Improving the Interoperability between MPI and Task-Based Programming Models

Kevin Sala*
Barcelona Supercomputing Center
kevin.sala@bsc.es

Jorge Bellón*
Barcelona Supercomputing Center
jorge.bellon@bsc.es

Pau Farré*
Barcelona Supercomputing Center
pau.farre@bsc.es

Xavier Teruel*
Barcelona Supercomputing Center
xavier.teruel@bsc.es

Josep M. Perez*
Barcelona Supercomputing Center
josep.m.perez@bsc.es

Antonio J. Peña*
Barcelona Supercomputing Center
antonio.pena@bsc.es

Daniel Holmes*
EPCC
d.holmes@epcc.ed.ac.uk

Vicenç Beltran*
Barcelona Supercomputing Center
vbeltran@bsc.es

Jesus Labarta*
Barcelona Supercomputing Center
jesus.labarta@bsc.es

ABSTRACT

In this paper we propose an API to pause and resume task execution depending on external events. We leverage this generic API to improve the interoperability between MPI synchronous communication primitives and tasks. When an MPI operation blocks, the task running is paused so that the runtime system can schedule a new task on the core that became idle. Once the MPI operation is completed, the paused task is put again on the runtime system's ready queue. We expose our proposal through a new MPI threading level which we implement through two approaches.

The first approach is an MPI wrapper library that works with any MPI implementation by intercepting MPI synchronous calls, implementing them on top of their asynchronous counterparts. In this case, the task-based runtime system is also extended to periodically check for pending MPI operations and resume the corresponding tasks once MPI operations complete. The second approach consists in directly modifying the MPICH runtime system, a well-known implementation of MPI, to directly call the pause/resume API when a synchronous MPI operation blocks and completes, respectively.

Our experiments reveal that this proposal not only simplifies the development of hybrid MPI+OpenMP applications that naturally overlap computation and communication phases; it also improves application performance and scalability by removing artificial dependencies across communication tasks.

KEYWORDS

MPI, OpenMP, OmpSs-2, interoperability, task, asynchronous

*ORCIDs: 0000-{0001-8233-1185, 0002-4400-4180, 0002-5705-8620, 0001-5181-7545, 0002-0558-7600, 0002-3575-4617, 0002-2776-2609, 0002-3580-9630, 0002-7489-4727}

1 INTRODUCTION

Current near-term and mid-term high-performance computing (HPC) architecture trends suggest that the first generation of exascale computing systems will consist of distributed memory nodes, where each node is powerful and contains a large number of compute cores. A well-established practice in the HPC community is to develop hybrid applications combining APIs such as MPI and OpenMP, which are specialized in exploiting inter-node and intra-node parallelism, respectively. Although MPI and OpenMP were not originally designed to be used together, these have evolved to provide some interoperability support. However, this minimal support heavily determines how both models can be safely combined to develop hybrid applications, posing performance implications.

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 funneled 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.

The final publication is available at ACM via http://dx.doi.org/10.1145/3236367.3236382

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 hardware 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.

In this work, (1) we introduce a generic API to programmatically pause and resume task execution; (2) we propose MPI_TASK_MULTIPLE, a new level of thread support for MPI that leverages this generic API to better support blocking MPI operations inside tasks; (3) we implement this support into a portable MPI wrapper library that works with any MPI implementation; (4) we also extend a well-known MPI implementation to directly use the pause/resume API, and (5) we provide an in-depth performance and scalability evaluation of our proposals.

The rest of the paper is structured as follows. Section 2 provides an introduction to the OpenMP and MPI programming models. In Section 3 we review related literature. In Section 4 we present the pause and resume API. Section 5 describes the MPI interception library and the MPICH native implementations. We evaluate our work in Section 6. In Section 7 we describe the impact of our proposal to the OpenMP and MPI standards. Finally, Section 8 provides concluding remarks.

2 BACKGROUND

In this section we provide a brief overview of the OmpSs and MPI programming models along with the implementations we leverage.

2.1 OmpSs-2 and the BSC Implementation

OmpSs-2 is the second generation of the OmpSs programming model developed at the Barcelona Supercomputing Center. 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 the parallelism in a data-flow way. 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 [1] compiler to generate the corresponding calls to the Nanos6 [11] 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.

2.2 MPI and the MPICH Implementation

MPI is a message-passing standard [10] broadly used by the HPC community. MPICH is a popular open source MPI implementation (see http://www.mpich.org), and its derivatives (such as Intel's, Cray's, or IBM's MPI) are default in 9 out of the top 10 supercomputers in the current TOP500 list [14]. In this paper we introduce calls to the proposed API into the latest MPICH stable release, version 3.2.1 dated Nov. 2017.

