Best Practice Guide for hybridizing pure MPI applications with tasks

Version 1.0

Table of Contents

Table of Figures

Table of Codes

1 Introduction

In this *Best Practice Guide* for parallel application developers we will introduce how to develop hybrid applications using the MPI version as the starting point. The idea then will be to incorporate a task-based programming model such as OpenMP Tasking. Among all the techniques presented in this guide we also include the use of features that are not yet implemented in these two programming models, so we will use the OmpSs-2 model that already incorporates them as a representative for the task-based runtime system, and an MPI interposition library, that will simulate the implementation of the required services in MPI.

The content of this document is the result of the synthesis of several publications that have been presented in the context of the INTERTWinE European project. In a first publication we present the *TAMPI interposition library* (Task Aware MPI) [1]. A library that allows the insertion of *task scheduling/switching points* in the communication services of the MPI library. These points will allow applications to exploit the ability of runtime systems to start the execution of another task while carrying out a sending/receiving service. By being able to keep busy all the working threads with the execution of another task, the communication and computation overlap technique can be implemented by the mere fact of annotating the code that computes the result of a given operation and the code that receives and sends the inputs or results, respectively, with a considerably performance improvement.

The second publication that we present in this guide of best practices is a new methodology approach of parallel decomposition: the *Hierarchical Domain Over-decomposition using Tasks* (HDOT) [2]. This system tries to mimic the parallel decomposition that occurs through the MPI programming model using the tasking model provided by the task-based runtime system. For programmers of pure MPI applications, this decomposition should be much simpler and, by following the same pattern as the first level parallelization, it will allow a better matching of the different phases that compose the algorithm.

This Best Practice Guide also aims to be a complement to the previously published guides in the context of the INTERTWinE European project on hybrid programming based on the Message Passing Library [10][11].

The rest of the sections composing this guide of best practices are as follows. We will start by describing, in Section 2, the main paradigms of HPC application programming in cluster-based systems, including their main advantages and disadvantages. In Section 3 we will develop the algorithm methodology that mimics the behavior of parallel decomposition carried out by MPI (i.e., the Hierarchical Domain Over-decomposition with Tasking). Finally, Section 4 will show the use of the interposition library that allows a greater interaction between the message passing library and the task-based runtime system.

1.1 The Message Passing Interface

MPI [3] is a commonly used standard for the programming of cluster environments. With the past of the years, the standard has shown that MPI programming matters and is commonly applied to parallelize large-scale applications across scientific communities. The paradigm requires algorithmic design for concurrency which makes these applications scale to hundreds of cluster nodes.

MPI programs execute multiple processes, each one executing their own code and using the MPI communication primitives in order to explicitly communicate data between nodes and synchronize sections of code running in parallel. Each process executes in its own memory address space.

1.2 The OpenMP standard

The OpenMP standard [4] is a widely known multi-threaded shared memory programming model. It allows parallelizing applications by using a set of compiler directives, library routines and environment variables. Directive annotations can be used to declare a parallel region, define values of a data environment, synchronize among threads, as well as to create tasks within that parallel region. OpenMP is flexible and supports most parallel patterns: from a manual approach, where programmers may distribute the work according with the thread id and the maximum number of threads; to the OpenMP accelerator model, exploiting parallelism on massive-thread devices; and including the OpenMP tasking model, dealing with unstructured parallelism. There is a range of possibilities allowing programmers to define their own parallel decomposition. However taking a closer look at existing codes shows that the prevalent use of OpenMP is the expression of data-parallel algorithms with an execution model called fork-join. Since OpenMP 3.0 [5], the standard allows programmers to express dynamic task parallelism through its task generating constructs (including `task`, `taskgroup`, `taskyield` and `taskwait`). A task is a unit of work used to express portions of code that could be executed concurrently by the participant threads (according to certain restrictions).

1.3 OmpSs-2 and the BSC Implementation

OmpSs-2 [6] (i.e., OpenMP Superscalar v2) is the second generation of the OmpSs programming model developed at the Barcelona Supercomputing Center. It is a high-level, task-based, parallel programming model for shared-memory systems consisting of a language specification, a source-to-source compiler for C, C++ and Fortran and a runtime library. The language defines a set of directives that allow a descriptive expression of tasks. It is open source and mainly used as a research platform to conceive, implement and test new ideas that can be exported to the OpenMP tasking model. OmpSs-2 (like OpenMP) is based on directives and it enables parallelism in a data-flow style. The developer is in charge of decomposing the code into tasks and identifying their data dependencies. This information is later used by the source–to–source Mercurium [7] compiler to generate the corresponding calls to the Nanos6 [8] runtime API. The runtime library is responsible for scheduling and executing the annotated tasks, preserving the implied task dependency constraints. The Nanos6 runtime and the Mercurium compiler are publicly available at http://pm.bsc.es.

While OmpSs-2 is similar to tasking in the recent specification of OpenMP, the OmpSs runtime implements a different execution model. In OmpSs-2, every application starts with a predefined set of execution resources and an explicit parallel region does not exist. This view avoids the exposure of threading to the programmer as well as the requirement to handle an additional scope, the scope of a parallel region. At compile time, the OmpSs-2 compiler processes the directives and generates an intermediate code file. This file includes both user code as well as all required code for task generation, synchronization and error handling. In the final step of compilation, OmpSs-2 invokes the native compiler to create a binary file.

