Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests

Alexandre Denis

Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests

Alexandre DENIS
Inria Bordeaux – Sud-Ouest, France
E-mail: Alexandre.Denis@inria.fr

***Abstract*—New kinds of applications with lots of threads or irregular communication patterns which rely a lot on point-to-point MPI communications have emerged. It stresses the MPI library with potentially a lot of simultaneous MPI requests for sending and receiving at the same time. To deal with large numbers of simultaneous requests, the bottleneck lies in two main mechanisms: the tag-matching (the algorithm that matches an incoming packet with a posted receive request), and the progression engine.**

In this paper, we propose algorithms and implementations that overcome these issues so as to scale up to thousands of requests if needed. In particular our algorithms are able to perform constant-time tag-matching even with any-source and any-tag support. We have implemented these mechanisms in our NewMadeleine communication library. Through micro-benchmarks and computation kernel benchmarks, we demonstrate that our MPI library exhibits better performance than state-of-the-art MPI implementations in cases with many simultaneous requests.

I. INTRODUCTION

New kinds of HPC applications have emerged, that rely on a communication scheme different from regular MPI applications. For examples multi-threaded stencil applications push MPI implementations beyond their limits [1]. Applications using a task-based runtime system [2], [3] are another class of applications that stresses MPI communications in an unusual way: the MPI communication schemes when driven by a task-based runtime and when written explicitly in the application are radically different. Most notably, for heavy communication phases, regular MPI applications typically use collective operations, that let the MPI library decide on the communication graph and scheduling. By contrast, a task-based runtime translates data dependencies as a set of point-to-point communications. Since MPI collective operations must be executed by all involved nodes, it does not fit the model of task-based runtimes with unsynchronized nodes.

However, most existing MPI libraries are not designed with such scheme in mind. They are not able to undergo a large number of unrelated point-to-point communication requests at the same time. Scalability issues arise at multiple levels: contention for driver access, linear tag-matching algorithms, and contention on progression mechanisms.

In this paper, we propose algorithms and implementations that overcome this issues so as to scale up to as many simultaneous requests as wanted in the communication library. We propose algorithms that are able to perform tag-matching in constant time in any case, and we propose mechanisms so that the progression engine can sustain thousands of simultaneous requests without contention.

In short, this paper makes the following contributions:

- we propose an algorithm for tag-matching in constant time even in the presence of `MPI_ANY_SOURCE` and `MPI_ANY_TAG` requests.
- we propose original mechanisms to reduce contention between requests submission and progression, involving lock-free submission.
- we implemented these solutions in our *NewMadeleine* [4] communication library.
- we performed a performance comparison of our solution with a large panel of state-of-the art MPI libraries.

The rest of this paper is organized as follows. Section II presents related works about scalability for large number of point-to-point requests. Section III draws a state of play of bottlenecks on our path to scalability. Section IV introduces an algorithm for matching in constant time. Section V presents solutions so that bursts of requests submissions do not hinder communication progression. In Section VI, we evaluate our solution and compare it against other MPI libraries, and Section VII concludes.

II. RELATED WORKS

Scalability of MPI libraries for large number of point-to-point requests has already been talked about in some related works about performance of tag matching and message queues, and performance of progression.

Some work has been published [5][6] about analysis of message queues in some applications. Solutions [7][8] based on context or node ranks have been proposed, but do not address scalability when large number of tags are used.

A bin-based matching algorithm [9] has been proposed for MPICH. The algorithm is not constant-time as soon as there are wildcard requests (any source or any tag), and the bin based hashing structures are quite different from ours. We include MPICH and Intel MPI in our benchmark in Section VI and observe that our algorithm behaves differently.

An adaptive algorithm for tag-matching has been proposed [10] for MVAPICH2, its bin-based variant is using hash-tables which is related to our proposed solution. However,

authors say little details about the algorithm itself, do not evaluate their algorithm with regard to number of requests, and do not compare with other MPI libraries, thus it is hard to tell how close it is. Our own benchmark (see Figure 3 in Section VI) leads us to believe that our solution and theirs is quite different.

Mechanisms for fine-grain locking have been proposed [11][7], which in some case should improve scalability with the number of requests, but our solution pushes the concept further.

Finally, the decoupling between the upper and lower layers of the communication library results directly from the design of *NewMadeleine*, which was described in previous work [4].

III. Study of Bottlenecks

In this section, we study the bottlenecks that arise when a user submits a large number of point-to-point requests to an MPI library. We have identified three possible bottlenecks: network, tag-matching, and progression.

A. Context of Study

Our work has been done in our *NewMadeleine* communication library [4]. *NewMadeleine* exhibits its own native interface and in addition comes with a thin MPI layer called MadMPI. The work described in this paper is located in the lower layers of *NewMadeleine* and thus benefits to both MadMPI and users utilizing *NewMadeleine* native interface.

The originality of *NewMadeleine* compared to other communication libraries and MPI implementations is that it decouples the network activity from the calls to the API by the user. In the interface presented to the end-user, primitives send and receive *messages*. *NewMadeleine* applies an optimizing strategy so as to form *packets* ready to be sent to the network. A *packet* may contain multiple *messages* (aggregation), a *message* may be split across multiple *packets* (multi-rail), and *messages* may be actually sent on the wire out-of-order if the packet scheduler decides so.

NewMadeleine core activity is triggered by the network. When the network is busy, *messages* to be sent are simply enqueued; when the network becomes ready, an optimization strategy is called to form a new *packet* from the pending *messages*. For receiving, large messages use a rendez-vous protocol. For small messages, a receive is always posted to the driver, and all the activity is made of up-calls (event notifiers) triggered from the lowest layer when the receive is completed.

B. Driver and network

At the lowest level, when sending lots of messages simultaneously, the networking hardware or its driver (kernel or user-space library, such as `libibverbs`) may cause a bottleneck. The reasons may be various, such as competition to access the physical medium, interleaved memory transfers that defeat locality, non-scalable driver, driver with a big lock, etc.

This issue may be encountered by regular MPI implementations that directly map user-level messages to network-level packets. In this case, a burst of MPI requests directly triggers a burst of network submissions; conversely, lots of pending MPI receive requests directly maps to lots of pending receive requests in the driver, which may be able – or not – to manage lots of pending requests.

This is not the case for *NewMadeleine* since, by design, the requests submitted to the network are decided by *NewMadeleine* core. In particular, there will always be at most one send request per peer active at any time. On the receiving side, *NewMadeleine* always keeps one receive request posted globally (for drivers/networks that support any source), whatever the user posted for receive, for small packets. For large packets, there will always be at most one active receive request per peer.

