A Checkpoint/Restart Directive-Based Approach

Marcos Maroñas Bravo
(marcos.maronas@bsc.es)

Director: Vicenç Beltran Querol (vbeltran@bsc.es)
Co-director: Sergi Mateo Bellido (sergi.mateo@bsc.es)
Tutor: Eduard Ayguadé Parra (eduard@ac.upc.edu)

Computer Architecture Department (DAC)
“Success consists of going from failure to failure without loss of enthusiasm.”

– Winston Churchill

Similarly, resilience consists of going from failure to failure without loss of progress.
Acknowledgments

I would like to thank my director, Vicenç Beltran, for the great opportunity he offered me to work on exciting and motivating projects. Thanks for believing in me, for your support and for all your ideas.

This project would not be possible without my co-director, Sergi Mateo. Thanks for all your help during the development.

To all my colleagues at BSC-CS Programming Models group, thanks for making an enjoyable working environment.

Special thanks to my mother. It is really important for me knowing that you are always there to help me in all I can need. You believe in me even more than I do, and that is really heartwarming.

To my lovely girl Neus, thanks for your support. Thanks for the great moments but, especially, thanks for helping me to overcome the difficulties.

Javier, your valuable comments have improved my work. Thank you, friend.

Last but not least, I wish to thank all my friends and family.

Also, this work is supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology (project TIN2015-65316-P) and by the Generalitat de Catalunya (grant 2014-SGR-1051).
Abstract

Exascale platforms require programming models incorporating support for resilience capabilities since the huge number of components they are expected to have is going to increase the number of errors.

Checkpoint/restart is a widely used resilience technique due to its robustness and low overhead compared to other techniques. There already exists several solutions implementing this technique, such as FTI or SCR, which focus mainly on providing advanced I/O capabilities to minimize checkpoint/restart time. However, application developers are still in charge of: (1) manually serialize and deserialize the application state using a low-level API; (2) modify the natural flow of the application depending whether the current execution is a restart or not; and (3) reimplement their code regarding checkpoint/restart whenever they have to change the backend library.

We present a new directive-based approach to performing application-level checkpoint/restart in a simplified and portable way. We propose a solution based on compiler directives, such as OpenMP ones, that allows users to easily specify the state of the application that has to be saved and restored, leaving the tedious and error-prone serialization and deserialization activities to our intermediate library, which relies on a backend library (FTI/SCR) to perform scalable and efficient I/O operations.

Our results, including several benchmarks and two large applications, reveal no extra overhead compared to the direct use of FTI/SCR checkpoint/restart libraries while significantly reducing the effort required by the application developers.
# Table of Contents

1 Introduction .................................................. 1
  1.1 Motivation ............................................... 2
  1.2 Objectives ............................................... 3
  1.3 Contributions .......................................... 3
  1.4 Document Structure .................................... 4

2 Related work .................................................. 5

3 OmpSs Parallel Programming Model .............................. 8
  3.1 Introduction and philosophy ............................... 8
  3.2 Parallelism ............................................... 10
    3.2.1 Tasks ................................................. 10
    3.2.2 Loops ............................................... 12
    3.2.3 Nesting or multi-level parallelism ..................... 12
  3.3 Synchronization .......................................... 12
    3.3.1 Data dependences .................................... 13
    3.3.2 Explicit synchronization points ....................... 16
  3.4 Heterogeneity ............................................. 17
    3.4.1 Single unified address space .......................... 17
    3.4.2 Architectures supported ............................... 20
  3.5 Influence in OpenMP ...................................... 21
List of Figures

3.1 Snippet of code highlighting compiler directives. .......................... 9
3.2 Snippet of code using outline tasks. .............................................. 11
3.3 Snippet of code using inline tasks. .............................................. 11
3.4 Snippet of code applying inline tasks to a structured block. .............. 11
3.5 Snippet of code applying for directive. ........................................ 12
3.6 Snippet of code using data dependences. ....................................... 13
3.7 Snippet of code showing illegal memory references in nested tasks. ...... 14
3.8 Snippet of code using data dependences. ....................................... 15
3.9 Snippet of code using array sections feature. ................................. 16
3.10 Snippet of code using shaping expressions feature. ......................... 17
3.11 High-level diagram of the unified memory concept of OmpSs. .......... 17
3.12 Snippet of code presenting copies mechanism. ............................... 19
3.13 Snippet of code showing illegal accesses from the parent task. ......... 19
3.14 Snippet of code showing illegal accesses from the parent task using noflush clause. .............................................................. 20
3.15 Summary of the contributions that OmpSs has done to OpenMP. ...... 21
4.1 Mercurium compiler structure. ..................................................... 23
4.2 Nanos++ runtime library structure. .............................................. 24
4.3 Three cycle view of Design Science Research [32] ........................ 26
4.4 Relevance Cycle of Three cycle view [32] ..................................... 27
4.5 Rigor Cycle of Three cycle view [32] ........................................... 28
4.6 Design Cycle of Three cycle view [32] .......................................... 29
5.1 Diagram of the three-layer architecture. ........................................... 31
5.2 Snippet of code presenting new OmpSs checkpoint/restart semantics ... 32
5.3 Snippet of code presenting new OmpSs checkpoint/restart semantics providing a condition expression ................................. 32
5.4 Behavior of OmpSs application-level checkpoint/restart ................. 33
5.5 Snippet of code clarifying the synchronization requirement. ............ 35
5.6 Snippet of code presenting N-Body application using OmpSs checkpoint/ restart mechanism ......................................................... 37

6.1 Trace of the execution of REF version of GERShWIN. ................. 42
6.2 Trace of the execution of SCR version of GERShWIN. .................. 42
6.3 Trace of the execution of CHK version of GERShWIN. .................. 42
6.4 Traces of the execution of REF, SCR and CHK versions of GERShWIN. . 42
6.5 Trace of the execution of SCR version from a restart of GERShWIN. .... 43
6.6 Trace of the execution of CHK version from a restart of GERShWIN. .... 43
6.7 Traces of the execution of SCR and CHK versions from a restart of GERShWIN. ................................................................. 43
6.8 Chart of the overhead introduced by FTI and CHK versions compared to REF version. ......................................................... 45
6.9 Chart of the overhead introduced by SCR and CHK versions compared to REF version. ......................................................... 45
List of Tables

6.1 Marenostrum architecture [34]. ............................................ 39
6.2 Table describing the software used to perform the experiments and their versions. ........................................ 40
6.3 Number of lines of code required to perform application-level checkpoint/restart for different tools. .................. 46
1 | Introduction

It is widely accepted that exascale systems will undergo a drastic increase in the overall failure rate due to several factors [1], mainly the growth in the overall number of components of the system, which leads to a reduction in the Mean Time Between Failures (MTBF) from hours or days in Petascale systems [2] to a few minutes in Exascale systems [3], [4]. Hence, the high-performance computing (HPC) research community is making big efforts in resilience and fault-tolerance. Several approaches, that address different kinds of errors already exist, such as application-specific resilience [5], including algorithm-specific [6], or more generic solutions [7].

Application-level checkpoint/restart is becoming very popular due to its efficiency in terms of space and performance when compared to other fault tolerance techniques. Nevertheless this technique as applied in current approaches requires great effort on the part of the user: after identifying the application state, the data has to be serialized and deserialized in order to be stored and restored; the program flow has to be modified by taking into account whether the execution is a restart or not, and the code regarding checkpoint/restart has to be rewritten whenever the backend library is changed.

With the objective of helping application developers to deal with this drastic increase of failure rates, an ecosystem is emerging composed of a variety of programming models, libraries, and tools providing fault-tolerance in a simplified way.

Scalable Checkpoint and Restart (SCR) and Fault Tolerant Interface (FTI) are two multi-level checkpointing libraries offering optimized I/O and redundancy schemes. They provide an efficient application programming interface (API) when performing checkpoint/restart. However, they still require considerable effort from the user side in comparison with other techniques such as transparent checkpoint/restart that needs no participation of the user, but these latter are far less efficient.

OmpSs is adding new features to be part of this new ecosystem targeting Exascale. In this work, we focus on providing application-level checkpoint/restart through compiler directives, offering this functionality to the end-user in a simplified fashion. As FTI/SCR, our solution supports soft and fail-stop errors (such as process abortions or hardware failures). Undetected (silent) failures are not considered in this work.
1.1 Motivation

New Exascale systems threaten to jeopardize the progress of large HPC applications, making fault-tolerance a key feature to guarantee their completion. Many HPC applications perform large simulations based on iterative algorithms. These kinds of applications require very long execution times for their completion, and thus the likelihood of a system error occurring is higher. The likelihood is greater in Exascale systems, where the high number of system components causes error frequency to increase to such an extent that it is possible to find applications that require more time to be completed than the MTBF of the system. Hence, fault-tolerant techniques, such as checkpoint/restart, are crucial to enable large applications to finish successfully.

Checkpoint/restart 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, thereby allowing the execution to continue from that point instead of from scratch. There are several approaches to this technique, which are detailed in Section 2. We focus our work on persistent checkpoint/restart where the data is stored in a persistent way. The checkpoints are usually located in the Parallel File System (PFS) because if a node is completely down and the checkpoints are done in its local storage, the data is not accessible, and thus the checkpoints are useless.

Current checkpoint/restart approaches involve some drawbacks; namely, poor portability and the issue of complex APIs. Solutions to performing application-level checkpoint/restart, range from ad-hoc solutions where the developer directly deals with low-level details, such as I/O; to libraries providing APIs abstracting the developer from such low-level details and nuances. Two popular libraries are FTI and SCR. Both provide multi-level checkpoint/restart with optimizations in I/O, as well as redundancy schemes. However, some of the current approaches present a challenge to inexpert programmers because of complex processes to perform checkpoint/restart. Furthermore, portability across different systems becomes a problem when applying checkpoint/restart techniques.

Firstly, as regards complexity, with current solutions the user is in charge of serializing the data when checkpointing, and deserializing it when restoring, then modifying the natural program flow in order to detect whether an execution is a restart from a previous checkpoint or whether it is a run from scratch, and then act consequently. Moreover, in some of the approaches, users must deal with I/O operations directly. In other words, there are many details to address apart from the application itself.

