MPI detach - Towards automatic asynchronous local completion

Joachim Protze, Marc-André Hermanns, Matthias S Müller, Van Man Nguyen, Julien Jaeger, Emmanuelle Saillard, Patrick Carribault, Denis Barthou

MPI Detach - Towards Automatic Asynchronous Local Completion

Joachim Protze[a], Marc-André Hermanns[a], Matthias S. Müller[a], Van Man Nguyen[b,c,d], Julien Jaeger[b,e], Emmanuelle Saillard[c], Patrick Carribault[b,e], Denis Barthou[c,d,f]

[a] *RWTH Aachen University, ITC, Seffenter Weg 23, Aachen, 52074, Germany*
[b] *CEA, DAM, DIF, Arpajon, F-91297, France*
[c] *Inria, 200 avenue de la vieille tour, Talence, 33400, France*
[d] *University of Bordeaux, LaBRI, Talence, 33400, France*
[e] *Universite Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, Bruyeres-le-chatel, 91680, France*
[f] *Bordeaux Institute of Technology, Talence, 33400, France*

Abstract

When aiming for large scale parallel computing, waiting time due to network latency, synchronization, and load imbalance are the primary opponents of high parallel efficiency. A common approach to hide latency with computation is the use of non-blocking communication. In the presence of a consistent load imbalance, synchronization cost is just the visible symptom of the load imbalance. Tasking approaches as in OpenMP, TBB, OmpSs, or C++20 coroutines promise to expose a higher degree of concurrency, which can be distributed on available execution units and significantly increase load balance. Available MPI non-blocking functionality does not integrate seamlessly into such tasking parallelization. In this work, we present a slim extension of the MPI interface to allow seamless integration of non-blocking communication with available concepts of asynchronous execution in OpenMP and C++. We furthermore investigate compile-time analysis necessary to transform an application using blocking MPI communication into an application integrating OpenMP tasks with our proposed MPI interface extension.

1. Introduction

Next to theory and experimentation, simulation has long established its place as the third pillar of science. This development has lead to an ever-growing increase in demand for computational power in high-performance computing (HPC). To provide this computational power, HPC systems comprise up to thousands of separate compute nodes in a single distributed-memory system. The de-facto standard programming model for distributed-memory systems in a scientific and HPC context is the Message Passing Interface (MPI) [1]. Here, processes exchange explicit messages to coordinate work and distribute data.

While conceptually targeting distributed-memory platforms with its explicit exchange of messages, MPI can be employed on distributed-memory systems, shared-memory systems, and hybrids of the two. This means multiple processes can exchange messages transparently, regardless of whether they execute on the same compute node or not. While the available execution units on a shared-memory domain were limited, applications could use shared-memory as a "fast interconnect" between MPI processes without the need to employ separate (orthogonal) programming models for each type of memory domain. With the rise of large multicore processors that today comprise tens to hundreds of cores in a single shared-memory domain with a decreasing memory-per-core ratio, application developers more and more adopt shared-memory programming into their MPI applications to overcome some of the limitations of classic multi-processing, such as data replication.

While MPI provides an interface to shared-memory programming using remote-memory access, the prevalent programming model for shared-memory programming remains multi-threading using threading interfaces, such as OpenMP, POSIX threads, and C++ threads. MPI is largely thread-agnostic in that it acknowledges the existence of threads and how to ensure a thread-safe use, but not much beyond that. Proposals to introduce Endpoints [2] or Finepoints [3] highlight the increasing interest in providing better support and performance for multi-threaded MPI applications.

OpenMP [4] has a long tradition in HPC. OpenMP work-sharing has long been the preferred easy-to-use parallelization construct in bulk-synchronous applications that iteratively alternate communication phases (using MPI) and computation phases (using OpenMP). This operation mode fits very well in the past, enabling the MPI communication to be mostly separate from the thread parallelization. With the introduction of tasking in OpenMP 3.0 in 2008, task-level programming became more critical for modern application codes, with its reliance on tracking data dependencies and efficient scheduling through a runtime rather than the programmer, trying to minimize waiting time in irregular applications. Using tasks can dissolve the strict

Email addresses: protze@itc.rwth-aachen.de (Joachim Protze), hermanns@itc.rwth-aachen.de (Marc-André Hermanns), mueller@itc.rwth-aachen.de (Matthias S. Müller), van-man.nguyen.ocre@cea.fr (Van Man Nguyen), julien.jaeger@cea.fr (Julien Jaeger), emmanuelle.saillard@inria.fr (Emmanuelle Saillard), patrick.carribault@cea.fr (Patrick Carribault), denis.barthou@inria.fr (Denis Barthou)

separation of communication and computation phases, creating a need for tasks performing MPI communication. The use of OpenMP `taskyield` [5] has proven as not reliable for MPI communication in OpenMP tasks because both OpenMP and MPI provide too weak guarantees. The framework Chameleon [6] already takes the next step and supports the migration of tasks to other MPI ranks in order to improve the load balance further.

MPI provides different threading levels (see Section 2.1), but for irregular tasking applications, this may lead to a requirement of the highest level (`MPI_THREAD_MULTIPLE`) negatively impacting performance. TAMPI, a task-aware MPI implementation [7, 8], marks a first step towards integration of MPI and tasking parallelism. While this work shows the potential of this kind of programming, it assumes the MPI implementation to be explicitly aware of a tasking paradigm.

In our work, we propose a slim extension of the MPI interface, which integrates into existing MPI interfaces to provide functionality very similar to TAMPI. However, our approach keeps MPI and tasking separate while encouraging the interoperability of the two paradigms. We will show that even with the task-agnostic interface, we can write clear code and improve performance. Furthermore, this interface is also applicable to use cases beyond OpenMP/OmpSs tasks, such as C++ promises.

In addition to the MPI extension and the associated runtime extension, we also explore possible compile-time analysis to help using this new functionality in existing MPI programs. In such programs, our compiler extension gathers information about MPI blocking procedures and their surrounding statements. It automatically transforms these blocking procedures into non-blocking ones, using our extension of the MPI interface. This compiler extension can help programmers integrate the proposed interface in their code without extensive knowledge of non-blocking MPI communication or the OpenMP tasking model. We provide a working proof-of-concept of such compiler extension as an LLVM pass.

This paper presents the following contributions: 1. interface extensions to MPI that enable programmers to transfer control over request completion to the MPI library while providing a callback-driven notification of completion back to the application; 2. a prototype implementation of the proposed interface that is independent of the MPI implementation used; and 3. a compile-time analysis to convert blocking communications into their non-blocking counterparts relying on the proposed interface extensions.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction into MPI non-blocking communication requests. Section 3 provides a motivational overview of desired interpretations of MPI non-blocking communication with other interfaces, such as OpenMP and modern C++. Section 4 proposes interface extensions to MPI to hand over requests to the MPI library for asynchronous local completion. Section 5 introduces a proof-of-concept compiler transformation pass that aims at automatically adapting MPI codes to this new interface. The implementation details of both the interface extensions and the compiler pass are described in Section 6. We present measurements on modified codes that compare our implementation to other approaches using Block Cholesky Factorization as a benchmark in Section 7. Finally, Section 8 concludes this paper and provides an outlook on future work.

2. The Message Passing Interface

This section recaps the relevant MPI features regarding thread support, non-blocking procedures, and progress to motivate the proposed interface extensions.