As a summary, by design *NewMadeleine* is less prone to overloading the network/driver than MPI libraries that directly map user-level requests to network-level requests.

C. Tag-matching

On the path of incoming messages, tag-matching is the next bottleneck. When a packet is received from the network hardware, it goes through a step called *tag-matching*: the MPI library looks up for a pending receive request previously posted by the user with a combination of specified communicator, source and tag that *matches* the one of the received packet. If there is no matching, the packet is stored in a list of *unexpected* packets. When a user posts a receive requests, the MPI library first check in the list of unexpected packets whether it already arrived; this is another aspect of *matching*.

A naive implementation would use a list for both steps: to store pending receive requests, and to store unexpected packets. However, a list-based implementation leads to sequential search in lists, with a complexity linear with the number of requests or the number of unexpected packets. It causes a scalability issue [1] with the number of posted receive requests and scalability with the number of unexpected packets.

D. Progression and multi-threading

In order to amortize the cost of communications, applications try to overlap communication with computation by using non-blocking communication primitives. This assumes that communication actually progresses in background. To ensure background progression, MPI libraries usually rely on a progression thread or equivalent. It requires some locking to protect the internal state of the library from concurrent accesses from the progression thread and from application threads at the same time.

A large number of pending requests causes the progression thread to take more time for polling, thus with a naive approach it holds the lock for a longer time. Conversely, an application that submits bursts of requests will acquire the lock many times. When combined, these phenomena are very likely to cause contention on the lock.

It means that bursts of requests submission will prevent the progression thread to acquire the lock, so progression will be slower. And longer progression caused by longer queues will delay application threads waiting for the lock.

IV. Constant-Time Request Matching Algorithm

In this section, we present our constant-time request matching algorithm that scales to large number of point-to-point receive requests of any kind.

In common MPI libraries, *matching* is about pairing packets with requests based on source, tag, and communicator. However, *NewMadeleine* in its native programming interface uses the concept of *sessions*, which are multiplexing channels that can be used by multiple libraries using *NewMadeleine*. The session ID is included in the 96 bits of *NewMadeleine* internal tags. In MadMPI, the MPI interface atop *NewMadeleine*, MPI communicators are mapped to sessions, thus matching both MPI tags and MPI communicators is done at once when matching *NewMadeleine* internal tags. As a consequence, in the following we will talk about matching tags and source, but not communicators except where it makes sense to distinguish communicators from tags in the case of *partially-specified* requests in Section IV-A3.

A. Data Structures for Matching

Since matching is essentially a lookup operation, the key for a good scalability is the data structure used to store the manipulated elements, namely communication requests (`MPI_Recv`, `MPI_Irecv`, ...), and unexpected packets arriving before their matching receive request has been posted.

We distinguish three categories of receive requests, depending on their properties for lookup:

- *fully-specified* requests, which are requests with a specified receive tag and source, *i.e.* both the sender and the tag are known in advance;
- *wildcard* requests, which are requests with wildcard for both source and tag at the same time, *i.e.* the user gave `MPI_ANY_SOURCE` as the source in the receive request, and `MPI_ANY_TAG` as tag. In this case, neither source nor tag are known in advance and will be determined only at runtime depending on the next incoming packets.
- *partially-specified* requests, which are requests with one wildcard for source or tag, but not both at the same time. In this case, either the source or the tag is not known in advance and will be determined at runtime depending on the incoming packet that matches the specified part of the request.

1) Matching fully-specified requests: To match incoming packets with *fully-specified* receive requests already posted by the user, we need a data structure that will allow a fast lookup of the request by tag and by source. This case is easy since all the matching information is known in advance, so we can use a container indexed by the matching information, implemented as a BST (binary search tree), as a hash-table, or any structure used to implement associative arrays with fast lookup.

In *NewMadeleine*, we already have a structure attached to each pair *(source, tag)* to store sequence numbers used to reorder packets that may arrive out-of-order because of multi-rail [12] or by the packet scheduler. This structure is stored using a *hash-table*, which has in the general case a constant-time lookup when doing an amortized performance analysis [13]. Actually, the variable-time cost that needs to be amortized is the re-hashing operation that happens on some occasions when storing a new entry in the hash-table. Since we insert an entry in the hash-table when creating a *(source, tag)* pair, *i.e.* the first time we use this precise combination, and always reuse it once it has been created, we are in the favorable case where an entry is created once and looked-up many times thereafter, thus the hypothesis for the amortized analysis are true.

Since the lookup in the hash-table is in the communication critical path, we want to avoid collisions at all costs to avoid pathological behavior. Among the different ways of implementing hash-tables, we choose linked lists, a.k.a *separate chaining* (actually, separate chaining with list head cells, to reduce cache miss) which is known [14] to reduce collisions compared to open addressing. The drawback of linked lists is that it uses dynamic memory allocation in case of collision, which causes a performance penalty. However, it happens only upon entry insertion, which we only do once the first time a given *(source, tag)* pair is used. To even more avoid collisions, our hash-tables use Jenkins *one-at-a-time* [15] hashing and a prime number of cells, with a pre-computed list of prime numbers.

To manage the case of multiple pending receive requests on the same source and same tag, we actually attach a *list* of requests to the entry in the hash-table. Since the MPI specification [16] states that the matching operation must respect packet ordering, we store pending requests in the order they were submitted. The next matching request for the given source and tag is then always at list head (accessed in constant time), and new requests are stored at list tail (stored in constant-time).

Matching a newly posted *fully-specified* receive request with stored unexpected packets is symmetrical with the case of matching incoming packets with fully-specified receive requests.

We need an associative array to attach a list of unexpected packets to their *(source, tag)* matching information. It is done exactly the same with a list of unexpected packets, in the same hash-table entry as above. The list is maintained ordered to comply with MPI semantics, with arriving packets inserted at list tail (constant-time) and dequeued from head (constant-time) when a matching receive is posted.

2) Matching wildcard requests: To match *wildcard requests* in both places — match posted requests with unexpected packets, and match incoming packets with pending requests — it is not possible to use hash-tables or any associative array structures, since the precise *(source, tag)* pair is not known in advance and consequently cannot be used as hashing key.

We treat them as a special case. We store *wildcard* requests in a dedicated list, enqueued in chronological order, one list per communicator. This way, if an incoming packet matches a *wildcard* request, it is always the request at the head of the list, extracted in constant time, without needing any linear search.