Secondly, the proliferation of several checkpoint/restart libraries with different interfaces hinders the portability between systems. Installation and tuning of the checkpoint/restart libraries are usually performed by expert system administrators in order to make the most of the storage hierarchy and to obtain the best performance. User’s options are therefore reduced to rewriting the code with the interface of this optimized library available in the target system.
The above-mentioned problems do not arise in a directive-based approach because several of the user responsibilities are transferred to the compiler and the intermediate library. Since this solution adds extra abstraction layers, it is possible to support multiple backend libraries (such as FTI/SCR) with the same interface, thereby enhancing portability. In addition, users can specify the data in a simpler way, and the effort of serializing and deserializing it is also passed to the intermediate library and compiler. Summarizing, this approach allows users to focus on the application itself, thus providing higher productivity and portability, together with an easy-to-use interface.

1.2 Objectives

The main objective of this project is to support application-level checkpoint/restart into the OmpSs parallel programming model. By using OmpSs checkpoint/restart, application developers have to be able to focus their efforts on the application itself, dedicating almost no time to protect the applications against faults. Then, the OmpSs compiler and runtime perform all the required steps to add checkpoint/restart in the application.

Therefore, more specifically, the objectives are:

- Develop support to the new semantics (such as checkpoint clause) in the compiler.
- Develop, in the runtime, the ability to manage checkpoints and restarts in combination with the backend libraries.

By introducing this feature into OmpSs, the goal is to enhance productivity and portability when coding resilient applications without extra overhead. Hence, another objective is to check the productivity and portability as well as the overhead of the developed approach. For that purpose, we chose some benchmarks and applications that use other approaches to performing checkpoint/restart and compare our approach with them.

1.3 Contributions

This work presents the new checkpoint clause applied to the task construct. With this approach we are able to:

1. Provide a straightforward way of indicate the data to be checkpointed. The data annotated as input or input/output dependence will be stored and restored when required.
2. Indicate when to perform a checkpoint, through the optional condition expression accepted by the checkpoint clause.
3. Avoid the requirement of modifying the natural program flow in order to check whether the current execution is a restart or not.

Our solution is working not only for sequential applications but also for parallel (OmpSs), distributed (MPI) and hybrid (OmpSs+MPI) applications.

In this work we (1) present the design of an efficient, easy-to-use and portable mechanism to perform application-level checkpoint/restart; and (2) provide an in-depth performance evaluation of the overhead introduced by the proposed mechanism. To the best of our knowledge, this is the first directive-based approach to performing checkpoint/restart at application-level. Our evaluations, involving several benchmarks and production applications, reveal that our solution is able to maintain efficiency and robustness, as current approaches, while enhancing portability and usability.

1.4 Document Structure

The rest of the document follows the next structure. Chapter 2 reviews the most relevant related work. Chapter 3 contains an introduction to the OmpSs parallel programming model. Chapter 4 details the tools and methods used in this project. Chapter 5 then explains the design of our proposal and also provides details of the implementation. After that, Chapter 6 consists of an evaluation of our work. Finally, Chapter 7 summarizes the work done and provides some concluding remarks, and future work proposals are presented in Chapter 8.
Related work

This section is devoted to reviewing prior work related to the contributions set out in this paper. We describe different checkpoint/restart approaches focusing on persistent solutions. We discuss different kinds of checkpointing as well as examining some checkpointing tools such as BLCR, FTI or SCR.

Checkpoint/restart is a traditional mechanism for protecting application data by regularly saving it. In this way, when an error occurs the data can be restored, and the program is able to benefit from previous work instead of restarting from scratch. This mechanism can be implemented by addressing soft errors or hard errors. For soft errors, the checkpoints are often performed in memory (non-persistent) whereas for hard errors they are done in storage (persistent).

Several different checkpoint/restart approaches exist. Checkpoint/restart can be application-level or system-level depending where it is implemented. Furthermore, there are persistent approaches and diskless approaches which depend on whether or not the data is stored in a persistent way.

Diskless Checkpointing [8] emerged in order to remove stable storage from checkpointing, thereby eliminating the main source of overhead. However, this approach is less resilient than its persistent counterpart since it does not tolerate complete system failures such as a power outage. Besides, it introduces overhead in the memory, processor, and network.

Regarding persistent approaches, there exist both system-level and application-level solutions. Some relevant system-level checkpoint/restart tools are [9], [10], [11], [12]. In this kind of checkpointing, we wish to highlight Berkeley Lab Checkpoint/Restart [7]. On the one hand, this tool developed at Lawrence National Berkeley Lab offers transparency as its main advantage, so no changes are required in the application code. On the other hand, the main drawbacks are the greater overhead usually introduced in comparison with application-level solutions, as well as the low space efficiency.

There exists a hybrid system/application-level checkpoint restart proposed by Bronevetsky et. al. [13] for shared memory programs. Even though they call it an application-level, it is very similar to system level approaches because the user does not decide which data must be checkpointed or how often a checkpoint is being performed. The user only places some calls to a given method to indicate that a checkpoint may be taken at some point, but then the system is saving the heap, the call stack, and the local and
global variables. User liberty is too restricted to consider it a pure application-level. Additionally, in most applications, there is no need to save all the information but just a subset in order to restart the application correctly. Hence, it is usually a source of overhead in terms of performance and storage space.

At the application-level, different libraries and tools also exist that offer different solutions. There are several checkpointing systems [14], [15], [16], [17], some at a single level and a few offering multi-level approaches. Multi-level checkpointing is key when one consider the gap between CPU and I/O performance, especially when one writes in the Parallel File System (PFS). An application that only writes checkpoints at PFS can easily introduce large amounts of overhead [18], [3], [19], [2], [20]. At multi-level solutions [21], [22], these problems are alleviated by writing the checkpoints in RAM disks, local node storage, or SSDs. In summary, multi-level proposals seek to reduce the overhead by reducing the frequency of checkpointing in the PFS, thus keeping the checkpoints in more efficient components and moving them only when required in an asynchronous and transparent way. Most popular multi-level checkpoint/restart libraries are Fault Tolerant Interface (FTI) and Scalable Checkpoint Restart (SCR).

As previously mentioned, FTI [23] is a multi-level checkpoint/restart library. L. Bautista-Gomez et al. proposed a low-overhead solution where they also integrate several redundancy schemes such as Reed-Solomon or XOR encoding for enhanced reliability. A distinctive feature of FTI is what the authors call a Fault-Tolerance dedicated thread. This is a thread per node devoted only to performing fault-tolerance tasks, hiding the overhead of encoding and some other resiliency-related tasks. FTI is able to achieve a high-performance when compared with other solutions, and that is crucial for us since we wish to use it as one of our backend libraries. Note that FTI is unable to have more than a single checkpoint alive.

SCR [24], proposed by A. Moody et al., is another tool that implements a multi-level checkpoint/restart approach. The authors also offer different redundancy schemes, the most efficient but less reliable one being the LOCAL scheme, where the data is simply stored in the local node storage without any redundancy. The intermediate one is PARTNER, in which the data is stored in the local node and a partner. Finally, as FTI, they offer the XOR scheme which performs a XOR encoding. In contrast to FTI, which provides a fixed number of checkpoint levels, SCR can have an arbitrary number of levels. Similarly to FTI, SCR also provides high-performance checkpoint/restart, so we included it as an underlying library in our solution. Like FTI, SCR only allows a single checkpoint alive.

Our approach is different from FTI and SCR because we work on a different layer. In fact, our solution complements these libraries. These libraries offer I/O optimizations and redundancy schemes with an improvable API and poor portability across systems while we offer a straightforward API and portability. In combination, the incorporation of these libraries into our approach enables us to provide a portable, easy-to-use, resilient and high-performance solution.
O. Subasi et al. already proposed a checkpoint/restart mechanism in OmpSs[25], but it was a non-persistent approach, contrasting with our persistent approach. Hence, only soft-errors were addressed, offering no support for fail-stop faults. Concretely, that approach does a checkpoint of the task dependences in memory. Then, if a task fails during its execution, the mechanism restores the data required to be executed and issues the reexecution of the task.
OmpSs Parallel Programming Model

The current project extends the OmpSs parallel programming model [26][27] with a new feature. For that reason, we provide this chapter explaining the model and the main pieces of software implementing the model to ease the understanding of the work done in this project.

A brief introduction and the philosophy behind OmpSs is the first section of this chapter. After that, Section 3.2 provides a description about how to express parallelism using OmpSs. Following, Section 3.3 explains the available mechanisms to synchronize the parallel parts of the code. Details about developing applications for heterogeneous systems and disjoint memory address spaces are explained in Section 3.4. Finally, Section 3.5 points out how OmpSs have influenced in the OpenMP standard.

3.1 Introduction and philosophy

The OmpSs 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. The OmpSs environment developed at BSC is composed of the Mercurium and the Nanos++ runtime library.

OmpSs tries to provide an environment where developing applications for modern High-Performance Computing (HPC) systems becomes easier while maintaining a reasonable performance. On the one hand, ease of use is something difficult to measure, but OmpSs’ design has been done using principles praised by their effectiveness in that area. On the other hand, reasonable performance means delivering similar performance to other programming models targeting the same architectures. Overall, ease of use and reasonable performance makes OmpSs a productive programming model.

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 like the ones shown in Figure 3.1. 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. Productivity increases with this methodology because of error detection, as well as performance analysis, becomes easier.

OmpSs’ methodology (inherited by OpenMP) tries to hide architectural details to prevent users from managing low-level resources by handling them inside a runtime library. This approach contrasts with more explicit approaches that expose the resources to users, forcing them to manage low-level nuances. An example of this explicit programming models is MPI. MPI requires coding inside the application part of the network logic to exploit parallelism. The main drawback of this approach is the dependence on the underlying technology. In other words, a specific implementation is bound to a single technology, reducing portability across systems and dificulting the debugging and testing process.

Also following the compiler directives philosophy, StarSs is a family of programming models developed at BSC. Contrasting with OpenMP, which uses a fork-join model, StarSs uses a thread-pool execution model. Furthermore, StarSs targets heterogeneous architectures while OpenMP targets only shared memory systems until version 4.0. Finally, asynchronous parallelism is the main mechanism of expressing parallelism in StarSs. This feature was not supported in OpenMP until its version 3.0.