2.1. Thread support

MPI provides several levels of thread support, where requirements by the application and guarantees by the MPI runtime system are communicated during initialization: 1. `MPI_THREAD_SINGLE`, 2. `MPI_THREAD_FUNNELED`, 3. `MPI_THREAD_SERIALIZED`, and 4. `MPI_THREAD_MULTIPLE`.

In `MPI_THREAD_SINGLE` only a single application thread will execute. In `MPI_THREAD_FUNNELED` multiple application threads may execute, but only the thread that called `MPI_Init_thread` will call other MPI functions. In `MPI_THREAD_SERIALIZED` multiple application threads may issue calls to the MPI library, but never concurrently. In `MPI_THREAD_MULTIPLE` multiple application threads may issue calls to the MPI library, with no restrictions on concurrency of those calls.

The application negotiates the actual thread support level by providing the required thread support level to `MPI_Init_thread`. The MPI library returns the actual supported thread support level, which may differ from the *required* value provided by the application. The application is responsible for ensuring that all interactions with the MPI library are supported in the actual thread support level after initialization.

Threads are not separately addressable in MPI communication. Assuming the highest thread support level, this means a message addressed to an MPI process can be received by any thread in this process.

2.2. Requests

MPI provides blocking and non-blocking communication schemes. Blocking communication functions will return when the respective part of a message exchange is completed locally. Non-blocking communication functions may return before the respective part of a message exchange is completed locally. One or more further calls to the MPI library are needed to complete the communication operation locally.

To allow applications to query and ensure the completion of specific communication operations initiated in non-blocking communication, MPI provides opaque handles to so-called *requests*.

MPI provides two flavors of such requests: 1. *standard requests*[1] that are created ad-hoc during message initiation and discarded during completion; and 2. *persistent requests*, where message parameters are defined separately before message initiation and which can be reused. Persistent requests may become

[1] We name these requests *standard* to easily differentiate them from *persistent* requests where needed. The MPI standard itself does not specifically name them other than *requests*.

active when the communication is initiated and inactive when the associated communication operation is complete locally.

Request handles are the interface to ongoing operations that have to be completed using extra completion functions. Both request types—standard and persistent—can be completed using the same set of completion functions. These are either wait or test functions: `MPI_{Wait|Test}{any|some|all}`. Wait functions will block until the corresponding operation is completed locally. Test functions will return a flag indicating the completion status of the corresponding operation. If the flag indicates completion of the corresponding operation, the request is considered in the same state as if an equivalent wait function would have been called instead. It is erroneous for a request to appear in two or more concurrent calls to wait or test functions other than `MPI_Test` [1, p. 485].

2.3. Progress and fairness

When we talk about progress in communication, we mean the process of bringing a message from application memory onto the network or at the receiver side from the network to the application buffer. Depending on available hardware capabilities and utilization, the communication library might actively need to control the transmission of data.

Compared to other HPC communication interfaces, such as OpenSHMEM [9], MPI provides a weak progress guarantee. For point-to-point communication, MPI guarantees that once a matching pair of send and receive is initiated, one of them will (eventually) complete. For collective communication, MPI states that once initiated, a non-blocking collective operation *may* progress independently of any computation or other communication at participating processes, but it is not required to progress. This provides much freedom of implementation for a progress engine, for example, through RDMA hardware support, progress threads, or even progressing pending communication whenever the application calls into the MPI library.

While enabling the implementation on a broad range of platforms, the latter approach may not provide communication-computation overlap possibly expected by the programmer. When a pending communication cannot progress, although matching communication calls are posted, the application may suffer from waiting time [10] when the communication partners do not call into the MPI library for long periods. Hence, some application codes, in the hope of driving pending communications in the absence of a strong progress engine, repeatedly call into the MPI library to ensure regular progress for their application.

Furthermore, while MPI guarantees message order for consecutive messages, it does not provide any fairness guarantee in handling communication from different sources. This means a pending send operation may stay open even though the receiving process posts matching receives that would match this open send if other send operations from different sources match these receives.

This extends to logically concurrent messages of the same process with the same message envelope when multiple threads in an MPI process initiate non-blocking communication concurrently.

Source Code 1: Receive data to be processed in function `work_with`. `other_work` could execute while waiting.

```
#pragma omp task depend(out: data)
  MPI_Recv(data, ...);
#pragma omp task depend(in: data)
  work_with(data);
#pragma omp task
  other_work(other_data);
```

Source Code 2: Introducing non-blocking communication and `taskyield` can help to overlap communication with independent computation.

```
#pragma omp task depend(out: data)
{
  MPI_Request req;
  int received = 0;
  MPI_Irecv(data, ..., &req);
  MPI_Test(&req, received, ...);
  while(!received){
    #pragma omp taskyield
    MPI_Test(&req, received, ...);
  }
}
#pragma omp task depend(in: data)
  work_with(data);
#pragma omp task
  other_work(other_data);
```

3. Interoperability

As a motivation for the proposed interface, we sketch the interoperability of MPI with OpenMP tasks. We start with a simple tasking code, where we have work depending on incoming data and other work that can be processed independently, e.g., on local data. The execution of Source Code 1 will block in `MPI_Recv`, so that no other task can execute while waiting on the data.

Based on OpenMP 4.5, Schuchart et al. [5] suggested to use an OpenMP `taskyield` based approach as sketched in Source Code 2. In the worst case, this approach can result in deadlock due to the limited guarantees OpenMP provides for `taskyield`.

With OpenMP 5.0 [4], the new feature of detached tasks was introduced. Instead of manually delaying the completion of a task – and therefore leaving the dependency open – the task can finish execution, while task completion depends on a call to `omp_fulfill_event`. Only if the task completes, the dependencies are released and depending tasks can start execution.

Source Code 3 shows a possible integration of the new OpenMP feature with MPI non-blocking communication. The new MPI function `MPIX_Foo` takes the request handle and starts monitoring the request for completion of the communication in the background. Once the request is completed, the implementation will call `omp_fulfill_event(ev_handle)`. This will resolve the out dependency to `data` so that the depending task

Source Code 3: Allow the communication task to finish, but only complete when the MPI communication is completed

```
omp_event_handle_t ev_handle;
#pragma omp task detach(ev_handle) depend(out: data)
{
  MPI_Request req;
  MPI_Irecv(data, ..., &req);
  MPIX_Foo(&req, omp_fulfill_event, ev_handle);
}
#pragma omp task depend(in: data)
  work_with(data);
#pragma omp task
  other_work(other_data);
```

can execute `work_with(data)`. After the call to `MPIX_Foo`, the task has no use for the request handle, so the function should consume the handle and set it to `MPI_REQUEST_NULL`. Section 4.1 will make a concrete proposal for this function.

Other than Source Code 2, this code does not regularly call into the MPI runtime library. As a result, we can still run into a deadlock because MPI might not make local progress, and therefore `omp_fulfill_event` is never called. We will address this issue by an additionally proposed interface in Section 4.3.

4. MPI Interface Extensions

The interfaces proposed in this section follow two objectives: 1. the detach interface allows to use non-blocking communication as locally asynchronous communication; 2. the progress interface allows triggering progress in MPI from another parallel runtime.