2 HPC applications for clusters

Non-coherent, distributed memory is a common characteristic of modern High-Performance Computing systems. Such memory organization allows to assemble large systems from commodity components which reduces cost, increases versatility of use and offers flexibility to quickly adapt to application trends. This kind of architecture is referred to as a *cluster of nodes* architecture (or just *cluster*).

2.1 Pure MPI applications

In this scenario, programmers work at the process level, where each of these processes has its own memory address space and therefore requires explicit communication for data exchange. The programmer uses calls such as `send` and `receive` to exchange data between processes and to implement synchronization between them. In other words, the developer implements data distribution and coherence, such that this functionality becomes part of the algorithm.

Designing algorithms with concurrency in mind makes software development more difficult, which can be overwhelming for novice programmers. However, it turns out that the nature of MPI, and imperative parallel programming models in general, obliges the developer to consider concurrency early on in the development. Typically this results in a good design that favours concurrency and scalability. As a consequence, many pure MPI applications tend to achieve better performance scalability compared to applications using incremental parallelism.

Algorithmic design for performance and scalability typically follows one rule: *"keep everyone busy with meaningful work"*. Under the hood this means: (i) implementing ordering of computation and communication for overlaps; (ii) ensuring balanced execution; (iii) keeping overheads low; and finally (iv) creating enough parallelism. It turns out that this is becoming increasingly challenging.

The current trend to increase application performance is to feed them with more and more processing elements (i.e., processor cores) and the way a pure MPI application will use these new processor cores is mapping them one-to-one with MPI processes. The count of processor cores may increase due two different reasons: (i) increasing the number of available cluster nodes in the system; or (ii) increasing the number of processor cores per node.

Increasing the number of nodes puts more and more pressure on the network system, since as the number of processes increases, the inter-node communication traffic potentially may also increase, which will easily leads to a situation of higher contention, higher overheads, etc. Increasing the number of processor cores per node also makes it more and more difficult to achieve scalability on these systems. This is due to the fact that as the number of processes within the same node increases, efficiency of each MPI process drops. The main causes for this behaviour are data overheads, system heterogeneity, load-imbalance and suboptimal communication patterns, all of which are difficult to eliminate or to optimize.

However, many-core systems share resources at core level and so have different properties in terms of bandwidth to memory and I/O (networking), memory size and exclusive use of caches, which affects application behaviour. As a consequence, while many MPI processes occupy all cores, their effective performance and scalability degrade. From the application perspective, there are three reasons for this.

Firstly, each process adds *memory overhead*. Memory overhead originates from internal data structures in MPI libraries associated with each process and, for the most part, from coding practices. For example, many iterative solvers use stencil operations. Stencil operations typically access neighbouring data which, for corner cases, is located on remote processes. The common technique to provide this data to the current process is using data chunks called *halos*. Halos represent temporary data copies. Equally, this

applies to all algorithms with frequent accesses to shared data and therefore requires local copies in order to reduce the communication overhead.

# of ranks	Local domain size	Local halo size	% of data in halo
2	128 x 64 = 8192	2 x 128 = 256	1.6
4	128 x 32 = 4096	(4 - 2) x 4 x 128 = 768	4.7
8	128 x 16 = 2048	(8 - 2) x 4 x 128 = 1792	10.9
16	128 x 8 = 1024	(16 - 2) x 4 x 128 = 3840	23.4
32	128 x 4 = 512	(32 - 2) x 4 x 128 = 7936	48.4

Figure 1: Common programming practice of allocating process-local data results in increasing memory overhead (Gauss-Seidel solver).

Figure 1 shows memory overheads originating from halos for the Gauss-Seidel iterative solver of the Heat2D application. In case of a the two-dimensional Gauss-Seidel solver and a simulation domain of 128 x 128 grid points with horizontal MPI subdomains, halos take up to 48% of the total allocated memory.

But in addition to memory overhead, each process adds potential for *load-imbalance*. Even though MPI applications in scientific computing are mostly SPMD (*Single-Program-Multiple-Data*), where each process is a parametrized instance of the same code, load imbalances do occur. They originate from irregular data sets as well non-uniformity of the system architecture. System boards today integrate processors in layouts with non-uniform performance characteristics. Such layouts place cores at different proximities to memory or source of cooling, affecting memory access latency and operating frequency. As MPI applications are tightly knit through the sequences of send and receive calls, slow processes affect other processes, so the overall application progresses as fast as the slowest MPI process. Implementing a data repartitioning to increase workload balance is programmatically non-trivial and often impossible due to the dynamic nature of the input data sets or non-uniform hardware.

Finally, each process *increases the complexity of the communication pattern*. The communication pattern is the sequence of send and receive MPI calls that is interleaved with computation. Finding a suitable order of execution is important to avoid idling and causing imbalances. Tuning for ordering, however, is difficult for larger applications and not always possible due to the dynamic, irregular nature of some algorithms.

2.2 Is threading the solution?

If many processes per node reduce efficiency, then reducing the degree of concurrency on process-level and increasing it on thread-level could be a solution. Threads are building blocks to express concurrency within one process and share the data environment. They help load balancing, reduce the amount of data replicas and do not require explicit MPI communication to exchange data between them. However, to benefit from threading, the MPI application's code must be adapted.

A variety of ways exist to express concurrency on thread level of which OpenMP is a particularly popular one (see Section 1.2). A programming model that builds on top of node-local threads is called shared-memory programming model. An application that combines multiple programming models is called a hybrid application.

