Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

Facultat d'Informàtica de Barcelona (FIB)

Final Master Thesis (FMT)

Master in Innovation and Research in Informatics (MIRI)

High Performance Computing (HPC)

Enhancing the Interoperability between Distributed-Memory and Task-Based Programming Models

Kevin Sala Penadés
(kevin.sala@bsc.es)

Advisors:
Vicenç Beltran Querol (vbeltran@bsc.es)
Eduard Ayguadé Parra (eduard@ac.upc.edu)

Computer Architecture Department (DAC)

Barcelona, 27 June 2018

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

Acknowledgments

I would like to thank my advisors, Vicenç Beltran and Eduard Ayguadé, for offering me the opportunity of working with them on several interesting and motivating projects. Also, thanks for the support that I have received and thanks for facilitating my way in the field of research.

Thanks to Josep M. Perez and Xavier Teruel, since this project would not be possible without all their help.

I would like to thank all my family and friends, especially my parents, for supporting me during all this project. Every day, they encourage me to improve both professionally and personally.

I would like to especially thank my friends Toni Navarro and Marc Marí, who have supported and helped me unconditionally every day. They always make me smile and they make me take things more humorously. Thanks for all, fellas.

Last but not least, I would like to thank all my colleagues at the BSC Programming Models group for creating a very nice working environment.

Abstract

Current high-performance computing architecture trends suggest that exascale computing systems will consist of several distributed memory nodes, where each will contain a large number of compute cores. Following these trends, HPC users usually develop hybrid applications combining both distributed-memory and shared-memory programming models to exploit the inter-node and intra-node parallelism, respectively.

However the common parallelization strategies limit the performance and the programmability of hybrid applications, since these programming models were not designed to be combined. When trying to enhance the performance of hybrid application by applying more advanced techniques, such as double-buffering, the complexity of the code increases dramatically, and in addition, they still continue to be suboptimal.

There are several proposals which improve the interoperability between MPI and OpenMP. Some of them propose the taskification of communications along with an interoperability mechanism. However, most of them are specific to a particular task-based programming model.

We propose the enhancement of the interoperability between distributed-memory and task-based programming models in order to (1) improve the programmability of user applications, to (2) improve significantly their performance and (3) being compatible with any task-based programming model. We enhance an existing interoperability mechanism for MPI and we propose a similar mechanism for the GASPI programming model.

Our results, which include the evaluation of some benchmarks in different architectures, reveal that our approaches for both MPI and GASPI programming models improve the performance and programmability of hybrid applications. We also demonstrate that the mechanism can be easily integrated into any distributed-memory programming model, or even into any blocking service not related to programming models.

Table of Contents

List of Figures

List of Tables

List of Code Samples

1 | Introduction

Current near-term and mid-term high-performance computing (HPC) architecture trends suggest that the first generation of exascale computing systems will consist of distributed memory nodes, where each node contains a large number of compute cores that provide a high computing capacity.

Following this idea, the most common practice in the HPC community is to develop hybrid applications combining programming models specialized in exploiting inter-node and intra-node parallelism [1–3]. Usually, distributed-memory programming models, such as MPI [4] or GASPI [5], are used for inter-node parallelism, while shared-memory programming models, such as OpenMP [6] or OmpSs [7], are used for exploiting the parallelism within each node.

Although these programming models were not originally designed to be combined in hybrid applications, they have evolved to provide some interoperability support. However, this minimal support only determines how both models can be safely combined in order to develop hybrid applications, and also, they usually introduce some overhead when enabling the interoperability modes.

In the case of MPI, the standard guarantees that point-to-point communications among two ranks are always ordered as long as these leverage the same tag and communicator. However, when multiple threads communicate simultaneously, operations can be re-ordered causing a deadlock in the execution of the application. To avoid this kind of problems, most hybrid applications use the MPI's thread funneled mode, and then, they serialize communication phases, while computation phases are performed in parallel. Therefore, these applications follow a fork-join parallelization strategy, which opens the parallelism when entering computation phases and closes it before entering a communication phase.

Although this is an easy approach, it presents some programmability and performance problems. Firstly, parallelism may be potentially hindered due to the strict synchronization enforced among computation phases and across nodes. Secondly, usually this synchronization point introduces some overhead by itself. Finally, it is very difficult to overlap computation and communication phases with this kind of approaches.

Occasionally, more advanced strategies are implemented in hybrid applications which require the restructuring of the application code to manually overlap computation and communication phases of the algorithm. These techniques, such as double-buffering, use asynchronous communication primitives to start communication operations, and

while these operations do not complete, the application is able to perform some computations. However, they require complex code modifications, which could be even infeasible depending on the application. Moreover, most times this technique is applied, the performance achieved is still suboptimal.

An easy solution for the previous issues would be to use tasks to implement both computation and communication phases, relying on task dependencies to deal with inter-node and intra-node synchronization. However, this approach cannot be efficiently implemented with current MPI, GASPI and OpenMP specifications. MPI provides the MPI_THREAD_MULTIPLE mode that supports the concurrent invocation of MPI functions from multiple threads, while GASPI allows concurrent calls to its procedures by default. Nevertheless, this is not sufficient to efficiently support task-based programming models such as OpenMP or OmpSs.

The main issue is that tasks are not aware of the synchronous MPI/GASPI operations, which might block not only the task but also the underlying hardware thread that runs it. Even if the MPI/GASPI implementation does not rely on busy-waiting to check for operation completion and the hardware thread becomes idle, the task runtime has no means to discover that the hardware thread is available without an explicit notification from the MPI/GASPI side. Without this notification mechanism, if the number of in-flight operations blocked reaches the number of available hardware threads, the application will hang due to lack of progress [8]. With the current specifications, the user is responsible for avoiding this situation, and therefore, it severely limits their ability to benefit from task-based programming models.

The interoperability between the MPI and OpenMP programming models has been discussed for several years. Most of the proposed approaches require significant changes in both hybrid applications and involved programming models. Moreover, many of these approaches are designed specifically for a particular task-based programming model.

In order to solve these problems, in this project we propose the enhancement of an existing interoperability approach for MPI [9] by (1) improving the programmability in hybrid applications, by (2) increasing their performance and by (3) generalizing the interface between both distributed-memory and task-based programming models. Furthermore, we propose a new interoperability approach for GASPI by using the same interface and trying to follow a similar strategy.

In the following sections we explain the specific objectives and contributions of our project, and finally, we describe the organization of this document.

1.1 Objectives

The main objective of this project is to provide a well-defined mechanism for allowing the interoperation between distributed-memory programming models, such as MPI, and task-based programming models, such as OmpSs and OpenMP, in hybrid application environments. In a task-based ecosystem where tasks perform synchronous

operations, which are provided by a distributed-memory programming model, the interoperability mechanism prevents the underlying hardware threads that execute those tasks from being blocked inside these synchronous operation procedures. Thus these threads can execute other ready tasks while the operations do not complete.

This project aims to properly integrate the interoperability mechanism into both MPI and GASPI distributed-memory programming models. However, it should be applicable to any kind of API featuring blocking or synchronous operations, even APIs not related to distributed-memory programming models (e.g. file accesses). Similarly, the mechanism is designed having in mind the OmpSs programming model, however it should work with any task-based programming model.

More specifically, the objectives of this project are the following:

- Generalize the interoperability mechanism in order to be applicable to any synchronous API and to any task-based programming model.
- Enhance an existing interoperability approach for the MPI programming model.
- Design and develop a new interoperability approach for the GASPI programming model.

By introducing these functionalities, users could improve the performance of their hybrid applications in an easy and portable way. Furthermore, it could facilitate the task of developing a similar interoperability mechanism for other blocking APIs.

1.2 Contributions

With the interoperability mechanism presented in this project we are able to:

1. Taskify both computation and communication phases in hybrid applications, allowing multiple communication tasks to run in parallel, without the risk of generating a deadlock. This can result in the overlap of computations and communications, and also, the overlap of different application's timesteps.
2. Provide a consistent and portable way to enable the aforementioned mechanism in hybrid applications (i.e. MPI and GASPI).
3. Facilitate the integration of the mechanism into other synchronous APIs, thanks to a generic API to block/unblock tasks, and also, thanks to the experience gained by integrating it into both MPI and GASPI programming models.

1.3 Document Structure

This document starts with the current Chapter, which presents the motivations and the general objectives of this thesis. Chapters 3 and 4 introduce the MPI and GASPI distributed-memory programming models, respectively. Although this project does not target any specific task-based programming model, we use the OmpSs-2 programming model in order to exemplify the interoperability mechanism. This programming model is introduced in Chapter 5. Chapter 6 explains the existing MPI+OmpSs interoperability library, which will be the base of our contributions to the MPI interoperability. Finally, Chapter 7 explains the tools used and the methodology followed to successfully complete this project.

The previous chapters explaining the background are followed by the chapters defining our proposals. Firstly, Chapter 8 explains in detail our proposal for the enhancement of the interoperability mechanism between MPI and task-based programming models. Similarly, Chapter 9 describes our proposal for the development of an interoperability mechanism for GASPI and task-based programming models. Then, Chapter 10 proposes a non-blocking interoperability mechanism, which does not block communication tasks, and aims to improve the previous interoperability mechanism.

Chapter 11 explains the evaluation that we have performed using two applications, which are also analyzed in detail. Finally, Chapter 12 presents the conclusions of this thesis, followed by Chapter 13, which proposes the next steps for this project.

2 | Related Work

One of the objectives of this project is to allow the overlap between computation and communication phases of hybrid MPI/GASPI applications in an efficient, clear and comfortable way. The topic of overlapping computation and communication has been studied for a long time and in several contexts.

[10, 11] propose the development of a threading library for the Cell B.E. processor, which transparently overlaps the computation and communication phases of different threads running on the same SPU (Synergistic Processor Unit in the Cell B.E. architecture). When a running thread is about to block on a DMA operation, the thread execution is suspended until the DMA operation completes. Meanwhile, the execution of another thread is resumed on the SPU, thus the communication phase of the suspended thread overlaps with the computation phase of the currently executing thread. Other techniques such as double and multi-buffering are studied. These techniques also allow overlapping but are limited to applications with a regular and static communication patterns, while the approach based on threads supports irregular applications with a dynamic communication pattern.

The study of hybrid approaches [1–3] combining communication libraries and shared memory programming models has been considered over the last years both in research and in performance analysis publications.

[8] proposes the use of the comm_thread approach from the hybrid MPI+SMPSs programming model, which allows to exploit distant parallelism separated by taskified MPI calls. These tasks are identified as *communication tasks* and are executed by an additional thread called *communication thread*. With this information, the scheduler of the runtime system can reorder the execution of computation and communication tasks so that communication can happen as soon as possible, increasing the parallelism within and across MPI processes. This proposal requires changes to the programming model in order to identify the tasks that perform blocking MPI operations, and also, these are executed sequentially by the communication thread. In contrast, we propose an approach which does not require any change in the task-based programming model, there is no need to identify communication tasks and it supports multiple communications in parallel and out of order.

In the Habanero-C MPI (HCMPI) proposal [12], MPI calls are tightly integrated with the task dependency system. HCMPI treats all MPI calls as tasks, which brings well-known issues inherent to excessively fine-grained tasking, such as increased scheduling

overhead and load imbalance. In contrast, the approach we propose is orthogonal to the dependency system and does not require to taskify every MPI operation. In addition, it is especially well suited to parallelize legacy and library code, as it does not require to taskify every MPI operation, resulting in a more natural and flexible approach.

An interoperability library for the MPI and OmpSs programming models is proposed in [9]. This library reproduces a behavior similar to the one proposed in [10, 11] but for MPI. It intercepts blocking MPI calls and blocks the calling task until the operation is complete, so the underlying hardware thread is able to execute other ready tasks. Our project takes this library as the base for enhancing the interoperability with MPI by introducing some extensions. The original version of the library is explained in detail in Chapter 6.

To the best of our knowledge, this is the first approach which aims to develop an interoperability mechanism for the GASPI and shared-memory programming models. The usual parallelization strategy in hybrid applications (e.g. GASPI+OpenMP) is to start RDMA operations concurrently (e.g. within a parallel for), and then, to wait for the completion of all these operations from the master thread. The next computation phase is started once all operations complete. However, this approach is extremely suboptimal since most communication is serialized, the synchronization point after the parallel region introduces overhead and does not allow the overlap of computation and communication. Other techniques such as double buffering are also recommended, but as mentioned above, these techniques are limited to applications with regular and static patterns and they usually require several changes in the application.

In addition, there are some articles about the interoperability between GASPI and MPI programming models, but they are out of the scope of our project.

3 | Message-Passing Interface (MPI)

MPI [4] stands for Message Passing Interface and is a well-known message-passing library interface specification. This specification has been defined through an open process by a community of parallel computing vendors, computer scientists and application developers. MPI addresses mainly the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to this classical message-passing model are also provided in collective operations, remote-memory access (RMA) operations, dynamic process creation and parallel I/O.

This specification is for a library interface. MPI is not a language and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings. The MPI standard includes support for C and Fortran bindings.

The main advantages of establishing a message-passing standard are portability and ease of use. In a distributed-memory communication environment in which the higher level routines and/or abstractions are built upon the lower level message-passing routines, the benefits of standardization are particularly apparent. Moreover, a message-passing standard helps vendors since it defines a clear base set of routines which they can implement efficiently, for instance, targeting their system platforms.

The main objectives of the Message-Passing Interface are:

- Design an application programming interface.
- Allow efficient communication: avoiding memory-to-memory copying, allowing the overlap of computation and communication and offload as much as possible to communication co-processors, when available.
- Support for heterogeneous environments.
- Allow convenient C and Fortran bindings for the interface.
- Assume a reliable communication interface. The user does not need to take care of communication failures, since they are managed by the underlying communication infrastructure.
- Define an interface that can be implemented on many vendor's platforms.
- The semantics of the interface should be language independent.

- The interface should be designed to allow several thread safety modes.

3.1 History

A preliminary draft proposal, known as MPI-1.0, was presented in November 1992 and a revised version was complete in February 1993. It originated aiming the standardization of all message-passing systems in order to facilitate the task of both system vendors and users. MPI was designed to use the most attractive features from several existing message-passing libraries. Some of the most important influencers were the work performed at the IBM T. J. Watson Research Center, the Intel's NX/2 and Chameleon [4].

The MPI standardization effort involved about 60 people from 40 organizations, mainly from the US and Europe. Most of the major vendors of parallel computers were involved along with researchers from universities, government laboratories and industry. The standardization process started in the Workshop on Standards for Message-Passing in a Distributed Memory Environment. The basic features of a message-passing interface were discussed at this workshop. The standardization process continued by a working group after that workshop [4].

Since 1992 MPI has been evolving and it has been enhanced with several new functionalities. The first standard, MPI-1.0, was followed by MPI-1.1, MPI-1.2, MPI-1.3, MPI-2.0, MPI-2.1, MPI-2.2, MPI-3.0, and finally, MPI-3.1, which is the current standard and the one we describe in this project. At this moment, the MPI Forum members are working on the new MPI-4.0.

3.2 Basic Concepts

In this section we briefly introduce the basic concepts of the MPI standard. In the next sections we will explain more in detail the main functionalities of the standard.

MPI programs consist of autonomous processes executing their own code in a multiple instruction multiple data (MIMD) style. Typically, each process executes in its own address space and communicates by calling MPI operations or primitives. Each process is identified by a unique rank throughout the execution.

MPI has two main types of communication. On the one hand, the point-to-point communication is the one that involves two MPI processes: the process that sends the data and the process which receives it. Both processes are required to call the corresponding function (i.e. send or receive) in order to successfully complete this communication. That is why it is also referred to as two-sided communication. On the other hand, the collective communication is the one that involves a whole subset of MPI processes. All processes in the group are required to call the collective procedure in order to successfully complete the operation. In addition, the MPI standard provides

one-sided communication operations, which are based on Random Memory Access (RMA).

Communicators (MPI_Comm) bring together the concept of process groups and context. A communicator contains a group of valid participant processes. MPI communication operations require a specific communicator to determine the scope and the *communication universe* in which a point-to-point or collective operates. In point-to-point operations, both source and destination ranks of a message are identified by the process rank within the group. In collective communication, the communicator specifies the set of processes that participate in the collective operation. In this project we will only use the default communicator (MPI_COMM_WORLD), which encompasses all processes.

In addition to the communication types mentioned above, both point-to-point and collective operations can also be blocking or non-blocking. A blocking operation blocks the calling thread inside the MPI library until the corresponding operation completes. In contrast, non-blocking operations return *immediately*, whilst the operation may not be completed yet. Non-blocking procedures return a handle to the communication request (MPI_Request), which can be used later to check the completeness of the operation, either with a blocking or non-blocking procedure. Thus, non-blocking operations can be used to perform other useful computation while the MPI operation is not finished, allowing the overlap of computation and communication.

Lastly, it is important to highlight that MPI provides the user with a reliable interface. A sent message is always received correctly, and the user does not need to check for transmission errors, connection time-outs, or other network errors. In fact, users are only responsible for using the interface correctly. All MPI functions return an error code. In case the function completes correctly, it returns MPI_SUCCESS. Otherwise, it returns the specific code of the error.

3.3 Initialization

The first step in an MPI application is to initialize the MPI library. MPI offers a simple procedure named MPI_Init which initializes the MPI library. As can be seen below, the function just takes the arguments of the main function as parameters.

```
int MPI_Init(int *argc, char ***argv);
```

As mentioned previously, MPI allows different thread safety levels. For that reason, MPI can be initialized with several levels of thread support (or threading levels), which are described in the following list:

- **MPI_THREAD_SINGLE** indicates that only one thread is executed. It is equivalent to calling to MPI_Init.
- **MPI_THREAD_FUNNELED** indicates that the process may be multi-threaded, but the application ensures that only the main thread makes MPI calls.

- **MPI_THREAD_SERIALIZED** indicates that multiple threads in the process can make MPI calls, but only one at a time.
- **MPI_THREAD_MULTIPLE** indicates that multiple threads in the process can make MPI calls, with no restrictions.

Threading levels are only allowed in the MPI_Init_thread initialization function, which is shown below. It takes the main's arguments along with the threading level required by the user. The communication library will enable the threading level returned in the provided parameter. If possible, the library will provide the required mode. Failing this, it will return the least supported level such that provided > required (thus providing a higher mode than the required by the user). If the user requirement cannot be satisfied, then it will provide the highest supported mode.

```
int MPI_Init_thread(int *argc, char ***argv, int desired, int *provided);
```

In addition, MPI provides a couple of functions to retrieve the process rank and the total number of processes within a given communicator. They are shown below.

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Comm_size(MPI_Comm comm, int *size);
```

3.4 Data Messages

In MPI communication operations, processes are required to indicate the involved source/destination buffer. Memory buffers are defined by its starting memory address, the number of elements to be transmitted from the buffer and the elements' datatype.

MPI offers several predefined types of data, e.g. MPI_INT for integers, MPI_DOUBLE for double-precision floating-points and MPI_BYTE for raw bytes. In addition, MPI allows the creation of new data types. Users can specify which data fields comprise their custom data types, the strides between elements in an array, etc.

3.5 Point-to-Point Communication

As mentioned previously, one of the main MPI functionalities is the point-to-point communication. Point-to-point operations involve both the source and the destination processes. The former must call the send procedure (e.g. MPI_Send), while the latter has to call the receive procedure (e.g. MPI_Recv). Otherwise, the communication may not complete properly. In addition, point-to-point messages are identified by the sender's rank, an integer tag and the involved communicator. Thus the receiver process can distinguish multiple messages from the same sender.

In the following subsections we show the main procedures for sending an receiving data in both blocking and non-blocking modes.

3.5.1 Blocking Operations

The MPI_Send and MPI_Recv functions are the blocking point-to-point operations par excellence. Both are the most common MPI primitives in user applications.

On the one hand, the MPI_Send function, shown below, sends a data message labeled with tag to a remote process with rank dest. The message's data consist of a memory buffer with count elements of the type datatype, starting at address buf. This function returns once it is safe to modify the source memory buffer. Note that it does not guarantee that the remote process has already received the data or that the transmission has begun [4].

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
     int tag, MPI_Comm comm);
```

On the other hand, the destination process must call to MPI_Recv to receive the message. This function is shown below. It stores the message's data into the receive buffer which consists of count elements of the type datatype, starting at address buf.

The receiving procedure must select a message matching with the specified tag, the source rank and the comm communicator. However, users can specify the MPI_ANY_TAG and/or MPI_ANY_SOURCE constants to match a message with any tag and/or from any source process, respectively.

MPI_Recv returns once the data has been received in the receive buffer and can be actually consumed by the application. Finally, the status can be used to get information about the message (e.g. the source rank and the tag).

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
     int tag, MPI_Comm comm, MPI_Status *status);
```

3.5.2 Non-Blocking Operations

The non-blocking counterparts are similar to the previous functions, as can be seen below. Both MPI_Isend and MPI_Irecv register a new communication request and they return *immediately*, thus the operation may not be completed. A handle to the request is returned through the request parameter.

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest,
     int tag, MPI_Comm comm, MPI_Request *request);

int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
     int tag, MPI_Comm comm, MPI_Request *request);
```

Later, users can check the completeness of pending requests using non-blocking or blocking checking procedures. These are the MPI_Test and MPI_Wait procedures, respectively, which are shown below. MPI_Test just tests the completeness of the request

and returns whether it has completed or not through the flag parameter. In contrast, MPI_Wait blocks until the request is completed.

```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

In addition, MPI offers other procedures to test or wait for multiple requests, e.g. MPI_Testall and MPI_Waitall. More information about them can be found in [4].

Normally, users issue multiple non-blocking operations in order to perform useful computation while the MPI requests do not complete, thus overlapping computation and communication.

3.6 Collective Communication

MPI offers several collective operations which involve all processes from a specific communicator. All processes from the communicator must call the same collective procedure to successfully complete the operation.

There are many collective operations in the MPI standard. For instance, the MPI_Barrier blocks until all processes have called the barrier procedure and the MPI_Bcast is the procedure which a specific process known as root sends the same data to all participating processes. Both procedures are shown below. More information about collective operations can be found in [4].

