The final publication is available at ACM via http://dx.doi.org/10.1145/3330345.3330379

Optimizing Computation-Communication Overlap in Asynchronous Task-Based Programs

Emilio Castillo[1,2], Nikhil Jain[3], Marc Casas[1], Miquel Moreto[2,1], Martin Schulz[4], Ramon Beivide[5], Mateo Valero[1,2], Abhinav Bhatele[6]

Barcelona Supercomputing Center[1], Polytechnic University of Catalonia[2], NVIDIA, Inc.[3], Technical University of Munich[4], University of Cantabria[5], Lawrence Livermore National Laboratory[6]

Abstract

Asynchronous task-based programming models are gaining popularity to address the programmability and performance challenges in high performance computing. One of the main attractions of these models and runtimes is their potential to automatically expose and exploit overlap of computation with communication. However, we find that inefficient interactions between these programming models and the underlying messaging layer (in most cases, MPI) limit the achievable computation-communication overlap and negatively impact the performance of parallel programs. We address this challenge by exposing and exploiting information about MPI internals in a task-based runtime system to make better task-creation and scheduling decisions. In particular, we present two mechanisms for exchanging information between MPI and a task-based runtime, and analyze their trade-offs. Further, we present a detailed evaluation of the proposed mechanisms implemented in MPI and a task-based runtime. We show performance improvements of up to 16.3% and 34.5% for proxy applications with point-to-point and collective communication, respectively.

Keywords: task-based programming model, computation-communication overlap, mpi

1 Introduction

The end of Dennard scaling and subsequent stagnation of CPU clock frequencies has led to the rise of multi-core systems with large core counts and heterogeneous systems with accelerators [29, 27]. Asynchronous Task-based Programming (ATaP) has emerged as a popular solution to address the challenges of portable and scalable execution on such complex architectures. ATaP models and runtimes, such as OpenMP [23], Charm++ [1], HPX [17], OmpSs [10], and Legion [3] support a data-flow execution model to orchestrate the execution of parallel tasks while respecting their control and data dependencies. These models delegate the responsibility of scheduling work to a runtime system, and thus enable the user to focus on programming aspects related to their problem domain without worrying about cross-platform performance issues. Under the hood, the runtime systems are designed to automatically optimize for different application scenarios and system specifications.

One of the main attractions of ATaP models and runtimes is their potential to automatically expose and exploit overlap of computation with communication. This refers to the benefit that comes from having multiple tasks assigned to a physical core or process – when one task is waiting for messages to arrive, another task can use the idle core for useful computation. On distributed memory systems, inter-node communication in ATaP applications is typically handled by calls to a messaging library (in most cases, MPI). Some models allow explicit calls to MPI whereas in other models, communication primitives are translated to MPI calls by the runtime [10, 26].

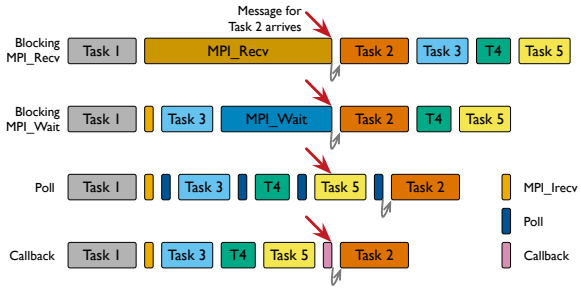


Fig. 1: Early invocation of a blocking `MPI_Recv` by one task can prevent other tasks from making progress (top row). The remaining three rows represent alternatives. In each row, the red arrow marks the arrival in time of an MPI message for a specific task (Task 2).

We have observed that inefficient interactions between ATaP models and MPI limit the achievable computation-communication overlap and negatively impact the performance of parallel programs. Let us consider several common mechanisms used in different ATaP models for making MPI calls from within tasks. In one mechanism, tasks make blocking MPI calls, such as `MPI_Recv`. This prevents other tasks from using the idle core while the task in question is blocked waiting for messages and is clearly inefficient as shown in the top row of Figure 1. A second mechanism, as a potential solution to the above problem, is to use a non-blocking `MPI_Irecv` and `MPI_Wait`. However, this still has the problem of being blocked at the `MPI_Wait` if it is called too early. A third mechanism is to periodically poll for message arrivals; this avoids blocking but can require multiple trials. Thus, none of these mechanisms is perfect and they all waste valuable CPU resources. Despite such issues, the use of MPI as the underlying communication mechanism for ATaP models is attractive, since it represents a convenient portability layer that is available on virtually every high performance computing platform.

In this paper, we address inefficiencies in interactions between ATaP models and MPI. Our high level idea is this – if the runtime system of an ATaP model is aware of the progress or state of communication in MPI, it can make better task-creation and scheduling decisions, and efficiently overlap computation with communication. Our approach tracks certain events in MPI and exposes them to the ATaP runtime system in order to efficiently schedule blocking MPI primitives or initiate a specific request to poll. Further, using our approach, the runtime system can execute tasks that utilize partially received data of an on-going MPI collective operation, thus providing opportunities for computation-communication overlap that were not exposed previously.

We present two mechanisms, similar to the MPI tools interface (MPI_T) [13], for exchanging information between MPI and an ATaP runtime system and analyze their trade-offs: 1. a fast mechanism to poll events when idle using a lock-free queue, and 2. a delivery solution based on callbacks that can benefit from a hardware implementation, shown in the bottom row of Figure 1. These mechanisms allow ATaP runtimes to seamlessly interoperate with MPI by reducing or completely eliminating the need for explicit polling or waiting on specific requests, and instead deliberately invoking the progress engine only when needed, driven by runtime events. The main contributions of our paper are:

- We present a novel approach to optimize interactions between an ATaP runtime system and MPI by exploiting knowledge of MPI internal events.
- We expose new opportunities to overlap MPI collectives with tasks that can compute on partially received collective data.
- We present a detailed evaluation of the proposed ideas using MPI and OmpSs [10], an ATaP model that follows the semantics of OpenMP 4.0 tasks. When compared to state-of-the-art solutions with task-based communication and dedicated communication threads, we show improvements of up to 16.3% and 34.5% for proxy applications with point-to-point and collective communication, respectively.

2 Background and Motivation

Below, we describe the salient features of asynchronous task-based programming models, popular approaches for distributed memory communication in these models, and the challenges that undermine cooperation between ATaP models and messaging libraries.

2.1 Overview of ATaP Models

ATaP models such as OpenMP 4.0 [23] and OmpSs [10], conceive the execution of a parallel program as a set of tasks that may depend upon one another. Typically, the programmer defines code blocks (functions and/or classes) and adds annotations to declare 1) what constitutes a task, 2) what data is used by each task, called input dependencies or input, and 3) what data is produced by each task, called output dependencies or output. Based on this information, the runtime system manages the parallel execution using a Task Dependency Graph (TDG), a directed acyclic graph where

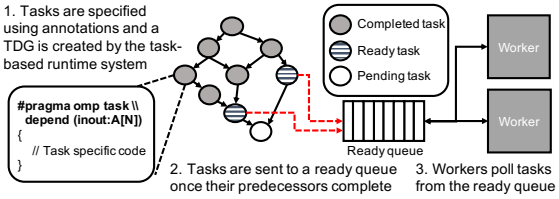


Fig. 2: Execution flow in an asynchronous task-based programming model that uses a task dependency graph.

