Performance improvements of parallel applications thanks to MPI-4.0 Hints

Maxim Moraru, Adrien Roussel, Hugo Taboada, Christophe Jaillet, Marc Pérache, Michaël Krajecki

Performance Improvements of Parallel Applications thanks to MPI-4.0 Hints

1st Maxim MORARU
LICIIS
Université de Reims Champagne Ardenne
51097 Reims, France
maxim.moraru@univ-reims.fr

2nd Adrien ROUSSEL
CEA, DAM, DIF, LRC DIGIT
F-91297 Arpajon, France
Université Paris-Saclay, CEA,
Laboratoire en Informatique
Haute Performance pour
le Calcul et la simulation
Bruyères le Châtel, France
adrien.roussel@cea.fr

3rd Hugo TABOADA
CEA, DAM, DIF
F-91297 Arpajon, France
Université Paris-Saclay, CEA,
Laboratoire en Informatique
Haute Performance pour
le Calcul et la simulation
Bruyères le Châtel, France
hugo.taboada@cea.fr

4th Christophe JAILLET
LICIIS, LRC DIGIT
Université de Reims Champagne Ardenne
51097 Reims, France
christophe.jaillet@univ-reims.fr

5th Marc PÉRACHE
CEA, DAM, DIF, LRC DIGIT
F-91297 Arpajon, France
Université Paris-Saclay, CEA,
Laboratoire en Informatique
Haute Performance pour
le Calcul et la simulation
Bruyères le Châtel, France
marc.perache@cea.fr

6th Michael KRAJECKI
LICIIS, LRC DIGIT
Université de Reims Champagne Ardenne
51097 Reims, France
michael.krajecki@univ-reims.fr

***Abstract*—HPC systems have experienced significant growth over the past years, with modern machines having hundreds of thousands of nodes. Message Passing Interface (MPI) is the *de facto* standard for distributed computing on these architectures. On the MPI critical path, the message-matching process is one of the most time-consuming operations. In this process, searching for a specific request in a message queue represents a significant part of the communication latency. So far, no miracle algorithm performs well in all cases. This paper explores potential matching specializations thanks to hints introduced in the latest MPI 4.0 standard. We propose a hash-table-based algorithm that performs constant time message-matching for no wildcard requests. This approach is suitable for intensive point-to-point communication phases in many applications (more than 50% of CORAL benchmarks). We demonstrate that our approach can improve the overall execution time of real HPC applications by up to 25%. Also, we analyze the limitations of our method and propose a strategy for identifying the most suitable algorithm for a given application. Indeed, we apply machine learning techniques for classifying applications depending on their message pattern characteristics.**

***Index Terms*—HPC, Distributed programming, MPI Matching, MPI 4.0 Sessions**

I. Introduction

Supercomputer systems continuously include more compute nodes and more computing resources per node. The most powerful ones have thousands of nodes. Furthermore, the HPC community aims to reach the Exascale by 2022. This target is highly related to the capacity of the system to provide efficient inter-node communication. Current solutions strongly rely on the Message Passing Interface (MPI) standard. It is the most used message passing environment and has become almost a synonym for distributed computing. It describes a list of routines used for parallel computing in a distributed system and has several implementations.

Under MPI, send and receive operations are identified by a header, conventionally composed of a communicator identifier, source process identifier, tag, and a buffer location. The receiver posts an envelope with these identifiers, and if it matches an incoming message, data is delivered to the buffer location. This operation is usually called MPI matching and is performed on the receiver side. MPI matching is on the critical path and is known to be computationally complex [1], [2]. Thus, its performance significantly impacts the message latency [3].

There are several implementations of the matching algorithm. The most common one uses two different queues for receive operations. The posted receive queue (PRQ) is used to store receive requests that were not resolved, and the unexpected message queue (UMQ) is used for storing messages that arrived without a prior receive request. During the matching operation, queues are traversed in search of a match between a message and a request. Depending on the message position, the overhead can have a significant or negligible impact on the overall performance. An ideal scenario would be one where the first message of the queue always corresponds to the searched

request. Nevertheless, its position depends on the application behavior and system noise. Thus, messages that arrive in an unpredictable order can significantly slow down the execution.