```
int MPI_Barrier(MPI_Comm comm);
int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root,
     MPI_Comm comm);
```

In addition, MPI offers the non-blocking counterparts, which work in the same way as the non-blocking point-to-point communications. They also return a handle to the request, so they can be checked with MPI_Test and MPI_Wait procedures.

It is worth noting that MPI does not support tags for collective operations. Since the communicator is the only operation's identifier, users can issue multiple collectives at the same time only when each collective uses a different communicator.

4 | Global Address Space Programming Interface (GASPI)

GASPI [5] stands for Global Address Space Programming Interface and is a Partitioned Global Address Space (PGAS) API. Generally, PGAS approaches offer the developer an abstract shared address space which simplifies the programming task, and at the same time, it facilitates the data-locality, thread-based programming and asynchronous communication. Besides the general benefits of being a PGAS approach, GASPI is designed specifically to provide an extreme scalability, high flexibility and failure tolerance in distributed-memory environments.

The main objective of GASPI is to introduce a paradigm shift from bulk-synchronous two-sided communication patterns towards an asynchronous communication and execution model. In order to achieve this objective, GASPI provides remote completion and one-sided RDMA driven communication in a Partitioned Global Address Space.

It is worth noting that GASPI is neither a new language (e.g. Chapel from Cray), nor an extension to a language (e.g. Unified Parallel C). In fact, GASPI follows the spirit of MPI by complementing existing languages (i.e. C, C++ and Fortran) with a PGAS API which allows the user application to take benefit from the PGAS approach.

Nevertheless, GASPI is not just a memory model, but also provides a way to dynamically allocate and configure RDMA PGAS memory segments. It allows users to map the memory heterogeneity of a modern supercomputer node to these PGAS segments. For instance, GASPI allows users to map the main memory of a GPGPU or an Intel Xeon Phi to a specific segment or to map Non-Volatile RAM to another segment. Then, all these segments can be directly read and written from any node, either from the same node or from a remote one [5].

Figure 4.1 shows the Partitioned Global Address Space supported by GASPI in a heterogeneous architecture. In this case, there are two usual compute nodes (NUMA systems) and two co-processors (e.g. Intel Xeon Phi) connected through an interconnection network supporting RDMA operations.

In addition to the main features explained above, GASPI defines more interesting mechanisms. As mentioned above, this standard is failure tolerant since it allows the use of timeout mechanisms for all non-local operations, failure detection and the possibility of adapting to shrinking or growing node sets. It also provides asynchronous collectives and atomic operations for groups of processes. In addition, GASPI is very

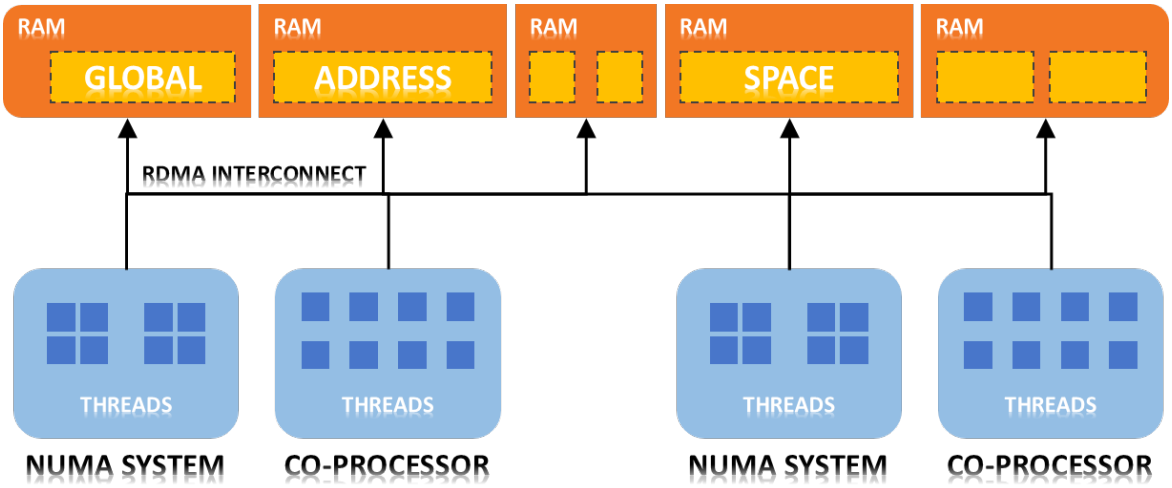


Fig. 4.1: GASPI overview in a heterogeneous architecture [5].

flexible in the number of message queues (to read/write data from/to remote segments) and the size of these queues.

Last but not least, GASPI provides a clean, consistent and convenient standard API, which is relatively easy to use from user applications targeting distributed-memory systems. Furthermore, it gives the maximum freedom to GASPI implementors by leaving many details to the implementation.

4.1 History

The GASPI specification originates from the PGAS API of the Fraunhofer ITWM (Fraunhofer Virtual Machine or FVM), which has been developed since 2005. Starting from 2007 this PGAS API has evolved into a robust commercial product (named GPI) which is used in the industry projects of the Fraunhofer ITWM. GPI is a highly efficient and scalable programming model for Partitioned Global Address Spaces. In 2011, the partners of Fraunhofer ITWM launched the GASPI project to define a novel specification for a PGAS API (GASPI, based on GPI) and to make this novel GASPI specification a reliable, scalable and universal tool for the HPC community [5].

4.2 Remote Direct Memory Access

The only requirement of GASPI is the availability of an interconnection network supporting Remote Direct Memory Access (RDMA). If that is the case, the underlying network infrastructure manages the actual data transfers in one-sided operations, allowing these operations to be fully asynchronous and non-blocking. Since the CPU is excluded from the data transfer, the user application can continue its execution,

thus allowing the overlap between computation (CPU) and communication (network infrastructure) [5].

4.3 Basic Concepts

Similarly to the previous chapter, in this section we just describe briefly the basic concepts of GASPI. In the subsequent sections, we will explain more in detail the functionalities showing the most important API procedures.

The most basic concepts of GASPI are the processes and the groups of processes. GASPI preserves the concept of ranks present in the MPI standard, so each process is identified by a unique rank throughout the execution. At the same time, processes can be grouped allowing collective operations across member processes [5].

As stated previously, GASPI is a communication API that implements a Partitioned Global Address Space or PGAS model. Each GASPI process can host several parts of the global address space, also known as segments. Segments are useful for mapping a large variety of hardware layer (e.g. SSD and GPU memory) to the software layer [5]. Moreover, segments are accessible from every thread of every process, facilitating the use of other parallel programming models within every process (e.g. OpenMP).

Once explained the basics of segments, it is time to describe their main utility: the one-sided asynchronous communication between processes by reading and writing to segments. Local segments can be accessed with the usual load/store operations and remote segments can be accessed using the GASPI read and write operations.

Write operations in GASPI apply the concept of remote completion in the form of notifications. For instance, the user is able to write to a remote segment by posting the write request along with a notification for the remote process. Once the remote process receives the notification, it knows that the write operation has completed.

In addition, GASPI defines a timeout mechanism for its potentially blocking procedures, which is very useful for failure tolerant applications. Timeouts are specified in milliseconds, but there are also a couple of predefined values. These are GASPI_BLOCK, which indicates the procedure to block until it completes, and GASPI_TEST, which indicates the procedure to block the shortest time possible (e.g. processing a portion of the work).

Lastly, all GASPI procedures have a return value which specifies the state of the operation after returning from the procedure. Thus, user programs should always check the return value of procedure calls to know whether they successfully completed (i.e. GASPI_SUCCESS), they detected an error (i.e. GASPI_ERROR) or they ran into a timeout (i.e. GASPI_TIMEOUT). If a procedure call returns a timeout, the procedure should be invoked subsequently in order to complete the operation.

4.4 Initialization

The first phase in a GASPI application is the setup & initialization of the GASPI library. GASPI gives the option to the user to change some parameters of the library configuration before actually initialize it. Some of these parameters are the maximum number of allowed queues, the maximum number of allowed memory segments or the number of precreated queues.

In order to change these parameters, GASPI provides a function to retrieve a configuration structure with the default values. Then, users can change some of these values directly in that structure and set it back to the library with a setter function. These two functions are:

```
gaspi_return_t gaspi_config_get(gaspi_config_t *config);
gaspi_return_t gaspi_config_set(gaspi_config_t config);
```

Once the user has changed (or maintained) the default configuration, the library is ready to be initialized. Similarly to MPI, GASPI provides the following functions to initialize and finalize the GASPI process, both with a timeout parameter.

```
gaspi_return_t gaspi_proc_init(gaspi_timeout_t timeout);
gaspi_return_t gaspi_proc_term(gaspi_timeout_t timeout);
```

In the same way as MPI, GASPI has the following two procedures to get the rank which identifies the current process and the total number of processes being used, respectively.

```
gaspi_return_t gaspi_proc_rank(gaspi_rank_t *rank);
gaspi_return_t gaspi_proc_num(gaspi_rank_t *proc_num);
```

Once the library is initialized, users can create groups of processes, register memory segments, communicate with other processes, etc. It is worth noting that no GASPI functions should be called before the initialization or after the finalization (unless configuration functions).

4.5 Groups

Users have the possibility to create custom groups, which are subsets of all processes. Groups have to be created by invoking the gaspi_group_create procedure, which creates an empty group. Processes can be added into the group by calling to the gaspi_group_add function, which adds the corresponding process rank into the group.

After adding all participating processes, groups must be committed by invoking gaspi_group_commit, which is a collective operation that actually establishes the corresponding group. In fact, all processes in the group have to perform the same steps to establish

the group. Moreover, the group must be identical in all participating processes. Once the group is committed, group members are allowed to perform collective operations within the group.

Finally, groups can be deleted by calling to gaspi_group_delete.

```
gaspi_return_t gaspi_group_create(gaspi_group_t *group);
gaspi_return_t gaspi_group_add(gaspi_group_t group, gaspi_rank_t rank);
gaspi_return_t gaspi_group_commit(gaspi_group_t group, gaspi_timeout_t to);
gaspi_return_t gaspi_group_delete(gaspi_group_t group);
```

In addition, GASPI predefines a global group containing all processes, named GASPI_GROUP_ALL, which is similar to the concept of MPI_COMM_WORLD in MPI. This predefined group can be used after being committed by also calling to gaspi_group_commit and passing GASPI_GROUP_ALL as the group parameter.

4.6 Memory Segments

As stated previously, GASPI provides the functionality of creating and registering memory segments. The API contains a function named gaspi_segment_create, which allocates and registers a memory segment with a specific segment identifier, the segment size (in bytes), the group of processes related to the segment and an allocation policy (not covered in this project). This function allocates a local memory buffer (e.g. with malloc) with the given size and then registers it as a GASPI segment. The actual pointer to the local memory buffer can be retrieved with the function gaspi_segment_ptr.

```
gaspi_return_t gaspi_segment_create(
      gaspi_segment_id_t segment_id, gaspi_size_t size,
      gaspi_group_t group, gaspi_timeout_t timeout,
      gaspi_alloc_t alloc_policy);

gaspi_return_t gaspi_segment_ptr(
      gaspi_segment_id_t segment_id, gaspi_pointer_t *ptr);
```

The API also defines a function named gaspi_segment_use, which registers an existing local memory buffer as a memory segment with a specific segment identifier, the pointer to the buffer, the segment size and the group of processes. Both gaspi_segment_create and gaspi_segment_use perform collective operations, thus the assigned group must be already established. A segment is ready to be written from a remote process once the segment is registered.

```
gaspi_return_t gaspi_segment_use(
      gaspi_segment_id_t segment_id, gaspi_pointer_t ptr,
      gaspi_size_t size, gaspi_group_t group, gaspi_timeout_t timeout,
      gaspi_memory_description_t memory_description);
```

Finally, the gaspi_segment_delete function deletes a segment passing its segment identifier, regardless of whether it was registered with gaspi_segment_create or gaspi_segment_use.

```
gaspi_return_t gaspi_segment_delete(gaspi_segment_id_t segment_id);
```

Users are responsible for not calling to the registration/deletion procedures more than once with the same identifier.

4.7 One-Sided Communication

In this section we describe the GASPI procedures to perform one-sided read/write operations, to notify remote processes and to wait for these notifications.

4.7.1 Write & Read Operations

GASPI provides several procedures to perform one-sided read/write operations. Write operations transfer data from a local segment to a remote segment, while read operations transfer data from a remote segment to a local segment.

On the one hand, write operations support four variants that allow different communication patterns. They are 1) gaspi_write, 2) gaspi_write_notify, 3) gaspi_write_list and 4) gaspi_write_list_notify. The first two variants post a write request to a specific queue in order to write a chunk of data from a local segment to a remote segment. In addition, the second variant posts a notification along with the write request, therefore the remote process is notified once the write operation is completed.

The third and fourth variants are an extension of the first two. They submit multiple write requests in order to write multiple chunks of data to the remote segment. In the fourth variant, the remote process is notified once all write operations are completed.

Both gaspi_write and gaspi_write_notify are shown below; the rest has been omitted.

```
gaspi_return_t gaspi_write(gaspi_segment_id_t segment_local,
      gaspi_offset_t offset_local, gaspi_rank_t rank,
      gaspi_segment_id_t segment_remote,
      gaspi_offset_t offset_remote, gaspi_size_t size,
      gaspi_queue_id_t queue, gaspi_timeout_t timeout);

gaspi_return_t gaspi_write_notify(gaspi_segment_id_t segment_local,
      gaspi_offset_t offset_local, gaspi_rank_t rank,
      gaspi_segment_id_t segment_remote,
      gaspi_offset_t offset_remote, gaspi_size_t size,
      gaspi_notification_id_t notification_id,
      gaspi_notification_t notification_value,
      gaspi_queue_id_t queue, gaspi_timeout_t timeout);
```

On the other hand, read operations do not support notification variants. This is due to the fact that a notification can only be transferred after ensuring that the communication request has been processed. This would imply that a subsequent wait call has to be invoked directly after invoking the read operation. However, this can be managed more effectively by the application [5].

The only two variants that read operations support are gaspi_read and gaspi_read_list. The former posts a read request to a specific queue in order to read a chunk of data from a remote segment to a local segment. Similarly, gaspi_read_list submits multiple read requests in order to read multiple chunks of data from a remote segment.

Once again, only gaspi_read is shown below, while gaspi_read_list has been omitted.

```
gaspi_return_t gaspi_read(gaspi_segment_id_t segment_local,
     gaspi_offset_t offset_local, gaspi_rank_t rank,
     gaspi_segment_id_t segment_remote,
     gaspi_offset_t offset_remote, gaspi_size_t size,
     gaspi_queue_id_t queue, gaspi_timeout_t timeout);
```

A chunk of data in GASPI is defined by the segment that contains it, the offset within the segment and the chunk size (both in bytes). As can be seen above, all variants require the local segment and the local offset. Similarly, they also require the remote segment and the remote offset. Furthermore, they need the destination rank, the target queue and the timeout.

Finally, notify variants require the notification id and the notification value. We explain more in detail both parameters in the next subsection.

4.7.2 Notify Operation

Besides the write & notify procedures, GASPI has an explicit function to notify a remote process. The procedure shown below posts a notification to a remote process, which will set the notification value in the notification id of the given remote segment. A similar behavior is also observed in write & notify procedures. It is worth noting that multiple calls to gaspi_write or gaspi_write_list followed by a call to gaspi_notify implies that the remote process will be notified once all writes are completed.

```
gaspi_return_t gaspi_notify(gaspi_segment_id_t segment_remote,
     gaspi_rank_t rank, gaspi_notification_id_t notification_id,
     gaspi_notification_t notification_value,
     gaspi_queue_id_t queue, gaspi_timeout_t timeout);
```

4.7.3 Waiting Notifications

In order to wait for notifications from remote processes, GASPI provides the function shown below. The gaspi_notify_waitsome function waits until a notification id within

the range [notification_begin, notification_begin+notification_num) is notified. Once a notification id from the range has been notified, its id is returned through the first_id parameter. Since it is a blocking procedure, it requires a timeout. It is worth noting that multiple threads can wait for the same range of notification, so all of them may return from the procedure when a notification is received.

```
gaspi_return_t gaspi_notify_waitsome(gaspi_segment_id_t segment_local,
     gaspi_notification_id_t notification_begin,
     gaspi_number_t notification_num,
     gaspi_notification_id_t *first_id,
     gaspi_timeout_t timeout);
```

To solve the risk of multiple threads waiting the same notification range, GASPI provides the function shown below to reset a notification. The gaspi_notify_reset function atomically resets the value of the notification_id to zero and returns the old value before the reset. Therefore, only one thread will get the actual notification value, while the rest will get a zero. This is very useful in applications where multiple threads wait for the same notifications and perform some actions when one of them is notified. It is important to mention that a notification id has to be reset before receiving another notification in that position.

```
gaspi_return_t gaspi_notify_reset(gaspi_segment_id_t segment_local,
     gaspi_notification_id_t notification_id,
     gaspi_notification_t *old_notification_value);
```

4.8 Queues

As explained in the previous section, one-sided read/write requests are posted to a specific queue. GASPI offers to users the possibility to create multiple queues, which are totally independent of each other. The specification recommends the use of several queues to multiplex the different types of requests, so users can maximize the scalability of their programs.

To do so, the API contains two functions to create and delete GASPI queues. On the one hand, gaspi_queue_create creates a new queue and assigns an identifier to it. On the other hand, gaspi_queue_delete deletes a queue given its identifier. The maximum number of allowed queues is relatively limited in order to keep resources requirements low. Users are recommended to keep the lifetime of queues as long as possible, since the creation of a queue is an expensive operation.

```
gaspi_return_t gaspi_queue_create(
     gaspi_queue_id_t *queue_id, gaspi_timeout_t timeout);

gaspi_return_t gaspi_queue_delete(gaspi_queue_id_t queue_id);
```

One of the most important GASPI procedures is the gaspi_wait function. It waits until all requests posted to a specific queue have been processed by the network infrastructure. The successful completion of this procedures does not mean that write requests have already completed on the remote side, but only indicates their local completion.

It is worth noting that this procedure is thread exclusive and it needs privileged access to the queue. Thus other threads calling to this procedure with the same queue or posting read/write operations to that queue may block. Notice that threads calling to gaspi_wait passing different queues can run totally in parallel.

In addition, a timeout can be used to wait for the local completion of the requests.

```
gaspi_return_t gaspi_wait(gaspi_queue_id_t queue, gaspi_timeout_t timeout);
```

4.9 Other Functionalities

In addition to the previous features, GASPI provides several functionalities that we will not cover in detail. The API includes some collective operations involving groups of processes, such as gaspi_barrier which block the caller until all group members have invoked the procedure and gaspi_allreduce which performs a reduction across all participating processes, and in which everyone gets the final result.

Moreover, GASPI provides two-sided communication procedures to send and receive data, which are gaspi_passive_send and gaspi_passive_receive, respectively. Furthermore, it defines some procedures to deal with global atomic variables.

5 | OmpSs-2 Task-based Programming Model

OmpSs-2 [13] is the second generation of the OmpSs programming model [7], both developed at the Barcelona Supercomputing Center (BSC). The name originates from merging the name of the OpenMP and StarSs programming models. The design principles of both programming models constitute the fundamental ideas used to conceive the OmpSs philosophy [13].

On the one hand, OmpSs takes from OpenMP the main idea of providing a way to produce a parallel version of a sequential program by introducing compiler annotations (or directives) in the source code. These annotations do not imply an explicit effect on the semantics of the program, but they allow the compiler to produce a parallel version of it. This functionality allows users to parallelize their applications incrementally by adding new OmpSs directives to different parts of the application [13].

On the other hand, Star SuperScalar (StarSs) is a family of programming models that also offers implicit parallelism through a set of compiler annotations. StarSs differs from OpenMP in some important areas. Firstly, StarSs uses a thread-pool execution model, whilst OpenMP implements a fork-join parallelism model. Secondly, StarSs provides an environment where parallelism is implicitly created from the beginning of the execution, thus users can omit the declaration of parallel regions. Parallelism is defined by tasks, which are pieces of code that can be executed asynchronously. Thirdly, StarSs provides a dependency mechanism to define data dependencies between tasks, allowing tasks to be run in the correct order. Finally, StarSs targets heterogeneous architectures through leveraging native kernel implementations.

5.1 Basic Concepts

OmpSs-2 is a programming model composed of several directives and library routines which can be used along with a high level programming language (C, C++ and Fortran) in order to develop concurrent applications. OmpSs-2 extends the tasking model of OmpSs and OpenMP to support both task nesting and fine-grained dependencies across different nesting levels.

The main goal of OmpSs-2 is to provide a productive environment to develop parallel applications for modern HPC systems. In order to achieve it, the programming model should provide a competitive performance, but also and more important, it must be easy to use for all users. Furthermore, OmpSs-2 aspires to extend the OpenMP standard with new directives, clauses and API services.

As mentioned previously, OmpSs-2 is a task-based programming model, in which the elementary units of work are the tasks. Tasks represent pieces of code which can be executed asynchronously. In addition, OmpSs-2 defines a powerful dependency model for tasks. Users annotate tasks using dependencies in order to determine the data flow of their applications. This information is then analyzed at runtime to determine if the parallel execution of any two tasks may cause data races.

There are several functionalities supported by the OmpSs-2 programming model. The most distinguished features are:

- The **lifetime of task data environment**. A task is *completed* once the last statement of its code is executed, and it becomes *deeply completed* once all its children tasks become deeply completed. The data environment of a task, which refers to the variables captured at task creation, is preserved until the task is deeply completed. Note that the stack is lost when the task completes.
- The **nested dependency domain connection**. Incoming dependencies of a task are propagated to its children. When a task finishes, its outgoing dependencies are replaced by those generated by its children.
- The **early release of dependencies**. When a task completes, it releases all dependencies that are not included on any uncompleted children task. In addition, a `wait` clause specified in a task declaration makes all its dependencies to be released once the task is deeply completed.
- The **weak dependencies**. The `weakin`/`weakout` clauses specifies potential dependencies only required by children tasks. Thus, these annotations do not delay the execution of the task.
- The **native offload API**. An asynchronous API to execute OmpSs-2 kernels on a specified set of CPUs from any kind of application.

5.2 Influencing OpenMP

Over the years, OmpSs and StarSs have introduced several ideas into the OpenMP programming model. Figure 5.1 shows the main contributions in each OpenMP standard. Asynchronous tasks, task dependencies, taskloops and task priorities are some of these contributions. Upcoming OpenMP versions will include multi-dependencies, task reductions and commutative dependencies, among others [13].

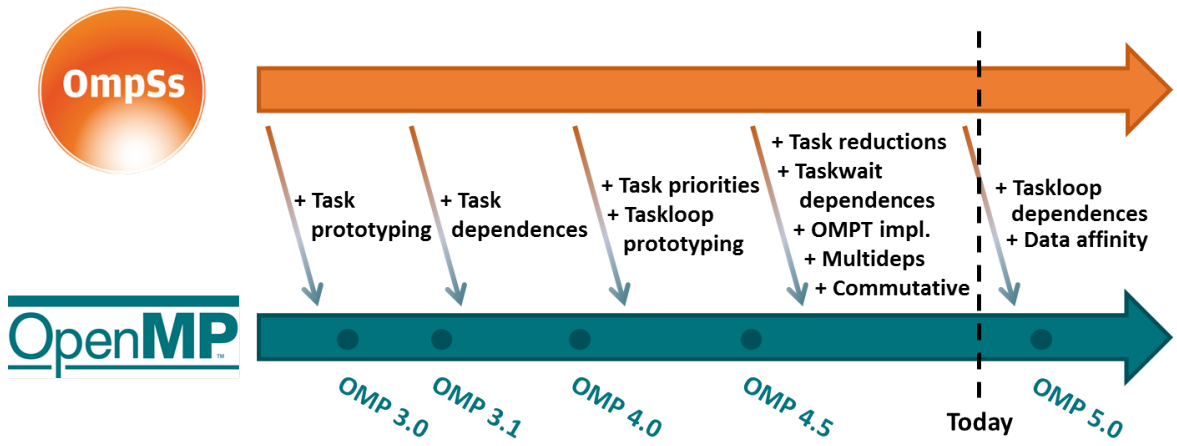


Fig. 5.1: OmpSs features introduced into the OpenMP programming model [13].

5.3 Annotating Programs

In this section, we describe how to annotate programs in order to enable OmpSs-2 functionalities. For the sake of simplicity, from now on we just show the annotations for C/C++. More information about Fortran annotations is presented in [13].

OmpSs-2 annotations must be written following a specific format. Directives are specified with the sentinel oss followed by the directive name and the clauses applied to it, as shown below.

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

5.4 Execution Model

One of the most notable differences between OpenMP and OmpSs-2 is that the latter does not require a parallel directive in the user application. Since OpenMP has a fork-join execution model, users have to indicate where starts and ends the parallel region by using that directive. In contrast, OmpSs-2, following the ideas of StarSs, implements a thread-pool execution model, so it implicitly creates the parallelism at the beginning of the application. Parallel resources can be seen as a pool of threads that the underlying runtime system will use to execute user tasks. However, the user is not able to control the pool of threads, since it is efficiently managed by the runtime system.

At the beginning of the execution, the OmpSs-2 runtime system creates an initial pool of worker threads (i.e. there is no master thread in OmpSs-2). The main function is wrapped in an implicit task, which is called the main task, and it is added into the queue of ready tasks. Then, once one of the worker threads gets that task from the

queue, it starts to execute it. Meanwhile, the rest of threads wait for more ready tasks, which could be instantiated by the main task, or even by other running tasks.

Multiple threads execute tasks defined explicitly or implicitly by OmpSs-2 directives. In the next sections, we explain the functionalities of creating tasks, OmpSs-2 loops and the explicit synchronization points.

5.4.1 Tasks

OmpSs-2 allows the expression of parallelism through tasks, which are independent pieces of code that are executed by parallel resources at runtime. Once the program flow reaches a task declaration, it does not execute the task code, but it creates an instance of a task and delegates its execution to the OmpSs-2 runtime system. The runtime system uses the dependency information provided by the user to compute the dependencies with the previously created tasks, allowing the runtime to know the correct execution order of tasks. Once all dependencies of a task are satisfied, the task becomes ready and it will be eventually executed on a parallel resource. The execution can be immediate or deferred until later, according to task scheduling constraints and thread availability. In addition, a task execution blocks yielding the resources to another ready task once a *task scheduling point* (e.g. a taskwait) is reached.

In a program, users can declare a task using the following syntax:

```
#pragma oss task [clauses...]
{ ... }
```

Tasks allow multiple clauses in their declaration, e.g. the variable's data-sharings. The private data-sharing privatizes the variable with an uninitialized value. firstprivate also privatizes the variable, but it is initialized with the original value when the construct was encountered. Finally, the shared data-sharing makes the task to work with the original variable.

Since some tasks can depend on data produced by other tasks, OmpSs-2 allows dependencies to be added into the task's clauses. Input, output and input-output dependencies on multiple variables can be defined for a task with the clauses in(variable-list), out(variable-list) and inout(variable-list), respectively. Dependencies are explained in more detail in Section 5.5.

In addition, other clauses can be specified like priority, final and if, among others. More information can be found in [13].

5.4.2 Loops

In addition to tasks, parallelism can be also explicitly defined by OmpSs-2 loops. The OmpSs-2 loop construct is similar to an OpenMP/OmpSs taskloop, however, it does not explicitly create a specific number of tasks. In contrast, the runtime system creates

as many internal tasks as the number of CPUs involved in the loop computation. These internal tasks execute chunks of consecutive iterations, so it is a similar concept to the OpenMP for clause. The chunk size can be specified using the clause chunksize(size).

Dependencies can be also added into the loop's clauses. The dependency is over the whole iteration space, so loop iterations can be executed once all its dependencies are satisfied. Notice that loop iterations must be independent of each other.

The syntax to define a loop covering one or more for loops is the following:

```
#pragma oss loop [clauses...]
for (int i = 0; i < N; ++i) {
   ...
}
```

5.4.3 Nesting

Any directive that defines a task or a series of tasks can also appear within a task definition, allowing multiple levels of parallelism. The definition of multiple levels of parallelism can lead to a better performance of applications, since the OmpSs-2 runtime system can exploit factors like data or temporal locality between tasks. This functionality allows the implementation of recursive algorithms, e.g. the Fibonacci program.

5.4.4 Explicit Synchronization

OmpSs-2 also allows explicit synchronization points with the taskwait stand-alone directive. It specifies a wait on the *deep completion* of children tasks. Its use should be avoided as much as possible, since it totally cuts the parallelism of the parallel context from which is called. In addition, the OmpSs-2 dependency model provides alternative techniques to avoid the use of taskwait's, which are discussed in Section 5.5.

5.4.5 Example

An example of the previously explained concepts can be seen in Figure 5.2. In the example, the main function creates 500 individual tasks which initialize all positions of the array A. Then, it creates an OmpSs-2 loop which updates the values from the 100th to the 199th position in chunks of 20 positions (5 chunks). Both tasks and loop declare their corresponding dependencies defining a correct execution order. Finally, a taskwait is executed in order to wait for the previous tasks before returning from the main function. Note that this code is just an example to show the basic tasking constructs.