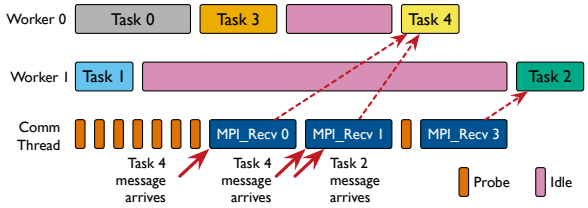


Fig. 3: Communication thread can become a serial bottleneck if one thread is responsible for many workers.

the nodes are tasks and the edges are dependencies between these tasks.

Figure 2 shows a simple example of a TDG. A task is marked as ready or unlocked only when all its predecessors have completed their execution, otherwise it is considered a pending task. Ready tasks are added to a *ready queue* (or another appropriate data structure depending on the scheduling algorithm). When idle, worker threads interact with the scheduler and retrieve tasks from the ready queue for execution. When a task completes, it is marked as such in the TDG and its successors are unlocked. Examples of such a programming model are tasks in OpenMP 4.0 [23] with dependency clause extensions and Legion [3], in which dependencies between tasks are expressed using regions.

This work uses the task constructions of OmpSs [10], which have been adopted as the task extensions for OpenMP 4.0. The programmer creates tasks using pragma annotations with the input and output specified as shown in Figure 2. The compiler replaces these annotations with calls to the runtime system, and the tasks are dynamically created and destroyed during application execution. In this work, we employ a reduced version of Nanos++ 0.10a [10], the runtime of OmpSs, which uses pthreads bound to specific cores as worker threads.

2.2 Inefficiencies in Distributed Memory Communication

The de-facto standard for distributed memory communication is the Message Passing Interface (MPI) [28]. MPI implementations provide an easy-to-use, portable, high-performing abstraction on top of most low-level communication technologies available on distributed memory clusters. ATaP models usually rely on MPI when running in distributed memory environments by explicitly making MPI calls at the source code level, possibly inside tasks or at synchronization points. We call this approach *explicit communication*. Within this class, some runtime systems such as OmpSs [22], rely on invocation of MPI calls within tasks, and require a manual orchestration and deployment of MPI processes. Other programming models such as Charm++ and Habanero, build their communication interface on top of MPI and abstract the MPI process related orchestration. Alternatively, programming systems such as Legion [3] hide communication from the programmer. They let the runtime system detect accesses to remote data and perform the required data transfers. We call this approach *implicit communication*.

In both explicit and implicit styles of communication, some inefficiencies prevent the interaction between MPI and ATaP models from reaching its full potential. ATaP models seek to exploit asynchrony and to overlap communication with computation across tasks to improve performance. In contrast, several MPI features have traditionally been influenced by a bulk synchronous programming model, in which communication and computation occur in phases. This results in performance inefficiencies such as those shown in the top two rows of Figure 1 – resources can remain idle if blocking calls are made early before the messages have arrived.

ATaP models typically deploy communication threads to improve computation-communication overlap. A dedicated thread is made responsible for data transfers in order to avoid blocking worker threads. However, a communication thread does not execute computation tasks, which results in resource underutilization if the thread is assigned a dedicated core. If communication threads are not assigned dedicated cores, they can perform poorly. They can also become a serial bottleneck, as shown in Figure 3, in which the communication thread is responsible for sending, probing, and receiving messages for all workers. In this example, worker 1 is idle for a long time because the communication thread is busy processing messages for task 4 of worker 0.

Another potential source of inefficiency is the implicit global synchronization that MPI collectives impose. While it is possible to overlap computation and

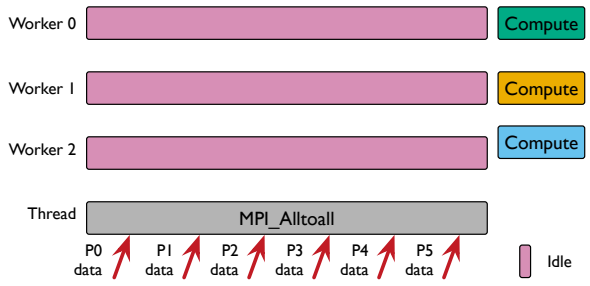


Fig. 4: Tasks that can begin computation with partial data (P0, P1 -¿ worker 0; P2, P3 -¿ worker 1; P4, P5 -¿ worker 2) need to wait for the collective to complete.

collective communication to a certain extent by using non-blocking collectives [15] as standardized with MPI 3.0, it is currently not possible to use partially received data while the collective is in progress. For example, in the `MPI_Alltoall` operation, although data from some processes is received earlier than others, execution of tasks that only need that partial data cannot begin until the collective completes as shown in Figure 4. In this example, although each worker depends only on data from two processes (P0, P1 -¿ worker 0; P2, P3 -¿ worker 1; P4, P5 -¿ worker 2), all workers have to wait until data from all processes is received and the `MPI_Alltoall` call completes.

This work aims at removing the inefficiencies that undermine the cooperation between task-based runtimes and distributed memory communication libraries such as MPI. By exposing information about events happening at the communication layer (e.g., incoming messages, data transfer completions or progress of collective operations) to an ATaP runtime, we believe the runtime system can schedule tasks more efficiently, reduce idle time and maximize computation-communication overlap as a result. Section 3 describes the interface we propose to achieve these goals.

3 Exposing MPI Activity to ATaP Runtimes

We propose to make ATaP runtime systems aware of MPI activity to overcome the performance issues highlighted in Section 2.2. Further, we propose to drive this information sharing by triggering callbacks in hardware or via low-level system software. This section describes the proposed interactions between MPI and ATaP runtimes, and a mechanism to enable computation-communication overlap in the case of collective communications.

3.1 Extending MPI to Support Event Handling

Our approach proposes a set of events triggered by MPI and captured by ATaP runtime systems. To be consistent with the MPI standard, we implement our techniques on top of existing solutions like MPI_T, the MPI Tool Information interface introduced in MPI 3.0 [11], as well as the recently proposed MPI_T_Events extensions [13]. The latter provides the necessary infrastructure for callbacks in MPI, intended for the support of tracing tools, but does not define any concrete events matching the philosophy of MPI_T. In particular, we propose adding the following events to MPI:

- `MPI_INCOMING_PTP` signals the arrival of a point-to-point message. It saves the tag and source of the message, and the associated `MPI_Request` handle, if any. For a message expected to use the rendezvous protocol, this event may indicate the arrival of the control message.
- `MPI_OUTGOING_PTP` signals the completion of a non-blocking point-to-point send operation. It saves the `MPI_Request` handle.
- `MPI_COLLECTIVE_PARTIAL_INCOMING` signals the arrival of some data in the context of a collective communication. It saves the source rank in the communicator being used.
- `MPI_COLLECTIVE_PARTIAL_OUTGOING` signals the sending of some data in the context of a collective communication. It saves the MPI rank of the receiver. When this event is triggered, it is safe to overwrite the corresponding portion of the outgoing buffer.

