Master in Innovation and Research in Informatics
Spec. in High Performance Computing

Improving interoperability between OmpSs and MPI

Jorge Bellón Castro
Director: Jesús Labarta Mancho

9th April, 2017
Abstract

Programming tools and application libraries are the second level of the software stack in HPC systems, on top of the operative system and the middleware. They allow users to build their applications and simplify their development. In addition, these tools and libraries are specifically designed and optimized to make the most efficient use of the processors installed in these systems.

The following generations of computers will introduce many new challenges to HPC software development. As an example, Sunway TaihuLight supercomputer, a Chinese supercomputer among the most powerful computer systems in the world, incorporates 260 cores per processor with an unusual memory hierarchy. Maintaining such level of complexity in an application is unfeasible. Furthermore, applications dealing with these hardware particularities would not be portable, requiring constant changes when moving from one system to another.

Message passing libraries and parallel programming frameworks are one of the core parts in HPC systems, that cope with the details of their hardware components and provide a standard interface to applications, allowing these to run on multiple compute nodes while maintaining portability.

Current software efforts tend towards moving the common software parts to specialized libraries, meanwhile applications are left with the specific logic of the problem they are meant to solve. Therefore, being able to combine these libraries in an application is essential.

In this work, we present OmpSs-MPI interoperability library. This library is conceived to simplify the development of hybrid programs with OmpSs (a parallel programming framework) and MPI (a message passing library). In addition, it paves the way for existing OmpSs-only or MPI-only applications to adopt the benefits of MPI and OmpSs models respectively.
Contents

1 Introduction 4
   1.1 MPI ............................................................ 4
       1.1.1 Point-to-point communications .............................. 5
       1.1.2 Collective communications .................................. 6
       1.1.3 Non-blocking communication ................................. 6
       1.1.4 MPI datatypes .............................................. 7
       1.1.5 MPI profiling interface .................................... 8
   1.2 OmpSs ......................................................... 9
       1.2.1 OmpSs data dependences .................................... 10
       1.2.2 Task scheduling points ..................................... 10
   1.3 Hybrid parallel programs ...................................... 11
       1.3.1 MPI blocking communications and task-based parallelism 11

2 Implementation 13
   2.1 External explicit synchronization of task execution .......... 15
   2.2 External function polling mechanism .......................... 16
   2.3 Putting it all together ....................................... 18

3 Evaluation 20
   3.1 Performance metrics ......................................... 20
   3.2 Evaluation environment ...................................... 23
       3.2.1 Marenostrum 3 ............................................ 23
       3.2.2 Minotauro M2090 .......................................... 23
       3.2.3 Software .................................................. 24
   3.3 IFSker ....................................................... 24
       3.3.1 Analysis .................................................. 26
       3.3.2 Improvements ............................................. 28
       3.3.3 Results ................................................... 32
   3.4 High performance Linpack benchmark .......................... 35
       3.4.1 Analysis .................................................. 36
       3.4.2 Task-based parallelization ................................ 38
       3.4.3 Results ................................................... 40

4 Conclusions and future work 45
   4.1 Related work ................................................ 46
# List of Figures

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Computation communication overlap</td>
<td>6</td>
</tr>
<tr>
<td>1.2</td>
<td>Non-contiguous MPI datatypes</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>MPI profiling interface sequence diagram</td>
<td>8</td>
</tr>
<tr>
<td>1.4</td>
<td>MPI point to point messages as inter-process extension of OmpSs data dependences</td>
<td>11</td>
</tr>
<tr>
<td>2.1</td>
<td>OmpSs-MPI library software stack</td>
<td>13</td>
</tr>
<tr>
<td>2.2</td>
<td>Blocking communication interpose sequence</td>
<td>18</td>
</tr>
<tr>
<td>2.3</td>
<td>Polling function sequence</td>
<td>18</td>
</tr>
<tr>
<td>3.1</td>
<td>IFSker data distribution</td>
<td>25</td>
</tr>
<tr>
<td>3.2</td>
<td>IFSker baseline task execution diagram</td>
<td>26</td>
</tr>
<tr>
<td>3.3</td>
<td>Physics and FFT task durations</td>
<td>27</td>
</tr>
<tr>
<td>3.4</td>
<td>IFSker baseline task dependence graph</td>
<td>27</td>
</tr>
<tr>
<td>3.5</td>
<td>Using multiple MPI communicators</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>IFS data redistribution</td>
<td>29</td>
</tr>
<tr>
<td>3.7</td>
<td>Non-contiguous MPI datatypes memory layouts</td>
<td>30</td>
</tr>
<tr>
<td>3.8</td>
<td>IFSker improved task dependence graph</td>
<td>31</td>
</tr>
<tr>
<td>3.9</td>
<td>Norm computation comparison</td>
<td>32</td>
</tr>
<tr>
<td>3.10</td>
<td>IFSker improved execution diagram</td>
<td>33</td>
</tr>
<tr>
<td>3.11</td>
<td>IFSker CPU load comparison</td>
<td>33</td>
</tr>
<tr>
<td>3.12</td>
<td>IFSker scalability results</td>
<td>34</td>
</tr>
<tr>
<td>3.13</td>
<td>HPL data partition</td>
<td>35</td>
</tr>
<tr>
<td>3.14</td>
<td>HPL submatrices</td>
<td>36</td>
</tr>
<tr>
<td>3.15</td>
<td>HPL communication patterns</td>
<td>37</td>
</tr>
<tr>
<td>3.16</td>
<td>Average CPU load</td>
<td>37</td>
</tr>
<tr>
<td>3.17</td>
<td>dgemm duration</td>
<td>38</td>
</tr>
<tr>
<td>3.18</td>
<td>HPL execution time and throughput</td>
<td>38</td>
</tr>
<tr>
<td>3.19</td>
<td>HPL task dependence diagram</td>
<td>39</td>
</tr>
<tr>
<td>3.20</td>
<td>HPL task execution diagram</td>
<td>40</td>
</tr>
<tr>
<td>3.21</td>
<td>Task duration and number of switches due to pending communications.</td>
<td>41</td>
</tr>
<tr>
<td>3.22</td>
<td>Task execution diagram zoom</td>
<td>41</td>
</tr>
<tr>
<td>3.23</td>
<td>Execution diagram comparison with comm. thread</td>
<td>42</td>
</tr>
<tr>
<td>3.24</td>
<td>CPU load comparison</td>
<td>42</td>
</tr>
<tr>
<td>3.25</td>
<td>HPL execution diagram of the last iterations</td>
<td>43</td>
</tr>
<tr>
<td>3.26</td>
<td>HPL benchmark results</td>
<td>44</td>
</tr>
</tbody>
</table>
3.27 dgemm throughput comparison between original and task based implementations.
Chapter 1

Introduction

The capability of building next generation exascale computers implies solving many new and complex challenges. It will not simply consist on building systems with current technologies a thousand times bigger. Simply scaling up from today’s requirements for a petaflop computer, the exaflop computer in 2020 would consume 200 MW, which is not affordable.

Memory bandwidth and capacity do not increase at the same rate than processing performance (a phenomena known as memory wall). This means that the amount of memory per core will fall compared to current systems. Consequently, many of the current applications, which rely on solving big problems to achieve an acceptable amount of parallelism, will not be able to scale.

On the other hand, clock frequencies are expected to decrease in order to save power: exascale architectures will likely be high-concurrency: higher number of processing units per chip will be necessary to keep increasing the overall computational capacity of a system.

On top of that, the cost of data movement in terms of energy consumption and performance is expected to dominate.

Therefore, developing applications for these exascale systems will be very complex. Most codes will need to be either adapted or completely re-designed to cope with all these limitations and be able to efficiently manage their compute capabilities. It is critical to anticipate this complexity, as applications adopt technology changes at a slower pace than hardware technology.

Current trends show that most HPC developers use OpenMP and MPI programming models to exploit parallelism in their applications [1]. In many cases, they combine them together to fully exploit their potential.

In this work, we study the effects on application’s efficiency when combining MPI and OmpSs (a shared-memory programming model similar to OpenMP). In particular, we show that MPI routines can be used within OmpSs tasks to improve computation and communication overlap, and how this can be used in a safe and effective way.