Symmetrically, unexpected packets are enqueued in a special list, in chronological order, one list per communicator. We notice that, at this point, unexpected packets are enqueued in two separate lists: the list in the hash-table, for fully-specified requests; and the list, for wildcard requests. Since we use linked-lists, insertion and deletion is done in constant-time. In particular, when a fully-specified request matches an unexpected packet, we do not have to iterate over the wildcard list to remove it from there, since we have a direct pointer to the cell in the list. We will see later that the packet will actually be stored in 4 different lists, using the same constant-time mechanisms.

A tricky aspect of matching wildcard requests resides in how we interleave wildcard requests and fully-specified requests, *i.e.* what happens if an incoming packet matches both a pending fully-specified request and a pending wildcard request. The MPI specifications states that requests should be processed in order, so among all the requests that match the packet, it should be delivered to the oldest request. However, since wildcard requests and fully-specified requests are not stored in the same list nor the same data structure at all, we cannot rely on the order of the data structure.

To comply with this ordering semantics, we introduce a *request sequence number*. A sequence number is given in chronological order to the receive requests that have globally been posted by the process. We use a 64 bits counter which is large enough so that we do not have to worry about counter overflow. When multiple requests are candidate for matching with a given packet, their sequence number is compared so as to determine which one was posted first and should get data. The incrementation of the global counter to get the next sequence number is no point of contention thanks to mechanisms described in Section V-B.

3) Matching partially-specified requests: The matching of *partially-specified* requests, *i.e.* requests on `MPI_ANY_SOURCE` but not any tag, or requests on `MPI_ANY_TAG` but not any source, is more tricky to get at the same time constant-time matching and maintain the right semantics for packet ordering.

We propose to add hash-tables by source and hash-tables by tag, in addition to the hash-table by *(source, tag)* pairs. Partially-specified requests are enqueued in an ordered list stored as an entry in the appropriate hash-table — hashed by tag if only tag is given, hashed by source if only source is given. Since "any communicator" does not make any sense in MPI, and since *NewMadeleine* embed the communicator ID as part of the internal tags, as explained in Section IV, the hash key for table by source is actually *(source, communicator)*.

Symmetrically, unexpected packets are stored in hash-table by source and the hash-table by tag, in addition to the hash-table by *(source, tag)* pair and the special list for *wildcard* requests (any source and any tag at the same time).

Upon packet arrival, four places need to be checked for a receive request: the list for fully-specified requests (hashed by pairs), both lists for partially-specified requests (hashed by source and hashed by tag), and finally the list for wildcard requests. In each list, only the head needs to be checked. If multiple requests are candidate, then the request with the lowest sequence number is selected. This whole operation is performed in constant time. Symmetrically, when a receive request is posted, a single place needs to be checked for unexpected packets.

4) Data structures summary: As a summary, the data structures for our solution are as follows:

- a hash-table with entries hashed by *(source, tag)* pairs. Each entry contains an ordered list of fully-specified pending receive requests and an ordered list of unexpected packets from the given source and tag.
- an ordered list of wildcard requests, and an ordered list of unexpected packets, per communicator. Cells of list of unexpected packets are shared between the global list and the list in the hash-table.
- a hash-table with entries hashed by *(source, communicator)*. Each entry contains an orders list of partially-specified receive requests where only source is given, and a list of unexpected packets from the given source.
- a hash-table with entries hashed by *tag*. Each entry contains an ordered list of partially-specified receive requests where only tag is given, and a list of unexpected packets on the given tag.
- a global counter to assign a sequence number to receive requests, used as a time-stamp to check requests ordering.

In this solution, unexpected packets are enqueued in four different lists but the cell is shared between lists. Receive requests are enqueued in a single list, which one depends on the kind of the request.

B. Matching Algorithm

1) Algorithm implementation: Building upon the presented data structures, we use the following algorithms for packet matching. The hash-table for fully specified requests is $specs$; the hash-tables for partially-specified requests are $tags$ (hashed by tags, for any source) and src (hashed by source, for any tag). The data structure to hold wildcard requests is $wildcards$. Each entry (in hash-tables and $wildcards$ itself) is made of $*.reqs$ the list to contain pending requests and $*.unexpected$ the list to contain unexpected packets.

When a packet is received from the network, it goes through the Algorithm 1 to find whether a matching receive request has been posted previously.

When the user posts a receive request, it goes through the Algorithm 2 to find whether a matching unexpected packet has arrived previously.

The version of the algorithms presented here is a little bit simplified for the sake of brevity. In particular, we assume here that the *lookup* operation in the hash-table will always return an entry, while in reality entries are lazily created – if lookup fails, an entry containing an empty list is created and inserted into the hash-table. Moreover, the removal of req from the list in Algorithm 1 and the removal of unexpected packets from the lists in Algorithm 2 are elided for readability, but do not present any kind of difficulty. Finally, the $wildcards$ structure

Algorithm 1 Matching algorithm upon packet arrival

```
packet arrival from source on tag tag
spec_entry ← lookup(specs.reqs, (source, tag))
spec_req ← head(spec_entry.reqs)
req ← spec_req
w_req ← head(wildcards.reqs)
if req = ∅ ∨ (w_req ≠ ∅ ∧ w_req.seq < req.seq) then
   req ← wildcard_req
end if
tags_entry ← lookup(tags.reqs, tag)
tags_req ← head(tags_entry.reqs)
if req = ∅ ∨ (tags_req ≠ ∅ ∧ tags_req.seq < req.seq) then
   req ← tags_req
end if
src_entry ← lookup(src.reqs, (source, comm))
src_req ← head(src_entry.reqs)
if req = ∅ ∨ (src_req ≠ ∅ ∧ src_req.seq < req.seq) then
   req ← src_req
end if
{No matching request; enqueue packet as unexpected}
if req = ∅ then
   pushback(packet, spec_entry.unexpected)
   pushback(packet, wildcards.unexpected)
   pushback(packet, tags_entry.unexpected)
   pushback(packet, src_entry.unexpected)
end if
return req
```

Algorithm 2 Matching algorithm upon receive request posted

```
receive request posted for source and tag
if source ≠ ∅ ∧ tag ≠ ∅ then
   spec_entry ← lookup(spec.unexpected, (source, tag))
   if ¬ empty(spec_entry.unexpected then
      return head(spec_entry.unexpected)
   else
      pushback(request, spec_entry.reqs)
   end if
else if source = ∅ ∧ tag = ∅ then
   if ¬ empty(wildcards.unexpected) then
      return head(wildcards.unexpected)
   else
      pushback(request, wildcard_reqs)
   end if
else if source = ∅ then
   src_entry ← lookup(src.unexpected, (source, comm))
   if ¬ empty(src_entry.unexpected then
      return head(src_entry.unexpected)
   else
      pushback(request, src_entry.reqs)
   end if
else if tag = ∅ then
   tags_entry ← lookup(tags.unexpected, tag)
   if ¬ empty(tags_entry.unexpected then
      return head(tags_entry.unexpected)
   else
      pushback(request, tags_entry.reqs)
   end if
end if
```

is presented as if it were global, while in reality it is stored in the communicator.

