interleave(size, &bitmask,
    block_size);

// Case 2: Allocate all data in a single NUMA node (node 0)
int node_id = 0;
nanos6_bitmask_clearall(&bitmask);
nanos6_bitmask_setbit(&bitmask, node_id);
size_t block_size = size;
void *B = nanos6_numa_alloc_block_interleave(size, &bitmask,
    block_size);

// Case 3: Block-cyclic among all the NUMA nodes
nanos6_bitmask_set_wildcard(&bitmask, NUMA_ALL);
size_t block_size = 512*sizeof(int);
void *C = nanos6_numa_alloc_block_interleave(size, &bitmask,
    block_size);

```

5.3.1 Support for Representatives

In our implementation, we leverage the *libnuma* to place memory where the user requests. We first allocate the virtual memory using `mmap`, and then, we use some *libnuma* methods to place the allocated memory where the user asked to. Nevertheless, the minimum granularity that can be moved is a memory page. In consequence, if the user tries to allocate a chunk of memory smaller than page size, or asks to interleave blocks that are smaller than page size, the OS cannot handle it.

This is a possible scenario in applications that are written using representatives/sentinels instead of the actual data. Listing 5.2 shows a real example of such a scenario. The code simulates the Heat equation using Gauss-Seidel solver.

Listing 5.2: Gauss-Seidel using representatives

```

static inline void gaussSeidelSolver(int64_t rows, int64_t cols,
    int rbs, int cbs, int nrb, int ncb, double M[rows][cols], char
    reps[nrb][ncb])
{
    for (int R = 1; R < nrb-1; ++R) {
        for (int C = 1; C < ncb-1; ++C) {
            #pragma oss task label(block computation) \

```

```

        in(reps[R-1][C]) in(reps[R
            +1][C]) \
        in(reps[R][C-1]) in(reps[R
            ][C+1]) \
        inout(reps[R][C])
    computeBlock(rows, cols, (R-1)*rbs+1, R*rbs
        , (C-1)*cbs+1, C*cbs, M);
    }
}

void solve(HeatConfiguration *conf, int64_t rows, int64_t cols, int
    timesteps, void *extraData)
{
    double (*matrix)[cols] = (double (*)[cols]) conf->matrix;
    const int rbs = conf->rbs;
    const int cbs = conf->cbs;

    const int nrb = (rows-2)/rbs+2;
    const int ncb = (cols-2)/cbs+2;
    char representatives[nrb][ncb];

    for (int t = 0; t < timesteps; ++t) {
        gaussSeidelSolver(rows, cols, rbs, cbs, nrb, ncb,
            matrix, representatives);
    }
    #pragma omp taskwait
}

```

The actual data is blocked, and the dependences are over a whole block, all of them of the same size. To facilitate the writing of the dependences, it uses a matrix of representatives where each block is represented by a single element. In this way, regardless of the block size, or the matrix size, the dependences are always correct and easy to understand. Representatives are similar to sentinels but present a key difference: usually, sentinels are the first element of a block of the real accessed data, while representatives are a new portion of memory with a single element per block, representing the whole block. In both cases a single element represents a whole block, but for sentinels it is an element of the real accessed data, while in representatives it is not the real accessed data.

Notwithstanding, for our proposal, this is a problem because the representatives

matrix is much smaller than the actual matrix, and so, we cannot actually distribute data in different sockets, because the blocks are smaller than page size, and, as said, *libnuma* does not provide any possibility of moving blocks smaller than page size. As a result, when running the tasks with dependences expressed using representatives, we would do a wrong scheduling, because data is not where we believe.

To overcome this limitation, we have decided to allocate data in a single node, but annotate in our global directory (detailed in Section 5.4) the data regions as if we could actually distribute them the way the user asked to. We can do so by using the `nanos6_numa_alloc_sentinels` method. As this is not the data that is being actually accessed, it is not really important where it resides. In fact, the important thing is to mimic as accurately as possible the distribution of the data actually accessed, and we truly believe that this is the best solution to support this scenario. Note that the data actually accessed must be allocated using our API too.

5.4 Nanos6 Data Tracking System

To perform a proper work scheduling, we need to know (1) where is the data, and (2) what data reads/writes each task. In this section, we detail how we obtain this knowledge to be later used in the scheduling system.

Our data tracking system contains a global directory that stores in which NUMA node resides all the data that has been allocated using the Nanos6 API. However, a single allocation may reside in several different NUMA nodes, since each of the pages can be placed in a different socket. Figure 5.1 shows the distribution of data and its entries in the directory of the allocations done in Listing 5.1. In the directory, there is an entry for each of the regions that belong to a different socket.

Figure 5.2 shows a more complex situation, where rather than just dividing the overall data region in as many blocks as the available NUMA nodes, the data is divided in many more regions, performing a block-cyclic distribution. Again, the directory stores an entry for each of the regions belonging to a different socket.

The global directory is protected with a lock, to prevent data corruption. This could be a big source of overhead if not properly managed, because there could be many concurrent interactions with the directory, but only a single thread can access it at a time. Firstly, we use a *read-write* lock. This kind of lock allows multiple readers to enter the protected area concurrently, while it only permits one single writer (and no readers) at a time.

The most frequent pattern for memory allocation/deallocation is to perform the

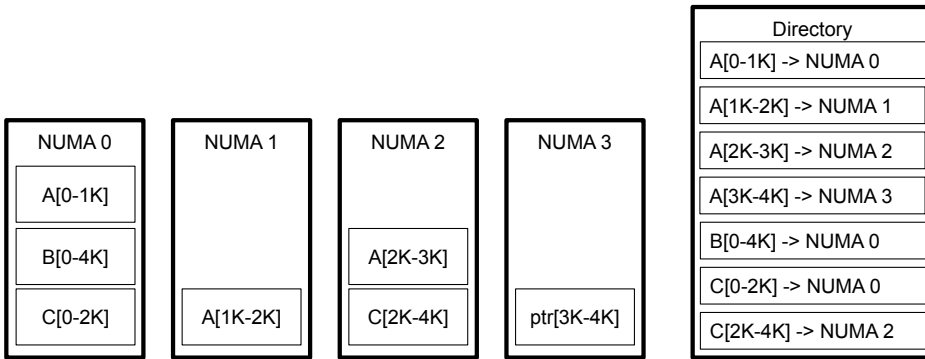


Figure 5.1: Nanos6 directory tracking different memory regions.

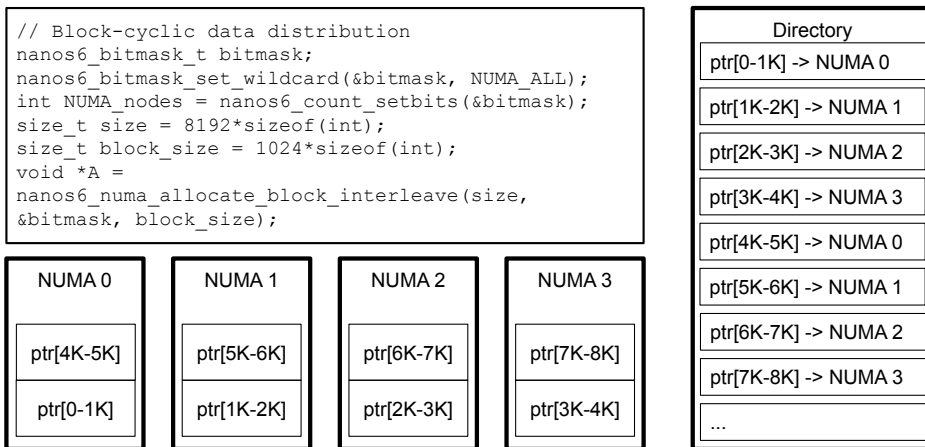


Figure 5.2: Nanos6 directory tracking different memory regions in a more complex scenario.

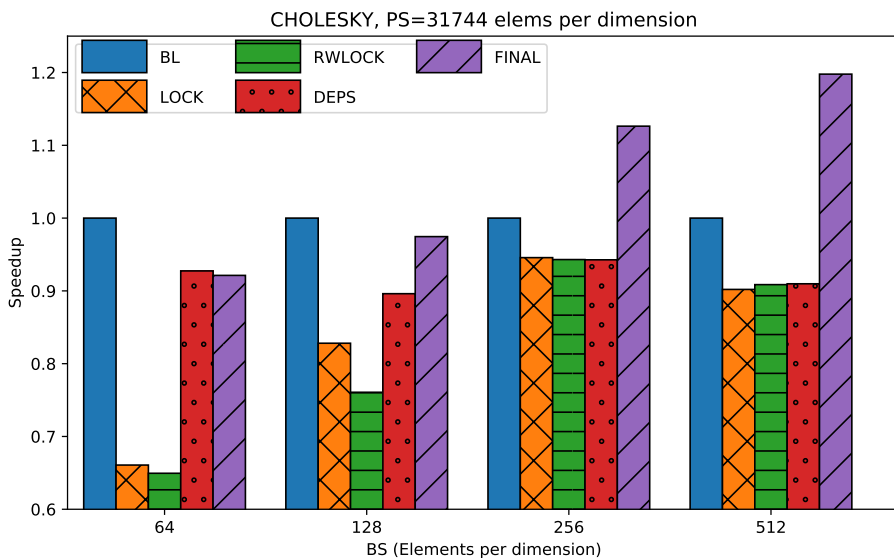


Figure 5.3: Detailed analysis of different implementations of the Nanos6 Data Tracking System on different granularities of the Cholesky benchmark

allocation at the beginning and the deallocation at the end of applications. With such a pattern, all the modifications of the directory are done at the beginning and at the end, because it is only modified to insert new entries on allocations, and to erase entries on deallocations. Additionally, in this pattern of allocations at the beginning and deallocations at the end, the usual case is that there is no threading yet, so there is a single thread writing in the directory. Thus, theoretically, no significant overhead is introduced up to this point.

Thanks to the data dependences of the tasks, we can easily know which data accesses each task. Thus, we can simply query the directory to know where resides the data of each task. Given that queries do not modify the directory, they can be performed as a reader, and so as many threads as desired can enter the protected area, minimizing the overhead. Nevertheless, we saw that it still introduced some overhead, especially with fine-grained tasks.

Therefore, we implemented a further optimization to minimize the number of queries to the directory. When tasks register their dependences, before querying the directory, we try to propagate the location of data from parents or siblings. We

use the same mechanism that propagates satisfiability (i.e. when a task becomes ready) to propagate the location, so the overhead is kept very low. If no propagation is possible, then, we do a query. In practice, this means that only the first task that accesses a data region performs a query, and all the rest are able to get it by propagation. Overall, introducing negligible overhead.

Figure 5.3 shows a comparison between several implementations of the tracking system. In the figure, BL is the baseline, a regular OmpSs-2 parallelization with no numa-awareness at all; in the LOCK version, the global directory is protected by a regular lock; in the RWLOCK version, a read-write lock protects the global directory; the DEPS version uses the propagation through the dependency system. In these versions, there is no NUMA scheduling, they are just intended to measure the overhead of the tracking using the different implementations. Finally, there is the FINAL version, that uses the propagation through the dependency system, and adds the NUMA scheduling. The chart shows speedup compared to the BL version in the y-axis, and block size in the x-axis. It is possible to see how the tracking introduces more overhead in the fine granularities. Also, we can observe that the DEPS version is the one that introduces fewer overhead across all the granularities. Finally, we see a boost in performance when adding the NUMA scheduling, actually reaching speedup against the baseline.

We measured the performance of the explained versions for all the benchmarks of our evaluation. Based on the previous analysis, we selected a fine granularity to see if the behavior remains across the different applications. We show the results of this measurement in Figure 5.4. We can see almost no differences between the LOCK and RWLOCK versions. Nonetheless, the DEPS version is able to outperform the other two versions across all the benchmarks because it reduces considerably the number of queries. Finally, we can observe the FINAL version increasing the performance with respect to the DEPS version in most of the benchmarks.

5.5 Nanos6 NUMA-Aware Scheduling System

The scheduling system enables us to minimize remote memory accesses by scheduling tasks to the NUMA node with better affinity. Our NUMA-aware scheduling system contains one ready-queue per NUMA node available in the execution. To decide in which ready queue we have to add a task, we compute an affinity *score*. The NUMA node with the highest score, is the NUMA node where the runtime system enqueues the task.

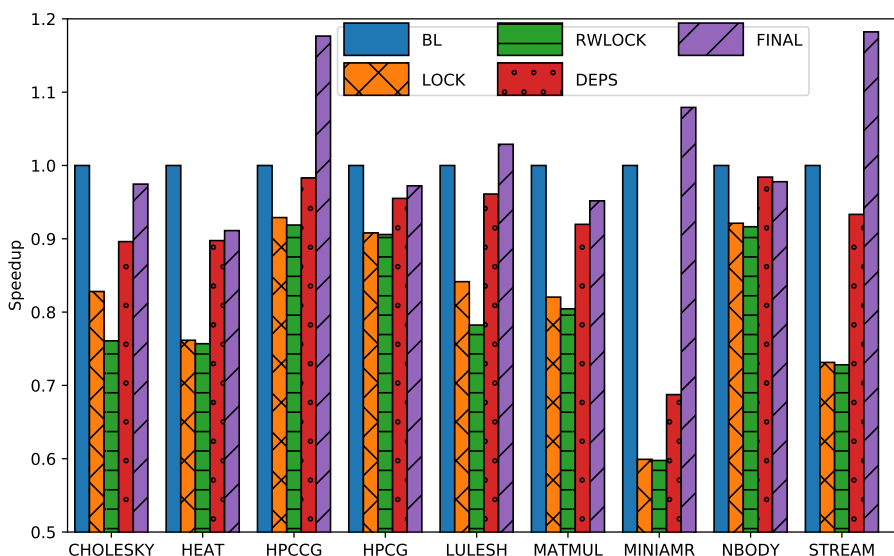


Figure 5.4: Analysis of different implementations of the Nanos6 Data Tracking System

The runtime library computes the *score* of a task using the information of the data tracking system detailed in Section 5.4. Each of the accesses of a task contains the location where it resides. Using the locations, the type of access (in, out or inout), and the weights of each access, we can easily compute the amount of bytes that each NUMA node contains, and derive a *score*. Each task contains its own accesses, and can get the location, type and weight of each access without any kind of synchronization, introducing a minimal overhead. Concretely, the score is the number of bytes of the task that a NUMA node contains. In case an access is an inout access, then, the number of bytes of that access counts double. The pseudo-code in Listing 5.3 shows how we compute the score.

Listing 5.3: Compute of NUMA score

```

for (node in NUMANodes)
    score[node] = 0.0

for (access in task.getAccesses())
    homenode = access.getHomenode()
    if (access.isReadWrite())

```

```
    score[homenode] += access.getSize() * 2.0;
else
    score[homenode] += access.getSize()
```

So the tasks are enqueued in the ready queue of the NUMA node that obtained the highest score. Thus, when a compute unit requests work from the scheduling, the runtime system checks its NUMA socket, and tries to get a task from the specific ready queue of that socket. If it is not possible, because there are no ready tasks, it steals tasks from the ready queues of other NUMA nodes. The stealing is done based on two factors: (1) the *distance* between the NUMA nodes, and (2) the load balance. We try to steal tasks from closer NUMA nodes to reduce the NUMA effect penalty. However, if a NUMA node contains only a few tasks, we steal from other queues with a higher number of tasks to prevent further stealing, due to the associated NUMA effect penalty that it has.

Additionally, our runtime system implements an immediate successor mechanism. This mechanism tries to exploit temporal locality by skipping the regular logic of the scheduler and directly executing successor tasks. When a task finishes and releases its dependences, some of its successor tasks may become ready. If that is the case, the finished task and the successor task are obviously sharing some data, so it is a good idea to run the successor task immediately in the same core to exploit the temporal locality. This mechanism has priority over the NUMA mechanism, because the temporal locality may expire while the spatial locality of the NUMA does not.

Overall, this system minimizes data motion by moving compute to where data is rather than the other way around. As a result, the NUMA effect is mitigated, and the performance improves.

5.6 Changes Required in the Application

To benefit from our approach, users require very few and simple changes in their codes. They just have to replace regular allocation/deallocation calls by the allocation/deallocation functions of the Nanos6 NUMA API, and specify the size of the regions accessed when writing task dependences. We would like to highlight that users do not have to replace all the allocations/deallocations, but only those that are used within tasks dependences, which is the information used by the runtime to guide the scheduling.

ORIGINAL	NUMA-AWARE
<pre> // Allocate vectors double *a = new double[N]; double *b = new double[N]; double *c = new double[N]; // Initialization for (int i = 0; i < NUM_BLOCKS; i++) #pragma oss task out(a[i*TS], b[i*TS], [i*TS]) // init a, b, c for (int step = 0; step < timesteps; step++) for (int i = 0; i < NUM_BLOCKS; i++) { #pragma oss task in(a[i*TS]) out(c[i*TS]) // copy kernel #pragma oss task in(c[i*TS]) out(b[i*TS]) // scale kernel #pragma oss task in(a[i*TS], b[i*TS]) out(c[i*TS]) // add kernel #pragma oss task in(b[i*TS], c[i*TS]) out(a[i*TS]) // triad kernel } // Release memory delete a; delete b; delete c; </pre>	<pre> nanos6_bitmask_t bitmask; nanos6_bitmask_set_wildcard(&bitmask, NUMA_ALL); size_t numa_nodes = nanos6_count_setbits(&bitmask); size_t size = N*sizeof(double); size_t block_size = size/numa_nodes; // Allocate vectors double *a = nanos6_numa_alloc_block_interleave(size, &bitmask, block_size); double *b = nanos6_numa_alloc_block_interleave(size, &bitmask, block_size); double *c = nanos6_numa_alloc_block_interleave(size, &bitmask, block_size); // Initialization for (int i = 0; i < NUM_BLOCKS; i++) #pragma oss task out(a[i*TS :TS], b[i*TS:TS]) c[i*TS:TS]) // init a, b, c for (int step = 0; step < timesteps; step++) for (int i = 0; i < NUM_BLOCKS; i++) { #pragma oss task in(a[i*TS :TS]) out(c[i*TS :TS]) // copy kernel #pragma oss task in(c[i*TS :TS]) out(b[i*TS :TS]) // scale kernel #pragma oss task in(a[i*TS :TS], b[i*TS :TS]) out(c[i*TS :TS]) // add kernel #pragma oss task in(b[i*TS :TS], c[i*TS :TS]) out(a[block*TS :TS]) } // Release memory nanos6_numa_free(a); nanos6_numa_free(b); nanos6_numa_free(c); </pre>

Figure 5.5: Changes required in the application to use the Nanos6 NUMA-aware system

Figure 5.5 show a baseline code in the left, and the same code with the modifications required to exploit our solution in the right. The code is a simplification of the Stream benchmark. As can be seen, we replace the regular allocation/deallocation by the Nanos6 NUMA API methods, and we add the size of the region in each of the dependences.

5.6.1 Dealing with C++ Standard Library

Using the Nanos6 NUMA API is a bit more complicated if we use containers from the C++ Standard Library, such as `std::vector`. In this case, we need to define a STL-compliant allocator that uses the Nanos6 NUMA API for allocating/deallocating, and pass this allocator to each of the containers when they are created. Listing 5.4 shows a simplified example of how to this.

Listing 5.4: Using Nanos6 NUMA API with C++ STL

```

template typename<T>
class STL_allocator {
    ...
    pointer allocate (size_type n) {
        size_t size = n*sizeof(T);
        size_t block_size = size/numa_nodes;

```