4.1. Detaching from non-blocking completion

As the fundamental concept of the detach interface, the application registers a callback for notification of local non-blocking request completion. The combined execution of a detach routine, such as `MPIX_Detach`, and the provided callback has the same semantics as a call to `MPI_Wait` or a call to `MPI_Test` returning `flag=true`, completing the request locally. Similar to wait and test calls, a detach call will consider a null or inactive request as immediately completed. In such a case, the callback might be called immediately.

For consistent behavior, it is essential to ensure that the application does not interfere with the completion of any detached MPI requests—i.e., complete a detached request manually. Therefore, a detach call will consume *standard request* handles passed to it and set them to `MPI_REQUEST_NULL` on return, thus taking over full ownership of the requests. This behavior is similar to a call to `MPI_Request_free`, with the difference that upon local completion of the request, the application is informed through the invocation of the callback function.

In its simplest form, a single request is detached and handed to the MPI library along with a function pointer to the callback function that shall be executed on request completion of this specific request.

```
typedef void MPIX_Detach_function(void *data);

int MPIX_Detach(
    MPI_Request *request,
    MPIX_Detach_function *callback,
    void *data);
```

The callback function type `MPIX_Detach_function` takes as the single argument, which is the pointer to `data` provided to the detach call. The invocation of the callback signals the completion of the request. At this point, any non-persistent request is deallocated.

Existing completion routines—the `MPI_{Wait|Test}` family of calls—also provide a possibility to provide a status object, which will contain additional information about the completed message. For existing MPI functions, this means the application provides the storage, and the MPI library writes to the status object unless `MPI_STATUS(ES)_IGNORE` is provided. The status object is caller-owned, and this behavior would be desirable to retain. The use of `MPI_STATUS(ES)_IGNORE` indicates that the programmer is not interested in status information to be returned and can avoid the cost of collecting and filling the status information, so we also want to support this in the detach interface. For the existing completion routines, the programmer has full control over the execution context of when the request is completed. Thus, it is no problem to provide a memory location on the stack or the heap. With a detached request, the programmer explicitly gives up control over the completion context, so status objects allocated on the stack become problematic, as request completion may occur outside of the execution scope where the stack variable was defined.

To retain the ability to allocate a status object either on the heap or on the stack, the detach interface provides an orthogonal set of functions and callbacks to allow for the return of a status object to the application. However, to keep the concept of caller-owned status objects—in this case, the detach runtime is the caller of the completion—the MPI library will provide storage for the status object based on the call used to detach a corresponding request, and pass this to the callback function on invocation. The callback executes any status-dependent code before the runtime frees the status object on return from the callback function. For type safety, we propose a variant of the callback function type, and accordingly, a new detach function to ask for the status:

```
typedef void MPIX_Detach_status_function(
    void *data,
    MPI_Status status);

int MPIX_Detach_status(
    MPI_Request *request,
    MPIX_Detach_status_function *callback,
    void *data);
```

`MPI_Waitsome`, `MPI_Waitany`, and their analogous calls of the test family allow us to wait for arbitrary completion of one or more of multiple requests. We will typically use those functions to continue execution based on which requests are finished. For the detach interface, similar functionality can be achieved

by detaching multiple requests using `MPI_Detach_each`. This function registers a single callback for an array of requests and an associated array of data objects. Similar to the detachment of single requests, the registered callback is invoked for each (hence the name) request completion. Here, the detach interface deviates from the interface for wait and test. The calls to `MPI_Waitsome` and `MPI_Testsome` may return with multiple completion for two reasons:1. to reduce the overhead of calling into the MPI layer multiple times for each pending request (as needed with `MPI_Waitany` and `MPI_Testany`, respectively), and 2. to reduce the chance of starving a single client in a client-server scenario with one client's requests being repeatedly favored by the selection of a single completed request. The former situation of additional overhead does not apply in this situation, as an MPI implementation may implement the invocation of the callback on a desired low level of the runtime, hence already minimizing the overhead created by calling into the top layer interface of MPI. The latter also does not necessarily apply to the detach interface either, as here the runtime can (and should) ensure that no request completion is held back indefinitely—e.g., by employing the round-robin principle—and then immediately invoke the callback. Therefore, an MPI library can already implement a starvation-free method of completing multiple pending requests. In symmetry with the single request detach interface, separate detach procedures are available for completing with and without status information.

```
int MPIX_Detach_each(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_function *callback,
    void *array_of_data[]);

int MPIX_Detach_each_status(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_status_function *callback,
    void *array_of_data[]);
```

In some cases, the application depends on all pending requests to be finished, as expressed by a call to `MPI_Waitall` or `MPI_Testall`. The detach interface proposed here provides a detachment mechanism with similar semantics in that just a single callback is registered and only invoked once when all corresponding requests are completed. Again, in symmetry to the rest of the detach interface, separate detach procedures are available for completing with and without status information. However, as the callback function receiving status information registered will need to process multiple status objects, a separate callback function type ensures type safety.

```
int MPIX_Detach_all(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_function *callback,
    void *data);

typedef void MPIX_Detach_all_statuses_function(
    void *,
    int,
    MPI_Status *);

int MPIX_Detach_all_status(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_all_statuses_function *callback,
    void *data);
```

The integer argument of the callback provides the length of the status array. The value is the same as passed to the corresponding call of `MPIX_Detach_all_status`.

4.2. Persistent communication

Persistent requests need special treatment, as these are also not freed on completion by wait and test calls but rather set to an inactive state. To model the ownership transfer, we introduce in addition to the active and inactive state of persistent requests a new *detached* state. The application retains access to the handle—i.e., it is not reset on detachment—and can refer to it later in the code. We introduce the additional *start-detached* functions listed in Source Code 4 for a detached start of persistent requests. Effectively, the semantic of these functions is a combined call to `MPI_Start{all}`, and the corresponding *detach* function afterward. The functions mark active, persistent requests as *detached*. Using a detached request in any MPI function is erroneous. In our prototype, calling any *detach* function with an active, persistent request will be erroneous because the call might accidentally set the request to `MPI_REQUEST_NULL`.

4.3. Driving Progress

As discussed in Section 2.3, MPI provides only weak progress guarantees and does not provide a specific call to advance pending communication. In case the MPI library uses hardware support or a progress thread to drive asynchronous progress, the application does not need to call into MPI itself to ensure progress. In the style of the Deep Computing Messaging Framework (DCMF) [11]. Therefore, we propose the specific procedure call `MPIX_Progress` which programmers can employ if they want to communicate opportune moments for the MPI library to provide progress.

```
int MPIX_Progress(void *data);
```

This non-blocking progress function can be employed by polling services of tasking runtimes such as OmpSS-2 to drive progress even without particular knowledge about specific pending requests. The only requirement from an application perspective is that the completion callback will finally be called after completion if this function is called repeatedly. If an implementation provides other means of progress than just by calling into the MPI library, this procedure may be implemented as a no-op. However, in case an implementation does provide communication progress when calling into the MPI library, a call to this procedure should do exactly that. While the use of `MPI_Test` or `MPI_Iprobe` will provide some form of progress on common MPI implementations, it is not required by MPI and is rather a matter of quality of implementation. This call, therefore, enables

Source Code 4: Allow the communication task to finish, but only complete when the MPI communication is completed

```
int MPIX_Start_detached(
    MPI_Request *request,
    MPIX_Detach_function *callback,
    void *data);

int MPIX_Start_detached_status(
    MPI_Request *request,
    MPIX_Detach_status_function *callback,
    void *data);

int MPIX_Start_detached_each(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_function *callback,
    void *array_of_data[]);

int MPIX_Start_detached_each_status(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_status_function *callback,
    void *array_of_data[]);

int MPIX_Start_detached_all(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_function *callback,
    void *data);

int MPIX_Start_detached_all_status(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_all_statuses_function *callback,
    void *data);
```

an easy and clear way of enabling MPI to drive the communication, as needed in the code example 2, where the repeated call to `MPI_Test` allows the MPI library to provide progress. This function could also allow us to receive the callback on a thread with low utilization if the polling service has some knowledge about the utilization of threads. Finally, by providing an explicit call to drive communication progress, it will be easier for performance tools to filter such general progress driving calls and potentially optimize its wrapper implementation.