Threading allows a reduction in the number of processes per node while maintaining the same degree of concurrency of the application. The fact that threads share the data environment of a process helps to resolve the shortcomings described above. In first place, threading helps to reduce memory overhead. The use of halos among threads becomes obsolete as any thread running on a processor core can access the neighbouring data of another thread. Further, threads can react to imbalances. If one thread finishes quickly, it simply picks up work of another thread within that process.

Finally, there is no need of explicit MPI communication between threads inside the same node.

```
01  ...
02  T D = getDomain(Domain, N); //set up domain for current process
03  for(auto t : timesteps)
04  {
05     comm(D);
06     f(D);
07     g(D);
08     collective_comm(D);
09     h(D);
10  }
11  MPI_Barrier(...); //synchronize processes
```

Code 1: A representative MPI application with interleaved point-to-point and collective communication.

Code 1 shows a pure MPI application where a sequence of functions is called in a simulation loop over a predefined set of time steps. Comm and collective comm contain MPI point-to-point and collective communication respectively.

```
01   ...
02   N=rank; //assign process ID
03   T D = getDomain(data, N); //set up domain for current process
04   for(auto && t : timesteps)
05   {
06     #pragma oss task inout(D)
07       comm(D);
08
09     #pragma oss task inout(D)
10     {
11       f(D);
12       g(D);
13     }
14     #pragma oss task inout(D)
15       collective_comm(D);
16     #pragma oss task inout(D)
17       h(D);
18   }
19   #pragma oss taskwait
20   MPI_Barrier(...); //synchronize processes
```

Code 2: Sample code showing the top-down parallelization of computation and communication of an MPI application with OmpSs-2 tasks.

Once the execution of the simulation loop finished, processes synchronize at an MPI barrier. Code 2 shows the same code but now enhanced with OpenMP. As the number of processes is reduced in favour of threading, the compute functions `f`, `g` and `h` are now placed inside a parallel region and executed on each thread (*fork*). Threaded execution

requires to adjust the implementations of these functions, hence we name them `f` parallel, `g` parallel and `h` parallel. At the end of each parallel region, an implicit barrier synchronizes threads (*join*). Such thread synchronization is necessary in order to maintain the correct ordering of communication and computation.

Figure 2 shows an execution diagram that visualizes the fork-join parallelism of the compute functions of hybrid codes with MPI and OpenMP. The execution diagram also illustrates the additional amount of data copies for pure MPI applications (multiple data objects) as well as the ability of load-balancing of the hybrid version.

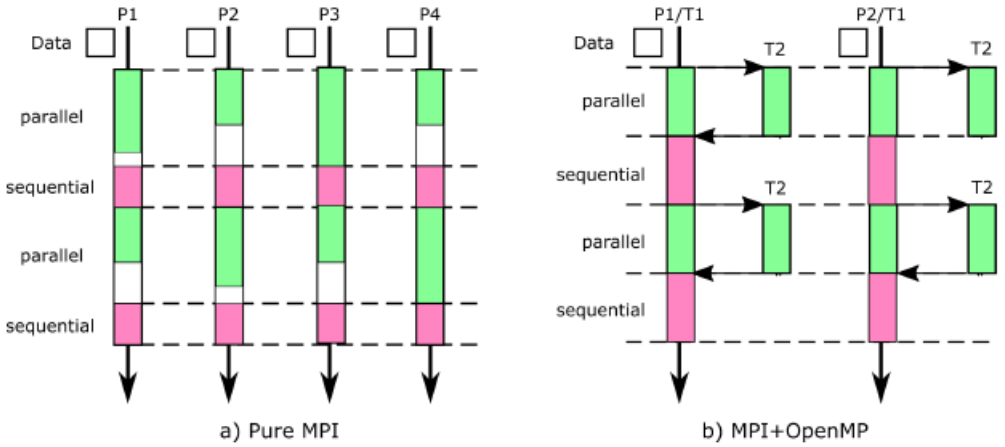

Figure 2: Execution diagram of an application with interleaved communication and computation implemented with pure MPI (a) and MPI and OpenMP (b).

Figure 3 (in the right-hand side) shows the typical performance behaviour of a parallel application implemented with pure MPI and MPI + OpenMP. For small processor counts, the hybrid version shows degraded performance due to serial communication phases while for higher processor counts, the hybrid version shows improved performance over the pure MPI implementation as it can effectively reduce the overheads associated with MPI. Other performance pitfalls exist but apply to parallel programming in general and therefore we omit their discussion.

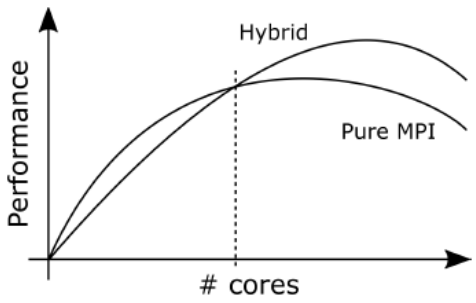

Figure 3: Performance comparison between hybrid and Pure MPI versions (with respect to the number of cores).

