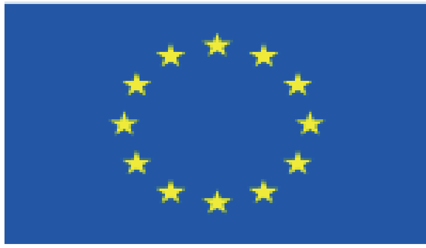
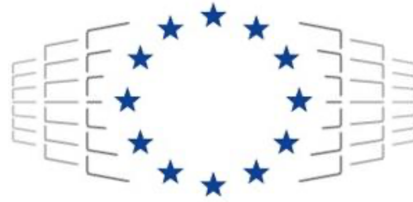
EUROCC - National Competence Centres in the framework of EuroHPC

EuroHPC-04-2019: HPC Competence Centres

Hybrid parallel programming with tasks

J. Mark Bull and Jiehong Yu
EPCC, University of Edinburgh





This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 951732. The JU receives support from the European Union's Horizon 2020 research and innovation programme and Germany, Bulgaria, Austria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, Greece, Hungary, Ireland, Italy, Lithuania, Latvia, Poland, Portugal, Romania, Slovenia, Spain, Sweden, United Kingdom, France, Netherlands, Belgium, Luxembourg, Slovakia, Norway, Switzerland, Turkey, Republic of North Macedonia, Iceland, Montenegro

Table of Contents

1 Introduction ... 4
1.1 Motivation ... 4
1.2 Programming with tasks and dependencies ... 5
2 Combining tasks with MPI – problems and solutions ... 5
2.1 The TAMPI library ... 6
2.1.1 Blocking mode ... 6
2.1.2 Non-blocking mode ... 7
3 Experiences with MPI + tasks ... 8
3.1 Porting code to the MPI + tasks model ... 8
3.1.1 Refactoring ... 8
3.1.2 Debugging ... 9
3.2 Optimisation ... 9
3.2.1 Understanding performance ... 9
3.2.2 Task granularity ... 10
3.2.3 Anti-dependencies ... 10
3.2.4 Task locality ... 11
4 Resources ... 11
5 Conclusions ... 11
6 Acknowledgements ... 12
7 References ... 12

1 Introduction

This technical report is intended to provide an introduction to the hybrid HPC programming model developed at Barcelona Supercomputer Center (BSC) which combines MPI [1] between nodes with dependent tasks within each node, and is supported by TAMPI [7][8], a task-aware version of the MPI library. In the rest of this section, we motivate this model, and give a brief introduction to dependent tasks. In Section 2, we explain why using dependent tasks with a normal MPI library can cause problems, and how these are solved by TAMPI. Section 0 reports some practical experiences with this model, offers some advice and hints to programmers, and points out some potential problems. Section 4 provides some pointers to resources for anyone interested in trying out the model in practice, and Section **Error! Reference source not found.** provides a brief summary.

1.1 Motivation

Traditional HPC programming models, such as (two-sided) MPI, or hybrid MPI and OpenMP [5], encourage the programmer to over-specify both the ordering of computational tasks, and the synchronisation between them. In task-based programming models, the programmer specifies computational tasks with their input and output parameters and lets the runtime system figure out the data dependences between them. This approach gives extensive freedom to the runtime to schedule tasks, offers the opportunity to reduce the ordering and synchronisation constraints to (or close to) the minimum required by the underlying algorithm.

Many current HPC applications make implicit assumptions about performance homogeneity: they assume that the same sequence of instructions executed concurrently on multiple hardware units will complete in approximately the same time and, similarly, that transfers of data from the same source to the same destination (for example, memory to CPU core, or node to node across the network) will also take approximately the same time. As hardware design evolves towards systems capable of Exascale performance, there are signs that the performance predictability of hardware will get worse, as architects start to try to exploit fine-grained power saving features and possibly introduce more heterogeneous compute resources. This means that the assumptions described above may start to become invalid, and so applications will fail to scale adequately to the very high numbers of execution threads required for Exascale performance. In any case, traditional parallel application designs that rely on a bulk-synchronous approach are likely to fail to scale to very high core counts, as the cost of global synchronisation, and the related load imbalance, will start to limit performance gains.