2) Complexity: Algorithms contain no loop, only basic operations and control structures using only `if`, which is obviously bounded in time. Operations on lists are: find the head, enqueue an element at list tail, and remove an element from list knowing a pointer on it; these operations are all realized in constant time given that we use doubly linked lists.

Operations on hash-tables are insertion and lookup. Lookup is a constant-time operation when there are no collisions; it should be the case given that we use a good-enough hash function and a prime number as hash-table size. This assertion has been checked in practice through code instrumentation. Insertion is done in bounded-time thanks to the use of *incremental rehashing* [17]: when a rehashing is needed, only a few entries are transfered from the old table to the new table at a time (16 in our implementation), and rehashing is continued in the next call to hashtable primitive until all entries have been transfered. Compared to regular full rehashing that has bounded time in average only through *amortized analysis* [13] but may from time to time have linear complexity when rehashing is needed, with incremental rehashing every single insertion is done in *bounded time* which greatly reduces the jitter.

As a summary, all operations are constant-time or bounded in time, so both algorithms have a time complexity of $O(1)$. The constant "hidden" in the big-O notation is essentially three look-ups in hash-tables, which is not so heavy.

3) Memory consumption: Regarding memory consumption, let R be the number of posted requests and U the number of unexpected packets. All data structures are hash-tables and linked lists. A cell is allocated for every request (actually, the request object itself acts as a linked-list cell) and for every unexpected packet, leading to a space complexity of the cells of $O(R+U)$. The index of hash-tables is linear in size with the number of contained elements: since our hash-tables trigger rehashing when load reaches over $75\,\%$, and since the size roughly doubles at each rehash (size follows a series of prime numbers roughly doubling at each step), the load stays between $37.5\,\%$ and $75\,\%$, making the size of the index linear with number of entries. If we assume requests and packets may be of different sources and different tags, in the worst case the number of elements in hash-tables is linear with $R+U$. However, as an optimization, we only grow the hash-table index and never shrink it, therefore the space complexity is $O(max(R)+max(U))$. The constant "hidden" in the big-O is essentially four pointers in the cells used to store unexpected packets.

V. Scalable Multi-Threaded Progression Engine

In this section, we present mechanisms to mitigate contention in the communication progression engine, so as to reach good scalability with a large number of point-to-point requests. As exposed in Section III, bottlenecks come from contention to acquire locks, and burst of requests submission that prevent progression threads to acquire locks. Therefore we propose mechanisms to reduce the number of lock acquire/release cycles on the critical path and to hold it for a shorter time.

A. Lock-free request submission

It seems natural to push data to the network as soon as a send request is posted, and to call algorithm 2 as soon as a receive request is posted by the user to get the fastest possible completion. However, *NewMadeleine* being a multi-threaded

library [18], locking is needed around these methods which access global state. Locking done from application threads on *NewMadeleine* structures disturbs the background progression engine which eventually forms a bottleneck that eventually reduces the message rate.

We propose an original solution where the lock is dropped on the request submission path: instead of immediately accessing global state as soon as the user posts a request, we simply enqueue the request in a *lock-free* submission list, and we completely rely on the asynchronous progression engine to make the communication progress. In the case of *non-blocking* communication primitives (`MPI_Irecv`, `MPI_Isend`), they will be processed entirely in background. In the case of *blocking* primitives (`MPI_Wait`, `MPI_Send` and `MPI_Recv`), during the wait phase, the thread is temporarily integrated in the thread pool of the communication engine [18], which makes progress all requests, and implements mechanisms [19] to mitigate contention (adaptive passive wait).

B. Defer critical sections

We can push the same principle further and *defer* tasks that would need to acquire a lock (or tasks that would need to release a lock) instead of immediately acquiring/releasing the needed lock.

The lock-free request submission uses such a mechanism: requests are enqueued in a lock-free list; when the communication engine acquires the lock to make progression, it takes benefit of the already acquired lock to dequeue request from the lock-free list and to integrate them at the right place (in a hash-table or in the list for wildcard requests).

We apply this principle to all other places where lock acquire/release would be needed. In particular, *NewMadeleine* being event-based, the receive path is essentially made of upcalls from driver to the packet scheduling layer. When a driver completes a packet receipt, instead of locking immediately, it enqueues the packet in a lock-free list of completed packets that will be dispatched later by the progression engine. Higher in the stack, the end-user can register callbacks on requests (using *NewMadeleine* native interface, not MPI). These callbacks are specified as being called with all locks released. Instead of immediately releasing the lock to call the callback, then reacquiring the lock, we enqueue the event, and dispatch their notification in batch just after we released the lock.

The same goes for the request sequence number introduced in Section IV-A2. Since it is a global counter, it would need locking if it were assigned upon request submission. We choose to assign it only at the time where the progression engine already holds the lock and we dequeue requests from the lock-free submission list. In practice, the lock that protects global structures is only acquired for matching, *i.e.* algorithms 1 and 2, and all pending operations that needs the lock are executed while the lock has been acquired for matching.

C. Lock as little as possible

Finally, the lock that has the lower cost is the lock that is never acquired. We carefully refactored the library so as to take locks as little as possible. In particular, all the low-level activity at driver level runs without lock. Pushing a packet on the network or pulling data from the network doesn't access the global state, only our driver private data, which we designed to be isolated and thus re-entrant. For the *NewMadeleine* core, locking is mandatory but, as described in Section V-B, the lock acquisition is done mostly from inside progression engine, decorrelated as much as possible from the driver layer and from the user calls to the API. As a consequence, most non-blocking calls to the library involve no lock. Blocking calls are integrated with the policy of the progression engine. No blocking lock is done: since locking is done from inside the progression engine, a busy lock means another thread is already making progression, so we give hand instead of waiting to do something that was already done by someone else. Hence we rely on non-blocking locking (a.k.a. `pthread_spin_trylock`). Eventually, on the critical path to send or receive data, the lock is acquired only once in most cases.