Hybrid codes can indeed offer an improvement over pure MPI codes on modern many-core systems. However, it is up to the skill of the programmer to combine both efficiently to achieve performance gains. In case of OpenMP, the programmer is in charge of creating, using and closing parallel regions (i.e., fork-join parallelism) and of synchronizing threads with MPI calls to implement a correct communication pattern. The process of adding thread-level parallelism to an MPI application in practice is characterized by a sequence of *"looking for code sections with significant duration"*, and then *"adding annotations to parallelize these code sections"*. This is strenuous and as a result, hybrid applications often end up with interleaved execution of concurrent computation and serial communication phases with OpenMP and MPI respectively. Such serial phases limit the maximum speed-up that can be obtained parallelizing the application. We call this methodology *two-phase programming*.

Consequently, threading is an enabler towards programming of many-core systems today; however, a solution requires a methodology that would help developers avoid the

limitations of two-phase programming, as well as a runtime system that allows a more natural coexistence between MPI and a data-flow programming models.

2.3 Tasking with OmpSs-2

OmpSs-2 (see Section 1.3) allows the programmer to annotate task parameters with `in`, `out` and `inout` clauses that correspond to the input, output or input-output access type semantic of this parameter within that task. The task directive is defined (in C/C++) as:

```
#pragma oss task in out inout (target)
```

The access type establishes a producer-consumer relationship between tasks, also called task dependency or data-flow. With this information the runtime is capable of scheduling tasks automatically that maintain correctness of code while alleviating the programmer of implementing manual synchronization. Furthermore, the taskwait construct allows task synchronization and instructs the calling thread to wait on all previously created tasks.

At runtime, the main function is executed, which creates tasks and stops at (explicit or implicit) synchronization points. Task creation is composed of two parts: (i) the creation of the task object itself that carries all its descriptive information and (ii) its data dependencies. Once a task object has been created, the runtime inspects the dependency graph to determine the relationship with respect to previously created tasks. If a dependency has been found, a representative node is added to the graph. In the opposite case, the task is placed into a ready-queue. Tasks in the ready-queue are picked up by worker threads, removed from the queue and executed.

Interestingly, one question arises: is it possible to leverage the dynamic, data-flow driven task execution in OmpSs-2 to improve the performance of MPI applications? In the following section we present a new approach which enables a natural coexistence for the OmpSs-2 shared memory programming model with MPI.

2.4 From processes to tasks: MPI + Y

The *Hierarchical Domain Over-decomposition with Tasking* is an approach that simplifies hybrid programming and improves the execution performance of such applications. HDOT applies the domain decomposition of an existing MPI application at process-level to thread-level. At thread-level, domains are decomposed into sub-domains and processed by tasks. HDOT maximizes pattern and code reuse and adds the advantages of tasking. Tasking simplifies the development of hybrid applications by eliminating two-phase programming and by reducing the complexity of synchronization of parallel work and MPI calls. HDOT consists of a *methodology*, a *shared-memory task-based programming model* and an *interoperability library*:

- The methodology defines a top-down approach where the developer uses tasks to encapsulate work from a coarse-grained level down to tasks as small as individual MPI send and receive messages. This includes the definition of sub-domains and support of common algorithmic patterns such as reductions and global variables.
- The task-based programming model eliminates the notion of threads and emphasizes the use of a task as a building primitive for concurrency. Tasks are expressed declaratively and can encapsulate computation and MPI communication alike. Although the OpenMP Tasking model could be used for this purpose, some of the required features are not yet supported by its official implementations. In order to validate the HDOT solution we have used the OmpSs-2 programming model. At the moment of writing this Best Practice Guide the OpenMP 5.0 standard has been released incorporating detached tasks which partially cover the needs of the proposed methodology.
- The Task-Aware MPI (TAMPI) library ensures the correct execution of MPI calls within tasks. Finally, the programming model's support for data-flow programming allows a streamlined execution of computation and communication tasks.

In the following two chapters we will present the aforementioned methodology (Chapter 3) and the Task-Aware MPI interposition library (Chapter 4) respectively. The list of features required by the task-based programming model are not yet fully supported by any commercial standard and it is neither possible to develop any interposition library (like TAMPI does for MPI). So, in the rest of this document will adopt OmpSs-2 as the task-based programming model which actually fulfils all the requirements to use the HDOT methodology. To check the complete list of task-based runtime system features required to fully deploy the HDOT methodology, sees ections A.1 and A.2 (*Block and unblock API* and *Register polling services*, respectively). During the discussion of the methodology we will also use additional text boxes to highlight other features that, although they are not required by the methodology, are implemented in the OmpSs-2 standard and help to better express the parallelism pattern.

3 The HDOT Methodology

HDOT leverages the parallelization and domain decomposition schemes of the original MPI application by promoting their reuse on task-level. In this scheme, domains are split hierarchically, first at process-level, then down to task-level at which domains become sub-domains.

Applying a hierarchical over-decomposition with tasking follows a set of underlying ideas. Firstly, it minimizes requirements for code changes and allows reuse of original MPI code and its data partitioning. Secondly, it decouples tight synchronization between units of work and MPI communication. And lastly, it emphasizes the use of a top-down approach where concurrency is added on different nesting levels to create opportunity for concurrency at a small development effort. We discuss this in three steps as follows.

3.1 Taskifying code

Let discuss the code example from Code 1 (in page 8). The maximal speed-up is limited by the fork-join pattern with sequential sections and the tight synchronization between computation and MPI communication.

To apply the proposed methodology, we start with the original MPI code. In the first step, we taskify code sections. That is, we start adding the OmpSs-2 task directives to the original code (stating that the enclosed code is adept for concurrent execution). It is important to point out that taskifying most of the application's code helps to reduce the number of synchronization points. The execution of tasks in the correct order is guaranteed by following the application's data-flow. Code 2 is based on the previous example but now includes tasks with their respective data dependencies. In this case, tasks also include the MPI communication.