```
1  void func(int *B, int size) {
2     for(int i = 0; i < size; ++i) {
3        #pragma oss task out(B[i])
4        { B[i] = 10; }
5     }
6  }
7
8  int main () {
9     int A[500];
10    func(A, 500);
11
12    #pragma oss loop chunksize(20) inout(A[100;100])
13    for (int i = 100; i < 200; ++i) {
14       A[i] *= 2;
15    }
16    #pragma oss taskwait
17 }
```

Fig. 5.2: Snippet of code highlighting tasking constructs.

5.5 Dependency Model

In OmpSs-2, asynchronous parallelism is enabled by the use of data-dependencies between the different tasks of the application. Tasks usually require data in order to perform useful computation. For instance, a task could use some input data to do computation, producing new data that could be the input of other tasks.

When an OmpSs-2 program is being executed, the runtime system uses the information of data dependencies and the creation order of each task. Both information is used to produce execution-order constraints between tasks, which result in a correct execution order of the program. These execution-order constraints are called task dependencies.

When a new task is created, its dependencies are matched against of those of existing tasks. If a dependency, either Read-after-Write (RaW), Write-after-Write (WaW) or Write-after-Read(WaR), is found the task becomes a successor of the corresponding existing tasks. Tasks are scheduled once all their predecessors have finished or at creation if no predecessors have been found [13].

Dependencies allow the user to indicate which data is waiting for and producing each task. Notice that the user is the responsible for correctly setting the dependencies in each task. OmpSs-2 offers four basic types of dependencies:

- A task with in(*variable-list*) will not be able to run until all out, inout and concurrent predecessors are finished.
- A task with out(*variable-list*) will not be able to run until all in, out, inout and concurrent predecessors are finished.

- Declaring a task with inout(*variable-list*) is the same as declaring both in and out dependencies.
- A task with concurrent(*variable-list*) will not be able to run until all in, out and inout predecessors are finished. It is able to run concurrently with concurrent predecessors.

In addition, OmpSs-2 allows the declaration of a dependency on a memory position pointed by a pointer. Note that if the pointer is NULL, the dependency is ignored. For instance, the following example actually declares an output dependency on the a variable.

```
int a = 10;
int *ptr = &a;
#pragma oss task out(*ptr)
{ ... }
```

5.5.1 Region Dependencies

OmpSs-2 enables the possibility to declare dependencies on multiple elements of an array in a single expression. The runtime system is the responsible for efficiently managing these region dependencies. Users can define them in two different ways, which are shown in the example below. In the first, all elements of array in the range [lower, upper] (both included) are referenced. In the second, all elements of array in the range [lower, lower+(size-1)] (both included) are referenced.

```
#pragma oss task inout(array[lower:upper]) out(array[lower;size])
{ ... }
```

In addition, OmpSs-2 offers shaping expressions which allow the recasting of pointers into array types to recover the size of the dimensions that could have been lost across function calls. A shaping expression is one or more [size] expressions before a pointer. For instance, the following example declares an output dependency on the first two elements of the third and fourth rows of the ptr matrix.

```
int *ptr = malloc(N * N * sizeof(int));
...
#pragma oss task out(([N][N]ptr)[3:4][0;2])
{ ... }
```

5.5.2 Fine-Grained Early Release

In OmpSs-2 task nesting and dependencies can be perfectly combined, allowing some applications to take profit of this technique. In OpenMP/OmpSs, the dependency

```
#pragma oss task out(a, b, c) // Task T1
{
   #pragma oss task out(a) // Task T1.1
   { a = 100; }
   #pragma oss task out(b) // Task T1.2
   { b = 420; }

   c = 10;
}
#pragma oss task inout(a, b) // Task T2
{
   #pragma oss task inout(b) // Task T2.1
   { ++b }
   #pragma oss task in(b) inout(a) // Task T2.2
   { a += b * 10; }
}
```

Fig. 5.3: Snippet of code combining nesting and dependencies.

domain of a parent task and the dependency domains of its children tasks can be connected using a taskwait at the end of the parent's code. With this, the parent releases all its dependencies once all its children have finished. However, the taskwait prevents the parent from finishing its execution, in addition to the fact that some dependencies could be released before all children tasks have finished.

To solve it, OmpSs-2 presents a new model of dependencies in which the user is able to connect dependency domains from parents and children without using a taskwait. When a parent task finishes its execution, the runtime knows that it does not need the enforcement of its own dependencies and it will not create any further subtask. Therefore, all dependencies declared by the parent that are not declared by any alive subtask can be released. Only the dependencies needed by its alive subtasks need to be preserved.

This can be seen in Figure 5.3. In this example, T1 can finish its execution without waiting for its subtasks T1.1 and T1.2. In case they are still alive, T1 can only release the dependency on the variable c. Once T1.1 finishes, it will release the dependency on the variable a. The same will happen with T1.2 and the variable b. Finally, T2 will not be executed until T1, T1.1 and T1.2 have finished, although it does not access any of the conflicting variables.

5.5.3 Weak Dependencies

Some of the elements in the dependency clauses may be needed by the task itself, others may be needed only by its subtasks and others may be needed by both. Dependencies needed only by its subtasks only serve as a mechanism to connect the outer dependency

domain to the inner one. Therefore, it is not necessary to defer the task execution for these kind of dependencies, since the task does not perform any access by itself.

In order to solve it, OmpSs-2 provides weak counterparts of in, out and inout dependencies. The weak variants indicate that the task does not perform by itself any action that requires the enforcement of the dependency, but it will be performed by its subtasks.

In the previous example (Figure 5.3), T1 and T2's dependencies on variables a and b could be changed to weakout(a, b) and weakinout(a, b), respectively. This is because T1 and T2 do not actually access these variables. However, the dependency of T1 on c must remain as a strong dependency, since it is accessed by T1.

5.6 Task Scheduling

When a task reaches a scheduling point, the runtime system can decide to switch to another eligible task. A scheduling point may occur in a task generating code, in a taskyield directive, in a taskwait directive or just after completing a task.

The set of eligible tasks is initially formed by the tasks in the ready queue (i.e. tasks with all dependencies satisfied). Task switching implies the beginning of a non-previously executed task or resuming the execution of partially executed tasks. However, once a task has been executed by a thread, it can only be resumed by the same thread.

In addition, when a task is created with the if clause evaluating to false, the current task must suspend its execution until the new created task completes. Similarly, when a task generating code is encountered in a final context, the children task is immediately executed just after creating it.

5.7 Reference Implementation

The reference implementation of OmpSs-2 is based on Mercurium and Nanos6 tools. On the one hand, Mercurium is a source-to-source compiler which provides the necessary support for transforming the OmpSs-2 directives into a parallelized version of the application. On the other hand, Nanos6 is a runtime system which provides all services to manage the parallelism in user applications. We describe more in detail both tools in Section 7.1.

6 | MPI+OmpSs Interoperability Library

Normally, hybrid applications have different phases of computation and communication. Computation phases are usually parallelized with a parallel region or tasks, however communication phases are only executed by a single thread. Thus, parallelism is open at the beginning of computation phases and it is closed before entering communication phases, following a fork-join approach. The synchronization point at the end of the computation phase prevents the overlap of computation and communication. Furthermore, if the application performs several timesteps repeating the computation & communication pattern, the fork-join mechanism introduces a significant overhead as well as it prevents the overlap of different timesteps.

These problems can be avoided when using the MPI+OmpSs Interoperability library [9] which seeks the enhancement in performance and programmability of hybrid applications by managing the interaction between both MPI and OmpSs programming models.

Figure 6.1 shows the different software levels involved in a hybrid MPI+OmpSs application along with the interoperability library. The MPI implementation library and the OmpSs' runtime system, which is called Nanox [14], work on top of the operating system. Then it is the turn of the interoperability library which redefines all blocking MPI routines and interacts with the OmpSs runtime library. Finally, the application is placed on top of those three libraries.

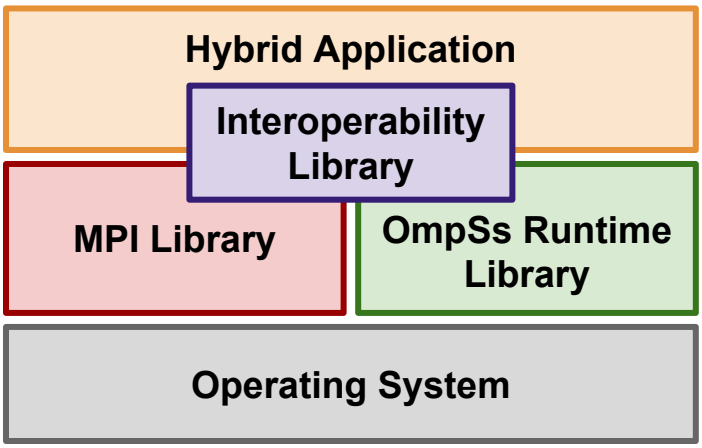


Fig. 6.1: Software stack in a hybrid MPI+OmpSs+Interoperability application.

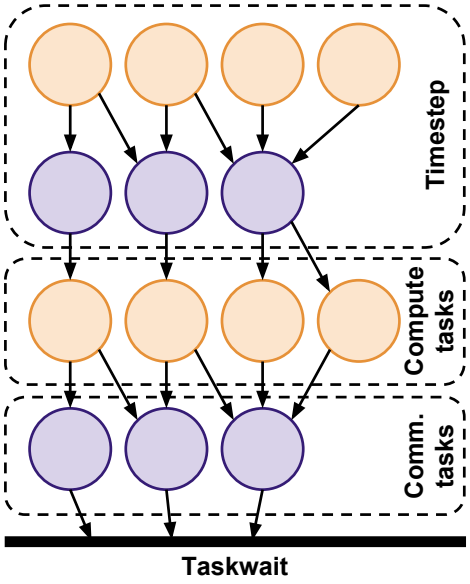


Fig. 6.2: Dependency graph with both computation and communication tasks.

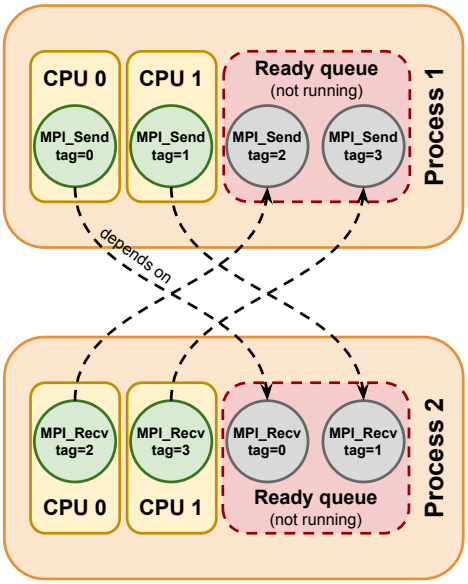


Fig. 6.3: Deadlock situation between 2 processes and 2 CPUs per process.

6.1 Taskifying Communications

To make the most of the interoperability mechanism, users should taskify both computation and communication phases of the hybrid application with multiple tasks in each phase, in order to exploit the parallelism in both of them. Computation phases should be parallelized in a similar way than the original hybrid application. Communication phases could create a task for sending/receiving blocks of data. All tasks must declare the corresponding dependencies on its input and output data, thus guaranteeing a correct execution order of the tasks. In this way, tasks from different timesteps are also connected through dependencies, thus avoiding the use of an explicit synchronization point (e.g. taskwait) between timesteps.

An example of this behavior is shown in Figure 6.2. In this case the application performs two iterations/timesteps, where each one is composed by a computation and a communication phase. Tasks are ordered using the corresponding dependencies and the only synchronization point is located at the end of the program.

We discussed dependencies in computation tasks in Section 5.5. Dependencies of communication tasks follow the same idea. From the local point of view of a process, tasks sending data (e.g. with MPI_Send) to another process must declare an input dependency on the source data buffer. In contrast, tasks receiving data (e.g. with MPI_Recv) from another process must declare an output dependency on the destination data buffer.

In the following subsection, we explain the risk of a deadlock which appears when taskifying blocking MPI operations.

6.1.1 Risk of Deadlock

As stated in [8], tasks that encapsulate blocking MPI calls have an unpredictable execution time depending on the MPI synchronization with the matching call in the remote process. This may cause deadlocks if the runtime system assigns cores to communication tasks and the task execution-order in the remote process is different.

Figure 6.3 shows a simple deadlock situation. The first MPI process devotes its two CPUs to execute the first two communication tasks which send data with tags 0 and 1, respectively. At the same time, the second MPI process devotes its two CPUs to execute the last two communication tasks which receive data messages with tags 2 and 3, respectively. Since they are not sending and receiving the same messages, both processes end up in a deadlock within the MPI blocking procedures.

Without the interoperability mechanism, deadlocks can be avoided by declaring a sentinel dependency in communication tasks. This guarantees the same execution-order of communication tasks in all processes. Although it does not allow the overlap of different communication operations, *one* communication task can be executed concurrently with computation tasks.

6.2 Benefits

The interoperability mechanism prevents processors from blocking inside the MPI library when calling blocking MPI operations. With this, the aforementioned risk of deadlocks disappears, so sentinel dependencies of communication tasks can be totally removed. This allows multiple communication tasks to run in parallel with other computation tasks, without any risk. Furthermore, if the application has multiple timesteps, the library allows the overlap of different timesteps. In summary, this library allows the overlapping of multiple computation and communication tasks, and also between different application timesteps.

In addition, the interoperability library supports applications which use non-blocking operations. Although non-blocking routines are not intercepted, the interoperability mechanism will be enabled when calling to blocking wait procedures as MPI_Wait.

6.3 Software Requirements

This library requires an MPI implementation supporting the multi-threaded mode (i.e. MPI_THREAD_MULTIPLE) and non-blocking MPI operations. These requirements are very easy to fulfill, since almost all MPI implementors (e.g. MPICH, Intel MPI and OpenMPI) integrated these features a long time ago. In addition, it requires an OmpSs implementation supporting the condition variable and the polling service APIs, which are explained in the next subsections.

```
// An opaque condition variable object
typedef void* nanos_wait_cond_t;

// Creates a new condition variable
void nanos_create_wait_condition(nanos_wait_cond_t *cond);

// Blocks current task on the condition variable
void nanos_block_current_task(nanos_wait_cond_t *cond);

// Wakes a task waiting on a condition variable
void nanos_signal_wait_condition(nanos_wait_cond_t *cond);
```

Fig. 6.4: OmpSs task condition variable API.

6.3.1 Task Condition Variable API

The Nanox [14] runtime library has to provide an API routine to block a task in a condition variable and another one to signal the condition variable in order to resume the task. The interoperability library uses both API routines to block a task within the interception code of a blocking MPI operation and to resume it once the MPI request has completed, respectively. The complete API is shown in Figure 6.4.

6.3.2 Polling Service API

The interoperability library defines a function that checks the completeness of all pending MPI requests, which have been produced when intercepting blocking MPI routines. In order to guarantee program progress, this function should be called *periodically* by the OmpSs runtime system.

The polling service API (Figure 6.5) provides the capability of registering polling functions with an opaque parameter, which will be periodically called passing the corresponding parameter. The same function can be registered with different parameters, considering them as different polling services. A polling service can be unregistered in two ways: 1) by returning a '1' from the polling service or 2) by explicitly unregistering it through the corresponding API routine. The API should be implemented by the OmpSs runtime system.

6.4 Operation

The interoperability library takes advantage of the fact that MPI defines the functions starting with the "MPI_" prefix as weak symbols. This means that they can be redefined in order to intercept calls performed by user applications. Furthermore, MPI routines starting with the "PMPI_" prefix implement the original MPI operations and cannot

```
// An opaque condition variable object
typedef int (*nanos_polling_function_t)(void *service_data);

// Registers a function and a parameter to be called periodically
// Registering the same function with different parameters is allowed
// The function is unregistered automatically if it returns a '1'
void nanos_register_polling_service(const char *service_name,
     nanos_polling_function_t service_function, void *service_data);

// Unregisters a function and parameter combination
void nanos_unregister_polling_service(const char *service_name,
     nanos_polling_function_t service_function, void *service_data);
```

Fig. 6.5: OmpSs polling service API.

be redefined. Thus, calls to the PMPI interface cannot be intercepted and they are usually used by external libraries to directly call to the authentic MPI routines.

We explain the interoperability mechanism through an example. Figure 6.6 shows the code that is executed when the application performs an MPI_Recv call from inside a task. The first operation performed at line 3 is to transform the blocking call to its non-blocking counterpart, in this case an MPI_Irecv. The code then checks whether the operation is immediately completed with MPI_Test (line 6). In such case, the function returns without blocking the task, since the MPI operation has completed. Otherwise, a ticket object is created and filled with the information about the ongoing MPI operation and a new condition variable (line 9). The ticket is next added into the internal list of pending tickets.

In case the interoperability mechanism is not enabled, the polling function is called (line 14) until the current operation completes (i.e. to make progress). In fact, when it is not enabled, the library code simulates the blocking routine's behavior. The task is then blocked on the condition variable (line 17) independently of whether the mechanism is enabled or not. However, in case it is not enabled, the task should return almost immediately, since the condition variable may have already been signaled by the polling function.

The library implements a polling function (line 22) which is periodically called by the OmpSs runtime system. This function checks the completeness of all pending MPI operations (line 25). When an MPI operation completes, the task waiting for the MPI operation is signaled (line 28) and returned to the runtime system's ready queue.

It is worth noting that all other point-to-point and collective blocking MPI primitives are intercepted and managed similarly. This mechanism can be enabled and disabled through the MPI_Pcontrol routine, which is designed for external tools and libraries.

In addition, some changes were made in the Nanox runtime system [14] to efficiently support the interoperability library. For instance, a new scheduler was implemented in order to dedicate one or more threads to executing communication tasks [9]. This

idea is similar to the one proposed in [8], which marks communication tasks with the target(comm_thread) clause in order to execute them in a communication thread.

```
int MPI_Recv(void *buf, ..., MPI_Status *status) {
   MPI_Request request;
   int err = MPI_Irecv(buf, ..., &request);

   int completed = 0;
   MPI_Test(&request, &completed, status);
   if (!completed) {
      Ticket ticket(&request, status);
      nanos_create_wait_condition(&ticket._cond);
      _pendingTickets.add(ticket);

      if (!Interop::isEnabled()) {
         while (!ticket.isFinished()) {
            Interop::poll();
         }
      }
      nanos_block_current_task(&ticket._cond);
   }
   return err;
}