1.1 MPI

MPI (Message-Passing Interface) [2] is a message-passing library interface. It mainly delivers the message-passing parallel programming model, in which data is moved from the address space of one process to the one of another process, by means of coordinated operations. MPI is the specification for a library interface, which is provided by implementations from many different vendors.
Therefore, its operations are expressed as functions, subroutines or methods, depending on the programming language.

The main advantages of using a standard interface are portability and ease of use. The definition of a message-passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases for which they can provide hardware support.

MPI, since its appearance in 1994, is one of the most widespread programming models for distributed memory environments. It adapted itself to facilitate its propagation to new hardware architectures and has evolved through the years to include more functionality, extending the classic message-passing model with collective operations, remote-memory access operations, dynamic process creation, and parallel I/O.

The following sections explain some features and details of MPI that are considered important to understand some concepts that are developed further on in this document.

### 1.1.1 Point-to-point communications

Send and receive operations are the most simple form of communication in MPI. These are also known as point to point because the communication always takes place between one transmitter and one receiver.

Messages carry additional information that allows identification of messages, so that selective reception is possible. This information is called message envelope, and is composed by the following fields:

- **source** who is sending the message
- **destination** who receives it
- **communicator** specifies a communication context
- **tag** a numeric value specified by the user

This information is usually provided by the user, either explicitly through routine arguments, or implicitly (e.g. source is well known from the sender point of view). Therefore, we say that send and receive operations match when all the parameters in both sides share the same message envelope.

MPI standard guarantees that point to point communications are always ordered in single thread applications. This means that if a process sends two messages in a succession and both of them match the same receive operation, this receive will not receive the second message if the first one is still pending. However, when multiple threads simultaneously issue reception of messages, the operations are logically concurrent and these threads can receive them in any order.

Therefore, it is specially important to uniquely identify concurrent messages transmitted in multithreaded applications, so that the receiver knows exactly where to place the data. To ensure this identification, the user must make sure that the message envelope is not repeated in potentially concurrent messages. If restriction is satisfied, multiple messages can be safely sent and received in any order.

---

1. Semantics of point-to-point communication. MPI standard 3.1, section 3.5
Figure 1.1: It is possible to take advantage of the idle CPU time during communications to do profitable work. In this example, we split all stages in two halves. The delivery of the first half starts meanwhile the second one is still being computed.

1.1.2 Collective communications

Collective communications involve the participation of a group of an arbitrary number of processes. In contrast to point-to-point communications, collectives do not use a tag argument, but the messages generated as a result are guaranteed not to interfere with other point-to-point communications performed by the user.

The group of processes involved in a collective communication is not defined explicitly through arguments, but rather implicit in the communicator. However, it is possible to restrict a collective’s scope to a reduced number of processes through the creation of new derived communicators.

Multithreaded applications may use multiple concurrent collectives, meanwhile the same communicator is not used more than once. In addition, the relative order of the collective communications must be the same in all processes of an application in order to avoid deadlocks.

1.1.3 Non-blocking communication

MPI point-to-point and collective communication routines are blocking by default. This means that, for example, a call to a receive operation does not return until the communication is completed.

Non-blocking communications decouple the initialization and synchronization of a communication. This way, an application can make progress with computations which do not depend directly on a given communication while it is still in progress. Therefore, this mechanism allows to effectively overlap communication with useful computation. Figure 1.1 shows how overlapping can benefit the execution of an application.

MPI non-blocking communication routines return a request object. This object is used by MPI_Test and MPI_Wait family of functions later on. These routines respectively check or wait until a communication is completed:

- **MPI_Test**/*MPI_Wait** used to wait for a single request.
- **MPI_Testany**/*MPI_Waitany** used to wait for one out of multiple requests finishes and returns its position.
- **MPI_Testall**/*MPI_Waitall** used to wait for the completion of all requests.
- **MPI_Testsome**/*MPI_Waitsome** used to wait for the completion of one or more requests, returning their positions.
code 1.1: Example code for process $P_0$ in figure 1.1. MPI non-blocking routines allow computation-communication overlap, at the cost of higher code complexity.

```c
void sender ( int * data , int count ) {
    int half = count / 2 , rem = count - half;
    MPI_Request reqs[2];

    computation( data , half );
    MPI_Isend ( data , half , ... , & reqs[0] );

    computation( data+half , rem );
    MPI_Isend ( data+half , rem , ... , & reqs[1] );

    MPI_Waitall ( 2 , reqs , ... );
}
```

There are different performance implications in using one or another. For example, the user can imitate the functionality of MPI_Waitall by manually calling MPI_Wait for each request in a list. This, however, would mean calling the MPI library multiple times, paying the cost involved in it. MPI_Waitsome operation is particularly recommended by MPI due to efficiency reasons. For example, if two requests are completed by the time MPI_Waitany is called, it would take up to two MPI calls to realize so. In this particular case, MPI_Waitsome would only require a single call to the library, returning the two requests’ positions at once.

1.1.4 MPI datatypes

Data transmitted in MPI communications consists on count successive elements of a given datatype. MPI provides by default the definition of basic datatypes, which correspond to the basic datatypes of the host language (e.g. int, float or char in C). These datatypes can be extended to describe any data layout, which allows one to transfer heterogeneous and non-contiguous data efficiently (see figure 1.2).

Figure 1.2: MPI datatypes allow to directly specify communication buffers with non-contiguous layouts, saving additional data movement.

When transmitting complex data layouts not supported by MPI basic types, one solution is to pack this data into a contiguous buffer at the sender side and unpack it at the receiver’s. This has the disadvantage of requiring additional memory-to-memory copy operations at both ends, even when the communication subsystem has scatter-gather capabilities. On the other hand, MPI derived

---

2 MPI standard 3.1 section 3.7.5
datatypes allows programs to specify non-contiguous storage as communication buffers directly, thus allowing the implementation to decide whether data should be first packed in a contiguous buffer before being transmitted, or if it can be collected directly from where it resides.

1.1.5 MPI profiling interface

In addition to the common application interface, MPI standard describes an additional interface that allows easy placement of external tools between user applications and the MPI library.

MPI profiling interface defines an alternative function symbol for each routine, which is preceded with PMPI instead of MPI (figure 1.2). Tools use these symbols to call the actual routine and avoid conflicts (see figure 1.3). The main advantage is that tools are not required to re-implement all MPI interface, but only routines of interest.

Furthermore, MPI profiling interface provides MPI_Pcontrol, an auxiliary function. This function does nothing by default, but it can be used by tool implementations to give the user the ability to enable/disable features at runtime.

Code 1.2: Using MPI profiling interface we can time MPI_Send call durations.

```c
int MPI_Send( const void *buf, int count, MPI_Datatype datatype,
              int dest, int tag, MPI_Comm comm )
{
    double start = MPI_Wtime();
    int err = PMPI_Send( buf, count, datatype, dest, tag, comm );
    double end    = MPI_Wtime();
    // Elapsed = end - start
    return err;
}
```

Figure 1.3: External tools can be placed between applications and the MPI library using MPI profiling interface. In this figure, a tool intercepts MPI_Send routine and calls the actual send operation using PMPI_Send symbol.

It must be noted that, since MPI is designed primarily for C and Fortran languages, profiling tools should provide implementations for both languages, as only defining C symbols does not guarantee that corresponding Fortran symbols will be intercepted as well.
1.2 OmpSs

OmpSs [3] is a programming model used to develop parallel programs. It is composed by a set of library routines and compiler directives that can be used in a high level programming language such as C/C++ or Fortran.

Parallelism is expressed with multiple independent portions of sequential instructions, known as tasks, that can be run at a given point in time. Tasks run asynchronously, that is, coordination is not performed by the main flow of the program, but it is rather restricted to those tasks they depend with.

Therefore, OmpSs primarily focuses on tasks and dependences: tasks are the elementary unit of work and data dependences drive the flow of the program. Dependence information is provided by the user, and it is used during the program execution to check if the parallel execution of a set of tasks may cause data races.