The asynchronous execution of MPI calls by including them in tasks and in the application's data-flow is an important feature towards scalability and programmability.

In this example code, task dependencies of type `inout` serialize the execution of the three tasks. The code thus requires a finer task granularity, which is discussed in the following section.

3.2 Adding subdomains

To increase the degree of concurrency of the application shown in Code 2 (in page 8), we implement what we refer to as a domain over-decomposition that splits domains into subdomains of arbitrary sizes. A domain is a data partition implemented originally by the pure MPI application. A subdomain is a data partition implemented for the use of a shared-memory programming model with the aim of increasing the node-level parallelism. Subdomains follow the same idea of data partitioning found on process level. In case of subdomains, tasks and task functions operate on smaller, process-local data with occasional communication.

The implementation of those functions remains unchanged. Code 3 shows the code changes made in order to accommodate subdomains.

```
01   ...
02   N=rank; //assign process ID
03   T D = getDomain(data, N); //set up domain for current process
04   for(auto && t : timesteps)
05   {
06     #pragma oss task weakinout(D)
07     for(auto && subdomain : D.getSubDomains())
08     if (subdamain.isBoundary())
```

```
09        #pragma oss task inout(subdomain)
10        {
11          comm(subdomain);
12        }
13
14    #pragma oss task weakinout(D)
15      for(auto && subdomain : D.getSubDomains())
16      #pragma oss task inout(subdomain)
17      {
18        if(subdomain);
19        g(subdomain);
20      }
21      #pragma oss task inout (D[0:D.getNumSubDomains()-1])
22      collective_comm(D[0;NB-1]);
23
24      #pragma oss task weakinout(D)
25      for(auto && subdomain : D.getSubDomains())
26        if (subdamain.isBoundary())
27        #pragma oss task inout (subdomain)
28        {
29          h(subdomain);
30        }
31      }
32    #pragma oss taskwait
33    MPI_Barrier(...); //synchronize processes
```

Code 3: The addition of subdomains allows the reuse of domain partitioning at process-level however requires checking for boundary subdomains and the use of weak dependencies.

We have added an additional task nesting level. On the inner nesting level, new tasks are created in for-loops where the number of inner tasks corresponds to the number of subdomains. To add these tasks to the data-flow of the application, each newly created task defines an `inout` dependency over an individual subdomain. However, since the encapsulating tasks from the original code would still serialize the execution between loops, we apply the concept of weak dependencies.

A weak dependency is a key feature in the OmpSs-2 programming model to allow top-down parallelization of code. It allows breaking coarse-grained dependencies between tasks under the assumption that inner tasks will fulfil the dependency requirements.

The use of weak dependencies in this case results in fine-grained dependencies between tasks where communication task over a subdomain can be immediately run once the prior computation task over that subdomain has finished execution. Towards the end of the code sample, a collective MPI operation accesses all subdomains and therefore defines an `inout` dependency over all of them.

Finally, since not all subdomains are equal in their correspondence to the geometric position in the original domain, we have added a check `isBoundary` to the subdomain type to probe if MPI communication is required.

3.3 Including support for common programming patterns

In many scientific codes, developers use reductions and global (static) variables. Reductions are commutative and associative operations that possess an identity element. This allows to compute reductions in parallel as each concurrent unit of work can initialize a private set of operands to the identity element and reduce them in an arbitrary order. We make use of these properties in HDOT and show how reductions are computed on task- as well as process-level. In these cases the reduction value either represents an intermediate result at task-level or a final result on process-level.

Code 4 is a continuation of the code samples shown previously; however we have removed irrelevant code lines. We have added the reduction clause to the inner tasks. This instructs the runtime system to provide a thread-safe storage such that the execution of any two concurrent tasks is data-race free. Once all participating tasks complete, the task implementing the MPI all reduce call can be executed. In OmpSs-2, a reduction creates an `input-output` dependency implicitly.

```
01   ...
02  residual = DOUBLE_MAX; //assign value to reduction variable
03  T D = getDomain(data, N); //set up domain for current process
04  while(residual > norm)
05  {
06    auto rlocal = DOUBLE_MAX;
07    #pragma oss task weakinout(D)
08    for(auto && subdomain : D.getSubDomains())
09      #pragma oss task inout(subdomain) reduction (MAX:rlocal)
10      {
11         rlocal = MAX(rlocal, f(subdomain));
12      }
13
14      #pragma oss task inout (residual) in(rlocal)
15      MPI_all_reduce(rlocal, residual, ...); //MPI collective
16  }
17  #pragma oss taskwait
18  MPI_Barrier(...); //synchronize processes
```

Code 4: The support for reductions and for codes with static variables requires the addition of the reduction clause and local (stack-) variables.

A task reduction in OmpSs-2 is registered at the first instantiation of a task annotated with the reduction clause and its final result is guaranteed at the next synchronization point (i.e., task dependence or taskwait) with respect to the reduction variable.

In order to maintain the dependency between the computation and communication tasks, we have added a stack-local reduction variable called `rlocal`. This creates a

dependency between the computation tasks that perform the reduction and the communication task that perform the MPI all reduce that defines an input over that variable.