VI. Evaluation

In this section, we evaluate the performance obtained by the mechanisms proposed in this paper and compare it against state-of-the-art MPI libraries. Implementation has been performed in the *NewMadeleine* communication library. We evaluate it through its MPI layer called MadMPI.

We have run latency benchmarks to measure overhead, a micro-benchmark to measure scalability with messages in order and out-of-order, and a Cholesky factorization to measure the impact on real-world computation codes. Most relevant benchmarks, *i.e.* where a gain is expected, are represented with larger graphs for better readability.

We have run the benchmarks on two different clusters. The first one is `inti` from CEA. Nodes are dual Xeon E5-2680 at 2.7 GHz, with 16 cores and 64 GB RAM, and equipped with Connect-IB *InfiniBand* QDR (MT27600). The default MPI implementation on this cluster is OpenMPI 1.8. Since this version is ancient, for the sake of fairness we have compiled OpenMPI 3.0, MPICH 3.2 with `libfabric`, and MVAPICH2 2.3b for a large panel of MPI libraries.

The second one is `plafrim`. Nodes are dual-Xeon E5-2680 with 24 cores and 128 GB RAM, equipped with Intel Omni-Path 100 Gb/s (HFI Silicon 100 Series). The default MPI implementation on this cluster is OpenMPI 1.10. Since this version is ancient, for the sake of fairness we have compiled OpenMPI 3.0. As an alternative, Intel MPI 2017 is available as a module. Additionally, we compiled MVAPICH2 2.3b with PSM2.

A. Overhead

Since the mechanisms presented in this paper introduce a significant amount of additional instructions on the communication critical path, we first measure the latency to evaluate whether the overhead is acceptable.

We used the `sendrecv` benchmark from our MPI benchmark suite [20], [21] and ran it on the choice of MPI

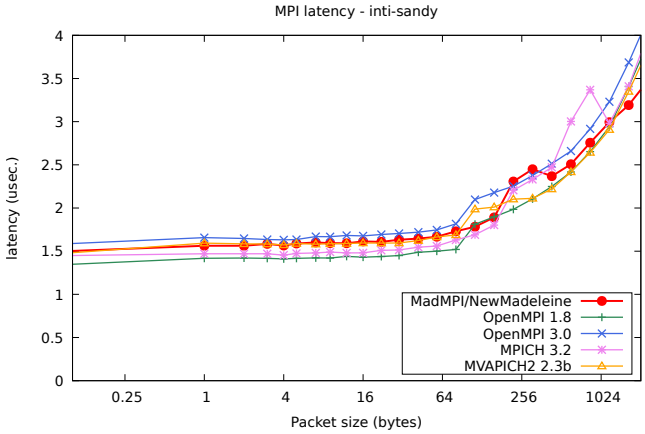

Fig. 1. Ping-pong latency on cluster `inti` (InfiniBand).

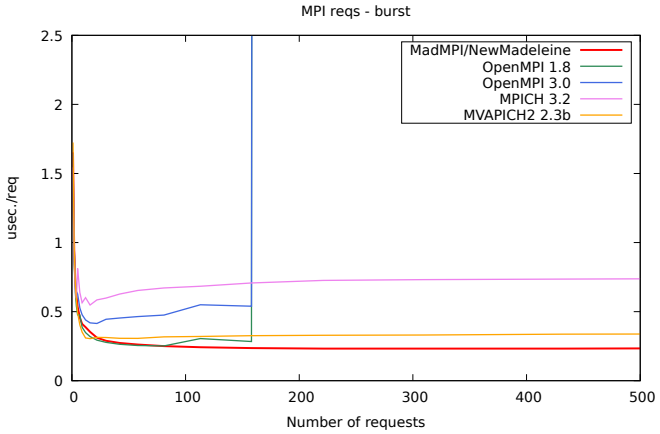

Fig. 2. Burst of reqs, in order, same tag, on cluster `inti` (InfiniBand)

libraries available on cluster `inti`. The benchmark performs a pingpong on a pair of MPI ranks on different nodes. The results are depicted in Figure 1. We observe that the difference of latency between the various MPI implementations is small. The latency measured for *NewMadeleine* is a little bit higher than average, but not the worst. As a total, the cost of the presented mechanisms on latency is bearable.

Further micro-benchmarks of collective communications, one-sided communications, and NAS benchmarks have shown that the overhead caused by the mechanisms presented in this paper has a negligible impact on overall communication performance.

B. Micro-benchmarks

To evaluate the behavior of MPI libraries in the presence of large numbers of point-to-point requests, we have developed our own benchmarks which are distributed in the MadMPI benchmark suite [20], [21].

1) Benchmarks: The first benchmark is *burst*. It is based on ping-pong between two nodes. The sender performs a burst of N send of 1 byte with `MPI_Isend`, with a variable N, then calls `MPI_Waitall`; the receiver posts a burst of N receive of 1 byte with `MPI_Irecv`, then calls `MPI_Waitall`. This benchmark evaluates only the ability for the MPI library to sustain N simultaneous requests. However, all messages being sent in order, on the same tag, even naive tag-matching algorithms are expected to get good performance.

The second benchmark is *shuffle*. It is the same as the *burst* benchmark except that messages are all sent on different tags, and the receiver shuffles its tags set so as to post the receive request in a random order. This benchmark is expected to exercise the tag-matching algorithm, and looks more like the communication scheme of irregular applications with lots of point-to-point communications.

For both benchmarks, we compute the time to transfer a single message (total time divided by N), and draw the time in function of N. Since a barrier is used to synchronize nodes to detect the end of the benchmark, it is not surprising that

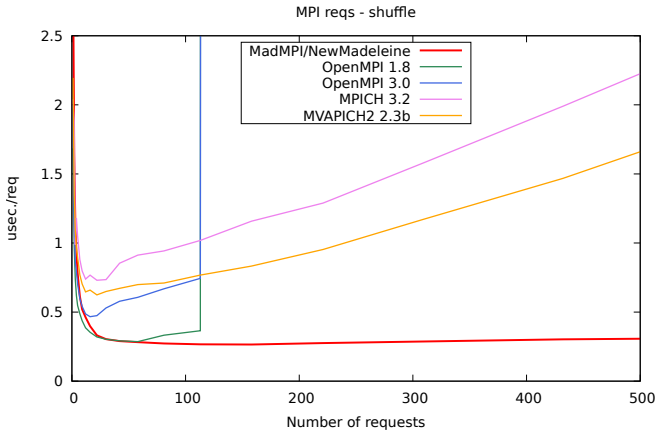

Fig. 3. Burst of reqs, out-of-order, multiple tags, on cluster `inti` (InfiniBand)

measured time may be higher than single-message latency for small values of N; however, this cost is amortized for higher values of N.