OmpSs philosophy consists on producing parallel programs starting from their sequential form, where the source code is modified by introducing annotations which do not have an explicit effect in the semantics of the program. Instead, they can be interpreted by compilers to produce a parallel version. This characteristic feature was inspired by OpenMP, and it allows users to parallelize applications incrementally, which has a big impact on productivity.

More explicit programming models require applications to be redesigned: the user is responsible of how the parallelism is expressed and exploited, which increases maintenance efforts of the source code. For example, common activities such as debugging and testing become more difficult.

The goal is to provide a productive and efficient environment for High Performance Computing (HPC) applications. Programs developed in OmpSs must be able to deliver a reasonable performance when compared to other programming models that target the same architectures. In particular, one of the most ambitious objectives is to extend OpenMP standard specification to better support asynchronous data-flow parallelism and heterogeneous system programming.

During the initialization of an OmpSs program, a team of threads is created by the runtime library upon initialization. This team of threads is composed by a single master thread and a variable number of worker threads. The master thread runs the main function in the context of an implicit task (i.e. not annotated by the user). Worker threads wait until tasks are made available.

Task constructs explicitly produce the creation of a new task. The execution of the task can be immediate or deferred following dependence constraints, scheduling policies and thread availability.

Furthermore, the execution of a task may be suspended before its completion. This can be produced by scheduling restrictions (e.g. wait until descendant tasks are finished) or decisions (immediate execution of a task after its creation, yielding the execution resources in favor of other

---

Code 1.3: OmpSs program example. Ignoring compiler directives still produces a valid program.

```c
int main () {
    #pragma omp task
    printf("Hello, world!\n");

    #pragma omp taskwait
    return 0;
}
```
tasks, etc.). In these scenarios, known as task scheduling points, threads are allowed to stop the current task’s execution and start or resume another.

Eventually, the task execution completes and all the dependences with its successors can be satisfied. Then, it is safe to destroy the task and release all resources that it was using.

1.2.1 OmpSs data dependences

Asynchronous coordination of OmpSs tasks is guided by data dependences. Typically a program performs meaningful computation by processing some input data and generating a result. Since OmpSs programs split computation in multiple tasks, a common scenario consists on some tasks generating intermediate results, which are used by others.

Depending on the way multiple tasks access the shared data, different data dependences are used:

**read-only** multiple reader tasks may access shared data simultaneously.

**write-only/read-write** only a single task is allowed to write.

**concurrent** relaxes read-write restriction by allowing multiple tasks using concurrent access type to run simultaneously. The user is responsible for correctness of the results.

**commutative** it relax read-write restriction by allowing multiple tasks using this access type to run out of order.

All of these dependence types are defined using a list of memory references. These memory references are used to dynamically compute the task dependence graph during the execution of the program. The edges of this graph define how tasks synchronize with each other.

Building the dependence graph at runtime allows applications to produce a different dependence graphs depending on the input data of the program.

1.2.2 Task scheduling points

During a task execution, OmpSs allows computing resources (i.e. threads) to stop their execution in favor of other tasks. This may only take place in specified contexts known as scheduling points, which include:

- The generation of a new task.
- The completion of the task.
- Reaching a **taskwait** directive.
- Reaching a **taskyield** directive.

The fact of switching one task’s execution with another’s is known as task switch. It may imply starting the execution of a non-previously executed task or resuming the execution of a partially executed task.

OmpSs allows the user to explicitly insert task scheduling points by means of **taskwait** and **taskyield** directives.
Figure 1.4: MPI point to point messages as inter-process extension of OmpSs data dependences.

**taskwait** Blocks the execution of the task until all its sibling tasks have completed their execution.

**taskyield** The task execution is suspended in favor of another. The task is not blocked, so it may be immediately resumed.

### 1.3 Hybrid parallel programs

Hybrid applications mix several parallel programming models altogether with the goal of achieving better performance or increased efficiency. For example, one of the most popular compositions consists on mixing message passing and shared memory programming models. There are two principal reasons why this combination may benefit an application. Shared memory parallel programs can not scale beyond the node level in distributed memory systems. On the other hand, distributed memory parallel programs usually have higher memory requirements and synchronization overhead for processes sharing the same node, in comparison to a shared memory equivalent (shared data is not replicated inside a process).

Unfortunately, even though hybrid applications are common in HPC, they typically use one or another model depending on the phase of the algorithm. For example, using OmpSs for computation intensive parts and MPI for communication parts, where each part is synchronized in such a way that they can not overlap.

In addition, MPI and OmpSs models are totally decoupled. OmpSs tasks are not aware of whether a MPI operation will block a thread or not. Likewise, MPI libraries are not aware of whether MPI operations called in an application are part of OmpSs tasks. The main reason is that these two programming models focus on exploiting parallelism at a different level: MPI simplifies communication between processes in programs with multiple processes (inter-process parallelism) while OmpSs exploits parallelism within a process. Mixing both in the same application is tricky, as features of both languages might easily interact in an unexpected way, resulting in dead-locks, incorrect results, fatal errors or performance issues.

#### 1.3.1 MPI blocking communications and task-based parallelism

The user can extend OmpSs data dependences in hybrid MPI+OmpSs programs by placing MPI blocking communication routines inside a task. This will allow the program to synchronize task execution beyond the process level, since a task will not finish until the communication is completed, as seen in figure 1.4.

Even though it is possible to completely wrap MPI routines with OmpSs tasks, the user should be aware that doing so may introduce the possibility of deadlock: if all the threads are dedicated
to run tasks with MPI blocking routines, in such a way that the global progress of the application does not advance, the blocking routines will wait permanently.

Code 1.4: Alternative implementation using non-blocking MPI routines and taskyield.

```c
int buffer[SIZE];
#pragma omp task out(buffer)
{
    int flag = 0;
    MPI_Request req;
    MPI_Irecv(buffer, SIZE, MPI_INT, source, MPI_COMM_WORLD, &req);
    do {
        MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
        if(flag == 0)
            #pragma omp taskyield
    } while(flag == 0);
}
```

One alternative is to replace blocking routine calls in the application with their respective non-blocking version, as in the example shown in code 1.4. Meanwhile the communication is not completed, the task pauses its execution with a taskyield clause.

The main drawback of this technique is that modifying every single MPI routine in big applications is cumbersome. In addition, task execution can be resumed even though the communication they are waiting for has not been completed yet, thus producing unnecessary task switches.

The following chapters present the design and evaluation of the OmpSs-MPI interoperability library, which is conceived to efficiently address these problems, allowing the use of blocking communication routines in OmpSs tasks.
Chapter 2

Implementation

The goal of the OmpSs-MPI interoperability library is to support MPI blocking routines inside tasks, such that no deadlocks are produced no matter how many tasks with communication routines are generated.

Therefore, the library will provide an alternative implementation for MPI point-to-point and collective blocking routines (figure 2.1). Thanks to the MPI profiling interface, it can intercept these routine calls done by applications. Once a blocking communication routine has been called, we start the communication using its respective non-blocking version and immediately check whether or not the communication is completed. In case it is not, the task execution is stopped (code 2.1).

Applications that already make use of MPI non-blocking routines can also be supported. Non-blocking routines such as MPI_Isend or MPI_Test need not to be intercepted because they do not produce any means of synchronization. However, it is very common to use MPI_Wait family of functions with non-blocking communications. These functions are indeed blocking, but they can also be intercepted and replaced with their corresponding non-blocking MPI_Test version, in a similar way it is done with blocking communications (code 2.2).

Moreover, the library takes advantage of the MPI_Pcontrol function, to allow the user to selectively disable the task switch produced by pending MPI communications. Applications with bursts of short communications may benefit from better response times if the task execution is not stopped

![Diagram](image)

Figure 2.1: The new library provides alternate implementations for MPI blocking communication routines, but leaving other MPI and OmpSs operations as-is.
in this case.