Algorithms often use global variables to store the state of the simulation or geometric properties describing a domain. An over-decomposition of the simulation domain, however, requires stack-local variables that hold pointers to such data a structure for each task. For this purpose both code samples (i.e, Code 3 and Code 4) implement a `getSubDomains` and a `getNumSubDomains` that return a set of subdomains and their number respectively. Using pragma annotations such as `firstprivate` for custom data types is not optimal as this typically invokes the copy constructor of that object, resulting in replication of potentially large data.

3.4 Wrapping up the HDOT methodology

The *Hierarchical Domain Over-decomposition with Tasking* (HDOT) methodology relies on the Task-Aware MPI (TAMPI) interoperability library (see Chapter 4), as well as the advanced support of task nesting and fine-grained dependencies (e.g., weak dependencies used in Code 3, page 13) provided by OmpSs-2 programming model. Although this later feature is not mandatorily required by the HDOT methodology, it simplifies the top-down approach that this technique tries to follow.

The main point of the methodology is to apply the original MPI domain decomposition strategy as implemented on process-level to task-level. In this way each part of the global domain assigned to one MPI rank is divided into sub-domains that will be processed by coarse-grained OmpSs-2 tasks. This section describes how common computation and communication patterns have to be modified to obtain the best performance while maximizing code reuse. The HDOT methodology also exploits synergies between MPI and OmpSs-2. On one hand, it reuses the original MPI parallelization strategy to expose coarse-grained parallelism inside a node. On the other hand, the OmpSs-2 data-flow execution based on fine-grained dependencies is leveraged to provide fine-grained inter-node synchronization.

4 Task-Aware MPI

TAMPI is a MPI wrapper library that allows the asynchronous execution of synchronous MPI calls in OmpSs-2 tasks. To do so, it intercepts synchronous calls and redirects them to their asynchronous counterparts. However, in order to guarantee the correct message ordering and invocation of potentially blocked tasks, it relies on the underlying task-based runtime systems to pause and resume tasks. In case of OmpSs-2, the runtime implements an association of task completion with an external event, which in this case is an MPI call. Further, the library offers a generalization of this approach through a common interface which allows to support any type of asynchronous services using an external event API. In case of MPI, the OmpSs-2 runtime system implements a polling service to check for pending MPI operations and to resume the corresponding tasks once the MPI operations complete.

The main goal of the TAMPI implementation is to simplify the development of hybrid MPI + OpenMP and MPI + OmpSs-2 applications and allow an automatic overlapping of computation and communication. We consider it as a key feature towards achieving performance and scalability of hybrid applications on modern systems today. This library is a refinement of the hybrid MPI + SMPSs approach presented in [9].

4.1 MPI point-to-point communications

The MPI 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, the operations are logically concurrent and hence these threads can receive them in any order. To avoid ordering problems on hybrid applications, in practice MPI communications are usually restricted to sequential parts of the application (what is known as MPI's thread funnelled mode), while most computations are performed in parallel. This results in a common pattern that interleaves parallel computation phases (fork–join) with sequential communication phases. This is the easiest and most common way to combine both programming models, but it is not free of drawbacks. On the one hand, it is not easy to overlap computation and communication phases; on the other hand, both inter-node and intra-node parallelism may be potentially hindered due to the strict synchronization enforced among computation phases and across nodes. Hybrid applications may be restructured to manually overlap computation and communication phases of the algorithm using asynchronous communication primitives and techniques such as double-buffering. However, these techniques require complex modifications of the code that, depending on the application complexity, are not even feasible.

An easy way to solve 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 synchronizations. However, this approach cannot be efficiently implemented with current MPI and OpenMP specifications. MPI provides the `MPI_THREAD_MULTIPLE` mode that supports the concurrent invocation of MPI calls from multiple threads, but this is not sufficient to efficiently support task-based programming models such as OpenMP. The main issue is that tasks are not aware of the synchronous MPI primitives, which might block not only the task but also the underlying thread that runs it. Even if the MPI 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 side. Without this notification mechanism, if the number of in-flight MPI operations blocked reaches the number of available hardware threads, the application will hang due to lack of progress. With the current specification of MPI, it is the responsibility of the application developer to avoid this situation. However, this severely limits the ability of application developers to fully benefit from task-based programming models.

4.2 MPI Interoperability Layer

The MPI interoperability library uses the standard MPI interception techniques that enable transparent interception of all the MPI calls performed by an application. Code 5 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 check if the interception library 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).

```
 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._waiter = get_current_blocking_context();
10      _pendingTickets.add(ticket);
11       block_current_task(ticket._waiter);
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._waiter);
25     }
26   }
27 }
```

Code 5: Implementation of the MPI_RECV function in the interoperability library and the polling function executed periodically by the runtime system

The code then checks if the operation is immediately completed. 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 (line 9). The ticket is next registered inside the interception library 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 defines a polling service callback (line 18), which the runtime system calls periodically to check if any MPI operation has completed (line 21). When an MPI

operation completes, the task waiting for that MPI operation is resumed (line 24) and returned to the runtime system's ready queue. All other blocking MPI primitives, including collective operations, are intercepted and managed similarly.

4.3 MPI Progress and Thread Support Levels

The specification of the thread support levels and the modifications to the progress rules necessary for a thread-compliant MPI library are given in Chapter 12.4 of the MPI-3.1 Standard [3]. This section proposes changes to the MPI Standard to support interoperability with task-based runtime pause/resume ability.