The signature of `MPIX_Progress` follows the callback type `polling_service_t` defined and used in Nanos6. The `*data` argument is an opaque argument, which is passed while registering the polling service and then provided to the callback. Although we do not use the argument in our current implementation, we might use the argument to restrict polling, for example, to make progress on a specific communicator.

5. Towards a Compile-Time Automatic Transformation

This section proposes a proof-of-concept compile-time analysis and transformation to ease the use of our new interface. It aims at automatically transforming blocking and non-blocking calls to non-blocking communications encapsulated inside OpenMP tasks with the proposed detach semantics. For this proof-of-concept, we mainly focus on correctness to provide a working prototype ensuring valid application outputs.

Source Code 5: Basic MPI code using blocking communication.

```
work_with1(data);
MPI_Recv(data, ...);
other_work1(other_data1);
work_with2(data);
other_work2(other_data2);
```

Source Code 6: Transformation of Source Code 5 with the analysis in [13].

```
work_with1(data);
MPI_Irecv(data, ..., req);
other_work1(other_data1);
other_work2(other_data2);
MPI_Wait(req)
work_with2(data);
```

5.1. Automatic Code Motion to Extend the MPI Non-blocking Overlap Window

As non-blocking MPI operations are still very seldom used in applications, previous works propose to automatically transform MPI blocking operations into their non-blocking and persistent counterparts [12, 13]. In the latest, blocking and non-blocking MPI communications are optimized with a static approach based on a data-flow analysis to find all dependencies on a communication buffer. The compilation pass then performs code motion to move the initiation and the completion as far as possible to increase overlapping potential. The analysis finds all statements independent from the communication and puts them between the initiation and the completion calls of the newly inserted MPI procedures. Based on this analysis, all statements preceding the communication and on which the communication depends are packed together before the initiation call. Likewise, all statements depending on the communication (directly or recursively) are packed together after the completion call. The transformation of Source Code 5 with this analysis is presented in Source Code 6. In the example, `work_with2(data)` can be moved after `other_work2(other_data2)` as `data` and `other_data2` are independent.

5.2. Adapting the Automatic Transformation to Detach Semantics

We adapted the existing compiler extension to automatically transform MPI communications to tasked communications using the detach interface.

We target to produce a code similar to Source Code 7. This target minimizes code transformations while maintaining a

Source Code 7: Transformation of Source Code 5 with the new analysis.

```
omp_event_handle_t ev_handle;
work_with1(data);
#pragma omp task detach(ev_handle) \
    depend(out: data)
{
   MPI_Request req;
   MPI_Irecv(data, ..., &req);
   MPIX_Detach(&req, omp_fulfill_event,
       ev_handle);
}
other_work1(other_data1);
other_work2(other_data2);
#pragma omp taskwait depend(inout: data)
work_with2(data);
```

reasonable level of parallelism. In OpenMP, tasks are created dynamically when a `#pragma omp task` is encountered. It means that the task in Source Code 7 cannot run without `work_with1(data)` being done. Once the task is created, it runs independently and concurrently to the parent thread's code. Hence, the communication tasks in Source Code 7 may run concurrently to the functions `other_work1(other_data1)` and `other_work2(other_data2)`. Finally, `work_with2(data)` is inserted after a `#pragma omp taskwait`, which imposes to wait for all pending tasks to be done before executing this instruction. Therefore, with Source Code 7 semantics, all work related to the communication task will execute in order, while independent computation can run concurrently to the communication task.

The algorithm presented in [13] was designed to transform blocking communications into non-blocking communications while maximizing the overlapping window. This work modified the algorithm to insert a detached communication task instead of the plain non-blocking initiation call and insert a taskwait directive instead of the completion call.

Now dealing with OpenMP task semantics, an insertion point for an initiation MPI call is a statement that matches one of the following:

- The statement is the first statement of the current function
- There is a control flow dependency
- The statement is an MPI call
- The statement is detected to implement an OpenMP taskwait directive

Any previously created task might use a taskwait as a synchronization against the MPI communication. Thus, our transformation needs to obey the OpenMP synchronizing semantics.

The same condition applies to the algorithm that searches an insertion point for the completion procedure. We have to stop iterating over the statements if we encounter a `#pragma omp taskwait`. In our new transformation, instead of inserting the completion call, we insert a `#pragma omp taskwait` with a `depend` clause matching the associated communication task. The taskwait is sufficient to synchronize the communication task.

Source Code 8: Example of control flow limitation.

```
for (int i=0; i < n; ++i) {
  MPI_Send(data[i], ...);
  other_work(other_data);
}
```

Source Code 9: Source Code 8 after applying the transformation.

```
for (int i=0; i < n; ++i) {
  #pragma omp task detach(ev_handle) \
      depend(in:data[i])
  {
    MPI_Request req;
    MPI_Send(data[i], ..., &req);
    MPIX_Detach(&req, omp_fulfill_event,
        ev_handle);

  }
  other_work(other_data);
  #pragma omp taskwait depend(inout: data[i])
}
```

5.3. Limitations and Propositions to Achieve Performance

While building this static transformation pass, we emphasized more on preserving the semantic and avoiding deadlocks. Consequently, this automatic transformation is very conservative and suffers from the same limitations as enumerated in [13].

Merging completion calls. Control flow dependencies severely limit our transformation pass as it is based on strict rules on the domination and post-domination of blocks of code to ensure a transformation's validity. As an example, the code shown in Source Code 8 displays a case where our analysis pass cannot perform well since both the initiation and completion calls have to stay inside the loop to preserve the number of communication calls. In this situation, the pass will generate one communication task per iteration, which can be detrimental to performances, as shown in Source Code 9.

Some of those constraints can be relaxed, specifically on the placement of completion calls, or `#pragma omp taskwait` directives. In this specific example, since there are no dependencies on the communication buffers between the iterations, the completion call or the task wait directive can be placed right after the loop without compromising the code's semantic. This transformation is correct as long as the `#pragma omp taskwait` has the same dependency as the communication task it is linked to, thus ensuring that all of the communication tasks started inside the loop are properly completed. Once safely moved out of the loop, it might be possible for the taskwait directive to be moved further down to extend the overlapping interval. Such situations still need to be correctly identified by our static analysis to perform such transformations automatically.