```

        pointer ptr = (pointer) nanos6_numa_alloc_block_interleave(
            size, &bitmask, block_size);
    }

    void deallocate (pointer p, size_type n) {
        nanos6_numa_free(p);
    }
};

std::vector<int , STL_allocator<int> > v;

```

5.7 Evaluation and Discussion

In our evaluation, we include a detailed analysis of a well-known memory-bound benchmark, the Stream benchmark. In this analysis, we add the different components step by step to see their impact.

After that, we perform an analysis of several applications/benchmarks using all the optimizations to prove that our approach is able to get benefit across different scenarios.

5.7.1 Environment

Regarding the execution environment, all the experiments were carried out on two different platforms. A node of platform A is composed of 2 sockets Intel Xeon Platinum 8160 2.1GHz 24-core [21], each socket containing a single NUMA node. A node of platform B is composed of 2 sockets AMD EPYC 7H12 2.6Ghz 64-core, each socket containing 4 NUMA nodes. Figure 5.6 shows the distances between NUMA nodes of both platforms. Regarding the software, we used the Mercurium compiler [11] (v2.3.0), the Nanos6 runtime [12] (v2.4.1), the gcc and gfortran compilers (v7.2.0 in platform A, v10.1.0 in platform B), and the Intel compilers (v17.0.4 in platform A, and v19.0.5 in platform B).

5.7.2 Applications and Benchmarks

The applications and benchmarks used in the evaluation are listed following:

Cholesky. This benchmark performs a Cholesky factorization.

Figure 5.6: Distances between NUMA nodes

			Platform B								
			0	1	2	3	4	5	6	7	
Platform A			0	10	12	12	12	32	32	32	32
0	1		1	12	10	12	12	32	32	32	32
			2	12	12	10	12	32	32	32	32
			3	12	12	12	10	32	32	32	32
			4	32	32	32	32	10	12	12	12
			5	32	32	32	32	12	10	12	12
			6	32	32	32	32	12	12	10	12
			7	32	32	32	32	12	12	12	10

Heat (Gauss-Seidel). This benchmark performs a 2D heat transfer simulation using a Gauss-Seidel solver.

HPCCG. This is a simple conjugate gradient benchmark code for a 3D chimney domain. It is part of the Mantevo project.

HPCG. This application implements a preconditioned conjugate gradient. It is a well-known application used to benchmark HPC systems as a complement of the High Performance Linpack (HPL).

LULESH. This application models the propagation of a Sedov blast wave. Developed by the Lawrence Livermore National Lab, it is one of the five challenge problems in the DARPA UHPC program.

Matmul. This benchmark is a typical matrix multiplication kernel using Intel Math Kernel Library.

MiniAMR. This application applies a stencil calculation on a unit cube computational domain. It supports adaptive mesh refinement. It is part of the Mantevo project.

N-Body. This benchmark simulates a dynamical system of particles.

Stream: The Stream benchmark measures the sustainable bandwidth and the corresponding computation rate for a simple vector kernel. It is part of the HPC Challenge Benchmarks.

All of them have been ported to OmpSs-2, using a fully taskified parallelization. Table 5.1 shows the number of lines of code changed to use the OmpSs-2 NUMA-aware system. Although, usually, it is only required to allocate the main data structures of the applications to benefit from the OmpSs-2 NUMA-aware sys-

Table 5.1: Number of lines of code changed to use the OmpSs-2 NUMA-aware system.

	Number of lines changed
CHOLESKY	5
HEAT	10
HPCCG	41
HPCG	59
LULESH	157
MATMUL	9
MINIAMR	86
NBODY	12
STREAM	9

tem, we replaced all the regular allocations/deallocations by those of the Nanos6 NUMA API.

5.7.3 Performance

As previously introduced, we include two different experiments in this evaluation. One detailed step-by-step analysis using the Stream benchmark, to see the impact of the different components of the Nanos6 NUMA-aware system; and one experiment using several applications and benchmarks with the whole set of components of the Nanos6 NUMA-aware system to demonstrate that it is beneficial across different scenarios.

Detailed Analysis of Stream Benchmark

Figure 5.7 shows the results of the detailed analysis. In the x-axis, we show two different block sizes, `OPTIMAL BS` and `BIG BS`. The first block size is one that reaches the peak performance, thanks to a good exploitation of cache locality. The second block size is one bigger block size that does not fit in cache, so it gets much lower performance. The y-axis shows speedups, using 1-FT as the baseline. Each of the bars represent a different version:

1-FT. Regular OmpSs-2 version using the default allocation policy (first touch) running with a single NUMA node.

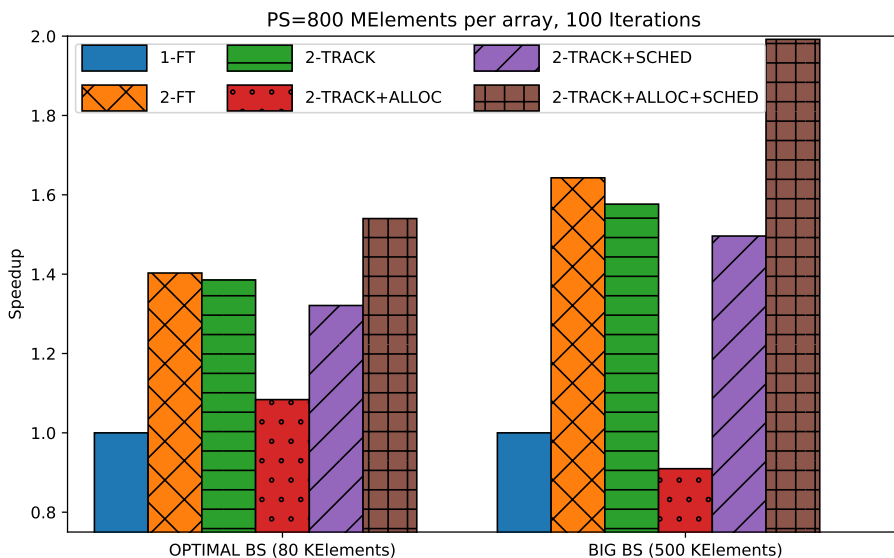


Figure 5.7: Detailed analysis of Nanos6 NUMA-aware system using the Stream benchmark

2-FT. Regular OmpSs-2 version using the default allocation policy (first touch) running with two NUMA nodes.

2-TRACK. OmpSs-2 version running with two NUMA nodes, with Nanos6 NUMA tracking system enabled. This version allows us to prove that the tracking introduces no significant overhead.

2-TRACK+ALLOC. OmpSs-2 version running with two NUMA nodes, with Nanos6 NUMA tracking system enabled and NUMA-aware allocation using the Nanos6 NUMA API. This version allows us to prove that a proper allocation is not enough to mitigate the NUMA effect.

2-TRACK+SCHED. OmpSs-2 version running with two NUMA nodes, with Nanos6 NUMA tracking system enabled and NUMA-aware scheduling. This version shows that the NUMA-aware work scheduling is not effective without the information provided during the allocation.

2-TRACK+ALLOC+SCHED. OmpSs-2 version running with two NUMA nodes, with complete Nanos6 NUMA-aware system enabled. This version shows that the combination of a good data distribution and a proper NUMA-aware work scheduling can mitigate the NUMA effect.

This experiment has been performed in platform A, so the machine used in this experiments contains two NUMA sockets. Therefore, ideally, we should be able to double the performance of 1-FT when using two NUMA nodes. In Figure 5.7 we can see different results for the two block sizes. In the case of OPTIMAL BS, none of the versions is able to double the performance. The reason behind this is the cache locality. This block size uses a data set that fits in cache, and so, there are very few accesses to main memory, so the NUMA effect has a smaller impact. Even though, we can see that 2-TRACK has very similar performance than 2-FT, so the overhead of the tracking system is minimal.

2-TRACK+ALLOC presents worse performance than 2-FT. 2-TRACK+ALLOC distributes data among the two available NUMA nodes, but then there is no NUMA-aware scheduling, so there are many accesses to the remote node. In the case of 2-FT, it relies on the default NUMA policy (i.e. first touch), so pages are placed in the NUMA node of the CPU that runs the initialization tasks. Therefore, for 2-FT all the initialization tasks suffer no NUMA effect, while the NUMA effect may penalize 2-TRACK+ALLOC. In the rest of the execution, usually the same CPU that runs the initialization tasks gets the successors, so, at least the first bunch of tasks usually run in the right NUMA socket. In contrast, in the 2-TRACK+ALLOC, if the initialization task ran in the wrong NUMA socket, probably the successors will do so, overall being more penalized.

2-TRACK+SCHED bar is slightly worse than 2-FT. This is because we are paying the penalty of performing the data tracking and the NUMA-aware scheduling, but it is useless because we have no information about the data, since we get it from the allocation.

Finally, 2-TRACK+ALLOC+SCHED gets 10% more performance than 2-FT. This proves that with all the components enabled, the NUMA effect is mitigated, and performance improves.

The trends are very similar for the BIG BS, and the explanations too. The only significant difference is that 2-TRACK+ALLOC+SCHED actually gets very close to ideal speedup (i.e. 2x speedup) because in this case the cache is useless, and the vast majority of accesses use main memory. Thus, when we mitigate NUMA effect, we are able to get all the bandwidth from both NUMA nodes.

Analysis of Several Applications/Benchmarks

This experiment involves several versions. The names that appears in the legends of figures as follows: 1-FT. Regular OmpSs-2 version running with a single NUMA

node.

N-FT. Regular OmpSs-2 version running with N^1 NUMA nodes.

N-INTERLEAVED. OmpSs-2 version using *numactl* to perform data interleaving, running with N^1 NUMA nodes.

N-TRACKING). OmpSs-2 version with NUMA-awareness enabled running with N^1 NUMA nodes. NUMA-awareness includes proper allocation using Nanos6 NUMA api, data tracking and NUMA-aware scheduling.

The versions 1-FT serve us as baseline. The optimal scenario is to get as many speedup, compared to the baseline (a single NUMA node), as NUMA nodes available in the system.

For each of the benchmarks/applications, we selected a big problem size that does not fit in none of the levels of cache. We selected big block sizes as well, also to prevent them from fitting in cache. The overall objective of these choices is to focus on NUMA ignoring caches. Nevertheless, is fair to point out that for the used benchmarks/applications, the optimal block size (i.e. the one that reaches peak performance) does not fit in cache except for the Stream benchmark, which we already analyzed in detail in the previous experiment. So, in this experiment, we are using the optimal block size for all the benchmarks/applications but the Stream benchmark.

Figure 5.8 shows the results for the benchmarks in platform A, and Figure 5.9 shows the results of the applications/benchmarks platform B. In the x-axis there are different benchmarks, and in the y-axis there are speedups. The speedups are computed using 1-FT as the baseline.

Regarding Figure 5.8, for Cholesky, 2-FT reaches almost 1.75x speedup compared to 1-FT, that is quite good considering that the optimal is 2x. 2-TRACKING slightly enhances the speedup up to 1.85x, close to the optimal. Finally, 2-INTERLEAVED presents a small downgrade compared to 2-FT. This fact stresses the importance of combining a good data distribution with a proper work scheduling, showing that simply distributing data is not enough.

For Heat, 2-FT performs very similarly to 1-FT. This fact suggests a huge impact of the NUMA effect in 2-FT. This is confirmed looking at the results of 2-TRACKING and 2-INTERLEAVED, that are able to reach up to 1.86x and 1.95x speedup, respectively.

In HPCCG, we see that 2-FT performs even worse than 1-FT, and so does 2-INTERLEAVED. In contrast, 2-TRACKING reaches a 1.85x speedup, very close to the ideal speedup of

¹ N is the number of NUMA nodes available in the system

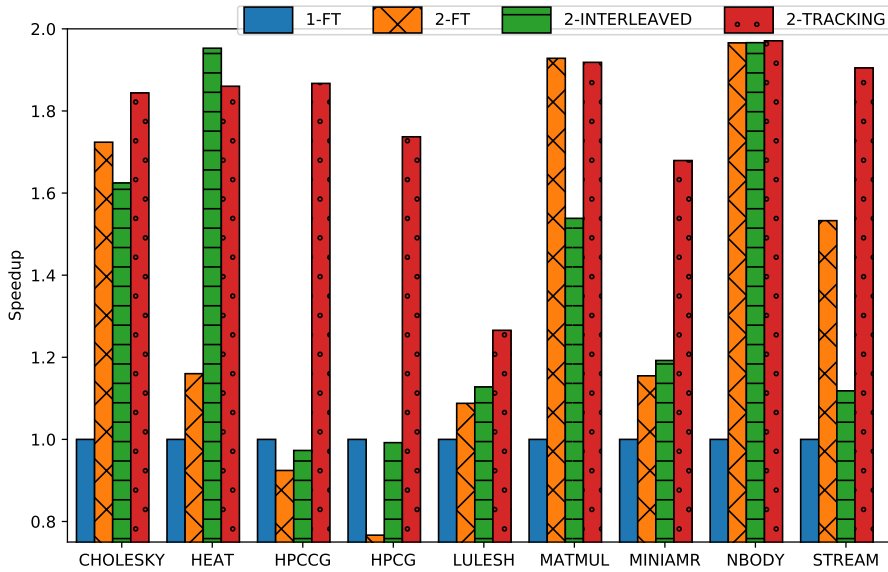


Figure 5.8: Analysis of Nanos6 NUMA-aware system in platform A

2x.

The HPCG behaves similarly to the HPCCG. In this case, 1-FT outperforms 2-FT, and performs similarly to 2-INTERLEAVED. Again, 2-TRACKING outperforms all the versions, reaching up to 1.75x speedup.

In LULESH, none of the versions is able to reach more than 1.25x speedup. LULESH presents a very complex access pattern that makes very difficult to prevent accesses to the remote node. 2-FT is slightly better than 1-FT, but as said, due to the complex access pattern of the application, there are many accesses to the remote node, jeopardizing the performance. 2-INTERLEAVED shows a speedup very similar to 2-FT. 2-FT(TRACKING) boosts performance a bit, but it is still far from the ideal speedup. In this case, our system is not so effective because the parallelization of this application is focused on maximizing the parallelism of the application while assuring a correct result, but the data annotations do not always contain the data that is really accessed because the parallelization is coded using sentinels.

In Matmul, 2-FT is very close to the optimal speedup with a 1.9x speedup, and so does 2-TRACKING. In contrast, 2-INTERLEAVED shows a significant performance reduction, stressing again the importance of a good work scheduling.

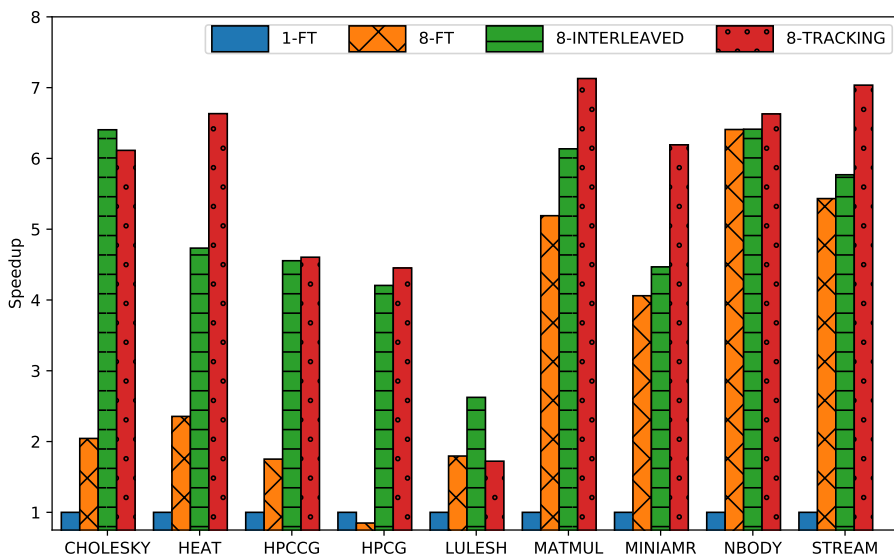


Figure 5.9: Analysis of Nanos6 NUMA-aware system in platform B

MiniAMR also presents a complex access pattern that difficultly prevents accesses from remote nodes. As a result, 2-FT presents very low improvement compared to 1-FT. Our NUMA-aware system is able to mitigate the NUMA effect, and scale up to 1.68x. Regarding 2-INTERLEAVED introduces a very slight improvement compared to 2-FT, but far from 2-TRACKING.

The N-body simulation is a compute-bound benchmark, so the NUMA effect does not affect it, because the memory bandwidth is not the bottleneck. As a result, all the versions using 2 NUMA nodes are very close to the ideal speedup.

For the Stream benchmark, 2-FT reaches 1.5x speedup, leaving some room for improvement. 2-TRACKING is able to boost performance up to a 1.99x, reaching the optimal speedup. Finally, 2-INTERLEAVED again show the necessity of a good work scheduling.

Respecting Figure 5.9, trends are similar to those in platform A, where 8-FT usually suffers NUMA effect, and 8-TRACKING and 8-INTERLEAVED mitigate the mentioned NUMA effect, with 8-TRACKING usually delivering the best performance.

In Cholesky, 8-FT barely reaches a 2x speedup, while the ideal speedup is 8x. Both 8-TRACKING and 8-INTERLEAVED heavily boost performance, reaching up to

6.1x and 6.4x speedup, respectively.

Similarly, in the Heat benchmark, 8-FT only reaches a 2.35x speedup, while 8-TRACKING and 8-INTERLEAVED reach 6.6x and 4.73x respectively, mitigating the NUMA effect.

The trend continues in the HPCCG benchmark, where 8-FT does not even reach a 2x speedup, while 8-TRACKING and 8-INTERLEAVED can mitigate the NUMA effect and speed up the application up to 4.6x.

The NUMA effect is terrible in the HPCG benchmark, where 8-FT performs even worse than 1-FT. One more time, 8-TRACKING and 8-INTERLEAVED are able to mitigate the NUMA effect, but are still affected.

In platform B, the issue of LULESH is even more clear. None of the versions is able to reach not even a 3x speedup, although the resources are multiplied by 8. Additionally, the problem with the parallelization not really specifying the accessed data already explained in platform A, makes that 8-TRACKING delivers worse performance than 8-FT. This makes clear that our system only works well when the data annotations contain the data that is accessed.

The Matmul benchmark shown no NUMA effect in platform A. In this case, in platform B, it really affects, although its effect is much smoother than in other benchmarks. Again, 8-TRACKING and 8-INTERLEAVED can mitigate the mentioned effect, reaching speedups of 7.1x and 6.1x, respectively.

Like the Matmul benchmark, the N-body simulation shown no NUMA effect in platform A. In platform B, there is a slight NUMA effect, that causes that an increase of 8x in the resources only becomes a 6.4x speedup in performance. 8-TRACKING slightly boosts performance reaching up to a 6.6x speedup.

Finally, the Stream benchmark shows also a smooth NUMA effect. In this case, 8-INTERLEAVED barely improves the performance of 8-FT. However, 8-TRACKING is able to reach up to 7x speedup.

Overall, our tracking system is able to mitigate the NUMA effect across several different applications/benchmarks and platforms, reaching the optimal speedup in some of them. Additionally, it is able to outperform in most of the applications/benchmarks, in both platforms, the interleaving done using the *numactl* command even though this command is able to interleave also the memory managed inside the runtime, while our approach only affects to application memory.

5.8 Opportunities at Scale

In this work, we focused our evaluation on single-node environments. However, based on the results shown on the evaluation, our work opens opportunities at scale.

Usually, applications run at scale rely on MPI parallelizations or hybrid MPI+X parallelizations. Based on previous works [73], the best configurations for MPI use 1 MPI process per core, and for MPI+X use 1 MPI process per NUMA node. In the case of MPI+X, it is done to prevent NUMA effect from jeopardizing application performance. With our approach, we could be using a single MPI process per node, dramatically reducing the total number of MPI processes required. Considering the platforms used in the evaluation of this work, while a pure MPI parallelization would require 48 processes per node and 128 processes per node in platform A and platform B, respectively, our approach would require only 1 process per node. An MPI+X hybrid parallelization would require 2 process per node and 8 processes per node in platform A and platform B, respectively, while our approach would only require 1 process per node.

Reducing the number of MPI processes may have important implications in terms of performance [51][52], correctness [68], and memory [9][52].

Several works report scalability issues related to the management of communicators and groups [52][58][32] when the number of processes grow, affecting both performance and memory. Based on the work done by Bernholdt, et. al., communicators and groups management is a feature widely used in current applications, and it will continue to be in exascale applications [23]. Other works suggest that the time spent in communication grows when the number of processes increases [62]. Vetter, et. al. [116] studied the impact of the number of processes in the total number of messages sent, finding a relationship between them: the more processes, the more messages. In the same work, they found that, usually, increasing the number of processes also causes to send smaller messages. Traditionally, the communication library introduces a constant penalty per communication, regardless of the actual size of the message [116]. Therefore, if messages are smaller, the penalty is proportionally bigger, incurring in an increased communication overhead.

In the future, we will study if our approach is able to mitigate this issues by its ability to reduce the total number of MPI processes. Additionally, our approach can also deal better with load imbalance. Fewer processes means coarser chunks of the dataset per process. Besides that, within each process, we use a task-based

programming model which inherently copes better with load imbalance. Coarser chunks per process means more tasks per process, so our runtime system could do a better handling of the load imbalance and mitigate its effect.

Finally, another advantage of reducing the number of processes is the ability to scale to more cores. In strong scaling scenarios, it may happen that a dataset of an application cannot be split in more MPI processes [73]. With our approach, we require fewer MPI processes to use the same total core count.

5.9 Conclusions

The stagnation of single-thread performance led the industry to introduce multi-processor architectures to keep enhancing performance of processors. This trend of adding more compute units inside a single processor made frequent to find several sockets working together in a single shared-memory space by using NUMA. Nevertheless, the NUMA effect might jeopardize the performance of applications if not properly handled.

In this work, we propose the Nanos6 NUMA-aware system that provides (1) a simple memory management API to distribute data among different NUMA sockets, (2) a data tracking system that keeps track of the location of the memory regions allocated using (1), and (3) a NUMA-aware scheduler that schedules tasks in the more appropriate NUMA node, based on their data dependences and data location.

Our solution is very simple and require little effort from the user side, while reducing the NUMA effect. Our evaluation shown that we are able to outperform other state-of-the-art approaches such as the use of `numactl` across several different benchmarks in different platforms, being able to reach the optimal speedup in several of these benchmarks.

Fault Tolerance

In this chapter, we present the contributions done regarding fault tolerance. In the past chapters, we presented several contributions to enhance performance and productivity. However, all of them are meaningless if programs are not able to complete. This is something that might happen in Exascale systems due to the enormous amount of components they will include, which happen to increase the likelihood of errors.

6.1 Introduction and Motivation

Given that exascale systems are expected to be composed of a large number of components, the mean time between failures (MTBF) will drastically decrease from the order of days in petascale systems [91] to the order of hours in exascale ones [28] [2]. It is expected that exascale systems present deep and complex memory/storage hierarchies, hindering the possibility of non-expert users to exploit optimally. Considering these facts, the high-performance computing (HPC) research community is focusing on resilience and fault tolerance to mitigate the impact of system faults more easily. Accordingly, several libraries and tools are being developed that leverage low-level details and system nuances to exploit exascale systems optimally, regardless of a user's expertise. Depending on the errors those systems address, they can be application-specific [46], algorithm-specific [39], or more generic solutions [48].

One technique that is increasing in popularity is application-level checkpoint/restart (CR). The main reason is its efficiency in terms of space and performance compared to other fault-tolerance techniques. However, current approaches offering application-level CR require considerable effort from the user. Users are in charge of identifying the application state, serializing and deserializing data for checkpoints or recovery, and modifying the program flow to check whether the ex-

ecution is a restart. Additionally, moving from one system to another may require rewriting the code, at least for tuning.

Among the wide variety of tools and libraries available, three of them stand out due to their support for multi-level checkpointing: Fault Tolerance Interface (FTI) [22], Scalable Checkpoint and Restart (SCR) [77], and Very Low Overhead Checkpointing System (VeloC) [5]. These state-of-the-art libraries also provide optimized I/O capabilities and several redundancy schemes. Each of them provides its own flexible application programming interface (API). Nonetheless, compared to other techniques, such as transparent CR (which requires no participation from the user but introduces high overhead), they still demand significant work from the user.

In this chapter, we contribute to the aforementioned set of libraries and tools with an application-level CR mechanism based on compiler directives. Using compiler directives, we enhance portability and programmability. Our solution—as FTI, SCR, and VeloC— supports fail-stop errors (i.e., process abortions or hardware failures) and soft errors, although undetected errors are not tolerated.

We present the **OpenCHK** programming model [15] for C/C++ and Fortran, based on our previous work presented in [71]. Our model is based on compiler directives such as in OpenMP [84]. The sentinel used to recognize the directives and clauses of the OpenCHK model is `chk`. Currently, the model supports several clauses and directives, which are detailed in Section 6.3.1, providing users the ability to:

- Initialize and finalize the CR environment in an easy and portable way.
- Easily indicate the data to be protected.
- Specify checkpoint conditions (e.g., frequency).
- Set checkpoint properties, such as identifiers and levels.
- Select among different kinds of checkpoints (full/differential).
- Avoid the requirement of modifying the natural program flow to check whether the current execution is a restart.

6.2 Background

As introduced in the previous section, exascale systems threaten to jeopardize the successful completion of large HPC applications. Therefore, fault-tolerance systems become crucial to mitigate the impact of errors and guarantee the completion of applications. There are many HPC applications performing large simulations based on iterative algorithms, usually requiring long execution times to complete successfully. Such long execution times make them more likely to experience system faults. In fact, given the increased likelihood of system faults in the exascale era, it is possible to find applications requiring more execution time to finish than the MTBF of the system. Figure 6.1 illustrates a scenario where an application takes more than three times the system's MTBF to complete. Thus, the application is very unlikely to complete. In this kind of scenario, techniques providing resilience, like CR, become essential.

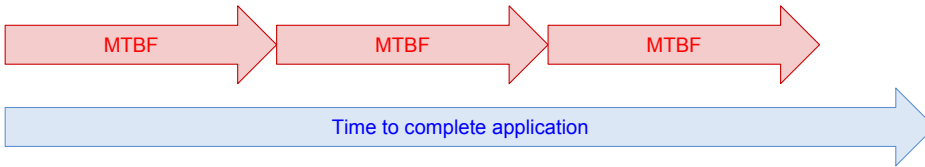


Figure 6.1: Long-running applications hardly complete when MTBF is too small.

CR is a widely used technique for saving the full state of an application in such a way that if an error occurs it can be restored, allowing the execution to continue from that point instead of from the beginning. Section 3.3 outlines several approaches to provide this functionality. Our proposal focuses on providing persistent CR. Persistent approaches store the data in a persistent way, usually in a parallel file system (PFS). If a node fails and cannot be recovered, the checkpoints are still accessible.

There are several approaches that provide CR, ranging from ad-hoc solutions where developers face low-level details (such as I/O operations), to libraries offering APIs for abstracting users from such low-level details and nuances. However, those tools still involve some difficulties, such as poor portability and complex APIs.

Regarding complexity, current approaches require users to (1) serialize/deserialize data, and (2) modify the natural program flow to check if a previous checkpoint is available for restart or if the application must run from scratch. Serialization/de-

serialization is the process of translating data structures or objects into a format that can be stored efficiently. For instance, pointers cannot be stored directly. Instead, their contents must first be retrieved and then stored. In a restart, the contents must be read from disk and then assigned to the corresponding pointers. Regarding the flow, current CR approaches require users to check for checkpoint existence explicitly and, if any exist, explicitly ask to recover data. Listing 6.1 shows a very brief example of these responsibilities. A more complete and real example can be seen in the appendices, where a full example of a real application is included using several state-of-the-art libraries, such as FTI, SCR, and VeloC. Furthermore, some approaches force users to deal directly with I/O, as in Appendix A.4.

Listing 6.1: Brief example of the serialization/deserialization process and the modification in the program flow