Replacing blocking operations with non-blocking operations via the profiling interface will work, in practice, for all correct thread-compliant MPI library implementations because one set of correct MPI function calls is replacing another. Strictly, however, this replacement does not comply with the MPI Standard because there could be situations where multiple function calls, issued by multiple threads, are not serializable. Specifically, their combined effect is not the same as any of the possible linear orderings of these function calls. For example, consider a single thread in a single process executing a task-based runtime with two tasks (one calls a blocking synchronous-mode `send` and the other calls a blocking `receive`, these calls match and there are no other calls that can match). In MPI, as it is today, this must certainly dead-lock and is therefore erroneous by definition. Whichever order the blocking `send` and blocking `receive` are executed in, either the `MPI_SSEND` will block until the `MPI_RECV` is issued, or vice versa. The only execution thread available cannot proceed to the second call without completing the first. However, with the block and unblock APIs (described in Annex A.1), the first blocking function call can pause the calling task, enter the task-based runtime, schedule the other task, issue the second MPI blocking function call, complete it because now both `send` and `receive` have been posted, then return and resume the first task, which can complete its MPI service. These MPI functions were not executed in some order and their effects (as well as their execution) were interleaved. Therefore, the "*executed in some order*" requirement in the MPI Standard would have to be weakened or removed.

The requirements and guarantees for this new support level are not clear from the existing text. In addition, the use of this new block and unblock API requires stronger progress rules than those currently in the MPI Standard. Specifically, we propose that these rules include an additional statement similar to the following:

> *"MPI functions are not permitted to block the calling thread indefinitely. Every MPI function call must either complete or yield to other runnable threads or tasks in finite time."*

In combination with the existing rules, this gives the guarantee that users and task-based runtimes need, but it also forces MPI to implement a mechanism like the block and unblock API proposed in this extension. Since this requires a stronger guarantee from MPI and permits otherwise erroneous code, it should be exposed via an *opt-in* requested/provided mechanism.

Consequently, we propose that MPI should define a new thread support level, which each MPI library can choose to support or not during initialization of MPI. The new thread support level could be called `MPI_TASK_MULTIPLE` and its constant value would be monotonically greater than the existing `MPI_THREAD_MULTIPLE` constant. In this way, applications can request support for the block and unblock tasks functionality via the `MPI_Init_thread` call and check whether the underlying MPI library provides it.

Code 6 shows an example of how a hybrid MPI + OmpSs code may use this new thread support level to write portable applications. First, the application checks if the `MPI_TASK_MULTIPLE` threading level is supported by the underlying MPI library. If this is the case, it defines a sentinel variable pointing to `NULL`, which will be ignored by the runtime system; otherwise, it sets the sentinel variable to one, so that communication tasks will be serialized. This is shown in lines 11–12, where communication tasks are

created with a regular dependency over the block these will work on, as well as an artificial `inout` dependency on the address pointed by the sentinel variable to serialize the execution of these tasks and avoid deadlocks.

```
01 int *sentinel; // Sentinel used to serialize communication tasks
02
03 int main(int argc, char * argv[]) {
04   int provided;
05   MPI_Init_thread(&argc, &argv, MPI_TASK_MULTIPLE, &provided);
06   if (provided == MPI_TASK_MULTIPLE) sentinel = 0;
07   else sentinel = (int *) 1;
08
09   for (int i=0; i<NT; i++) {
10     // Dependency enforced only if *sentinel != 0
11     #pragma oss task inout(tile[i]) inout(*sentinel)
12     communication_task(tile[i]);
13   }
14 }
```

Code 6: Portable initialization using `MPI_TASK_MULTIPLE`.

5 References

[1] Kevin Sala, Jorge Bellón, Pau Farré, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Daniel Holmes, Vicenç Beltran, and Jesus Labarta. 2018. Improving the Interoperability between MPI and Task-Based Programming Models. In Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI'18). ACM, New York, NY, USA, Article 6, 11 pages

[2] Jan Ciesko, Pedro J. Martínez-Ferrer, Raúl Peñacoba Veigas, Xavier Teruel, and Vicenç Beltran. HDOT – An Approach Towards Productive Programming of Hybrid Applications. Submitted to Journal of Parallel and Distributed Computing, (2018).

[3] MPI Documents. Accessed on November 13th, 2018.Available at: https://www.mpi-forum.org/docs/

[4] OpenMP 5.0 Specification. [on-line] Accessed on November 13th, 2018. Available at: http://www.openmp.org/specifications/

[5] OpenMP 3.0 Specification. [on-line] Accessed on November 13th, 2018. Available at: http://www.openmp.org/specifications/

[6] Barcelona Supercomputing Center. OmpSs-2 Specification. [on-line] Accessed on November 13th, 2018. Available at: https://pm.bsc.es/ompss-2-docs/spec

[7] Barcelona Supercomputing Center. Mercurium source-to-source compiler. [on-line] Accessed on November 13th, 2018. Available at: https://github.com/bsc-pm/mcxx

[8] Barcelona Supercomputing Center. Nanos6 Runtime Library. [on-line] Accessed on November 13th, 2018. Available at: https://github.com/bsc-pm/nanos6

[9] V. Marjanovic, J. Labarta, E. Ayguade, M. Valero. Overlapping communication and computation by using a hybrid MPI/SMPSs approach. In Proceedings of the 24th ACM International Conference on Supercomputing, 2010, pp. 5-16.