Improving OpenMP lowering analysis. Working at the Intermediate Representation (IR) level, which is the representation upon

Source Code 10: Transformation of Source Code 5 to full tasks and Detach semantics.

```
omp_event_handle_t ev_handle;
#pragma omp task depend(out: data)
{
   work_with1(data);
}
#pragma omp task detach(ev_handle) depend(out: data)
{
   MPI_Request req;
   MPI_Irecv(data, ..., &req);
   MPIX_Detach(&req, omp_fulfill_event,
       ev_handle);
}
#pragma omp task depend(out: other_data1) \
                depend(in: other_data2)
{
   other_work1(other_data1);
   other_work2(other_data2);
}
#pragma omp task depend(in: data)
{
   work_with2(data);
}
```

which the compiler performs analyses and code transformations also cause several limitations. In the IR, OpenMP pragmas are lowered to internal functions and structures, making it very difficult to generate OpenMP codes at this level without a proper understanding of the IR generation mechanisms. As a result, we currently cannot provide an automatic transformation pass to put both communications and computations in tasks that match the code displayed in Source Code 10.

The first task contains all statements necessary for the execution of the MPI communication. The communication task contains the initiation and the detach functions and directly depends on this preceding task. The third task contains all the statements which are entirely independent of the communication. This new task will share no dependencies with the previously described tasks and can be executed while the communication task is performed. The last task has all the statements that directly or indirectly require some of the communication arguments and depends on the communication task.

When strictly comparing the code snippets in both examples, Source Code 7 provides the same parallelism semantics as the code in Source Code 10.

As described in Section 5.2, `work_with1(data)`, the communication task and `work_with2(data)` will run in order, while `other_work1(other_data1)` and `other_work2(other_data2)` can run concurrently to the communication task. Source Code 10 displays the same behavior in a full task semantics. The first, second and lasts tasks will run in order due to their respective dependencies of `data`, while the third task, having no common dependencies with the other tasks, can run concurrently.

Note that if the semantics is the same for the two presented code snippets, generating the computing tasks semantics may provide better parallelism. Indeed, if some other work is realized by the master thread outside of the code snippet in Source Code 7, this work will be serialized with `work_with1(data)` and `work_with2(data)`. Having these functions in their own task would allow more concurrency. However, since generating such code is more difficult, and our primary concern with this proof-of-concept being correctness and not performance, we limited our transformation to target a simpler code, similar to Source Code 7.

In-depth OpenMP dependencies analysis. With better support for OpenMP directives lowering in the IR, we can also improve the static analysis to check all task dependencies. For now, our transformation inserts a `#pragma omp taskwait` for each communication to ensure correctness. However, it is possible that such taskwait directive may not be necessary. If the original code already contains OpenMP tasks and taskwait directives, the existing constructs may use the same dependencies as the inserted communication task. If the dependencies are similar, then the communication tasks will be automatically ordered with the other tasks without inserting an extra taskwait.

Moreover, task dependencies may reference array specific cells through the use of a variable or an iterator. At compile time, this can be error-prone. The same variable may represent several values (cells) at runtime yet be considered the same dependency in the IR due to the same address being used. For the dependency analysis to be thorough and ensure that all dependencies are correctly discovered, all variables used in task dependencies should be coupled with some polyhedral analysis to confirm dependency domains are the same. This analysis is not trivial and is our ultimate goal regarding the automatic transformation. Another complication for this analysis will be to consider the memory access pattern performed by an MPI communication call. For trivial MPI datatypes, this seems feasible, but MPI also allows to define dynamic and irregular memory access patterns by defining derived datatypes at runtime.

6. Implementation

In this section we discuss three aspects of our implementation regarding the detach interface and the automatic transformation: 1. a core library, implementing the functions proposed in the previous section, 2. how we handle persistent requests in this core library, and 3. the implementation of the compiler transformation using a small wrapper library.

6.1. MPI detach interface

We provide a proof of concept implementation[2] for the proposed interface. The code is independent of a specific MPI implementation but needs to be built against the MPI library in use. It can be either compiled into the application or built as a library and linked into the application. The implementation is thread-safe and can be used in two modes to drive progress: 1. with an external polling service, and 2. with an internal progress thread.

[2] `https://github.com/RWTH-HPC/mpi-detach`

As mentioned in Section 4.3, MPI currently provides no explicit interface to drive progress. We therefore proposed and implemented the call to `MPIX_Progress` as a context-free call into the MPI communication layer. This call can then be called by the runtime at regular intervals to ensure progress.

This call can come from an external polling service, as provided by OMPSS-2[3] or Boost.Asio[4]. In this case, the required threading level for MPI depends on the behavior of the polling service. When considering the MPI threading level, we must include all MPIX functions proposed in this paper in the set of MPI functions. `MPI_THREAD_FUNNELED` or even `MPI_THREAD_SERIAL` might be sufficient if the polling service is only active on the master thread and the application also only communicates on the master thread.

This call can also come from a progress thread spawned by our library. If the progress thread is used, the necessary threading level is `MPI_THREAD_MULTIPLE`. If the environmental variable `MPIX_DETACH` is exported and set to `progress`, the library will start a progress thread to poll on `MPIX_Progress` and make progress for the detached requests. This progress thread takes advantage of internal knowledge and suspends itself if no detached request is in the queue. A conditional variable controls the suspension of the polling thread. The detach functions signal to this conditional variable when requests are enqueued. This is done via condition variables that are modified when requests are queued or completed. As long as there are detached requests in the queue, the progress thread will repeatedly call `MPIX_Progress`. However, if the application also uses non-blocking communication outside of the detach runtime, the progress thread might be suspended by the detach runtime while these other requests are still pending and could benefit from a progress thread. In such a scenario, a polling service might be advantageous.

The implementation of the various proposed detach functions directly test for completion of the communication when called. In case the corresponding completion is already returned from the MPI library, the provided callback is called immediately. Otherwise, the provided requests are enqueued. The implementation uses two distinct queues for 1. for single requests and 2. for multiple requests with *wait-all* semantics.

During lazy initialization, i.e., when one of the detach functions is first called, the library decides whether to launch a progress thread. The progress function tests all requests in the queue for completion and calls the corresponding callback in case of completion. If the polling thread is used, `MPI_THREAD_MULTIPLE` is necessary because of the `MPI_Test` calls performed by the polling thread. To avoid the polling thread becoming a possible bottle-neck, the application developer needs to make sure that the callback does not cause excessive execution on this thread but rather schedules work to be executed by the application threads.

In the current implementation, the *detach-each* functions behave like repeated calls to the *detach* functions. Here, the implementation tests each request individually using `MPI_Test`, for both variants with and without a status object.

The *detach-all* functions need to be handled differently, as the call to the callback depends on the completion of all provided requests. In the case that no status is requested, we can easily compress the vector of requests by removing finished requests from the vector of requests. For this purpose, we use `MPI_Testsome`. In the case that the application requests the status, the use of `MPI_Testsome` would mean rather complex book-keeping of indexes and re-mappings. As proof of concept, the implementation uses `MPI_Testall` in this case. This call provides all required statuses when all requests are finished.