void Interop::poll() {
   for (Ticket &ticket : _pendingTickets) {
      int completed = 0;
      MPI_Test(ticket._request, &completed, ticket._status);
      if (completed) {
         _pendingTickets.remove(ticket);
         nanos_signal_wait_condition(&ticket._cond);
      }
   }
}
```

Fig. 6.6: Original interception of the MPI_Recv routine and implementation of the polling function in the interoperability library.

7 | Tools and Methodology

This chapter describes the main tools used to develop this project and the methodology followed.

7.1 Tools

As stated in Section 1.1, the main objectives of this project are to improve the interoperability between MPI and task-based programming models as well as to propose a similar interoperability mechanism for GASPI. Although the mechanism is for any task-based programming model, we use OmpSs-2 in order to exemplify the use of the mechanism. In the following subsections, we describe the implementations of OmpSs-2, MPI and GASPI used in this project.

Moreover, we describe the MPI+OmpSs interoperability library, given that some of our contributions relate to the improvement of that library.

Finally, we briefly explain the Extrae and Paraver tools, which we will use in the evaluation of our approach. Extrae and Paraver are a tracing tool and a visualization tool, respectively.

7.1.1 Mercurium

Mercurium [15] is a source-to-source compilation infrastructure developed at the BSC, which supports the C, C++ and Fortran languages. Mercurium is mainly used in the Nanos environment to implement OpenMP but since it is easily extensible, it has been used to implement other programming models and compiler transformations. Some examples include Cell Superscalar(CellSs) and Software Transactional Memory.

Mercurium is designed using a plugin-based architecture, where plugins represent different phases of the compiler. These plugins are written in C++ and dynamically loaded by the compiler according to the chosen configuration. Code transformations can be implemented in terms of source code, thus there is no need to modify or know the internal syntactic representation of the compiler.

Mercurium supports the family of OmpSs programming models. It is in charge of processing the OmpSs compiler directives and transforming them into runtime library

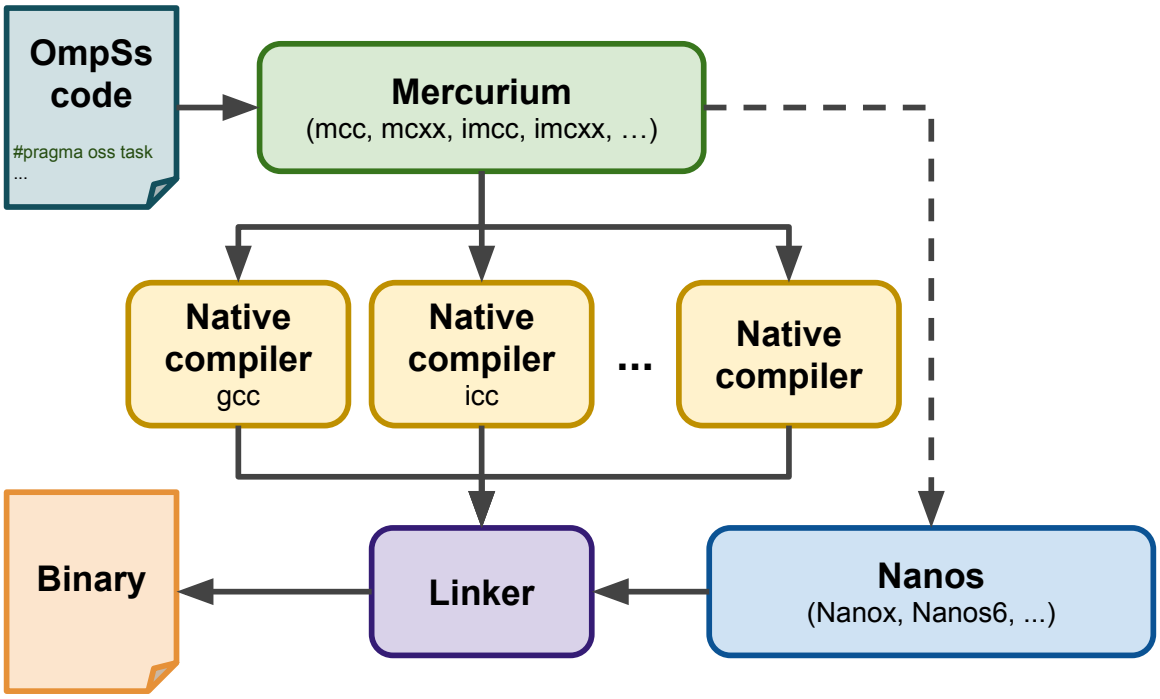


Fig. 7.1: Mercurium workflow for OmpSs source codes.

function calls. Moreover, Mercurium is able to generate code for different platforms depending on its target device (e.g. CPU, GPU and FPGA).

The workflow of Mercurium for OmpSs is shown in Figure 7.1. As can be seen, the user is responsible for delivering a directive-based source code which is processed by Mercurium. This latter transforms the OmpSs directives into the corresponding function calls of the Nanos runtime system. The transformed code is then compiled by the chosen native compiler (e.g. gcc and icc). Finally, the binary is linked to the Nanos runtime system's library.

7.1.2 Nanos6

Nanos6 [16] is a runtime system library developed at the BSC, which provides the services to manage all the parallelism in user applications. Nanos6 is responsible for scheduling and executing the tasks annotated in user applications, preserving the implied task dependency constraints.

Actually, Nanos6 is the successor of the Nanox [14] runtime system and it provides support for all constructs of the OmpSs-2 programming model, which is explained in Section 5. Some of the services that Nanos6 provides are task creation, synchronization, data movement, several task scheduling policies and support for resource heterogeneity. The Nanos6 structure is shown in Figure 7.2.

The Nanos6 library provides a well-defined API which can be called from any user application. Nevertheless, it is highly recommended to use it along with Mercurium,

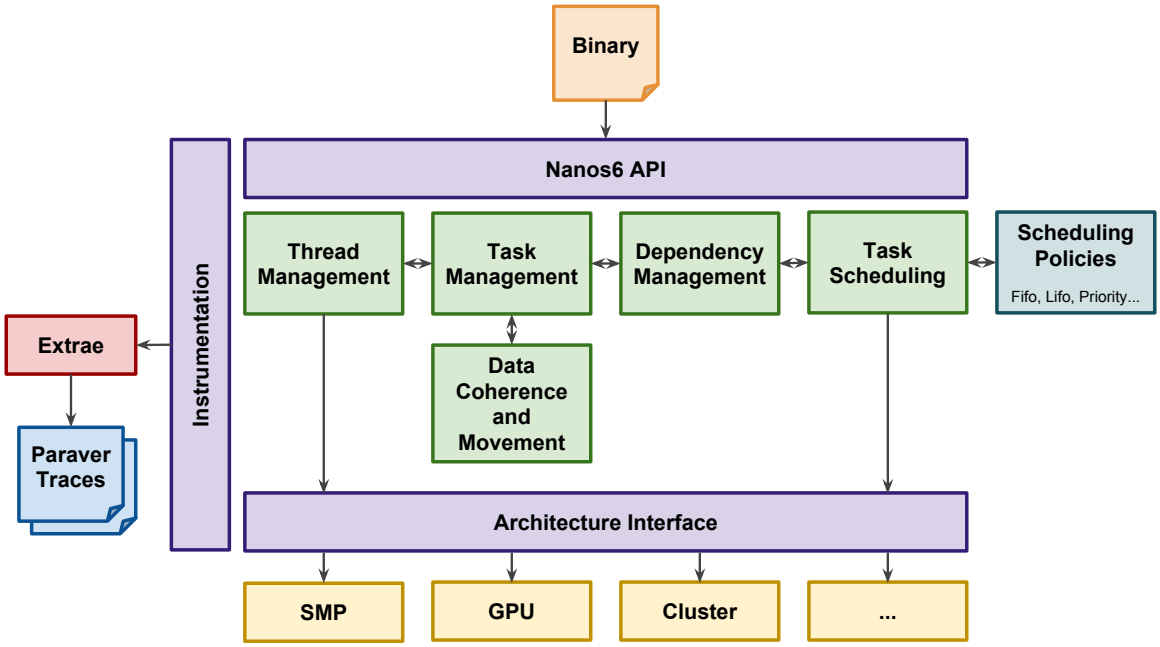


Fig. 7.2: Nanos6 runtime system structure.

which will transform the OmpSs directives from the desired application into calls to the Nanos6 API, and then, it will execute the native compiler passing the transformed code.

In addition, Nanos6 provides an instrumented version which allows the generation of execution traces with Extrae. Basically, Nanos6 calls the Extrae API to report when tasks are started, threads are paused/resumed, etc. The generated traces can be then visualized with Paraver for a post-mortem analysis. For instance, users are able to visualize what tasks and threads are being executed at each moment of the execution. Both tracing tools are explained in the following subsections.

7.1.3 MPICH

MPICH [17] is a high-performance and widely portable implementation of the Message Passing Interface (MPI) standard. This implementation is developed at the Argonne National Laboratory along with other collaborators.

The MPICH implementation is distributed as an open-source license and it has two main objectives. The first is to provide an MPI implementation that efficiently supports different computation and communication platforms, including commodity clusters (e.g. multicore architectures), high-speed networks (e.g. 10 Gigabit Ethernet, InfiniBand and Myrinet) and proprietary high-end computing systems (e.g. Blue Gene and Cray). The second objective is to enable cutting-edge research in MPI through an easy-to-extend modular framework for other derived implementations. In summary, MPICH is a high-quality reference implementation of the latest MPI standard.

7.1.4 GPI-2

GPI-2 [18] is an implementation of the GASPI specification and it is developed and maintained by the Fraunhofer ITWM. In fact, GASPI is an API specification which originates from the ideas and concepts from GPI (Global address Programming Interface). GPI-2 is the next generation of the original GPI, extending it with more features.

GPI-2 is mainly designed for asynchronous communication and provides a flexible, scalable and fault tolerant interface for parallel applications. By default, its interface is totally thread-safe and it is implemented taking into account that many user applications are multi-threaded. Moreover, it provides a mechanism to define memory segments to support heterogeneous systems (e.g. Intel Xeon Phi). In addition, it supports synchronous point-to-point primitives, which could be equivalent to the MPI_Send and MPI_Recv operations from the MPI standard.

7.1.5 MPI+OmpSs Interoperability Library

The MPI+OmpSs Interoperability Library [9], which was developed at the BSC, seeks to improve the interoperability between these two programming models. This library targets hybrid MPI+OmpSs applications, specifically the ones that taskify computation and MPI communication across the processes. Thus, computation and communication can overlap. This mechanism allows the execution of other ready tasks when a communication task is blocked inside MPI.

This library intercepts each blocking MPI call (e.g. MPI_Recv) and converts it to a call to its non-blocking counterpart (i.e. MPI_Irecv). The resulting MPI requests are stored and managed internally. Then, the calling task is blocked through a specific API. Furthermore, thanks to a polling service API, these pending requests are checked periodically to find out which of them are complete. When all the requests of a task are complete, the task is resumed, and finally, it can return from the MPI call.

7.1.6 Extrae

Extrae [19] is a library, also developed at the BSC, devoted to generating Paraver trace-files for a post-mortem analysis. Extrae is a tool that uses different interposition mechanisms to inject probes into the target application in order to gather information regarding the application performance. In summary, it collects performance metrics at known application points to finally provide the performance analyst a correlation between performance and the application execution. In addition, Extrae supports multiple platforms (e.g. Linux clusters, Cray and nVidia GPUs) as well as several parallel programming models (e.g. MPI, OpenMP and OmpSs).

7.1.7 Paraver

Paraver [20] is a visualization tool developed at the BSC, which allows users to have a qualitative global perception of the application behavior by visual inspection, and then, to be able to focus on the detailed quantitative analysis of the problems.

The analytical power of Paraver is based on two main pillars. The first is that its trace format has no semantics. This means that extending the tool to support new performance data or new programming models requires no changes to the visualizer, just to capture such data in a Paraver trace.

The second pillar is that the metrics are not hardwired on the tool but programmed. To compute them, the tool offers a large set of time functions, a filter module, and a mechanism to combine two time lines. This approach allows to display a huge number of metrics with the available data (e.g. cache misses, frequency and IPC).

Along with Extrae, they form a very powerful tool to analyze application executions. They are especially useful when an application does not achieve the desired performance, thus users can look in detail which are the performance problems. In addition, they allow to easily understand the application's behavior, even without having seen the application's code.

7.2 Methodology

In this section we describe the methodology followed to satisfactorily develop this project. We followed the *Design Science Research Cycles* [21] methodology, which clearly identifies three different research cycles: the *Relevance* cycle, the *Rigor* cycle and the *Design* cycle.

As an introduction, the relevance cycle bridges the contextual environment of the research project with the design science activities. The rigor cycle links these design science activities with the knowledge base of scientific foundations. Finally, the design cycle iterates between the core activities of designing, building and evaluating the artifacts and processes of the research [21]. Figure 7.3 shows the flow of this methodology and these cycles. The following subsections describe them more in detail.

7.2.1 Relevance Cycle

Design science research is motivated mainly by the desire to improve the environment by introducing new and innovative artifacts, and also, the processes for building these artifacts. The *environment* or *application domain* is formed by the people, the organizational systems and the technical systems that interact to work toward an objective. Finally, opportunities and problems are identified from this application domain, being the ones to address during the research project [21].

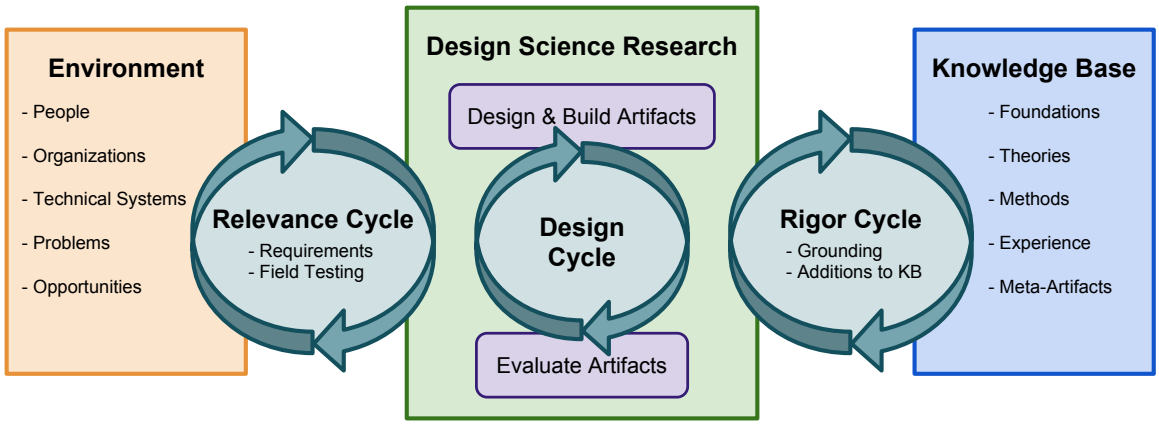


Fig. 7.3: Design Science Research Cycles.

The *relevance* cycle initiates design science research with an application domain which provides the requirements of the research (e.g. opportunities and problems to solve) as inputs, along with the acceptance criteria for the ultimate evaluation of the research results. The output of the design science research is then returned to the environment to evaluate whether more iterations are needed or not. This evaluation is performed by field testing, which should detect whether the artifact has functionality deficiencies or whether the input (e.g. the requirements) for the design science research was incorrect [21].

The application environment and the relevance cycle can be seen on the left side of the Figure 7.3. As stated above, the relevance cycle mainly provides the input for the design science research.

7.2.2 Rigor Cycle

Design science draws from a large *knowledge base* (also *KB* in Figure 7.3) of scientific theories and engineering methods that provides the foundations for rigorous design science research. The knowledge base also includes 1) the experience and expertise that define the state-of-the-art in the application domain and 2) the existing artifacts and processes (or meta-artifacts) found in the application domain [21].

The *rigor* cycle provides past knowledge to the research project to ensure that the produced designs are actually research contributions and not routine designs based upon the application of well-known processes. In the cycle, there are additions to the knowledge base which include extensions to the original theories and methods made during the research, the new meta-artifacts and all experience gained while performing the research. These knowledge base contributions are important to produce an impact on the academic audience as well as the environment contributions for the professional audience [21].

In this case, the knowledge base and the rigor cycle can be seen on the right side of the Figure 7.3.

7.2.3 Design Cycle

The *design* cycle is the core of the *design science research*, as can be seen in Figure 7.3. It iterates between the design and construction of an artifact, its evaluation and the subsequent feedback to refine the next designs [21]. The nature of this cycle is the generation of design alternatives and its evaluation against the requirements until a satisfactory design is achieved [22].

The requirements are provided by the relevance cycle, whilst the design and process theories and methods are provided by the rigor cycle. The design cycle is where the hard work of design science research is done. It is worth noting that the design cycle has a relative independence on the other cycles while the research is performed. In addition, it is important to maintain a proper balance between the efforts spent in the building and the evaluation of an artifact. In case these efforts are not well-balanced, the resulting artifact could have important deficiencies [21].

8 | Enhancing the MPI Interoperability

This chapter covers the details of our proposal for enhancing the interoperability mechanism between the MPI and task-based programming models. On the one hand, we explain the design of our proposal. The design part is one of the most important ones, since it defines the way users can enable and take benefit from the interoperability mechanism.

It also defines the interface that the distributed-memory and shared-memory programming models should use to interact with each other. In this case, the interface is designed thinking in MPI and OmpSs, but it should be completely generalized to any programming model.

On the other hand, we explain how the proposed functionalities have been integrated into the interoperability library in order to offer a competitive performance and an appropriate programmability to users.

Notice that this chapter is part of the Design Cycle of the Design Research Method, which is described in Section 7.2.

8.1 New Level of Thread Support

One of the most important aspects, if not the most important, should be to facilitate the use of the proposed functionalities to users. Designers must define what the users can expect from the functionality, which is its exact behavior and how can be activated. For that reason, in this section we define how the interoperability mechanism can be enabled from user applications. We also explain which is the behavior when the mechanism is enabled and disabled.

The MPI standard already defines a way to initialize the MPI library with different levels of thread support. This is explained in detail in Section 3.3. The highest level defined by the current MPI standard is MPI_THREAD_MULTIPLE, which is in turn the least restrictive when using multiple threads.

Following the idea of MPI levels of thread support, we propose a new threading level called MPI_TASK_MULTIPLE, being its constant value monotonically greater than the

existing MPI_THREAD_MULTIPLE constant. This threading level activates the interoperability mechanism (Chapter 6) during all the execution of the hybrid application. Note that MPI implementors could easily support this mechanism through this new level of thread support.

By initializing MPI with this threading level, users can call blocking MPI operations from an *infinite* number of tasks at the same time. The risk of a deadlock (Section 6.1.1) disappears, since underlying hardware threads are never blocked within blocking MPI routines and they are able to execute other ready tasks while the corresponding operations are not completed. Even so, the original *blocking* semantics is maintained, since a blocking MPI procedure does not return until the operation completes.

Figure 8.1 shows an example of how a hybrid MPI+OmpSs code may use this new thread support level to write portable applications. First, the application initializes the MPI library passing MPI_TASK_MULTIPLE as the desired threading level (line 6). The MPI initialization routine returns (in provided) the provided threading level.

If the new threading level is supported, the application defines a sentinel variable to NULL (line 7), which will be ignored by the runtime system. Otherwise, it sets the sentinel variable to one (line 8), so that communication tasks will be serialized. This is used in lines 12-13, where the communication tasks are created featuring a regular dependency on the block these will work on, as well as an artificial inout annotation of the sentinel variable to serialize the execution of these tasks and avoid deadlocks. Note that this can be also performed using a regular if statement by instantiating tasks with or without the artificial dependency depending on the provided threading level.

```
1  int *sentinel; // Sentinel used to serialize communication tasks
2
3  int main(int argc, char * argv[]) {
4     int provided;
5     // MPI_TASK_MULTIPLE = MPI_THREAD_MULTIPLE + 1
6     MPI_Init_thread(&argc, &argv, MPI_TASK_MULTIPLE, &provided);
7     if (provided == MPI_TASK_MULTIPLE) sentinel = NULL;
8     else sentinel = (int *) 1;
9
10    for (int i = 0; i < NT; i++) {
11       // Dependency enforced only if *sentinel != 0
12       #pragma oss task inout(tile[i]) inout(*sentinel)
13       communication_task(tile[i]);
14    }
15    #pragma oss taskwait
16 }
```

Fig. 8.1: Example of a portable MPI initialization using MPI_TASK_MULTIPLE.

8.2 Generalizing the API

Another important matter is the definition of a generic programming interface (or API) between the distributed-memory and the task-based shared-memory programming model. In our case, these are the MPI and OmpSs programming model, however the interface should be as generic as possible, allowing other programming models to easily implement it. The main idea is to facilitate the task of designing and implementing the interoperability mechanism for other programming models. Furthermore, we will use this same interface when we propose the interoperability mechanism for GASPI, detailed in Chapter 9.

In this section we propose a generic API to block and unblock tasks, replacing the previous task condition variable API (Section 6.3.1). This task block/unblock API aims to support efficient execution of blocking operations in any task-based runtime system. It is composed of three functions, which are shown in Figure 8.2. This API is also known as the pause/resume API.

The first function, get_current_blocking_context, informs the runtime system that the current task is about to enter a block-unblock cycle. The function configures everything needed to handle one round trip, and returns an opaque pointer to runtime-specific data. This data is also known as a blocking context. A blocking context is valid only for one pause-resume cycle, and requesting a new context invalidates the currently active one.

Once obtained a blocking context, the block_current_task function can be called in order to suspend the execution of the current task. The parameter must be the current blocking context of the invoking task. The unblock_task function indicates that the task associated to the blocking context can be resumed. This function can be called by any thread over a valid blocking context. In addition, this API does not enforce any specific order in which block_current_task and unblock_task should be called in a block-unblock cycle. For instance, it is correct to call to unblock_task before the task has actually called to block_current_task. In this case, the calling task should return from block_current_task as soon as possible. Therefore, the only requirements are that (1) each block_current_task call must match to an unblock_task call and (2) each blocking context can be used in a single block-unblock cycle.

```
// Get the blocking context of the current task
void *get_current_blocking_context();

// Block the current task
void block_current_task(void *blocking_context);

// Unblock a task passing its blocking context
void unblock_task(void *blocking_context);
```

Fig. 8.2: Task block/unblock API.

```
async_handler = start_async_op(...);
void *blocking_ctx = get_current_blocking_context();
associate(async_handler, blocking_ctx);
block_current_task(blocking_ctx);
```

Fig. 8.3: Code that performs the block operation.

```
async_handler = wait_until_one_async_op_finishes();
void *blocking_ctx = get_assigned_blocking_context(async_handler);
unblock_task(blocking_ctx);
```

Fig. 8.4: Body of the code that handles the unblocking of the operation.

The general usage pattern consists in replacing blocking operations by either asynchronous or non-blocking counterparts, and to let the runtime perform the actual blocking. The runtime should then schedule other ready tasks during the blocking period. This usage scheme is shown in Figure 8.3.

Asynchronous operations that support callbacks can use the callback function to unblock the task. However, some operations do not support callbacks, so a different thread has to periodically check their completion and unblock the corresponding tasks when they finish. Figure 8.4 shows the pattern that the body of the main loop of that thread could contain. Note that the thread that checks the completeness of operations and unblocks the corresponding tasks must have access to their blocking contexts.

In addition, the task block/unblock API can be used in conjunction with the polling service API (Section 6.3.2). The main idea is that a single polling service callback can be used for checking the completeness of multiple blocking operations. Figure 8.5 shows that pattern. During initialization, the function should be registered to ensure that the runtime calls it periodically. The body of the function is basically the code already shown in Figure 8.4, but adapted to work in a non-blocking fashion and with multiple operations.

In addition, this API also supports the registration of a callback for each operation. This is possible by passing the operation information through the service data parame-

```
int polling_callback(void *service_data) {
   while(have_ready_operations()) {
      async_handler = get_ready_operation();
      void *ctx = get_assigned_blocking_context(async_handler);
      unblock_task(ctx);
   }
   return 0;
}
```

Fig. 8.5: Callback code that handles multiple operations.

```c
int polling_callback(void *service_data) {
   operation_info_t *oi = (operation_info_t *) service_data;
   int finished = operation_has_finished(oi->async_handler);
   if (finished) {
      unblock_task(oi->blocking_ctx);
   }
   return finished;
}
```

Fig. 8.6: Callback code that handles a single operation.

```c
operation_info_t oi;
oi.async_handler = start_async_op(...);
oi.blocking_ctx = get_current_blocking_context();
register_polling_service("single-operation", polling_callback, &oi);
block_current_task(oi.blocking_ctx);
```

Fig. 8.7: Modified blocking code to use one callback per operation.

ter and by automatically unregistering the callback through its return value. Figure 8.7 shows the blocking-side code. In this case the callback is not registered during initialization. Instead, before blocking the current task, a callback function is registered along with the data associated with the operation. The callback function, which is shown in Figure 8.6, uses that data to recover the actual asynchronous operation and its associated blocking context. If it detects that the operation has finished, in addition to unblocking the task, it also returns a value that indicates that the callback should be automatically unregistered.

8.3 Bringing the Mechanism to Users

These proposals can be easily implemented inside any MPI implementation. However, we have implemented them on top of the existing interoperability library [9]. In this way, users can enable the pause/resume mechanism by just linking their hybrid applications to the library and requesting the new MPI_TASK_MULTIPLE threading level. The underlying MPI library can be any unmodified MPI implementation supporting at least the multi-threading level (e.g. MPICH, Intel MPI and OpenMPI).

When the interoperability library is properly linked to an application, it intercepts all MPI initialization/finalization function calls issued by the application. Thus the underlying MPI library is not initialized directly by the application, but by the interoperability library. This latter activates its interoperability mechanism if and only if the user has requested MPI_TASK_MULTIPLE and the MPI library does not support it. If the underlying MPI library already supports the new threading level or the user has requested a lower threading level, the interoperability library fully disables the

pause/resume mechanism. Although the mechanism is disabled, the library still intercepts all blocking MPI calls by default. In this case, when a blocking MPI function call is intercepted, the call is *redirected* to the original MPI blocking function defined by the underlying MPI library (i.e. using the PMPI interface).

8.4 Implementation Details

As mentioned above, we have implemented our proposals on top of the interoperability library. The first thing to do was to intercept both MPI initialization and finalization function in order to decide whether the interoperability mechanism should be enabled or not.

This is shown in Figure 8.8. When the library intercepts MPI_Init_thread, it calls the actual initialization function of the underlying MPI library (line 2). The interoperability mechanism is enabled if the user requests the new MPI_TASK_MULTIPLE threading level and the MPI library provides MPI_THREAD_MULTIPLE (lines 4-5). Otherwise, it is disabled (line 7). Note that a polling service is registered (line 5) so that the polling function is periodically called by the runtime system.