**Code 2.1:** MPI\_Send transformation using MPI profiling interface. No PMPI symbols are used because the call to the operation is replaced with the non-blocking version.

```c
int MPI_Send( const void *buf, int count, MPI\_Datatype datatype,
             int dest, int tag, MPI\_Comm comm )
{
    int err, flag;
    MPI\_Request req;
    err = MPI\_I\_send( buf, count, datatype, dest, tag, comm, &req );
    err = MPI\_Test( req, &flag, MPI\_STATUS\_IGNORE );
    if( flag == 0 ) {
        // - Save the request
        // - Stop task execution
    }
    return err;
}
```

**Code 2.2:** MPI\_Wait\_all transformation using MPI profiling interface.

```c
int MPI\_Wait\_any( int count, MPI\_Request array\_of\_requests[],
                    MPI\_Status array\_of\_statuses[] )
{
    int err, flag;
    err = MPI\_Test\_all( count, array\_of\_requests,
                          &flag, array\_of\_statuses );
    if( flag == 0 ) {
        // - Save all pending requests
        // - Stop task execution
    }
    return err;
}
```

The examples shown so far are uncompleted: some key parts of the implementation are missing. Unfortunately, it is necessary to solve the following problems before the library is fully functional:

- MPI standard does not have a callback mechanism to notify when a communication is completed. It is necessary to check pending requests’ completion status from time to time.

- OmpSs programming model does not include a mechanism to explicitly stop task execution. The only task scheduling points available to the user are `taskyield` and `taskwait`, which do not provide this functionality.

As a result, we propose the following two extensions to OmpSs programming model:
1. A synchronization mechanism that allows blocking indefinitely and releasing the execution of a task.

2. A polling mechanism that performs periodic calls to one or multiple arbitrary functions.

These mechanisms will allow us to periodically check the completion state of all pending communications. In the case one or multiple of these communications are completed, it will be possible to resume the execution of the tasks that were waiting for them.

2.1 External explicit synchronization of task execution

Using a task scheduling point such as *taskyield*, as previously proposed in code 1.1, is not optimum enough. Even though it allows the execution of other tasks, a task remains ready (eligible for execution by any thread) after the task switch. If its execution can not make progress (some requests are still pending) it might cause additional unnecessary task switches: a thread resumes the task, checks the completion status and performs a task switch again.

It is possible to produce a much more efficient implementation if part of the mechanism that control changes between ready and blocked states, is exposed through an interface. We call this mechanism *task condition variables*, because their functionality and interface (proposed in code 2.3), resembles to condition variables in POSIX threads library.

It is important to understand that blocking a task’s execution does not cause the underlying thread to block: a task does not make any further progress but the thread is allowed to start or resume other available tasks. In this regard, the mechanism is similar to a *taskwait* mechanism, with the difference that the task is explicitly set to ready, instead of depending on the amount of sibling tasks remaining. This means that programs such as the example shown in code 2.4 can even run with a single thread.

Code 2.3: OmpSs task condition variable interface.

```c
// 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 );
```

15
Code 2.4: Task condition variables allow explicit synchronization without blocking the execution resources.

```c
int value = 0;
nanos_wait_cond_t cond;
nanos_create_wait_condition(&cond);

#pragma omp task shared(value, cond) label(waiter)
{ // Wait until value is equal to 1
    while (value != 1)
        nanos_block_current_task(&cond);
}

#pragma omp task shared(value, cond) label(signaler)
{ // Make waiter task ready again
    value = 1;
    nanos_signal_wait_condition(&cond);
}
#pragma omp task
```

### 2.2 External function polling mechanism

MPI request handling imposes the necessity of a mechanism that allows periodic check of a specific condition. In this case, the check consists on querying the completion state of all pending requests which tasks are waiting for. Furthermore, this simple test if something is satisfied functionality is present in other libraries (e.g. `clGetEventInfo` in OpenCL). Designing an abstract mechanism makes its future extension to other libraries easier.

We expect that the polling functionality, initially required by OmpSs-MPI interoperability library, has multiple potential uses in the future. Therefore, we propose an extension to the OmpSs runtime library that provides polling functionality, targeting external gasket libraries.

External libraries or applications need to provide all necessary functions and arguments that need to be called periodically. This is done through a common interface such as the proposed in code 2.5. This way, OmpSs runtime library can store each combination of functions and an arguments, what we call polling services. It is allowed to specify the same function multiple times, which enables calling the same function with different arguments.

At any unspecified point in time, OmpSs runtime library will call the services’ function with the argument that we initially specified when registering the service. Service function arguments are defined as opaque pointer types, which give enough flexibility to pass any type of parameters (see code 2.6). In addition, service functions will return a logical integer number. If the return value is exactly 1, then the service will be unregistered: function and argument combination will never be called again.

---

1[https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/clGetEventInfo.html](https://www.khronos.org/registry/OpenCL/sdk/1.1/docs/man/xhtml/clGetEventInfo.html)
Code 2.5: OmpSs polling interface.

```c
// Function that will be periodically called.
typedef int (*nanos_polling_function_t)(void *service_data);

/* Registers a function and an opaque parameter that must be
periodically called.
* Using the same function with different parameters is allowed.
* The function is never called again if it returns a
* logical 'true' (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);
```

Code 2.6: Using polling interface to register multiple polling services.

```c
// Function that will be periodically called.
int myfunc( int * args ) {
    if( *args == 0 ) {
        printf("The condition is now satisfied!\n");
        return 0;
    }
}

int args_1 = 1, args_2 = 2;

int main() {
    nanos_register_polling_service( "Polling test",
        myfunc, &args_1 );

    nanos_register_polling_service( "Polling test",
        myfunc, &args_2 );
    // ...
    args_2 = 0;
    // ...
    return 0;
}
```
2.3 Putting it all together

Now that the extensions to Ompss programming model are available, task execution can stop whenever it reaches a blocking communication routine. Figure 2.2 shows how the procedure looks like for the particular case of `MPI_Send`.

Moreover, the polling mechanism is used to periodically check for the completion status of the pending communications. A polling function is registered by the time MPI is initialized (during `MPI_Init` or `MPI_Init_thread`) and unregistered before it is finalized (`MPI_Finalize`). Whenever MPI informs that some communications are completed, the polling function will be responsible of resuming the tasks waiting for them (figure 2.3).

There are some details that must be taken into consideration:

- Some tasks may wait for more than one MPI request object simultaneously. For example, using `MPI_Waitall`, which waits for multiple requests at once.
- Some tasks may not be eligible for blocking, due to the usage of `MPI_Pcontrol` function.
Therefore, the library stores additional information related to how each task waits:

- A task condition variable that allows blocking/resuming a task.
- A counter of the remaining operations before the task can be resumed.
- A flag indicating if the task can become blocked (due to `MPI_Pcontrol`).
- A flag indicating whether the task is waiting or not. Functions such as `MPI_Waitany` may allow the task to resume even though not all requests are completed.
- A pointer to output status argument passed by the user in calls such as `MPI_Recv`. If necessary, output status is copied from the result of `MPI_Testsome` to the array of statuses that the user has specified.

Whenever MPI informs that a communication request is completed, we decrease its corresponding task counter. When the counter reaches 0, the library signals the task condition variable, returning the waiting task to a ready state.
Chapter 3

Evaluation

3.1 Performance metrics

Applications and tools in HPC aim to achieve the best performance possible, using the available hardware platforms. Therefore, its measurement is of vital importance. This section briefly explains the most important metrics used in this work, and what kind of behavior they try to expose.

Latency

Latency measures the amount of time spent by a given program to solve a problem. It can cover the execution of the whole program or only the portions under analysis. Latency is also referred to service time, execution time or just time.

Latency is obtained by taking two clock samples: one at the beginning of the region of interest, and another at its end.

\[ T = t_{\text{end}} - t_{\text{start}} \]  \hspace{1cm} (3.1)

Throughput

Throughput measures the rate of production of a given system. This means that systems with higher throughput are able to solve big problems faster.

Depending on the system we are to analyze, we compute throughput differently. For example, productivity in a processor is not comparable to that of a network interface card. In computer architecture specifically, we use different metrics depending on what kind of analysis we are interested in.