3 RELATED WORK

Overlapping computation and communication phases is a critical issue that has already been studied in several contexts. In [2, 3], the authors developed a threading library for the Cell B.E. processor that transparently overlaps the computation and communication phases of different threads running on the same SPU (Synergistic Processor Unit in the Cell B.E. architecture). When the running threads are about to block on a DMA operation (which would be equivalent to a wait all or wait any operation in the MPI interface), the execution of the thread is suspended until the DMA operation completes. In the meantime, the execution of another thread is resumed on the SPU to overlap the communication phase of the suspended thread(s) with the computation phase of the current thread. This work also studies double- and multi-buffering techniques which also allows overlapping, but are limited to applications with a regular and static communication patterns, while the approach based on threads supports irregular applications with a dynamic communication pattern.

The study of hybrid approaches [8, 12, 13] combining communication libraries and shared memory programming models has been considered over the last years both in research and in performance analysis publications.

Using the comm_thread approach of the hybrid MPI+SMPSs programming model [9], authors allowed to exploit distant parallelism separated by taskified MPI calls. These tasks were also identified as *communication tasks* and were executed by an additional thread called *communication thread*. The runtime's task scheduler could reorder the execution of communication and computational tasks in such a way that communication can happen as soon as possible, increasing the parallelism within and across MPI processes. That proposal requires changes to the programming model to allow to identify ahead of time those tasks that have blocking-like behavior. In addition, only one thread can execute them, and it must do so in sequential order. Hence, this solution

is suboptimal. In this paper we propose a runtime-agnostic solution that does not require to pre-classify the work units, that allows tasks to contain any mixture of computations and communications, and that supports several communications in parallel and out of order.

In the Habanero-C MPI (HCMPI) proposal [4], MPI calls are tightly integrated with the task dependency system. HCMPI treats all MPI calls as (asynchronous) tasks, which brings well-known issues inherent to excessively fine-grained tasking, such as increased scheduling overhead and load imbalance. In contrast, our proposal is orthogonal to the dependency system and is specially well suited to parallelize legacy and library code, as it does not require to taskify every MPI operation, resulting in a more natural and flexible approach.

4 INTEROPERABILITY BETWEEN PARALLEL RUNTIMES AND BLOCKING OPERATIONS

This section overviews and proposes solutions to the challenges of interoperating efficiently parallel runtimes with operations that have blocking-like semantics.

For instance, synchronous I/O operations over files may block the thread that invokes them for the duration of the operation. On an environment with multiple processes competing for CPU time, the time that the thread is blocked may be used by another thread. However, on hosts dedicated to a single multithreaded HPC job, the core is most likely to remain idle.

In this section we discuss our proposal in the context of OmpSs-2 and MPI.

4.1 Block and Unblock

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.

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 Figure 1a.

Asynchronous operations that support callbacks can use the callback function to unblock the task. If the operation does not support callbacks, then another thread has to (1) periodically test for its completion and (2) unblock the tasks when it finishes. Figure 1b 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.

4.2 Polling

The detection of finished operations can be either blocking or non-blocking. 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.

Figure 2a 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 Figure 1b 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

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

(a) Code that performs the blocking operation

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

(b) Body of the code that handles the unblocking of the operation

Figure 1: Pause and resume pattern to handle a synchronous operation

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

(a) Callback code that handles multiple operations

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

(b) Callback code that handles only one operation

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

(c) Modified blocking code to use one callback per operation

Figure 2: Pause and resume patterns with two polling approaches

service data parameter and by automatically unregistering the callback through its return value. Figure 2c shows the blocking-side code. Unlike in the code previously shown in Figure 1a, 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 Figure 2b, 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.

4.3 Blocking and Unblocking in Nanos6

The blocking call in Nanos6 forces a scheduling point in the task. At this point, the task will not be able to resume until it is sent back to the scheduler. If there are ready tasks, the scheduler will assign one to the core. Otherwise, the core will become idle.

The unblocking call sends the task back to the scheduler. During this process, the scheduler may choose to wake up an idle core and assign the task to it. In that case, the runtime resumes the execution on that core. Otherwise, the task will eventually resume when there is a core available for it.

4.4 Polling in Nanos6

Nanos6 invokes the polling callbacks both at periodic intervals and opportunistically. The runtime has a thread dedicated to management operations, which processes the list of callbacks every 1 ms. Performing calls at regular intervals allows it to support implementations that require them to guarantee progress.

In addition, worker threads serve the list of callbacks before letting their core become idle. The implementation allows several threads to process the list concurrently. However, at this time we assume that callbacks may not support concurrent execution.

The current polling API and its implementation in Nanos6 are at an early stage. In the future we may add options related to callback concurrency and quality of service requirements.

4.5 Genericity

While we focus on OpenMP tasks and MPI, the API 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.