```
1  int MPI_Init_thread(..., int required, int *provided) {
2     int err = PMPI_Init_thread(..., required, provided);
3     if (required == MPI_TASK_MULTIPLE && *provided == MPI_THREAD_MULTIPLE) {
4        Interop::enable();
5        register_polling_service("interop", Interop::poll, NULL);
6        ...
7     } else Interop::disable();
8     return err;
9  }
10
11 int MPI_Finalize() {
12    int err = PMPI_Finalize();
13    if (Interop::isEnabled()) {
14       Interop::disable();
15       unregister_polling_service("interop", Interop::poll, NULL);
16       ...
17    }
18 }
```

Fig. 8.8: Interception of both MPI initialization and finalization routines.

Similarly, the finalization function is intercepted in order to unregister the polling service and finalize the mechanism, if it was enabled. Note that the MPI_Init is not shown since it automatically enables the lowest threading level.

Moreover, we have improved the processing of blocking MPI calls by using the proposed task block/unblock API and other minor improvements. Figure 8.9 shows the new code

that is executed when the application performs an MPI_Recv call from inside a task. The first operation performed at line 3 is to check if the interoperability mechanism is enabled. If this is not the case, the original blocking MPI_Recv operation is executed (line 15) using the PMPI interface. Otherwise, the blocking call is transformed into its non-blocking counterpart, in this case an MPI_Irecv (line 5). The code then checks if the operation is completed immediately. In such case, the function returns without blocking the task, since the MPI operation has been completed. Otherwise, a ticket object is created and filled with the information about the ongoing MPI operation and the current task blocking context (line 9). The ticket is next registered in an internal list of pending tickets and the task is paused (line 11).

MPI asynchronous operations do not feature a callback to wake up the thread once the operation is completed. To handle this, the library implements a polling service (line 18), that periodically checks if any pending MPI operation has completed (line 21). When an MPI operation completes, the task waiting for that MPI operation is

```
1  int MPI_Recv(void *buf, ..., MPI_Status *status) {
2     int err, completed = 0;
3     if (Interop::isEnabled()) {
4        MPI_Request request;
5        err = MPI_Irecv(buf, ..., &request);
6        MPI_Test(&request, &completed, status);
7        if (!completed) {
8           Ticket ticket(&request, status);
9           ticket._blocking_ctx = get_current_blocking_context();
10          _pendingTickets.add(ticket);
11          block_current_task(ticket._blocking_ctx);
12       }
13       return err;
14    }
15    return PMPI_Recv(buf, ..., status);
16 }
17
18 void Interop::poll() {
19    for (Ticket &ticket : _pendingTickets) {
20       int completed = 0;
21       MPI_Test(ticket._request, &completed, ticket._status);
22       if (completed) {
23          _pendingTickets.remove(ticket);
24          unblock_task(ticket._blocking_ctx);
25       }
26    }
27 }
```

Fig. 8.9: New interception of the MPI_Recv routine and implementation of the polling function in the interoperability library.

resumed (line 24) and returned to the runtime system's ready queue. Note that it uses the approach of having a single polling service for multiple blocking operations.

For the sake of simplicity, locking is omitted in Figure 8.9. However, in both the original and our updated implementation, threads must acquire a mutex to safely perform operations on the _pendingTickets list. The statement which registers a new ticket into the list (line 10) and the whole body of the polling function (line 18) should be protected with a mutex.

It is worth noting that in our implementation the polling function is only called by the polling thread, while in the original implementation [9] it is constantly called by all communication tasks inside the interception code when the interoperability mechanism is not enabled. This could degrade the overall performance, since it could produce significant overhead and contention. For this reason, our implementation is more efficient when the mechanism is not enabled (i.e. standard behavior).

Finally, other secondary tasks were performed as minor bug-fixes, simplification of the original code, etc.

9 | Enhancing the GASPI Interoperability

This chapter contains the details of the interoperability mechanism that we propose for the GASPI and task-based programming models. This chapter follows an organization similar to the previous one.

One the one hand, we explain in detail the design of our approach. We first introduce the main specific hindrances when trying to build a scalable hybrid GASPI+OpenMP application. We also explain which are the main ideas behind our proposal to solve all these problems. Then, we define how the user can enable our mechanism through a new GASPI initialization routine which offers the possibility of requesting a GASPI mode, as in MPI_Init_thread. Next, we explain in detail how the mechanism is applied to the waiting routines for both local completion (i.e. gaspi_wait) and remote completion (i.e. gaspi_notify_waitsome). Finally, we introduce our proposal for grouping different GASPI queues in order to facilitate and improve the queue's multiplexing in user applications.

On the other hand, we explain how the GPI-2 implementation has been extended to support our proposed functionalities. Our extension benefits from the generic task block/unblock API (Section 8.2) and the polling service API (Section 6.3.2).

Notice that this chapter is also part of the Design Cycle of the Design Research Method, which is described in Section 7.2.

9.1 Main Ideas

Generally, hybrid applications only parallelize computation phases, while communication phases are executed by a single thread. As explained in Chapter 6, these techniques are suboptimal and prevent applications from being scalable.

In addition, hybrid GASPI+OpenMP applications take advantage of the asynchronous operations (e.g. gaspi_write and gaspi_read). As explained in Section 4.7, their routines are asynchronous and return *immediately*, so they cannot produce a deadlock. Therefore, hybrid applications usually instantiate a different task for each communication operation to be performed. A taskwait is declared after creating the communication tasks. Once these tasks are finished, the main thread waits for the completion of all

```
// Parallel communication
for (int i = 0; i < NT; ++i) {
   #pragma oss task
   communicate(tile[i]);
}
#pragma oss taskwait
wait_all_communications(NT);
// Parallel computation ...
```

Fig. 9.1: Example of a common GASPI+OmpSs parallelization strategy.

operations by using gaspi_wait or gaspi_notify_waitsome, depending on the communication pattern. Figure 9.1 shows this strategy. Note that this parallelization could be easily implemented with an OpenMP for.

Nevertheless, this parallelization strategy is still suboptimal, since the application has to wait until all communications finish in order to execute the next computation phase. Further advanced strategies could be implemented, but they are extremely complex and they require a major change in applications.

The idea we propose is to instantiate multiple tasks performing one or more communication operations. These tasks could wait their own operations, independently of the operations performed by other tasks. Figure 9.2 shows the strategy which we are proposing. The function that waits the operations performed by a task should apply the pause/resume mechanism in order to execute other ready tasks while the operations are not completed, thus avoiding deadlocks. Notice that the application should be taskified following the ideas presented in Section 6.1.

The two available GASPI routines for waiting operations are gaspi_wait and gaspi_notify_waitsome for local and remote completion, respectively. The gaspi_notify_waitsome routine allows to wait for a specific range of notifications, thus users can wait for the remote completion of a specific group of operations. However, gaspi_wait waits for the local completion of all operations posted to a specific queue. Since users cannot wait for specific operations, we should reconsider the functionality of this routine. This is discussed in the following sections.

9.2 Initialization Modes

GASPI provides a basic initialization routine, which is called gaspi_proc_init, and it does only accept a single timeout parameter. However, we want to offer to users the possibility of enabling the interoperability mechanism in a similar way than in MPI. For this reason, we propose a new concept for the GASPI standard, which we call GASPI modes. This will be really useful in the future when other modes have to be supported. The proposed modes are the following ones:

```
// Parallel communication
for (int i = 0; i < NT; ++i) {
   #pragma oss task inout(tile[i])
   {
      communicate(tile[i]);
      wait_operation(i);
   }
}
// Parallel computation ...
```

Fig. 9.2: Example of the proposed GASPI+OmpSs parallelization strategy.

- **GASPI_MODE_STANDARD** enables the original GASPI operation, and it is equivalent to calling to gaspi_proc_init.
- **GASPI_MODE_TASK** enables the interoperability mechanism on top of the original GASPI operation.

These values are monotonic, being GASPI_MODE_STANDARD $<$ GASPI_MODE_TASK.

The proposed initialization function is called gaspi_proc_init_mode, which is shown below. The idea is that users can initialize the GASPI library requesting a specific mode. If possible, the library will provide the required mode. Failing this, the call will return the least supported level such that provided $>$ required (thus providing a higher mode than the required by the user). If the user requirement cannot be satisfied, then the call will provide the highest supported mode. Note that this function works exactly like MPI_Init_thread.

```
gaspi_return_t gaspi_proc_init_mode(gaspi_mode_t required,
      gaspi_mode_t *provided, gaspi_timeout_t timeout);
```

By properly initializing the library with GASPI_MODE_TASK, all functionalities related to the interoperability mechanism will be enabled. In this case, we do not show how to perform a portable GASPI initialization since Figure 8.1 already shows the same for MPI.

9.3 New Blocking/Timeout Mode

In addition to GASPI modes, we also propose a new timeout (or blocking mode) to define the behavior of specific blocking calls. Since all blocking routines already accepts a timeout parameter, no routine has to be modified. The new blocking mode is called MPI_BLOCK_TASK. When passing this blocking mode to a blocking call, the pause/resume mechanism will be enabled and the calling task will be blocked, allowing other ready tasks to be executed. The runtime services will unblock the task once the event or the operation is complete.

This mode can be used only when the GASPI_MODE_TASK is enabled. Note that although this mode is enabled, users have to explicitly specify in which routine calls want to apply the interoperability mechanism. This is useful since sometimes users may not want to enable this mechanism in some blocking function calls. This blocking mode is valid only in gaspi_wait and gaspi_notify_waitsome, since they are the only ones in which we could take benefit from the pause/resume mechanism.

9.4 Local Completion

As mentioned above, the gaspi_wait routine allows to wait for the local completion of *all* operations in a specific queue. This means that a task cannot wait only for its own operations. One of the solutions is to use as many queues as possible in order to multiplex operations. However the maximum number of queues allowed by GASPI implementations is usually low (i.e. 16 queues in GPI-2), so this technique could not be enough when a lot of communication tasks are being executed simultaneously.

Another possible solution could be to extend the GASPI API to return a handle to a pending request (equivalent to MPI_Request) in GASPI routines as gaspi_write and gaspi_write_notify. In addition, a new wait function could be defined to wait for specific requests (equivalent to MPI_Wait). Nevertheless, this approach is dismissed since it requires a lot of effort and a major change in the GASPI API.

The last solution, which is the one that we propose, is to change slightly the semantics of the gaspi_wait when passing the GASPI_BLOCK_TASK timeout. When passing this timeout value, the routine waits for all operations performed by the calling task, independently of the queue used. Note that the queue parameter in this case is just a hint. In addition, the pause/resume mechanism is enabled so the task is blocked and the underlying hardware thread can execute other ready tasks while its operations do not complete. Thus, the calling task returns from the gaspi_wait function once all its operations complete.

Figure 9.3 shows an example in which a process, known as the sender process, executes the sender_code function and another one, known as the receiver process, executes the receiver_code function. The sender process instantiates one task for each tile to be written to the other process. The write is performed through the gaspi_write_notify function, which starts a write operation and notifies the remote process once the data is ready on the remote side. The notification id is the id of the tile to be sent. Then, the task blocks (but not the underlying hardware thread) until the write and notify operation is completed locally. Notice that this is similar to a sender task in MPI, and the dependencies of the communication task are released once the source buffer can be modified.

9.5 Remote Completion

The remote completion in GASPI is performed through gaspi_notify_waitsome. This function waits for at least one notification from a range of consecutive notification ids. The notification is actually sent by a remote process. The pause/resume mechanism can be enabled passing GASPI_BLOCK_TASK as the timeout parameter.

Figure 9.3 shows in the receiver_code function how to wait for a specific notification from inside a task. The task i from the sender process sends the tile i and a notification, which is being waited by the task i from the receiver process. Since it uses the GASPI_BLOCK_TASK timeout, the task is blocked and the underlying hardware thread can execute other ready tasks while the notification does not arrive.

The runtime services unblock the task once the notification arrives. When it arrives, the data sent by the remote process is guaranteed to be already written in the corresponding tile. Note that this is similar to a receiver task in MPI, and the dependencies of the communication task are released once the received data is ready.

9.6 Queue Groups

Generally, GASPI recommends multiplexing write/read operations into multiple queues, instead of using a single one. However, it is difficult to use different queues in an application, since usually requires to change substantially the application code. These changes are necessary to know which queues are the best candidates for being the target queue of new operations (e.g. which are the emptier ones) and which operations have been posted to each queue. For this reason, we propose a new GASPI functionality to create and manage queue groups, but being consistent with the original GASPI API.

The new GASPI functions for queue groups are shown in Figure 9.4. The idea is that a group of queues can be created by calling to gaspi_queue_group_create, passing the desired queue group id, the range of queues which composes the group (i.e. [queue_begin, queue_begin+queue_num)) and a queue distribution policy. Note that the participating queues should be already created before calling to this routine.

The API provides a routine to get a queue from a queue group, which is called gaspi_queue_group_get_queue. The distribution policy defines how queues from the group are distributed between different calls to the gaspi_queue_group_get_queue. For the moment, it provides the following distribution policies:

- **GASPI_QUEUE_GROUP_POLICY_DEFAULT** provides a different queue in each call to gaspi_queue_group_get_queue (i.e. Round-Robin).
- **GASPI_QUEUE_GROUP_POLICY_CPU_RR** distributes the forming queues across CPUs using Round-Robin. Each CPU will always get the same queue.

```
// Code executed by the sender
void sender_code(tile_t *tiles, int NT) {
   for (int i = 0; i < NT; ++i) {
      #pragma oss task in(tiles[i])
      {
         gaspi_queue_id_t queue;
         gaspi_queue_group_get_queue(0, &queue);
         gaspi_write_notify(...,
               i, /* Notification id */
               1, /* Notification value */
               queue, GASPI_BLOCK);

         gaspi_wait(queue, GASPI_BLOCK_TASK);
      }
   }
}

// Code executed by the receiver
void receiver_code(tile_t *tiles, int NT) {
   for (int i = 0; i < NT; ++i) {
      #pragma oss task out(tiles[i])
      {
         gaspi_notification_id_t notified_id;
         gaspi_notification_t value;

         gaspi_notify_waitsome(...,
            i, /* First notification id */
            1, /* Number of notifications */
            &notified_id, GASPI_BLOCK_TASK);
         assert(notified_id == i);

         gaspi_notify_reset(..., notified_id, &value);
         assert(value == 1);
      }
   }
}
```

Fig. 9.3: Example of sending and receiving data with the pause/resume interoperability.

```
gaspi_return_t gaspi_queue_group_create(gaspi_queue_group_id_t queue_group,
      gaspi_queue_id_t queue_begin,
      gaspi_number_t queue_num,
      gaspi_queue_group_policy_t policy);

gaspi_return_t gaspi_queue_group_get_queue(
      gaspi_queue_group_id_t queue_group,
      gaspi_queue_id_t * queue);

gaspi_return_t gaspi_queue_group_delete(gaspi_queue_group_id_t queue_group);

gaspi_return_t gaspi_queue_group_max(gaspi_number_t *queue_group_max);
```

Fig. 9.4: Proposed GASPI routines for managing queue groups.

When the number of CPUs exceeds the number of queues, these are distributed in a Round-Robin way. In addition, it tries to not provide the same queue to CPUs from different NUMA nodes.

Moreover, it provides a routine to delete a group of queues and another one to get the maximum number of allowed queue groups. These routines are gaspi_queue_group_delete and gaspi_queue_group_max, respectively.

An example of its usage is shown in Figure 9.3. Sender tasks get a queue from the queue group 0, where each task uses a different queue to post an operation. To take the maximum benefit of the queue group, it should be created with the GASPI_QUEUE_GROUP_POLICY_CPU_RR distribution policy. For instance, it could be created with the following parameters:

```
gaspi_queue_group_create(
      0, /* Queue group id */
      0, /* First queue id */
      16, /* Number of queues */
      GASPI_QUEUE_GROUP_POLICY_CPU_RR);
```

9.7 Implementation Details

In this section, we explain the details of our extension in GPI-2. Our changes follow the development rules established by GPI-2 developers and their code style.

Before implementing our approach, some GASPI types and constants were declared. For instance, the types for GASPI modes (gaspi_mode_t), for queue groups (e.g. gaspi_queue_group_id_t) and the new timeout constant (GASPI_BLOCK_TASK).

The next section describes briefly the original implementation of GPI-2, which is the base for our extensions. The following sections explain how we have integrated the interoperability mechanism into the gaspi_wait and gaspi_notify_waitsome procedures, and also, we explain the implementation of queue groups.

9.7.1 Original Implementation

In this section, we briefly explain the original implementation of GPI-2. GPI-2 works directly on top of the Infiniband library interface, also called libverbs. This library allows to use Remote Direct Memory Access (RDMA) by creating queues and posting RDMA requests into these queues. Furthermore, GPI-2 can be used with a simple TCP mode, which defines a similar interface for posting requests.

These internal requests can be configured for writing, reading or notifying the remote process. In addition, each request has an id and a pointer to the next request. This pointer allows to create a linked list of requests in order to post them together into a specific queue. For instance, gaspi_write_notify operation translates to a write request and a special notification request at the remote side. These two requests are linked using the aforementioned pointer.

Once a request completes locally, the libverbs library adds it into a queue of completed requests, which corresponds to the source queue. Thus, each libverbs queue has its own completion queue. In addition, libverbs provides a routine to poll the completed requests from a specific completion queue. When a request is retrieved from the completion queue, it is possible to get its id. It is worth noting that only one thread can poll a completion queue at a time, so it must be protected with a mutex.

9.7.2 Extending gaspi_wait

The gaspi_wait function does not provide information about which requests have to be waited for. When calling to it and passing GASPI_BLOCK_TASK, the routine has to wait for all operations performed by the calling task. The only way to know which operations have performed a task is to take note of which is the calling task in routines as gaspi_write_notify, gaspi_read, etc.

Figure 9.5 shows the flow of the tasks when they call to the gaspi_write routine, and then, they call to gaspi_wait in order to wait the local completion of that operation. Figure 9.6 shows the pseudocode of the pause/resume mechanism in gaspi_wait.

Firstly, when a task calls to a routine like gaspi_write, the task handle or its id is retrieved by calling to the get_current_task API function (line 8). This is used to identify the different operations performed by the task. Then, the structure which contains information about the task's requests, also called info_t, is retrieved by calling to get_assigned_info. This internal function should access a structure indexed by the task handle (or id) and it should return a pointer to its information structure. If there is no entry for the calling task, it should create a new entry initializing the

pending_requests counter to 1. After returning the info structure, the pending_request counter should be increased with as many units as the number of generated requests (line 10). In this case, gaspi_write generates a single request. Finally, the libverbs request is identified with the pointer to the information structure (line 11), and it is posted into the corresponding queue (line 12).

Once the task calls the gaspi_wait routine passing GASPI_BLOCK_TASK, its info structure is retrieved again (line 19). The counter of pending requests is decremented (line 20), and if the resulting value is zero, it means that all its requests are completed. Otherwise, there are still some pending requests, so the task should block. In this case, it saves its blocking context into the information structure and it blocks using the block/unblock API (lines 21-22).

When initializing GPI-2 with GASPI_MODE_TASK, a polling service is registered to execute the poll_queues function periodically. This function polls all active completion queues (line 29). When a request finishes, it gets the pointer pointing to the corresponding info structure, which is saved in the request id (line 31). Then, the counter of pending requests is decreased. If it becomes 0, the task is unblocked using its blocking context (line 33), which was saved in its information structure.

Note that the original GPI-2 implementation did not use the request ids, since it waited for all requests without the need of identifying them. Also notice that all operations on the pending_requests counter must be atomic. In addition, this implementation needs the void* get_current_task() API function, which is present in almost all task-based programming models. This function should return a handle to the current task or its identifier.

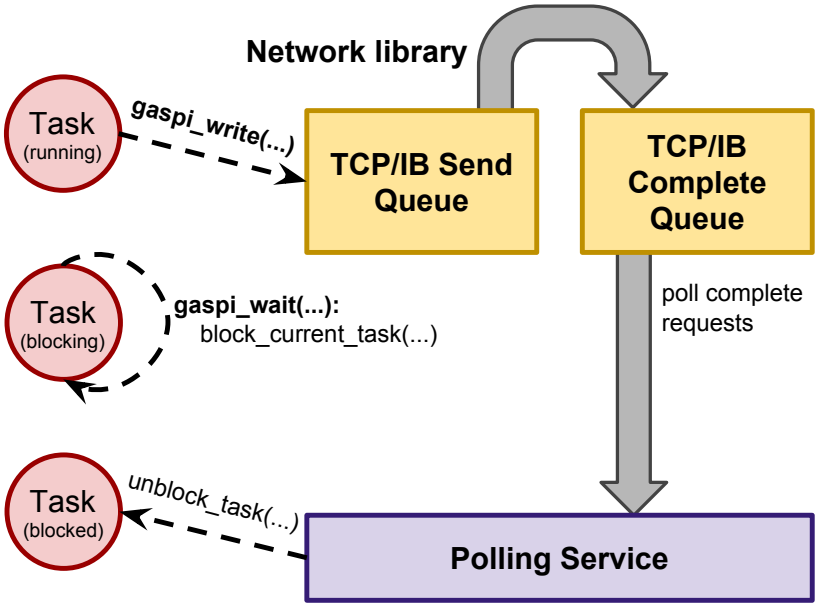


Fig. 9.5: Task flow when calling gaspi_wait with GASPI_BLOCK_TASK.

```
struct info_t {
   int pending_requests;
   void *blocking_ctx;
};

int gaspi_write(...) {
   ...
   void *task_handle = get_current_task();
   info_t *info = get_assigned_info(task_handle);
   ++info->pending_requests;
   request.id = info;
   queue->post(request);
   ...
}

int gaspi_wait(...) {
   ...
   void *task_handle = get_current_task();
   info_t *info = get_assigned_info(task_handle);
   if (--info->pending_requests > 0) {
      info->blocking_ctx = get_current_blocking_context();
      block_current_task(info->blocking_ctx);
   }
   ...
}

int poll_queues(...) {
   ...
   request = complete_queue->poll();
   if (request != NULL) {
      info_t *info = request->id;
      if (--info->pending_requests == 0) {
         unblock_task(info->blocking_ctx);
      }
   }
   ...
}
```

Fig. 9.6: Pseudocode of the pause/resume mechanism in gaspi_wait.

9.7.3 Extending gaspi_notify_waitsome

The gaspi_notify_waitsome function waits for at least one of the notifications from the specified range. This function is totally lock-free, therefore the extension for supporting the interoperability mechanism should maintain this property. This mechanism is enabled when passing GASPI_BLOCK_TASK as the timeout parameter.

Figure 9.7 shows the flow of tasks when they call to gaspi_notify_waitsome with the interoperability mode enabled. Figure 9.8 shows the pseudocode of the pause/resume mechanism in this function.

Firstly, there is a new internal type called waiting_range_t, which holds the required information about a particular task when waits for a range of notifications. It has the begin_id and the num_ids fields that define the range of notifications. It also has a field to save the first notified id once the waiting has finished. Finally, blocking_ctx stores the blocking context of the calling task, so it can be unblocked later.

The gaspi_notify_waitsome function creates a waiting range with the parameters passed to that function (line 10). This range is immediately tested by calling to test_waiting_range (line 11), which checks if any of the notifications has arrived. This function returns the first notified id in its second argument, if any of them has been notified. In case no notification has arrived, the task should block. The task saves its blocking context in the waiting range (line 13), it enqueues the range to a lock-free queue (line 14), and finally, it blocks through the block/unblock API (line 15). Once the task is unblocked, it returns the notified id through the first_id parameter.

In turn, there is a polling service devoted to checking all active waiting ranges, which calls to the poll_waiting_ranges function periodically. The first thing it does is to dequeue all waiting ranges from the lock-free queue and add them into a regular

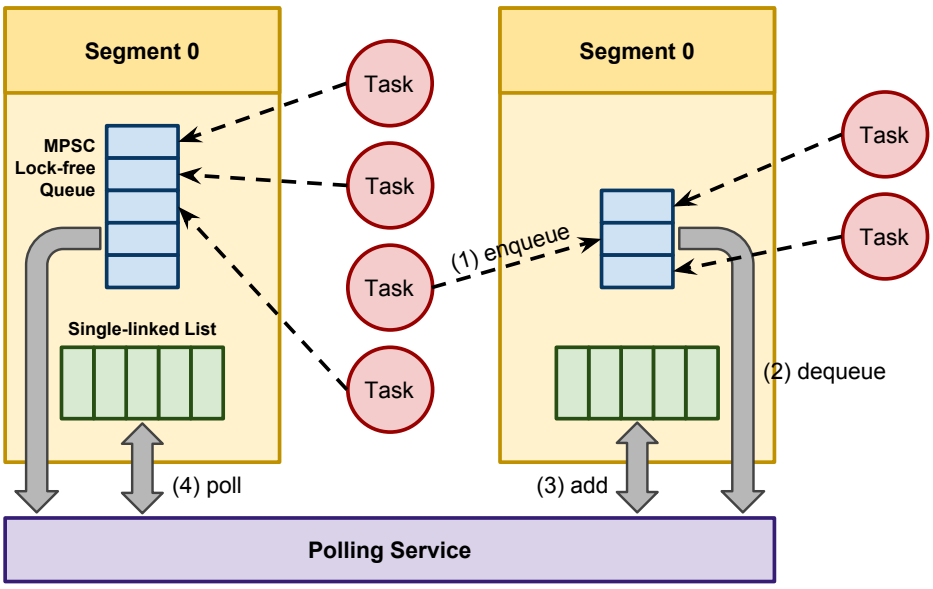


Fig. 9.7: Task flow when calling gaspi_notify_waitsome with GASPI_BLOCK_TASK.

single-linked list (line 23). The lock-free queue is a multiple producer (communication tasks) and a single consumer (polling service) queue. With this, we get the gaspi_notify_waitsome function to be lock-free, allowing multiple tasks to call to it without generating too much contention. Note that the polling function is protected through a mutex, so the list can be modified from there without any risk. Then, the polling thread goes through all waiting ranges testing their notifications. In case a notification has arrived, it removes the range from the list and unblocks the corresponding task (lines 27-28).

Note that notifications belong to a specific segment. Therefore, we define a different queue and list for each segment, basically to reduce the overhead and contention.

9.7.4 Implementing Queue Groups

The implementation of queue groups follows the GPI-2's implementation of queues, segments, etc. The maximum number of allowed queue groups is limited by the implementation itself, and in this case, it is set to 16. Since they are very limited, a static array can hold the information about all possible queue groups, indexed by the queue group id.

On the one hand, the GASPI_QUEUE_GROUP_POLICY_DEFAULT distribution policy does not need to compute anything when a queue group is created with this policy. It provides different queues each time the getter function is called. Thus, it only needs an integer which is increased each time it is called, and takes the values of the queue ids from the range. Operations over the integer are atomic (but relaxed), since the getter function can be called from multiple threads.

On the other hand, the GASPI_QUEUE_GROUP_POLICY_CPU_RR distribution policy needs to compute the actual distribution of queues across CPUs when creating a queue group. This policy needs a static array with as many integers as CPUs. Once configured, tasks can call the getter function which will return the assigned queue to the current CPU.

```
struct waiting_range_t {
   gaspi_notification_id_t begin_id;
   gaspi_number_t num_ids;
   gaspi_notification_id_t notified_id;
   void *blocking_ctx;
};

int gaspi_notify_waitsome(..., begin_id, num_ids, first_id, ...) {
   ...
   waiting_range_t range = {begin_id, num_ids, NULL, NULL};
   notified = test_waiting_range(&range, &range.notified_id);
   if (!notified) {
      range.blocking_ctx = get_current_blocking_context();
      range_queue->enqueue(&range);
      block_current_task(range.blocking_ctx);
   }
   *first_id = range.notified_id;
   ...
}

int poll_waiting_ranges(...) {
   ...
   range_list->dequeue_and_add(range_queue);
   for (waiting_range_t *range : range_list) {
      notified = test_waiting_range(range, &range->notified_id);
      if (notified) {
         range_list->remove(range);
         unblock_task(range->blocking_ctx);
      }
   }
   ...
}
```

Fig. 9.8: Pseudocode of the pause/resume mechanism in gaspi_notify_waitsome.

10 | Proposing a Non-blocking Interoperability Mechanism

The interoperability mechanism presented previously blocks the current task when calling to a blocking MPI/GASPI procedure and returns from that procedure once the operation completes. However, there are some cases in which the task that performs operations does not need to wait for the completion of these operations, since it does not access the sent/received data. For instance, a task could receive data by calling to MPI_Recv, while another task could consume the received data by declaring the proper dependencies between both tasks. In this case, the receiving task could avoid blocking, thus avoiding the overhead of performing a block/unblock cycle.

For this reason, we propose a new interoperability mechanism which avoids the blocking of tasks. The new approach proposes two new MPI-like non-blocking functions called MPI_Iwait and MPI_Iwaitall, which have the same parameters as the original MPI_Wait and MPI_Waitall, respectively. They bind the pending requests passed as parameters to the calling task, preventing the release of its dependencies until all these requests complete. It is worth noting that the task can finish its execution, although the requests are not completed. Also note that the data cannot be reused/consumed after calling to these new functions.

Figure 10.1 shows an example of this new interoperability approach. The first task performs two non-blocking receive and send, and it declares the dependencies on the destination/source buffers. Then, it binds the generated requests to itself, so its dependencies will be released once both requests complete. Finally, the task can finish its execution without blocking, becoming a *zombie* task until the dependencies can be released.

Once both requests complete, the dependencies are released and the next two tasks become ready. In that moment, the received data can be consumed and the sent data can be modified again safely.

Note that statuses are also saved at the address provided at the second and third parameter of MPI_Iwait and MPI_Iwaitall, respectively. In this way, the statuses can be checked later (e.g. to know the source rank or the number of received elements). The statuses will not be saved when passing MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE, respectively.