IPC measures the mean number of instructions executed per cycle. This is useful to compare different processors with an equivalent instruction set, while their working frequency, which impacts on how fast the work, changes continuously during an execution, so computing the throughput in base of time would not be possible if, for example, we want to evaluate their instruction-level parallelism.

\[ IPC = \frac{\#\text{instructions}}{\#\text{cycles}} \]  \hspace{1cm} (3.2)
On the other hand, FLOPs (floating point operations per second) measures the mean number of floating points executed in a second. It measures the amount of floating point instructions performed by some processor or system in a second. This metric is really important in HPC, as scientific applications perform most of their calculations with floating point numbers. In addition, this metric also allows us to compare different processors when their instruction sets are not comparable. For example, SIMD units in modern processors may execute different amounts of operations per cycle, so that one instruction in one processor actually performs more work than one in another.

We can compute FLOPs measuring the number of floating point operations a given program performs to solve a problem, and how much time it took to solve it.

\[
FLOPs = \frac{\#\text{operations}}{T} \quad (3.3)
\]

**CPU load** is the ratio of useful computational work performed by a CPU. In contrast to IPC and FLOPs, this metric is mostly used from the point of view of applications and the operative systems, where it is possible to identify whether or not the processor is really working on the parts of computation we are interested in.

We use CPU load to analyze how the performance of a system is affected by loading imbalance and synchronization, as some processors remain idle waiting for more work to become available. We measure CPU load as the ratio between the amount of effective working time and the total time of the region of interest.

\[
L = \frac{T - T_{\text{idle}}}{T} \quad (3.4)
\]

**Speedup**

We call speedup to the ratio of two latency values. It measures the improvement that involves using one process instead of another to solve the same problem. It is specially useful to evaluate the impact of changes made in programs or the system.

\[
S = \frac{T_{\text{old}}}{T_{\text{new}}} \quad (3.5)
\]

Speedup is also used to measure the benefits of running parallel programs with an increased number of processors. Equation 3.6 computes the speedup obtained of using \(n\) processors in comparison to the sequential execution (a single processor is used).

\[
S(n) = \frac{T(1)}{T(n)} \quad (3.6)
\]

Speedup is one of the most popular metrics in HPC. It is possible to theoretically approximate it, so that we get an idea of how fast it could become without needing to spend time and resources to make the measurements.

**Amdahl’s law**

Amdahl’s law is a theoretical approximation of the speedup expected from an application when the amount of resources changes. Equation 3.7 computes the theoretical speedup for a program using \(n\) processors, where \(p\) is the parallel portion of the application. This approximation sets an
Amdahl’s law is limited in the sense that the approximation is only valid for programs processing the same workload, that is, solving very same problem.

Gustafson’s Law

Gustafson’s law is based on the concept that larger problems can be solved in the same amount of time when higher number or better resources becomes available. It address the limitations of the problem size in Amdahl’s law. Equation 3.8 predicts the speedup of an application for a amount of resources \( n \), where \( p \) is the parallel portion of the application related to the workload (not execution time).

\[
S(n) = 1 - p + np
\] (3.8)

Scalability

Scalability in parallel programs is known as the ability to work efficiently when the problem size or the amount of resources it uses becomes larger. There are two types of scalability:

1. Strong scaling: analyzes how much the latency is reduced with a greater number of processors using a fixed problem size.
2. Weak scaling, how the latency varies with an increasing number of processors and fixed problem size per processor.

This work tries to study the effects on applications that our proposal has, in terms parallelism and efficiency. Therefore, strong scalability is specially interesting because these two terms have a big impact on it.

Parallel efficiency

We use parallel efficiency to measure the overhead of a parallel application when additional compute resources are used. It basically analyses the gap between the application speedup with respect to the ideal one.

\[
E(n) = \frac{S(n)}{n}
\] (3.9)

For example, if an application halves the execution time when we double the amount of processors, its parallel efficiency is 1, meaning that there is no apparent overhead that limits the scalability.
3.2 Evaluation environment

In this section, we present the properties of the systems we have performed our tests and the software tools that we have used.

3.2.1 Marenstrohm 3

IBM dx360 M4 compute nodes with the following technical characteristics:

- 2 × Intel Xeon E5-2670 processor
  - Frequency: 2601 MHz
  - Hdw Threads per core : 1
  - Cores per Socket : 8
  - Sockets : 2
  - Total CPUs per node : 16
  - L1i/d cache: 32K
  - L2 cache: 256K
  - L3 cache: 20480K

- Main memory: 32 GB

- 2 × Mellanox FDR10 Infiniband cards
  - Bandwidth: 56Gbit/s
  - Data rate: FDR10

- OS: SUSE Linux Enterprise Server 11.3
  - GNU/Linux 3.0.101-0.47.90

3.2.2 Minotauro M2090

Bull B505 blades with the following technical characteristics:

- 2 × Intel Xeon E5649 processor
  - Frequency: 2527 MHz
  - Hdw Threads per core : 1
  - Cores per Socket : 6
  - Sockets : 2
  - Total CPUs per node : 12
  - L1i/d cache: 32K
  - L2 cache: 256K
  - L3 cache: 12288K
• Main memory: 24 GB
• 2 × Mellanox Technologies MT26428 Infiniband cards
  – Bandwidth: 40Gbit/s
  – Data rate: QDR
• OS: Red Hat Enterprise Linux Server release 6.7 (Santiago)
  – GNU/Linux 2.6.32-642.6.2

3.2.3 Software
Compilers and libraries:
• GNU Compiler Collection 6.2.0
• Intel Math Kernel Library 2017.1
• Intel MPI 5.1.3.181
• Mercurium compiler 2.0.0
• Nanos++ runtime library v0.12a

Performance analysis tools:
• PAPI 5.5.1
• Paraver 4.6.3
• Extrae 3.4.3

3.3 IFSker
IFSker, for IFS kernel, is a mock-up application written in Fortran and parallelized with MPI. It models the communication and computational patterns that take place in a meteorological forecasting model known as Integrated Forecast System (IFS). IFS is developed and maintained by the European Centre For Medium-Range Weather Forecasts (ECMWF).

The goal of this kernel is to explore the effects of task based parallelization on its scalability, without having to deal with the size and complexity of the real source code. In addition, the code is proprietary and not publicly distributed.

IFS employs a spectral transform method. As opposed to a grid-point model, where a field is represented directly by its value at discrete positions in a grid, spectral methods represent fields using a set of coefficients of a basis function (e.g. a sine function). Spectral methods achieve higher accuracy in partial differential equations, due to the fact that the derivatives of the basis function are well known (in the case of the sine, its derivative is the cosine).

The algorithmic structure consists in time step cycles mainly divided in two phases: grid-point physics computations and Fourier transforms. Data representation and distribution is stage dependent.
Parallelization and partitioning are driven by the independent dimensions of the data in each specific stage. Therefore, communication between processes takes place only during the transitions from one stage to the following, where the data needs to be transposed and redistributed (figure 3.1).

Stage transitions not only involve data redistribution but also its transposition. This is performed by pack and unpack tasks. A pack operation copies the data into contiguous memory, ready to be sent, meanwhile an unpack operation copies it from the reception buffer into its final position.

Each process makes the following steps during one time iteration:

1. Norm computation for all parameters in all processes.
2. Grid-point physics computation.
3. GP to FFT transition. For each latitude:
   - Send latitude to the process which belongs to in FFT stage.
   - Receive latitude from the process which belongs to in GP stage.
4. FFT computations.
5. FFT to GP transition. For each peer process:
   - Pack all data that belongs to that process in GP stage.
   - Send the data to that process.
   - Receive the data that belongs to this process.
   - Unpack the data onto the GP stage data structure.

At the beginning of each time iteration, IFSker performs a control of the results obtained so far. This mainly consists on computing the maximum, minimum and average values for each grid-point parameter used in the model, which is known as the norm. The goal of the norm is allowing the user a means to verify results over time.

IFSker originally computes the norm of parameters by gathering all their values in a single master process. Afterwards, the master process computes the maximum, minimum and average values. This is done for each grid-point parameter.

The following sections analyze performance issues in IFSker and how we have improved the application to solve them. All the executions were performed in Minotauro system, detailed in section 3.2.2.
3.3.1 Analysis

IFSker baseline parallelization is very restrictive: there are multiple parts of sequential code and a synchronization barrier takes place always at the end of each phase, leaving no possibility for any overlap.