5 MPI

In this Section we present the modification of the MPI model required to improve the interoperability between MPI and OmpSs. The ability to notify the task runtime that a task can be paused, and to permit that task to be resumed by the runtime, is specific to each runtime. However, the changes needed to the MPI semantics to benefit from the pause/resume ability could be useful in other over-subscription scenarios. For example, in Cooperative Multithreading (CMT), one user-level thread could explicitly yield to others during a blocking MPI operation and be resumed only when the MPI operation is ready to be completed. Similarly, this mechanism could be used in any form of Asynchronous Multi-Tasking (AMT) environment, such as OpenMP tasks, HPX-5, or PaRSEC.

The key components of the MPI changes proposed in this paper are: 1) introduction of an API to notify that an MPI operation has blocked/unblocked; 2) modifications to the progress rule for MPI communication operations in multi-threaded and multi-tasking; and 3) creation of an additional thread support level to expose a provided/requested "opt-in" mechanism for both users and MPI libraries.

5.1 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. Figure 3 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). 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.

```
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 }
```

Figure 3: Implementation of the MPI_RECV function in the interoperability library and the polling function executed periodically by the runtime system

5.2 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. 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 those 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 deadlock 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 pause/resume API, the first blocking function call can pause the calling task, enter the task-based

```
1  int *sentinel; // Sentinel used to serialize communication tasks
2
3  int main(int argc, char * argv[]) {
4      int provided;
5      MPI_Init_thread(&argc, &argv, MPI_TASK_MULTIPLE, &provided);
6      if (provided == MPI_TASK_MULTIPLE) sentinel = 0;
7      else sentinel = (int *) 1;
8
9      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 }
```

Figure 4: Portable initialization using MPI_TASK_MULTIPLE

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 function. These MPI functions were *n*ot executed in some order—their effects (as well as their execution) were interleaved. Therefore, the "executed in some order" requirement in the MPI Standard [10] should 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 pause/resume 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 pause/resume API proposed in this paper. 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 pause/resume functionality via the MPI_Init_thread call and check whether the underlying MPI library provides it.

Figure 4 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.

5.3 Native MPI Implementations

The modifications in an MPI library in order to use the proposed API are relatively simple and nonintrusive. In this section we describe the two approaches we have implemented.

5.3.1 Interop(mpich): Interoperability Calls within the MPI Runtime. The modifications in this case mainly involve the introduction of a block_current_task call during blocking MPI calls and an unblock_task call once the corresponding blocking operation is determined to have finished. The former should be introduced after performing the appropriate networking interactions—for the sake of performance, and if this information is available within the MPI implementation, only in case the underlying networking operations returned reporting deferred completion. The latter, on the other hand, is called during the actions performed to set the status of a pending request as completed.

Since the task blocking call actually blocks the execution of the calling task context until the corresponding unblocking operation is issued—yielding execution control back to the tasking runtime—this requires the MPI implementation to leverage an asynchronous progress thread to ensure all request completions are adequately handled and hence the unblock_task function is properly called. Note that the MPI progress engine has to be aware of the appropriate task to unblock, which should be easily accomplished, e.g. by attaching the blocking context to the internal request information before calling the blocking function. Figure 5 illustrates in pseudocode the described proposed implementation.

We advocate that this interoperability option should pose very low intrusiveness into MPI implementations. First, the required additions are highly localized and comprise just a few lines of source code. Second, the additional proposed code is easily protected by a new threading level (MPI_TASK_MULTIPLE) which is checked and selected by applications according to the current MPI_Init_thread semantics. Last, most MPI implementations already feature the necessary asynchronous progress thread.

5.3.2 Interop(polling): Leveraging the Progress Engine from the Tasking Runtime. Since the former implementation may suffer from oversubscription due to the additional MPI progress engine thread, we propose alternatively leveraging the Nanos6 polling service to make MPI progress. On top of the native MPI implementation, in MPI_Init_thread we insert a call to the function defined in Section 4.2 to register in Nanos6 a call to the MPI progress engine as a polling service. Similarly, we modified MPI_Finalize to unregister the MPI polling service in Nanos6 before program termination.

```
1  int MPI_Recv(...) {
2      ...
3      if (!request_is_complete(req)) {
4          req->blocking_ctx = get_current_blocking_context();
5          block_current_task(req->blocking_ctx);
6          ...
7      }
8      ...
9  }
10
11 void request_complete(req) { // Called by the progress engine
12     ...
13     unblock_task(req->blocking_ctx);
14     ...
15 }
```

Figure 5: Pseudocode illustrating main modifications in an MPI implementation to leverage the proposed API