While several attempts aim to reduce this overhead at a software and hardware level, there is still no consensus. Hence, each MPI implementation uses a different approach. The primary factor which makes this operation complex is the presence of MPI wildcards (i.e. `MPI_ANY_SOURCE` and `MPI_ANY_TAG`). Because it is not mandatory to specify a source and a tag, communication libraries need to keep either one large list for all messages or try to separate wildcards in a way that preserves global order. The new MPI-4.0 standard proposes some special hints which allow users to provide information to direct optimization. The `mpi_assert_no_any_source` and `mpi_assert_no_any_tag` hints, when set, inform the MPI implementation that the application will not use wildcards for a specific communicator. In particular, this can help an MPI implementation improve the matching algorithm.

Nevertheless, attaching them to a specific communicator can present some disadvantages. The fact that the user can specify this information at any moment makes its exploitation more complex. The MPI library is initialized when the user adds an attribute to the communicator. Therefore all data structures are already built. Moreover, changing the algorithm on the fly could be challenging if there are already messages in the matching lists. Also, setting and getting communicator attributes is an operation on the critical path and could add an overhead, especially for smaller buffers [4]. We propose to use the MPI hints approach at a session level to solve these limitations. The Session model is a new MPI-4.0 addition that offers an isolated environment and allows it to be customized. For example, it is possible to specify two different thread-level supports for two sessions inside an MPI program. Since the hints are passed at initialization, the communication library can prepare all necessary data structures and apply an optimized matching algorithm directly.

This paper presents a detailed study of the possible performance gain when `mpi_assert_no_any_source` and `mpi_assert_no_any_tag` hints are set at a session level. We propose a constant time matching algorithm for applications that do not use wildcards and compare it to a common one used by several MPI implementations. We show how this approach can result in a speedup of 93% for some microbenchmarks and 25% for some proxy applications. Also, we take care to verify its behavior in disadvantageous cases. Indeed, we performed an exhaustive study and show how the performance of the matching algorithms could drastically change depending on the context. Finally, we present a solution for identifying applications that could benefit from our matching algorithm. Our contributions are the following:

- The design and evaluation of a hint-driven MPI matching algorithm;
- Its integration into a communication library via the MPI 4.0 Sessions;
- The implementation of a matching profiler to analyze cases where this approach may cause an overhead;
- A methodology based on a simple machine learning model to detect applications that could benefit from our solution.

The rest of this paper is organized as follows. The next section gives an overview of related work and background. Section 3 describes an optimized hint-driven MPI matching algorithm and also provides a theoretical analysis of its performance. In section 4, we test our algorithm on some microbenchmarks and HPC applications and highlight its behavior in good, bad, and realistic scenarios. Section 5 provides a solution for identifying the most appropriate matching algorithm for a given application. The last section presents our conclusions and future work.