2) Results on InfiniBand: The results we obtained on cluster `inti` on InfiniBand are depicted in Figure 2 for the *burst* benchmark.

We observe that for this benchmark, *NewMadeleine*, MPICH and MVAPICH2 get almost constant-time results, while both versions of OpenMPI collapse past approximately 160 requests, with a jump to more than 500 μs (graph cropped for the sake of readability). However, even if MPICH and MVAPICH2 get results that converge asymptotically to a constant, this constant is higher than the result obtained by *NewMadeleine*.

The results for the *shuffle* benchmark that really measure the performance of the tag-matching facility are depicted in Figure 3. We observe that *NewMadeleine* gets a bounded time, while MPICH and MVAPICH2 get a latency linear with the number of requests. Both versions of OpenMPI are linear for small numbers of requests, but still explode past roughly 120 requests. This graph clearly demonstrate the strength of our

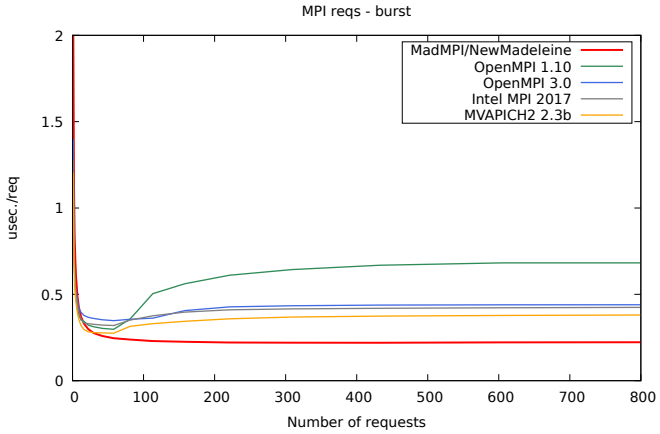

Fig. 4. Burst of reqs, in order, same tag, on cluster `plafrim` (OmniPath)

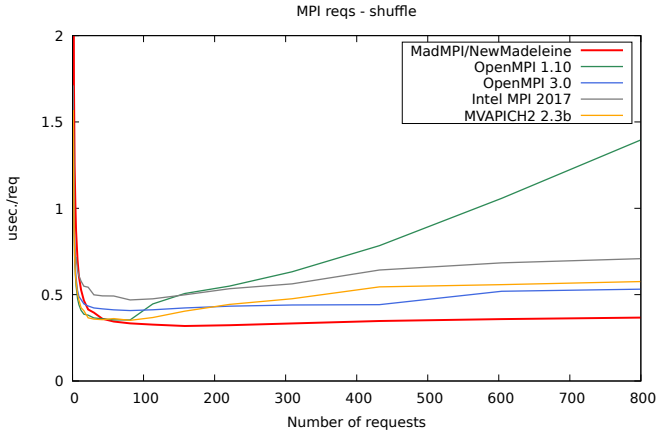

Fig. 5. Burst of reqs, out-of-order, multiple tags, on cluster `plafrim` (OmniPath)

tag-matching algorithm.

We have tested up to *millions* of concurrent requests (graph not shown here): *NewMadeleine* always gets constant time per requests, while no other MPI library was able to complete the test in a reasonable time.

3) Results on Omni-Path: The results we obtained on cluster `plafrim` on Intel Omni-Path are depicted in Figure 4 for the *burst* benchmark. We observe that all MPI libraries converge asymptotically towards a constant, this constant being lower for *NewMadeleine*. This shows the strength of *NewMadeleine* to sustain bursts of many messages.

The results for the *shuffle* benchmark, with out-of-order message, on Omni-Path are depicted in Figure 5. We observe that all MPI libraries get bounded results except the old OpenMPI 1.10. It is explained by the fact that on Omni-Path, MPI libraries rely on PSM2 Matched Queues (MQ) which directly provide tag-matching infrastructure. It is not surprising that all libraries benefit from PSM2 properties. However, *NewMadeleine* once again gets the best results.

As a summary, on InfiniBand for a benchmark that sends messages in random order, as depicted in Figure 3, *NewMadeleine* gets constant and low time where competitors obtain linear time or collapse past a given threshold. On Omni-Path, the complexity is asymptotically the same but *NewMadeleine* gets overall better performance.

C. Cholesky Factorization

To evaluate the impact of this work on a computation code, we have used the task-based tile Cholesky factorization from the Chameleon software [3]. This dense linear algebra code is based on tasks scheduled through the StarPU [22] runtime system, which itself uses MPI for inter-node communications. Since communications are triggered by tasks data-flow without synchronization, they do not use MPI collectives at all and rely exclusively on MPI point-to-point communications. Code instrumentation has shown that 50+ communications are typically active at the same time.

We have run `time_spotrf_tile`, the benchmark integrated in the Chameleon package, on both clusters and using various MPI libraries. The results are shown as GFlop/s on the Y axis and matrix size on the X axis. Some graphs are shorter than the others and do not reach large matrix size; this is due to the fact that slower runs have exceeded the time allocated for the job and have been killed by `slurm` before they completed.

The performance obtained on cluster `inti` on 4 nodes (64 cores) is depicted in Figure 7. We observe that the best overall performance is obtained by *NewMadeleine* and MPICH. Performance obtained by other MPI libraries is not so bad but not on par with the leaders. We added as a thin red line (labeled *old NawMadeleine* on graphs) the revision of *NewMadeleine* (svn `r27200`) just before the optimizations presented in this paper were implemented, so as to be able to distinguish the precise impact of the mechanisms presented in this paper and not only compare an MPI library against another as a whole.