StarSs provides much implicitness. The reference is OpenMP where users must explicitly specify the parallel regions, express how the inner code of the parallel regions has to be executed and synchronize the different parts of the code. StarSs removes the need to
declare the parallel parts, because, given its execution model, the threads are created at the beginning of the execution. Moreover, the parallel code is defined using pieces of code that can be asynchronously executed in parallel called tasks. A task can be annotated with data dependences to specify the data a task is using and producing. With this information, the runtime decides which tasks can be executed in parallel and which tasks must wait for others. Using this mechanism, a much more fine-grained synchronization is achieved resulting in a more efficient use of the system resources.

Combining the best features of StarSSs and OpenMP, OmpSs tries to be the evolution that OpenMP needs to target new architectures and systems.

### 3.2 Parallelism

Note that a thread-pool execution model implies that a parallel region is automatically created at the execution start. Therefore, users just have to annotate those pieces of code that should be run in parallel.

#### 3.2.1 Tasks

The main computation unit of OmpSs, although it supports OpenMP parallel loops, is the task. Tasks are independent pieces of code that can be executed in parallel. Whenever there is a piece of code annotated as a task, the code will not be executed, but a task instance will be created. Then, the OmpSs’ runtime library is in charge of eventually executing the task.

In OmpSs, we distinguish two types of tasks depending on where the task construct is placed. There are outline tasks and inline tasks. Outline tasks are those with the task construct in the function declaration/definition. Inline tasks are those with the task construct inside the function code. Note that a task construct placed in the function declaration/definition applies to all the function calls. Therefore, in Figure 3.2, where the function Foo is called ten times, ten different tasks will be created. The task construct can also be placed before the function call, as in Figure 3.3, and the effect will be the same, a task instance for each call. Inline tasks can also be applied to structured blocks and not only to a function call. An example of this is shown in Figure 3.4 where each task instance calls to gen, compute and print functions. The piece of code that will be executed as a task is delimited by curly braces or curly brackets.

Note that tasks cannot have return value. In other words, the return must be void. In consequence, if you apply the task construct to a function, the function must be void.
```c
#pragma omp task
void foo (int Y[size], int size);

int main () {
    int X[100];
    for(int i=0; i<10; i++)
        #pragma omp task
        foo(X, 100);
    #pragma omp taskwait
}
```

Fig. 3.2: Snippet of code using outline tasks.

```c
int main () {
    int X[100];
    for(int i=0; i<10; i++)
        #pragma omp task
        foo(X, 100);
    #pragma omp taskwait
}
```

Fig. 3.3: Snippet of code using inline tasks.

```c
int main () {
    int X[100];
    for(int i=0; i<10; i++)
        #pragma omp task
            { 
            gen(X, i);
            compute(X, i);
            print(X, i);
        }
    #pragma omp taskwait
}
```

Fig. 3.4: Snippet of code applying inline tasks to a structured block.
Fig. 3.5: Snippet of code applying for directive.

3.2.2 Loops

OmpSs also inherits from OpenMP the for directive. This directive must be applied to a for loop (in C or C++). In OmpSs, the iterations become tasks. The number of tasks is determined by the runtime and depends on the number of available processing elements. The user can specify the scheduling using the `schedule` clause. An example can be observed in Figure 3.5.

3.2.3 Nesting or multi-level parallelism

An interesting OmpSs feature is the possibility to create nested tasks, resulting in multiple levels of parallelism that can help some kinds of applications to perform better. This simply means that tasks can also create tasks. With this feature, appears the concepts of child tasks, ancestor tasks and parent tasks. If a task $t_a$ creates a task $t_b$, $t_a$ becomes the parent of $t_b$ and $t_b$ is a child of $t_a$. Ancestor tasks is a set formed by your parent and all of its ancestor tasks. For instance, if now we have a third task $t_c$ created by $t_b$, its ancestors are $t_a$ and $t_b$. This feature’s main advantage is providing users the ability to implement parallel recursive algorithms.

3.3 Synchronization

Often, when using parallelism, synchronization is required to assure correctness due to the usual dependences among tasks. OmpSs provides two mechanisms in order to perform synchronization. They are data dependences and explicit synchronization points.
3.3.1 Data dependences

The task construct accepts the clauses `in(memory-reference-list)`, `out(memory-reference-list)` and `inout(memory-reference-list)` to specify read-only, write-only and read/write data dependences, respectively. Often, a task uses data produced by a former task, performs some computation and produces new results for a subsequent task or program block. With the above-mentioned clauses, it is quite easy for users to specify this kind of behaviors. An example of the syntax can be observed in Figure 3.6.

Using this feature, users indicate the runtime which data a task must wait for and which data a task must signal its readiness. The model is not performing any code analysis, so the correctness of the data dependences is the programmer responsibility. For instance, if a task has three different references as input, but it actually uses only two, the runtime won’t schedule the execution of the task until all of them are ready.

At each task creation, with the information specified by the user, the runtime matches the data dependences with those of existing tasks. If a data dependency between tasks is found, the new task becomes a successor of the corresponding tasks, and there is a task dependence. In other words, there is a task dependence between $t_a$ and $t_b$ if:

- $t_a$ is created before $t_b$ and both have the same parent task.
- $t_b$ has a memory reference overlapping with a memory reference of $t_a$ and at least one of these references is specified using an `out` clause.
int main () {
    int X[100], Y[100], V[100];
    #pragma omp task in(X) out(Y)
    {
        #pragma omp task inout(X,V)
    }
    #pragma omp taskwait
}

Fig. 3.7: Snippet of code showing illegal memory references in nested tasks.

Therefore, a task is not scheduled for execution until all its predecessors have finished. Regarding the references to be declared as data dependence, there are some constraints:

- A task can only reference memory also referenced by its parent or memory that is not referenced by any other task.

- Data directionality must be consistent with the directionality specified by its parent.

For instance, in Figure 3.7 the nested task cannot reference $V$ because its not referenced by its parent task. In addition, the directionality of $X$ in the parent task is in, but the inner task specifies it as inout. This is also wrong because the outer task specifies that $X$ will only be read, but the inner task specifies that it will also be written.

There is an extra clause applicable to the task construct that modifies the behavior of the data dependences. This is the concurrent clause. When using this clause, the synchronization between tasks is relaxed: concurrent dependencies respect in, out or inout dependencies, but they does not respect other concurrent dependencies. For instance, see Figure 3.8. The reduce task will not start until generate ends, but the inner tasks created inside the reduce function will execute concurrently, so users must ensure synchronization.

Expressiveness is an important issue when dealing with data dependencies. For making the most of the available resources, users must specify dependencies as well as possible. For that purpose, OmpSs allows C/C++ extended lvalues in the dependence clauses as well as two extensions. Array sections feature permits users to refer to multiple elements of an array or pointer data in a single expression. Shaping expressions feature give users the ability to recast pointers in order to recover the size of dimensions that may be lost across calls.

There are two ways to refer to multiple elements of an array or pointer data. They are listed below:
Fig. 3.8: Snippet of code using data dependences.
void generate ( int V[size], int start, int size );

int main () {
    int X[100];
    int size = 100;
    // Option 1
    #pragma omp task out(X[0:(size/2)-1])
    // Option 2
    #pragma omp task out(x[0;size/2])
    generate_half( X, 0, size );
    // Option 1
    #pragma omp task out(X[size/2:size-1])
    // Option 2
    #pragma omp task out(X[size/2;size-(size/2)])
    generate_half( X, 50, size );
    #pragma omp taskwait
}

Fig. 3.9: Snippet of code using array sections feature.

1. array[lower:upper]. When using this option, all the elements in the range, including both lower and upper, are referenced. Note that if no lower is specified, it is assumed to be 0. In a similar way, if no upper is specified, it is assumed to be the last element.

2. array[lower:size]. This option includes all the elements from lower to lower+size-1, both included.

Shaping expressions can be used by adding one or more times [size] expressions before a pointer.

Figure 3.9 presents examples using array sections feature and Figure 3.10 presents an example using shaping expressions feature.

3.3.2 Explicit synchronization points

OmpSs also provides users the possibility of introducing explicit synchronization points in the program code. For that purpose, there is the taskwait directive. Whenever a program arrives at a taskwait, it will wait there until the completion of all the previously created child tasks. There is the possibility to add the clause on(memory-reference-list) to a taskwait. Using this clause, this taskwait will only wait for all the tasks producing the specified data.
3.4 Heterogeneity

One key feature of OmpSs is the ability to handle disjoint address spaces. This feature enables OmpSs to work in systems with distributed resources, like clusters of SMPs, or even heterogeneous systems composed by accelerators with private memory.

3.4.1 Single unified address space

OmpSs presents a single unified address space hiding the multiple different address spaces that actually exists in a system. Figure 3.11 shows this behavior. With this feature, OmpSs programs can run on different system configurations without further modification required.

For a proper support of systems with disjoint address spaces, OmpSs needs from the programmer to specify the data accessed by the tasks. Often this information matches
with the data dependences, but there are some cases where this is not happening. Hence, OmpSs provides two distinct mechanisms.

In order to specify this information, there is a set of directives available. They are:

- **copy_in**(memory-reference-list) Data specified inside this clause must be available in the address space where the task executes. Withal, this data is read-only.

- **copy_out**(memory-reference-list) Data specified inside this clause will be generated in the address space where the task executes, so it must be copied back to the host address space. Additionally, this data is write-only.

- **copy_inout**(memory-reference-list) Data specified inside this clause must be available in the address space where the task executes. Moreover, it will be updated. This data is read/write.

- **copy_deps** This clause means that dependencies clauses are also understood as copy clauses.

Note that a task can use only local variables or data specified as a copy. In addition, as happens with data dependences, memory references not specified in the parent task cannot be used in a child task and the directionality of the child task must not conflict with the directionality specified in the parent task.

An example using this feature is provided in Figure 3.12. In that program, there are two different tasks. Note that the first one is annotated with `target device(smp)`, so it must run in a CPU while the second one is annotated with `target device(cuda)` which means that it will run on a CUDA GPU. Taking into account this, the system running the program must have at least a CPU and a GPU.