We implement these events in the context of Intel's MVAPICH 2.2 [31], where PSM2 [7] is primarily responsible for conducting point-to-point communication. Thus, in our implementation, events such as the arrival of an incoming point-to-point message originate at the PSM2 layer, which in turn notifies MPI of the associated point-to-point event. PSM2 uses lightweight helper threads to handle communication, which efficiently share resources with other threads in the system. Event notification to MPI is triggered by these helper threads. On the other hand, the creation of events associated with the progress of collective communication is handled by MPI. In both cases, MPI is responsible for the delivery of the events to the ATaP runtime system.

3.2 Mechanisms for Event Delivery

We consider two mechanisms to deliver the events described in Section 3.1 to the ATaP runtime: a polling-based mechanism and a callback-based approach.

3.2.1 Polling-based Notification

Our first approach is to add a polling interface to MPI based on the MPI_T_Events proposal. An ATaP thread can use this interface to query events at its convenience. When invoked, the polling call checks if an event has occurred and, if so, returns the data associated with the event. This is significantly different from the polling capabilities currently available in MPI through the set of `MPI_Test` calls. These only return the state of a specific request, which is inefficient as it requires users to individually poll on all outstanding requests until one of them is in a desired state. In contrast, our approach returns completed events across all event sources, and therefore prevents unnecessary queries on requests that had not experienced any change in their status. Conceptually, our polling interface can be viewed as an extension of `MPI_Probe`: in addition to information about the incoming message that `MPI_Probe` returns, our extension can also return information regarding events related to non-blocking send/receive requests and collectives.

We propose an additional function that implements the actual polling: `MPI_T_Event_poll (MPI_T_event* event)`. This function returns whether one of our defined events has occurred since its last invocation and, if yes, returns an opaque event object containing information regarding the event. The type of this object is identical to the event object type used in the MPI_T_Events proposal, and hence can be decoded with a matching call to `MPI_T_Event_read`. In our implementation, a lock-free event queue, from the C++ Boost library [5], is used to store the events until they are consumed by the ATaP runtime system. We also modify the specific runtime used in this paper, Nanos++, to use its worker threads to invoke the polling interface. These invocations are done either between consecutive task executions or when worker threads are idle. Figure 5 summarizes the polling-based delivery mechanism described in this section.

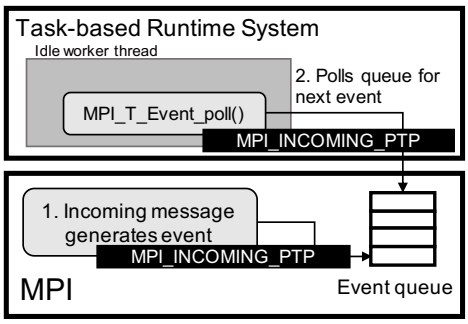


Fig. 5: MPI_T event creation, addition to the event queue, and consumption by a worker thread in polling mode.

3.2.2 Callback-based Notification

Callback-based notification works by associating handler functions with specific events in a way the corresponding handler function is invoked to perform some action once the event occurs. In particular, by having the events described in Section 3.1 handled by callbacks, we release the ATaP runtime system from the need for polling the event queue. For this functionality, we directly rely on the MPI_T_Events proposal [13], which provides generic callbacks mainly intended to implement tracing tools. We use it to track the events described in Section 3.1 and notify the ATaP runtime, which can then associate a handler function by invoking the *MPI_T_Event_handle_alloc* call, as described by Hermanns et al. [13]. Once invoked, the runtime receives an MPI events object, which can be decoded with *MPI_T_Event_read*.

The primary concern with the use of callbacks is the impossibility of knowing beforehand when an event will occur, which makes it impossible to know which thread will handle the event and execute the callback. Thus, to ensure correctness, the implementation of any callback handler must respect some restrictions:

- Callbacks should not take any locks that may already be taken by the thread executing the callback.
- Callbacks have restricted MPI capabilities as described in [13].
- Callbacks should not be nested.

These restrictions can be easily satisfied in our context. The primary purpose of event notification is to satisfy an dependency for a task and, once all dependencies for the task are met, push it to the scheduler. These actions just require locks that maintain the state of runtime system metadata as well as locks that control the interaction with the scheduler queue. Neither of these locks can be taken by a worker thread if it executes the callback while invoking MPI inside a task. Similarly, a worker thread does not hold any lock if it executes the callback while invoking MPI when idle to progress communication. If callbacks are invoked by MPI helper threads, no runtime system locks will be taken by them. These actions also do not require any calls to MPI and thus cannot invoke other callbacks. Thus, inside the ATaP runtime, we use callbacks to identify, unlock and push ready tasks to the scheduler.

Hardware-induced callbacks: Callbacks can also be triggered as user-level interrupts by the Network Hardware Interface (NIC) when it detects associated

MPI events. Having the NIC detect events and trigger callbacks can improve both flexibility and performance by providing a more accurate and fast notification of an incoming message, message delivery completion, and RDMA operations. For reference, Keppitiyagama et al. [19] showed that Myrinet hardware with a programmable network processor could drive the MPI progress engine and networks like the Scalable Coherent Interface included remote doorbell capabilities. In addition, some of the current Intel Xeon processors already integrate the OmniPath NIC in the processor package. We hope that this work will lead to the addition of such capabilities in hardware. In this paper, we emulate this capability by using a thread running on a dedicated core to monitor MPI state.

3.3 Changes to the OmpSs runtime

Typical implementations of ATaP models such as OmpSs may indicate to the underlying runtime that a task performs/depends on communication, but do not require exposing more details. For the ATaP runtime system to match tasks with notifications, more information is required. To this end, we extend our ATaP model, OmpSs, to notify its underlying runtime, Nanos++, of messages being sent or received, as well as of MPI requests that are accessed in a task. This information is used to create a task dependency on the corresponding event. In our implementation, MPI calls inside tasks are identified by the OmpSs compiler, which introduces code to inform Nanos++ of the MPI call and its arguments such as source/destination rank and `MPI_Request` object.

Enabled by this communication-related information, the Nanos++ runtime system creates dependencies between tasks and their corresponding MPI_T events. This implies, for example, that a task performing a blocking MPI call is not allowed to run until the corresponding `MPI_INCOMING_PTP` MPI_T event, related to the task has taken place. Similarly, a task invoking a blocking `MPI_Wait` is not allowed to execute until the completion event associated with the incoming or outgoing message request takes place (Figure 6 (a)).