The performance obtained on cluster `inti` on 196 nodes (3136 cores) is depicted in Figure 6. We were not able to run MVAPICH2 on 100+ nodes on this cluster, so its graph is missing here. Notice that the MPICH graph is shorter due to Chameleon using the full 31-bits MPI tag space for large matrices, but MPICH provides only 30-bit tags, thus cannot reach large matrix (roughly size past 170k). The larger number of nodes allows for larger matrix size, which increases the number of messages exchanged and the stress imposed to the point-to-point communication system. We observe that for small matrices, where not so many messages are exchanged, MPICH is fastest. For matrices larger than 80k, *NewMadeleine* outperforms all the contenders by a factor of at least 35 %. We observe that the difference is huge between the current and the old version of *NewMadeleine*.

We have run the benchmark on the `plafrim` cluster with Omni-Path network too. Since support for PSM2 driver was added to *NewMadeleine* after the work described in this paper, it was not possible to compare old and new version of *NewMadeleine* on this cluster. The performance obtained on 4 nodes is depicted in Figure 8. The hierarchy is similar to what

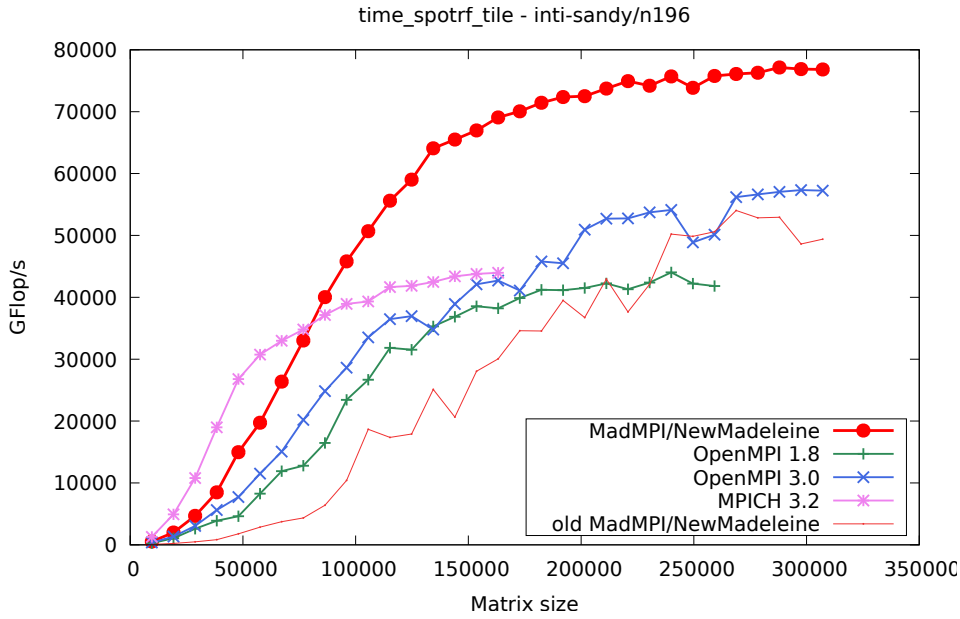

Fig. 6. Chameleon Cholesky factorization performance on cluster `inti` on 196 nodes (3136 cores).

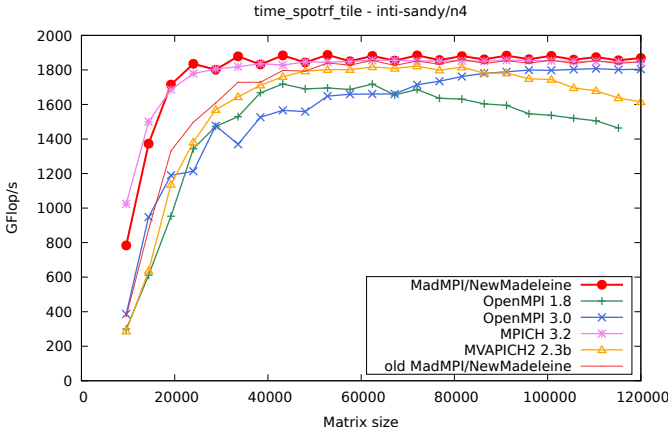

Fig. 7. Chameleon Cholesky factorization performance on cluster `inti` on 4 nodes (64 cores).

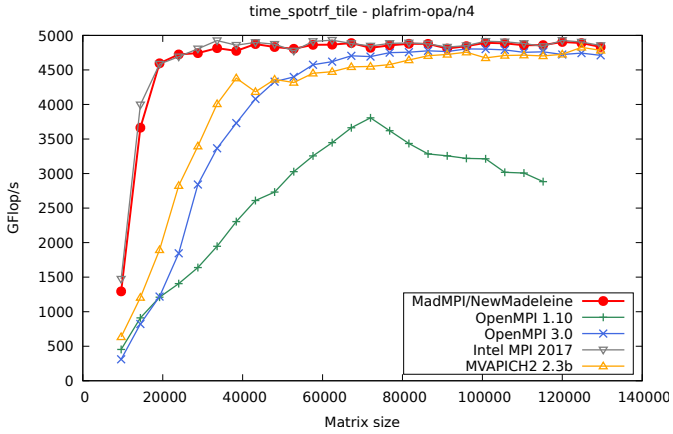

Fig. 8. Chameleon Cholesky factorization performance on cluster `plafrim` on 4 nodes (96 cores).

was obtained on 4 nodes on the other cluster (given that Intel MPI is a derivative of MPICH). The performance obtained on 16 nodes (96 cores) is presented in Figure 9. We were not able to make MVAPICH2 complete the benchmark on 16 nodes. On this configuration, *NewMadeleine* performance is clearly superior to OpenMPI and Intel MPI. Since the difference was not so large for the micro-benchmarks on this machine, we guess another property from the MPI library should make the difference and not only its ability to sustain flows of large number of requests.

As a summary, *NewMadeleine* is always the best performing MPI library except for very small matrices on 196 nodes. It shines especially in cases with many requests, namely on 196 nodes and large matrices. When comparing the performance of the current implementation and the old implementation before we implemented the mechanisms presented in this article, it is clear that the performance in the presence of a large number of requests is critical. However, this is not the only property of the MPI library that plays a role for this code.

VII. Conclusion and Future Works

New kinds of HPC applications and runtimes change the way the MPI library is used. These runtimes impose large numbers of point-to-point requests that the MPI library must sustain. It causes a scalability issue that arise in the tag-matching subsystem and in the progression engine of the communication library.