```
MPI_Status statuses[2];
...
#pragma oss out(dataA) in(dataB)
{
   MPI_Request requests[2];
   MPI_Irecv(&dataA, ..., &requests[0]);
   MPI_Isend(&dataB, ..., &requests[1]);
   MPI_Iwaitall(2, &requests, &statuses);
}

#pragma oss in(dataA)
processDataAndStatus(dataA, statuses[0]);

#pragma oss out(dataB)
resetData(dataB);
```

Fig. 10.1: Example of the non-blocking interoperability mechanism.

This mechanism is enabled when initializing MPI with the MPI_TASK_MULTIPLE threading level. Calls to the MPI_Iwait and MPI_Iwaitall functions when the interoperability is not enabled will result in the behavior of the original MPI_Wait and MPI_Waitall functions, respectively. In addition, it is worth noting that that both blocking and non-blocking interoperability mechanism can coexist without any problem, since the first is applied to blocking procedures and the second is only applied to those two new functions.

We are currently working on this new approach, which have been implemented but not evaluated yet. Therefore, this is just a proposal, which we will evaluate properly in the near future. In the next sections we explain the new API proposed for the interaction between the involved programming models and some details of the changes needed in the interoperability library.

In addition, this new non-blocking mechanism could be integrated into the GASPI programming model. In this case, we could define two new functions similar to the gaspi_wait and gaspi_notify_waitsome functions. The first could register all GASPI requests performed by the calling task, so its dependencies could be released once all requests complete. In contrast, the second could bind the range of notifications to the calling task, so its dependencies could be released once one of the notifications from the range arrives.

10.1 Task Event Counter API

In this section we propose a new API to allow the non-blocking interoperability mechanism, which is called the task event counter API. Figure 10.2 shows the complete API, which has to be implemented by the task-based programming model. As can be

```
// Get the event counter of the current task
void *get_current_event_counter();

// Increase the event counter of the current task
// to prevent the release of its dependencies
void increase_current_task_event_counter(void *event_counter,
     unsigned int increment);

// Decrease the event counter of a task and release
// its dependencies if the counter becomes zero and
// the task has finished its execution
void decrease_task_event_counter(void *event_counter,
     unsigned int decrement);
```

Fig. 10.2: Task event counter API.

seen, its format is very similar to the task block/unblock API. The main idea is that each task has an integer counter, which counts the number of pending external events.

The get_current_event_counter function provides a handle to the event counter of the current task. Then, new events can be registered by calling to the increase_current_task_event_counter function, which increases the event counter of the current task as many units as increment. Notice that only the task itself can increase the number of events, since otherwise, we could increase the counter of finished tasks.

The decrease_task_event_counter decreases the corresponding event counter, which can be the counter of any task, even the counter of the current task. Once the event counter becomes zero and the task finishes its execution, the dependencies of that task are released. Therefore, if a task finishes, but it has pending events to be fulfilled, the task cannot release its dependencies and it becomes *zombie*. In this case, the dependencies will be released when someone calls to decrease_task_event_counter and the number of pending events becomes zero.

Note that this API follows the format of task block/unblock API and it can be also used in the interoperability library along with the polling service API. Notice also that the user of this API is responsible for increasing and decreasing the same number of events of each task. In case the number of unregistered events is greater than the registered ones, the behavior is undefined. In contrast, if the number of unregistered events is smaller than the registered ones, the dependencies of the task will not be released.

10.2 Implementation Details

This section covers the implementation details of the new task event counter API and the integration of the new non-blocking interoperability mechanism into the updated

interoperability library. On the one hand, we have implemented this API in the Nanos6 runtime system by initializing the counter of each task to one. The dependencies of a task cannot be released while executing, since the counter is at least one, independently of the number of pending events. Once the task finishes its execution, the counter is decremented in one unit. In case this results in a value of zero, the dependencies can be released. Otherwise, it means that there are still some pending events, and the dependencies will be released when calling to decrease_task_event_counter and the event counter resulting in zero. Note that this is just an implementation and it is totally transparent to the user.

On the other hand, we have integrated the new non-blocking interoperability mechanism into the improved interoperability library, which has been presented in Chapter 8. Figure 10.3 shows the pseudocode of the implementation of both MPI_Iwait and the polling function in the interoperability library. As can be seen, the implementation is very similar to the one presented in Chapter 8.

```
1  int MPI_Iwait(MPI_Request request, MPI_Status *status) {
2     int err, completed = 0;
3     if (Interop::isEnabled()) {
4        MPI_Test(&request, &completed, status);
5        if (!completed) {
6           Ticket ticket = new Ticket(request, status);
7           ticket->_event_cnt = get_current_event_counter();
8           increase_current_task_event_counter(ticket->_event_cnt, 1);
9           _pendingTickets.add(ticket);
10       }
11       return err;
12    }
13    return MPI_Wait(request, status);
14 }
15
16 void Interop::poll() {
17    for (Ticket *ticket : _pendingTickets) {
18       int completed = 0;
19       MPI_Test(ticket->_request, &completed, ticket->_status);
20       if (completed) {
21          _pendingTickets.remove(ticket);
22          decrease_task_event_counter(ticket->_event_cnt, 1);
23          delete ticket;
24       }
25    }
26 }
```

Fig. 10.3: Implementation of the MPI_Iwait routine and the polling function in the interoperability library.

In MPI_Iwait, the ticket cannot reside on the stack, since the function returns *immediately* and the stack will not be valid. For this reason, a ticket is dynamically created at line 6. Then, the event counter of the current task is saved in the ticket (line 7) and the number of pending events is increased (line 8).

In the polling function, each time a request completes, the number of events of the corresponding task must be decreased. When all events of a task are fulfilled, the runtime system releases its dependencies. In this case, the ticket must be deleted since it was created dynamically.

The implementation of MPI_Iwaitall is very similar to the one shown in that figure. In our implementation, both blocking and non-blocking interoperability mechanisms coexist at the same time. Each ticket needs a boolean field to distinguish between the tickets of the blocking mechanism and the tickets of the non-blocking mechanism.

11 | Evaluation

This chapter covers the evaluation of our proposals for MPI and GASPI applications. Firstly, we explain the environment in which we have performed all our evaluation experiments. Secondly, we provide an in-depth analysis of the two applications which we have used to evaluate our approaches. These are an iterative Gauss–Seidel method and a mock-up of a meteorological forecasting application called IFSker. There is a section dedicated to each of these two applications. Finally, we show and discuss the performance results obtained with our approaches.

Notice that this chapter is also part of the Design Cycle of the Design Science Research methodology used in this project, which is described in Section 7.2.

11.1 Environment

The experiments were carried out on the Marenostrum4 and Nord3 supercomputers from the Barcelona Supercomputing Center (BSC). On the one hand, each compute node in Marenostrum4 is equipped with 2 sockets of Intel Xeon Platinum 8160 CPUs, with 24 cores each, totaling 48 cores per node and 96 GB of main memory (2 GB per core). The interconnection network is based on 100 Gbit/s Intel Omni-Path HFI technology.

On the other hand, each compute node in Nord3, which is a sub-part of the old Marenostrum3, is equipped with 2 sockets of Intel E5–2670 SandyBridge-EP CPUs, with 8 cores each, totaling 16 cores per node and 128 GB of main memory (8 GB per core). The interconnection network is based on Infiniband Mellanox FDR10 technology.

The MPI interoperability proposal has been evaluated in Marenostrum4 and Nord3. However, the proposed extensions for GASPI have only been evaluated in Nord3, since GPI-2 does not support Omni-Path technology yet. It only supports the common TCP mode and the Infiniband technology, which is present in Nord3 supercomputer. We could evaluate our approach in Marenostrum4 using the TCP mode, however this could deteriorate significantly the performance of the applications.

In addition, Table 11.1 shows the software and the versions used for our experiments.

Table 11.1: Table describing the software used to perform the experiments and their versions.

Software	Version
OmpSs-2	17.1
GNU C/Fortran Compilers	4.9.4
MPICH	3.2.1
Intel MPI	2017.1
GPI-2	1.3.0

11.2 Gauss–Seidel

In this section we explain the iterative Gauss–Seidel method [23] to solve the Heat equation [24], which is a parabolic partial differential equation that describes the distribution of heat in a given region over time. We developed this application from scratch and it features five different version of the Gauss–Seidel method for 2-D matrices. Two of these five versions are parallelized with MPI only, while the rest are hybrid MPI+OmpSs-2 implementations. We first developed these versions with MPI, and then, we *translated* them into GASPI variants, which follow the same parallelization strategy but using RDMA operations. These GASPI variants are described later. The next two are the MPI-based versions:

- *Pure MPI*: This version is a straightforward implementation of the algorithm using synchronous MPI primitives to exchange boundaries among neighboring ranks. The computation phase of the algorithm is sequential. The 2-D matrix is distributed across ranks assigning a consecutive set of rows to each one (a single block per rank). Boundary exchanges correspond to whole rows.
- *N-Buffer MPI*: This version is significantly more elaborate than *Pure MPI*. In this case the rows of each rank are horizontally divided by blocks, hence a distinct boundary exchange is performed for each block. This version starts to exchange block boundaries as soon as possible using asynchronous MPI primitives. For instance, a rank starts to send (MPI_Isend) its last row of a block once it has been computed, but also starts to receive (MPI_Irecv) the lower boundary for the next iteration. Before starting the computation of a block, it waits (MPI_Wait) for the completion of all pending MPI requests related to the block. Thus, the computation is partially overlapped by boundary exchanges.

The rest are hybrid MPI+OmpSs-2 versions which divide the matrix into squared blocks and these are distributed across MPI ranks. The left-hand side of Figure 11.1 shows how a domain of 3×12 blocks would be split across four MPI ranks. These hybrid versions are:

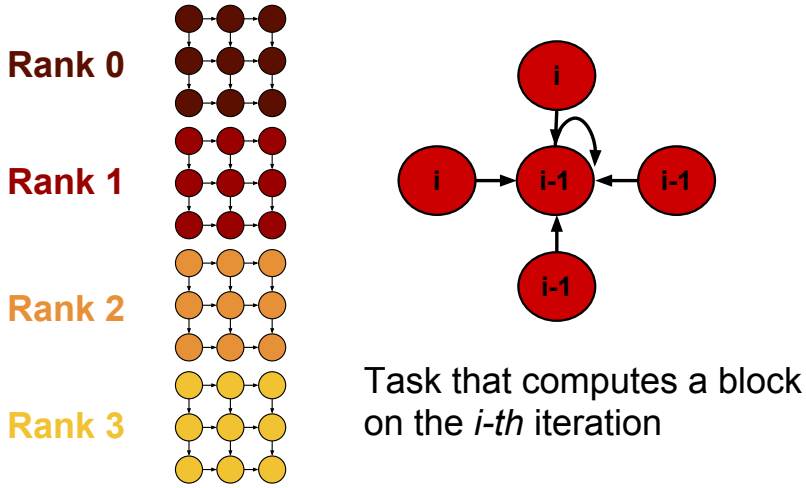


Fig. 11.1: 2-D matrix of 3×12 blocks split in four ranks. On the hybrid versions, for each iteration a task is created to update each block using values of both current (top and left blocks) and previous (current, right and bottom blocks) iterations.

- *Fork-Join*: This is a hybrid version with a sequential communication phase and a parallel computation phase. The communication phase uses synchronous primitives to exchange boundaries among neighbors as in *Pure MPI*. On the computation phase, a task is created to update each block using the top and left blocks of the current iteration, and the current, left and bottom blocks of the previous iteration, as shown in Figure 11.1. Tasks use fine-grained dependencies to exploit the spatial wave-front parallelism. However, there is a global synchronization point after each computation phase that prevents this version from exploiting parallelism across iterations (temporal wave-front).

- *Sentinel*: A hybrid version where both communication and computation are implemented using tasks. The communication phase uses tasks to execute the synchronous MPI primitives that exchange boundary blocks among neighbors. These communication tasks are serialized by a sentinel dependency to avoid deadlocks, as explained in Section 6.1. This version avoids the global synchronization (*taskwait*) by leveraging fine-grained dependencies between computation and communication tasks.

- *Interop*: This version uses the MPI_TASK_MULTIPLE multi-threading level proposed in this project to avoid the serialization of communication tasks. This is the only difference with the *Sentinel* version. This version has been evaluated with the enhanced interoperability library presented in Chapter 8.

In *Pure MPI* each process holds a single block of *total_rows/num_procs* rows and *total_cols* columns, while in *N-Buffer MPI* each block has *total_rows/num_procs* rows and 1K columns. In the hybrid versions, each compute task processes a block of 1K×1K elements. This is the smallest block size required to attain peak performance.

The complete code for both the *Sentinel* and *Interop* versions is shown in Appendix A.1.

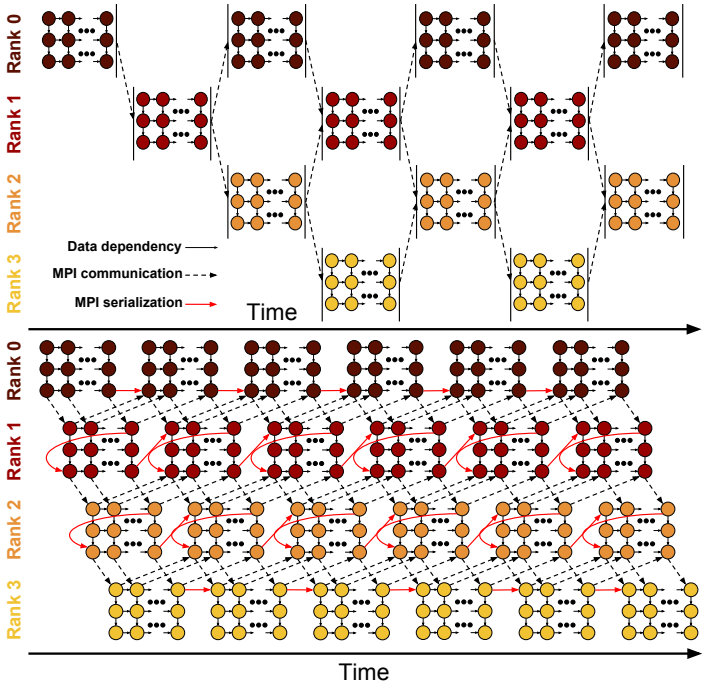


Fig. 11.2: Above: dependency graph for *Pure* and *Fork-Join*. Below: dependency graph for *N-Buffer*, *Sentinel* (with red deps) and *Interop* (no red deps).

As can be seen, both computation and communication phases are totally taskified using fine-grained dependencies between tasks. The *Sentinel* version corresponds to the application initializing the MPI library with MPI_THREAD_MULTIPLE. Then the artificial dependencies on communication tasks (i.e. inout dependency on sentinel) are enforced, allowing only a single communication task to be executed at a time. In contrast, the *Interop* version corresponds to the MPI library being initialized with MPI_TASK_MULTIPLE. In that case, the artificial dependencies on communication tasks are ignored, allowing communication tasks to run in parallel without any risk.

The rest of versions are omitted since they can be inferred from the taskified code. Firstly, the *Pure MPI* version uses the same code but ignoring the OmpSs directives, and setting the BSX constant to *total_rows/num_procs* and BSY to *total_cols*. Secondly, the *Fork-Join* version can be obtained by removing the OmpSs directives from the communication phases and adding a taskwait directive at the end of the computation phase. Finally, the N-Buffer MPI can obtained by changing several parts of the code and increasing the complexity of the algorithm, as explained previously.

Figure 11.2 compares the dependency graph of the *Pure MPI* and *Fork-Join* versions (above) and *N-Buffer MPI*, *Sentinel*, *Interop* versions (below). For the sake of clarity, both graphs have been simplified by showing up to the first six iterations, fusing the explicit communication tasks with the tasks that compute boundary blocks and also other redundant dependencies such as anti-dependencies. In the *Pure MPI* and *Fork-Join* versions, the execution of each iteration inside an MPI rank depends on the completion of the previous iteration of their neighbor MPI ranks, which results in a strong serialization effect that affects the execution of the whole program.

In the *N-Buffer MPI* version, the strong serialization effect can be avoided by exchanging block boundaries as soon as possible, performing calls to the corresponding asynchronous MPI primitives right after processing each block. The *Sentinel* version also exchanges block boundaries at the earliest, using tasks with fine-grained dependencies to execute MPI primitives. Nevertheless, given that this version uses synchronous primitives, it still has to serialize the communication tasks to avoid deadlocks. This introduces the red dependencies that also reduce significantly the parallelism within and across iterations.

Finally, the *Interop* version that uses MPI_TASK_MULTIPLE removes the red dependencies and thus can fully exploit both spatial and temporal wave-front parallelism. Moreover, in this version, tasks blocked on MPI calls never block the underlying hardware thread, so resource under-subscription is also avoided. In summary, MPI_TASK_MULTIPLE allows the programmer to parallelize applications in a more natural way, without requiring artificial dependencies that hinder the available parallelism.

11.2.1 GASPI Versions

After developing the Gauss–Seidel application for MPI, we developed the same versions for the GASPI programming model. In this case, the complete code of the GASPI *Sentinel* and *Interop* versions is shown in Appendix A.2. The *Sentinel* version corresponds to the GASPI library being initialized with GASPI_MODE_STANDARD. In contrast, the *Interop* version corresponds to the MPI library being initialized with GASPI_MODE_TASK. The enforcement of the artificial dependencies on sentinel works exactly the same as in MPI.

As can be seen, each sending task gets a specific queue which will be used for posting all its GASPI requests. Next, it calls to the gaspi_write_notify function, which starts a write operation of the boundary to its corresponding neighbor process. Furthermore, the neighbor process will be notified once the data is actually written. Finally, the task waits for the local completion of its operations to ensure that the source buffer can be modified again safely, before finishing and releasing its dependencies.

In contrast, the receiving tasks just wait for the notification sent by the sender process. When the notification arrives, the destination buffer can be safely consumed by the computation tasks.

Notice that the parallelization strategy is almost the same as in MPI. The rest of versions can be also inferred from this code. In the *Pure* and *Fork-Join* versions, some gaspi_wait calls can be omitted, since the order of transmissions is always the same and some assumptions can be made. For instance, if a process A writes and notifies a remote process B, and then, a notification of a dependent operation arrives at the process A from the remote process B, it can be assumed that the first operation is completed.

Finally, N-Buffer GASPI is a more advanced version which starts to send and receive boundaries as soon as possible. Notice that the idea is the same as in the N-Buffer MPI. However the implementation is still more complex.