6.2. Handling persistent requests

Compared to standard requests, handling persistent requests requires some special treatment. As the detach function would need to react differently to the two types of requests, the implementation needs to be able to distinguish between them. Any non-persistent request is set to `MPI_REQUEST_NULL` before the detach function returns. Persistent requests cannot be reset but are returned as-is to the application. An MPI implementation can infer from inside knowledge whether a request is persistent or not. However, our prototype implementation only acts as a shim layer on top of MPI using the PMPI interface, and it, therefore, does not have access to such low-level information. Unfortunately, MPI does not provide a function to check whether a request is persistent. Also, there is no possibility to attach an attribute to a request, as it is possible for communicators, windows, and datatypes. Such an attribute system would enable our prototype to intercept the init functions for persistent requests and attach an attribute to the request, allowing state-less tracking of persistent requests. In our implementation, we would need a map to store persistent requests at creation time and lookup every request passed to the library in this map. Full integration into the MPI library would provide the potential for optimization in this regard. To avoid this issue, we decided to provide a separate interface, which starts and detaches persistent requests in one function. To avoid the need to track the creation of persistent requests, we disallow the use of the detach functions for active persistent requests.

6.3. Compiler Transformation using a wrapper library

The adapted algorithm to automatically insert *detach* semantics is implemented as a compilation pass in the LLVM compiler [14]: the code is represented as an intermediate representation (IR), which allows us to be completely independent of the source language. LLVM defines many analysis passes whose results can be reused in other optimizations and user-defined passes. These passes provide us the list of loops, the dominator and post-dominator trees for a given function, and the use-def and def-use chains of each value. This information is required to identify which statements are related to the communications calls and which statements are independent of them.

As we explained in Section 5.2, the lowering of OpenMP pragmas in LLVM produces a quite complex set of IR statements. Automatically generate such codes to create new OpenMP `task`,

[3] `http://pm.bsc.es/ompss-2`

[4] `https://www.boost.org/doc/libs/`

Source Code 11: Transformation of Source Code 5 with the new analysis using the wrapper library.

```
omp_event_handle_t ev_handle;
work_with1(data);
MPITask_Recv_Detach(data, ...);
other_work1(other_data1);
other_work2(other_data2);
MPITask_Wait_Detach(data);
work_with2(data);
```

Source Code 12: Example of wrapper functions used in Source Code 11.

```
int MPITask_{Recv/Send}_Detach(data, ...){
 omp_event_handle_t ev_handle;
 #pragma omp task detach(ev_handle) \
    depend({out/in}: *data)
 {
    MPI_Request req;
    MPI_{Irecv/Isend}(data, ..., &req);
    MPIX_Detach(&req, omp_fulfill_event, ev_handle);
 }
}


int MPITask_Wait_Detach(data){
  #pragma omp taskwait depend(inout: *data)
}
```

and `taskwait` pragmas requires much effort and much time to acquire the necessary expertise. To circumvent this difficulty, we chose to use a wrapper library. This wrapper library provides two sorts of wrappers. First, communication initialization wrapper functions, which take the arguments from the original communication call and contain the OpenMP task, including the communication initialization call with the detach interface. Second, a *wait* wrapper function, which contains the taskwait directive to be called to sync with the detach task.

This wrapper library eases the transformation to produce the desired program with the detach interface. Instead of generating a complex set of IR statements, it is now possible to just insert the calls to the corresponding wrapper functions (see Source Code 11 for the generated code and Source Code 12 for the corresponding wrapper functions).

The original algorithm and compilation pass we modified were already designed to insert new function calls in the analyzed program. Thus, the compilation pass has been modified to insert our wrapper functions once the insertion points are found, instead of the MPI initiation and completion procedures.

7. Experiments

We test the implementation of our implementation of asynchronous local completion as well as the automatic code transformation with distributed Block Cholesky Factorization using OpenMP tasks with fine-grained dependencies. The different versions of the code highlight the possible performance improvements as well as improved readability when using MPI detach with an OpenMP tasking code. We perform two sets of performance studies using this code. During the experiments, we also fixed a race condition in the detached task implementation in the LLVM/OpenMP runtime.

7.1. Code

Cholesky Factorization separates a Hermitian, positive-finite matrix into the product of a triangular matrix and its conjugate transpose ($A = LL^*$). The Blocked Cholesky Factorization splits the matrix A of size $n \times n$ into $b \times b$ equally sized blocks. In each iteration k, the algorithm first solves the kth block on the diagonal using `potrf`, the second step updates the $b-k-1$ blocks below using `trsm`, then the remaining triangle of blocks below the diagonal is updates by calls to `gemm` and finally calls to `syrk` update the diagonal. Only the last two steps are independent, while the other steps have data dependencies. Solving the $k+1$st block with `potrf` depends on the result of `syrk` for this block from the kth iteration.

Schuchart et al. [5] provided two MPI-distributed and taskified versions of Block Cholesky Factorization. In this distributed implementation, the blocks are evenly distributed to the MPI processes. Data dependencies involving blocks on different processes result in MPI communication for exchanging these blocks. One version funnels all MPI communication related to a step through a single communication task (*singlecom*); the other version has fine-grained dependencies and performs non-blocking communication in tasks (*taskyield*). The latter version uses OpenMP taskyield and MPI test similar to Source Code 2.

We extend the taskyield code and integrate our MPI detach implementation into the code (*detach*). The necessary changes to the code are conceptually already presented by the transition from Source Code 2 to Source Code 3, where we use `MPI_Detach` instead of `MPI_Foo`. The used codes are available in the mpi-detach branch of our fork[5].

To investigate the challenges for an automatic code transformation, we go one step back from the *singlecom* version and implement the necessary exchange of blocks using blocking MPI communication (*blocking*). This version represents an application with non-trivial control flow and blocking communication. The hand-written MPI-detach code illustrates the goal of the automatic code transformation.

7.2. Environment

We run the experiments on dual-socket nodes of Claix-18, which are equipped with two 24 core Intel Skylake Platinum 8160 CPUs and 192GB memory. The nodes have hyper-threading disabled, and sub-NUMA clustering enabled. The operating system is CentOS 7.9.

For the experiments, we use the Clang compiler from the LLVM 11 release. This version of clang includes fixes for two

[5] `https://github.com/RWTH-HPC/cholesky_omptasks`

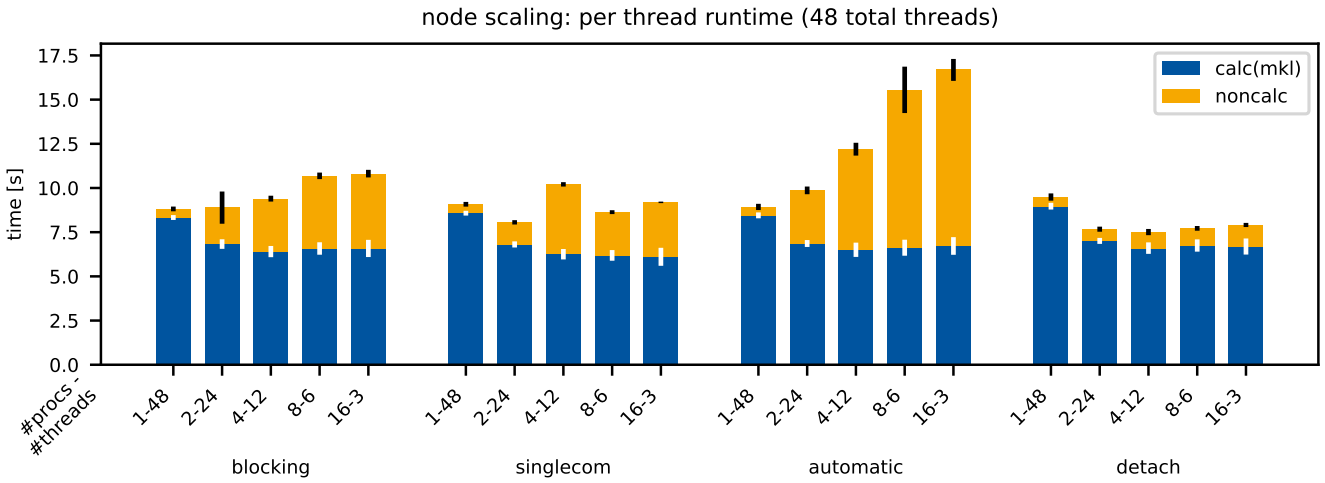

Figure 1: Node scaling measurements of distributed Block Cholesky Factorization

race conditions in the OpenMP runtime[6], that we identified during our initial experiments.