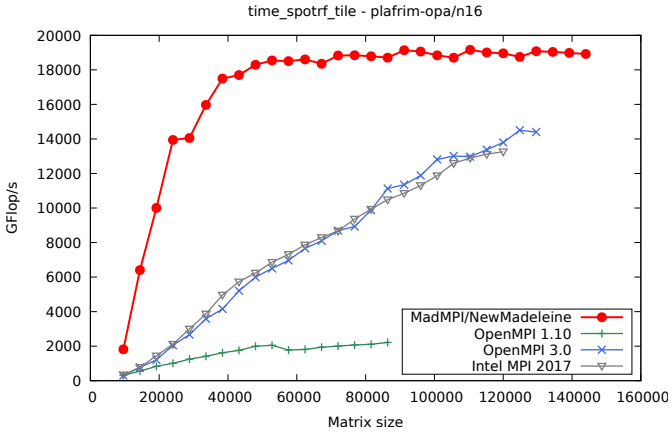

Fig. 9. Chameleon Cholesky factorization performance on cluster `plafrim` on 16 nodes (384 cores).

In this paper we have proposed an algorithm that is able to perform matching in constant time in all cases, even with requests for any source or any tag. We have proposed mechanisms to avoid contention between progression and requests submissions, so that bursts of requests submissions do not hinder communication background progression. We have implemented these solutions in our *NewMadeleine* communication library. Micro-benchmarks with large number of point-to-point requests show that our solution scales better than state of the art MPI libraries, especially on InfiniBand hardware. Benchmarks with computation code exhibit a dramatic improvement of performance with our solution, with an increase of 35-40 % of computation speed compared to other MPI libraries.

The code for the MadMPI library and the benchmarks has been released as open source and is available for download [21]. The mechanisms described in this paper are enabled by default in the released code.

Acknowledgment

Experiments presented in this paper were carried out using the PLAFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from LABRI and IMB and other entities: Conseil Régional d'Aquitaine, FeDER, Université de Bordeaux and CNRS (see https://plafrim.bordeaux.inria.fr/).

References

[1] W. Schonbein, M. G. F. Dosanjh, R. E. Grant, and P. G. Bridges, "Measuring multithreaded message matching misery," in *Euro-Par 2018: Parallel Processing*, M. Aldinucci, L. Padovani, and M. Torquati, Eds. Cham: Springer International Publishing, 2018, pp. 480–491.

[2] J. Dongarra, M. Abalenkovs, A. Abdelfattah, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, I. Yamazaki, and A. YarKhan, "Parallel programming models for dense linear algebra on heterogeneous systems," *Supercomputing Frontiers and Innovations*, vol. 2, no. 4, 2016. [Online]. Available: http://superfri.org/superfri/article/view/90

[3] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. Thibault, "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model," *IEEE Transactions on Parallel and Distributed Systems*, 2017. [Online]. Available: https://hal.inria.fr/hal-01618526

[4] O. Aumage, E. Brunet, N. Furmento, and R. Namyst, "NewMadeleine: a Fast Communication Scheduling Engine for High Performance Networks," in *Workshop on Communication Architecture for Clusters (CAC 2007), workshop held in conjunction with IPDPS 2007*, Long Beach, California, United States, Mar. 2007. [Online]. Available: https://hal.inria.fr/inria-00127356

[5] R. Brightwell, S. Goudy, and K. Underwood, "A preliminary analysis of the mpi queue characterisitics of several applications," in *2005 International Conference on Parallel Processing (ICPP'05)*, June 2005, pp. 175–183.

[6] R. Keller and R. L. Graham, "Characteristics of the unexpected message queue of mpi applications," in *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 179–188.

[7] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "Enabling concurrent multithreaded mpi communication on multicore petascale systems," in *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 11–20.

[8] J. A. Zounmevo and A. Afsahi, "An efficient mpi message queue mechanism for large-scale jobs," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec 2012, pp. 464–471.

[9] M. Flajslik, J. Dinan, and K. D. Underwood, "Mitigating mpi message matching misery," in *High Performance Computing*, J. M. Kunkel, P. Balaji, and J. Dongarra, Eds. Cham: Springer International Publishing, 2016, pp. 281–299.

[10] M. Bayatpour, H. Subramoni, S. Chakraborty, and D. K. Panda, "Adaptive and dynamic design for mpi tag matching," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2016, pp. 1–10.

[11] T. Hoefler, G. Bronevetsky, B. Barrett, B. R. de Supinski, and A. Lumsdaine, "Efficient mpi support for advanced hybrid programming models," in *Recent Advances in the Message Passing Interface*, R. Keller, E. Gabriel, M. Resch, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 50–61.

[12] É. Brunet, F. Trahay, and A. Denis, "A Multicore-enabled Multirail Communication Engine," in *Proceedings of the IEEE International Conference on Cluster Computing*. Tsukuba, Japan: IEEE Computer Society Press, Sep. 2008, pp. 316–321, poster Session. [Online]. Available: http://hal.inria.fr/inria-00327158

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[14] D. Liu, Z. Cui, S. Xu, and H. Liu, "An empirical study on the performance of hash table," in *2014 IEEE/ACIS 13th International Conference on Computer and Information Science (ICIS)*, June 2014, pp. 477–484.

[15] B. Jenkins, "Hash functions," *Dr Dobb's Journal*, Sep. 1997.

[16] MPI Forum, "MPI: A Message-Passing Interface Standard Version 3.1," Jun. 2015.

[17] S. Friedman, N. Leidenfrost, B. C. Brodie, and R. K. Cytron, "Hashtables for embedded and real-time systems," in *in Proceedings of the IEEE Workshop on Real-Time Embedded Systems*, 2001, p. 2001.

[18] A. Denis, "pioman: a pthread-based Multithreaded Communication Engine," in *Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Turku, Finland, Mar. 2015. [Online]. Available: https://hal.inria.fr/hal-01087775

[19] F. Trahay, É. Brunet, and A. Denis, "An analysis of the impact of multi-threading on communication performance," in *CAC 2009: The 9th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2009*. Rome, Italy: IEEE Computer Society Press, May 2009. [Online]. Available: http://hal.inria.fr/inria-00381670

[20] A. Denis and F. Trahay, "MPI Overlap: Benchmark and Analysis," in *International Conference on Parallel Processing*, ser. 45th International Conference on Parallel Processing, Philadelphia, United States, Aug. 2016. [Online]. Available: https://hal.inria.fr/hal-01324179

[21] "PM2 high performance runtime system," http://pm2.gforge.inria.fr/.

[22] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par 2009*, ser. LNCS, Delft, Netherlands, Aug. 2009. [Online]. Available: https://hal.inria.fr/inria-00384363