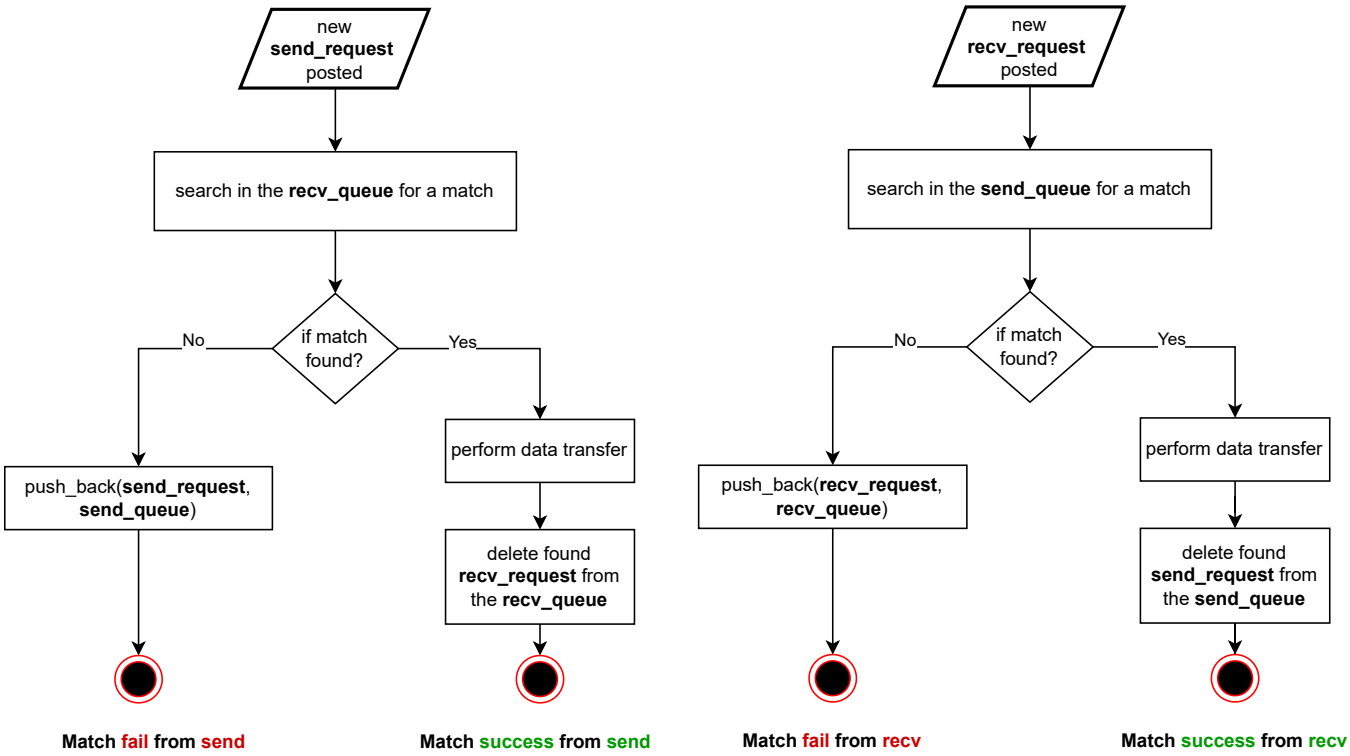
II. Background and Related Work

Under MPI, there are two matching queues: one for receive and one for send requests. The MPI matching algorithm is triggered when a new request is posted. Depending on the request type (i.e. receive or send), the algorithm's workflow can differ. Figure 1 presents the general workflow of the MPI matching operation. When a new send request arrives (Figure 1a), we traverse the receive queue in search of a match. The two elements match if they have the same communicator id, source, and tag. It is also possible to use wildcards for the source and tag of a receive request. If the match is found, we perform the data transfer and delete the receive request from the receive queue. Otherwise, the send request is added to the send queue. Symmetrically when a new receive request arrives (Figure 1b), we search for a match in the send queue. Thus, there are four paths: match failed upon a send request arrival, match succeeded upon a send request arrival, match failed upon a receive request arrival, and match succeeded upon a receive request arrival. The `queue` refers to a FIFO data structure that can vary depending on the MPI implementation.

Modern MPI implementations use different approaches for the matching algorithm. For example, the OpenMPI implementation [5] maintains a separate queue for each *(communicator id, source)* pair. Thus, the queue is traversed only for searching the tag. While this approach improves the average search depth, it can lead to high memory consumption [6]. In contrast, MPICH [7] uses a single linked list for storing all *(communicator id, source, tag)* requests. This data structure offers minimal memory consumption, but in the worst cases, message matching requires visiting any element of the entire queue.