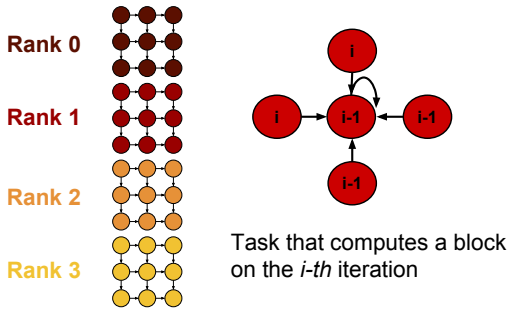

Figure 6: 2-D matrix of 3×12 blocks split in four ranks. On the hybrid versions, for each iteration a task is created to update each block using values of both current (top and left blocks) and previous (current, right and bottom blocks) iterations

6 EVALUATION

In this section we provide an in-depth evaluation of the programmability and performance of our proposal to improve MPI and OpenMP interoperability. We analyze our results based on two benchmarks: an iterative Gauss–Seidel method and a mock-up of a meteorological forecasting application. We have used up to 64 compute nodes of the Marenostrum 4 supercomputer to run the experimental validation. Each compute node is equipped with 2 sockets of Intel Xeon Platinum 8160 CPUs, with 24 cores each, totaling 48 cores per node, and 96 GB of main memory. The interconnection network is based on 100 Gbit/s Intel Omni-Path HFI technology. We have used the latest stable release of MPICH (3.2.1) and OmpSs-2 (17.11).

6.1 Gauss–Seidel

In this section we use the iterative Gauss–Seidel method [6] to solve the Heat equation [5], which is a parabolic partial differential equation that describes the distribution of heat in a given region over time. We have developed five versions of the Gauss–Seidel method for 2-D matrices. The next two are MPI-based:

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

The rest are hybrid MPI+OmpSs versions which divide the matrix into squared blocks and these are distributed across MPI ranks. The left-hand side of Figure 6 shows how a domain of 3×12 blocks would be split across four MPI ranks. These hybrid versions are:

- *Fork-Join*: This is a hybrid version with a sequential communication phase and a parallel computation phase. The communication phase uses synchronous primitives to exchange boundaries among neighbours as in *Pure MPI*. On the computation phase, a task is created to update each block using the top and left blocks of the current iteration, and the current, left and bottom blocks of the previous iteration, as shown in Figure 6. Tasks use fine-grained dependencies to exploit the spatial wave-front parallelism. However, there is a global synchronization point after each computation phase that prevents this version from exploiting parallelism across iterations (temporal wave-front).
- *Sentinel*: A hybrid version where both communication and computation are implemented using tasks. The communication phase uses tasks to execute the synchronous MPI primitives that exchange boundary blocks among neighbours. These communication tasks are serialized by a sentinel dependency to avoid deadlocks (as explained in Section 5). This version avoids the global synchronization (*taskwait*) by leveraging fine-grained dependencies between computation and communication tasks.
- *Interop*: This version uses the MPI_TASK_MULTIPLE multithreading level proposed in this paper to avoid the serialization of communication tasks. This is the only difference with the *Sentinel* version. It has been evaluated with the interoperability library presented in Section 5.1 and the two modified versions of MPICH described in Section 5.3.

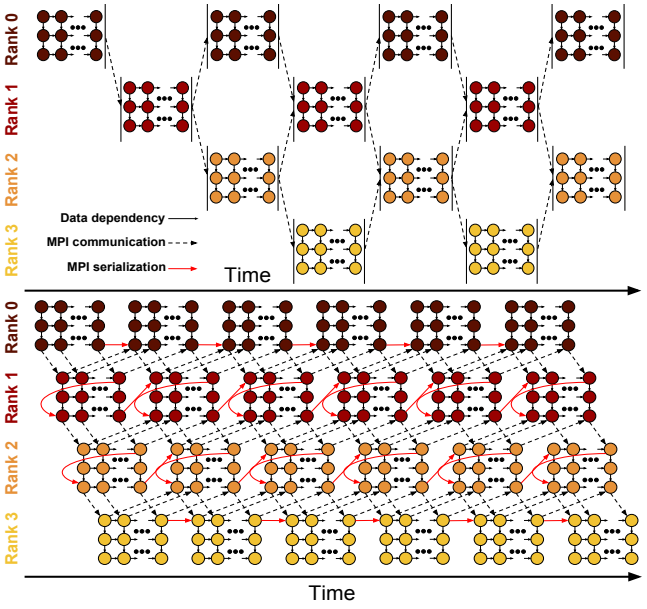

Figure 7: Above: dependency graph for *Pure MPI* and *Fork-Join*. Below: dependency graph for *N-Buffer MPI*, *Sentinel* (with red dependencies) and *Interop* (no red dependencies)