Combining MPI and OpenMP in the same application is now quite a common technique to try to overcome scalability problems experienced by pure MPI codes. The simplest approach to this keeps the two layers of parallelism separate, by only making calls to the MPI library from the main thread, outside of any OpenMP parallel regions. This is the so-call *master-only* or *fork-join* style of hybrid programming. While this makes the implementation straightforward, it has performance disadvantages: there may be frequent barrier synchronisation points, which are quite expensive and can expose load imbalance between threads, and the threads other than the main thread are necessarily idle while MPI communication is taking place.

Another problem with MPI applications is that, despite the support in MPI for non-blocking communication (both for point-to-point and collectives), realising overlapping of computation with communication is difficult in practice. This is because the MPI library is in many cases unable to recruit the required CPU resources to make background progress on handling communications at the same time as the CPU is being used for user-level computation. The result is that the completion of the communication is delayed until a blocking MPI call (for example MPI_Wait) is made. The master-only

style of hybrid program also suffers from this as, since MPI is called outside of parallel regions, there are (logically, at least) no threads available to explicitly compute while communication is in flight.

Efficiently implementing full tasking model on top of distributed memory is an especially hard problem. A compromise solution is to use tasks with dependencies within a node, and a conventional message-passing API such as MPI between nodes, but with the MPI calls contained inside tasks. This still allows the program to be written in such a way as to specify only (or nearly only) the orderings and synchronisation that are necessary for correctness, and permit as much dynamic asynchrony as possible, and therefore minimise the overheads that arise from cores being idle.

1.2 Programming with tasks and dependencies

Both OpenMP and OmpSs-2 [6] provide similar mechanisms for creating computational tasks and expressing data dependencies between them. In OmpSs-2, this coarse-grained dataflow approach is the main computational model: there is a single thread of execution which can create tasks, and there is a thread pool that is available to execute them. Dependencies are expressed by annotating the tasks with the way that they use data items in the program: at the simplest level an **`in`** dependency on an address means that this task must execute after any previously generated tasks with an **`out`** dependency on the same address. The runtime is responsible for building the task dependency graph and executing tasks in some order that respects the dependencies.

OpenMP is a much richer API but supports a similar mechanism (which was originally inspired by OmpSs-2). The same coarse-grained dataflow approach can be implemented by creating a parallel region around the entire program (which provides the thread pool) and then choosing one thread to generate tasks with dependencies, while the other threads are available to execute them.

For more details of the programming models, see Section 4 for pointers to resources.

2 Combining tasks with MPI – problems and solutions

To avoid any unnecessary barrier synchronisation, we would like to include the MPI calls in a hybrid program inside tasks, so that they can execute sends as soon as the data to be communicated is ready, and permit tasks that use received data to run as soon as this data has arrived. However, by including blocking MPI calls inside tasks, it is easy to generate situations where deadlock can occur. 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 software and hardware thread that runs it. With the current specification of MPI, it is the responsibility of the application developer to avoid this situation.

An example is shown in Figure 1: Process 1 wants to send four messages with different tags to Process 2. If no dependencies are specified between the tasks, they may execute in any order, and we could arrive at the situation shown, where there are two CPUs in Process 1 which are currently executing two tasks, containing MPI_Sends for messages with tag 0 and tag 1. Meanwhile there are two CPUs in Process 2, which are executing tasks including MPI_Recvs for messages with tag 2 and tag 3. In this case, deadlock would happen between these two processes since none of the MPI calls can be finished.