```
int **data;
for(int i = 0; i < N; i++){
    data[i] = (int *) malloc(i*sizeof(int));
}

// Modifying program flow
if(restart_available()) {
    // Deserializing after restart
    int *restarted_data = nullptr;
    size_t restart_size = 0;
    int id = -1;
    for(int i = 0; i < N; i++) {
        restart(&restarted_data, &restart_size, &id);
        assert(id == i);
        assert(restart_size == i);
        memcpy(data[i], restarted_data, restart_size);
    }
}
else {
    init_data(data, N);
}

// Serializing for checkpoint
int *cp_data = nullptr;
size_t cp_size = 0;
for(int i = 0; i < N; i++) {
    cp_data = data[i];
    cp_size = i;
}
```

```
    checkpoint(cp_data, cp_size, i /* id */);  
}
```

Second, the proliferation of several CR libraries with different interfaces hinders portability between systems. As there is no standard software stack for CR, it is possible that different systems offer different CR software. Thus, writing CR code using native APIs may cause portability issues when moving to a different system. In such a situation, a user's options are constrained to the following: (a) rewriting the code using the interface of the library available in the new system or (b) installing the original library. Notwithstanding, the installation of the CR libraries requires deep knowledge of the storage hierarchy for adequately tuning the installation to maximize performance. Additionally, it may require special permissions that common users do not have. In any case, both options are costly and non-trivial, affecting portability.

Using a directive-based approach, the aforementioned problems disappear. The model takes several of the user responsibilities (i.e., data serialization/deserialization and checking whether a restart is possible) and the users only need to specify which data must be checkpointed/restored in a simple way, maximizing programmability. Our solution adds a new abstraction layer with a unique interface that enables us to leverage several backend libraries, thereby enhancing portability. The enhancement of portability comes from enabling developers to use the already tuned installation present in every system without changing any code. This approach enables users to focus on applications, thereby increasing productivity and portability.

6.3 OpenCHK Model

In this section, we detail the specification of the OpenCHK programming model, including the directives and clauses supported and the functionalities provided. The aim of our model is to provide a standard way of coding CR. We offer a new level of abstraction that hides implementation details from users, enabling them to focus on the application. The approach presented in this chapter is similar to the one used in some programming models, such as OpenMP, to exploit parallelism in shared-memory environments. The rest of this section is structured as follows: first, we present the directives and clauses of the OpenCHK model, and then we detail the functionalities offered by the model.

6.3.1 Directives and Clauses

The model supports four directives. Some of these may also be annotated with clauses that can modify their semantics in some way. Details on both directives and clauses are provided as follows.

Directives

`init [clauses]`: The `init` directive defines the initialization of a checkpoint context. A checkpoint context is necessary to use the other directives. It accepts the clause:

- `comm(comm-expr)`: `comm-expr` becomes the MPI communicator that should be used by the user in the checkpoint context that is being created. This clause is mandatory.

`load(data-expr-list) [clauses]`: This directive triggers a load of the data expressed inside the parentheses. The load directive accepts the clause:

- `if(bool-expr)`: The `if` clause is used as a switch-off mechanism: the load will be ignored if the `bool-expr` evaluates to false.

`store(data-expr-list) [clauses]`: The `store` directive may request the library to save the specified data. It accepts the clauses:

- `if(bool-expr)`: The `if` clause is used as a switch-off mechanism: the store will be ignored if the `bool-expr` evaluates to false. This clause is useful for specifying the desired checkpoint frequency.
- `id(integer-expr)`: Assigns an identifier to the checkpoint. This clause is mandatory for the `store` directive. The `id` of a checkpoint is helpful for later identification of an application's progress upon failure. For instance, if users are checkpointing steps in a loop, and the `id` is the loop induction variable, users can easily infer which step was last checkpointed. However, only the last successful checkpoint is kept.
- `level(integer-expr)`: Selects the checkpoint level which is associated with where the data is stored (e.g., local node storage, PFS, etc.) and the redundancy schemes applied. This clause is mandatory for the `store` directive. Users must consider that the backend libraries provide a different number of levels and different behaviors for equivalent levels. This

is the only parameter that must be tuned depending on the underlying backend library. In a future release, we plan to make this clause optional and rely on the parameters passed in the configuration file.

- `kind(kind-expr)`: Selects the checkpoint kind. Currently, two kinds are supported. They are `CHK_FULL`, which performs a full checkpoint; and `CHK_DIFF`, which performs a differential checkpoint.

`shutdown`: Closes a checkpoint context.

Listings 6.2, 6.3, and 6.4 show how the directives and clauses are used in C/C++ and Fortran. Specifically, Listing 6.2 shows how to initialize and shut down a checkpoint context; Listing 6.3 shows how to load several types of data, ranging from simple scalars to 2-dimensional arrays, including contiguous and non-contiguous regions; and Listing 6.4 is the same as the previous listing but for storing instead of loading. However, as it is a store, we must assign an identifier and a level, as shown in the listing.

Listing 6.2: Use example of OpenCHK init and shutdown directives in C/C++ and Fortran

```
// C/C++ syntax
#pragma chk init comm(mpi_communicator)
#pragma chk shutdown

// Fortran syntax
!$chk init comm(mpi_communicator)
!$chk shutdown
```

Listing 6.3: Use example of OpenCHK load directive in C/C++ and Fortran

```
// Load a) scalar;
// b) all array elements from 0 to size-1;
// c) array2 elements from 2 to 4;
// d) 2dArray elements from 2 to 4
//     of all the rows from 0 to n-1

// C/C++ syntax
#pragma chk load(scalar, array[0;size], \
                array2[2:4], 2dArray[0;n][2:4]) \
                [if(cond)]

// Fortran syntax
```

```
!$chk load(scalar, array(0:size-1), &
&         array2(2:4), 2dArray(0:n-1)(2:4) &
&         [if(cond)])
```

Listing 6.4: Use example of OpenCHK store directive in C/C++ and Fortran

```
// Store a) scalar;
// b) all array elements from 0 to size-1;
// c) array2 elements from 2 to 4;
// d) 2dArray elements from 2 to 4
//   of all the rows from 0 to n-1

// C/C++ syntax
#pragma chk store(scalar, array[0:size], \
                 array2[2:4], 2dArray[0;n][2:4]) \
                 kind(CHK_FULL/CHK_DIFF) id(0) \
                 level(1) [if(cond)]

// Fortran syntax
!$chk store(scalar, array(0:size-1), &
&         array2(2:4), 2dArray(0:n-1)(2:4) &
&         kind(CHK_FULL/CHK_DIFF) id(0) &
&         level(1) [if(cond)])
```

To see a full example, see Appendix A.1.

6.3.2 Functionalities

Our model is intended to standardize a common interface for the different existing CR solutions, and we aim to provide the same functionalities that they all offer but in a generic way. In what follows, we explain the main functionalities supported in the OpenCHK model, and how they fit in the currently supported backend libraries.

Basic Checkpoint/Restart

As a basic functionality, OpenCHK supports checkpoint and recovery of user-defined application data using the multi-level redundancy schemes of the backend libraries. Users can define the levels and their respective checkpoint frequency as desired. Currently, OpenCHK provides a coordinated CR mechanism because

the backend libraries only support coordinated CR. However, given its flexibility, OpenCHK could provide non-coordinated CR if required.

CP-dedicated Threads

This functionality consists of spawning a thread per node that is devoted only to CR work. By doing so, work related to CR can be conducted in parallel with the application work. This feature may be useful when running on hybrid CPU-GPU systems or systems using coprocessors where the main part is executed on GPU/-coprocessors and the CPUs are idle. The idle CPU time can be used to perform CR tasks, relieving the GPU/coprocessors of doing such tasks and focusing on the actual application work. Overall, resources are better used in this way, and we can achieve performance gains. This may increase the memory pressure in some scenarios, but this effect can be mitigated using local node storage (SSD, NVMe).

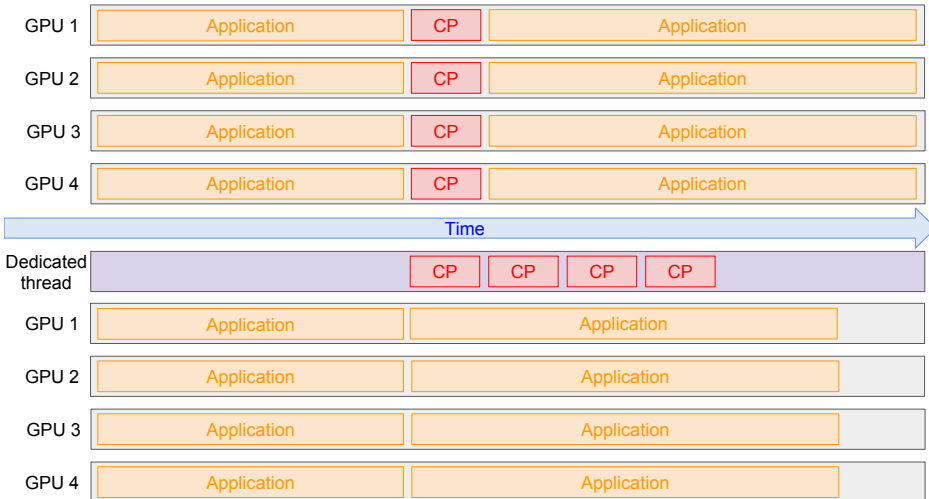


Figure 6.2: Comparison between a CP-dedicated threads scheme and a traditional scheme.

Figure 6.2 shows a comparison of an application using the traditional scheme and the same application using this CP-dedicated thread scheme. A CP-dedicated thread performs all the tasks related to fault tolerance, while the GPUs can devote their resources to the application. Up to now, this feature of the model is only supported by FTI and VeloC.

Differential Checkpointing

Differential checkpointing is a method that decreases the I/O load of consecutive checkpoints by updating only those data blocks that have changed since the last checkpoint was created. Differential checkpointing has also been called incremental checkpointing [87]. For our purposes, incremental checkpointing is a different technique that consists in building a checkpoint in pieces in several separated write operations that are performed as soon as the data is ready. We plan to support incremental checkpointing in the future. More information on incremental checkpointing can be found in Section 9.3. For a more detailed explanation of our terminology, and why we prefer these terms, please refer to [61].

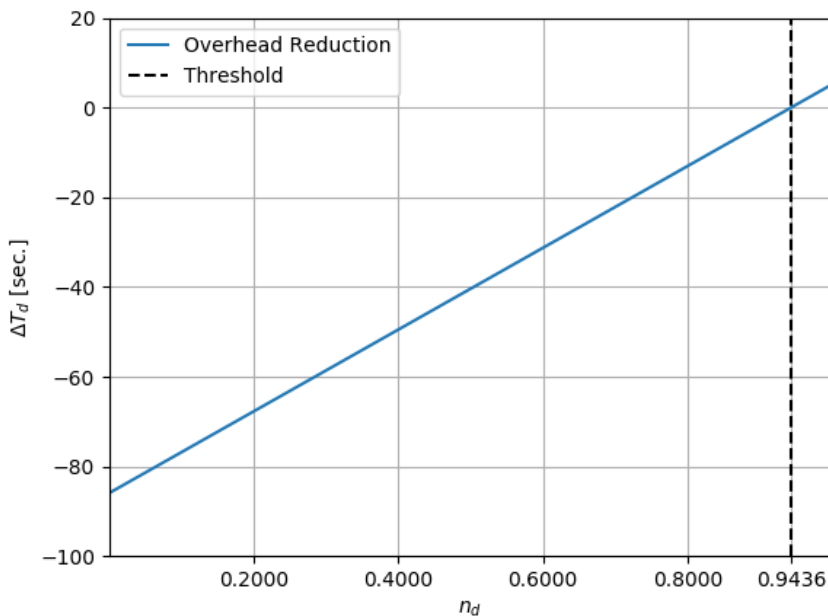


Figure 6.3: Overhead reduction with differential checkpoint for a certain scenario (2400 processes write 1 GB per process to the PFS). n_d corresponds to the ratio of dirty data blocks to protected data blocks.

Differential checkpointing uses a user-defined block size to evaluate which sections of the checkpoint have changed. This granularity is important and offers a

trade-off: smaller block sizes can capture with higher precision small changes in the dataset, which allows the backend to only update small sections and avoid having to rewrite data that has not changed. However, performing hash calculations on many small blocks and performing many small writes to the storage can reduce performance. On the other hand, large blocks are more suited to file system performance and lead to a reduction in the number of hashes to be calculated, but they also lead to more unchanged data having to be rewritten. Besides the block size, the performance of differential checkpointing also depends on the application itself. Applications that update entire datasets at every iteration are not well suited for differential checkpointing. Applications in which only parts of the dataset change, might get more benefits from differential checkpointing.

For instance, FTI has recently demonstrated [61] that applications updating less than 95% of their protected data due to changes within two consecutive checkpoints will be able to reduce their checkpoint overhead using differential checkpointing. In other words, a reduction in I/O size by as little as 5% already shows significant benefits through differential checkpointing. The overhead reduction depends linearly on the update rate. The slope of the regression characterizes this linear relationship. The ratio between the I/O rate and the hashing determines the slope. If the hashing is expensive and the I/O rate is low, the overhead reduction will be low even for low update rates, whereas, if the hashing is cheap and the I/O rate high, even high update rates could obtain a benefit. It was demonstrated for the LULESH and xPic applications that differential checkpointing can reduce the checkpointing overhead by 35% and 62%, respectively [61].

Given the model presented in [61], we can estimate the performance benefits for a certain scenario. Figure 6.3 shows the behavior of the overhead depending on the differential data ratio, n_d , corresponding to the ratio of dirty data blocks to protected data blocks. The x-axis represents the value of the differential data ratio, while the y-axis represents the increment of time that differential checkpointing introduces with respect to a common full checkpoint. Thus, positive values are additional overhead introduced by differential checkpointing with respect to a common full checkpoint, while negative values are benefit against a common full checkpoint. The presented scenario comprises 2400 processes that write 1 GB per process to the PFS. The time needed to complete a full checkpoint on all ranks is about 88 seconds. The threshold is at around 95%, which is the point where differential checkpointing and common full checkpointing introduce the same amount of overhead. If the differential data ratio is above this threshold, differential checkpointing introduces a penalty of up to 5s compared to a common full checkpoint.

In other words, if more than 95% of the data must be checkpointed, differential checkpointing introduces more overhead than a common full checkpoint. Nevertheless, if the differential data ratio is below the threshold, there is a benefit. The overhead reduction is about 9 seconds for every 10% of data that we do not have to write due to differential checkpointing. Thus, differential checkpointing becomes very quickly beneficial for updates below 95%. Of all the checkpointing libraries studied in this work, the only backend library supporting this functionality is FTI.

HDF5 support

HDF5[110] allows the structuring of datasets inside of groups and to order groups hierarchically as in a file system. The dimensionality of the datasets can also be stored inside the file. HDF5 provides a vast functionality to archive scientific data inside a file in persistent storage. In addition to this, HDF5 is optimized for both sequential and parallel I/O.

Our model allows checkpoints to be stored using the HDF5 file format. The protected datasets that serve for the successful restart are written in this format so that users can use any tool that is capable of interacting with HDF5 files for scientific analyses. Thanks to this feature, resiliency and scientific data analysis can be merged into one single I/O operation. Interacting with HDF5 files can be relatively complex and not always intuitive. Therefore, this feature to support HDF5 files enhances the flexibility of OpenCHK.

Listing 6.5 shows an example of the structure of a checkpoint file in the HDF5 format. The file contains three protected variables: Dataset_0 and Dataset_1, which are scalars, and Dataset_2, which is an array. Listing 6.6 provides two different codes able to obtain a checkpoint like the one in Listing 6.5. The first one uses OpenCHK while the second uses the native HDF5 API. Using OpenCHK, users just need to use the store directive, indicating the data to be stored along with an id and a level. Using the native HDF5 API, they must create a dataspace per variable, indicating the size and shape; create a dataset per variable indicating the data elements, layout, and some other information necessary to write, read, and interpret the data; write the data, and, finally, close the datasets and dataspace. Looking at the codes, it is possible to see how OpenCHK reduces the complexity compared to the native implementation.

Listing 6.5: HDF5 checkpoint file structure using OpenCHK. The protected data consists of 2 scalars and one array