Regarding the transfers between the host and the GPU, the runtime library of OmpSs is the one in charge of performing the required transfers to produce a correct result of the execution. Note that users do not need to reference different address spaces, there is a single unified address space that we call *the host address space*.

Given that all the data transfers are internally done by the runtime library, a parent task cannot access child tasks data before a synchronization point. Otherwise, the data status will be undefined because the execution or the transfers may not be completed yet. The runtime library guarantees that the correct data will be available after a synchronization point. See an example in Figure 3.13. Being the parent task the implicitly created task that executes the main and the child task the one created in line 4, the first access done in line 8 is illegal because the task may not be completed yet. The access in line 13 is legal because the synchronization point in line 10 guarantees the correctness of the data.

Nevertheless, there is a modifier clause that can be applied to the `taskwait`. The `noflush` clause avoids spreading the data generated in a single address space to the other
int main () {
    int X[100];
    int Y[100];
    for(int i=0;i<100;i++) {
        #pragma omp target device(smp)
        #pragma omp task copy_inout(X)
        computeCPU(X);
        #pragma omp target device(cuda)
        #pragma omp task copy_inout(Y)
        computeGPU(Y);
    }
} #pragma omp taskwait

Fig. 3.12: Snippet of code presenting copies mechanism.

int main () {
    int x[100];
    #pragma omp target device(cuda)
    #pragma omp task copy_inout(X)
    computeGPU(X);

    // Illegal access, undefined data status
    int y = X[0];

    #pragma omp taskwait

    // Legal access
    int z = X[0];
}

Fig. 3.13: Snippet of code showing illegal accesses from the parent task.
```c
int main () {
    int x[100];
    #pragma omp target device(cuda)
    #pragma omp task copy_inout(X)
    computeGPU(X);

    // Illegal access, undefined data status
    int y = X[0];

    #pragma omp taskwait noflush

    // Still illegal access
    int z = X[0];

    #pragma omp taskwait

    // Legal access
    z = X[0];
}
```

Fig. 3.14: Snippet of code showing illegal accesses from the parent task using noflush clause.

address spaces. This is useful when a very large amount of data must be updated because this operation will be very costly. Therefore in a `taskwait noflush`, the runtime will wait for all the child tasks, but it will not synchronize data across separate address spaces. In the example shown in Figure 3.14, the first `taskwait` in line 10 does not synchronize data although the task has completed, so the access in line 13 is illegal yet. Notwithstanding, the access done in line 18 is correct because of the `taskwait` in line 15.

### 3.4.2 Architectures supported

Nowadays, OmpSs supports GPUs running with CUDA and OpenCL, FPGAs, clusters of SMPs and clusters of GPUs. For marking a task to be executed by a specific device, users must use the `target device(smp/cuda/opencl/fpga)` clause. If no specific target is indicated, it is assumed to be smp. Note that a task marked to be executed by a specific hardware requires the presence of the specified hardware in the system.

Combining this feature with the previously explained management of disjoint memory address spaces, OmpSs is able to increase productivity when comparing with coding using device-specific approaches like CUDA because users do not have to manually
perform data transfers, synchronizations, and other low-level architecture dependent nuances.

### 3.5 Influence in OpenMP

OmpSs is a forerunner of OpenMP. Many OmpSs features have been introduced into the OpenMP standard.

The starting point is OpenMP 3.0, on May 2008, when support for asynchronous tasks was included. The reference implementation to measure the benefits of using tasks was developed at BSC. Later on, OpenMP 4.0, which was released in July 2013, extended the tasking model to support data dependences, a feature previously available in OmpSs, allowing users to define fine-grain synchronization among tasks. The latest version of OpenMP, OpenMP 4.5, introduced the `taskloop` construct. This feature was already available in OmpSs, and, in fact, the BSC implementation was the reference implementation. Moreover, the priority clause was added to the standard to be used in combination with `task` and `taskloop` constructs.

OmpSs is working in new features to be added in upcoming OpenMP versions such as task reductions or new extensions to the data dependences of the tasking model.

Figure 3.15 summarizes the contributions that OmpSs has done to OpenMP.

---

Fig. 3.15: Summary of the contributions that OmpSs has done to OpenMP.
4 | Tools and Methodology

This chapter covers the main tools to develop the project as well as the methodology followed.

4.1 Tools

The project’s goal is to extend OmpSs to support application-level checkpoint/restart. As stated in Section 3.1, the environment developed at BSC is composed of the Mercurium compiler and the Nanos++ runtime library. Therefore, these are two tools used in the project.

Moreover, our design relies on a couple of state-of-the-art checkpoint/restart libraries that will act as backend libraries. Those libraries are FTI and SCR.

Finally, in order to evaluate the approach developed in this project, we will use Extrae and Paraver. Respectively, they are a tracing tool and a visualization tool.

The rest of this section describes each of the previously mentioned tools.

4.1.1 Mercurium

Mercurium [28] 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. 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 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.

Mercurium structure is shown in Figure 4.1.

### 4.1.2 Nanos++

Nanos++ [29] is a runtime library designed to be used in parallel environments. Nowadays, its main uses are OmpSs and OpenMP although it supports some more parallel programming models.

This library provides support for asynchronous task-based parallelism relying on data-dependences. It also provides support for heterogeneity by maintaining coherence across different disjoint address spaces by means of a directory and cache modules.

Nanos++ main goal is to be used in research about parallel programming environments. To that end, Nanos++ has an easily extensible design based on plugins. Some currently available plugins are:

- Task scheduling policy
- Thread barrier
Nanos++ can be instrumented by using its Extrae support which can generate execution traces for being visualized with Paraver. Also, Nanos++ is able to draw a task dependency graph of your application where you can see the potential parallelism in each phase of the code.

Nanos++ provide an API. Notwithstanding, its preferred to be used in combination with Mercurium by using some of the supported programming models.

Nanos++ structure is shown in Figure 4.2.

4.1.3 FTI

Fault Tolerance Interface (FTI) is a library aiming to give domain specific scientists a way of performing multilevel checkpointing in large-scale supercomputers efficiently and fast. This library provides data replication and multiple redundancy schemes to provide several levels of reliability. Moreover, it leverages local storage to deliver good performance. FTI offers application-level checkpoint/restart enabling users to decide which data must be checkpointed avoiding waste of disk space, time and energy.
4.1.4 SCR

Scalable Checkpoint Restart (SCR) is a library developed at Lawrence Livermore National Laboratory. Similarly to FTI, it is a library providing multilevel checkpointing in large-scale supercomputers. This library also provides several levels of reliability by means of redundancy schemes as well as competitive performance thanks to using advanced storage resources like RAM or Flash. Users can select the data to be stored reducing the overhead of disk space, time and energy.

4.1.5 Extrae

The Extrae [30] library is a software meant to generate execution traces which can be later visualized with Paraver. It uses several distinct interpose mechanisms in order to inject elements in the application code with the final goal of collecting performance information. It supports several programming models, namely MPI, OmpSs or CUDA.

A trace execution is very valuable to analyze the performance of an application. Instrumenting an OmpSs code can provide you information such as the total number of tasks, how much time a task is running or waiting, communications among tasks, and more useful information. This tool is especially useful when a program doesn’t present the expected performance because looking at the trace users can easily detect where are the problems.

4.1.6 Paraver

Paraver [31] tool is a visualization and analysis tool. It provides visualization of the traces generated by Extrae. It offers great flexibility to explore the information retrieved by Extrae deeply. In addition to the possibility of detecting performance problems, Paraver enables users to understand applications’ behavior.

4.2 Methodology

This project follows the Design Science Research Method or Design Research Method. Design research is an iterative methodology based on pragmatic research aimed to solve real-world problems by means of artifacts. This paradigm provides a set of guidelines for executing as well as evaluating research. In fact, the evaluation and the execution feedback each other. There is a first cycle where the requirements are defined, and the problem stated. Then, iteratively, a design is done, implemented and evaluated. After each evaluation step, a new design is built improving the weak points pointed out by the evaluation. This process is repeated as many times as needed until all the requirements are fulfilled.
One of the most relevant visions of the Design Science Research Method is the *Three Cycle View*. It is a way to join the different steps in the design research process into three major groups or cycles of closely related activities. They are the Relevance Cycle, the Rigor Cycle, and the Design Cycle. The relevance cycle starts the design science research analyzing the requirements for the research and defining the acceptance criteria for the research results. The rigor cycle introduces the previous knowledge into the research ensuring that the resulting artifact will be innovative and that the researchers reference the existing knowledge as a base. The Design Cycle is the central one, and it iterates between building the designed artifacts and evaluating it and processing the research [32]. In Figure 4.3, the three cycles can be seen as well as how Relevance and Rigor Cycles introduce information to Design Cycle.

![Fig. 4.3: Three cycle view of Design Science Research [32]](image)

The rest of this section details each of the three cycles previously mentioned.

### 4.2.1 Relevance Cycle

The Relevance Cycle is the starting point with the goal of improving the existing environment by designing and building new artifacts. It is about identifying new opportunities to improve existing systems or recognize problems to fix them. These opportunities/problems must be relevant for the people or organizations, which are the application domain of the project environment. All of these are considered as requirements for the design research method [32]. This information flow can be seen in Figure 4.4 where the Relevance cycle is the one which iterates getting information from the environment and introducing the requirements information in the Design Cycle.

The Relevance Cycle also defines the acceptance criteria for the ultimate evaluation of the research results. These criteria define how to measure the artifact improvement of
Fig. 4.4: Relevance Cycle of Three cycle view [32]

the environment and, in fact if the artifact is an improvement or not. These evaluations
must be returned to the environment in order to be studied and evaluated against
the application domain. Moreover, this study and evaluation determine whether more
iterations of the Relevance Cycle are needed since the new artifact may have deficiencies
in its functionalities or it is required to develop new features. Another possibility is the
incorrectness of requirements defined in the Design Cycle. This will be seen if the
artifact satisfies the requirements, but it does not fill the opportunity or problem. In
this case, another Relevance Cycle iteration is needed to redefine the requirements for
the Design Cycle [32].

4.2.2 Rigor Cycle

The Design Cycle analyzes different types of existing backgrounds at the beginning of
the project that should be taken into account in the Design Research Method:

- Scientific theories.
- Engineering methods.
- Experiences and expertise defined by the state-of-the-art of the artifact application
domain.
- Existing artifacts and processes in the application domain.