When an event is delivered to Nanos++, either via polling or execution of a callback, it is used to unlock the associated task for execution as depicted in Figure 6 (b). For every task with an event dependency, Nanos++ contains an entry in a reverse look-up table based on the identifiers (message tag, source, or the `MPI_Request` object). This table is used to identify the task, which is then scheduled for execution if all its dependencies are met. In this way, by waiting for communication events to occur before the tasks are scheduled, we are able to avoid unnecessary blocking of

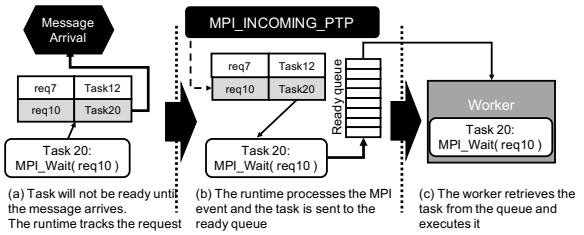


Fig. 6: The runtime creates dependencies between tasks and corresponding MPI_T events and the notification of an MPI_T event unlocks a task waiting for it.

worker threads.

Even under this scheme, use of blocking MPI calls for sending or receiving can still block threads unnecessarily when a rendezvous protocol is deployed in MPI. For example, a receiving task is unlocked upon the arrival of the control message of the sender-initiated rendezvous message. Following this, the task will be active and the thread will be blocked for the time the message data is transferred over the network from the source. We recommend that, in such situations, non-blocking send/receive should be used in the task, and another task with an associated `MPI_Wait` should be marked for execution when the actual data arrives.

3.4 Overlapping Computation with Collectives

So far, we have discussed exposing point-to-point communication related MPI events to an ATaP runtime system to potentially increase its responsiveness and reduce the time for which threads remain blocked. We now switch our attention to exploiting computation-communication overlap for MPI collectives. MPI 3.0 introduced non-blocking collectives, akin to non-blocking point-to-point operations, to allow programmers to perform computation while the collective progresses in the background. However, like point-to-point operations, this can be restrictive due to the need for a wait or frequent test calls.

Traditionally, an MPI collective call is viewed as a monolithic operation whose intermediate progress is not exposed to the programmer. We propose to notify ATaP runtime systems of partially received data so that they can trigger dependent computations as early as possible and maximize computation-communication overlap. In particular, we focus on many-to-one or many-to-many style collectives such as `MPI_Alltoall`, `MPI_Gather`, and `MPI_Allgather`, in which tasks can be run even when partial data has been received.

Figure 7 shows an example in which an all-to-all op-

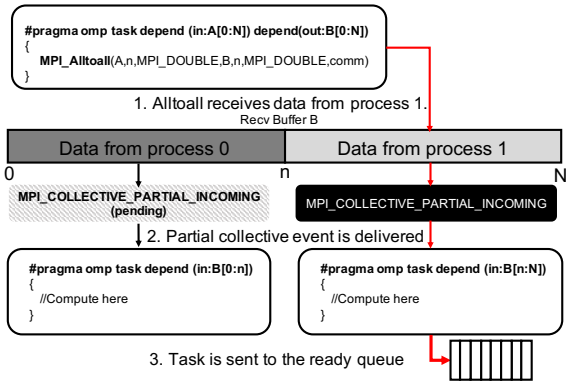


Fig. 7: Unlocking of the right task when partial data from process 1 is received leading to a collective callback.

eration is used to share data among tasks on different nodes. Tasks are created in this example such that some tasks need only part of the data received during the collective to start execution. For example, the left task depends only on data received from process 0, while the right task depends on data from process 1. With the existing MPI semantics, all these tasks will be ready for execution when the entire collective completes and data from all processes is received. However, as several collectives in MPI are typically implemented using point-to-point communication, it is likely that data required to unlock one of these tasks is received earlier than the remaining data. Thus it is possible to start the execution of one task earlier, as shown in the figure.

In order to enable computation overlap with collectives, we add two events: `MPI_COLLECTIVE_PARTIAL_INCOMING` and `MPI_COLLECTIVE_PARTIAL_OUTGOING`. MPI can use these events to notify the runtime when part of the data expected in the collective has arrived or has been sent, respectively.

The runtime system is already aware of the memory locations that a task reads or writes, as they are specified in the task creation pragma (see examples in Section 2.1 and Figure 7). With our extensions of Section 3.3, the runtime system also knows the send/receive locations and the volume of the data sent/received in the collectives. Hence, when the `MPI_COLLECTIVE_PARTIAL_`$*$ event is triggered, the runtime system matches the partial data received with the task that depends on it, and if all its dependencies are satisfied, executes it without waiting for the collective to finish. Non-blocking collectives can also benefit from this approach, since even for them, there is no existing mechanism to signal when it is safe to use partial data.

4 Experimental Setup

In this section, we describe the platform and the parallel benchmarks we consider in our experimental campaign.

4.1 Platform Used for Experiments

We use the MareNostrum 4 supercomputer at Barcelona Supercomputing Center for running our experiments. MareNostrum consists of 3,456 compute nodes; every node has two Intel Xeon Platinum 8160 processors each with 24 cores and 96 GB of DDR4-2667 main memory. The interconnection network is a 100 Gb Intel OmniPath full bisection fat-tree. The software stack includes MVAPICH 2.2 running on top of Intel PSM2 with our modifications. All benchmarks are written in C++ and compiled with gcc 7.2.0. As the ATaP model, we employ a reduced version of Nanos++ 0.10a [10], the runtime of OmpSs, which uses pthreads bound to specific cores as worker threads. We experimented with several worker threads per process and process per node combinations ranging from 32 processes of 1 thread to 1 process of 32 threads. We found that 8 threads per process is optimal for the baseline due to the bottlenecks of MPI multi-threading support. Thus, for all benchmarks, 4 MPI processes are spawned per node, each of which creates 8 worker threads. Node count is varied from 16 to 128.

4.2 Point-to-point Benchmarks

For point-to-point communication, we consider two stencil benchmarks using task semantics. The first benchmark is based on HPCG [9], a multi-grid Conjugate Gradient solver with a Gauss-Seidel preconditioner. HPCG uses a 27-point stencil where every block performs a total of 11 halo-exchanges with its neighbors in each iteration due to the preconditioning step. In addition, an `MPI_Allreduce` is performed at the end of each iteration. The resulting communication pattern of HPCG is shown in Figure 8 (left), where darker colors display a larger communications volume. The `MPI_Allreduce` pattern is represented by a light background color as it just involves communication of a scalar value among all the nodes. In our experiments, we apply weak scaling and solve global problem sizes of $1024 \times 512 \times 512$, $1024 \times 1024 \times 512$, $1024 \times 1024 \times 1024$ and $2048 \times 1024 \times 1024$ on 64, 128, 256, 512 MPI processes respectively.