```
GROUP "/" {
```

```

DATASET "Dataset_0" {
    DATATYPE  H5T_STD_I32LE
    DATASPACE SIMPLE { ( 1 ) / ( 1 ) }
    DATA {
        (0): 1
    }
}
DATASET "Dataset_1" {
    DATATYPE  H5T_STD_I32LE
    DATASPACE SIMPLE { ( 1 ) / ( 1 ) }
    DATA {
        (0): 1
    }
}
DATASET "Dataset_2" {
    DATATYPE  H5T_STD_I8LE
    DATASPACE SIMPLE { ( 22279025 ) / ( 22279025 ) }
    DATA {
        (0): 50, 50, 32, 115, 101, 114, 105, 97, 108, 105,

        ...

        (22279021): 0, 0, 0, 0
    }
}
}

```

Listing 6.6: Code snippet to produce an HDF5 file with a structure similar to the one shown in Listing 6.5. On top using OpenCHK and below using native HDF5 routines

```

// OpenCHK implementation
#pragma chk store(data_ptr[0], data_ptr[1], \
    data_ptr[2][0;N]) id(0) level(1)

// HDF5 Native implementation
hid_t      dataset_id[3], dataspace_id[3];
hsize_t    dims0[1], dims1[0], dims2[0];

dims0[0] = 1; dims1[0] = 1; dims2[0] = N;
dataspace_id[0] = H5Screate_simple(1, dims0, NULL);
dataspace_id[1] = H5Screate_simple(1, dims1, NULL);
dataspace_id[2] = H5Screate_simple(1, dims2, NULL);

```

```
dataset_id[0] = H5Dcreate2(file_id, "/Dataset_0", ... );
dataset_id[1] = H5Dcreate2(file_id, "/Dataset_1", ... );
dataset_id[2] = H5Dcreate2(file_id, "/Dataset_2", ... );

H5Dwrite(dataset_id[0], ... , data_ptr[0]);
H5Dwrite(dataset_id[1], ... , data_ptr[1]);
H5Dwrite(dataset_id[2], ... , data_ptr[2]);

status = H5Dclose(dataset_id[0]);
status = H5Dclose(dataset_id[1]);
status = H5Dclose(dataset_id[2]);
status = H5Sclose(dataspace_id[0]);
status = H5Sclose(dataspace_id[1]);
status = H5Sclose(dataspace_id[2]);
```

6.4 Implementation Details

This section provides some insight into the implementation details of our proposed solution. We provide our implementation of the model on top of the Mercurium C/C++ and Fortran source-to-source compiler [11] and the Transparent Checkpoint Library (TCL) [18] intermediate library. Currently, we support FTI, SCR, and VeloC as backend libraries.

Following, we detail the architecture of our implementation, the changes effected at the Mercurium compiler level, and the implementation of TCL.

6.4.1 Architecture

We have designed an implementation based on three components: a compiler (Mercurium) that translates directives and clauses into calls to an intermediate library, an intermediate library (TCL) which oversees forwarding the user-requested actions to the adequate backend library, and several backend libraries. Figure 6.4 shows our three-layer architecture. This approach allows us to extend the model to support new features as the backend libraries evolve.

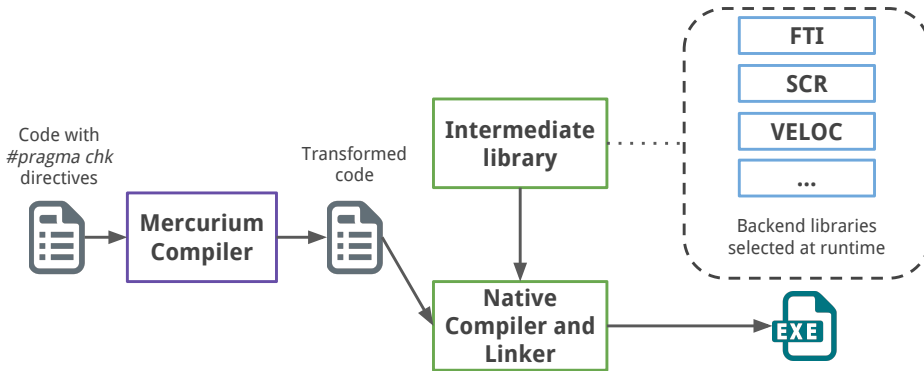


Figure 6.4: Three-layer architecture.

6.4.2 Mercurium

For supporting the OpenCHK model, Mercurium must process the OpenCHK directives and clauses to enable the application-level CR functionalities. One of the main duties of Mercurium is processing these directives and clauses to transform them into calls to TCL. Following, we detail the compiler transformations done for each of the directives and clauses.

- `chk init [clauses]`: The compiler triggers the initialization of TCL. Clauses accepted:
 - `comm(comm-expr)`: Mercurium passes `comm-expr`, which is a pointer, to TCL, which sets the MPI communicator that the user should use in the checkpoint context.
- `chk load(data-expr-list) [clauses]`: The compiler introduces a call to TCL when it finds this statement informing it of the start of a restart. Then, Mercurium performs several calls registering the data to be restarted. Accordingly, the compiler must also send some information about the data to be restored, such as the sizes and the pointers, for overwriting the current data with the recovered data. The compiler extracts all this information from the data specified and its own knowledge of the program symbols. When all the data is registered, Mercurium calls a TCL method that performs the restart. This is part of the deserialization process that would otherwise be done by the user. Additionally, this directive implies a transparent way of checking

whether a restart exists, which would also be done by a user modifying the program flow in other approaches. Clauses accepted:

- `if(bool-expr)`: The calls to TCL are only effective when the condition expressed in this clause is true. This means that none of the calls are done if the condition is not satisfied.
- `chk store(data-expr-list) [clauses]`: The compiler does very similar things as those performed for the load directive. The compiler oversees doing a part of the data serialization, equivalent to the deserialization process conducted for the load directive. The only difference is that the action being performed is a checkpoint instead of a restart, meaning some additional information must be passed. In the first call, the one that notifies a checkpoint is starting, Mercurium adds the kind, id, and level of the checkpoint. This information is extracted from the following clauses:
 - `if(bool-expr)`: The calls to TCL are only effective when the condition expressed in this clause is true. Thus, no calls are executed if the condition is not satisfied.
 - `id(integer-expr)`: The checkpoint that is being performed has the identifier set in this clause. It is mandatory to specify an identifier.
 - `level(integer-expr)`: The checkpoint that is being performed will be written at the specified level. It is mandatory to specify a level.
 - `kind(kind-expr)`: Chooses the kind of checkpoint to be done between full or differential, which are the currently supported options. The default value is full.
- `chk shutdown`: The compiler triggers the finalization of TCL.

The order specified in the load/store clauses is critical. The compiler forwards the data to TCL in the very same order set by the user when writing the load/store clauses. Thus, if the order of the data in loads and stores does not match, there may be problems when recovering data from a restart.

Also, it is important to mention that the three back-end libraries are capable of deciding the checkpoint level automatically based on the configuration file. However, our current implementation in OpenCHK does require the user to supply the checkpoint level. This is a small limitation that will be lifted in future versions of OpenCHK.

A further duty of Mercurium, which is crucial for improving code programmability, is extracting the information that must pass to TCL from the annotations done by the user. In most cases, backend libraries need the base address and the size of data to perform CR. For non-array expressions, this is just the size of the type of the data. However, for more complex data structures, like array expressions, we may need additional information, such as the accessed bounds of each dimension, the size of each dimension, and the base element type of the data structure.

All the aforementioned information required by the backend libraries is only retrievable by a tool with a full understanding of the supported programming languages—C, C++, and Fortran— such as the Mercurium compiler. Additionally, this is an error-prone task. Thus, automatizing it minimizes the possibility of error while reducing debugging time. Apart from that, it prevents developers from writing boilerplate code.

A further functionality added by Mercurium is *self-iterative data expressions*. Listing 6.7 shows an example of this. This is a kind of `for` loop inside the load/store clauses, which allows iterating over data structures instead of writing the data one by one, simplifying users' work. Self-iterative data expressions are useful in scenarios where many elements of a data structure must be checkpointed/loaded and users must write it manually. In these cases, annotating the data that must be checkpointed/restored becomes a tedious task. It also becomes error-prone, because writing the same code several times, changing only a few characters, may cause errors to be more difficult to find. Self-iterative data expressions enable users to perform this work in a much easier way.

Listing 6.7: Use example of OpenCHK self-iterative data expressions

```
// Self iterative data expression
#pragma chk store({data[i], i=0;4})