The goal of Rigor cycle is to provide previous knowledge of the application domain in
the project to guarantee that it is innovative. It forces the researchers to thoroughly
research and reference the previous knowledge in order to assert that the new artifacts
are research contributions and not routine designs based upon the application of well-
known processes [32].
The different types of explored knowledge base have different proposals. The scientific theories and the engineering methods must be considered in the artifact construction and evaluation. However, find the appropriate ones may be laborious or simply impossible since the new artifact may be absolutely innovative and it is only inspired in some problems/opportunities from the Relevance Cycle. The experiences, expertise and previous artifacts must be considered in the Design Cycle to really innovate and not re-design the wheel. Moreover, they can be very useful for the project in order to avoid known issues or common problems in the application domain.

The Design Research Method results generate more content than the artifact. As the previous artifacts and previous experiences are considered in the design and evaluation of the new artifact, the construction process and evaluation on the new artifact generate new experience in the application domain. All these new knowledge may include theories extensions, new methods or extensions to the existing ones and experiences gained from the field testing. These research contributions are very important in the academic context and for the academic audience but they are too for the practitioner audience [32].

Figure 4.5 completes the image of the Three Cycle View with the information analyzed from the knowledge base in the Rigor Cycle that is used as input information in the Design Cycle.

### 4.2.3 Design Cycle

The Design Cycle is the main cycle in the Design Research Projects. It iterates between the design and construction of the new artifact and the evaluation of it, always taking into account the information provided by the Rigor and the Relevance cycles [32]. Some researchers view this iterative process as a generation of multiple design alternatives and evaluating them against the requirements until all of them are satisfied [33]. As it uses
the requirements from the Relevance Cycle for evaluating the designs and also uses the theories, methods, expertise, etc. from the Rigor Cycle, the Design Cycle depends on both. However, it is independent of those two cycles during its internal iteration [32].

The construction and the evaluation of the artifact inside the Design Cycle have the same importance and must be hardly based on the relevance and rigor. If some part lacks, the whole artifact will be weak. An artifact with a strong background in its construction won’t be a strong artifact without a meticulous evaluation and testing. Moreover, having a strong evaluation will not provide a good artifact since the construction may be a mess [32].

As we mentioned above, the Design Cycle returns valuable information to the Relevance and Rigor cycles. On the one hand, it provides the artifact and the results of the artifact evaluation according to the acceptance criteria to the Relevance Cycle in order to check the value of the artifact in its application field. On the other hand, it provides experience, expertise, method improvements and theories extensions directly to the Rigor Cycle or directly derived from the work done.

---

Figure 4.6 has the complete picture of the three cycles with all the previous information. It contains the information provided from one cycle to the others and the main two steps inside the Design Cycle.
This chapter covers the details of the proposed work. On the one hand, the high-level design of the approach will be explained. The design is very important because it enables users to express application-level checkpoint/restart in an abstract way by means of compiler directives, enhancing productivity and portability. On the other hand, implementation details will be offered. Our implementation produces no overhead compared to the direct use of the backend libraries and the details offered will help to understand why this is possible.

Note that this section fits into the Design Cycle of the Design Research Method explained in Section 4.2.3.

5.1 Design

This section consists of a discussion about the high-level design of our approach, the main goal of which is to provide a straightforward and portable way of performing application-level checkpoint/restart, abstracting users from the implementation details and allowing them to concentrate on the application itself. We achieve these objectives by means of three components: a compiler, an intermediate library, and a backend library. Figure 5.1 shows our three-layer architecture.

In our three-layer solution, Mercurium, the compiler, is the one in charge of transforming the directives placed by the user in calls to the API of our intermediate library. The intermediate library is Nanos++ runtime library which forwards the calls to the backend library API. This intermediate library also serializes and deserializes the data in the store or restore, respectively. Finally, a backend library, nowadays FTI or SCR, actually performs the checkpoint or the restore.

Our mechanism checkpoints all the input (inout are included) dependences of a task annotated as checkpoint. When restarting, OmpSs transparently detects that the execution is a restart and fast-forwards the application to the point where it crashed last time and restores the appropriate data.

The rest of the section is structured as follows. Firstly, we present the directives and clauses of the #pragma chk family; then the functionality is explained, and finally, we
offer some relevant remarks that should be taken into account by users when using our approach.

The rest of the section is structured as follows. Firstly, the new syntax and its semantics included in the OmpSs programming model are detailed; then, the functionality is explained, and finally, we offer some relevant remarks that should be taken into account by users when using our approach.

5.1.1 New semantics

In order to support the new checkpoint/restart functionality, we added to OmpSs the `checkpoint(opt-cond-expr)` clause. This new clause must be applied to a task construct and combined with the dependencies semantics of OmpSs. Concretely, a `checkpoint task` will store the input (or inout) dependencies. An example of this semantics is provided in Figure 5.2 where `input_data` is checkpointed. The checkpoint clause accepts an optional condition expression. Depending on whether the user provides a condition or not, the checkpoint frequency changes:

(a) Without condition: the runtime relies on the underlying library to decide whether it needs to checkpoint or not. In other words, the checkpoint frequency is given by the underlying checkpoint/restart library. This option is presented in Figure 5.2.

(b) With condition: there will be a mandatory checkpoint whenever the provided condition becomes true. On the other hand, whenever the condition is false, there will be no checkpoint at all. This preference is shown in Figure 5.3 where a checkpoint will be performed every five steps.
for( int step = 0; step < N_STEPS; step++) {
    #pragma omp task in(input_data) checkpoint()
    task_body(input_data);
}

Fig. 5.2: Snippet of code presenting new OmpSs checkpoint/restart semantics

for( int step = 0; step < N_STEPS; step++) {
    #pragma omp task in(input_data) checkpoint(step%5==0)
    task_body(input_data);
}

Fig. 5.3: Snippet of code presenting new OmpSs checkpoint/restart semantics providing a condition expression

5.1.2 Functionality

We offer the same functionalities and performance as current solutions, but with an easy-to-use interface and enhanced portability. Therefore, we provide a mechanism that is able to checkpoint and restore specified data, on request. This kind of mechanism enables checkpointing at a given frequency, in large applications, in order to mitigate failure effects. Furthermore, it is also possible to restore the last checkpoint, on request. This is useful for resuming failed executions from this data instead of from scratch.

The OmpSs application-level checkpoint/restart functionality is designed using FTI and SCR as backend libraries. Whenever a checkpoint has to be done, OmpSs calls FTI/SCR to store the appropriate data. With respect to the restart, when OmpSs realizes that an application counts with checkpoints of previous executions, it is able to fast-forward the application until the point where the application stored its data last time by skipping all the tasks until reaching the last checkpoint. Once it is in the last checkpointed task, it restores the data by means of FTI/SCR and follows the execution normally. Figure 5.4 summarizes the behavior of the mechanism. In the figure, there are represented three executions: the reference execution with no checkpoints; the execution doing checkpoint/restart with FTI/SCR; and, finally, the run performing checkpoint/restart with OmpSs. As it can be seen, the main difference between native FTI/SCR and OmpSs is the fast-forward part. In Section 6, where we provide some traces of real executions where this theoretical behavior is shown, it can be seen that the fast-forward part is negligible.
5.1.3 Considerations and Remarks

Applications intended for use in the proposed application-level checkpoint/restart mechanism must take into account some important points. The model is very simple, which is a key factor in providing users with a straightforward interface. However, since parallelism always makes things harder, in this section we make some considerations and remarks to be taken into account when using the mechanism in parallel applications.

Most of the applications applying checkpoint/restart are implemented using MPI. In fact, these kinds of applications are more error-prone because they are spread among several nodes. Thus, since the likelihood of an error is greater, it makes sense to apply fault tolerance.

Applications using MPI usually partition the data among different processes, and since there is no shared memory space, each process must checkpoint its own part of the data. This fact merits two important considerations:

1. The checkpoints must be synchronized. In other words, all the processes must check if the rest have carried out a successful checkpoint before continuing the execution. If just one process in the checkpointing fails, all the processes must mark the checkpoint as invalid. It is crucial to ensure that this occurs in MPI programs, because if just one process remains unsynchronized, it will have some
lost MPI messages when restarting, which will probably result in a deadlock. This is already guaranteed in our mechanism by the backend libraries.

2. When restarting, each MPI rank must recover the data from the same rank in the failed execution; this is also guaranteed in our solution. When using SCR, it is guaranteed in the intermediate library, and when using FTI in the FTI library itself.

Our backend libraries are designed to deal with MPI applications. In fact, they are developed on top of MPI, and for that reason, they are able to deal with the afore-mentioned requirements. However, they are not prepared to handle intranode parallelism automatically even though this also presents some challenges:

1. Application-level checkpoint/restart requires saving the whole state of the program to achieve a proper restart if a crash occurs. In parallel executions, many different threads are participating concurrently in this state. Nevertheless, this constitutes a problem when checkpointing data because, potentially, a thread that is different from the one performing the checkpoint may potentially be modifying the data in such a way that the stored data is not the correct one. In order to avoid this drawback, the user must guarantee that nobody else is modifying the data when the checkpoint is being performed.

2. Note also that synchronization between threads is necessary. Everybody must have reached the point where the checkpoint is being carried out. Otherwise, they may be checkpointing outdated data, and thus the restarted application will not work properly (see Figure 5.5 to find an example). Users must ensure that Phase 2 does not start until Phase 1 is completed by all the threads. Otherwise, the checkpointed data may be incorrect.

These challenges are automatically solved in our mechanism by design. A checkpoint task stores all the data specified as input or inout. It also implies that this task cannot be scheduled for execution until all the producers of such data have finished. Moreover, no other successor tasks can be updating this value because of the OmpSs data dependences model too. Therefore, both challenges are solved since the model guarantees nobody else is modifying data and also assures that all the data producers have already produced the correct data when the checkpoint starts.

We would like to remark that our backend libraries only support a single checkpoint alive. Consequently, we also support only one checkpoint. In other words, only a single task construct can contain the checkpoint clause. Note that we say a task construct, this means that the whole code can only contain the checkpoint clause once, but this task can be instantiated several times. Each instance of the checkpoint task will update the checkpoint information with the newest data.