This excess of synchronization limits the parallelism, specially in situations of load imbalance. Figure 3.2 shows how the uneven distribution of physics computations produces some processes to wait for others. In addition, at the beginning of each time step, the computation of the norm which is not parallel at all, further delays the execution of the program.

The irregular distribution of the data produces imbalance during physics computation phase. In figure 3.3 processes $P_0$ and $P_3$ execute tasks with greater variation in granularity. Fourier computation, on the other hand, is clearly much more regular. The ratio between the biggest and the smallest task duration is reduced, in comparison, by an order of magnitude.

A closer look to the graph of task dependences (figure 3.4) reveals that tasks with MPI communications are serialized (they can not be reordered or executed in parallel because they are tied with a dependence edge). This serialization is forced to avoid deadlocks.

Our parallelization strategy, therefore, relies on reducing the synchronization granularity, by removing all dependences between communications tasks and ensuring that the application is able to run without any kind of barrier (i.e. do not use any taskwait, but rather only use data dependences as a mean of synchronization).
Figure 3.3: Task execution time distribution by MPI process. Each point represents the execution time of one task. physics tasks duration (left) has much more variability for some processes in comparison to others. In contrast, fft tasks (right), have a similar load distribution across processes.

Figure 3.4: Baseline task dependence graph for a single time step using two processes. The number of tasks is reduced with respect to a real execution to improve readability.
3.3.2 Improvements

During the analysis of the application, we have found several caveats that limit both the parallelism and efficiency of the application. This section focuses on addressing these limitations in the following steps: communication concurrency, pack and unpack operations, collective communications and synchronization granularity.

OmpSs-MPI interoperability library proposed in this work allows to safely wrap MPI blocking calls into tasks without any risk of deadlock, so it is no longer necessary to serialize MPI communication routines. However, the fact that now multiple messages can be sent and received concurrently, requires some effort to keep the correctness of the program.

When the data is being redistributed, multiple messages can be exchanged between the same pair of source and destination processes (figure 3.6). IFSker implementation originally used the same tag for all these communications, which would produce errors if those communications are performed no matter if they are parallel or out of order. Therefore, it is necessary to uniquely identify every message transmitted, so that the receiver knows where to place the data. To ensure this identification, we must make sure that the message envelope (the combination of source, destination, tag and communicator) is not repeated in potentially concurrent messages. We tag each piece of data being sent using its position. In addition, communications taking place in different transitions use different communicators, so that is possible simultaneously send or receive messages of different stage transitions without trouble.

![Figure 3.5: Using different communicators for each stage transition allows easier identification of the messages, and avoids crosstalk due to tag conflicts.](image-url)
Figure 3.6: Redistribution of the data involves multiple communication messages between same source and destination processes. In this example, $P_3$ receives two messages from $P_2$: $(x_2, y_1)$ and $(x_2, y_2)$. The order $P_2$ will send the messages is not deterministic, so it is necessary to properly identify which message is received. This way, $P_3$ can place the data into its proper place.

Splitting communications in smaller pieces during a stage transition not only allows overlapping computation and communications, but also allows some processes to start as soon as their data is available, regardless of how much remaining work other processes still have. In Figure 3.6, process $P_4$ can start Stage B once blocks $(x_2, y_1)$ and $(x_2, y_2)$ have been received, even if $P_2$ has not completed $(x_1, y_1)$ and $(x_1, y_2)$.

Pack and unpack tasks can be completely removed from the application. Instead of using auxiliary buffers and explicitly moving the data into a contiguous memory layout, it is possible to specify the data layout in memory through MPI datatypes. MPI library, guided by datatype information will be able to find the right place for the data, so that it is sent without any additional copies.

Since IFSker uses different grid-point representations, depending on stage of the algorithm, we define one MPI datatype for each of those stages. Communication routine calls become much more simple and pack/unpack tasks are no longer necessary.

We create four different datatypes to define how a single grid-point is represented in each stage of the application:

1. A datatype for real numbers, which is usually MPI_REAL or MPI_REAL8 datatypes. We construct it using MPI_Type_create_f90_real, to ensure that the MPI type meets the precision requirements of IFSker floating point values.

2. A grid-point in physics stage. Grid points are stored in a points $\times$ parameters matrix, where each point is contained in a column. Values in a column are contiguous in Fortran programs, so we construct the type using MPI_Type_contiguous construction function.

3. A grid-point in Fourier transforms stage. The points $\times$ parameters matrix is transposed in this stage, so each grid-point is now stored in a row. Grid-point parameters can be accessed with a constant stride. This type is constructed in MPI using MPI_Type_vector.
MPI_Type_vector memory layout with blocks = 3 and stride = 3:

How MPI access buffer values in a communication with count = 2:

Memory layout when the datatype extent is redefined:

Figure 3.7: Memory layout of the new defined types. It is necessary to redefine the extent of the vector datatype when it is used in communication operations with count > 1.

4. A helper type that with a redefined distance between elements (known as extent), which allows MPI communication operations to transfer several rows at once. Using the vector type as-is is incorrect, as they do not overlap by default (see figure 3.7).

Norm computation is serialized in the baseline version. It originally computed the norm of a field parameter by gathering all its values in a single root process. This root processes then computes the maximum, minimum and average values for that parameter. This is repeated for each grid-point parameter.

However, it is way more efficient to compute the norms using MPI_Reduce collective reductions. Instead of transferring all the values, only the local reduction result is sent (in the case of a maximum reduction, each process only sends its local maximum to the root). This has two big advantages: it reduces the network traffic and performs the local part of the reductions parallel at MPI process level. Maximum, minimum and sum are part of the reduction operators defined in MPI standard. Unfortunately, these operators can only be used one at a time.

In order to achieve the maximum efficiency, we improve the norm computation by the definition of a MPI custom reduction operator, which performs all reduction operations at once, reducing the number of collective communications to only one. This is possible because MPI_Reduce is able to perform multiple independent reductions in the same collective, thus reducing all parameters simultaneously.

The custom reduction operator is defined at the beginning of the application. First, we implement the logic of the reduction in a new function. The function generates global maximum, minimum, total and count values from the local values received from each process (total and count values are used to compute the average). The next step is defining a custom MPI datatype that contains those four values and a custom MPI reduction operator that uses the function implemented in the first step. Finally, during the reduction stage, it is only necessary to compute the local reduced values (local maximum, minimum, sum and count values) and to start the collective operation.

Along with the increase of communication and computation efficiency, the new implemented reduction opens the opportunity for additional parallel code: we place local reductions inside tasks, so that the local reduction is performed step by step as soon as it becomes available. Once each step completes, the physics computation task of next time iteration becomes ready. This was the last milestone before every coarse grain synchronization is removed: OmpSs dependences and MPI messages have turned the application into a pipeline (see figure 3.8).
Figure 3.8: Final version task dependence graph (a single MPI process), which contains three time steps. Task dependences avoid the need for synchronization barriers, which reduces the synchronization granularity.
3.3.3 Results

Norm computation really benefited from merging all \texttt{MPI	extunderscore gather} calls to a single, user defined, \texttt{MPI	extunderscore reduce}. This did not only help to relax yet another process barrier, but also reduced its computational and communication cost, as seen in figure 3.9.

![Figure 3.9: Time spent in norm computation (all processes) for an execution with 10 time iterations. User defined MPI reductions obtain the same result but using less network bandwidth and CPU time.](image)

However, most of the original performance problems in IFSker come from excessive synchronization and load imbalance. Transforming synchronization barriers into fine grained task dependences made possible for some MPI processes to make further progress, even though others had not finished with the current stage. For example, MPI processes no longer execute FFT stage simultaneously, as can be seen in figure 3.10, but they rather do so as soon as the data they depend on arrives (Physics-FFT \texttt{Recv} tasks complete, and their successors become satisfied). This turns into more stable CPU loads across all the execution (see figure 3.11): communication does not immediately pauses the execution but rather progresses in the background, meanwhile the algorithm makes progress on independent work.

Consequently, the parallel efficiency of the algorithm does not suffer from a higher use of resources. Figure 3.12 demonstrates that reducing synchronization and overlapping communications with useful computation, is key to maintain an efficient usage of the resources. Even though the baseline version is already parallel, we achieve a speedup of nearly 2x running with 8 nodes and 12 threads per MPI process.
Figure 3.10: Execution diagram for the final version, showing the same time window as in figure 3.2. The amount of time the processes remain idle is reduced, which contributes to finishing the execution faster.