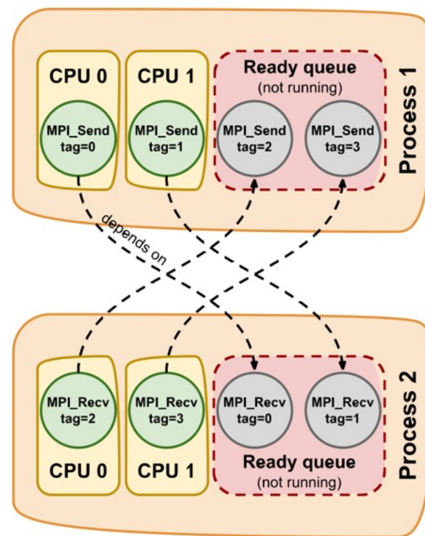


Figure 1: Potential deadlock in taskified MPI (Credit: BSC)

This type of problem can be avoided by adding dependencies between tasks that call MPI, for example by making all communication tasks have an **`in`** and **`out`** dependency on the same address, which means that no MPI calls can be reordered. However, achieving correctness without sacrificing performance turns out to be difficult. A better solution is to integrate the MPI library with the task-based runtime. The TAMPI (Task Aware MPI) library has been developed to do just this. The essential idea here is that whenever a task would block in an MPI call, the hardware resources can be released to run other tasks until all the events necessary for the blocked call to complete have occurred.

2.1 The TAMPI library

TAMPI extends the functionality of standard MPI libraries and makes it suitable for task-based programming. By using TAMPI, MPI calls inside asynchronous tasks can be executed by threads in a safe and efficient way. TAMPI sits on top of conventional MPI libraries, and only minimally extends the standard MPI interface. TAMPI has two different mechanisms to achieve such secure communication: a blocking mode and a non-blocking mode. TAMPI works with both OmpSs-2 and OpenMP, but while both OmpSs-2 and OpenMP support the non-blocking mode of TAMPI, only OmpSs-2 can support the TAMPI blocking mode.

2.1.1 Blocking mode

This mode uses the blocking communications in MPI library such as MPI_Send and MPI_Recv, and some of the more commonly used collectives. Just as for computational tasks, tasks with MPI communications are first instanced, then wait in the ready queue for scheduling and execution on a thread. As shown in the Figure 3.2, when. a task is being executed by a thread and the communication inside it is pending, this task will be sent to a paused queue, where it will wait for the MPI blocking communication to be finished. During this process, the thread, which initially execute this task, is allowed to run any other ready tasks instead. Once the MPI communication is done, the paused task will then be sent back to the ready task pool and waits to be reschedule and execute its remaining

computations. Once all the computations inside this task are finished, the task will be regarded as complete and its output dependencies on other tasks will then be released.

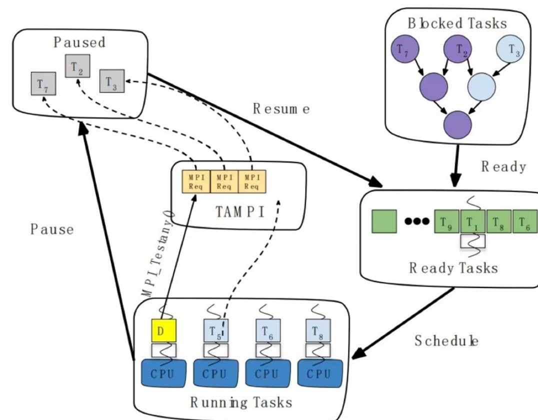


Figure 2: Blocking communication in TAMPI (Credit: BSC)

2.1.2 Non-blocking mode

The non-blocking mode of TAMPI supports non-blocking MPI primitives such as MPI_Isend and MPI_Irecv. It also extends the MPI specification to add non-blocking versions of MPI_Wait and MPI_Waitall (which may at first seem a bit strange at first!) The difference of executing non-blocking computation tasks compared with blocking communication tasks happens when the MPI communication is pending. In non-blocking mode, if an MPI communication is pending, the thread is allowed to continue executing the rest of the computations inside the task or to execute other ready tasks. Therefore, in such a task, the non-blocking communication inside it may be unfinished when all computations of it are done. As shown in Figure 3.3, when all computations inside this task are completed, the task will be regarded as a finished task, which means that all the statements inside the task have been executed, but its non-blocking communication has not completed yet. Once the non-blocking MPI communication inside this task is done, then this task will finally be regarded as complete and its output dependencies for other tasks will then be released.