The second benchmark is based on MiniFE, a finite element solver using a non-preconditioned Conjugate Gradient. In contrast to HPCG, MiniFE only performs a single halo exchange per iteration and has a more irregular communication pattern between pro-

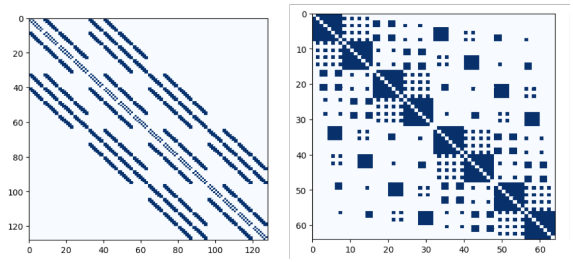


Fig. 8: Communication patterns of HPCG (left) and MiniFE (right). Darker colors indicate higher volume of communication among MPI processes.

cesses, as shown in Figure 8 (right). The lack of a preconditioning step in every iteration reduces the total number of tasks, thus providing insights on how the proposed mechanisms behave in environments with less overlap opportunities. Similar to HPCG, each iteration of MiniFE also ends with an `MPI_Allreduce`. As input, we use $1024 \times 512 \times 512$, $1024 \times 1024 \times 512$, $1024 \times 1024 \times 1024$ and $2048 \times 1024 \times 1024$ unstructured implicit finite volumes.

In both benchmarks, each process is assigned a sub-block of the initial 3D domain. Each sub-block maps to a set of rows in the sparse matrix to be solved by the conjugate gradient step. To overlap communication and computation among tasks, the sub-block assigned to a process is further overdecomposed into smaller sub-blocks. We consider decomposition factors between $1\times$ (one sub-block per core) and $16\times$ (16 sub-blocks per core), and report execution time for the best performing decomposition for every configuration.

4.3 Collective Communications Benchmarks

We consider several benchmarks that intensively use collective communication. The first benchmark is a two-dimensional (2D) FFT using a parallel zero-copy algorithm [14]. In 2D FFT, we initially divide the matrix among MPI processes using row-wise 1D block partitioning. This enables the creation of tasks executing 1D FFTs for each row. Next, we perform an `MPI_Alltoall` to transpose the matrix. Finally, 1D FFTs are calculated again for each row of the transposed matrix. The matrix is transposed during communication by using MPI derived datatypes, as described by Hoefler and Gottlieb [14].

When transposing the matrix using the `MPI_Alltoall` collective with derived datatypes, each process receives partial row data from every other process. Typically, it is not possible to overlap the collective with the computation tasks because tasks computing the 1D FFT require the entire matrix row. However, it is possible to further divide the 1D FFT into smaller tasks that process data blocks as soon as they are received. The block size is set to be the size of a row divided by the number of MPI processes, allowing the execution of partial 1D FFT tasks as the `MPI_Alltoall` progresses. We consider the 2D FFT for square matrices with $16384^2, 32768^2, 65536^2, 131072^2$, and 262144^2 elements.

The second benchmark is a three-dimensional (3D) FFT. Initially, the 3D volume is divided into subsets created by 2D decomposition in y and z dimensions. 1D FFT computations are performed along the x-axis, and are followed by `MPI_Alltoall` calls within subcommunicators defined along the y-axis. This transposes the volume such that the subsets are now decomposed in the x and z dimensions, and 1D FFTs along the y-axis are performed. Next, `MPI_Alltoall` calls within the subcommunicators defined along the z-axis transposes the grid to create the final set of subdomains in which 1D FFT can be performed along the z dimension. We have chosen a 2D decomposition over a 1D decomposition because of its better scalability in terms of memory and communication [25]. For 3D FFT, we test cubic volumes with $1024^3, 2048^3$, and 4096^3 elements.

We also consider two MapReduce [8] applications — a simple word-count algorithm and a dense matrix vector product. In MapReduce, the input data is split into independent chunks processed by the map tasks in parallel. Each map task produces a series of tuples in the form (Key, Value) (K, V). The N values associated to the same K_i are coalesced in a list $(V_{i,j})_{0 \leq j < N}$. Each process sends its $(K_i, (V_{i,j})_{0 \leq j < N})$ tuples to another process determined by a function of the key $Node_{id} = hash(K_i)$ in the shuffling stage. Shuffling is done using `MPI_Alltoallv`. Finally, every process applies the reduction operation to the list of values $(V_{i,j})_{0 \leq j < N}$ associated with each key K_i. In Word-Count, we consider random texts with 262, 524 and 1048 million words, while in the matrix vector product, we consider square matrices with 1024^2, 2048^2, and 4096^2 elements.

We implement a baseline MapReduce framework that uses `MPI_AlltoAllv` for data shuffling in OmpSs and MPI. In it, the reduction of a single key list of values is a serial operation, while the reduction for different keys can be performed in parallel. However, using proposed work, reduction tasks can start to execute as soon as the `MPI_AlltoAllv` receives data from any process. This leads to the creation of several parallel reduction tasks for the same key as multiple list of values for a single key might be received from different processes.

5 Performance Evaluation

In this section, we evaluate the performance impact of using our proposed MPI events to drive the task execution in the OmpSs programming model, as described above.

5.1 Results for Point-to-point Benchmarks

We compare the performance of baseline task-based executions of HPCG and MiniFE with executions in five other resource-equivalent scenarios. These executions are performed on 16, 32, 64, and 128 nodes, with four MPI processes running on each node. Each MPI process can use eight cores, which is the configuration that minimizes the execution time for the baseline implementation. In the baseline scenario, eight worker threads per MPI process are responsible for executing the computation as well as the communication tasks, and for invoking the MPI progress engine. It is worth mentioning that this is the only out-of-the-box configuration available in OmpSs+MPI and OpenMP 4.0+MPI.

We evaluate two additional baseline scenarios with communication threads: one in which we add a communication thread that shares hardware with worker threads on available cores, i.e., eight worker threads and one communication thread share eight cores (CT-SH), and the other in which a dedicated core is assigned to the communication thread, and the computation tasks are executed on the remaining seven cores by seven worker threads per MPI process (CT-DE). Such solutions represent the state-of-the-art communication model of most common ATaP models. However, as it was mentioned before, OmpSs and OpenMP do not integrate a communication thread in their available releases so we hope that this work can motivate the default inclusion of such configurations for hybrid applications.

We compare the baseline scenarios with three scenarios that represent variants of our proposals : i) polling-based notification (EV-PO), where worker threads poll for MPI_T events when idle (Section 3.2.1); ii) callback-based event delivery in software (CB-SW); and iii) a hardware-induced callback-based event delivery (CB-HW, Section 3.2.2). The hardware support is emulated by using an additional thread running on a dedicated core that monitors the internal status of MPI and PSM2 to trigger the callbacks; this core never executes a task.