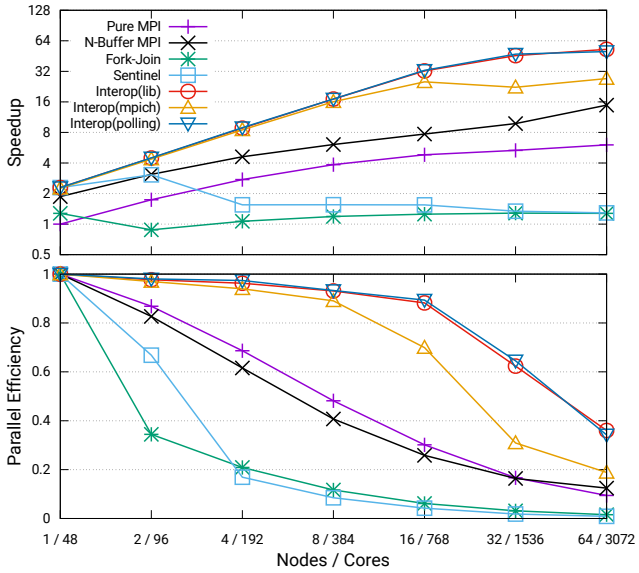

Figure 8: Speedup and parallel efficiency of the Gauss–Seidel strong scaling with 64K x 64K total elements

The three variants are named *Interop(lib)*, *Interop(mpich)* and *Interop(polling)*, respectively.

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

Figure 7 compares the dependency graph of *Pure MPI* and *Fork-Join* (above) and *N-Buffer MPI*, *Sentinel* and *Interop* (below). For the sake of clarity, both graphs have been simplified by showing up to the first six iterations, fusing the explicit communication tasks with the tasks that compute boundary blocks and also other redundant dependencies such as anti-dependencies. In the *Pure MPI* and *Fork-Join* versions, the execution of each iteration inside an MPI rank depends on the completion of the previous iteration of its neighbor MPI ranks, which results in a strong serialization effect that affects the execution of the whole program.

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

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

Pure MPI and *N-Buffer MPI* experiments have been performed using 48 MPI ranks per node. Hybrid versions have used 1 rank per node and 48 OmpSs threads per rank. The upper part of Figure 8 shows a strong-scaling study of the five versions using the performance of *Pure MPI* running on one node as a baseline. On a single node, all hybrid versions experience higher performance than *Pure MPI*. When the hybrid versions run on a single node (one rank), the MPI primitives are completely avoided. Thus the rigid serialization effect introduced by MPI is fully removed and these versions can fully exploit the spatial and temporal wave-front parallelism. It is worth noting that the *Fork-Join* version is significantly slower than the other task-based versions due to the global synchronization point after each iteration that prevents the exploitation of the temporal wave-front. As we increase the number of nodes, the performance of the *Pure MPI* version also increases, but the scalability is clearly sub-optimal. On the other hand, both *Fork-join* and *Sentinel* stop scaling at two and four nodes, respectively. Note that these versions are the only that can be easily implemented with current OpenMP and MPI standards.

The *N-Buffer MPI* version outperforms all previous versions since it avoids the strong serialization of iterations between ranks, which is observed in *Pure MPI* and *Fork-Join*. In addition, this version allows to overlap computation

and communication phases. However, the scalability is still sub-optimal and it is difficult to implement.

The three *Interop* versions have good scalability with up to 32 nodes. With 64 nodes the curve flattens because the problem size is too small to get sufficient parallelism to exploit 48 cores. The *Interop(lib)* and *Interop(polling)* versions reveal identical performance, since both approaches implement a similar strategy where both runtimes cooperate to check for the completion of MPI requests, registering a polling service on the Nanos6 runtime that ensures progress by testing pending requests periodically. On the *Interop(mpich)* version, the helper thread used by MPICH also ensures application progress but, since it runs on the same cores as the worker threads of the Nanos6 runtime, performance degrades due to oversubscription.

The lower part of Figure 8 shows the parallel efficiency of all five versions. In this case each version uses as a baseline its own performance on a single node. From 1 to 16 nodes the efficiency of the three *Interop* variants is almost the same, but then it quickly decreases, since the problem size becomes too small to feed all the cores. The parallel efficiency of *Pure MPI* and *N-Buffer MPI* steadily decrease from 1 to 0.1 at 64 nodes. *Fork-Join* and *Sentinel* have a big drop of parallel efficiency at two and four nodes, respectively.

Figure 9 shows five traces of *Pure MPI*, *N-Buffer*, *Fork-Join*, *Sentinel* and *Interop(lib)*, respectively, running on four nodes (192 cores) with the same time-scale. The traces show the time-line on the X axis and the MPI ranks/OmpSs threads on the Y axis. In *Pure MPI* and *N-Buffer* there are 192 ranks; in the other versions there are four ranks—one rank per node—and each rank has 48 OmpSs threads. On the three hybrid versions, the red lines correspond to the execution of the Gauss–Seidel tasks.