With the possibility of having several instances of a checkpoint, there may be a trouble. Our backend libraries can just process a checkpoint at once, so our checkpoint tasks
cannot be executed concurrently. This can be easily solved by specifying an inout
dependence into the checkpoint task. This way, a checkpoint task must wait for the
previous one to be executed.

Users must wait for the checkpoint tasks either implicitly, using data dependences, or
by means of explicit synchronization points (taskwait). Although they are special tasks
that may perform checkpoints, they are tasks and they must be synchronized.

As regards to the restart phase, OmpSs fast-forwards the application until reaching the
point were the former execution took the last checkpoint. This is done skipping all the
tasks until that point. Taking into account this fact, users must guarantee that there
are no tasks with side effects, such as preprocessing tasks reading data from files or
similar, because they will be skipped too.

Figure 5.6 shows a working example of an N-Body code using OmpSs application-level
checkpoint/restart taking into account all the considerations and remarks. Note that
the checkpoint task is encompassing all the body of the loop, with the whole status
as inout dependence so that just one checkpoint task can run concurrently and the
appropriate data is checkpointed. Regarding the way the dependence is expressed in
this task, OmpSs allows users to express array sections as dependences. For instance,
[n_blocks] local is expressing a dependency over an array of size n_blocks whose
first element is local[0]. As it contains nested tasks, a taskwait is placed at the end
of the checkpoint task in order to wait for nested tasks. Another taskwait is required
after the loop to wait for the last checkpoint task. Finally, consider the fact that there
are no tasks before the main loop, so nothing important will be skipped.

5.2 Implementation details

This section provides some insight into the implementation details of our proposed solution. We have implemented our proposal in our Mercurium C/C++ and Fortran source-to-source compiler. We have also implemented some modifications on our Nanos++ runtime library to act as the intermediate library that is between Mercurium and the backend libraries. Firstly, the changes effected at the Mercurium compiler level are explained. Secondly, we describe the Nanos++ runtime library modifications.

Note that, as mentioned in Section 2, the state-of-the-art checkpoint/restart libraries used as backend libraries in our solution do not provide support for more than one active checkpoint. Hence, we proceed in the same way: with just one alive checkpoint.

5.2.1 Mercurium compiler

The main modification regarding the compiler infrastructure is the new checkpoint clause that Mercurium has to process along with the task construct in order to enable the application-level checkpoint/restart functionality. Mercurium is the one in charge to process this clause and notify the runtime whether a task has the checkpoint modifier or not. Besides, is also a compiler work to pass the condition of the checkpoint clause, if there is any, to the runtime in order to perform the checkpoints when the user has specified. The compiler support has been developed for C/C++ and Fortran.

5.2.2 Nanos++ runtime library

There are three key aspects to cover with respect to Nanos++: (1) the way we get the data to perform the checkpoint; (2) how the data is stored and restored depending on the backend library; and (3) how the application is fast-forwarded in a restart. Those aspects are in-depth explained below:

- As we mentioned in Section 3.4.1, OmpSs is providing support for heterogeneous architectures. We have already explained that there is a copies mechanism which is used to perform the data movements between different disjoint address spaces. In the application-level checkpoint/restart we are taking advantage of the mentioned mechanism to extract the data from the user-level code and checkpoint it as well as restore it again in the restart phase so that the user-level code is able to work with the restored data in the rest of the execution without performing any effort.
```c
void solve_nbody(particles_block_t * __restrict__ local,
                 particles_block_t * __restrict__ tmp,
                 force_block_t * __restrict__ 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);

  const double start = wall_time();
  for (int t = 0; t < timesteps; t++) {
    #pragma omp task //CHECKPOINT_TASK
    inout([n_blocks] local) checkpoint()
    {
      particles_block_t * remote = local;
      for(int i=0; i < rank_size; i++){
        #pragma omp task
        in([n_blocks] local,[n_blocks] remote) \ 
        inout([n_blocks] forces)

        calculate_forces(forces, local,
                         remote, n_blocks);

        #pragma omp task
        in([n_blocks] remote) \ 
        out([n_blocks] tmp)

        exchange_particles(remote, tmp, n_blocks,
                            rank, rank_size, i, t);

        remote=tmp;
      }

      #pragma omp task
      inout([n_blocks] local, [n_blocks] forces)

      update_particles(n_blocks, local,
                       forces, time_interval);

      #pragma omp taskwait
    }
    #pragma omp taskwait
  }
} //pragma omp taskwait
```

Fig. 5.6: Snippet of code presenting N-Body application using OmpSs checkpoint/restart mechanism
• Once we have extracted the data in the way explained in the previous bullet, it is differently managed depending on the backend library. Notwithstanding, in both libraries the data is written directly in binary format and restored in the same way. Also, regarding data serialization, for both libraries, we are only storing the data itself, we do not need complementary information, like data size, because Nanos++ already has this information in the internal data structures synthesized with the information provided by the user in the dependencies. Then, when a restart is performed, the deserialization of the data is done using the Nanos++ internal information too. Regarding FTI, all the I/O is done by the library itself. Hence, Nanos++ is in charge of registering the data to be checkpointed and invoking the method to indicate that a checkpoint has to be performed or the method requesting a data restore. Contrasting with FTI, SCR requires Nanos++ to take some more actions, mainly related to I/O management. SCR interface provides a file descriptor, and we should interact with it using a POSIX I/O interface. So, for writing, we use the common write method, and for reading, we use memory mapped I/O. The file is mapped to the process address space, and we perform a memory copy from this address to the one where the user-code can access.

• Due to the fact that OmpSs is a task-based programming model, we are able to fast-forward the application skipping the tasks that were already executed in previous executions without additional overhead. The skipped tasks are not even created because when their creation is being requested to Nanos++, the runtime library detects that they were already executed and denies creation. Once the execution reaches the task that was the last checkpointed previously, Nanos++ turns back to the normal behavior and the tasks are normally created. We would like to highlight the importance of the no creation of tasks since it is a key point in terms of performance: if we were creating tasks and then skipping them in execution time, the overhead will be much higher.

Note that Nanos++ runtime library serializes and deserializes the data. This is a very important point because it constitutes a tedious and error-prone task for users and the use of our mechanism means that it can be performed with little or no effort.

Moreover, the runtime library detects whether an execution has previous checkpoints and is able to restore any data or run it from scratch. Hence, users do not need to modify the natural flow of the program to detect this, because the intermediate library does it for them.

In addition, Nanos++ also performs a further task, but only when the backend library is SCR, which is that it ensures that each rank restores the correct data when running an MPI program. It is SCR-specific because FTI already does it by itself.

Finally, we would like to highlight the extensibility of the developed mechanism. For the time being, we are only using FTI and SCR as backend libraries, but the addition of a new one is quite straightforward.
6 Evaluation

This section is intended to benchmark the performance of our approach when compared with other popular approaches such as FTI or SCR themselves. The structure of this section is as follows: firstly, the environment in which the experiments were conducted are described followed by the methodology employed. After that, an in-depth explanation of one application is provided. Finally, an evaluation and some discussion of the results will be given, along with a description of the applications and benchmarks used. The evaluation is a part of the Design Cycle of the Design Science Research methodology used in this project, explained in Section 4.2.3.

6.1 Environment

The experiments were carried out on the Marenostrum Supercomputer at the Barcelona Supercomputing Center with the configuration given in Table 6.1.

The software used for our experiments, along with their versions, can be seen in Table 6.2.

Table 6.1: Marenostrum architecture [34].

<table>
<thead>
<tr>
<th>Component</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>3056x IBM dx360 M4</td>
</tr>
<tr>
<td>CPU</td>
<td>2x Intel E5-2670 SandyBridge-EP 2.6GHz</td>
</tr>
<tr>
<td>Network</td>
<td>Infiniband FDR10 (for app. comms) &amp; 10Gbit Ethernet (for GPFS)</td>
</tr>
<tr>
<td>Memory</td>
<td>128GB or 64GB or 32 GB DDR3-1600 (Total: 103.5TB)</td>
</tr>
<tr>
<td>Local storage</td>
<td>500GB 7200rpm SATA II HDD</td>
</tr>
<tr>
<td>File system</td>
<td>1.9PB GPFS disk storage</td>
</tr>
<tr>
<td>OS</td>
<td>Linux-SuSe Distribution 11 SP3</td>
</tr>
</tbody>
</table>
Table 6.2: Table describing the software used to perform the experiments and their versions.

<table>
<thead>
<tr>
<th>Software</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nanos++ runtime library</td>
<td>0.10a</td>
</tr>
<tr>
<td>Mercurium source-to-source compiler</td>
<td>2.0.0</td>
</tr>
<tr>
<td>GNU C/Fortran Compilers</td>
<td>4.9.1</td>
</tr>
<tr>
<td>Intel C/Fortran Compilers</td>
<td>14.0.2</td>
</tr>
<tr>
<td>Intel MPI</td>
<td>5.1.1.1.109</td>
</tr>
<tr>
<td>SCR</td>
<td>Commit with hash dd4a388</td>
</tr>
<tr>
<td>FTI</td>
<td>Commit with hash 3022580</td>
</tr>
</tbody>
</table>

6.2 Methodology

All our experiments consist of three versions:

1. **REF**. This is the reference version. It is checkpoint-free and non-faulty.

2. **FTI/SCR**. This version is an implementation performing application-level checkpoint/restart directly using the APIs provided by FTI or SCR respectively.

3. **CHK**. In this version, the application-level checkpoint/restart is done by the mechanism proposed in this work.

We mainly focus on the overhead introduced by the application-level checkpoint/restart compared to a checkpoint-free and non-faulty execution. The results are obtained by averaging the execution times of 5 different runs for each version.

All the executions of the evaluation are conducted with 64 MPI processes. Depending on whether or not the original version of the benchmark/application has intranode parallelism, the number of threads per process varies: those having intranode parallelism were executed with 8 threads per process, while the others were executed with 1 thread per process.

With respect to the execution time, all the runs took about 10 minutes. The checkpoint frequency is 1 checkpoint per minute so that we perform 10 checkpoints per run. This frequency is expressed in terms of iterations. Hence, we carry out a checkpoint every 10% of the iterations. We are aware that this is a very high frequency, but our goal is to stress the mechanisms to facilitate comparison with the overhead introduced.