// Equivalent
#pragma chk store(data[0], data[1], data[2], data[3])
```

To adequately point out the importance of the compiler in the OpenCHK model, we have performed an analysis of the code complexity before and after the compiler transformation. For that purpose, we used the Lizard [121] tool to compute the cyclomatic complexity (CC) metric and SLOCCount [119] to calculate the development effort estimate (DEE). Table 6.1 shows the CC for a 2D heat simulation code performing CR using OpenCHK before (BT) and after (AT) compiler transformation. Moreover, it also presents the CC for the same code when using native

Table 6.1: Cyclomatic complexity of a 2D heat simulation using OpenCHK (before and after compiler transformation), FTI, SCR, and VeloC.

	BT	AT	FTI	SCR	VeloC
CC	10.0	11.0	15.0	36.0	13.0
DEE (Person-Months)	0.34	0.76	0.36	0.51	0.35

FTI, SCR, and VeloC to perform CR. As can be seen, OpenCHK remains the simplest before the compiler transformation in both metrics. After the compiler transformation, its CC grows by 1 point while its DEE more than doubles, becoming the most complex version.

The CC is higher for the native libraries because this metric is affected by the number of different paths that a program can take. For instance, each additional `if` increases the CC. The DEE, in contrast, is affected by the size of the code. In that case, it is important to highlight that Mercurium generates very verbose code when transforming code, so the size of it quickly becomes large, affecting the DEE.

6.4.3 TCL

To maximize the portability of our approach, TCL must process the information passed by Mercurium and forward it to the adequate backend library. This way, we enable users to write code agnostic from the backend library while allowing them to use any of the supported backend libraries. TCL is responsible for adequately formatting the information for each backend library and calling the appropriate methods to perform the user-requested actions, depending on the backend library chosen by the user.

Additionally, this library, in collaboration with the Mercurium compiler, serializes and deserializes the data. The serialization and deserialization process is tedious and error-prone for users. Using our mechanism, users can perform it with little or no effort.

Additionally, TCL prevents users from modifying the natural program flow to check whether a restart has to be done. The library does it transparently. If a checkpoint is available and a restart can be done, it recovers data. In consequence, codes are cleaner and more readable.

The mechanism we propose is easily extensible. Our first implementation of

TCL was providing support for only FTI and SCR. Now VeloC has been added as a third backend library to TCL, and the addition of others would be straightforward.

6.5 Evaluation and Discussion

In this section we compare the performance of our approach with natively using the backend libraries that we use in our model. The structure of this section is as follows: first, we describe the methodology used; then, we detail the environment in which the experiments were conducted, as well as the benchmarks and applications used. Finally, we provide the evaluation and discuss the results.

6.5.1 Methodology

First, the objectives of our evaluation are (1) demonstrating that our approach does not add significant overhead compared to using any of the supported backend libraries natively, and (2) showing the improvement that our mechanism provides in terms of code productivity.

For (1), we designed an experiment in which we launch a first run of an application/benchmark, with an injected fault. Then, we restart the application/benchmark from the last checkpoint until successful completion. The whole process, from the first run until successful completion, including the recovery, is timed. We take measurements using both OpenCHK with a given backend library and using the same backend library natively. Then, the OpenCHK time is divided by the native library time. To demonstrate that no significant overhead is introduced, the resulting number from the quotient should be 1 or close to 1.

Regarding (2), as there is no standard metric for measuring programmability, we have decided to consider the number of source lines of code (SLOC) required to express the CR functionality. Thus, we compare the number of code lines required using native APIs with the number of code lines required with OpenCHK.

We will use the following nomenclature for our experiments:

- FTI/SCR/VeloC. This version is an implementation performing application-level CR directly using the APIs provided by FTI, SCR or VeloC.
- OpenCHK. In this version, the application-level CR is conducted by the mechanism proposed in this work.

We obtained the results of all our experiments by averaging the execution times of 5 different runs per version.

The executions of the evaluation were run with 50 MPI processes, whenever it was possible. Nevertheless, there are some applications/benchmarks that constrain the number of processes to be used. For each of those applications, we specify the number of processes used. We consider 50 MPI processes an adequate scale for our experiments. Given that the possible sources of overhead in our approach are constant rather than dependent on the number of nodes, nothing suggests the possibility of scalability problems in larger experiments.

Some of the benchmarks/applications contain intra-node parallelism. In those, the number of threads per process is 48, whereas, for the rest, it is 1.

The runs performed for our experiments took about 10 minutes. This means that the whole process, from the first run until successful completion, including the restart, took about 10 minutes. Regarding the checkpoint frequency, we forced one checkpoint per minute, resulting in a total of ten checkpoints per run. The frequency is expressed in terms of iterations, so that we checkpoint data every 10% of the iterations of the program. This is a high checkpoint frequency, which was selected on purpose to stress the checkpointing mechanisms and ease the performance comparison between the different evaluated approaches. Coarser checkpoint frequencies should result in even lower overheads.

Regarding the faults, all of them were deterministically injected at 90% of the application progress. The faults introduced are exceptions that cause process abortion, and the degree of progress was arbitrarily chosen. The reason for an evaluation in the presence of faults is the possibility of measuring not only the overhead introduced in checkpointing but also in the restart process.

6.5.2 Environment and applications

In this subsection, we detail the environment where the experiments were run, as well as the applications and benchmarks that we used to evaluate our approach.

The experiments were carried out on a machine with the configuration given in Table 6.2. More details about the hardware can be found in [21].

The software used for our experiments, along with their versions, can be seen in Table 6.3.

¹Hash of the commit used in the experiments.

Table 6.2: Machine architecture

Component	Details
Nodes	3456
CPU	2x Intel Xeon Platinum 8160 2.1 GHz
Network	100 GB Intel Omni-Path Full-Fat Tree & 10G bit Ethernet
Memory	3240x 96 GB/node & 216x 384 GB/node (Total: 384.75 TB)
Local storage	240 GB Intel s3520 SSD
File system	14 PB GPFS disk storage
OS	Linux-SuSe Enterprise Server 12 SP2

Table 6.3: Software and versions used to perform the experiments

Software	Version
Transparent Checkpoint Library	1.0
Mercurium source-to-source compiler	2.3.0
gcc and gfortran	7.2.0
icc and ifort	17.0.4
Intel MPI	2017.3.196
SCR	1.2.2
FTI	d54a9e0 ¹

Following, we provide a brief explanation of the applications and benchmarks used in the evaluation. The size of the applications ranges from ≈ 500 to ≈ 15000 lines of code. Note that there are 7 applications using FTI, 5 applications using SCR, and 2 applications using VeloC. This is because we did not have the reference versions (native FTI/SCR/VeloC) of all the applications to compare against.

BT-MZ [79]: BT-MZ, extracted from the NAS Parallel Benchmarks, is a pseudo application that solves problems derived from CFD using a block tri-diagonal solver. This implementation contains OpenMP+MPI.

Duct [34]: This pure MPI application, from the CFD domain, performs a large eddy simulation of turbulent flow in a square duct.

GERShWIN [54]: The GERShWIN application was developed by INRIA under the umbrella of the DEEP-ER project. It studies human exposure to electromagnetic fields. To do so, it solves a system of Maxwell equations. The implementation, which contains OpenMP+MPI, presents some restrictions regarding the number of processes to run. Thus, it must be run with 48 nodes rather than 50.

Heat: This pure MPI benchmark performs a 2D heat transfer simulation.

LULESH2.0 [60]: This is a C++ OpenMP+MPI sample application from Lawrence Livermore National Laboratory that models the propagation of a Sedov blast wave. The problem is formulated using a three-dimensional unstructured mesh.

N-Body: This benchmark, which simulates a dynamical system of particles, uses OpenMP+MPI. Its implementation constrains the number of processes to run, so only 32 nodes were used.

SPECFEM3D: The SPECFEM3D application simulates seismic wave propagation using a Galerkin spectral element method. Its implementation relies on OpenMP+MPI, and presents some restrictions that force us to use only 32 nodes.

Stream [114]: Extracted from the HPC Challenge Benchmarks, the Stream benchmark measures the sustainable bandwidth and the corresponding computation rate for a simple vector kernel. It is implemented using OpenMP+MPI.

TurboRVB [35]: This pure MPI application was also developed under the umbrella of the DEEP-ER project, at SISSA. The goal of this application is to understand high-temperature superconductivity through Quantum Monte Carlo simulations.

xPic: This is a C++ OpenMP+MPI HPC application deduced from iPic3D [70]. It is designed for large scale production runs. xPic simulates space plasma in three-dimensional parallel code.

6.5.3 Evaluation and discussion

As stated previously, our evaluation covers two different aspects. We want to demonstrate that our model is introducing no significant overhead compared to using the native backend libraries directly, and we want to evaluate the programmability of our model.

Regarding the first aspect of the evaluation, Figure 6.5 shows three different charts, one for each backend library. Each of the charts shows the different applications and benchmarks executed in the x-axis, while in the y-axis shows the overhead calculated as previously described.

For the first chart starting from the left, which corresponds to FTI, it can be seen that the differences between OpenCHK and native FTI are always $< 2\%$. The worst case, TurboRVB, has a difference of 1.63% , while the rest are $< 1\%$. Moreover, the differences are always within the standard deviation of the runs, which range from $\approx 0.15\%$ to $\approx 2.6\%$, except for TurboRVB, which is $\approx 4.6\%$, so we conclude that negligible overhead is introduced by OpenCHK compared to native FTI.

The center chart, corresponding to SCR, shows differences that are always $< 0.5\%$, except for the GERSHWIN application that was 1.48% . However, this value fits within the standard deviation (1.49%), while the rest also remain within their respective standard deviation values. Therefore, the overhead introduced by OpenCHK is negligible compared to native SCR.

Finally, the right-most chart, which presents results for VeloC, exhibits differences of $< 0.5\%$. Furthermore, these values are within their corresponding standard deviations. Consequently, no significant overhead is introduced by OpenCHK compared to native VeloC.

Therefore, we can conclude that no significant overhead is introduced at all by the OpenCHK model when compared to its native counterparts.

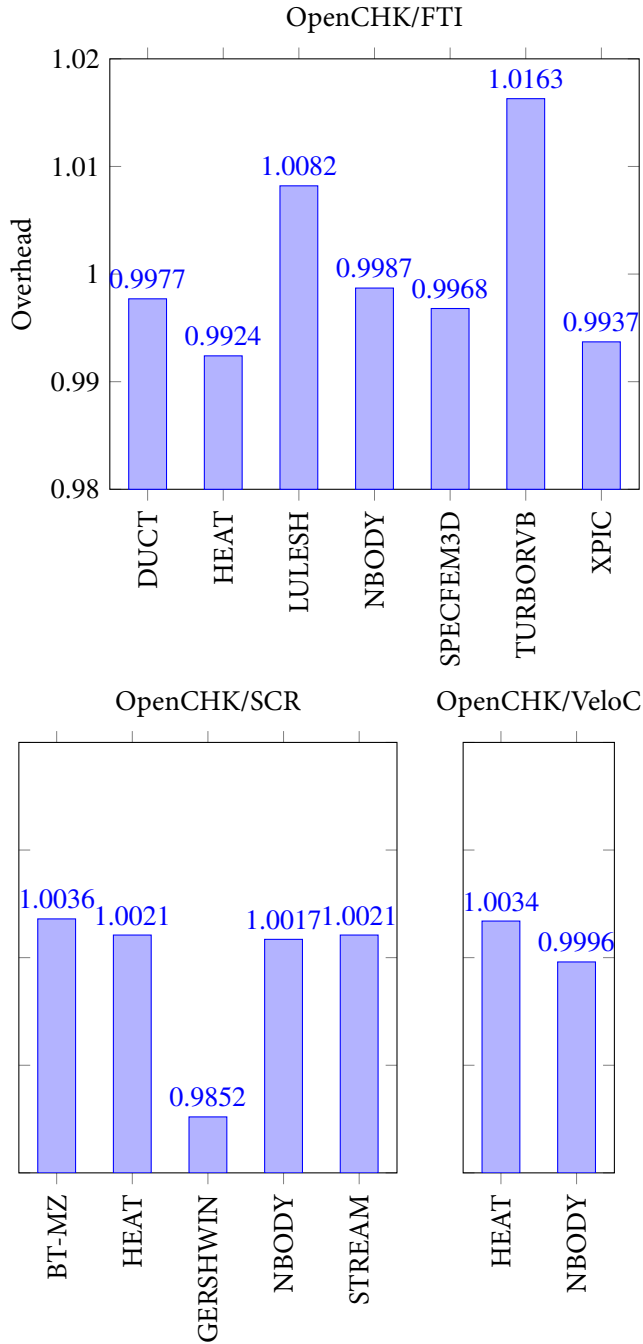


Figure 6.5: Overhead introduced by OpenCHK compared to using native FTI/SCR/VeloC

Table 6.4: Number of lines of code required to perform application-level CR using FTI and OpenCHK.

	FTI	OpenCHK	OpenCHK/FTI
DUCT	31	5	0.1613
HEAT	15	5	0.3333
LULESH	12	5	0.4167
NBODY	25	5	0.2
SPECFEM3D	28	6	0.2143
TURBORVB	80	6	0.075
XPIC	8	5	0.625
AVERAGE			0.2894

Now that we have demonstrated that our approach introduces negligible overhead, we wish to focus on the most important point of our approach: the programmability. We based our analysis on the SLOC metric, which stands for *source lines of code*. We measure it using SLOCCount [119]. We selected the lines of code needed to implement each of the different versions to make this measurement, the results of which are shown in Tables 6.4, 6.5, and 6.6 for FTI, SCR, and VeloC, respectively. Here it is possible to see the lines of code required to write application-level CR in FTI, SCR, VeloC, and OpenCHK. The code we evaluated provides the same functionality between OpenCHK and the native versions, but may not be 100% equivalent. This fact can be seen by checking the full example of code provided in the appendices. Moreover, native implementations include error handling, while OpenCHK manages errors inside the TCL library.

As can be observed in Table 6.4, our approach can drastically reduce the number of lines required to perform application-level CR. On average, the number of lines required by OpenCHK was reduced to around 30% of the lines required by FTI to provide the same functionality.

The comparison with SCR shows even better results in terms of programmability. Table 6.5 shows that OpenCHK can express CR in as little as 3% of the lines used by SCR, allowing to express a CR mechanism in five lines while SCR needs 165 lines for the same purpose. The code lines needed by OpenCHK to provide the same functionality as SCR represents, on average, only about 6% of those required by SCR.

Note that these results are for new applications that do not contain output I/O or

Table 6.5: Number of lines of code required to perform application-level CR using SCR and OpenCHK.

	SCR	OpenCHK	OpenCHK/SCR
BT-MZ	118	12	0.1017
GERSHWIN	200	8	0.04
HEAT	78	5	0.0641
NBODY	109	5	0.0459
STREAM	165	5	0.0303
AVERAGE			0.0564

Table 6.6: Number of lines of code required to perform application-level CR using VeloC and OpenCHK.

	VeloC	OpenCHK	OpenCHK/VeloC
HEAT	10	5	0.5
NBODY	23	5	0.2174
AVERAGE			0.3587

checkpointing. For those applications that already contain code to perform output I/O or checkpointing, our results are inflated, because the I/O code already exists and does not need to be added. For example, if we assume the output code already exists, SCR involves only 40 additional lines of code for the NBODY benchmark. In general, for legacy codes and applications that have already an I/O method implemented in the code, using SCR and/or VeloC can be more beneficial than using a new interface because it leverages existing code; however, for freshly developed applications or to use features of different backend checkpoint libraries, OpenCHK does provide an easy way to access those libraries through a simple interface.

As we are using VeloC in memory-based mode, the comparison results are very similar to FTI. This is because VeloC memory-based mode is much like FTI. Therefore, similarly to FTI, we only need around 36% of the lines required by VeloC to express CR. If we were using VeloC in file-based mode, as it is very similar to SCR, the comparison should be more like SCR.

In general, OpenCHK usually needs only five lines to express the entire CR code. Two lines for initialization— one for creating the MPI communicator to be passed to TCL, and one for the init directive—, another line for the load (unless there are

many variables), another line for the store (again, unless there are many variables), and, finally, another line for the shutdown directive. An additional important feature is that OpenCHK prevents users from modifying the natural program flow to check whether an execution is a restart or not. Overall, in light of the results, we can conclude that programmability is enhanced.

Finally, portability is also improved with our solution. Users can use their OpenCHK applications with whichever of the three backends supported. Consequently, moving from a system with one backend (e.g., FTI) to a system with a different backend (e.g., SCR or VeloC) requires no changes in the source code. Otherwise, if native APIs are used, the code related to CR must be completely rewritten.

6.6 Conclusions

Throughout this chapter, we have detailed the extension of a directive-based approach designed for providing application-level CR: the OpenCHK programming model. The model includes the new `#pragma chk` directive family, composed of several directives and clauses. They allow users to specify data to be checkpointed persistently, along with other details, such as checkpoint frequency, checkpoint identifier, or checkpoint level. Additionally, the model enables users to recover data from an existent checkpoint, in the case of a restart after failure, continuing the execution from the recovered state rather than from the beginning.

The directive-based approach presented in this chapter eases the use of application-level CR. The solution proposed (1) minimizes the modifications required in the source code to perform CR, (2) transfers the responsibility of serializing and deserializing data required by traditional approaches from the user side to the model side, and (3) prevents users from modifying the natural program flow to check whether data can be recovered from a checkpoint. Our solution incorporates state-of-the-art CR libraries (FTI, SCR, and VeloC) to maximize resiliency and performance, benefiting from their advanced redundancy schemes and their highly optimized I/O operations. Additionally, the OpenCHK model enables users to employ any of the backend libraries without modifying a single line of source code, thereby enhancing the portability of applications. The OpenCHK model can be combined with other programming models, such as OpenMP, MPI, and other similar programming models like OmpSs, as well as combinations of them.

Furthermore, the OpenCHK model not only supports the basic functionality but also advanced functionalities: CP-dedicated threads to reduce checkpointing

overhead in some architectures, differential checkpoints to store only the blocks of data that have been modified, saving time and space, and support for HDF5 to allow merging CR with data analytics. Moreover, given its nature, OpenCHK is easily extensible so that new features implemented in any of the backends can be added to the model. Our contribution consists not only of the model, but also an implementation. Our implementation provides robust features to help users increase their productivity, such as self-iterative data expressions, which are useful when dealing with arrays to avoid tedious and error-prone tasks.

Our evaluation, consisting of several benchmarks and production-level scientific applications, showed (1) no significant overhead compared to using the native APIs of state-of-the-art solutions such as FTI, SCR, and VeloC, and (2) a significant reduction of the required number of source code lines to perform application-level CR. On average, OpenCHK needs only 29%, 6%, and 36% of the code lines required by FTI, SCR, and VeloC, respectively, to perform the same functionalities. Finally, we enhanced portability, enabling users to choose among FTI, SCR, or VeloC at runtime, with no changes in the source code.

Combining Contributions

In this chapter, we combine all the contributions presented in this thesis in a single application. We start from a task-based Stream benchmark, and incrementally apply the different contributions of this thesis. We also evaluate the performance of the different versions. With this chapter, we show that all our contributions can work together helping to overcome different Exascale challenges.

Listing 7.1: Task-based Stream benchmark

```
// Allocate vectors
double *a = new double[N];
double *b = new double[N];
double *c = new double[N];
double scalar = 3.0;

/* --- Initialization --- */
for(size_t block = 0; block < NUM_BLOCKS; block++) {
    size_t aux = block*BSIZE;
    size_t size = aux+BSIZE > N ? N-aux : BSIZE;
    #pragma oss task out(a[aux;size], b[aux;size], c[aux;size])
    for(size_t j2=aux; j2 < aux+size; j2++) {...}
}

/** --- MAIN LOOP --- repeat test cases nTimes times --- */
for (int k=0; k<nTimes; k++) {
    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task in(a[aux;size]) out(c[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            c[j2] = a[j2];
    }
}
```

```

for(size_t block = 0; block < NUM_BLOCKS; block++) {
    size_t aux = block*BSIZE;
    size_t size = aux+BSIZE > N ? N-aux : BSIZE;
    #pragma oss task in(c[aux;size]) out(b[aux;size])
    for(size_t j2=aux; j2 < aux+size; j2++)
        b[j2] = scalar*c[j2];
}
for(size_t block = 0; block < NUM_BLOCKS; block++) {
    size_t aux = block*BSIZE;
    size_t size = aux+BSIZE > N ? N-aux : BSIZE;
    #pragma oss task in(a[aux;size], b[aux;size]) out(c[aux;size])
    for(size_t j2=aux; j2 < aux+size; j2++)
        c[j2] = a[j2]+b[j2];
}
for(size_t block = 0; block < NUM_BLOCKS; block++) {
    size_t aux = block*BSIZE;
    size_t size = aux+BSIZE > N ? N-aux : BSIZE;
    #pragma oss task in(b[aux;size], c[aux;size]) out(a[aux;size])
    for(size_t j2=aux; j2 < aux+size; j2++)
        a[j2] = b[j2]+scalar*c[j2];
}
}
#pragma oss taskwait

delete a;
delete b;
delete c;

```

We selected the Stream benchmark because it is a simple and well known benchmark broadly used to measure the memory bandwidth of systems. Listing 7.1 shows the baseline code of the stream benchmark. It is parallelized using tasks with blocking. From this version, we will apply the different contributions presented throughout this thesis. Firstly, we will apply the `for` clause, presented in Chapter 4, to replace regular tasks by worksharing tasks. With this small change, we are able to mitigate granularity issues. The version using `task for` is shown in Listing 7.2, with the changes highlighted in blue text.

Listing 7.2: Stream benchmark using worksharing tasks

```

// Allocate vectors
double *a = new double[N];
double *b = new double[N];

```

```

double *c = new double[N];
double scalar = 3.0;

/* --- Initialization --- */
for(size_t block = 0; block < NUM_BLOCKS; block++) {
    size_t aux = block*BSIZE;
    size_t size = aux+BSIZE > N ? N-aux : BSIZE;
    #pragma oss task for out(a[aux;size], b[aux;size], c[aux;size])
    for(size_t j2=aux; j2 < aux+size; j2++) {...}
}

/** --- MAIN LOOP --- repeat test cases nTimes times --- */
for (int k=0; k<nTimes; k++) {
    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task for in(a[aux;size]) out(c[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            c[j2] = a[j2];
    }

    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task for in(c[aux;size]) out(b[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            b[j2] = scalar*c[j2];
    }

    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task
            for in(a[aux;size], b[aux;size]) out(c[aux;size])
        for(size_t j2=aux; j2 < aux+size; j2++)
            c[j2] = a[j2]+b[j2];
    }

    for(size_t block = 0; block < NUM_BLOCKS; block++) {
        size_t aux = block*BSIZE;
        size_t size = aux+BSIZE > N ? N-aux : BSIZE;
        #pragma oss task
            for in(b[aux;size], c[aux;size]) out(a[aux;size])

```

```

        for(size_t j2=aux; j2 < aux+size; j2++)
            a[j2] = b[j2]+scalar*c[j2];
    }
}
#pragma oss taskwait

delete a;
delete b;
delete c;

```

Listing 7.3 shows some more differences compared to the baseline code. Thanks to the use of taskloop, we can remove the blocking technique, reducing the overall lines of source code. Note that in this version, there is a single loop per kernel rather than a nested loop as in the previous versions. Again, we highlight the differences using blue text.

Listing 7.3: Stream benchmark using taskloop

```

// Allocate vectors
double *a = new double[N];
double *b = new double[N];
double *c = new double[N];
size_t j;
double scalar = 3.0;

/* --- Initialization --- */
#pragma oss taskloop grainsize(BSIZE) out(a[j], b[j], c[j])
for(j = 0; j < N; j++) {...}

/* --- MAIN LOOP --- repeat test cases nTimes times --- */
for (int k=0; k<nTimes; k++) {
    #pragma oss taskloop grainsize(BSIZE) in(a[j]) out(c[j])
    for(j = 0; j < N; j++)
        c[j] = a[j];

    #pragma oss taskloop grainsize(BSIZE) in(c[j]) out(b[j])
    for(j = 0; j < N; j++)
        b[j] = scalar*c[j];

    #pragma oss taskloop grainsize(BSIZE) in(a[j], b[j]) out(c[j])
    for(j = 0; j < N; j++)
        c[j] = a[j]+b[j];
}

```



```

#pragma omp taskloop grainsize(BSIZE) in(b[j], c[j]) out(a[j])
for(j = 0; j < N; j++)
    a[j] = b[j]+scalar*c[j];
}
#pragma omp taskwait

delete a;
delete b;
delete c;

```

Listing 7.4 shows a version of the Stream benchmark parallelized using the `taskloop for` construct. In this case, as with the `taskloop` we are able to remove the blocking, having a single loop per kernel. However, adding the `for` clause to the `taskloop` construct, it creates worksharing tasks rather than regular tasks. So, in this version, we can improve the programmability thanks to the use of the `taskloop` construct, and we are able to mitigate granularity issues by using worksharing tasks.

Listing 7.4: Stream benchmark using `taskloop for`

```

// Allocate vectors
double *a = new double[N];
double *b = new double[N];
double *c = new double[N];
size_t j;
double scalar = 3.0;

/* --- Initialization --- */
#pragma omp taskloop for grainsize(BSIZE) out(a[j], b[j], c[j])
for(j = 0; j < N; j++) {...}

/* --- MAIN LOOP --- repeat test cases nTimes times --- */
for (int k=0; k<nTimes; k++) {
    #pragma omp taskloop for grainsize(BSIZE) in(a[j]) out(c[j])
    for(j = 0; j < N; j++)
        c[j] = a[j];

    #pragma omp taskloop for grainsize(BSIZE) in(c[j]) out(b[j])
    for(j = 0; j < N; j++)
        b[j] = scalar*c[j];

    #pragma omp taskloop for grainsize(BSIZE) in(a[j], b[j]) out(c[j])

```

```

    for(j = 0; j < N; j++)
        c[j] = a[j]+b[j];

    #pragma oss taskloop for grainsize(BSIZE) in(b[j], c[j]) out(a[j])
    for(j = 0; j < N; j++)
        a[j] = b[j]+scalar*c[j];
}
#pragma oss taskwait

delete a;
delete b;
delete c;

```

Up to this point, we applied the different contributions presented in Chapter 4. In Chapter 5, we presented our contribution related to the management of NUMA domains. We can apply the mentioned NUMA support to any of the versions already listed. For that purpose, we only require to replace the regular allocation-s/deallocations by the Nanos6 NUMA API allocations/deallocations. So we did in Listing 7.5, where we combine the `taskloop for` with the NUMA support. Thanks to this, we can mitigate the impact of the NUMA effect in systems with several NUMA domains.

Listing 7.5: Stream benchmark using `taskloop for` with NUMA support

```

// Allocate vectors
nanos6_bitmask_t bitmask;
nanos6_bitmask_set_wildcard(&bitmask, NUMA_ALL_ACTIVE);
int numa_nodes = (int) nanos6_count_setbits(&bitmask);
size_t size = sizeof(double)*N;
size_t block_size = std::ceil((double) size/(double) numa_nodes);
double *a = (double *) nanos6_numa_alloc_block_interleave(size, &bitmask,
    block_size);
double *b = (double *) nanos6_numa_alloc_block_interleave(size, &bitmask,
    block_size);
double *c = (double *) nanos6_numa_alloc_block_interleave(size, &bitmask,
    block_size);
size_t j;
double scalar = 3.0;

/* --- Initialization --- */
#pragma oss taskloop for grainsize(BSIZE) out(a[j], b[j], c[j])
for(j = 0; j < N; j++) {...}

```

```

/** --- MAIN LOOP --- repeat test cases nTimes times --- */
for (int k=0; k<nTimes; k++) {
    #pragma oss taskloop for grainsize(BSIZE) in(a[j]) out(c[j])
    for(j = 0; j < N; j++)
        c[j] = a[j];

    #pragma oss taskloop for grainsize(BSIZE) in(c[j]) out(b[j])
    for(j = 0; j < N; j++)
        b[j] = scalar*c[j];

    #pragma oss taskloop for grainsize(BSIZE) in(a[j], b[j]) out(c[j]
    ])
    for(j = 0; j < N; j++)
        c[j] = a[j]+b[j];

    #pragma oss taskloop for grainsize(BSIZE) in(b[j], c[j]) out(a[j]
    ])
    for(j = 0; j < N; j++)
        a[j] = b[j]+scalar*c[j];
}
#pragma oss taskwait

nanos6_numa_free(a);
nanos6_numa_free(b);
nanos6_numa_free(c);

```

Finally, in Chapter 6, we presented the OpenCHK model. This model provides checkpoint/restart through compiler directives. Listing 7.6 includes OpenCHK directives to introduce checkpoint/restart in the application. Concretely, we introduced a directive to initialize the OpenCHK model, a directive to restore data from a checkpoint in the event of a restart, a directive to store data, and finally, a directive to finalize the OpenCHK runtime library. These changes are highlighted in blue. As a result, we have an application combining all the contributions presented in this thesis.

Listing 7.6: Stream benchmark using taskloop for with NUMA support and OpenCHK

```

// Init MPI and OpenCHK
MPI_Init_thread(&argc, &argv, DESIRED_THREAD_LEVEL, &provided);
assert(provided == DESIRED_THREAD_LEVEL);
MPI_Comm comm = MPI_COMM_WORLD;

```

```

#pragma chk init comm(comm)

// Allocate vectors
nanos6_bitmask_t bitmask;
nanos6_bitmask_set_wildcard(&bitmask, NUMA_ALL_ACTIVE);
int numa_nodes = (int) nanos6_count_setbits(&bitmask);
size_t size = sizeof(double)*N;
size_t block_size = std::ceil((double) size/(double) numa_nodes);
double *a = (double *) nanos6_numa_alloc_block_interleave(size, &
    bitmask, block_size);
double *b = (double *) nanos6_numa_alloc_block_interleave(size, &
    bitmask, block_size);
double *c = (double *) nanos6_numa_alloc_block_interleave(size, &
    bitmask, block_size);
size_t j;
double scalar = 3.0;