Figure 9 (a) presents the speedups obtained with the scenarios described above normalized to the baseline for HPCG. The use of a dedicated core for communication (CT-DE) provides a speedup ranging from 12.7% to

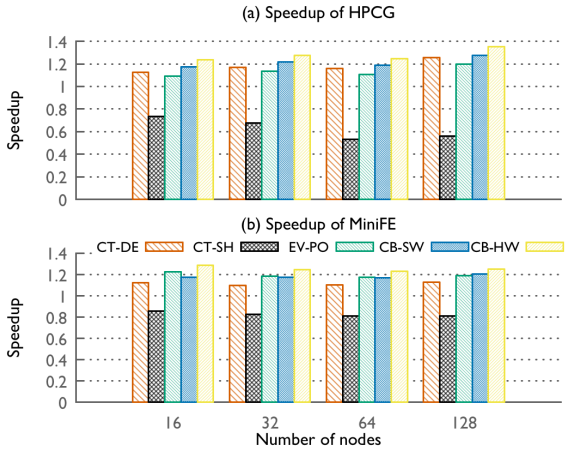


Fig. 9: Speedup for (a) HPCG and (b) MiniFE over the baseline implementation on 16, 32, 64 and 128 nodes with 64, 128, 256 and 512 MPI processes, each with 8 threads (cores).

25.7% with respect to the baseline approach for the 16 to 128 node configurations. These improvements are due to the early execution of communication tasks enabled by CT-DE as well as due to the avoidance of blocking in worker threads. On the other hand, when the communication thread is not assigned a dedicated core (CT-SH), performance degrades by up to 44.2%.

The polling-based event notification (EV-PO) mechanism yields slightly lower performance improvements than CT-DE (9.25%, 13.5%, 10.5% and 19.7%) for the four node counts. This is caused by computation tasks in HPCG delaying the polling for MPI events, and thus delaying communication. Hence, benefits of event notifications are sometimes neutralized by the lack of progress. With software callbacks we are able to unlock the tasks as soon as the events arrive and performance improvements rise to 17.4%, 21.7%, 19.0% and 27.4% respectively. Hardware-based support for event detection and triggering of callbacks (CB-HW) is further able to overcome the delays due to long running computation tasks and improves performance by 23.5% 27.6%, 24.3% and 35.2% (up to 9.5% over CT-DE). In CB-HW, as soon as an MPI_T event occurs, the associated task becomes ready for execution. Moreover, small granularity of the tasks doing the pre-conditioning of the matrix require communication to be done as early as possible, thus improving the performance over the baseline as the node count increases.

The time spent in communication in HPCG is approximately 10.7% of the total time executing MPI calls without event notification. This time is reduced to 3.6% when using callbacks as the delivery mechanism. This

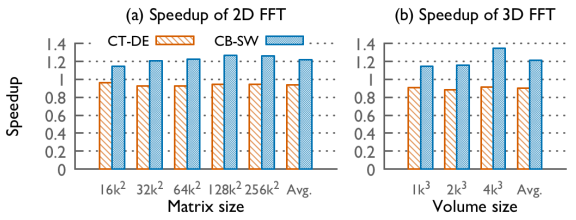


Fig. 10: Speedup for 2D FFT and 3D FFT over a baseline implementation on 128 nodes.

is due to MPI calls being only invoked when the associated event has arrived, and effectively minimizing waiting time. Moreover, as only ready MPI calls are executed, the time that was spent on checking the status of MPI requests is now devoted to computation leading to the aforementioned speedups.

Figure 9 (b) shows similar trends for MiniFE as HPCG. The key difference is that due to the smaller granularity of computation tasks in MiniFE, the polling-based event notification (EV-PO) is able to outperform the scenario with a dedicated communication thread (CT-DE). Aided by the relatively shorter delays for polling, the improvements for EV-PO are 22.5%, 18.6%, 17.5% and 19.2% in comparison to 12.2%, 9.5%, 10.3% and 13.0% for CT-DE. As was the case for HPCG, the presence of hardware support for callbacks further improves the performance and achieves speedups of 28.4%, 24.6%, 22.8% and 25.2% over the baseline and up to 16.3% over CT-DE. The lack of a pre-conditioner in MiniFE yields a lower communication frequency than HPCG, which does up to 11 neighbor communications per iteration, while MiniFE only does one. This lower communication/computation ratio translates into a constant scalability over the baseline for all node counts. Finally, similarly to HPCG, the baseline time spent in communication is 11.8% of the total execution time and is reduced to 3.3%.

Regarding the overheads of polling and callback based approaches, the average time spent polling for events is 9× and 15× that of callback for MiniFE and HPCG respectively, with polling happening around 100× more times than callbacks in both benchmarks.

5.2 Results for Collective Benchmarks

In this section, we evaluate the performance impact of the mechanism described in Section 3.4 for exposing the potential overlap of computation tasks with MPI collectives.

5.2.1 Fast Fourier Transform

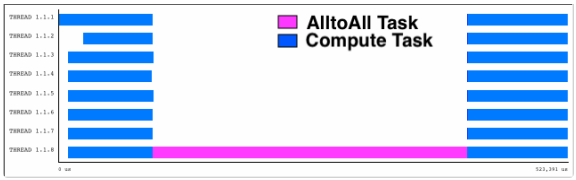
Similar to the point-to-point benchmarks, we compare the performance of 2D FFT and 3D FFT benchmarks for executions in the same five resource-equivalent scenarios and present speedup against the baseline execution on 128 nodes. For 2D FFT, we use square matrices of input sizes ranging from 16384 × 16384 to 262144 × 262144, and cubic volumes of sizes 1024^3, 2048^3, and 4096^3 for 3D FFT.

Since our experimental results did not show significant performance differences between the three event-based scenarios (EV-PO, CB-SW and CB-HW), we only present representative results for CB-SW. These equivalent performance results, as we will show in this section, are caused by the fact that collective calls only block a single worker thread per process. Hence, other worker threads are readily available for either polling MPI for events or executing callbacks in software. Thus, all event-based scenarios are able to promptly handle events and therefore provide equivalent performance. Further, since the performance for the scenario in which the communication thread shares cores with worker threads (CT-SH) never outperforms the scenario in which the communication thread is assigned a dedicated core (CT-DE), we do not show results for CT-SH.

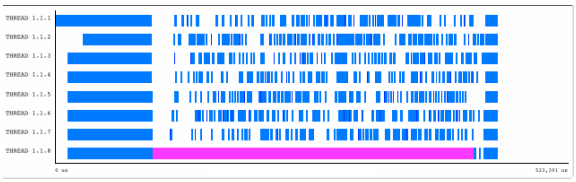
Figure 10 shows that CT-DE consistently performs slightly worse (∼4.0%) than the baseline approach for the 2D FFT benchmark. This is because, with CT-DE, the communication thread does not execute computation tasks once the collective communication finishes, and thus negatively impacts performance. In contrast, CB-SW consistently outperforms the baseline, and provides a maximum 26.8% performance gain for the matrix size 65536 × 65536. The average improvement of CB-SW for 2D FFT considering the five different input data sets is 21.9%.