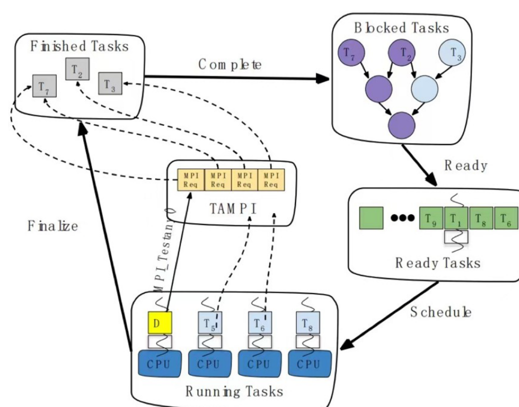


Figure 3: Non-blocking communication in TAMPI (Credit: BSC)

3 Experiences with MPI + tasks

There are a number of published studies [2][9][10] which describe this programming model, its use in a variety of mini-apps, and the performance benefits attained. In this section we summarise less formally some experiences with porting applications to the MPI + tasks model and optimising them, highlighting some of the difficulties encountered and techniques used to solve them.

3.1 Porting code to the MPI + tasks model

3.1.1 Refactoring

In practice, the starting point for an MPI + tasks version of an application code with normally be an existing pure MPI, or master-only style MPI + OpenMP version. This is not necessarily ideal, as there are some design decisions which could be better made with tasks in mind from the beginning.

Converting OpenMP parallel loops to tasks without dependencies and task barriers (using **`taskwait`**) is a useful first step in the process. This is relatively straightforward, and give us the opportunity to correctly implement the data-attribute scoping of variables (i.e. determining whether variables accessed inside a task should be **`shared`**, **`private`** or **`firstprivate`** with respect to a task) without yet worrying about getting the dependencies correct. At this stage, reduction patterns may need to be refactored to use atomic updates, since task reductions have limited support in OpenMP and OmpSs-2. After this intermediate stage, the **`taskwait`**s can be incrementally removed as we introduce the task dependencies.

In some applications, removing all barrier synchronisation can be difficult, as the dependencies are a function of the values of data. As an example, consider a typical pattern found in N-body simulations, where forces between bodies are calculated using neighbour lists, and once this is complete, the new velocities and positions of the particles can be updated. In pseudocode this can be represented as:

```
for each body
    for each neighbour
       calculate pairwise force
       update total force on this body and neighbour
    end for
end for

for each body
    update velocity and position
end for
```

In a parallel loop implementation, a barrier is required between the two outer loops, since any iteration of the first loop might update the total force on any other body, and it is not possible to start any iterations of the second outer loop until the first loop is completely finished.

To overcome this, one option is to use standard domain decomposition methods (with halo regions) within an MPI rank as well as across MPI ranks. Computations in different sub-domains can be assigned to different tasks and exchanging data between sub-domains in the same MPI ranks can be implemented as memory copies. Using this technique, tasks will only depend on other tasks computing on neighbouring sub-domains, allowing the removal of task barriers. The approach has been termed Hierarchical Decomposition Over Tasks (HDOT) and is discussed in more detail in [2].

3.1.2 Debugging

Difficulty in effective debugging is perhaps the biggest current drawback of the MPI + tasks model. Errors in task dependencies can cause race conditions, and it also possible to introduce race conditions between accesses to shared variables in tasks and in the generating thread. As with many race conditions, these may occur rarely, be difficult to reproduce, and may appear or disappear when any debugging code is added or removed. Using dependencies gives the runtime a lot of freedom to choose an ordering of tasks, so there can be many possible pairs of tasks being executed concurrently which need to be checked for races.