With regard to the faults, for the evaluation part, all the faults are deterministically injected when the application has already done half of the work; while for the detailed analysis the fault is injected immediately after the first checkpoint. The faults introduced are C++ exceptions that cause the abortion of the process.
6.2.1 Detailed analysis of an application

The application selected for explanation in detail is one of the applications used in the context of DEEP-ER project: GERShWIN [35]. This application analyzes the human exposure to electromagnetic fields. In this in-depth analysis, we wish to highlight some key aspects regarding application-level checkpoint/restart. Note that the executions represented on the traces were not 10-minutes executions with 10 checkpoints but smaller executions with only 2 checkpoints. It is also necessary to consider that these executions were done with 4 MPI processes and 8 threads per process.

Note that this application uses SCR natively and as backend library when using CHK. It is worth pointing out that FTI behaves like SCR, so the traces would be almost equal using FTI. As regards the traces, they are generated with Extrae, while the visualization is performed using Paraver.

First of all, it is well-known that all types of resilience mechanisms introduce some overhead into the execution. Application-level checkpoint/restart is no different in this regard, and it can be seen in the traces from 6.4. With regard to the annotations in Figure 6.2 and Figure 6.3, the first region marked is the initialization of the application; the next region is the first part of the computation that will be repeated two more times; finally, the other marked region maps the checkpoint time, which is repeated one more time. Moreover, the red bar placed in the middle of the trace indicates the point where the faults are injected. Note that the y-axis represents the number of processing elements used in the execution, while the x-axis is the execution time. Note that all the traces have the same time scale, so the overhead introduced by the checkpoints in Figure 6.2 and Figure 6.3 in comparison with Figure 6.1 may be easily observed. However, the most interesting part for us in these traces is that our approach adds no further overhead than that in the SCR as far as the checkpoint itself is concerned.

As regards the restart phase, some more traces are provided in 6.7, in which we only present traces for the SCR and CHK since the REF version has no restart. Once again the traces are annotated for a better understanding of each region. In this case, the first region performs more actions than just initialization: in Figure 6.5, it is also restoring the data from the last checkpoint, and in Figure 6.6 we find exactly the same behavior. No difference can be observed in terms of duration, between the first region of the SCR version and the CHK version, so that we are not introducing any significant overhead. Following this first region, there is an initial chunk of computation that will be repeated one more time, in both traces. Finally, there is a checkpoint region between the two computation parts.

In summary, in this section, we have seen that we are not introducing any additional overhead either in the checkpoint phase or the restart phase when compared with the direct use of the APIs provided by FTI/SCR.
Fig. 6.1: Trace of the execution of REF version of GERShWIN.

Fig. 6.2: Trace of the execution of SCR version of GERShWIN.

Fig. 6.3: Trace of the execution of CHK version of GERShWIN.

Fig. 6.4: Traces of the execution of REF, SCR and CHK versions of GERShWIN.
Fig. 6.5: Trace of the execution of SCR version from a restart of GERShWIN.

Fig. 6.6: Trace of the execution of CHK version from a restart of GERShWIN.

Fig. 6.7: Traces of the execution of SCR and CHK versions from a restart of GERShWIN.
6.3 Evaluation and discussion

Firstly, we provide a brief explanation of the applications and benchmarks used in this evaluation.

**BT-MZ**: This is a pseudo-application from the NAS Parallel Benchmarks coded using OmpSs+MPI. It uses a block tri-diagonal solver to solve problems derived from CFD [36].

**Duct**: A CFD application performing a Large Eddy Simulation (LES) of turbulent flow in square. This application is pure MPI [37].

**GERShWIN**: This is an application developed by INRIA in the context of DEEP-ER project. It mainly consists of studying the human exposure to electromagnetic fields by means of solving a system of Maxwell equations. This application is implemented using MPI+OmpSs [35].

**N-Body**: The classic N-body simulation of a dynamical system of particles. This benchmark uses MPI+OmpSs.

**SPECFEM3D**: A simulation of a seismic wave propagation using a Galerkin spectral element method. Implemented with MPI+OmpSs.

**Stream**: The classic stream benchmark, part of the HPC Challenge Benchmarks [38], is a synthetic benchmark measuring sustainable bandwidth and the corresponding computation rate for simple vector kernel. It uses MPI+OmpSs.

**TurboRVB**: Developed at SISSA, this application is used to understand high-temperature superconductivity by means of Quantum MonteCarlo simulations. This application, also in the context of the DEEP-ER project is pure MPI [39].

Figures 6.8 and 6.9 illustrate the overhead introduced by the application-level checkpoint/restart using both FTI/SCR and CHK versions when compared against the REF version. First of all, it is necessary to point out that the total overhead depends directly on the checkpoint frequency and the computational complexity of the application. Since we have used the same checkpoint frequency (one checkpoint per minute) for all the applications without considering the relationship between the computational complexity and the size of the application state, there is a noticeable difference between different applications, so that those applications with less computational complexity or a larger data set are more affected by the checkpoints, which results in a bigger overhead. We have divided the performance into two charts: one comparing applications implemented using SCR natively against CHK and using SCR as backend library, shown in Figure 6.9, and the other comparing applications using FTI natively against CHK and using FTI as the underlying library, as shown in Figure 6.8.
On the one hand, one may observe in Figure 6.8, that both versions yield very similar results. In fact, the bigger difference is in SPECFEM3D where CHK is about 1% worse than native FTI.

On the other hand, in Figure 6.9 one observes a trend that is repeated in three over four applications or benchmarks: CHK is slightly outperforming native SCR. After an in-depth investigation, we concluded that the drop in performance is related to how the versions decide whether or not to take a checkpoint: we simply evaluate the condition expression passed in the \texttt{checkpoint} clause whereas SCR uses MPI communications. When using SCR directly, the application uses the call \texttt{SCR\_Need\_checkpoint} to decide whether or not a checkpoint has to be done. Inside this method, rank 0 takes the decision and then broadcasts it to the rest of the processes using MPI. In contrast, CHK decides faster because its decision is based on the condition provided in the \texttt{if} clause. As all the processes receive the same condition, there is no need to broadcast because we already know they are going to take the same decision. Therefore, the difference is the cost of the MPI broadcast.

Apart from performance, we wish to provide some insights in the simplicity added in our approach. To date, there is no standard metric for measuring how easy a mechanism
or a tool is to use. We have therefore selected the number of lines of code needed to implement each of the different versions in order to make this measurement, the results of which are shown in Table 6.3. Here it is possible to see the lines of code required to write application-level checkpoint/restart in FTI, SCR, and CHK. There is a significant difference between SCR and FTI, SCR being the most complex option, at least in terms of lines of code. As a counterpart, we should point out that SCR is more configurable than FTI. For CHK, this difference is even bigger, requiring at most 7 lines in the applications/benchmarks evaluated. In comparison with FTI, CHK requires between 13 and 33 fewer lines to provide the same functionality. This number increases when compared with SCR: CHK needs from 35 to 77 fewer lines. In the light of these results, CHK provides a simpler and more straightforward way of performing the same functionalities as current options.

Finally, in the case of portability, this is enhanced with our solution. If a user has to move an application from a system using SCR to another system using FTI, there is no need to rewrite the code related to checkpoint/restart: CHK will work with the same code for both FTI and SCR backend libraries.

![Table 6.3: Number of lines of code required to perform application-level checkpoint/restart for different tools.](image-url)
Conclusions

In this work, we propose a new directive-based approach for providing application-level checkpoint/restart. Our proposal introduces the application-level checkpoint/restart feature in OmpSs. Using the adequate directives and clauses, the user can specify some data to be checkpointed in a persistent way. In the event of failure, the application is restarted and the stored data in the last checkpoint is loaded, thereby skipping all the work done previously and resuming the execution from the loaded state instead of from scratch.

Our approach, based on user directives, simplifies the process of application-level checkpoint/restart. Our solution (1) minimizes the changes in the source code to apply this technique; (2) completely removes the serialization and deserialization process required by traditional approaches; and (3) eliminates the need for modifying the natural program flow to detect restarts. We leverage state of the art checkpoint/restart libraries (SCR and FTI) to optimize I/O operations and provide redundancy schemes, thereby enhancing resiliency and performance. Moreover, our proposal enhances the portability of applications, because the implementation provides support for several underlying libraries without any changes in application code. It is worth noting that our approach is valid for OmpSs and hybrid MPI+OmpSs applications.

We have evaluated the performance of our proposal employing benchmarks and more complex applications. Our results show a competitive performance compared with the direct use of state-of-the-art solutions such as FTI and SCR, while at the same time minimizing the changes required by the application code and simplifying the effort required to move an application from a backend library to another one. Hence, we have maximized productivity and portability.
8 | Future work

Given that our solution facilitates the way of performing checkpoint/restart by relieving the user from the responsibilities of data serialization/deserialization among others, we believe that it provides us with the means of adding new functionalities that would be too tricky with current approaches. Some of these functionalities could be as follows:

- **Incremental checkpoint**: new checkpoints adding data to data stored on previous checkpoints. For example, saving a matrix by rows: each checkpoint stores a single different row in a collaborative way. Overall, therefore, the entire matrix is checkpointed, and can be loaded as a whole if required.

- **Differential checkpoint**: store only the difference between a previous checkpoint and the new one, thereby minimizing I/O operations and disk space.

However, the commented functionalities would need further support from the backend libraries.

Also, it would be interesting to work in the integration of this mechanism in different programming models such as OpenMP.

Finally, standardization is also an important topic to work in the future. There are several tools to perform application-level checkpoint/restart and each of them has its own API and particularities. A standard way of coding application-level checkpoint/restart would be very helpful for users.
Bibliography


In this appendix, we present one of the applications used in the evaluation. This is the N-Body simulation. Following, you can observe three versions of the code, one using FTI, another using SCR, and, finally, a version using the approach developed in this work. With those codes, it is easy to compare the difference of programmability among the different tools.