```
1  void receiveUpperBorder(block_t *matrix, int nbx, int nby, int rank) {
2     for (int by = 1; by < nby-1; ++by) {
3        #pragma oss task out(([nbx][nby]matrix)[0][by]) inout(*sentinel)
4        {
5           MPI_Request request;
6           MPI_Irecv(&matrix[by][BSX-1], BSY, MPI_DOUBLE, rank - 1,
7              by, MPI_COMM_WORLD, &request);
8           MPI_Iwait(&request, MPI_STATUS_IGNORE);
9        }
10    }
11 }
```

Fig. 11.3: Modified receiveUpperBorder to use the non-blocking interoperability mechanism.

11.2.2 Non-blocking Interoperability Variant

Although we have not evaluated the proposed non-blocking interoperability mechanism yet, in this section we explain the few changes that the Gauss–Seidel application needs in order to work with this mechanism. The code presented in Appendix A.1 just needs some modifications in the communication tasks. Firstly, the blocking MPI calls, MPI_Send and MPI_Recv, should be replaced by their non-blocking counterparts, MPI_Isend and MPI_Irecv, respectively. Secondly, the generated MPI request should then be passed to MPI_Iwait, which will prevent the release of the dependencies until the request completes. For instance, Figure 11.3 shows the modified code of the receiveUpperBorder function.

11.3 IFSker

IFSker is a mock-up application written in Fortran and parallelized with MPI. It mimics the communication and computational patterns of the Integrated Forecasting System (IFS), which is a meteorological forecasting model developed and maintained by the European Centre For Medium-Range Weather Forecasts (ECMWF). IFS employs a spectral transform method which represents fields by using a set of coefficients of a basis function (e.g. a sine function). This mock-up application was evaluated also in [9]. We have adapted it to the OmpSs-2 programming model and we have reevaluated it.

The algorithmic structure consists in timestep cycles divided into two phases: Grid-point Physics computations (GP) and Fast Fourier Transforms (FFT). Each MPI process holds a specific set of Earth's latitudes and both phases perform computations on these latitudes. Nevertheless, data representation and distribution among MPI processes is different in each phase. Therefore, the communication of latitudes between processes takes place during the transitions from one phase to the next one, where the data needs to be transposed and redistributed among the processes.

Each timestep in this mock-up application performs:

1. Grid-point physics computation phase.
2. Transition from GP to FFT. Each process should send all latitudes from GP that are not needed by the FFT phase to the corresponding processes, and it should receive all latitudes needed by FFT that were not present in the GP phase.
3. Fast Fourier Transform phase.
4. Transition from FFT to GP. Each process should send all latitudes from FFT that are not needed by the GP phase to the corresponding processes, and it should receive all latitudes needed by GP that were not present in the FFT phase.

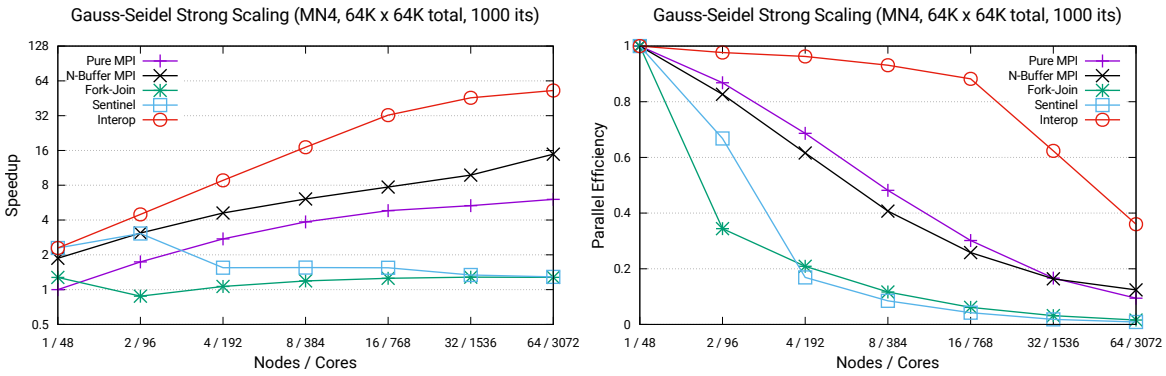
This means that processes should know the assigned process of each latitude in each computation phase.

The original implementation is based on MPI (*Pure MPI*), but we have implemented a new version (*Interop*) that uses tasks for both the compute and communication phases. However, in this application the compute phase is very fine-grained so it is not worth to fully parallelize it. Hence, we are only using tasks to have more in-fly MPI operations and to overlap communication and computation phases. In this evaluation there is one MPI rank per core for both the *Pure MPI* and *Interop* versions, so the *Fork-Join* and *Sentinel* versions used on the Gauss-Seidel version will be equivalent to the *Pure MPI* version.

With this application we want to demonstrate that even using a single thread, the interoperability mechanism still improves the performance of the application, since it allows the out-of-order execution of communication tasks. Notice that this application has only been implemented for MPI.

11.4 Evaluation and Discussion

In this section, we show the evaluation and we discuss the performance obtained with the applications explained previously. Firstly, we discuss the performance results of the Gauss–Seidel benchmark in Marenostrum4. This only evaluates the MPI variants, since GPI-2 does not support the Intel Omni-path network infrastructure. Secondly, we show the evaluation of Gauss–Seidel in Nord3, which in this case we evaluate both MPI and GASPI variants. We seize this moment to compare the performance between the MPI and GASPI variants. Finally, we evaluate the IFSker mock-up application in Marenostrum4.



(a) Speedup graph.

(b) Parallel efficiency graph.

Fig. 11.4: Gauss–Seidel strong scaling in Marenosturm4 with 64K x 64K total elements and 1000 timesteps.

11.4.1 Gauss–Seidel in Marenostrum4

This section covers the evaluation of the MPI variants of the Gauss–Seidel application. *Pure MPI* and *N-Buffer MPI* experiments have been performed using 48 MPI ranks per node, while hybrid versions have used 1 rank per node and 48 OmpSs threads per rank. All experiments in Marenostrum4 have been performed using MPICH 3.2.1.

Figure 11.4a shows a strong-scaling experiment of the five MPI versions using the performance of the *Pure MPI* version running on one node as a baseline. On a single node, all the hybrid versions experience higher performance than the *Pure MPI* version. When the hybrid versions run on a single node (one rank), the MPI primitives are completely avoided. Thus, the rigid serialization effect introduced by MPI is fully removed and these versions can fully exploit the spatial and temporal wave-front parallelism.

It is worth noting that the *Fork-Join* version is significantly slower than the other task-based versions due to the global synchronization point after each iteration that prevents the exploitation of the temporal wave-front. As we increase the number of nodes, the performance of the *Pure MPI* version also increases, but the scalability is clearly sub-optimal. On the other hand, both *Fork-join* and *Sentinel* stop scaling at two and four nodes, respectively. In addition, these versions are the only that can be easily implemented with current OpenMP and MPI standards.

The *N-Buffer MPI* version outperforms all previous versions since it avoids the strong serialization of iterations between ranks, which is observed in *Pure MPI* and *Fork-Join*. In addition, this version allows to overlap computation and communication phases. However, the scalability is still sub-optimal and it increases the complexity of the code.

The *Interop* version has good scalability with up to 32 nodes. With 64 nodes the curve flattens because the problem size is too small to get sufficient parallelism to exploit 48 cores.

Figure 11.4b shows the parallel efficiency of all five versions. In this case each version uses as a baseline its own performance on a single node. From 1 to 16 nodes the efficiency of *Interop* is almost the same, but then it quickly decreases, since the problem size becomes too small to feed all the cores. The parallel efficiency of *Pure MPI* and *N-Buffer MPI* steadily decrease from 1 to 0.1 at 64 nodes. *Fork-Join* and *Sentinel* have a big drop of parallel efficiency at two and four nodes, respectively.

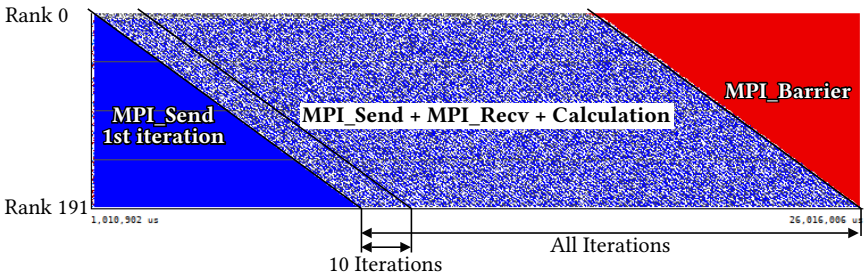
Figure 11.5 shows five traces of *Pure MPI*, *N-Buffer*, *Fork-Join*, *Sentinel* and *Interop*, respectively, running on four nodes (192 cores) with the same time-scale, and with a 32K x 32K matrix. The traces show the time-line on the X axis and the MPI ranks/OmpSs threads on the Y axis. In *Pure MPI* and *N-Buffer* there are 192 ranks, while in the hybrid versions there are four ranks (i.e. one rank per node) and each rank has 48 OmpSs threads. On the three hybrid versions, the red lines correspond to the execution of the Gauss–Seidel tasks.

In the *Pure MPI* version (Figure 11.5a), the last rank (191) cannot start computing the first iteration until all the other ranks have completed the first iteration. This introduces a big delay at the beginning that is also symmetrically reproduced at the end.

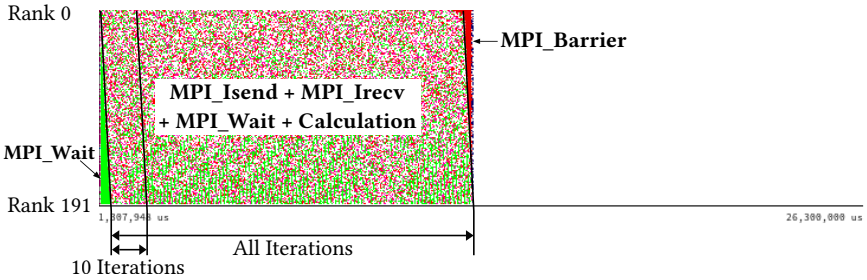
The same effect can be observed in the *Fork-Join* (Figure 11.5c) and *Sentinel* (Figure 11.5d) versions, but in this case there are only four ranks, so only four full iterations are required to have all the MPI ranks working. In the *Fork-Join* version, the global synchronization at the end of each iteration produces a strong serialization effect among iterations (that is the same effect found on the *Pure MPI* version), so one iteration cannot start until the same iteration of the previous MPI rank has been fully completed. Moreover, the global synchronization at the end of the compute phase also limits the available parallelism, so only 8 out of the 48 cores can work in parallel (running computation tasks).

The *Sentinel* version improves over the *Fork-Join* version because one MPI rank can start computing an iteration as soon as the previous rank has completed the computation of the first boundary block of the same iteration. This allows to partially overlap the computation of the same iteration across MPI ranks. However, the artificial dependencies introduced to serialize the communication tasks still hinder the available parallelism inside one iteration. In this case, 8 cores can run computation tasks in parallel with another core running a communication task. Although these hybrid versions take less time to complete a single iteration, *Pure MPI* pipelines iterations in a better way and ends up outperforming them in overall iteration throughput.

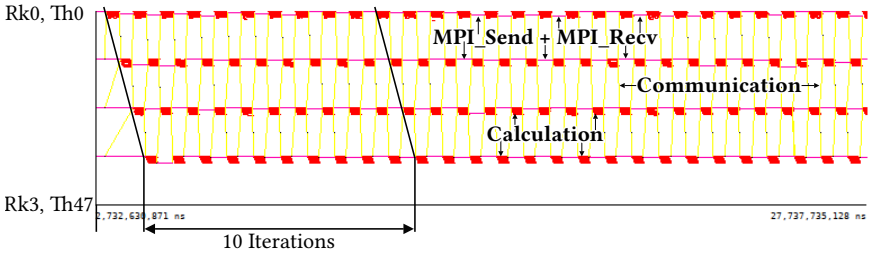
N-Buffer MPI (Figure 11.5b) does not show the big delay at the first iteration seen in *Pure MPI* and *Fork-Join*. This is because it exchanges boundaries as soon as possible, thus ranks can process different blocks from the same iteration concurrently. In addition, it is more flexible than the previous ones due to the use of asynchronous MPI primitives. The aforementioned reasons make this version outperform previous versions both in iteration latency and overall iteration throughput. However, it does not reach *Interop*'s performance and it requires more development effort than them.



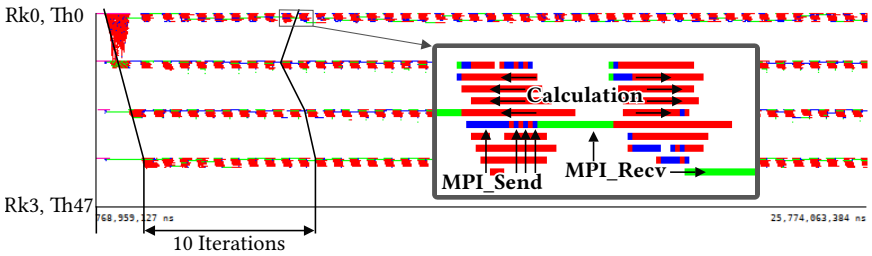
(a) Pure MPI.



(b) N-Buffer MPI.



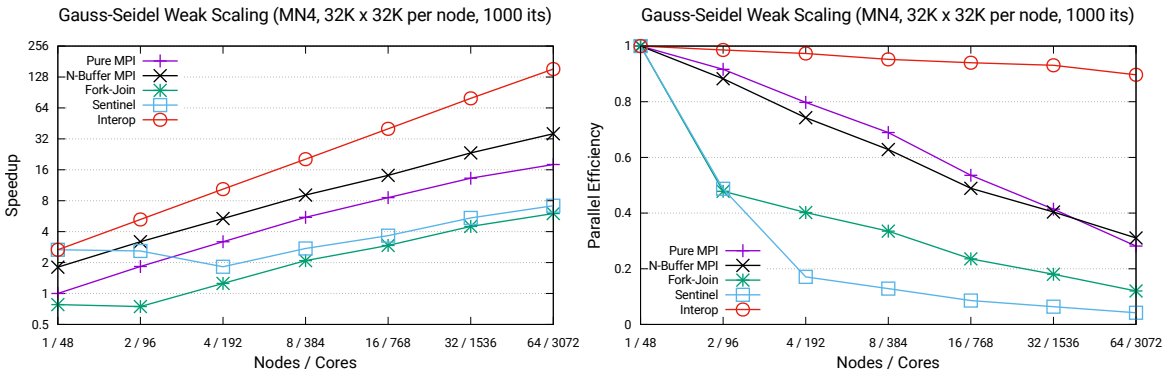
(c) Fork-Join.



(d) Sentinel.

(e) Interop.

Fig. 11.5: Execution traces with 4 nodes of Marenostrum4 with a 32K x 32K matrix. The Y axis shows MPI ranks/OmpSs threads, while the X axis shows the time-line.



(a) Speedup graph.

(b) Parallel efficiency graph.

Fig. 11.6: Gauss–Seidel weak scaling in Marenosturm4 with 32K x 32K elements per node and 1000 timesteps.

Finally, the *Interop* (Figure 11.5e) version avoids any global synchronization or serialization of communication tasks, so an iteration can be almost fully overlapped across the ranks. Moreover, this version is the only that can exploit both spatial wave-front and temporal wave-front parallelisms, benefiting from the 48 cores.

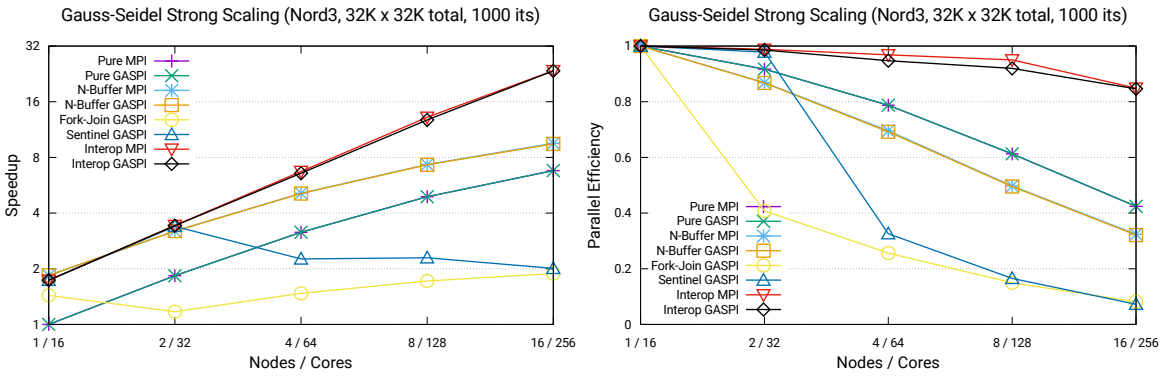
To finalize the performance analysis of Gauss–Seidel in Marenostrum4, we have performed a weak-scaling experiment. The speedup graph, shown in Figure 11.6a, uses the performance of the *Pure MPI* version on a single node as a baseline for all versions. For the parallel efficiency graph (Figure 11.6b), each version uses its own performance on one node as the baseline. This experiment shows again the good scalability of the *Interop* version which scales linearly up to 64 nodes. The parallel efficiency of *Pure MPI* and *N-Buffer MPI* steadily decrease from 1 to 0.3 at 64 nodes, while the *Fork-Join* and *Sentinel* versions feature a parallel efficiency of 0.4 and 0.2, respectively, with only four nodes.

11.4.2 Gauss–Seidel in Nord3

This section covers the evaluation of the Gauss–Seidel application in Nord3. We have evaluated both GASPI and MPI versions. However, we have only evaluated the most interesting MPI versions, which are Pure MPI, N-Buffer MPI and Interop MPI. These three programs have been executed with Intel MPI 2017.1.

In this case, the Pure MPI, N-Buffer MPI, Pure GASPI and N-Buffer GASPI have been executed with 16 ranks per node, while the hybrid versions have been executed with one rank per node and 16 OmpSs threads per rank.

Figure 11.7 shows both the speedup and parallel efficiency from the strong scaling experiment using a matrix of 32K x 32K elements and 1000 timesteps. Firstly, the difference in performance between MPI and GASPI is almost imperceptible. For instance, Pure MPI and Pure GASPI have the same performance in almost all cases. The same effect is seen in the N-Buffer versions.



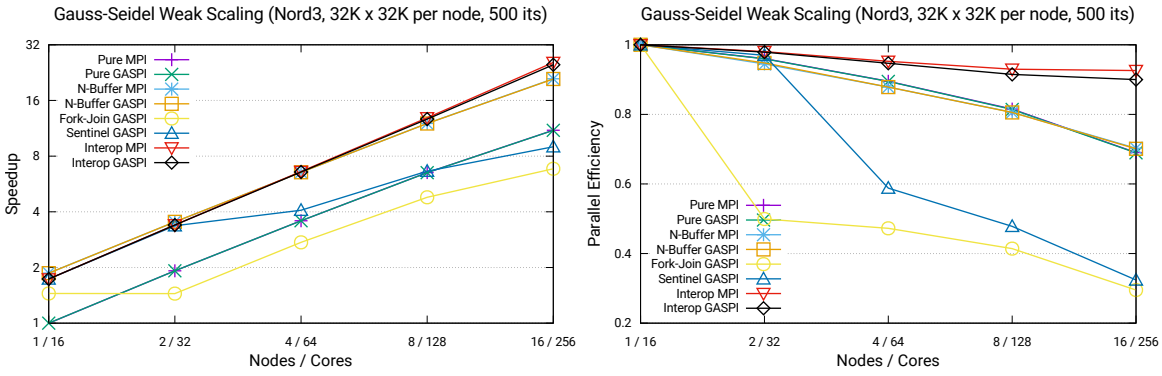
(a) Speedup graph. (b) Parallel efficiency graph.

Fig. 11.7: Gauss–Seidel strong scaling in Nord3 with 32K x 32K total elements and 1000 timesteps.

In this case, the difference between Pure and N-Buffer is still present, since the N-Buffer versions outperform both Pure versions. The parallel efficiency of Pure and N-Buffer steadily decreases from 1 to 0.35 and 0.4, respectively. In addition, the Fork-Join GASPI and Sentinel GASPI reproduces exactly the performance behavior from their MPI equivalents in Marenostrum4.

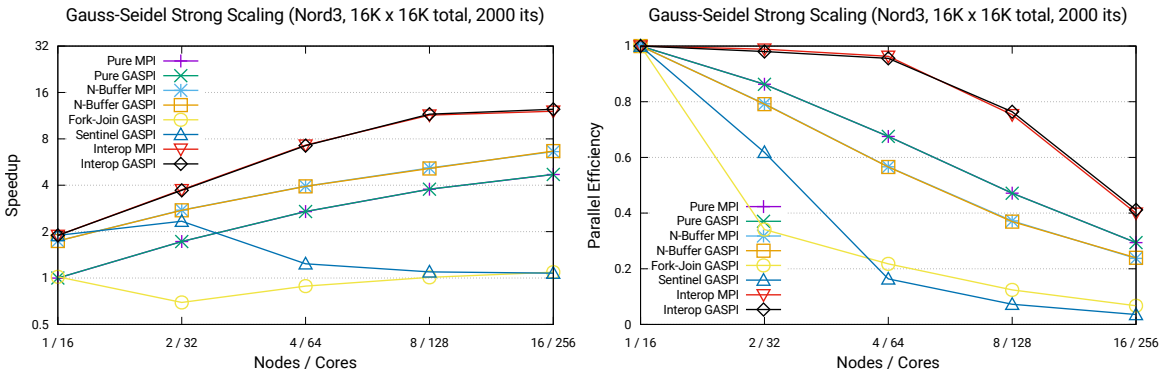
Both Interop versions scale linearly up to 8 nodes, thus maintaining a parallel efficiency greater than 0.95. With 16 nodes, the curve starts to flatten since the size is too small to get the maximum parallelism. Even so, Interop still improve more than 2X the N-Buffer versions.

In addition, we have performed a weak scaling study, which is shown in Figure 11.8. In this experiment, each node is in charge of 32K x 32K elements and the algorithm performs 500 timesteps. We have reduced the number of timesteps in order to fit the execution time into the maximum allowed execution time of a job.



(a) Speedup graph. (b) Parallel efficiency graph.

Fig. 11.8: Gauss–Seidel weak scaling in Nord3 with 32K x 32K elements per node and 500 timesteps.



(a) Speedup graph.

(b) Parallel efficiency graph.

Fig. 11.9: Gauss–Seidel strong scaling in Nord3 with 16K x 16K total elements and 2000 timesteps.

In this case, the N-Buffer versions are a bit better than Interop with 1 and 2 nodes. However, the former start to drop their parallel efficiency from 4 to 16 nodes, and it ends up with a parallel efficiency of 0.7 at 16 nodes. In turn, the Interop versions scale linearly again up to 16 nodes, and outperforms the N-Buffer versions at 8 and 16 nodes. Probably, we could see more difference between the Interop and N-Buffer versions when increasing the number of timesteps, since the Interop versions can overlap more efficiently multiple timesteps.

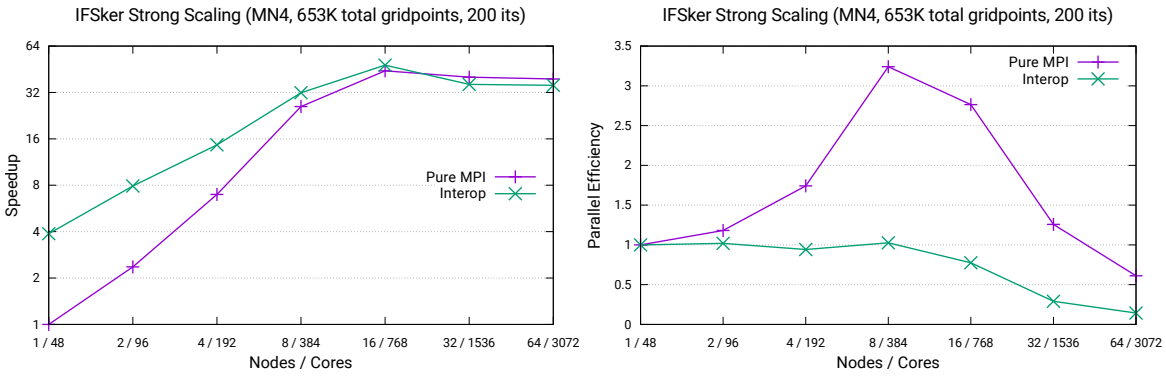
In addition, we have repeated the strong scaling study, but with a 16K x 16K matrix and 2000 timesteps. This experiment is shown in Figure 11.9. Although the problem size is extremely small, the *Interop* versions still get a more than 2X speedup with respect to the *N-Buffer* versions. Note that the parallel efficiency of *Interop* drops to 0.4 with 16 nodes.

These experiments indicate that the difference between the interoperability mechanism integrated into the MPI interoperability ibrary and the one proposed for GASPI have a similar behavior and performance.

11.4.3 IFSker in Marenostrum4

In this section we discuss the performance results of the IFSker application executed in Marenotrum4. As mentioned previously, it features the *Pure MPI* and the *Interop* versions. Both versions are evaluated using an MPI rank per core, and the *Interop* version leverages a single OmpSs thread (in addition to the helper thread) per rank. As in Section 11.4.1, the following experiments have been executed with MPICH 3.2.1.

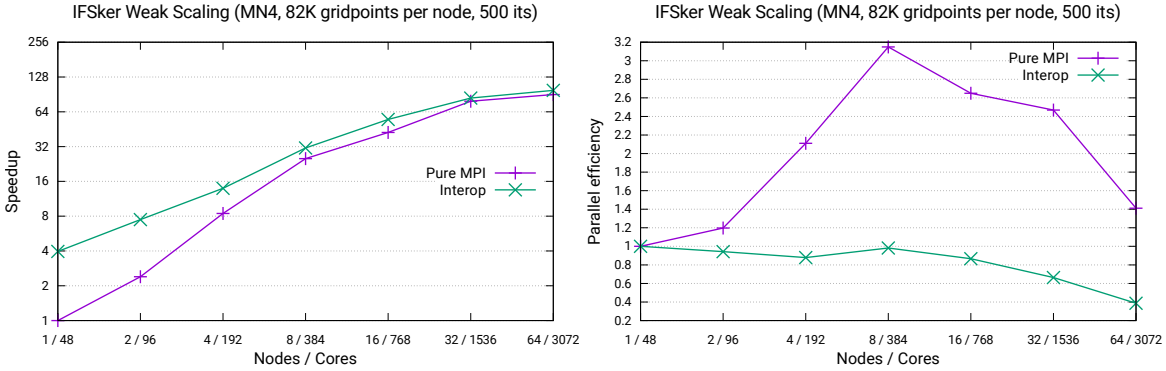
Figure 11.10 shows both the speedup and parallel efficiency graphs of the two versions on a strong-scaling scenario. In the speedup graph we have used the performance of the *Pure MPI* version running on a single node as a baseline. In contrast, in the parallel efficiency graph, each version uses as a baseline its performance on a single node.



(a) Speedup graph.

(b) Parallel efficiency graph.

Fig. 11.10: IFSker strong scaling in Marenostrum4 with 653K total grid-points and 200 timesteps.



(a) Speedup graph.

(b) Parallel efficiency graph.

Fig. 11.11: IFSker weak scaling in Marenostrum4 with 82K grid-points per node and 500 timesteps.

Figure 11.10a shows that, on a single node, the performance of the *Interop* version is 4x higher than that of the *Pure MPI* version. The *Interop* version scales linearly up to 16 nodes; after this point, the problem size becomes too small. It is worth noting that the *Pure MPI* version scales superlinearly and with 16 nodes it reaches the performance of the *Interop* version. This effect is clearly reflected on its parallel efficiency as shown in Figure 11.10b. The parallel efficiency of the *Pure MPI* version grows until it reaches 3.2x at 8 nodes. There are probably some performance problems with the *Pure MPI* version, which could be caused by the fact that when it runs on a single node the performance is much worse.

Lastly, we have performed a weak-scaling experiment, which is shown in Figure 11.11. The *Interop* version scales linearly up to 16 nodes, and after this point, the problem size becomes too small. The *Pure MPI* version repeats the superlinearly scaling up to 8 nodes, and then, it reaches the performance of the *Interop* version at 32 nodes.

As stated in Section 11.3, this application is useful to show that the interoperability mechanism allows applications to execute communication tasks in out-of-order, even with a single thread per MPI rank.

12 | Conclusions

In this project we have introduced a generic API to pause and resume the execution of tasks based on external events. This API has been used for developing a new MPI threading level called MPI_TASK_MULTIPLE, which notifies task-based runtime systems when a task blocks in a synchronous MPI operation, and also, when the operation completes and the task can be resumed. This mechanism prevents the underlying hardware thread from blocking inside the operation and allows the thread to execute other ready tasks. This new threading level has been integrated into the interoperability library that targets hybrid MPI applications.

By taking advantage of the new block/unblock API, we have also designed and developed a similar interoperability mechanism for the GASPI programming model. This mechanism is enabled from the application by using a new initialization procedure, which allows users to initialize the GASPI library with a specific mode. This mechanism is applied when waiting for the local and remote completion of RDMA operations. In addition, we provide other GASPI extensions and we propose some programming techniques for the user.

These mechanisms allow to improve the programmability of applications as well as offering the possibility of parallelizing communications without the risk of generating a deadlock. By taskifying computation and communication phases and using fine-grained dependencies between tasks, the mechanism is able to obtain a better performance from hybrid applications. Usually, this speedup cannot be obtained even when implementing very complex parallelization strategies, such as double-buffering. In addition, the presented API facilitates the task of implementing a similar mechanism for other distributed-memory programming models, or even other blocking services not related to programming models, such as file accesses.

We have evaluated both approaches in MPI and GASPI hybrid applications. The results show that applications can easily get a significant performance speedup with respect to other parallelization strategies, including the most advanced techniques that require important changes in user applications. We also show that applications require very few changes to enable and take advantage of the interoperability mechanism. Finally, we show that both interoperability approaches achieve similar performance results in the same application.

13 | Future Work

This chapter covers the future work that we propose for this project. Firstly, we plan to complete the integration of the proposed API for task event counters. We will repeat the evaluation after adapting the Gauss–Seidel application to use non-blocking MPI calls, thus we will be able to compare the performance of both external events and task block/unblock APIs.

Secondly, another part of our plan is to find and adapt larger applications which could potentially benefit from the interoperability mechanism that we have presented. For instance, communication-intensive applications could be interesting to analyze.

Finally, we want to study how the presented APIs perform in the MPI RMA operations. In addition, we plan to use the interoperability mechanism to improve the integration of task-based programming models with other synchronous APIs.

Bibliography

[1] R. Rabenseifner and G. Wellein. Comparison of parallel programming models on clusters of SMP nodes. In *Proceedings of the International Conference on High Performance Scientifc Computing*, March 2003. 1, 5

[2] Rolf Rabenseifner. Hybrid parallel programming: Performance problems and chances. In *45th Cray User Group Conference*, pages 12–16, 2003.

[3] Gabriele Jost, Haoqiang Jin, and Ferhat F. Hatay. Comparing the OpenMP, MPI, and hybrid programming paradigms on an SMP cluster. Technical report, NASA, September 2003. 1, 5

[4] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1, June 4th 2015. Available at: https://www.mpi-forum.org/docs/ (2018-05-28). 1, 7, 8, 11, 12

[5] GASPI Forum. GASPI: Global Address Space Programming Interface. Version 17.1, February 7th 2017. Available at: http://www.gaspi.de/gaspi/ (2018-05-25). 1, 13, 14, 15, 19

[6] OpenMP Architecture Review Board. OpenMP Application Programming Interface. Version 4.5, November 2015. Available at: https://www.openmp.org/ (2018-06-18). 1

[7] Barcelona Supercomputing Center. OmpSs Specification, . URL https://pm.bsc.es/ompss-docs/specs/. Accessed: 2018-05-31. 1, 22

[8] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, pages 5–16, 2010. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810091. 2, 5, 33, 36

[9] Jorge Bellón Castro. Improving interoperability between OmpSs and MPI. Master's thesis, Universitat Politècnica de Catalunya, 2017. 2, 6, 31, 35, 40, 48, 51, 75

[10] V. Beltran, D. Carrera, J. Torres, and E. Ayguadé. CellMT: A cooperative multithreading library for the Cell/B.E. In *2009 International Conference on High Performance Computing (HiPC)*, pages 245–253, Dec 2009. doi: 10.1109/HIPC.2009.5433205. 5, 6

[11] V. Beltran and E. Ayguadé. Optimizing resource utilization with software-based temporal multi-threading (stmt). In *2012 19th International Conference on High Performance Computing*, pages 1–10, Dec 2012. doi: 10.1109/HiPC.2012.6507499. 5, 6

[12] Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlic, Vincent Cave, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with MPI. In *27th International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 712–725. IEEE, 2013. 5

[13] Barcelona Supercomputing Center. OmpSs-2 Specification, . URL https://pm.bsc.es/ompss-2-docs/spec/. Accessed: 2018-05-31. 22, 23, 24, 25, 27

[14] Barcelona Supercomputing Center. Nanox Runtime System Library, . URL https://github.com/bsc-pm/nanox. Accessed: 2018-06-14. 31, 34, 35, 38

[15] Barcelona Supercomputing Center. Mercurium Compiler, . URL https://github.com/bsc-pm/mcxx. Accessed: 2018-05-21. 37

[16] Barcelona Supercomputing Center. Nanos6 Runtime System Library, . URL https://github.com/bsc-pm/nanos6. Accessed: 2018-05-21. 38

[17] Argonne National Laboratory. MPICH. URL https://www.mpich.org. Accessed: 2018-05-21. 39

[18] Fraunhofer ITWM. GPI-2. URL http://www.gpi-site.com/gpi2. Accessed: 2018-05-21. 40

[19] Barcelona Supercomputing Center. Extrae, . URL https://tools.bsc.es/extrae. Accessed: 2018-05-21. 40

[20] Barcelona Supercomputing Center. Paraver, . URL https://tools.bsc.es/paraver. Accessed: 2018-05-21. 41

[21] Hevner, Alan R. A Three Cycle View of Design Science Research. *Scandinavian Journal of Information Systems*, 19(2):4, 2007. 41, 42, 43

[22] Simon, Herbert A. *The Sciences of the Artificial*. MIT Press, 1996. 43

[23] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1997. ISBN 0-89871-396-X. 71

[24] Giampiero Esposito. *The Heat Equation*, pages 329–334. Springer International Publishing, Cham, 2017. ISBN 978-3-319-57544-5. doi: 10.1007/978-3-319-57544-5_23. URL https://doi.org/10.1007/978-3-319-57544-5_23. 71

A | Gauss–Seidel: Complete Code

This chapter shows the complete code for the Gauss–Seidel benchmark for both MPI and GASPI programming models.

A.1 Taskified MPI+OmpSs-2 Gauss–Seidel

In this section we show the taskified code of the Gauss–Seidel application for MPI and OmpSs-2, specifically the *Sentinel* and *Interop* versions. When launching the application, the user can decide whether to enable or disable the interoperability (e.g. with an argument). If he enables it, the execution will correspond to the *Interop* version. Otherwise, it will correspond to the *Sentinel* version, so the sentinel dependency of communication tasks will be enforced. In addition, there are some auxiliary functions which have been omitted since they are not relevant (e.g. initializeMatrix).