Reintroducing task barriers can be helpful in narrowing down the source of race conditions, but the inherent non-reproducibility of races is still a problem. Unfortunately, the dependent tasks model is not supported by any of the widely available race-condition detection tools such as Intel Inspector, TSan, or Helgrind/DRD. In difficult cases, progress may only be made with careful code analysis and/or heavy use of assertion checking. Trace visualisation (see Section 3.2.1 below) can also be helpful to spot where tasks are not executing in the expected order.

3.2 Optimisation

3.2.1 Understanding performance

Once we move away from loop-based (fork/join) parallelism, conventional profiling tools become less useful for diagnosing performance problems, since code from several different tasks may be executing concurrently. Often, what we are really looking for is to identify if and when cores are idle or executing inside the runtime, rather than executing tasks. To do this, timeline trace visualisation is much more helpful. For OmpSs-2, this can be enabled by using the Extrae tracing library and the Paraver visualisation tool (see Section 4). An example trace from a run on two nodes each with 48 cores is shown in Figure 4. Time runs from left to right, colours correspond to tasks generated from different sites in the code, and black areas indicate time where cores are not executing user tasks, which in this case is significant enough to warrant further investigation.

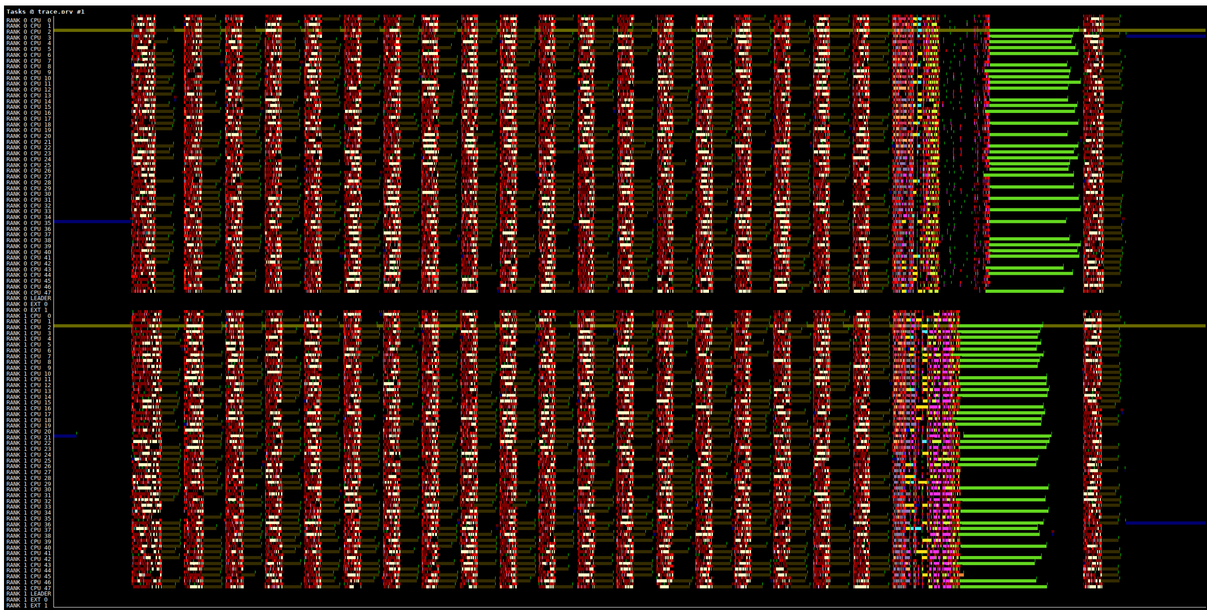


Figure 4: Example Extrae/Paraver trace (Credit: BSC)

Such traces clearly show *when* cores are idle but diagnosing *why* can still be difficult. Some additional information about the status of the runtime would be helpful: tasks could be waiting for dependencies to be satisfied, or there could be no tasks available because the generating thread is overloaded, or else the runtime is busy scheduling tasks, or waiting for MPI messages to arrive.