/* --- Initialization --- */
#pragma oss taskloop for grainsize(BSIZE) out(a[j], b[j], c[j])
for(j = 0; j < N; j++) {...}

// Try to recover data from checkpoint
#pragma chk load(a[0;N], b[0;N], c[0;N])

/* --- MAIN LOOP --- repeat test cases nTimes times --- */
for (int k=0; k<nTimes; k++) {
    #pragma oss taskloop for grainsize(BSIZE) in(a[j]) out(c[j])
    for(j = 0; j < N; j++)
        c[j] = a[j];

    #pragma oss taskloop for grainsize(BSIZE) in(c[j]) out(b[j])
    for(j = 0; j < N; j++)
        b[j] = scalar*c[j];

    #pragma oss taskloop for grainsize(BSIZE) in(a[j], b[j]) out(c[j
    ])
    for(j = 0; j < N; j++)
        c[j] = a[j]+b[j];

    #pragma oss taskloop for grainsize(BSIZE) in(b[j], c[j]) out(a[j
    ])
}

```

```

for(j = 0; j < N; j++)
    a[j] = b[j]+scalar*c[j];

#pragma oss task in(a[0;N], b[0;N], c[0;N])
{
    // Save data at half the execution
    #pragma chk store(a[0;N], b[0;N], c[0;N]) level(0) id(k) if (k ==
        nTimes/2)
}
}
#pragma oss taskwait

nanos6_numa_free(a);
nanos6_numa_free(b);
nanos6_numa_free(c);

// Shutdown OpenCHK and MPI
#pragma chk shutdown
MPI_Finalize();

```

We evaluated the different versions to check that the interaction between the different contributions works as expected. Figure 7.1 shows the performance of the different versions. Y-axis shows the performance, which in this case is the memory bandwidth, and x-axis shows the number of NUMA nodes used. The title of the chart includes the problem size, block size, and number of iterations used. However, for the versions including OpenCHK we had to increment the number of iterations to mitigate the impact of the checkpointing in the performance. In the case of the versions including OpenCHK, the execution time was around ten minutes, with a single checkpoint. In the chart we can see that all the versions perform very similarly running with a single NUMA node, with a small penalty for the version using OpenCHK, due to the checkpointing. When using two NUMA nodes, all the versions perform very similarly, except the two using the NUMA support. The versions using the NUMA support are able to outperform the rest. As with a single NUMA node, we can see a small penalty in the version using OpenCHK due to the checkpointing.

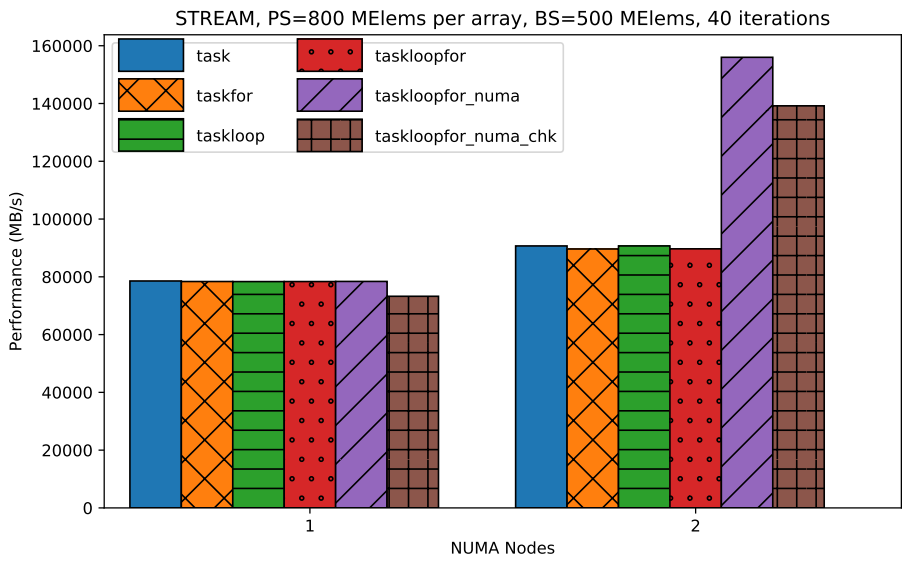


Figure 7.1: Evaluation of the combination of all the thesis contributions in platform A

Conclusions

In this chapter, we summarize the contributions presented throughout this thesis and provide concluding remarks. We also discuss the interactions between the different contributions.

8.1 Conclusions

This thesis focuses on easing the exploitation of Exascale systems. HPC systems are increasingly more and more complex, and so, becoming more and more difficult to program. Programming models are a good approach to reduce the complexity of programming such complex systems. Nevertheless, state-of-the-art programming models are still missing key features related to both performance and programmability. Specifically, the efficient management of massive compute resources and memory hierarchies, concretely NUMA domains, are still open challenges. Additionally, we must consider the necessity of fault tolerance in Exascale systems, given its enormous amount of components. HPC community has been doing a great job in fault tolerant systems and software, but there is still room for improvement in terms of programmability and portability.

The contributions of this thesis address the three challenges previously outlined, always bearing in mind the final objective: ease the use of complex HPC systems, enabling non-expert users to exploit the full power of the systems.

In Chapter 4, we address the issue of efficiently managing a modern multicore/-manycore processor. We introduce the problems that current approaches present, namely rigid synchronization in fork-join parallelism, and granularity issues in task-based parallelism. We propose a solution combining the best of both approaches: worksharing tasks. This special kind of tasks enable users to benefit from the data-flow lightweight synchronization of tasks, along with the low-overhead of worksharing techniques. In this chapter, we also present an enhancement to the

taskloop construct that simplifies the use of task-based parallelism, by enabling the use of data dependences in the taskloop construct. Finally, we combine both contributions in a single construct: `taskloop for`. This construct behaves as a regular taskloop but creating worksharing tasks rather than regular tasks. In the light of results, we can conclude that worksharing tasks are able to mitigate granularity issues, converting task-based parallelism in the best alternative in modern HPC systems; and the introduction of data dependences in the taskloop construct eases the development of task-based programs.

In Chapter 5, we cover the management of memory hierarchies, focused on NUMA domains. The Memory Wall is a well known issue in the HPC community. The processor performance increased much more than memory performance, to the point that memory accesses are a frequent bottleneck in applications. Chip manufacturers have been including several layers of memory, known as memory hierarchies, to mitigate the impact of the Memory Wall. Nevertheless, the management of the memory hierarchy is crucial to avoid spending too much time on data movements. We focus our work on mitigating the NUMA effect. Current processors may include several sockets in a single shared-memory address space, and multiple memory controllers in a single socket accessible from all the cores within the socket. Notwithstanding, accessing to remote sockets or memory controllers is more expensive than local accesses. This is a frequent bottleneck in memory-bound workloads. Analyzing the results of our evaluation, our NUMA-aware mechanism, that combines a data distribution API, a data-tracking system, the data dependences of tasks, and a NUMA-aware scheduler, is able to reduce data motion, mitigating the impact of the NUMA effect, and so, enhancing performance.

Finally, in Chapter 6 we focus on fault tolerance. It has been demonstrated that more components lead to more errors. Modern HPC systems incorporate a vast amount of components, and Exascale systems are expected to contain even more. This trend causes that errors are more likely to occur in Exascale systems, jeopardizing the progress of applications. Throughout the thesis we have contributed with mechanisms to enhance the performance (and programmability) of applications in modern HPC systems. Nonetheless, these contributions have no value if applications cannot progress. There are several state-of-the-art tools addressing fault tolerance issues, but there is still room for improvement in terms of programmability and portability. We propose the use of OpenCHK as a standard programming model to perform application-level CR. Our evaluation demonstrated that it introduces no additional overhead, while it reduces significantly the source lines of

code required to introduce fault tolerance in applications. Additionally, it enhances portability.

We would like to highlight the ability of our contributions to cooperate in a single execution, as shown in Chapter 7. We already mentioned that modern processors are increasingly including more cores and more memory controllers. Thus, we must be able to address both challenges at the same time. Our contributions regarding the efficient management of modern multicore/manycore processors and memory hierarchies are both developed in the OmpSs-2 programming model. As a result, they can be easily combined. We just have to parallelize the application using worksharing tasks and/or taskloop with dependences, while performing the memory management using the Nanos6 NUMA API. Given that both worksharing tasks and taskloops have data dependences like regular tasks, the Nanos6 NUMA-aware scheduling will be effective with no further effort. This combination could be especially interesting in hybrid MPI+OmpSs-2 applications, where we could be able to reduce the number of MPI processes per node because we would not have to worry about the NUMA effect.

Besides the challenges within the node (increase in the core count per node and the memory domains per node), we also mentioned the increase in the overall number of nodes and its corresponding increased error rate. So, fault tolerance should also be combined with the solutions to the previous challenges. The OpenCHK model can be combined with different programming models with no trouble. In fact, some of the applications used in the evaluation are using MPI+OpenMP or MPI+OmpSs implementations, using both fork-join parallelism and task-based parallelism. As our previous contributions rely on task-based parallelism, there would be no trouble in combining all our contributions in a single execution as shown in Chapter 7.

We wish to highlight also that all the contributions of this thesis have been introduced, or are in process of being introduced, in the production version of the OmpSs-2 programming model, except the OpenCHK programming model that is an independent programming model that has been already released and is publicly available. In other words, this is not a work focused only on research environments that never impact the real world, but a work that is (or will be in the near future) available to everyone. In fact, some of the contributions developed in this thesis have already been used in different EU projects such as DEEP-ER, DEEP-EST and INTERTWinE. Additionally, considering the presence of the BSC in the OpenMP board, and the background of contributions from the BSC to the OpenMP standard, especially to the tasking model, some of the developments of

this thesis could influentiate or serve as a base to new features introduced by the BSC in the OpenMP standard.

Future Work

In this chapter, we discuss possible future work regarding the concrete contributions presented and this thesis, and also more general suggestions to improve the exploitation of future Exascale systems.

9.1 Efficient Management of Modern Multicore/Manycore Processors

Regarding the efficient management of modern multicore/manycore processors, we presented two contributions in this thesis: worksharing tasks and taskloop with dependences. Besides that, we also presented the `taskloop for construct`, that combines both contributions. Nevertheless, there is still room for improvement.

Concretely, in Sections 4.6.1 and 4.6.2 we mentioned that OpenMP worksharings were outperforming worksharing tasks in scenarios where the problem size is very small. We explained that this effect was due to the better exploitation of data locality in OpenMP worksharing constructs compared to worksharing tasks. Thus, as an open line for the future, we should figure out strategies to improve the locality exploitation of worksharing tasks that could boost the performance in scenarios like the one detailed. One possible solution would be implementing a work partition and scheduling similar to the OpenMP worksharings (using the static schedule) one. A further future line regarding worksharing tasks is the flexibility in the number of cores that can collaborate in a single worksharing task. We explained in Section 4.3.6 that the teams of worksharing tasks are static, in the sense that they are set at the beginning of the execution and they cannot change. As well, we stated that all the teams have the same maximum size. Future work possibilities could include the possibility of having teams of different sizes, or even the ability to change the maximum size of teams dynamically during the execution.

In Section 4.4.2 we stated that our current implementation of the taskloop with dependences focuses on programmability and mention the possibility of applying optimizations related to the memory management and throttle. In our current implementation of the taskloop, the runtime library allocates space for each of the tasks individually. However, given that we know the total number of iterations and the grainsize, we can infer the total number of tasks that we must create. One possible optimization for the future would be allocate the space for all the tasks in a single allocation rather than performing one allocation per task. This could have impact in the performance given that grouping allocations is a well known technique to reduce overhead. Besides that, knowing the total number of tasks that a taskloop contains, we can easily apply throttle techniques to prevent memory overuse that may lead to performance degradation. In some scenarios, the runtime has too many tasks in flight, consuming too much memory. Throttle mechanisms seek to correct this behaviour by limiting the ability of the runtime to keep creating tasks until the memory use is below a given threshold, but it is usually a reactive mechanism. By using taskloops, we can know how many tasks are being created by each taskloop, and we can limit the creation of tasks before the memory use is too high that it leads to a degradation of the performance.

9.2 Efficient Management of Memory Hierarchies (NUMA)

We presented our OmpSs-2 NUMA-aware system to address the challenge related to the efficient management of memory hierarchies, focused on NUMA domains. One possibility of improvement is introducing more *wildcards*, if necessary.

A further future work could be adding the possibility of retrieving some key information of the topology of the system using the Nanos6 NUMA API. Currently, we already provide a method to retrieve the number of NUMA nodes available in the system, but it could be extended to retrieve more advanced information like the distances between different numa nodes.

Another open line for the future could be the automation of the conversion of regular mallocs/frees to methods of the Nanos6 NUMA API. One possibility could be the interception of these methods with wrappers that calls the Nanos6 NUMA API rather than the regular memory management library.

One more possibility for future work is considering page migration techniques. We could use PEBS sampling to analyze the memory behavior of applications, and under some scenarios, we could use page migration techniques, such as the

`move_pages` method, to improve the distribution of data. Currently, the user is in charge of specifying the data distribution, but it would be interesting to investigate the automation of such a process. A possible way of doing so is the combination of PEBS sampling to detect patterns and `move_pages` to dynamically move data across different NUMA nodes.

In a more general view, we plan to extend the NUMA-aware system to support CUDA accelerators. As `OmpSs-2` is a programming model with heterogeneity support, we plan to provide an allocation/deallocation API similar to the NUMA API, and extend the scheduling system to work with CUDA devices.

Besides that, we are working on an extension that enables us to track data not only at NUMA level, but also at deeper levels of the memory hierarchy, such as L2 or L3. Nevertheless, this is much more complicated because it requires to mimic the behavior of the cache replacement policy.

9.3 Fault Tolerance

With regard to fault tolerance, we presented the `OpenCHK` model. This model provides the basic features of an application-level CR mechanism, and some advanced features such as differential checkpoint. As future work, we would like to integrate incremental checkpoints into `OpenCHK`. This is a technique where a checkpoint is not fully written at one time, but incrementally built in several separated write operations. An example of this is an N-body simulation dealing with particle positions, velocities, and forces. Each one of these is calculated at a different time, starting with the forces, then the velocities, and finally the positions. When the forces have been updated, they can be written in the checkpoint, possibly while the velocities are being calculated. Then, when the velocities have been updated, they can be written in the checkpoint, and the same finally with the positions. Overall, all the variables are checkpointed, but the write operations are separated in time, to decrease storage congestion and maximize parallelization.

Another idea is decoupling the actual operation (load/store) and the data registration. Currently, the model does these together because the data is registered in the load/store clauses. However, it may become a problem when dealing with C++ classes due to the visibility of some members in different contexts. Therefore, allowing registration and actual load/store separately would help in some specific cases.

Finally, we plan to add GPU checkpointing to the model to accelerate fault-tolerance tasks and better exploit the resources of heterogeneous systems.

9.4 Other Challenges

In this thesis, we focused on three specific challenges of the future Exascale systems, and developed solutions to overcome them. In the previous sections of this chapter, we discussed concrete proposals to improve the contributions presented in this thesis or future work lines that are specifically related to the challenges addressed in this thesis. Nonetheless, Exascale systems present more challenges than the three addressed in this thesis.

One of them is the heterogeneity. Modern systems are increasingly incorporating accelerators within the compute nodes. Nowadays it is frequent to see compute nodes with CPUs and GPUs, or other specific purpose accelerators. Usually, the accelerators offer a great compute capacity with a reduced power budget, but they are not always easy to program and handle. One important problem of using accelerators is the presence of disjoint memory address spaces where the user is in charge of properly handling the data movements between different address spaces. As well, hardware accelerators usually have different architectures than CPUs in the same node, making the cooperation difficult. Great efforts are being done in this direction, but there is still room for improvement.

One further challenge of Exascale systems is the management of deep and complex memory hierarchies. New layers have been introduced in the traditional memory hierarchy. Traditionally, we had some levels of cache (2 or 3 usually), the main memory, and the storage. Nowadays, we can find a further layer between the caches and the main memory, such as the HBM of the Intel's Knights Landing processor. Also, there is a layer between the traditional storage and the main memory where we can find Non-Volatile Memories (NVM) or Storage Class Memories (SCM). To maximize the performance of modern HPC systems, it is crucial to properly manage the whole memory hierarchy, including the newest layers. As with the heterogeneity, the HPC community is devoting considerable effort in this challenge, but again, there is still room for improvement. The integration of such new layers in state-of-the-art programming models could be great idea.

APPENDICES

Using CR in N-body Simulation

In this Appendix, we provide full examples of how to introduce CR in an N-body simulation kernel, using OpenCHK, FTI, VeloC, and SCR. In each of the codes, the lines highlighted with a blue background are those required to implement CR.

A.1 Full Example of N-body Simulation Kernel Using OpenCHK

```
void solve_nbody(particles_block_t * local,
                particles_block_t * tmp,
                force_block_t * forces,
                const int n_blocks,
                const int timesteps,
                const float time_interval)
{
    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    int t = 0;

    // Load local and t vars, if any.
    #pragma chk load ([n_blocks] local, t)

    for (; t < timesteps; t++) {
        #pragma oss task inout([n_blocks] local)
        {
            // Store local and t vars, using t as id.
            // Each checkpoint done must be at level 4.
            // Checkpoint every 10 iterations.
            #pragma chk store ([n_blocks] local, t) id (t) level (4)
            \
            kind (CHK_FULL) if (t (timesteps/10)==0 && t
                != 0)
        }

        particles_block_t * remote = local;
        for(int i=0; i < rank_size; i++){
            #pragma oss task in([n_blocks] local,[n_blocks] remote)
            \
                inout([n_blocks] forces)
            calculate_forces(forces, local, remote, n_blocks);

            #pragma oss task in([n_blocks] remote) \
                out([n_blocks] tmp)
```

```
        exchange_particles(remote, tmp, n_blocks, rank,
                          rank_size, i, t);

        remote=tmp;
    }

    #pragma oss task inout([n_blocks] local) inout([n_blocks]
        forces)
    update_particles(n_blocks, local, forces, time_interval);
}
#pragma oss taskwait
}
```

A.2 Full Example of N-body Simulation Kernel Using FTI

```
#include <fti.h>

void solve_nbody(particles_block_t * local,
                particles_block_t * tmp,
                force_block_t * forces,
                const int n_blocks,
                const int timesteps,
                const float time_interval)
{
    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    int t = 0;
    // Create a new FTI data type
    FTIT_type ckptInfo;
    // Initialize the new FTI data type
    FTI_InitType(&ckptInfo, n_blocks*sizeof(particles_block_t));
    FTI_Protect(0, &t, sizeof(int), FTI_INTG);
    FTI_Protect(1, local, 1, ckptInfo);

    if( FTI_Status() ) {
        FTI_Recover();
    }

    for (; t < timesteps; t++) {
        #pragma oss task inout([n_blocks] local)
        {
            if(t ( timesteps/10)==0 && t != 0 && restarted_t != 0 &&
                t!=restarted_t) {
                int res = FTI_Checkpoint( t, 4 );
                if( res != FTI_DONE ) {
                    printf( "FTI internal error." );
                    MPI_Abort( MPI_COMM_WORLD, -1 );
                }
            }
        }
    }
}
```

```

    }
}

particles_block_t * remote = local;
for(int i=0; i < rank_size; i++){
    #pragma oss task in([n_blocks] local,[n_blocks] remote)
        \
            inout([n_blocks] forces)
    calculate_forces(forces, local, remote, n_blocks);

    #pragma oss task in([n_blocks] remote) \
        out([n_blocks] tmp)
    exchange_particles(remote, tmp, n_blocks, rank,
        rank_size, i, t);

    remote=tmp;
}

#pragma oss task inout([n_blocks] local) inout([n_blocks]
    forces)
update_particles(n_blocks, local, forces, time_interval);
}

#pragma oss taskwait
}

```

A.3 Full Example of N-body Simulation Kernel Using Veloc