```
#include <mpi.h>
#include ...

// Number of rows in each block
#ifndef BSX
#define BSX 1024
#endif

// Number of columns in each block
#ifndef BSY
#define BSY BSX
#endif

// Block and row definitions. Each block has BSX x BSY
// consecutive elements
typedef double row_t[BSY];
typedef row_t block_t[BSX];

// Structure to save parameters of the execution
struct conf_t {
   int timesteps; // Timesteps to perform
```

```
22    int rows; // Total number of rows
23    int cols; // Total number of cols
24    block_t *matrix; // Pointer to the matrix of blocks
25    bool interop; // Indicate if the interop should be enabled
26    ...
27 };
28
29 // Sentinel used to serialize communication tasks
30 // when the interoperability is disabled
31 int *sentinel;
32
33 void solveBlock(block_t *matrix, int nbx, int nby, int bx, int by) {
34    block_t &block = matrix[bx*nby + by];
35    block_t &topBlock = matrix[(bx-1)*nby + by];
36    block_t &leftBlock = matrix[bx*nby + (by-1)];
37    block_t &rightBlock = matrix[bx*nby + (by+1)];
38    block_t &bottomBlock = matrix[(bx+1)*nby + by];
39
40    for (int x = 0; x < BSX; ++x) {
41       row_t &topRow = (x > 0) ? block[x-1] : topBlock[BSX-1];
42       row_t &bottomRow = (x < BSX-1) ? block[x+1] : bottomBlock[0];
43
44       for (int y = 0; y < BSY; ++y) {
45          double left = (y > 0) ? block[x][y-1] : leftBlock[x][BSY-1];
46          double right = (y < BSY-1) ? block[x][y+1] : rightBlock[x][0];
47          block[x][y] = 0.25 * (topRow[y] + bottomRow[y] + left + right);
48       }
49    }
50 }
51
52 void sendFirstComputeRow(block_t *matrix, int nbx, int nby, int rank) {
53    for (int by = 1; by < nby-1; ++by) {
54       #pragma oss task in(([nbx][nby]matrix)[1][by]) inout(*sentinel)
55       MPI_Send(&matrix[nby+by][0], BSY, MPI_DOUBLE, rank - 1,
56             /* tag */ by, MPI_COMM_WORLD);
57    }
58 }
59
60 void sendLastComputeRow(block_t *matrix, int nbx, int nby, int rank) {
61    for (int by = 1; by < nby-1; ++by) {
62       #pragma oss task in(([nbx][nby]matrix)[nbx-2][by]) inout(*sentinel)
63       MPI_Send(&matrix[(nbx-2)*nby+by][BSX-1], BSY, MPI_DOUBLE, rank + 1,
64             /* tag */ by, MPI_COMM_WORLD);
65    }
66 }
67
68 void receiveUpperBorder(block_t *matrix, int nbx, int nby, int rank) {
69    for (int by = 1; by < nby-1; ++by) {
```

```
70        #pragma oss task out(([nbx][nby]matrix)[0][by]) inout(*sentinel)
71        MPI_Recv(&matrix[by][BSX-1], BSY, MPI_DOUBLE, rank - 1,
72              /* tag */ by, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
73     }
74  }
75
76  void receiveLowerBorder(block_t *matrix, int nbx, int nby, int rank) {
77     for (int by = 1; by < nby-1; ++by) {
78        #pragma oss task out(([nbx][nby]matrix)[nbx-1][by]) inout(*sentinel)
79        MPI_Recv(&matrix[(nbx-1)*nby+by][0], BSY, MPI_DOUBLE, rank + 1,
80              /* tag */ by, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
81     }
82  }
83
84  void solve(block_t *matrix, int nbx, int nby, int rank, int rank_size) {
85     if (rank != 0) {
86        // Create tasks to send the first row to the previous rank
87        sendFirstComputeRow(matrix, nbx, nby, rank, rank_size);
88
89        // Create tasks to receive the upper border from the previous rank
90        receiveUpperBorder(matrix, nbx, nby, rank, rank_size);
91     }
92
93     if (rank != rank_size - 1) {
94        // Create tasks to receive the lower border from the next rank
95        receiveLowerBorder(matrix, nbx, nby, rank, rank_size);
96     }
97
98     // Create a task to compute each block
99     for (int bx = 1; bx < nbx-1; ++bx) {
100       for (int by = 1; by < nby-1; ++by) {
101          #pragma oss task inout(([nbx][nby]matrix)[bx][by]) \
102                in(([nbx][nby]matrix)[bx-1][by]) \
103                in(([nbx][nby]matrix)[bx][by-1]) \
104                in(([nbx][nby]matrix)[bx][by+1]) \
105                in(([nbx][nby]matrix)[bx+1][by])
106          solveBlock(matrix, nbx, nby, bx, by);
107       }
108    }
109
110    if (rank != rank_size - 1) {
111       // Create tasks to send the last row to the next rank
112       sendLastComputeRow(matrix, nbx, nby, rank, rank_size);
113    }
114 }
115
116 void gaussSeidel(block_t *matrix, int nbx, int nby, int timesteps) {
117    int rank, rank_size;
```

```
118     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
119     MPI_Comm_size(MPI_COMM_WORLD, &rank_size);
120
121     for (int t = 0; t < timesteps; ++t) {
122        solve(matrix, nbx, nby, rank, rank_size);
123     }
124
125     #pragma oss taskwait
126     MPI_Barrier(MPI_COMM_WORLD);
127 }
128
129 int main(int argc, char **argv) {
130     int required, provided;
131     conf_t conf;
132
133     // Read input parameters
134     readConfiguration(argc, argv, &conf);
135
136     // Require the new threading level if the user wants to enable
137     // the interoperability mechanism
138     required = (conf.interop) ? MPI_TASK_MULTIPLE : MPI_THREAD_MULTIPLE;
139
140     // Initialize safely the MPI library
141     MPI_Init_thread(&argc, &argv, required, &provided);
142     if (provided < MPI_THREAD_MULTIPLE) {
143        printf("Error: Multi-threading level is not supported!");
144        return 1;
145     } else if (required == MPI_TASK_MULTIPLE && provided < required) {
146        printf("Task level is not supported! Enabling sentinel...");
147        conf.interop = false;
148     }
149
150     // Enable/disable the sentinel dependency
151     if (provided == MPI_TASK_MULTIPLE) {
152        sentinel = NULL;
153     } else {
154        sentinel = (int *) 1;
155     }
156
157     int rank, rank_size;
158     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
159     MPI_Comm_size(MPI_COMM_WORLD, &rank_size);
160
161     // Add an additional block in each border
162     int rowBlocks = (conf.rows / BSX) + 2;
163     int colBlocks = (conf.cols / BSY) + 2;
164     int rowBlocksPerRank = rowBlocks / rank_size + 2;
165     int blocksPerRank = rowBlocksPerRank * colBlocks;
```

```
166
167     // Allocate memory for the partial matrix corresponding to a rank
168     conf.matrix = (block_t *) malloc(blocksPerRank * sizeof(block_t));
169
170     // Initialize the matrix
171     initializeMatrix(conf, rowBlocksPerRank, colBlocks);
172     MPI_Barrier(MPI_COMM_WORLD);
173
174     // Solve the problem
175     double start = get_time();
176     gaussSeidel(conf.matrix, rowBlocksPerRank, colBlocks, conf.timesteps);
177     double end = get_time();
178
179     free(conf.matrix);
180     MPI_Finalize();
181     return 0;
182 }
```

Code A.1: Gauss–Seidel code of the MPI+OmpSs-2 Sentinel and Interop versions.

A.2 Taskified GASPI+OmpSs-2 Gauss–Seidel

Similarly, in this section we show the taskified code of the Gauss–Seidel application for GASPI and OmpSs-2, specifically the *Sentinel* and *Interop* versions. When launching the application, the user can decide whether to enable or disable the interoperability. If he enables it, the execution will correspond to the *Interop* version. Otherwise, it will correspond to the *Sentinel* version, so the sentinel dependency of communication tasks will be enforced.

There are also some auxiliary functions and macros which have been omitted since they are not relevant, such as the macros to compute relative offsets in a segment. For instance, the UB_OFFSET macro computes the base offset of the upper border of the matrix. If the previous base offset is saved to the baseOffset variable, the actual offset to the upper border of the column of blocks by can be computed by executing the macro OFFSET(baseOffset, by). The rest of macros work similarly. These computations have been encapsulated in order to improve the readability of the code.

```
1 #include <GASPI.h>
2 #include ...
3
4 // Number of rows in each block
5 #ifndef BSX
6 #define BSX 1024
7 #endif
8
```

```
9  // Number of columns in each block
10 #ifndef BSY
11 #define BSY BSX
12 #endif
13
14 // GASPI segment identifier
15 #define SEGMENT 0
16
17 // Macros to compute the offset (in bytes) of the boundaries in the matrix
18 #define FCR_OFFSET(...) ...
19 ...
20
21 // Block and row definitions. Each block has BSX x BSY
22 // consecutive elements
23 typedef double row_t[BSY];
24 typedef row_t block_t[BSX];
25
26 // Structure to save parameters of the execution
27 struct conf_t {
28    int timesteps; // Timesteps to perform
29    int rows; // Total number of rows
30    int cols; // Total number of cols
31    block_t *matrix; // Pointer to the matrix of blocks
32    bool interop; // Indicate if the interop should be enabled
33    ...
34 };
35
36 // Structure to save some GASPI information
37 struct gaspi_info_t {
38    int numQueues; // Number of queues used
39    int precreatedQueues; // Number of precreated queues at initialization
40    ...
41 }
42
43 // Sentinel used to serialize communication tasks
44 // when the interoperability is disabled
45 int *sentinel;
46
47 // Blocking mode (or timeout) used in blocking calls
48 gaspi_timeout_t blockMode;
49
50 void solveBlock(block_t *matrix, int nbx, int nby, int bx, int by) {
51    block_t &block = matrix[bx*nby + by];
52    block_t &topBlock = matrix[(bx-1)*nby + by];
53    block_t &leftBlock = matrix[bx*nby + (by-1)];
54    block_t &rightBlock = matrix[bx*nby + (by+1)];
55    block_t &bottomBlock = matrix[(bx+1)*nby + by];
56
```

```
57    for (int x = 0; x < BSX; ++x) {
58       row_t &topRow = (x > 0) ? block[x-1] : topBlock[BSX-1];
59       row_t &bottomRow = (x < BSX-1) ? block[x+1] : bottomBlock[0];
60
61       for (int y = 0; y < BSY; ++y) {
62          double left = (y > 0) ? block[x][y-1] : leftBlock[x][BSY-1];
63          double right = (y < BSY-1) ? block[x][y+1] : rightBlock[x][0];
64          block[x][y] = 0.25 * (topRow[y] + bottomRow[y] + left + right);
65       }
66    }
67 }
68
69 void sendFirstComputeRow(block_t *matrix, int nbx, int nby, int rank) {
70    const gaspi_offset_t baseOffset = FCR_OFFSET(nbx, nby);
71    const gaspi_offset_t remoteBaseOffset = LB_OFFSET(nbx, nby);
72
73    for (int by = 1; by < nby-1; ++by) {
74       #pragma oss task in(([nbx][nby]matrix)[1][by]) inout(*sentinel)
75       {
76          gaspi_queue_id_t queue;
77          gaspi_queue_group_get_queue(0, &queue);
78
79          gaspi_write_notify(SEGMENT, OFFSET(baseOffset, by),
80                rank - 1, SEGMENT, OFFSET(remoteBaseOffset, by),
81                BSY * sizeof(double), nby + by, 1, queue, GASPI_BLOCK);
82
83          gaspi_wait(queue, blockMode);
84       }
85    }
86 }
87
88 void sendLastComputeRow(block_t *matrix, int nbx, int nby, int rank) {
89    const gaspi_offset_t baseOffset = LCR_OFFSET(nbx, nby);
90    const gaspi_offset_t remoteBaseOffset = UB_OFFSET(nbx, nby);
91
92    for (int by = 1; by < nby-1; ++by) {
93       #pragma oss task in(([nbx][nby]matrix)[nbx-2][by]) inout(*sentinel)
94       {
95          gaspi_queue_id_t queue;
96          gaspi_queue_group_get_queue(0, &queue);
97
98          gaspi_write_notify(SEGMENT, OFFSET(baseOffset, by),
99                rank + 1, SEGMENT, OFFSET(remoteBaseOffset, by),
100               BSY * sizeof(double), by, 1, queue, GASPI_BLOCK);
101
102          gaspi_wait(queue, blockMode);
103       }
104    }
```

```
105 }
106
107 void receiveUpperBorder(block_t *matrix, int nbx, int nby, int rank) {
108    gaspi_notification_id_t notifiedId;
109    gaspi_notification_t notifiedValue;
110
111    for (int by = 1; by < nby-1; ++by) {
112       #pragma oss task out(([nbx][nby]matrix)[0][by]) inout(*sentinel)
113       {
114          gaspi_notify_waitsome(SEGMENT, by, 1, &notifiedId, blockMode);
115          assert(notifiedId == by);
116
117          gaspi_notify_reset(SEGMENT, notifiedId, &notifiedValue);
118          assert(notifiedValue == 1);
119       }
120    }
121 }
122
123 void receiveLowerBorder(block_t *matrix, int nbx, int nby, int rank) {
124    gaspi_notification_id_t notifiedId;
125    gaspi_notification_t notifiedValue;
126
127    for (int by = 1; by < nby-1; ++by) {
128       #pragma oss task out(([nbx][nby]matrix)[nbx-1][by]) inout(*sentinel)
129       {
130          gaspi_notify_waitsome(SEGMENT, nby+by, 1, &notifiedId, blockMode);
131          assert(notifiedId == nby+by);
132
133          gaspi_notify_reset(SEGMENT, notifiedId, &notifiedValue);
134          assert(notifiedValue == 1);
135       }
136    }
137 }
138
139 void solve(block_t *matrix, int nbx, int nby, int rank, int rank_size) {
140    if (rank != 0) {
141       // Create tasks to send the first row to the previous rank
142       sendFirstComputeRow(matrix, nbx, nby, rank, rank_size);
143
144       // Create tasks to receive the upper border from the previous rank
145       receiveUpperBorder(matrix, nbx, nby, rank, rank_size);
146    }
147
148    if (rank != rank_size - 1) {
149       // Create tasks to receive the lower border from the next rank
150       receiveLowerBorder(matrix, nbx, nby, rank, rank_size);
151    }
152
```

```
153    // Create a task to compute each block
154    for (int bx = 1; bx < nbx-1; ++bx) {
155       for (int by = 1; by < nby-1; ++by) {
156          #pragma oss task inout(([nbx][nby]matrix)[bx][by]) \
157                in(([nbx][nby]matrix)[bx-1][by]) \
158                in(([nbx][nby]matrix)[bx][by-1]) \
159                in(([nbx][nby]matrix)[bx][by+1]) \
160                in(([nbx][nby]matrix)[bx+1][by])
161          solveBlock(matrix, nbx, nby, bx, by);
162       }
163    }
164
165    if (rank != rank_size - 1) {
166       // Create tasks to send the last row to the next rank
167       sendLastComputeRow(matrix, nbx, nby, rank, rank_size);
168    }
169 }
170
171 void gaussSeidel(block_t *matrix, int nbx, int nby, int timesteps) {
172    gaspi_rank_t rank, rank_size;
173    gaspi_proc_rank(&rank);
174    gaspi_proc_num(&rank_size);
175
176    for (int t = 0; t < timesteps; ++t) {
177       solveGaussSeidel(matrix, nbx, nby, rank, rank_size, info);
178    }
179
180    #pragma oss taskwait
181    gaspi_barrier(GASPI_GROUP_ALL, GASPI_BLOCK);
182 }
183
184 void setupGaspiInfo(conf_t &conf, int nbx, int nby, gaspi_info_t &info) {
185    // Commit the default group
186    gaspi_group_commit(GASPI_GROUP_ALL, GASPI_BLOCK);
187
188    // Register a segment for the local buffer of particles
189    gaspi_segment_use(SEGMENT, conf.matrix,
190          rowBlocks * colBlocks * sizeof(block_t),
191          GASPI_GROUP_ALL, GASPI_BLOCK, 0);
192
193    // Set the desired number of queues
194    gaspi_queue_num(&info.precreatedQueues);
195    gaspi_queue_max(&info.numQueues);
196    if (!conf.interop) info.numQueues = 1;
197
198    // Create the rest of queues
199    gaspi_queue_id_t i, queue;
200    for (i = info.precreatedQueues; i < info.numQueues; ++i) {
```

```
201       gaspi_queue_create(&queue, GASPI_BLOCK);
202    }
203
204    // Create the queue group with the previous queues
205    gaspi_queue_group_policy_t policy;
206    if (conf.interop) {
207       policy = GASPI_QUEUE_GROUP_POLICY_CPU_RR;
208    } else {
209       policy = GASPI_QUEUE_GROUP_POLICY_DEFAULT;
210    }
211    gaspi_queue_group_create(0, 0, info.numQueues, policy);
212 }
213
214 void freeGaspiInfo(gaspi_info_t &info) {
215    gaspi_segment_delete(SEGMENT);
216
217    gaspi_queue_id_t q;
218    for (q = 0; q < info.numQueues; ++q) {
219       gaspi_wait(q, GASPI_BLOCK);
220    }
221
222    gaspi_queue_group_delete(0);
223
224    for (q = info.precreatedQueues; q < info.numQueues; ++q) {
225       gaspi_queue_delete(q);
226    }
227 }
228
229 int main(int argc, char **argv) {
230    int required, provided;
231    gaspi_info_t gaspiInfo;
232    conf_t conf;
233
234    // Read input parameters
235    readConfiguration(argc, argv, &conf);
236
237    // Require the new task mode if the user wants to enable
238    // the interoperability mechanism
239    required = (conf.interop) ? GASPI_MODE_TASK : GASPI_MODE_STANDARD;
240
241    // Initialize safely the GASPI library
242    gaspi_proc_init_mode(required, &provided, GASPI_BLOCK);
243    if (required == GASPI_MODE_TASK && provided < required) {
244       printf("Task mode is not supported! Enabling sentinel...");
245       conf.interop = false;
246    }
247
248    // Enable/disable the sentinel dependency and set the
```

```
249     // corresponding blocking mode for blocking calls
250     if (provided == GASPI_MODE_TASK) {
251        sentinel = NULL;
252        blockMode = GASPI_BLOCK_TASK;
253     } else {
254        sentinel = (int *) 1;
255        blockMode = GASPI_BLOCK;
256     }
257
258     gaspi_rank_t rank, rank_size;
259     gaspi_proc_rank(&rank);
260     gaspi_proc_num(&rank_size);
261
262     // Add an additional block in each border
263     int rowBlocks = (conf.rows / BSX) + 2;
264     int colBlocks = (conf.cols / BSY) + 2;
265     int rowBlocksPerRank = rowBlocks / rank_size + 2;
266     int blocksPerRank = rowBlocksPerRank * colBlocks;
267
268     // Allocate memory for the partial matrix corresponding to a rank
269     conf.matrix = (block_t *) malloc(blocksPerRank * sizeof(block_t));
270
271     // Initialize the matrix
272     initializeMatrix(conf, rowBlocksPerRank, colBlocks);
273
274     setupGaspiInfo(conf, rowBlocksPerRank, colBlocks, &gaspiInfo);
275     gaspi_barrier(GASPI_GROUP_ALL, GASPI_BLOCK);
276
277     // Solve the problem
278     double start = get_time();
279     solve(conf.matrix, rowBlocksPerRank, colBlocks, conf.timesteps);
280     double end = get_time();
281
282     freeGaspiInfo(gaspiInfo);
283     free(conf.matrix);
284     gaspi_proc_term(GASPI_BLOCK);
285     return 0;
286 }
```

Code A.2: Gauss–Seidel code of the GASPI+OmpSs-2 Sentinel and Interop versions.