To illustrate the reasons behind the significant performance improvements achieved by the CB-SW, Figure 11 presents 2D FFT parallel runs over the baseline and the event-based communication regime. Figure 11 (a) shows that all computation tasks need to wait for the `MPI_Alltoall` to finish before they can be executed. In contrast, Figure 11 (b) shows that event-based notification results in some computation tasks executing as soon as the necessary input data is received for them. As a result, we are able to overlap computation tasks which can be executed with the `MPI_Alltoall` that is in progress.

Results for 3D FFT are displayed in Figure 10 (b). Since the 3D FFT benchmark invokes two `MPI_Alltoall` operations instead of one in 2D FFT, it exposes more opportunities for overlapping computation with the collective calls. Therefore, the CB-SW ap-



(a) Baseline with no communication-computation overlap (Naive).



(b) Event-based communication-computation overlap (CB-SW).

Fig. 11: Parallel execution traces showing the effect of collective-computation overlapping 2D FFT. Same time range is shown for both figures. A single MPI process and its threads are shown for the sake of clarity.

proach achieves higher performance improvements for 3D FFT. Overall, CB-SW provides 21.2% average improvement, with a maximum improvement of 34.5% for the 4096^3 sized volume. In contrast, dedicating a core for the communication thread (CT-DE) degrades performance by 9.8% on average in comparison to the baseline approach.

In summary, the event-based exposure of a collective's progress effectively enables an overlap between computation and collective operations in both 2D FFT and 3D FFT, thus providing substantial performance gains.

5.2.2 MapReduce

Next, we evaluate the effect of overlapping computation with MPI collectives for the MapReduce framework. We experiments with two applications: Word Count (WC) and a dense Matrix Vector product (MV). Figure 12 shows the speedup results for both applications over a baseline implementation.

For the WC application, CB-SW provides 7.2% improvement on an average, and a maximum gain of 10.7% for a dataset consisting of 262 million words. In this application, reduce operations are extremely small as they only increase the counter associated with the key. Consequently, as the size of the dataset grows, the map tasks consume a higher proportion of the runtime. As a result, the impact of computation-communication overlap decreases, and the performance gains reduce to

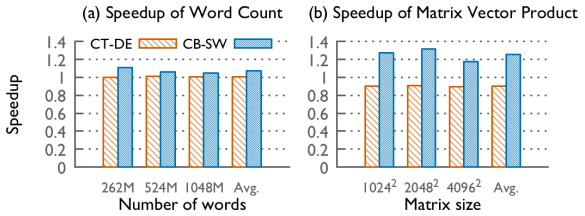


Fig. 12: Speedup for the MapReduce-based Word-Count (WC) and dense Matrix Vector product (MV) benchmarks with different problem sizes.

4.9% for a dataset of 1048 million words.

Unlike the WC application, in the MV application, a similar amount of time is spent in the map and the reduce tasks. This translates to higher impact due to dedicating a thread to communication as well as enabling computation-communication overlap for the reduce tasks. For CT-DE, the inability to use the communication thread to execute tasks degrades performance up to 10.7%. In contrast, performance improvements, ranging from 17.4% to 31.4% are obtained as CB-SW enables overlap of reduce tasks with the `MPI_Alltoallv` collective executed for aggregating keys across processes.

5.2.3 Scalability of Collectives Benchmarks

The results presented in this section are obtained using 128 nodes. We have performed weak-scaling experiments on 16, 32, and 64 nodes and have verified that the speedup trends among the different input-sets correlate in all scenarios with performance differences of at most 4.0% in the case of the FFT 3D application. This allows us to conclude that the collective overlapping benefits hold regardless the node count.

5.3 Comparison with Task-Aware MPI Library

Task-Aware MPI Library (TAMPI) [20] provides MPI with a new level of threading support `MPI_TASK_MULTIPLE`. TAMPI works by intercepting blocking calls to MPI inside tasks and converting them to the non-blocking versions. The task execution is suspended and the `MPI_Request` object is added to a waiting list. This list is iterated by the workers in between task executions polling every request with the `MPI_Test` call and tasks whose requests have completed are rescheduled to keep executing. The key difference is that TAMPI polls every active request while our

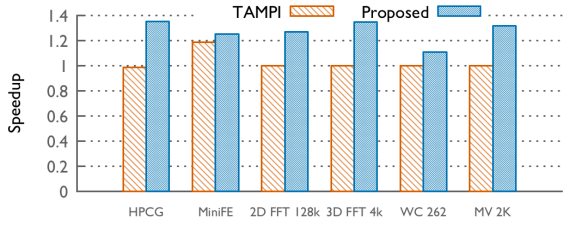


Fig. 13: Performance comparison of our best performing proposal with TAMPI for every benchmark using 128 nodes

proposal only reacts to requests where the MPI layer notifies progression.

Figure 13 shows a performance comparison of our proposed approach with TAMPI using 128 nodes. Results with other node counts show similar trends. For point-to-point benchmarks with a high communication-computation ratio such as HPCG, the worker threads iterate over the list of requests several times delaying the execution of useful computation by polling the status of requests with no changes. Hence, for HPCG, TAMPI performs 1.5% below the baseline. However, for benchmarks with a lower communication-computation ratio such as MiniFE, TAMPI is able to perform well yielding a speedup of 18.7% compared to 25.2% obtained by using MPI_T events. The relatively high computation time compared to communication allows a better overlap of the request polling with the computation task execution.

For the collective benchmarks in Figure 13, TAMPI performs exactly as the baseline solution. TAMPI makes use of MPI calls to test for request completion, but it has no means of accessing information about the partial completion of collectives. This makes TAMPI unable to overlap collective communications with tasks as proposed in this work.

6 Related work

Marjanovic et al. [22] present one of the first works focused on the interoperability of a task-based asynchronous programming models, such as OmpSs, with MPI. Chatterjee et al. [6] goes one step further and integrates MPI within Habanero providing wrapped MPI calls that the runtime system executes asynchronously in dedicated communication threads. The proposed work builds on top of these ideas to make the interaction more efficient. Jain et al. [16] present mechanisms to interoperate MPI and Charm++, and focus on application modules written in these models. In contrast, the proposed work is on the use of MPI as a communication interface by task-based models.