```
#include <veloc.h>

void solve_nbody(particles_block_t * local,
                particles_block_t * tmp,
                force_block_t * forces,
                const int n_blocks,
                const int timesteps,
                const float time_interval)
{
    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    int t = 0;
    VELOC_Mem_protect(0, &t, 1, sizeof(int));
    VELOC_Mem_protect(1, local, n_blocks, sizeof(particles_block_t)
);

    int restarted_version = VELOC_Restart_test("nbody", t);
    if(restarted_version != VELOC_FAILURE) {
        assert(VELOC_Restart("nbody", restarted_version) ==
VELOC_SUCCESS);
    }
    for (; t < timesteps; t++) {
        #pragma oss task inout([n_blocks] local)
        {
            if(t (timesteps/10)==0 && t != 0 && restarted_t != 0 &&
t!=restarted_t) {
                int res = VELOC_Checkpoint("nbody", t);
                if( res != VELOC_SUCCESS ) {
                    printf( "VELOC internal error." );
                    MPI_Abort( MPI_COMM_WORLD, -1 );
                }
            }
        }
    }
}
```

```

}

particles_block_t * remote = local;
for(int i=0; i < rank_size; i++){
    #pragma oss task in([n_blocks] local,[n_blocks] remote)
        \
            inout([n_blocks] forces)
        calculate_forces(forces, local, remote, n_blocks);

    #pragma oss task in([n_blocks] remote) \
        out([n_blocks] tmp)
    exchange_particles(remote, tmp, n_blocks, rank,
        rank_size, i, t);

    remote=tmp;
}

#pragma oss task inout([n_blocks] local) inout([n_blocks]
    forces)
update_particles(n_blocks, local, forces, time_interval);
}

#pragma oss taskwait
}

```

A.4 Full Example of N-body Simulation Kernel Using SCR

```
#include <scr.h>

void solve_nbody_cp(const int n_blocks,
                   const int rank,
                   particles_block_t const* __restrict__ local,
                   const int timestep);
int solve_nbody_rt(const int n_blocks,
                  const int rank,
                  particles_block_t* __restrict__ local,
                  int *timestep);

void solve_nbody(particles_block_t * local,
                particles_block_t * tmp,
                force_block_t * forces,
                const int n_blocks,
                const int timesteps,
                const float time_interval)
{
    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    int t = 0;
    solve_nbody_rt(n_blocks, rank, local, &t);
    for (; t < timesteps; t++) {
        #pragma oss task inout([n_blocks] local)
        {
            if(t (timesteps/10)==0 && t != 0 && restarted_t != 0 &&
                t!=restarted_t) {
                solve_nbody_cp(n_blocks, rank, local, t);
            }
        }

        particles_block_t * remote = local;
        for(int i=0; i < rank_size; i++){
            #pragma oss task in([n_blocks] local,[n_blocks] remote)
            \

```



```

        inout([n_blocks] forces)
calculate_forces(forces, local, remote, n_blocks);

#pragma oss task in([n_blocks] remote) \
        out([n_blocks] tmp)
exchange_particles(remote, tmp, n_blocks, rank,
        rank_size, i, t);

remote=tmp;
}

#pragma oss task inout([n_blocks] local) inout([n_blocks]
forces)
update_particles(n_blocks, local, forces, time_interval);
}

#pragma oss taskwait
}

```

```

void solve_nbody_cp(const int n_blocks,
        const int rank,
        particles_block_t const* __restrict__ local,
        const int timestep) {
    int status;
    int saved_data = 0;
    int saved_data_2 = 0;
    int i=0;
    int res;

    char name[256];
    char path[SCR_MAX_FILENAME];

    int perform_checkpoint;
    SCR_Need_checkpoint(&perform_checkpoint);
    if (perform_checkpoint == 1) {
        res = SCR_Start_checkpoint();
        if(res != SCR_SUCCESS)

```

```

        assert(0 && "SCR failed starting a checkpoint.");

        const char * scr_prefix = getenv("SCR_PREFIX");
        sprintf(name, "s/solve_nbody-nb d_bs d_r d.ckpt",
scr_prefix, n_blocks, BLOCK_SIZE, rank);

        // Get backup file path
        if(SCR_Route_file(name, path)==SCR_SUCCESS){
            cint fd = open (path,
                O_WRONLY | O_CREAT | O_TRUNC,
                S_IRUSR | S_IRGRP | S_IROTH | S_IWUSR |
                S_IWGRP | S_IWOTH);

            // Open, write and close file
            assert(fd >= 0);
            saved_data = write(fd, &timestep, sizeof(int));
            saved_data_2 = write(fd, local, sizeof(
particles_block_t)*n_blocks);
            assert(close(fd)==0);
        }
        int is_valid = (saved_data+saved_data_2) == (sizeof(
particles_block_t)*n_blocks + sizeof(int));
        SCR_Complete_checkpoint(is_valid);
    }
}

int solve_nbody_rt(const int n_blocks,
                  const int rank,
                  particles_block_t* __restrict__ local,
                  int *current_timestep) {
    int status = 0, found_cp = 0, temp_tstep, num_read,
num_read_particles, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);

```

```

char name[256], path[SCR_MAX_FILENAME], cp_name[
SCR_MAX_FILENAME];
sprintf(name, "solve_nbody-nb d_bs d_r d.ckpt", n_blocks,
BLOCK_SIZE, rank);

int res = SCR_Have_restart(&found_cp, path);
if(res != SCR_SUCCESS)
    assert(0 && "SCR failed when checking for available
restarts.");

if(!found_cp)
    return -1;
found_cp = 0;

if(SCR_Start_restart(cp_name) != SCR_SUCCESS)
    assert(0 && "SCR failed starting a restart.");

if (SCR_Route_file(name, path) == SCR_SUCCESS) {
    int fd = open (path,
                  O_RDWR | O_CREAT,
                  S_IRUSR | S_IRGRP | S_IROTH | S_IWUSR |
S_IWGRP | S_IWOTH);
    assert(fd >= 0);

    num_read = read(fd, &temp_tstep, sizeof(int));
    void *tmp = (void *) mmap(NULL,
                              n_blocks*sizeof(particles_block_t
)+sizeof(int),
                              PROT_WRITE|PROT_READ, MAP_SHARED,
fd, 0);
    assert(close(fd)==0);

    if(num_read == sizeof(int) && tmp != MAP_FAILED) {
        found_cp = 1;
    }
}

```

```

        char * aux = (char *) tmp + sizeof(int);
        memcpy( local, aux, n_blocks*sizeof(particles_block_t )
);
        munmap( tmp, n_blocks*sizeof(particles_block_t) );
    } else {
        status = -1;
    }
}

if(SCR_Complete_restart(found_cp) != SCR_SUCCESS)
    status = -5;

/* determine whether all tasks successfully read their
checkpoint file */
int all_found_checkpoint = 0;
MPI_Allreduce(&found_cp, &all_found_checkpoint, 1, MPI_INT,
MPI_LAND,
MPI_COMM_WORLD);
if (!all_found_checkpoint)
    status = -2;

/* check that everyone is at the same timestep */
int timestep_and, timestep_or;
MPI_Allreduce(current_timestep, &timestep_and, 1, MPI_INT,
MPI_BAND, MPI_COMM_WORLD);
MPI_Allreduce(current_timestep, &timestep_or, 1, MPI_INT,
MPI_BOR, MPI_COMM_WORLD);
if (timestep_and != timestep_or)
    status = -3;