3.2.2 Task granularity

As with any task-based programming, task granularity is a key factor that determines performance. If we have a smaller number of large tasks, then there may be insufficient parallelism available to keep all the cores busy. On the other hand, if we have too many small tasks, then the overheads of task generation and scheduling may be too large and dominate the execution time. In the typical "one thread generates" pattern supported by OmpSs-2, the generating thread can become the bottleneck: tasks cannot be generated fast enough to keep all the other threads busy (this is clearly visible on a trace, shown by the generating thread being always in a busy state).

The best performance is often to be found at some intermediate granularity, but this will depend on the number of threads executing as well as the input data, so designing parameterised granularity into the application is an important consideration. A simple example of this is loop blocking: where it might be straightforward to implement each loop iteration as a task, we can have better control by making a set of contiguous iterations into a task and making the number of iterations per task a tuneable parameter.

It is well worth profiling the application code early in the porting process to understand the number of tasks that will be generated by different possible designs and their expected execution times. It is also likely that different types of tasks in the same application will have different execution times, so we can end up with several different parameters that control granularity and face a complex tuning scenario to find their optimal values [4]. However, a useful rule of thumb is that, in the absence of large load imbalances, the number of tasks generated from a loop should be a small multiple (say 2-4) of the number of threads available for task execution.

The number and granularity of communication tasks is also important for performance. TAMPI requires **`MPI_THREAD_MULTIPLE`** support from the underlying MPI library so that it is safe for multiple tasks (and therefore threads) to be making concurrent MPI calls. However, to implement this thread-safety, the MPI library will typically use locks which sequentialise communications. With many tasks all sending or receiving messages at the same time, this can become a serious bottleneck. There is scope for better MPI implementations in this regard, but progress is in the hands of the MPI library implementors. Some experiments with the GASPI one-sided communication library [3], which has much finer-grained internal synchronisation, and its task-aware version TAGASPI [10] show that it is possible in principle to overcome this problem. If using TAMPI, however, combining the messages from different tasks, and thus sending and receiving fewer, larger messages can be beneficial, even if this reduces parallelism by introducing some additional dependencies.

3.2.3 Anti-dependencies

Some of the required dependencies between tasks may be anti-dependencies, i.e. the ordering of tasks is required to ensure that an old value of a variable is read before it is written by another task. This pattern may arise with MPI buffers: we must ensure that the message has actually been sent before refilling the buffer with new values. In some cases, for example neighbourwise halo exchanges, these dependencies can be removed by double buffering. Double buffering consists of creating two of more copies of the buffer and using them alternately: in the more general case we can create more copies and cycle through them.

For halo exchange patterns double buffering may be sufficient, as there may be an implicit dependency on the relevant task from a previous iteration, via an exchange of messages. For other patterns, multiple

buffering can be useful: we may still need an explicit dependency on the previous use of each copy, but with enough copies, this may never cause tasks to be blocked.

3.2.4 Task locality

A potential problem with any task-based programming model is that the programmer has little or no control over which threads, and therefore on which cores, tasks will be executed. This makes it difficult to exploit caches effectively by ensuring data reuse between tasks (if there is sufficient reuse *within* tasks, there is no such problem). The runtime's task scheduling strategy can help (for example by preferring to execute recently generated tasks) but this is often not sufficient, especially for patterns where loop parallelism naturally exploits locality. This is true for example where arrays are directly indexed by parallel loop iterators, so always mapping the same subset of loop iterations to the same threads results in good data affinity. When using tasks, the ability to ensure this kind of mapping is lost. The effect is most pronounced in strong scaling scenarios when sufficient cores are used such that the working set may fit into some level of cache.