All versions of the code use the Intel MKL in version 19.0.1.144 for the math kernels.

7.3. Run configurations

We perform two scaling measurements. Each measurement point represents ten repetitions, and we plot the average and the standard deviation of the measured time. The nature of the decomposition code only allows powers of two for the number of MPI processes.

The first measurement series fills a single node with 48 threads, which are equally distributed to a varying number of 1 to 16 MPI processes. We run this series with an $n \times n$ matrix of size $n = 32\,768$ and $b \times b$ blocks of size $b = 256$. This results in 16 384 blocks.

The second measurement series uses a fixed number of 12 threads per MPI process and scales from 1 to 32 nodes. We run this series with a matrix of size $n = 65\,536$ and blocks of size $b = 256$. This results in 65 536 blocks. As the complexity of the problem is $O(n^3)$, the expected execution time for the second series is eight times the execution time of the first series.

The reported number of threads is the number of dedicated cores associated with the process by the batch system, e.g., by using `taskset`.

For easier evaluation and reproducibility, we used the Jülich Benchmarking Environment (JUBE) [15] in version 2.2.2 to configure and run the measurements in the jube directory[5].

7.4. Results

Figures 1 and 2 show the collected results. We distinguish the time spent in the compute kernels in MKL (calc(mkl)) from the remaining time (noncalc). We expect the same execution time for the compute kernels independent of the build configuration, as we always use the same MKL routines. There is no redundant computation caused by the distribution. This expectation is held within the scope of measurement errors. The plotted bars show time averaged over the executing threads and 10 repetitions. The white error bar shows the standard deviation in the calculation time across all threads; this is a result of an imbalance in the distribution between processes and an imbalance in the scheduling on the available threads. The black error bar shows the standard deviation of the overall execution time.

We do not measure the time of the polling thread explicitly, but the polling thread will always share a core with a calculating thread due to the process placement. So, the polling thread is included in the execution time of the measured threads.

For the node scaling experiment in Figure 1, we can see the influence of memory accesses across NUMA domains when executing with just one MPI process. With two MPI processes, we still fill one socket with threads and can see a slight impact of sub-NUMA clustering. For most configurations, the runtime overhead increases when splitting to more than 4 thread. For both reasons, we chose the configuration with four MPI processes per node for the strong scaling experiment. We can see that the overhead of the *detach* code has the lowest overhead across all configurations.

For the strong scaling experiment in Figure 2, we show two representations of the data. In Figure 2a, we plot the average execution time of each single thread or, respectively, the wallclock time of the code, where we expect an inversely proportional decrease of the calculation time (calc). In Figure 2b, we plot the aggregated overall execution time of all threads, where we expect constant calculation time independent of the number of processes or threads. When zooming in, we saw a slight increase in the calculation time for 128 processes.

For a low number of processes, the single communication task in *singlecom* introduces more overhead, than the blocking communication. With more than eight processes the tasking overhead is amortized and the *singlecom* code shows less runtime overhead than the *blocking* code.

In the first plot, we observe a constant and low per thread overhead for the *detach* code among 4 to 32 processes, and a slight increase for higher process counts. In the second plot, we can

[6]LLVM patches D79702 and D80480

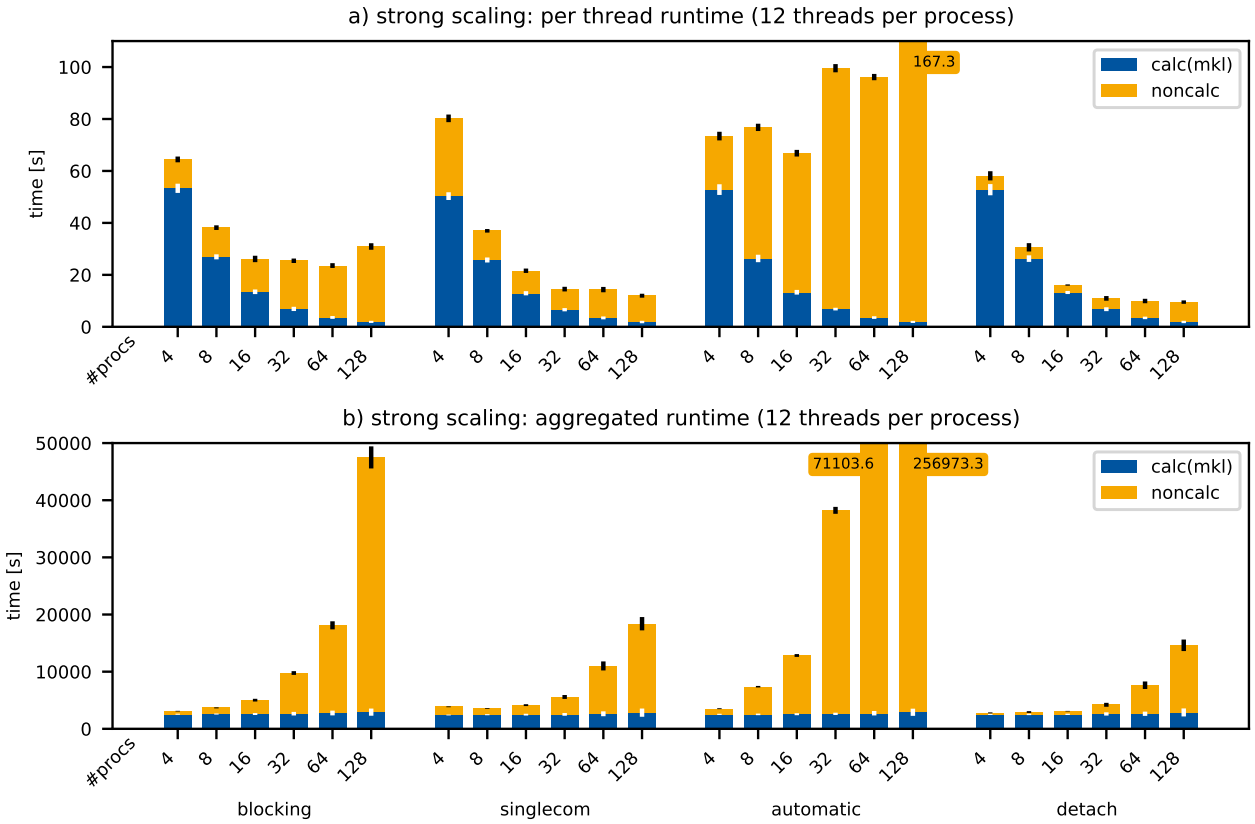

Figure 2: Strong scaling measurements of distributed Block Cholesky Factorization

observe how this constant per thread overhead rapidly amounts to significant overhead for the overall execution. Nevertheless, the *detach* code outperforms the other codes.