Figure A.1 presents the FTI version, Figure A.2 presents the SCR version, and Figure A.3 presents the CHK version. The lines that are required to perform the checkpoint/restart are highlighted.

```c
/* main */

#include "include/nbody.h"
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#undef NDEBUG
#include <assert.h>
#include <string.h>
#include "fti.h"

int main (int argc, char** argv)
{
    int provided;
    //MPI_Init( &argc, &argv );
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    FTI_Init("config.fti", MPI_COMM_WORLD);
    assert(MPI_THREAD_MULTIPLE == provided);

    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //nanos_init_resilience(rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);
```
int num_particles = (argc <= 1 ? default_num_particles :
    roundup(atoi(argv[1]), MIN_PARTICLES)) / rank_size;
assert(num_particles >= 4096);

const int timesteps = argc <= 2 ? default_timesteps : atoi(argv[2]);
const int fault = argc <= 3 ? -1 : atoi(argv[3]);
assert(timesteps > 0);

num_particles = num_particles / BLOCK_SIZE;

nbody_conf_t conf = {default_domain_size_x, default_domain_size_y,
    default_domain_size_z,
    default_mass_maximum, default_time_interval, default_seed,
    default_name,
    timesteps, num_particles};

nbody_t nbody = nbody_setup(&conf);

const double start = wall_time();
solve_nbody_fault(nbody.local, nbody.remote, nbody.forces,
    num_particles, timesteps, conf.time_interval, fault);
const double end = wall_time();

if(rank == 0) {
    printf("Total execution time: %g s.\n", end - start);
}

nbody_save_particles(&nbody, timesteps);

nbody_check(&nbody, timesteps);

nbody_free(&nbody);

FTI_Finalize();
MPI_Finalize();
return 0;

/* solve_nbody */
#pragma offload_attribute (push,target(mic))
#include "include/kernels.h"
#pragma offload_attribute (pop)
```c
#include "include/nbody.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#undef NDEBUG
#include <assert.h>

#include <fti.h>

void solve_nbody_fault(particles_block_t * __restrict__ local,
                        particles_block_t * __restrict__ tmp,
                        force_block_t * __restrict__ forces,
                        const int n_blocks, const int timesteps, const float time_interval, const int fault)
{
    assert(timesteps > 0);

    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    const double start = wall_time();

    int t = 0;
    int restarted_t = 0;
    int initial_id = 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();
        if( restarted_t != t )
            restarted_t = t;
    } else
        restarted_t = -1;

    for (; t < timesteps; t++) {
        // Code for each time step goes here...
    }
```
#pragma omp task inout([n_blocks] local)
{
    if(t%30==0 && restarted_t != 0 && t!=restarted_t) {
        int res = FTI_Checkpoint( initial_id++, 4 );
        if( res != FTI_DONE ) {
            printf( "FTI internal error." );
            MPI_Abort( MPI_COMM_WORLD, -1 );
        }
    }
    if(t==fault) {
        printf("CRASHING!
");
        MPI_Abort( MPI_COMM_WORLD, -1 );
    }
}

particles_block_t * remote = local;
for(int i=0; i < rank_size; i++){
    #pragma omp task in([n_blocks] local,[n_blocks] remote) inout([n_blocks] forces)
    calculate_forces(forces, local, remote, n_blocks);

    #pragma omp task in([n_blocks] remote) out([n_blocks] tmp)
    exchange_particles(remote, tmp, n_blocks, rank, rank_size, i, t);
    remote=tmp;
}
#pragma omp task inout([n_blocks] local) inout([n_blocks] forces)
update_particles(n_blocks, local, forces, time_interval);
#pragma omp taskwait

#pragma omp taskwait
const double end = wall_time();

Code A.1: Code of N-Body simulation performing checkpoint/restart with FTI.
int main (int argc, char** argv)
{
    int provided;
    //MPI_Init( &argc, &argv );
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    SCR_Init();
    assert(MPI_THREAD_MULTIPLE == provided);

    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    int num_particles = (argc <= 1 ? default_num_particles :
                        roundup(atoi(argv[1]),
                                MIN_PARTICLES)) / rank_size;
    assert(num_particles >= 4096);

    const int timesteps = argc <= 2 ? default_timesteps : atoi(argv[2]);
    const int fault = argc <= 3 ? -1 : atoi(argv[3]);
    assert(timesteps > 0);

    num_particles = num_particles / BLOCK_SIZE;

    nbody_conf_t conf = {default_domain_size_x, default_domain_size_y,
                          default_domain_size_z,
                          default_mass_maximum, default_time_interval,
                          default_seed, default_name,
                          timesteps, num_particles};

    nbody_t nbody = nbody_setup(&conf);

    const double start = wall_time();
solve_nbody_fault(nbody.local, nbody.remote, nbody.forces,
    num_particles, timesteps, conf.time_interval, fault);

const double end = wall_time ();

if(rank == 0) {
    printf("Total execution time: %g s.\n", end - start);
}

nbody_save_particles(&nbody, timesteps);

nbody_check(&nbody, timesteps);

nbody_free(&nbody);

SCR_Finalize();
MPI_Finalize();
return 0;
}

/* solve_nbody */

/*
#pragma offload_attribute (push,target(mic))
#include "include/kernels.h"
    #pragma offload_attribute (pop)
*/
#include "include/nbody.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#undef NDEBUG
#include <assert.h>

#include <scr.h>

void solve_nbody_fault(particles_block_t * __restrict__ local,
    particles_block_t * __restrict__ tmp, force_block_t * __restrict__
forces,
    const int n_blocks, const int timesteps, const float
time_interval, const int fault)
{
    assert(timesteps > 0);

    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    const double start = wall_time();

    int t = 0;
    int restarted_t = 0;
    solve_nbody_rt(n_blocks, rank, local, &restarted_t);
    if( restarted_t != t )
        t = restarted_t;
    else restarted_t = -1;
    for (; t < timesteps; t++) {
        #pragma omp task inout([n_blocks] local)
        {
            if(t%30==0 && restarted_t!=0 && t!=restarted_t) {
                solve_nbody_cp(n_blocks, rank, local, t);
            }

            if(t==fault) {
                printf("CRASHING!\n");
                MPI_Abort( MPI_COMM_WORLD, -1 );
            }
        }

        particles_block_t * remote = local;
        for(int i=0; i < rank_size; i++){
            #pragma omp task in([n_blocks] local,[n_blocks] remote) inout
            ([n_blocks] forces)
            calculate_forces(forces, local, remote, n_blocks);

            #pragma omp task in([n_blocks] remote) out([n_blocks] tmp)
            exchange_particles(remote, tmp, n_blocks, rank, rank_size, i, t);
            remote=tmp;
        }
        #pragma omp task inout([n_blocks] local) inout([n_blocks] forces)
update_particles(n_blocks, local, forces, time_interval);

#pragma omp taskwait

#pragma omp taskwait

const double end = wall_time();

// Checkpoint Key: number of blocks, block size, mpi_rank. Data saved:
local, current_timestep

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;

    char name[256];
    char path[SCR_MAX_FILENAME];

    int perform_checkpoint;
    SCR_Need_checkpoint(&perform_checkpoint);
    if (perform_checkpoint == 1) {
        SCR_Start_checkpoint();
        // Get backup file path
        sprintf(name, "solve_nbody-nb"
        if(SCR_Route_file(name, path)==SCR_SUCCESS){
            cint fd = open (path, O_WRONLY | O_CREAT | O_TRUNC, S_IRUSR | S_IRGRP | 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);
}
}

// Checkpoint Key: number of blocks, block size, mpi_rank. Data saved: local, current_timestep
int solve_nbody_rt(const int n_blocks, const int rank,
  particles_block_t* __restrict__ local,
  int *current_timestep) {
  int status = 0;
  int found_cp = 0;
  int temp_tstep;
  int num_read;
  int num_read_particles;
  int size;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  char name[256];
  sprintf(name, "solve_nbody-nb
  /* get backup file path */
  char path[SCR_MAX_FILENAME];
  if (SCR_Route_file(name, path) == SCR_SUCCESS) {
    cint fd = open (path, O_RDWR | O_CREAT, S_IRUSR | S_IRGRP |
      S_IROTH | S_IWUSR | S_IWGRP | S_IWOTH);
    if( fd < 0) perror("ERROR: ");
    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 {

perror("ERROR: ");
status = -1;
}
}

/* 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;

Code A.2: Code of N-Body simulation performing checkpoint/restart with SCR.
#include "include/nbody.h"
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#undef NDEBUG
#include <assert.h>
#include <string.h>

int main (int argc, char** argv)
{
    int provided;
    //MPI_Init( &argc, &argv );
    MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
    assert(MPI_THREAD_MULTIPLE == provided);

    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    //nanos_init_resilience(rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    int num_particles = (argc <= 1 ? default_num_particles :
        roundup(atoi(argv[1]),
            MIN_PARTICLES)) / rank_size;
    assert(num_particles >= 4096);

    const int timesteps = argc <= 2 ? default_timesteps : atoi(argv[2]);
    assert(timesteps > 0);

    num_particles=num_particles/BLOCK_SIZE;

    nbody_conf_t conf = {default_domain_size_x, default_domain_size_y,
        default_domain_size_z,
        default_mass_maximum, default_time_interval,
        default_seed, default_name,
        timesteps, num_particles};

    nbody_t nbody = nbody_setup(&conf);

    const double start = wall_time();
    solve_nbody(nbody.local, nbody.remote, nbody.forces, num_particles,
        timesteps, conf.time_interval);
const double end = wall_time();

if(rank == 0) {
    printf("Total execution time: %g s.\n", end - start);
}

nbody_save_particles(&nbody, timesteps);
nbody_check(&nbody, timesteps);
nbody_free(&nbody);
MPI_Finalize();
return 0;
}

/* nbody.c */

/*
#pragma offload_attribute (push,target(mic))
#include "include/kernels.h"
#pragma offload_attribute (pop)
*/
#include "include/nbody.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mpi.h>
#undef NDEBUG
#include <assert.h>

void solve_nbody(particles_block_t * __restrict__ local,
                 particles_block_t * __restrict__ tmp, force_block_t * __restrict__
                 forces,
                 const int n_blocks, const int timesteps, const float
                 time_interval)
{
    assert(timesteps > 0);

    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);

    const double start = wall_time();
    for (int t = 0; t < timesteps; t++) {

Code A.3: Code of N-Body simulation performing checkpoint/restart with CHK.