The MPI matching has been studied for several years and in different contexts. Though, there are two main directions: hardware and software-based approaches. These are not necessarily interchangeable and can be combined to increase performance.

a) Hardware: Hemmert et al. [8] proposed a microcode engine for MPI matching composed of two ALUs: one for binary operations and the other for wildcarded messages. Their



(a) Matching algorithm upon send request arrival

(b) Matching algorithm upon receive request arrival

Fig. 1: General overview of the MPI matching workflow.

goal is to offer a flexible solution while maintaining good overall performance.

Another approach is to accelerate the matching algorithm by using modern hardware capabilities. Dosanjh et al. [2] proposed accelerating this operation by exploiting the processor's AVX-512 feature. This approach uses vectorized "integer-compare-equals" operations and a set of vectors for storing tags and sources. The output of these operations is a bit-map containing the id of matching messages. Xiong et al. [9] introduced the idea of offloading the matching part on FPGA hardware. Their solution covers messages from PRQ and UMQ.

Based on the above studies and other research, new generations of MPI message matching Network Interface Cards (NICs) were proposed: Bull eXascale Interconnect (BXI) [10], InfiniBand ConnectX-5 [11].

b) Software: Ghazimirsaeed et al. [12] showed how using multiple queues can impact the performance of the matching algorithm. They proposed to group MPI processes in separate classes and attach each cluster to distinct message queues. Thus, a specific process needs to iterate through its assigned queue instead of the global one. The way MPI processes are grouped can significantly impact the performance, as queues can contain a larger or smaller number of messages during program execution. Consequently, Ghazimirsaeed et al. proposed using a K-means algorithm to partition all MPI processes efficiently. This approach requires a first run of the application to gather the necessary data. Then, this information can be used for improving matching for further executions.

The NewMadeleine communication library proposed a constant-time MPI matching algorithm [13], even for wildcard requests. Its approach is based on four data structures (three hash tables and one linked list) allowing to store any combination of MPI requests: *(source, tag)*, *(source, wildcard)*, *(wildcard, tag)*, *(wildcard, wildcard)*. Thus, there are four places to be checked to perform the matching upon a send request arrival: three hash table lookups and four accesses to the head list. Symmetrically, upon a receive request arrival, there is only one place to be checked. Each time, we only check the head of the list, and therefore, the algorithm's complexity is O(*1*).

III. Hint-Driven Matching Algorithm

The presence of wildcarded requests makes the MPI matching algorithm non-trivial. To the best of our knowledge, there is no universal solution adapted for all communication patterns.

Thanks to the new MPI 4.0 hints the user can inform the communication library that no wildcarded requests will be used. Thus, by focusing on this particular context it is possible to explore even more optimized matching algorithms.

While a hint-driven solution may seem restrictive, there is a large number of real MPI applications that do not use

any_source and any_tag requests. For example, more than 50% of CORAL benchmarks [14] would benefit from this approach.

A. Constant Time Implementation

To further optimize the matching operation we propose an algorithm based on a single hash table.

All the table elements are identified by a hash key composed of the communicator identifier, source, and message tag. Each element contains a linked list with all messages for the corresponding identifier. Algorithm 1 describes the two main operations of our MPI matching implementation. Procedure *insertMsgToHash* allows inserting a new request in the hash table. First, we check if there is already an entry with the same key (lines 2-4). Depending on the lookup result, we either append the message to an existing list (line 5) or create a new hash table entry with this new element (lines 7-8). During this operation, we organize all the requests by their arrival order. Therefore, the searching operation (*searchForMatching* function) becomes a single lookup in a hash table since it is always the head of the list which corresponds to the searched message (line 15).