The **`taskfor`** construct in OmpSs-2 is designed to overcome this to some extent. This construct identifies a whole loop as a single task, with **`in`** and **`out`** dependencies, but allows the loop iterations to be distributed across multiple threads (though exactly how many threads is still determined by the runtime. This also helps to reduce some of the overheads of scheduling tasks with dependencies.

4 Resources

- The TAMPI library, and a short user guide, can be found at https://github.com/bsc-pm/tampi.
- The latest OmpSs-2 distribution can be downloaded from https://github.com/bsc-pm/ompss-2-releases and there are detailed installation instructions at https://pm.bsc.es/ftp/ompss-2/doc/user-guide/.
- The LLVM version that supports OpenMP task integration with TAMPI is available from https://github.com/bsc-pm/llvm.
- The Extrae tracing tool and Paraver visualisation tool are available from https://tools.bsc.es/downloads.
- The TAGASPI library can be found at https://github.com/bsc-pm/tagaspi.
- The OpenMP specification can be downloaded from https://www.openmp.org/specifications/

5 Conclusions

This hybrid programming model offers some interesting possibilities to overcome performance problems which result from the unnecessary synchronisation and ordering constraints imposed by the most commonly used hybrid model in HPC (MPI + OpenMP loops). However, using the model is not without its difficulties: it requires a change of mindset from the developer. The high degree of asynchrony makes reasoning about correctness and performance more difficult, and there is at present a lack of suitable tool support for developers which will take time and effort to address. Nevertheless, this is a promising direction for maximising application performance on future HPC platforms.

6 Acknowledgements

The authors would like to acknowledge the support from, and very helpful interactions with, colleagues at BSC, in particular Vicenç Beltran, Rodrigo Arias, Marcos Maroñas and Kevin Sala, as well as at EPCC, including Caoimhín Laoide-Kemp and Dominic Sloan-Murphy.

7 References

[1] Borchers, B. and Crawford, D., (1993) Message-Passing Interface. The International Journal of Supercomputing Applications, 7(2), pp.179-179.

[2] Ciesko, J., Martínez-Ferrer, P. J., Veigas, R. P., Teruel, X., & Beltran, V. (2020). HDOT—An approach towards productive programming of hybrid applications. Journal of Parallel and Distributed Computing, 137, 104-118.

[3] Grünewald, D. & Simmendinger, C. (2013) The GASPI API specification and its implementation GPI 2.0. In Proceedings of 7th International Conference on PGAS Programming Models.

[4] Maroñas, M., Teruel, X., Bull, J. M., Ayguadé, E., & Beltran, V. (2020). Evaluating Worksharing Tasks on Distributed Environments. In 2020 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 69-80). IEEE.

[5] openmp.org (2022) OpenMP Application Programming Interface. [online] Available at: <https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf> [Accessed 15 Dec 2022].

[6] pm.bsc.es (2022) OmpSs-2 Specification. [online] Available at: <https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf> [Accessed 15 Dec 2022].

[7] Sala, K., Bellón, J., Farré, P., Teruel, X., Perez, J. M., Peña, A. J., Holmes, D., Beltran. V., & Labarta, J. (2018). Improving the interoperability between MPI and task-based programming models. In Proceedings of the 25th European MPI Users' Group Meeting (p. 6).

[8] Sala, K., Teruel, X., Perez, J. M., Peña, A. J., Beltran, V., & Labarta, J. (2019). Integrating blocking and non-blocking MPI primitives with task-based programming models. Parallel Computing, 85, 153-166.

[9] Sala, K., Rico, A., & Beltran, V. (2020). Towards Data-Flow Parallelization for Adaptive Mesh Refinement Applications. In 2020 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 314-325). IEEE.

[10] Sala, K., Macia, S., & Beltran, V. (2021). Combining One-Sided Communications with Task-Based Programming Models. In 2021 IEEE International Conference on Cluster Computing (CLUSTER) (pp. 528-541). IEEE.