On the *Pure MPI* version (Figure 9a) the last rank (191) cannot start computing the first iteration until all the other ranks have completed the first iteration. This introduces a big delay at the beginning that is also symmetrically reproduced at the end.

The same effect can be observed on the *Fork-Join* (Figure 9c) and *Sentinel* (Figure 9d) versions, but in this case there are only four ranks, so only four full iterations are required to have all the MPI ranks working. In the *Fork-Join* version, the global synchronization at the end of each iteration produces a strong serialization effect among iterations (that is the same effect found on the *Pure MPI* version), so one iteration cannot start until the same iteration of the previous MPI rank has been fully completed. Moreover, the global synchronization at the end of the compute phase also limits the available parallelism, so only 8 out of the 48 cores can work in parallel (running computation tasks).

The *Sentinel* version improves over the *Fork-Join* version because one MPI rank can start computing an iteration as soon as the previous rank has completed the computation of the first boundary block of the same iteration. This allows to partially overlap the computation of the same iteration across MPI ranks. However, the artificial dependencies introduced to serialize the communication tasks still hinder the available

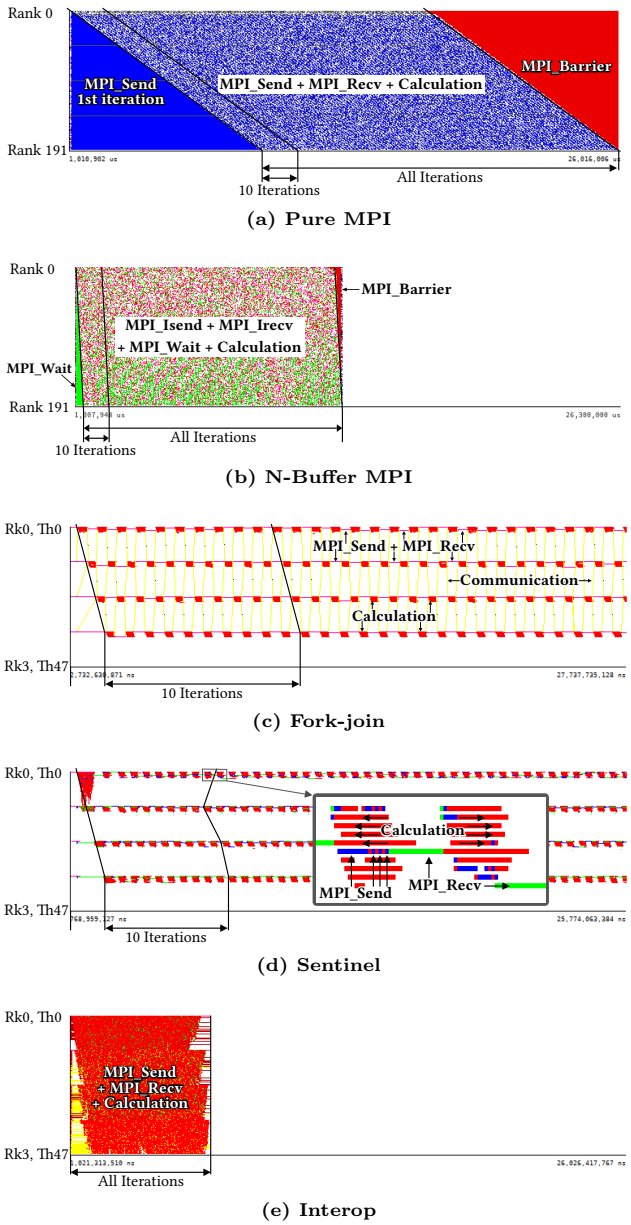

Figure 9: Execution traces with 4 nodes. The Y axis shows MPI ranks/OmpSs threads and the X axis is the time-line

parallelism inside one iteration. In this case, 8 cores can run computation tasks in parallel with another core running a communication task. Although these hybrid versions take less time to complete a single iteration, *Pure MPI* pipelines iterations in a better way and ends up outperforming them in overall iteration throughput.

N-Buffer MPI (Figure 9b) does not show the big delay at the first iteration seen in *Pure MPI* and *Fork-Join*. This is because it exchanges boundaries as soon as possible, thus