Algorithm 1 Hash Table Matching Algorithm

```
1:  procedure INSERTMSGTOHASH(h_tab, msg)
2:      key ⟵ {msg.comm_id, msg.source, msg.tag}
3:      msg_list ⟵ HASH_FIND(h_tab, key);
4:      if msg_list ≠ NULL then
5:          DL_APPEND(msg_list, msg);
6:      else
7:          new_msg_list = new_list(msg);
8:          HASH_ADD(h_table, new_msg_list, key);
9:      end if
10: end procedure
11:
12: function SEARCHFORMATCHING(h_tab, key)
13:     msg_list ⟵ HASH_FIND(h_tab, key);
14:     if msg_list ≠ NULL then
15:         matched_msg ⟵ pop_first(msg_list);
16:     else
17:         matched_msg ⟵ NULL;
18:     end if
19:     return matched_msg;
20: end function
```

The described algorithm has a time complexity of O(*1*) when doing an amortized performance analysis, compared to O(*n*) for a linked list approach. While it has a theoretical advantage, this approach is only practical when there are many messages in the matching queue and when these messages are in a shuffled order. A single hash table lookup is more efficient than traversing a long linked list but is less efficient than checking the first element of the list.

For the hash table data structure, we decided to use the uthash library[1]. It is an open-source implementation of a hash table that uses the *separate chaining* technique [15] for handling collisions. To provide more performance, uthash can dynamically adapt its data structure. When the number of elements in a bucket exceeds a certain threshold, the number of buckets is doubled, and the items are redistributed. Among all hash functions provided by the library, we choose the Jenkins one for performance concerns.

[1]Available at https://github.com/troydhanson/uthash

B. Integration into MPC-MPI

The evaluation of Algorithm 1 was possible thanks to the MPC-MPI communication library. We choose this particular library because it provided a Session implementation quickly after the release of the MPI 4.0 standard. Multi-Processor Computing [16] is a framework for programming distributed and shared memory architectures. The main idea of the framework is to represent MPI processes as user-level threads. It also includes an extension called MPC-MPI [17] which fully implements the MPI standard. Its main advantage concerns memory consumption. It uses significantly less memory than other MPI implementations. Also, the original Multi-Processor Computing matching algorithm uses a linked list approach by storing all requests in a global queue.

We integrated Algorithm 1 into the MPC-MPI library by using the MPI-4.0 Sessions. The hash table matching algorithm can be activated by setting both `mpi_assert_no_any_source` and `mpi_assert_no_any_tag` hints. While the MPI standard introduced these hints as info keys for communicators, it is also possible to use them for sessions. Indeed, in both cases, the user must specify this information via an MPI_Info object that fully respects the MPI 4.0 standard.

Accordingly, we verify the associated MPI_Info object at session initialization and set an appropriate context for the chosen algorithm. Indeed, this is the most opportune moment for preparing all necessary data structures, as the program has not started yet. Due to this approach, removing any overhead related to checking communicator attributes on the critical path is possible.

C. Theoretical Analysis

This section presents a theoretical analysis of our matching algorithm and compares it to NewMadeleine's approach. Both algorithms are constant time when doing an amortized analysis. However, assuming the absence of any source, any tag requests allowed us to even more optimize the matching operation.

To highlight the theoretical advantage of our approach, we implemented a program that simulates the four different paths of both matching algorithms.

Firstly, we implemented the NewMadeleine's algorithm. In our implementation, we use four hash tables instead of one linked list and three hash tables. The original algorithm uses the list to store the fully wildcarded request. In the case of *NewMadeleine*, a list is sufficient because it does not require a communicator matching. Indeed, the communicator matching is done automatically via some *NewMadeleine internal tags*.