if(status == 0)
    *current_timestep = temp_tstep;

```

```
return status;  
}
```


Bibliography

- [1] DAVID ALVAREZ, KEVIN SALA, MARCOS MARONAS, ALEIX ROCA, and VICENÇ BELTRAN. “Advanced Synchronization Techniques for Task-Based Runtime Systems”. *26th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP)*. Submitted pending acceptance. ACM (cited on p. 19)
- [2] SAMAN AMARASINGHE, DAN CAMPBELL, WILLIAM CARLSON, ANDREW CHIEN, WILLIAM DALLY, ELMOOTAZBELLAH ELNOHAZY, MARY HALL, ROBERT HARRISON, WILLIAM HARROD, KERRY HILL, et al. “Exascale software study: Software challenges in extreme scale systems”. *DARPA IPTO, Air Force Research Labs, Tech. Rep* (2009), 1–153 (cited on p. 127)
- [3] ANDI KLEEN, SUSE LABS. *numa(3) — Linux manual page*. Accessed: 2020-08-04. URL: <https://man7.org/linux/man-pages/man3/numa.3.html> (cited on p. 31)
- [4] ANDI KLEEN, SUSE LABS. *numactl(8) - Linux man page*. Accessed: 2020-08-04. URL: <https://linux.die.net/man/8/numactl> (cited on p. 31)
- [5] ARGONNE NATIONAL LABORATORY AND LAWRENCE LIVERMORE NATIONAL LABORATORY. *VELOC*. Accessed: 2019-03-24. URL: <https://veloc.readthedocs.io/en/latest/toc.html> (cited on pp. 33, 128)
- [6] CÉDRIC AUGONNET, SAMUEL THIBAUT, RAYMOND NAMYST, and PIERRE-ANDRÉ WACRENIER. “StarPU: a unified platform for task scheduling on heterogeneous multicore architectures”. *Concurrency and Computation: Practice and Experience* 23.2 (2011), 187–198 (cited on p. 36)
- [7] EDUARD AYGUADE, ROSA M BADIA, DANIEL CABRERA, ALEJANDRO DURAN, MARC GONZALEZ, FRANCISCO IGUAL, DANIEL JIMENEZ, JESUS LABARTA, XAVIER MARTORELL, RAFAEL MAYO, et al. “A proposal to extend the openmp tasking model for heterogeneous architectures”. *International Workshop on OpenMP*. Springer, 154–167 (cited on p. 36)

- [8] EDUARD AYGUADÉ and DANIEL JIMÉNEZ-GONZÁLEZ. “An approach to task-based parallel programming for undergraduate students”. *Journal of Parallel and Distributed Computing* 118 (2018), 140–156
- [9] PAVAN BALAJI, DARIUS BUNTINAS, DAVID GOODELL, WILLIAM GROPP, TORSTEN HOEFLER, SAMEER KUMAR, EWING LUSK, RAJEEV THAKUR, and JESPER LARSSON TRÄFF. “MPI on millions of cores”. *Parallel Processing Letters* 21.01 (2011), 45–60 (cited on p. 124)
- [10] BARCELONA SUPERCOMPUTING CENTER. *Extræ*. Accessed: 2020-05-24. URL: <https://tools.bsc.es/extrae> (cited on p. 74)
- [11] BARCELONA SUPERCOMPUTING CENTER. *Mercurium Compiler*. Accessed: 2019-03-24. URL: <https://github.com/bsc-pm/mcxx> (cited on pp. 19, 48, 74, 115, 140)
- [12] BARCELONA SUPERCOMPUTING CENTER. *Nanos6 Runtime*. Accessed: 2019-03-24. URL: <https://github.com/bsc-pm/nanos6> (cited on pp. 19, 48, 74, 115)
- [13] BARCELONA SUPERCOMPUTING CENTER. *OmpSs User Guide - How to exploit NUMA (socket) aware scheduling policy using Nanos++*. Accessed: 2020-11-04. Barcelona, Spain. URL: <https://pm.bsc.es/ftp/ompss/doc/user-guide/faq-numa-schedule.html> (cited on p. 29)
- [14] BARCELONA SUPERCOMPUTING CENTER. *OmpSs-2 Programming Model*. Accessed: 2019-03-24. URL: <https://pm.bsc.es/ompss-2> (cited on p. 12)
- [15] BARCELONA SUPERCOMPUTING CENTER. *OpenCHK*. Accessed: 2019-03-24. URL: <https://github.com/bsc-pm/OpenCHK-model> (cited on pp. 12, 128)
- [16] BARCELONA SUPERCOMPUTING CENTER. *Paraver*. Accessed: 2020-05-24. URL: <https://tools.bsc.es/paraver> (cited on p. 74)
- [17] BARCELONA SUPERCOMPUTING CENTER. *TAMPI library*. Accessed: 2020-05-24. URL: <https://github.com/bsc-pm/tampi> (cited on p. 20)
- [18] BARCELONA SUPERCOMPUTING CENTER. *Transparent Checkpoint Library (TCL)*. Accessed: 2019-03-24. URL: <https://github.com/bsc-pm/TCL> (cited on p. 140)
- [19] BARCELONA SUPERCOMPUTING CENTER. *OmpSs Specification*. Accessed: 2020-11-04. Barcelona, Spain, Dec. 2015. URL: <https://pm.bsc.es/ompss-docs/specs/> (cited on p. 14)

- [20] BARCELONA SUPERCOMPUTING CENTER. *Knights Landing CTE User's Guide*. Accessed: 2019-04-12. Barcelona, Spain, Apr. 2019. URL: <https://www.bsc.es/user-support/knl.php> (cited on p. 58)
- [21] BARCELONA SUPERCOMPUTING CENTER. *MareNostrum IV User's Guide*. Accessed: 2019-03-24. Barcelona, Spain, Mar. 2019. URL: <https://www.bsc.es/support/MareNostrum4-ug.pdf> (cited on pp. 58, 74, 115, 146)
- [22] LEONARDO BAUTISTA-GOMEZ, SEIJI TSUBOI, DIMITRI KOMATITSCH, FRANCK CAPPELLO, NAOYA MARUYAMA, and SATOSHI MATSUOKA. "FTI: high performance fault tolerance interface for hybrid systems". *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, 1–32 (cited on pp. 33, 128)
- [23] DAVID E BERNHOLDT, SWEN BOEHM, GEORGE BOSILCA, MANJUNATH GORENTLA VENKATA, RYAN E GRANT, THOMAS NAUGHTON, HOWARD P PRITCHARD, MARTIN SCHULZ, and GEOFFROY R VALLEE. "A survey of MPI usage in the US exascale computing project". *Concurrency and Computation: Practice and Experience* 32.3 (2020), e4851 (cited on p. 124)
- [24] GREG BRONEVETSKY, DANIEL MARQUES, KESHAV PINGALI, and PAUL STODGHILL. "Automated application-level checkpointing of MPI programs". *ACM Sigplan Notices*. Vol. 38. 10. ACM, 84–94 (cited on p. 32)
- [25] GREG BRONEVETSKY, DANIEL MARQUES, KESHAV PINGALI, PETER SZWED, and MARTIN SCHULZ. "Application-level checkpointing for shared memory programs". *ACM SIGARCH Computer Architecture News* 32.5 (2004), 235–247 (cited on p. 32)
- [26] PAUL CAHENY, LLUC ALVAREZ, SAID DERRADJI, MATEO VALERO, MIQUEL MORETO, and MARC CASAS. "Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach". *IEEE Transactions on Parallel and Distributed Systems*. IEEE, 1174–1187 (cited on p. 104)
- [27] PAUL CAHENY, MARC CASAS, MIQUEL MORETÓ, HERVÉ GLOAGUEN, MAXIME SAINTES, EDUARD AYGUADÉ, JESÚS LABARTA, and MATEO VALERO. "Reducing cache coherence traffic with hierarchical directory cache and NUMA-aware runtime scheduling". *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. IEEE, 275–286 (cited on p. 104)

- [28] FRANCK CAPPELLO, AL GEIST, WILLIAM GROPP, SANJAY KALE, BILL KRAMER, and MARC SNIR. “Toward exascale resilience: 2014 update”. *Supercomputing frontiers and innovations* 1.1 (2014), 5–28 (cited on p. 127)
- [29] CARLOS CARVALHO. “The gap between processor and memory speeds”. *Proc. of IEEE International Conference on Control and Automation* (cited on p. 3)
- [30] MARC CASAS, BRONIS R DE SUPINSKI, GREG BRONEVETSKY, and MARTIN SCHULZ. “Fault resilience of the algebraic multi-grid solver”. *Proceedings of the 26th ACM international conference on Supercomputing*, 91–100 (cited on p. 33)
- [31] VINCENT CAVÉ, JISHENG ZHAO, JUN SHIRAKO, and VIVEK SARKAR. “Habanero-Java: the new adventures of old X10”. *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, 51–61 (cited on p. 13)
- [32] MOHAMAD CHAARAWI and EDGAR GABRIEL. “Evaluating sparse data storage techniques for MPI groups and communicators”. *International Conference on Computational Science*. Springer, 297–306 (cited on p. 124)
- [33] SANJAY CHATTERJEE, SAGNAK TASIRLAR, ZORAN BUDIMLIC, VINCENT CAVE, MILIND CHABBI, MAX GROSSMAN, VIVEK SARKAR, and YONGHONG YAN. “Integrating asynchronous task parallelism with MPI”. *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 712–725 (cited on p. 25)
- [34] CHERNOUSOV, ANDREI. *Free CFD Source Codes*. Accessed: 2019-03-24. URL: <http://www.geocities.ws/MotorCity/Pit/9939/freecfd.htm> (cited on p. 148)
- [35] CINECA. *High Temperature Superconductivity*. Accessed: 2019-03-24. URL: <http://www.deep-er.eu/applications/project-applications/high-temperature-superconductivity.html> (cited on p. 149)
- [36] JOHN T DALY. “A higher order estimate of the optimum checkpoint interval for restart dumps”. *Future Generation Computer Systems* 22.3 (2006), 303–312 (cited on p. 33)

- [37] JAMES DINAN, PAVAN BALAJI, DAVID GOODELL, DOUGLAS MILLER, MARC SNIR, and RAJEEV THAKUR. “Enabling MPI interoperability through flexible communication endpoints”. *Proceedings of the 20th European MPI Users’ Group Meeting*, 13–18 (cited on p. 28)
- [38] JACK DONGARRA et al. “The international exascale software project roadmap”. *International Journal of High Performance Computing Applications* 25.1 (Feb. 2011), 3–60 (cited on p. 22)
- [39] PENG DU, AURELIEN BOUTELLER, GEORGE BOSILCA, THOMAS HERAULT, and JACK DONGARRA. “Algorithm-based fault tolerance for dense matrix factorizations”. *ACM SIGPLAN Notices* 47.8 (2012), 225–234 (cited on p. 127)
- [40] ANDRZEJ DUDA. “The effects of checkpointing on program execution time”. *Information Processing Letters* 16.5 (1983), 221–229 (cited on p. 33)
- [41] JASON DUELL. “The design and implementation of berkeley lab’s linux checkpoint/restart” (2005) (cited on p. 32)
- [42] JASON DUELL, PAUL H HARGROVE, and ERIC S ROMAN. *Requirements for linux checkpoint/restart*. Tech. rep. Ernest Orlando Lawrence Berkeley National Lab., CA (US), 2002 (cited on p. 32)
- [43] ALEJANDRO DURAN, EDUARD AYGUADÉ, ROSA M BADIA, JESÚS LABARTA, LUIS MARTINELL, XAVIER MARTORELL, and JUDIT PLANAS. “OmpSs: A proposal for programming heterogeneous multi-core architectures”. *Parallel Processing Letters* 21.2 (2011), 173–193 (cited on p. 36)
- [44] HIKMET DURSUN, KEN-ICHI NOMURA, LIU PENG, RICHARD SEYMOUR, WEIQIANG WANG, RAJIV K KALIA, AIICHIRO NAKANO, and PRIYA VASHISHTA. “A multilevel parallelization framework for high-order stencil computations”. *European Conference on Parallel Processing*. Springer, 642–653 (cited on p. 25)
- [45] EROL GELENBE. “A model of roll-back recovery with multiple checkpoints”. *Proceedings of the 2nd international conference on Software engineering*, 251–255 (cited on p. 33)
- [46] LEONARDO ARTURO BAUTISTA GOMEZ and FRANCK CAPPELLO. “Detecting and correcting data corruption in stencil applications through multivariate interpolation”. *2015 IEEE International Conference on Cluster Computing*. IEEE, 595–602 (cited on p. 127)

- [47] GARY GRIDER, JOSIP LONCARIC, and D LIMPART. “Roadrunner system management report”. *Los Alamos National Laboratory, Tech. Rep. LA-UR-07-7405* (2007) (cited on p. 33)
- [48] PAUL H HARGROVE and JASON C DUELL. “Berkeley lab checkpoint/restart (blcr) for linux clusters”. *Journal of Physics: Conference Series*. Vol. 46. 1, 494 (cited on pp. 32, 127)
- [49] TIM HARRIS and STEFAN KAESTLE. “Callisto-RTS: fine-grain parallel loops”. *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, 45–56 (cited on p. 37)
- [50] MICHAEL A HEROUX, DOUGLAS W DOERFLER, PAUL S CROZIER, JAMES M WILLENBRING, H CARTER EDWARDS, ALAN WILLIAMS, MAHESH RAJAN, ERIC R KEITER, HEIDI K THORNQUIST, and ROBERT W NUMRICH. “Improving performance via mini-applications”. *Sandia National Laboratories, Tech. Rep. SAND2009-5574 3* (2009) (cited on p. 72)
- [51] TORSTEN HOEFLER, CHRISTIAN SIEBERT, and WOLFGANG REHM. “A practically constant-time MPI Broadcast Algorithm for large-scale InfiniBand Clusters with Multicast”. *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 1–8 (cited on p. 124)
- [52] DANIEL HOLMES, KATHRYN MOHROR, RYAN E GRANT, ANTHONY SKJELUM, MARTIN SCHULZ, WESLEY BLAND, and JEFFREY M SQUYRES. “MPI sessions: leveraging runtime infrastructure to increase scalability of applications at exascale”. *Proceedings of the 23rd European MPI Users’ Group Meeting*, 121–129 (cited on p. 124)
- [53] *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory*. Tech. rep. LLNL-TR-490254. Livermore, CA, 1–17 (cited on p. 72)
- [54] INRIA. *Human Exposure to Electromagnetic Fields*. Accessed: 2019-03-24. URL: <https://www.deep-projects.eu/applications/project-applications/human-exposure-to-electromagnetic-fields.html> (cited on p. 148)
- [55] INTEL. *Intel C++ Compiler 19.0 Developer Guide and Reference*. Accessed: 2019-03-24. URL: <https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-cilk-for> (cited on p. 28)

- [56] LUC JAULMES, MARC CASAS, MIQUEL MORETÓ, EDUARD AYGUADÉ, JESÚS LABARTA, and MATEO VALERO. “Exploiting asynchrony from exact forward recovery for DUE in iterative solvers”. *SC’15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12
- [57] LUC JAULMES, MIQUEL MORETO, EDUARD AYGUADE, JESUS LABARTA, MATEO VALERO, and MARC CASAS. “Asynchronous and exact forward recovery for detected errors in iterative solvers”. *IEEE Transactions on Parallel and Distributed Systems* 29.9 (2018), 1961–1974 (cited on p. 33)
- [58] HUMAIRA KAMAL, SEYED M MIRTAHERI, and ALAN WAGNER. “Scalability of communicators and groups in MPI”. *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 264–275 (cited on p. 124)
- [59] KARL RUPP. *42 Years of Microprocessor Trend Data*. Accessed: 2020-08-04. URL: <https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/> (cited on p. 2)
- [60] IAN KARLIN, JEFF KEASLER, and JR NEELY. *Lulesh 2.0 updates and changes*. Tech. rep. Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2013 (cited on p. 148)
- [61] KAI KELLER and LEONARDO BAUTISTA GOMEZ. “Application-Level Differential Checkpointing for HPC Applications with Dynamic Datasets”. *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 52–61 (cited on pp. 136, 137)
- [62] BENJAMIN KLENK and HOLGER FRÖNING. “An overview of MPI characteristics of exascale proxy applications”. *International Supercomputing Conference*. Springer, 217–236 (cited on p. 124)
- [63] JANNIS KLINKENBERG, PHILIPP SAMFASS, CHRISTIAN TERBOVEN, ALEJANDRO DURAN, MICHAEL KLEMM, XAVIER TERUEL, SERGI MATEO, STEPHEN L. OLIVIER, and MATTHIAS S. MÜLLER. “Assessing Task-to-Data Affinity in the LLVM OpenMP Runtime”. *Evolving OpenMP for Evolving Architectures*. Springer International Publishing, 236–251 (cited on p. 30)
- [64] YUETSU KODAMA, TETSUYA ODAJIMA, AKIRA ASATO, and MITSUHIISA SATO. “Evaluation of the riken post-k processor simulator”. *arXiv preprint arXiv:1904.06451* (2019) (cited on p. 102)

- [65] ALEXEY KUKANOV and MICHAEL J VOSS. “The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks.” *Intel Technology Journal* 11.4 (2007) (cited on p. 11)
- [66] LAWRENCE LIVERMORE NATIONAL LABORATORY. *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH)*. Accessed: 2020-05-24. URL: <https://github.com/LLNL/LULESH> (cited on p. 72)
- [67] CHARLES LEISERSON and ASKE PLAAT. “Programming parallel applications in Cilk”. *SINEWS: SIAM News* 31.4 (1998), 6–7 (cited on p. 13)
- [68] HONGBO LI, ZIZHONG CHEN, RAJIV GUPTA, and MIN XIE. “Non-intrusively avoiding scaling problems in and out of mpi collectives”. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 415–424 (cited on p. 124)
- [69] DAVID LUEBKE. “CUDA: Scalable parallel programming for high-performance scientific computing”. *2008 5th IEEE international symposium on biomedical imaging: from nano to macro*. IEEE, 836–838 (cited on p. 11)
- [70] STEFANO MARKIDIS, GIOVANNI LAPENTA, et al. “Multi-scale simulations of plasma with iPIC3D”. *Mathematics and Computers in Simulation* 80.7 (2010), 1509–1519 (cited on p. 149)
- [71] MARCOS MARONAS, SERGI MATEO, VICENÇ BELTRAN, and EDUARD AYGUADÉ. “A directive-based approach to perform persistent checkpoint/restart”. *2017 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 442–451 (cited on p. 128)
- [72] MARCOS MAROÑAS, KEVIN SALA, SERGI MATEO, EDUARD AYGUADÉ, and VICENÇ BELTRAN. “Worksharing Tasks: An Efficient Way to Exploit Irregular and Fine-Grained Loop Parallelism”. *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 383–394 (cited on p. 103)
- [73] MARCOS MAROÑAS, XAVIER TERUEL, MARK BULL, EDUARD AYGUADE, and VICENÇ BELTRAN. “Evaluating Worksharing Tasks on Distributed Environments”. *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. Pending publication. IEEE (cited on p. 124, 125)
- [74] MESSAGE PASSING INTERFACE FORUM. *A Message-Passing Interface Standard Version 2.0*. Accessed: 2020-03-24. 1997. URL: <https://www.mpi-forum.org/docs/mpi-2.0/mpi2-report.pdf> (cited on p. 28)

- [75] MESSAGE PASSING INTERFACE FORUM. *A Message-Passing Interface Standard Version 3.0*. Accessed: 2020-03-24. 2012. URL: <https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf> (cited on p. 28)
- [76] MICHAEL A. HEROUX. *High Performance Computing Conjugate Gradients: The original Mantevo miniapp*. Accessed: 2019-03-24. URL: <https://github.com/Mantevo/HPCCG> (cited on pp. 58, 72)
- [77] ADAM MOODY, GREG BRONEVETSKY, KATHRYN MOHROR, and BRONIS R DE SUPINSKI. “Design, modeling, and evaluation of a scalable multi-level checkpointing system”. *SC’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11 (cited on pp. 33, 128)
- [78] RAMACHANDRA NANJEGOWDA, OSCAR HERNANDEZ, BARBARA CHAPMAN, and HAOQIANG H JIN. “Scalability evaluation of barrier algorithms for OpenMP”. *International Workshop on OpenMP*. Springer, 42–52 (cited on p. 27)
- [79] NASA. *NAS Parallel Benchmarks*. Accessed: 2019-03-24. URL: <http://www.nas.nasa.gov/publications/npb.html> (cited on p. 148)
- [80] ANTONI NAVARRO, SERGI MATEO, JOSEP MARIA PEREZ, VICENÇ BELTRAN, and EDUARD AYGUADÉ. “Adaptive and architecture-independent task granularity for recursive applications”. *International Workshop on OpenMP*. Springer, 169–182 (cited on pp. 38, 103)
- [81] RON A OLDFIELD, SARALA ARUNAGIRI, PATRICIA J TELLER, SEETHARAMI SEELAM, MARIA RUIZ VARELA, ROLF RIESEN, and PHILIP C ROTH. “Modeling the impact of checkpoints on next-generation systems”. *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*. IEEE, 30–46 (cited on p. 33)
- [82] RABAB AL-OMAIRY, GUILLERMO MIRANDA, HATEM LTAIEF, ROSA M BADA, XAVIER MARTORELL, JESUS LABARTA, and DAVID KEYES. “Dense matrix computations on numa architectures with distance-aware work stealing”. *Supercomputing Frontiers and Innovations* 2.1 (2015), 49–72 (cited on p. 29)
- [83] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Programming Interface - Memory Allocators*. Accessed: 2020-11-04. URL: <https://www.openmp.org/spec-html/5.0/openmpsu53.html> (cited on p. 30)

- [84] OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP Application Programming Interface*. Accessed: 2019-03-24. Nov. 2018. URL: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf> (cited on pp. 11, 35, 128)
- [85] JEAN-CHARLES PAPIN, CHRISTOPHE DENOVAL, LAURENT COLOMBET, and RAYMOND NAMYST. “SPAWN: An Iterative, Potentials-Based, Dynamic Scheduling and Partitioning Tool”. *SuperComputing’15-RESPA Workshop*
- [86] JOSEP M PEREZ, VICENÇ BELTRAN, JESUS LABARTA, and EDUARD AYGUADÉ. “Improving the integration of task nesting and dependencies in OpenMP”. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 809–818 (cited on p. 36)
- [87] JAMES S PLANK, MICAH BECK, GERRY KINGSLEY, and KAI LI. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1994 (cited on p. 136)
- [88] JAMES S PLANK, KAI LI, and MICHAEL A PUENING. “Diskless checkpointing”. *IEEE Transactions on Parallel and Distributed Systems* 9.10 (1998), 972–986 (cited on p. 32)
- [89] JAMES S PLANK and MICHAEL G THOMASON. “Processor allocation and checkpoint interval selection in cluster computing systems”. *Journal of Parallel and distributed Computing* 61.11 (2001), 1570–1590 (cited on p. 33)
- [90] ROLF RABENSEIFNER, GEORG HAGER, and GABRIELE JOST. “Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes”. *2009 17th Euromicro international conference on parallel, distributed and network-based processing*. IEEE, 427–436 (cited on p. 25)
- [91] DONALD REED. “High-end computing: The challenge of scale”. *Director’s Colloquium, Los Alamos National Laboratory*. Vol. 1. 6 (cited on pp. 33, 127)
- [92] ERIC ROMAN. “A survey of checkpoint/restart implementations”. *Lawrence Berkeley National Laboratory, Tech. Citeseer* (cited on p. 32)
- [93] SATISH KUMAR SADASIVAM, BRIAN W THOMPSON, RON KALLA, and WILLIAM J STARKE. “IBM Power9 processor architecture”. *IEEE Micro* 37.2 (2017), 40–51 (cited on p. 102)

- [94] FLORENTINO SAINZ, JORGE BELLON, VICENC BELTRAN, and JESUS LABARTA. “Collective offload for heterogeneous clusters”. *2015 IEEE 22nd international conference on high performance computing (HiPC)*. IEEE, 376–385 (cited on p. 28)
- [95] KEVIN SALA, JORGE BELLÓN, PAU FARRÉ, XAVIER TERUEL, JOSEP M PEREZ, ANTONIO J PEÑA, DANIEL HOLMES, VICENÇ BELTRAN, and JESUS LABARTA. “Improving the interoperability between MPI and task-based programming models”. *Proceedings of the 25th European MPI Users’ Group Meeting*, 1–11 (cited on pp. 29, 36)
- [96] KEVIN SALA, XAVIER TERUEL, JOSEP M PEREZ, ANTONIO J PEÑA, VICENÇ BELTRAN, and JESUS LABARTA. “Integrating blocking and non-blocking mpi primitives with task-based programming models”. *Parallel Computing* 85 (2019), 153–166 (cited on pp. 29, 36)
- [97] ISAAC SÁNCHEZ BARRERA, MIQUEL MORETÓ, EDUARD AYGUADÉ, JESÚS LABARTA, MATEO VALERO, and MARC CASAS. “Reducing data movement on large shared memory systems by exploiting computation dependencies”. *Proceedings of the 2018 International Conference on Supercomputing*, 207–217 (cited on p. 30)
- [98] SRIRAM SANKARAN, JEFFREY M SQUYRES, BRIAN BARRETT, VISHAL SAHAY, ANDREW LUMSDAINE, JASON DUELL, PAUL HARGROVE, and ERIC ROMAN. “The LAM/MPI checkpoint/restart framework: System-initiated checkpointing”. *The International Journal of High Performance Computing Applications* 19.4 (2005), 479–493 (cited on p. 32)
- [99] BIANCA SCHROEDER and GARTH A GIBSON. “Understanding failures in petascale computers”. *Journal of Physics: Conference Series*. Vol. 78. 1. IOP Publishing, 012022 (cited on p. 33)
- [100] BIANCA SCHROEDER and GARTH A GIBSON. “A large-scale study of failures in high-performance computing systems”. *Dependable and Secure Computing, IEEE Transactions on* 7.4 (2010), 337–350 (cited on p. 33)
- [101] AVINASH SODANI. “Knights landing (knl): 2nd generation intel® xeon phi processor”. *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 1–24 (cited on p. 102)

- [102] GEORG STELLNER. “CoCheck: Checkpointing and process migration for MPI”. *Proceedings of International Conference on Parallel Processing*. IEEE, 526–531 (cited on p. 32)
- [103] OMER SUBASI, JAVIER ARIAS, OSMAN UNSAL, JESUS LABARTA, and ADRIAN CRISTAL. “NanoCheckpoints: A Task-based Asynchronous Dataflow Framework for Efficient and Scalable Checkpoint/Restart”. *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 99–102 (cited on p. 32)
- [104] OMER SUBASI, TATIANA MARTSINKEVICH, FERAD ZYULKYAROV, OSMAN UNSAL, JESUS LABARTA, and FRANCK CAPPELLO. “Unified fault-tolerance framework for hybrid task-parallel message-passing applications”. *The International Journal of High Performance Computing Applications* 32.5 (2018), 641–657 (cited on p. 24)
- [105] OMER SUBASI, OSMAN UNSAL, JESUS LABARTA, GULAY YALCIN, and ADRIAN CRISTAL. “Crc-based memory reliability for task-parallel HPC applications”. *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1101–1112 (cited on p. 31)
- [106] OMER SUBASI, GULAY YALCIN, FERAD ZYULKYAROV, OSMAN UNSAL, and JESUS LABARTA. “A runtime heuristic to selectively replicate tasks for application-specific reliability targets”. *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 498–505 (cited on p. 32)
- [107] DAVID SUGGS, MAHESH SUBRAMONY, and DAN BOUVIER. “The AMD “Zen 2” Processor”. *IEEE Micro* 40.2 (2020), 45–52 (cited on p. 102)
- [108] BRONIS R. DE SUPINSKI, THOMAS R. W. SCOGLAND, ALEJANDRO DURAN, MICHAEL KLEMM, SERGI MATEO BELLIDO, STEPHEN L. OLIVIER, CHRISTIAN TERBOVEN, and TIMOTHY G. MATTSON. “The Ongoing Evolution of OpenMP”. *Proceedings of the IEEE* 106.11 (2018), 2004–2019 (cited on p. 30)
- [109] MASAHIRO TANAKA and OSAMU TATEBE. “Workflow scheduling to minimize data movement using multi-constraint graph partitioning”. *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE, 65–72 (cited on p. 30)
- [110] THE HDF GROUP. *HDF5 User’s Guide*. Sept. 2019. URL: <https://portal.hdfgroup.org/display/HDF5/HDF5+User27s+Guide> (cited on p. 138)

- [111] PETER THOMAN, KIRIL DICHEV, THOMAS HELLER, ROMAN IAKYMCHUK, XAVIER AGUILAR, KHALID HASANOV, PHILIPP GSCHWANDTNER, PIERRE LEMARINIER, STEFANO MARKIDIS, HERBERT JORDAN, et al. “A taxonomy of task-based parallel programming technologies for high-performance computing”. *The Journal of Supercomputing* 74.4 (2018), 1422–1434
- [112] UNIVERSITÉ DE BORDEAUX, CNRS AND INRIA. *StarPU Handbook*. Accessed: 2019-04-04. Apr. 2019. URL: <http://starpu.gforge.inria.fr/doc/html/index.html> (cited on p. 28)
- [113] UNIVERSITÉ DE BORDEAUX, CNRS AND INRIA. *StarPU Handbook - Parallel Tasks*. Accessed: 2019-04-04. Apr. 2019. URL: <http://starpu.gforge.inria.fr/doc/html/TasksInStarPU.html5C#ParallelTasks> (cited on p. 28)
- [114] UNIVERSITY OF TENNESSEE. *HPC Challenge Benchmark*. Accessed: 2019-03-24. URL: <http://icl.cs.utk.edu/hpcc/> (cited on pp. 58, 148)
- [115] NITIN HEMANT VAIDYA. *A case for multi-level distributed recovery schemes*. Citeseer, 1994 (cited on p. 33)
- [116] JEFFREY S VETTER and FRANK MUELLER. “Communication characteristics of large-scale scientific applications for contemporary cluster architectures”. *Proceedings 16th International Parallel and Distributed Processing Symposium*. IEEE, 10–pp (cited on p. 124)
- [117] DAVID W WALKER and JACK J DONGARRA. “MPI: a standard message passing interface”. *Supercomputer* 12 (1996), 56–68 (cited on p. 11)
- [118] HAO WANG, SREERAM POTLURI, MIAO LUO, ASHISH KUMAR SINGH, SAYANTAN SUR, and DHABALESWAR K PANDA. “MVAPICH2-GPU: optimized GPU to GPU communication for InfiniBand clusters”. *Computer Science-Research and Development* 26.3-4 (2011), 257 (cited on p. 28)
- [119] WHEELER, DAVID A. *SLOCCount*. Accessed: 2019-03-24. URL: <https://dwheeler.com/sloccount/> (cited on pp. 143, 151)
- [120] WM A WULF and SALLY A MCKEE. “Hitting the memory wall: implications of the obvious”. *ACM SIGARCH computer architecture news* 23.1 (1995), 20–24 (cited on p. 1)
- [121] YIN, TERRY. *Lizard*. Accessed: 2019-03-24. URL: <https://github.com/terryyin/lizard> (cited on p. 143)
- [122] JOHN W YOUNG. “A first order approximation to the optimum checkpoint interval”. *Communications of the ACM* 17.9 (1974), 530–531 (cited on p. 33)

- [123] GENGBIN ZHENG, LIXIA SHI, and LAXMIKANT V KALÉ. “FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI”. *2004 ieee international conference on cluster computing (ieee cat. no. 04EX935)*. IEEE, 93–103 (cited on p. 32)

Nomenclature

<i>API</i>	Application Programming Interface
<i>BS</i>	Block Size
<i>BSC</i>	Barcelona Supercomputing Center
<i>BSP</i>	Bulk Synchronous Parallel
<i>CC</i>	Cyclomatic Complexity
<i>CPU</i>	Central Processing Unit
<i>CR</i>	Checkpoint/Restart
<i>DARPA</i>	Defense Advanced Research Projects Agency
<i>DEE</i>	Development Estimate Effort
<i>EPCC</i>	Edinburgh Parallel Computing Centre
<i>EU</i>	European Union
<i>FOM</i>	Figure Of Merit
<i>FTI</i>	Fault Tolerant Interface
<i>GPU</i>	Graphics Processing Unit
<i>HBM</i>	High Bandwidth Memory
<i>HPC</i>	High Performance Computing
<i>HPCCG</i>	High Performance Computing Conjugate Gradient
<i>HPCG</i>	High Performance Conjugate Gradient

<i>KNL</i>	Knights Landing
<i>LULESH</i>	Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics
<i>MATMUL</i>	Matrix Multiplication
<i>MPI</i>	Message Passing Interface
<i>MTBF</i>	Mean Time Between Failures
<i>MTTF</i>	Mean Time To Failure
<i>NUMA</i>	Non Uniform Memory Access
<i>NVM</i>	Non-Volatile Memory
<i>OS</i>	Operating Systems
<i>PFS</i>	Parallel File System
<i>PS</i>	Problem Size
<i>RAM</i>	Random Access Memory
<i>RaW</i>	Read After Write
<i>SCM</i>	Storage Class Memory
<i>SCR</i>	Scalable Checkpoint Restart
<i>SLOC</i>	Source Lines Of Code
<i>SSD</i>	Solid State Drive
<i>TAMPI</i>	Task Aware MPI Library
<i>TBB</i>	Threading Building Blocks
<i>TCL</i>	Transparent Checkpoint Library
<i>TDG</i>	Task Dependency Graph
<i>TS</i>	Task Size

<i>UHPC</i>	Ubiquitous High Performance Computing
<i>VeloC</i>	Very Low Overhead Checkpointing System
<i>WaR</i>	Write After Read
<i>WaW</i>	Wead After Write