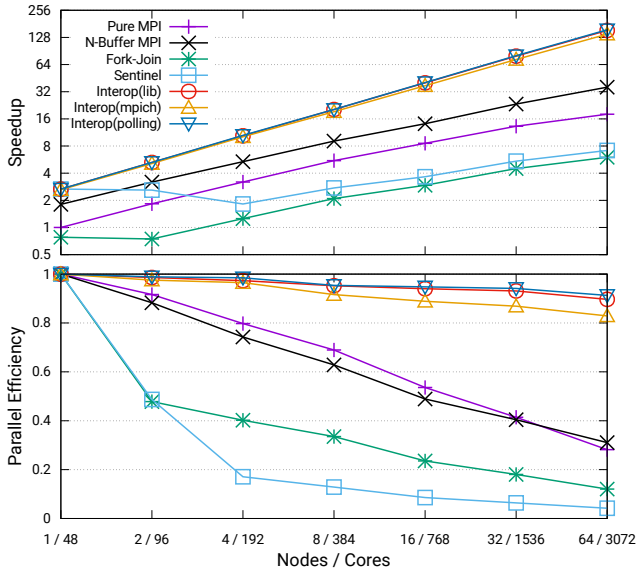

Figure 10: Speedup and parallel efficiency of the Gauss–Seidel weak scaling with 32K x 32K elements per node

ranks can process different blocks from the same iteration concurrently. In addition, it is more flexible than the previous ones due to the use of asynchronous MPI primitives. The aforementioned reasons make this version outperform previous versions both in iteration latency and overall iteration throughput. However, it does not reach *Interop*'s performance and it requires more development effort than them.

Finally, the *Interop(lib)* (Figure 9e) version avoids any global synchronization or serialization of communication tasks, so an iteration can be almost fully overlapped across the ranks. Moreover, this version is the only that can exploit both spatial wave-front and temporal wave-front parallelisms, benefiting from the 48 cores.

To finalize the performance analysis, we have performed a weak-scaling experiment. The speed-up graph (upper part of Figure 10) uses the performance of the *Pure MPI* version on a single node as a baseline for all versions. For the parallel efficiency graph (lower part of Figure 10), each version uses its own performance on one node as the baseline. This experiment shows again the good scalability of the three *Interop* versions that scale linearly up to 64 nodes. The parallel efficiency of *Pure MPI* and *N-Buffer MPI* steadily decreases from 1 to 0.3 at 64 nodes, while *Fork-Join* and *Sentinel* feature a parallel efficiency of 0.4 and 0.2, respectively, with only four nodes.

In addition, we obtained the expected proportional results when performing these experiments in other systems, e.g. Cray ARCHER.

6.2 IFSKer

IFSKer is a mock-up application parallelized with MPI. It mimics the communication and computational patterns of the meteorological forecasting model called Integrated Forecasting System (IFS). IFS employs a spectral transform method which represents fields by using a set of coefficients of a basis function (e.g. a sine function).

The algorithmic structure consists of time-step cycles divided into two phases: grid-point physics computations and Fast Fourier transforms. Data representation and distribution among MPI ranks is different in each stage. Therefore, communication among ranks occurs during the transitions among stages, where the data needs to be transposed and redistributed among the ranks.

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

We have executed the hybrid version with the three interoperability approaches previously explained: *Interop(lib)*, *Interop(mpich)* and *Interop(polling)*. Figure 11 shows the speed-up and parallel efficiency of the two versions on a strong-scaling scenario. In the speed-up graph (upper part) we have used the performance of the *Pure MPI* version running on a single node as a baseline. For the parallel efficiency graph (lower part), each version uses as baseline its performance on a single node. The speed-up graph shows that, on a single node, the performance of *Interop(lib)* and *Interop(polling)* is 4x higher than that of the *Pure MPI* version. However, *Interop(mpich)* is only 2x faster than *Pure MPI*. This can be explained by the oversubscription problems between the helper thread of MPICH and the worker thread of the OmpSs runtime. The *Interop* versions scale linearly up to 16 nodes; after this point the problem size becomes too small. It is worth noting that *Pure MPI* scales superlinearly and with 16 nodes it reaches the performance of the *Interop* versions. This effect is clearly reflected on the parallel efficiency graph (lower part of Figure 11). The parallel efficiency of *Pure MPI* grows until it reaches 3.2x at 8 nodes. Lastly, we have performed a weak-scaling analysis, but figures are not shown due to limited space.

7 STANDARDIZATION

In this section we will discuss the impact of the proposal in the two mentioned standard bodies. First we will discuss what changes we will need to include in the OpenMP standard and how the implementation could be affected. Then we will discuss how these features could be included in an MPI implementation.