[10] Best Practice Guide to Hybrid MPI + OpenMP Programming. Milestone of INTERTWinE project. [on-line] Accessed on November 13th, 2018. Available at: https://www.intertwine-project.eu/best-practice-guides

[11] Best Practice Guide for Writing MPI + OmpSs Interoperable Programs. Milestone of INTERTWinE project. [on-line] Accessed on November 13th, 2018. Available at: https://www.intertwine-project.eu/best-practice-guides

Annex A: Required features to support HDOT

In this annex we present the supporting features that enable the use of the Task-Aware MPI library. The proposal consists of a generic API to programmatically pause and resume task execution that must be supported by the task-based runtime system. While we focus this API to OpenMP tasks and MPI, it can also be applied to other task-based programming models and even other OpenMP work-sharing constructs. For instance, an OpenMP runtime could execute more parallel loop iterations while others are blocked on MPI calls. The API also supports other types of operations with blocking and asynchronous variants (e.g. file accesses).

A.1 Block and Unblock API

To support the efficient execution of blocking-like operations in parallel runtimes, we first propose an API to pause and resume tasks. It is composed of three functions. The first has the following prototype in C:

```
void *get_current_blocking_context();
```

This function informs the runtime that the current task is about to enter a pause–resume cycle. The function configures everything needed to handle one round trip and returns an opaque pointer to runtime-specific data. Throughout the rest of this text we call this data a *blocking context*. A blocking context is valid only for one pause–resume cycle, and requesting a new context invalidates the currently active one. The pause and resume operations are requested through the following functions:

```
void block_current_task (void *blocking_ctx);

void unblock_task (void *blocking_ctx);
```

On a call to the first function, the runtime suspends the execution of the invoking task. The parameter must be the current blocking context of the invoking task. The second 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.

```
01 async_handler = start_async_op(...);
02 void *blocking_ctx = get_current_blocking_context();
03 associate(async_handler, blocking_ctx);
04 block_current_task(blocking_ctx);
```

Code 7: Code that performs the blocking operation.

The general usage pattern consists in replacing blocking operations by either asynchronous or non-blocking equivalents, and to let the runtime perform the actual blocking. The runtime can then schedule other computations during the blocking period. This usage scheme is shown in Code 7.

Asynchronous operations that support callbacks can use the callback function to unblock the task. If the operation does not support callbacks, then another thread must (1) periodically test for its completion and (2) unblock the tasks when it finishes. Code 8 shows the pattern that the body of the main loop of such a thread would contain. Notice that the information that links an asynchronous operation with a blocking context must be made visible to the thread that will unblock it.

A.2 Register polling services

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

Code 8: Body of the code that handles the unblocking of the operation.

The detection of finished operations can be either blocking or nonblocking. To simplify the non-blocking case, we propose an additional API that avoids the need for the additional thread. Instead, the runtime can address of those actions at regular intervals or on a best-effort basis.

To make this part generic, the API provides a periodic callback mechanism. The callback should check for the completion of the asynchronous operation and perform the calls to unblock the associated task. The prototype to register the callback is the following:

```
void register_polling_service(char const *service_name,
polling_service_t service_function, void *service_data);
```

It receives a string parameter that is a description for debugging purposes, the callback function and an opaque pointer to data to pass to the callback. The prototype of the callback is the following:

```
typedef int (*polling_service_t)(void *service_data);
```

It receives as a parameter the opaque pointer and returns a boolean value that indicates whether its purpose has been attained: if true, the callback is automatically unregistered; otherwise the runtime will continue to call it. Throughout the rest of this text we will refer to callback as the pair composed by the callback function and the opaque data passed to the registration function.

```
01 int polling_callback(void *service_data) {
02   while (have_ready_operations()) {
03     async_handler = get_ready_operation();
04     void *blk_ctx = get_assigned_blocking_context(async_handler);
05     unblock_task(blk_ctx);
06   }
07   return 0;
08 }
```

Code 9: Callback code that handles multiple operations.

Code 9 shows a callback function that can be used for multiple operations. During initialization, the callback would be registered to ensure that the runtime calls it periodically. The body of the callback is essentially the code already shown in Code 8 but adapted to work in a non-blocking fashion and with multiple operations.

During finalization, the following function can be used to unregister a callback. It receives the same parameters as the registration function and returns once the callback has been disabled:

```
void unregister_polling_service(char const *service_name,
     polling_service_t service_function, void *service_data);
```

In addition to a single callback for many operations, the API also supports using one callback per operation. This is possible by passing the operation information through the service data parameter and by automatically unregistering the callback through its return value. Code 10 shows the blocking-side code. Unlike in the code previously shown in Code 7, the callback is not registered during initialization. Instead, before blocking, the new code registers the actual callback function together with the data associated with the operation. The callback function, which is shown in Code 11, 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.

```
01 operation_info_t oi;
```

```
02 oi.async_handler = start_async_op(...);
03 oi.blocking_ctx = get_current_blocking_context();
04 register_polling_service("service-per-request-example",
                            polling_callback, &oi);
05 block_current_task(oi.blocking_ctx);
```

Code 10: Modified blocking code to use one callback per operation.

```
01 int polling_callback(void *service_data) {
02   operation_info_t *oi = (operation_info_t *) service_data;
03   int finished = operation_has_finished(oi->async_handler);
04   if (finished) {
05     unblock_task(oi->blocking_ctx);
06   }
07   return finished;
08 }
```

Code 11: Callback code that handles only one operation.