Figure 3.11: Comparison of CPU load between the base case (left) and improved (right) versions.
Figure 3.12: Improved version achieves a much better parallel efficiency than the baseline, so it performs much better when the number of resources increases (strong scaling).
3.4 High performance Linpack benchmark

HPL is a software package that solves a random-generated dense linear system on distributed-memory computers using. It is an open source MPI implementation of the High Performance Computing Linpack Benchmark.

The benchmark was introduced by Jack Dongarra in 1979, and it is used to rank the most powerful computers in the world in a list known as TOP500. As it happens with every benchmark, it does not perfectly reflect the performance of a given system, but it makes a good approximation of its peak performance with floating point operations. The algorithm and its implementation are scalable in the sense that their parallel efficiency is maintained constant with respect to the per processor memory usage (weak scaling).

Most of the matrix operations in HPL use a de-facto standard interface for specialized mathematical libraries, known as Basic Linear Algebra Subprograms (BLAS). Most hardware vendors offer a software package with an implementation of these libraries that is designed and tuned to achieve the best performance on their platforms.

The benchmark has the following properties:

- Two-dimensional block-cyclic data distribution.
- Right-looking variant of the LU factorization with row partial pivoting with multiple lookahead depths.
- Recursive factorization with pivot search and column broadcast combined.
- Various broadcast topologies.

The HPL benchmark implements an LU decomposition with partial pivoting. The matrix to be factored has $N \times N$ elements (double-precision floats) and it is decomposed into blocks of size $NB \times NB$, which are distributed onto a grid of $P \times Q$ processes. The blocks are distributed among processes in a cyclic way to balance load, as seen in figure 3.13.

The LU factorization is done by iteratively. Each iteration is divided in two big steps: panel factorization and trailing submatrix update. Once the current panel is factorized (producing $L_1$ and $L_2$ in figure 3.14), it is forwarded to other processes in the same process row using a broadcast operation.

The broadcast operation is implemented in several point-to-point based algorithms. Another option is to use $\text{MPI}_{-}\text{Bcast}$ operation. This may be interesting in case the MPI library provides

![Figure 3.13: Data distribution in HPL for a matrix of $8 \times 8$ blocks, distributed among processes in a two dimensional $P \times Q$ grid, where $P = 2$ and $Q = 3$.](image-url)
a better implementation, which may be tuned to the particular system’s interconnection network where it is installed.

Consecutive iterations come after the completion of the trailing submatrix update. Despite the fact that the factorization does not start until the whole matrix is updated, it is actually only dependent on the next column panel: it can start as soon as this panel is updated, regardless of the rest of the columns. In fact, HPL also implements an alternative algorithm using look-ahead. This technique allows some panel factorizations to start as soon as their respective columns are updated, accelerating the execution of the critical path. Since non-critical updates are postponed, the algorithm can keep on updating the matrix meanwhile it waits for broadcast completion, enabling some overlap between computation and communication.

The trailing submatrix update repeats the row exchanges performed in the panel during the factorization, so that the matrix is kept consistent. Each process communicates with others inside the same grid column \( Q \) (see figure 3.15). Finally, the trailing submatrix is updated using a matrix multiplication between \( L_2 \) and \( U \) and subtracting \( A \) to its result. All processes contain the parts of \( L_2 \) and \( U \) that matches their position in the process grid. \( L_2 \) is transferred during the panel broadcast and \( U \) is distributed as a result of update’s pivoting phase.

Once the matrix is completely factorized, the linear system is solved using backwards substitution.

HPL offers a good case of study for several reasons. First, it contains multiple communication patterns. Second, parallelism and computation/communication ratio changes with time, so it is also challenging from the scheduling point of view.

All results analyzed in the following sections were obtained from executions in Marenostrum 3 system, detailed in section 3.2.1.

3.4.1 Analysis

HPL benchmark is a pure MPI implementation. The main flow of the application is single-threaded. It relies on BLAS libraries’ implementation to provide parallel execution at the thread level. Each time a BLAS routine is called, the library splits the work among all the available threads. Afterwards, when the routine completes and before the library returns the control to the application, all threads are synchronized together. This multi-threading model is known as fork/join: a master thread executes the main part of the application, splits the work when it reaches a compute
intensive part of the application (fork), and synchronizes all the threads at the end (join).

This fork join model produces spikes of CPU load, which drops during synchronization and sequential parts of the application, as seen in figure 3.16. Note that for a small problem size such as the execution shown in the figure, CPU load does not exceed 80% in average at any point of the execution. Furthermore, the parallelism decreases towards the end of the algorithm, which makes difficult to keep all the cores busy. The reason behind this is that the size of the trailing matrix (which gets updated after every block factorization) decreases with each iteration of the main loop.

When solving bigger problems, the amount of work performed by each linear algebra operation increases, reducing the impact of the synchronization and sequential parts of the application. Figure 3.17 shows how \texttt{dgemm} function tends to take most of the application’s total execution time when the problem size increases. The reason behind this is that this operation’s time complexity is cubic (doubling the problem size increases its execution time 8 times).

Therefore, when Linpack is used to benchmark a new system’s performance, typical configura-
Figure 3.17: Time spent in \texttt{dgemm} for an execution of 4 single-threaded MPI processes. This trend is repeated for larger number of processes using bigger problem sizes.

Figure 3.18: HPL execution time and throughput, using 4 MPI processes and 8 threads. Bigger problems help to make a better approximation of the system peak performance. However, the time necessary to solve them grows quickly.

...tions use as much memory in each compute node as possible, leaving some memory for the operative system. As a consequence, the total time consumed by the benchmark can take several hours to complete. For example, figure 3.18 shows that even for a relatively small problem size, it already takes nearly a minute using two compute nodes to reach a moderate performance (70% of theoretical peak performance).

Improving the efficiency on smaller problem sizes will allow us to reach asymptotic maximum performance earlier, reducing the required execution time for the whole benchmark. Therefore, instead of exploiting thread-level parallelism inside the BLAS library, we aim to develop a task-based parallel implementation based on the version that we have just analyzed.

3.4.2 Task-based parallelization

The goal of the task-based hybrid HPL is to accelerate the computations that are either on the critical path or close to it. From now on, thread-level parallelism is provided in the application, so multithreading BLAS is not used from now on.
Figure 3.19: Simplified task dependence diagram for two consecutive iterations using two MPI processes ($P = 1$ and $Q = 2$). The critical path is highlighted with red edges. On the left, the matrix is partitioned in blocks of columns that are processed by different tasks. Columns with red color are closer to the critical path.

The same matrix partitioning (blocks of $NB$ columns) is kept. Each block of columns is assigned to each task. Since trailing matrix updates of consecutive iterations of the algorithm overlap, tasks are synchronized using data dependences: the addresses of the columns are registered by OmpSs runtime library during the execution, which generates a dependence graph similar to the one shown in figure 3.19. In addition, we assign a priority to each task as a scheduling hint. The closer a task is to the critical path, the higher priority it has. Higher priority tasks usually spend less time waiting on ready queues before they start running.

It must be noted that some parts of the algorithm exchange messages with other MPI processes (e.g. pivoting during factorization and update). We leverage OmpSs-MPI interoperability library functionality to overlap communication of these tasks with the execution of others. To ensure point-to-point message correctness, concurrent point-to-point communications that share the same communicator use unique tag values. These values are generated with equation 3.10, where $code(op)$ is a unique identifier for each operation (e.g. factorization, update, etc.) and $j$ is the block index in the local trailing submatrix of each process. For example, messages in tasks $A6$ and $A8$ are tagged with the values $Tag(pivoting, 0)$ and $Tag(pivoting, 1)$.

$$Tag(op, j) = code(op) + j \tag{3.10}$$
Look-ahead is natural to OmpSs programming model. The main task creates and computes the dependences of all tasks until the end of the algorithm is processed. It is only necessary to ensure that the communications and auxiliary data structures used by HPL are not used by operations of different iterations simultaneously. This is solved by the use of a multiple buffering technique: we allocate several copies of these structures and assign them to iterations in a cyclic way. All the communicators are duplicated as well. Consequently, there is no risk of collision between messages of consecutive iterations (e.g. $A_6$ and $A_{12}$): their message envelopes will never match due to the usage of different communicators.