The numbers for the *automatic* code transformation are included to demonstrate, that the code runs and terminates. The results also successfully verify against a serial version of the algorithm.

7.5. Discussion

Overall the scaling experiment shows that an implementation of the algorithm based on our proposed extension can outperform any portable implementation based on currently available interfaces. We think, that an integration of the interface into an MPI implementation and an integration of a polling service in the OpenMP implementation can further improve the possible performance. This is subject to further investigation.

Performance with the automatic transformation. As explained in section 5.3, the static transformation pass was first and foremost built around the validity of transformations, and struggles to offer satisfying performances for the Cholesky benchmark. Specifically, we identified two improvements that will help reach the same level of performance than the hand-tuned version. First, all communication calls in the benchmark are in loops. This means our automatic transformation generates a taskwait for individual communication in the loop, instead a unique taskwait for the whole loop. All these synchronizations prevent the communication tasks to be efficiently overlapped.

Moreover, the original tasked version without the MPI detach interface presents computational parts already put into tasks. The dependencies of these tasks are similar to the ones generated for the communication tasks. With an in-depth dependencies analysis, the computational tasks could be matched to the generated communication tasks, hence avoiding a general synchronization point through a taskwait, and we could have better concurrency. The ultimate goal will be to reach the same dependency graph as shown in the *detach* code.

8. Conclusions

Initially driven from a C++ application use case, we proposed a new interface for MPI in this paper to integrate MPI communication into an asynchronous control flow of an MPI application. Our proposed interface provides a clear separation of distributed memory communication provided by MPI and the various asynchronous execution concepts such as OpenMP tasks or C++ promises. We presented a proof of concept implementation, which works as a PMPI wrapper library. By integrating the interface into a Block Cholesky Factorization code, we demonstrated the applicability of the approach. Other than a taskyield based approach of non-blocking communication in OpenMP tasking programs, non-blocking communication in combination with the proposed detach functions is completely conforming with the OpenMP specification. Integration into an MPI implementation has potential for further optimization and therefore reduced overhead. We work together with the MPI forum to

introduce an interface for asynchronous MPI communication into a future MPI standard.

Furthermore, we presented a proof of concept implementation for automatic code transformation to convert an MPI application using blocking communication into an application encapsulating MPI communication into OpenMP tasks. While the transformation aims to improve the computation communication overlap, this prototype focuses on performing sound transformations. Therefore the very conservative transformations do not yet provide a performance gain. The prototype nevertheless helped identify the kind of analysis necessary to finally reach performance gains like in a hand-tuned application using the proposed detach functions for asynchronous communication. Investigating these necessary additional analyses and ensuring their soundness is left for future work.

Acknowledgement

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 824080.

References

[1] M. P. I. Forum, MPI: A Message-passing Interface Standard, Version 3.1, 2015.

[2] J. D., R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, R. Thakur, Enabling communication concurrency through flexible MPI endpoints, Int. J. High Perform. Comput. Appl. 28 (4) (2014) 390–405. doi:10.1177/1094342014548772.
URL https://doi.org/10.1177/1094342014548772

[3] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, A. Skjellum, Finepoints: Partitioned multithreaded MPI communication, in: High Performance Computing - 34th Intl. Conf., ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proc., 2019, pp. 330–350. doi:10.1007/978-3-030-20656-7_17.
URL https://doi.org/10.1007/978-3-030-20656-7_17

[4] OpenMP Architecture Review Board, OpenMP application program interface version 5.0, Specification (November 2018).

[5] J. Schuchart, K. Tsugane, J. Gracia, M. Sato, The impact of taskyield on the design of tasks communicating through MPI, in: Evolving OpenMP for Evolving Architectures - Proc. of the 14th Intl. Workshop on OpenMP, IWOMP 2018, 2018, pp. 3–17. doi:10.1007/978-3-319-98521-3_1.
URL https://doi.org/10.1007/978-3-319-98521-3_1

[6] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, M. S. Müller, CHAMELEON: reactive load balancing for hybrid MPI+OpenMP task-parallel applications, J. Parallel Distributed Comput. 138 (2020) 55–64. doi:10.1016/j.jpdc.2019.12.005.
URL https://doi.org/10.1016/j.jpdc.2019.12.005

[7] K. Sala, J. Bellón, P. Farré, X. Teruel, J. M. Pérez, A. J. Peña, D. J. Holmes, V. Beltran, J. Labarta, Improving the interoperability between MPI and task-based programming models, in: Proc. of the 25th European MPI Users' Group Meeting, 2018, 2018, pp. 6:1–6:11. doi:10.1145/3236367.3236382.
URL https://doi.org/10.1145/3236367.3236382

[8] K. Sala, X. Teruel, J. M. Pérez, A. J. Peña, V. Beltran, J. Labarta, Integrating blocking and non-blocking MPI primitives with task-based programming models, CoRR abs/1901.03271 (2019). arXiv:1901.03271.
URL http://arxiv.org/abs/1901.03271

[9] M. B. Baker, et al., OpenSHMEM specification 1.4 (12 2017). doi:10.2172/1460190.

[10] M.-A. Hermanns, M. Geimer, B. Mohr, F. Wolf, Trace-Based Detection of Lock Contention in MPI One-Sided Communication, in: Tools for High Performance Computing 2016, Springer Intl. Publishing, Cham, 2017, pp. 97–114. doi:10.1007/978-3-319-56702-0_6.
URL http://link.springer.com/10.1007/978-3-319-56702-0{_}6

[11] S. Kumar, et al., The deep computing messaging framework: generalized scalable message passing on the blue gene/p supercomputer, in: Proc. of the 22nd Annual Intl. Conf. on Supercomputing, ICS 2008, 2008, 2008, pp. 94–103. doi:10.1145/1375527.1375544.
URL https://doi.org/10.1145/1375527.1375544

[12] H. Ahmed, A. Skjellum, P. Bangalore, P. Pirkelbauer, Transforming Blocking MPI Collectives to Non-Blocking and Persistent Operations, in: Proceedings of the 24th European MPI Users' Group Meeting, 2017, pp. 1–11.

[13] N. V.M., et al., Automatic code motion to extend mpi nonblocking overlap window, in: Jagode H., Anzt H., Juckeland G., Ltaief H. (eds) High Performance Computing. ISC High Performance. Lecture Notes in Computer Science, vol 12321. Springer, Cham, 2020.

[14] C. Lattner, V. Adve, LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., IEEE, 2004, pp. 75–86.

[15] S. Lührs, D. Rohe, A. Schnurpfeil, K. Thust, W. Frings, Flexible and Generic Workflow Management, in: Parallel Computing: On the Road to Exascale, Vol. 27 of Advances in parallel computing, Intl. Conf. on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep 2015, IOS Press, Amsterdam, 2016, pp. 431 – 438. doi:10.3233/978-1-61499-621-7-431.
URL https://www.fz-juelich.de/jsc/jube/