Kamal et al. [18] make use of User Level Threads (ULT) in the MPICH 2 [12] to build an MPI-aware scheduler for coroutines that are swapped in and out for execution depending on the status of the MPI runtime. Lu et al. [21] follow a similar approach by doing the context switch of ULTs inside the MPI to avoid the expensive MPI locking operations. Stark et al. [30] integrate MPI with Qthreads and convert blocking MPI calls to non-blocking calls, using their status to drive the scheduler. Labarta et al. [20] present the Task-Aware MPI library (TAMPI), a similar approach to improve the interoperability between MPI and OmpSs. Our proposal differs from all these approaches in its use of events and mechanisms for exposing those events to an independent task-based runtime system. TAMPI intercepts blocking calls to MPI and converts them into their non-blocking counterpart. The TAMPI library then periodically polls for the completion of the MPI calls and ensures correctness. However, TAMPI is limited to point-to-point communications and requires periodic polling, which affects overall performance as shown in the previous section.

In MPI 3.0, collective and computation overlapping is achieved through the use of non-blocking collectives [15]. However there is no mechanism to perform computation on the partial data already received, allowing only unrelated computation to be done in parallel.

Overall, the proposals in this paper enhance the interoperability between MPI and asynchronous task-based programming models that require explicit communication calls in the application code, such as OpenMP 4.0 [23], OmpSs [10], Codelets [32], Habanero [26], and StarPU [2]. For other programming models such as Cilk [4] and TBB [24], since they do not support data-flow annotations, collective overlapping is not easily attainable. Legion [3] and HPX [17] hide the communication from the programmer by delegating that responsibility to their runtime, while Charm++ [1] provides its own active message-based communication semantics. These runtimes can also benefit from our proposal of exposing MPI internals when built on top of MPI.

7 Conclusion

In this paper, we explored mechanisms to optimize overlap between computation and communication in asynchronous task-based programs executing on distributed memory systems. By exposing information about MPI communication events through the MPI_T events interface to a task-based runtime system, we significantly reduced idle time caused by waiting on specific MPI requests for point-to-point operations. We also pro-

posed a novel scheme to overlap communication with computation on partially received data from collective operations. Our detailed evaluation on a production system provided performance improvements of up to 16.3% in benchmarks with point-to-point communication patterns, and of up to 34.5% in benchmarks with collective communication. Overall, our approach provides a transparent solution that requires no changes to the source code of an application programmed in OmpSs and MPI, yet improves its performance by better exploiting the overlap of computation and communication in such asynchronous task-based programming models.

Acknowledgements

This work is partially supported by the Spanish Government through Programa Severo Ochoa (SEV-2015-0493), by the Spanish Ministry of Science and Technology projects (TIN2015-65316-P and TIN2016-76635-C2-2-R), by the Generalitat de Catalunya (2017-SGR-1414 and 2017-SGR-1328), and by the RoMoL ERC Advanced Grant (grant agreement 321253). M. Moretó and M. Casas have been partially supported by the Spanish Ministry of Economy, Industry and Competitiveness under Ramon y Cajal fellowship numbers RYC-2016-21104 and RYC-2017-23269. E. Castillo has been partially supported by the Spanish Ministry of Education, Culture and Sports under grant FPU2012/2254. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-772400).

References

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. SC, 2014.

[2] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andre Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. In *International Conference on Parallel Processing (Euro-Par)*, pages 863–874, 2009.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the 2012 ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society.

[4] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *International Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, 1995.

[5] Boost C++ Libraries, 2018.

[6] Sanjay Chatterjee, Sagnak Tasirlar, Zoran Budimlic, Vincent Cavé, Milind Chabbi, Max Grossman, Vivek Sarkar, and Yonghong Yan. Integrating asynchronous task parallelism with mpi. In *27th IEEE International Symposium on Parallel and Distributed Processing*, pages 712–725, May 2013.

[7] Intel Corporation. Intel performance scaled messaging 2 (psm2) programmers guide, 2015.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, 2004.

[9] Jack J. Dongarra, Michael A. Heroux, and Piotr Luszczek. Hpcg benchmark: a new metric for ranking high performance computing systems. 2015.

[10] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(2):173–193, June 2011.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 2012.

[12] William Gropp. Mpich2: A new start for mpi implementations. In *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–, London, UK, UK, 2002. Springer-Verlag.

[13] Marc Andre Hemanns, Nathan T. Hjlem, Michael Knobloch, Kathryn Mohror, and Martin Schulz. Enabling callback-driven runtime introspection via mpi_t. In *Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI)*, 2018.

[14] Torsten Hoefler and Steven Gottlieb. Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes. In

Recent Advances in the Message Passing Interface (EuroMPI'10), volume LNCS 6305, pages 132–141. Springer, Sep. 2010.

[15] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society/ACM, Nov. 2007.

[16] Nikhil Jain, Abhinav Bhatele, Jae-Seung Yeom, Mark F. Adams, Francesco Miniati, Chao Mei, and Laxmikant V. Kale. Charm++ & MPI: Combining the best of both worlds. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, IPDPS '15. IEEE Computer Society, May 2015. LLNL-CONF-663041.

[17] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM.

[18] Humaira Kamal and Alan Wagner. An integrated fine-grain runtime system for mpi. *Computing*, 96(4):293–309, Apr 2014.

[19] Chamath Keppitiyagama and Alan S. Wagner. Asynchronous mpi messaging on myrinet. In *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 50:1–50:8, Apr 2001.

[20] Jesus Labarta, Vicen Beltran, Antonio J. Pea, Josep M. Perez, Xavier Teruel, Jorge Bellon, Daniel Holmes, Pau Farre, and Kevin Sala. Improving the interoperability between mpi and task-based programming models. In *Proceedings of the 25th European MPI Users' Group Meeting (EuroMPI)*, 2018.

[21] Huiwei Lu, Sangmin Seo, and Pavan Balaji. Mpi+ult: Overlapping communication and computation with user-level threads. *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*, pages 444–454, 2015.

[22] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM.

[23] OpenMP Application Program Interface. Version 4.0. July 2013, 2013.

[24] James Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.

[25] Roland Schulz. 3 d fft with 2 d decomposition. 2008.

[26] Jun Shirako, Jisheng M. Zhao, V. Krishna Nandivada, and Vivek N. Sarkar. Chunking parallel loops in the presence of synchronization. In *International Conference on Supercomputing (ICS)*, pages 181–192, 2009.

[27] The Sierra Advanced Technology System. `http://computation.llnl.gov/computers/sierra-advanced-technology-system`, 2018.

[28] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998.

[29] Avinash Sodani, Roger Gramunt, Jesús Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights Landing: Second-generation intel Xeon Phi product. *IEEE Micro*, 36(2):34–46, 2016.

[30] Dylan T. Stark, Richard F. Barrett, Ryan E. Grant, Stephen L. Olivier, Kevin T. Pedretti, and Courtenay T. Vaughan. Early experiences co-scheduling work and communication tasks for hybrid mpi+x applications. In *Workshop on Exascale MPI at Supercomputing Conference*, pages 9–19, Nov 2014.

[31] The Ohio State University. Mvapich2: Mpi over infiniband, 10gige/iwarp and roce. http://mvapich.cse.ohio-state.edu/.

[32] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "Codelet" program execution model for exascale machines. In *International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, pages 64–69, 2011.