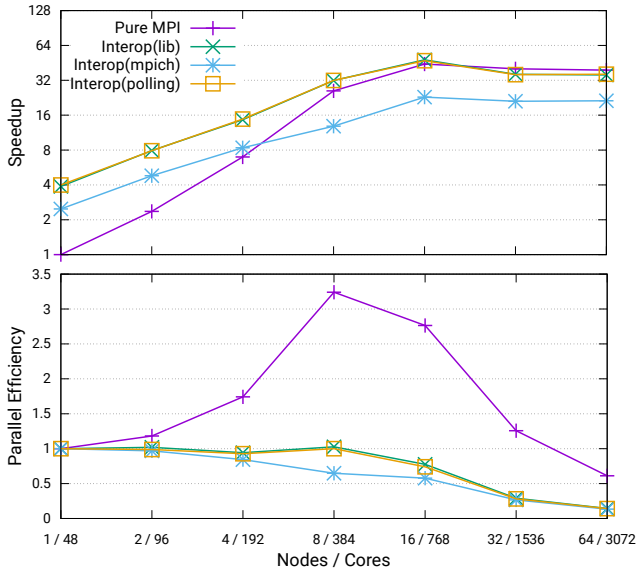

Figure 11: Speedup and parallel efficiency of the IFSKer strong scaling with 653K total gridpoints

7.1 OpenMP

The impact on the OpenMP standard could be measured according to two different fronts: language and implementation. The impact on language affects OpenMP programmers and the way they can interact with the programming model. The impact on the implementation affects compiler–library providers and the way the infrastructure should behave when executing the OpenMP program.

In terms of language the specification should include the four API routines described in Section 4 and provide the functionality of pause/resume tasks and register/unregister polling services.

In addition, this proposal should also impact on the specification's section concerning task scheduling and, more specifically, the inclusion of new task scheduling points (TSPs). The call to the blocking service must be considered as a TSP allowing the implementation to set aside the current task and start/resume the execution of any other task from the ready task pool. The unblock service could also be included as a TSP allowing the scheduler to continue with the execution of the current flow or resume the execution of the unblocked task, but in this case it will be optional.

7.2 MPI

In order for the MPI Standard to be made aware of the task blocking possibility, we propose a new threading level (e.g. MPI_TASK_MULTIPLE that can be checked and selected by MPI applications following the current MPI_Init_thread semantics. As mentioned in Section 5.2, this new threading level will have implications on the definition of the ordering of execution of MPI calls along with the progress rules. In addition, the syntax and semantics of the interoperability API should also be specified within the Standard. This cross-referenced standardization could be accomplished, e.g., by all implied standards pointing to a certain revision of a separate interoperability standard. We note that the inclusion of this tasking interoperability awareness in the MPI Standard would not be specific for OpenMP interoperability, but it would also work with other programming models featuring similar tasking features (such as Cilk or TBB) and implementing the proposed API.

We have begun the formal procedure of standardizing the changes to MPI proposed in this paper by creating an issue for consideration by the Hybrid Programming working group [7].

8 CONCLUSION AND FUTURE WORK

In this paper we have introduced a generic API to pause and resume task execution depending on external events. This API has been used to extend MPI with a new threading level called MPI_TASK_MULTIPLE, which notifies task-based runtime systems when a synchronous MPI operation blocks and unblocks. We have demonstrated how this new threading level improves the programmability and performance of hybrid MPI+OmpSs applications. Moreover, this threading level can be leveraged by any other task-based programming model that implements the pause/resume API.

As future work, we plan to study how the presented pause and resume API performs in MPI RMA operations. We also plan to use it to improve the integration of task-based programming models with other synchronous APIs.

ACKNOWLEDGMENTS

This work has been developed with the support of the European Union Horizon 2020 Programme through both the INTERTWinE project (agreement No. 671602) and the Marie Skłodowska-Curie grant (agreement No. 749516); the Spanish Government through the Severo Ochoa Program (SEV-2015-0493); the Spanish Ministry of Science and Innovation (TIN2015-65316-P) and the Generalitat de Catalunya (2017-SGR-1414).

REFERENCES

[1] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. 2004. Nanos Mercurium: A research compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*, Vol. 8. 56.

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

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

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

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

[6] Anne Greenbaum. 1997. *Iterative Methods for Solving Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.
[7] Daniel J. Holmes and Antonio J. Peña. 2018. Interoperability with task-based runtimes. https://github.com/mpi-forum/mpi-issues/issues/75.
[8] Gabriele Jost, Haoqiang Jin, and Ferhat F. Hatay. 2003. *Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster*. Technical Report. NASA.
[9] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. 2010. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*. 5–16. https://doi.org/10.1145/1810085.1810091
[10] Message Passing Interface Forum. 2015. *MPI: A message-passing interface standard. Version 3.1*. University of Tennessee.
[11] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. 2017. Improving the integration of task nesting and dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 809–818.
[12] Rolf Rabenseifner. 2003. Hybrid parallel programming: Performance problems and chances. In *45th Cray User Group Conference*. 12–16.
[13] R. Rabenseifner and G. Wellein. 2003. Comparison of Parallel Programming Models on Clusters of SMP Nodes. In *Proceedings of the International Conference on High Performance Scientifc Computing*.
[14] Erich Strohmaier et al. 2017. TOP500 Supercomputing Sites. http://www.top500.org/lists/2017/11.