3.4.3 Results

The current implementation as-is is unfortunately not enough to produce an efficient parallelization of the problem. Figure 3.20 shows that, despite the look-ahead technique and the prioritization tasks in the critical path, their execution is postponed until the end of the update. This limits the amount of parallelism, which makes most of the threads to wait until the end of the iteration.

OmpSs-MPI interoperability is conceived to take advantage of the wasted time that threads spend on blocking communication routines. This is specially useful when delaying their completion slightly does not affect the overall performance of the application. However, if communication tasks are located at the critical path, priority inversion is possible: coarse grain low priority tasks can take their place. In this case, high priority tasks will not be able to go back to execution until a task switching points is reached.

Figure 3.21 shows the average duration of the tasks and the maximum number of times they are replaced by others, as a result of unfinished communications. Note how the most critical tasks (factorization and pivoting tasks) suffer most of the task switches. On the other hand, update tasks have the longest duration and do not produce any task switch. In the particular case of the factorization, we can see from the previous execution diagram (figure 3.20) that it is composed by the burst of very short communications that takes place during row exchanges.

In order to avoid priority inversion, we design a new scheduling policy for OmpSs runtime library. The new scheduler specializes one worker thread, which focuses on execution of high priority tasks. We call it communications thread, as we will devote it to execute most of the tasks with communications, including the factorization. In addition, we improve the response time of factorization tasks by disabling OmpSs-MPI interoperability features (with MPI_Pcontrol) during the communication burst and by increasing the parallelism on the column updates they depend on (we call it urgent_dgemm, whose computation is split in multiple row blocks). With this
optimizations and the new scheduling policy, the resource usage improves (see figure 3.22).

Even though it looks like the communications thread generates inefficiencies (there are gaps in the execution diagram where it remains idle), it helps to keep all the other threads busy performing the updates and avoids the priority inversion scenario that happened before. Figure 3.24 shows how the overall CPU load reaches a stable 90% during the central part of the algorithm. Note how the amount of parallelism drops towards the end of the execution. This effect is caused by the reduced amount of work that update tasks perform, as the trailing submatrix size decreases with time. The factorization task duration now becomes apparent, as shown in figure 3.25.

In order to improve the efficiency of the application during the last group of iterations, we repartition the problem on-the-fly (data distribution at MPI level does not change), which consists on reducing the block size to half its original value. Consequently, the factorization is divided in two steps (the two diagonal submatrices of the original block). OmpSs allows us to do this without much effort: first we detail the memory region accessed by each task based on the current column and the block size, then we create the necessary additional tasks to finish the problem. Since the loop that generates all the tasks already iterates using columns and block size, we only need to change the block size at the proper place and the program will re-adapt itself automatically (see bottom diagram of figure 3.25).

Figure 3.26 compares the benchmark results of the original version with our task based im-

Figure 3.21: Task duration and number of switches due to pending communications.

Figure 3.22: Task execution diagram including two iterations. Factorizations and high priority tasks are processed by the communications thread (the 8th thread of each process).
Figure 3.23: Execution comparison between OmpSs base implementation and improved one with communications thread. Accelerating the critical path results in a faster execution.

Figure 3.24: CPU load comparison between original version (using multithreaded BLAS) and communications thread (using 4 processes with 8 threads). The average load increases from 80% to 90%, even though one of the threads has a lower amount of work.
Figure 3.25: Execution diagram of the last 24 iterations of a 16K problem size. The amount of work performed by update tasks goes down, so factorization times become more important (top diagram). We re-partition the problem dividing the block size by half, which reduces the factorization times by four (bottom figure), hereby improving slightly the last part of the application.

implementation. It is not surprising to see how the OmpSs version performance is really low when executing with only two threads. The communications thread does not execute any update tasks, other than the urgent matrix products used to accelerate the critical path. On the other hand, coarse grain parallelism obtained from OmpSs parallel implementation algorithm scales better with higher number of threads (8 and 16).

We expected that the performance gap between the original and task-based versions would be larger for the problem sizes we have studied. Unfortunately, the benchmark results show that it is actually not that big, despite the differences in synchronization. We have found that one of the reasons behind this is dgemm kernel’s throughput, which is 25% lower on average when running our task based implementation (see figure 3.27).

The good news are that the scheduling problems and variable performance bottlenecks of this benchmark has motivated the creation of new mechanisms such as the communications thread, that allow processing heterogeneity at the scheduler level. In this specific case, we have used the priority as a mean to select which type worker thread should execute a specific task, although more developed approaches could allow us to express this necessity in a better and cleaner way.
Figure 3.26: HPL benchmark result for different core counts. OmpSs implementation suffers at low core counts, where half of the resources are taken by communication threads. On the other hand, its efficiency increases when the amount of cores per process is higher.

Figure 3.27: \texttt{dgemm} throughput comparison between original and task based implementations.
Chapter 4

Conclusions and future work

OmpSs-MPI interoperability library allows the use of MPI blocking communication routines inside OmpSs tasks. This brings the opportunity to extend their dataflow model on both directions:

- OmpSs data dependence scope can go beyond the process level, removing the synchronization that was previously required to keep the correctness of the applications.
- MPI messages can be split into smaller pieces, allowing for more efficient computation-communication and communication-communication overlap, that would otherwise be much more complex to implement and maintain.

In addition, the implementation of this library has required us to extend OmpSs runtime library to offer additional features, such as polling services and increased control over task states and task switching points. This new features were designed to be easily extended and used for other purposes.

We have demonstrated the potential efficiency gains that this mechanism brings to imbalanced workloads such as the one in IFSker, an application where we were able to implement multiple optimizations related with advanced MPI features. This is also a motivating example to stress the importance of co-design: experts in computer architectures, parallel programming models and application developers must work side by side, with the aim of achieving the best efficiency possible for their programs in modern clusters.

As a result, we are collaborating with IFS experts to implement some of the optimizations that we applied to IFSker, so that these are also introduced into OpenIFS [4], a limited version of IFS weather forecast system, which is used for training and research.

On the other hand, our task-based implementation of the Linpack benchmark demonstrated that this efficiency gains don’t come for free: even though it is now easier to overlap computation and communication, we have seen that this does not necessarily mean that the overall performance increases for all applications in every scenario.

We plan to continue our study on the Linpack benchmark. Specifically, recent versions of Intel Math Kernel Library (a proprietary BLAS implementation optimized for Intel processors) include new extensions to \texttt{dgemm} kernels, which can be used as a mean to solve our performance issues. Moreover, the imminent arrival of the next generation of Marenostrum, also motivates us to improve our task-based implementation. Hence we still keep our target of reaching better performance for lower problem sizes.
4.1 Related work

Hybrid parallel applications are very popular in HPC due to their high efficiency. However, the learning curve necessary to achieve the best performance in these kind of application is steep. As a result, multiple approaches have been presented to either improve the current programming environment or offer new alternatives.

OmpSs-MPI interoperability library is based on the ideas presented by Vladimir et. al \cite{5}, which targeted the SmpSs programming model (predecessor of OmpSs). Vladimir’s approach restricts to one the number of tasks using communication routines, which limits the parallelism in some scenarios in favor of avoiding the need of ensuring the correctness of concurrent communications.

Similar approaches include Argobots \cite{6}. Argobots is a lightweight threading and synchronization framework similar to OmpSs runtime library. Parallel programs using argobots can perform calls to blocking communication routines without producing any type of deadlock, thanks to a modified version of the MPICH library. Therefore, it is not possible to use other implementations such as OpenMPI or other proprietary implementations.

On the other hand, many alternatives offer users the possibility of automatically distributing the data in a distributed memory system, so that the user does not necessarily need to concern about computation-communication overlap, data distribution etc. These alternatives include but are not limited to OmpSs cluster \cite{7}, Charm++ \cite{8}.
Bibliography