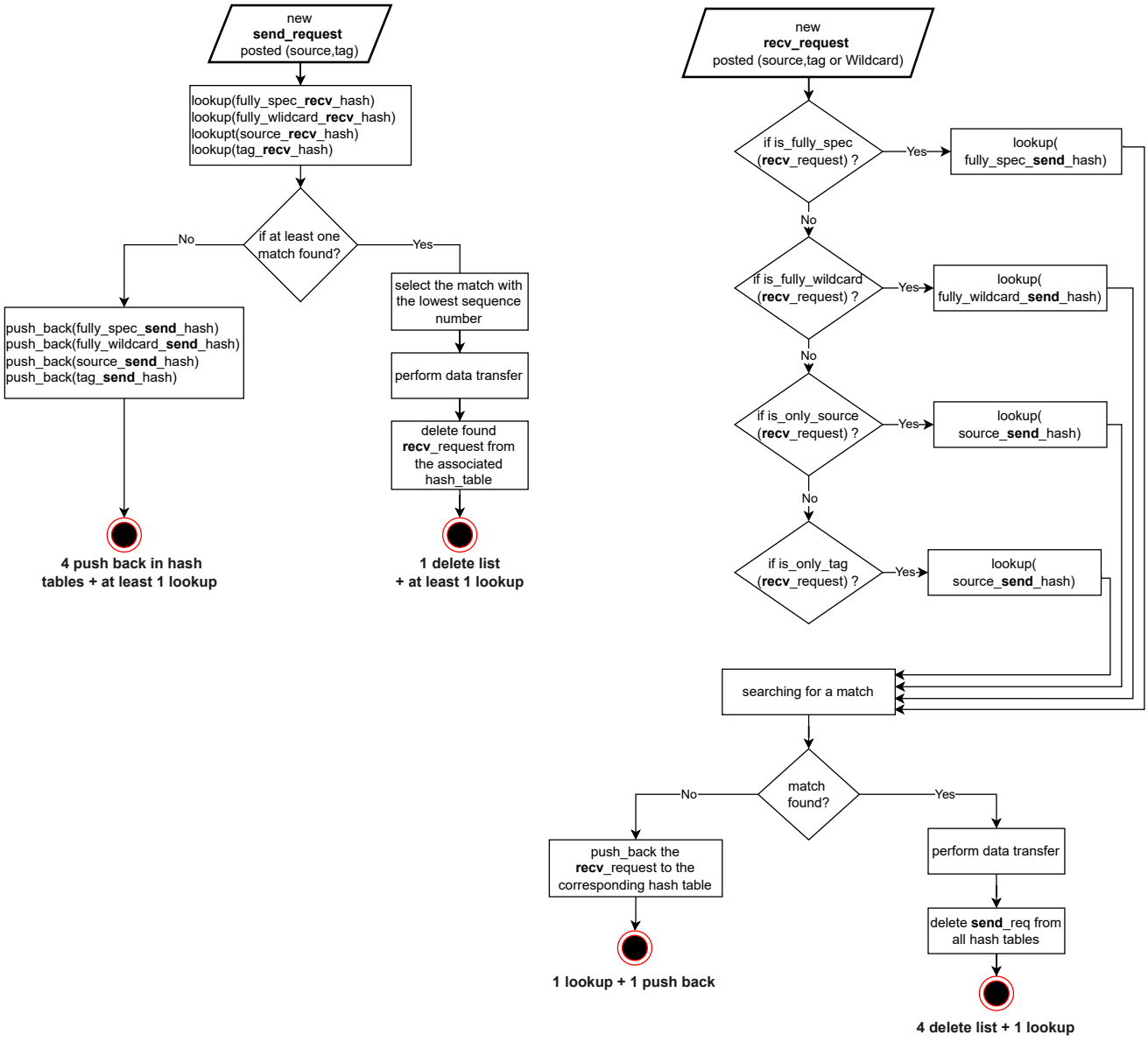


Fig. 2: NewMadeleine matching algorithm workflow.

For most other communication libraries, there is also a need to match the communicator id. Therefore we replace the linked list with a hash table. Figure 2 details our implementation and presents the four paths of the NewMadeleine's algorithm with their associated cost. To compare, our algorithm needs a single lookup and delete list operations when a matching request is found. Symmetrically, when a matching request is not found, a single lookup and push back operation is performed. Secondly we run the four matching matching paths for both algorithms. Our simulation program starts by posting N receive requests. All those requests will fail to find a match as there were no previous send requests. Next, we post N send requests, which will all succeed. We measure their associated execution times, and then we perform a symmetrical operation (i.e. start by posting N send requests which all fail). In this manner, it is possible to simulate the time for performing each of the four branches of a matching algorithm. Our algorithm was designed for a no wildcard context, so we use exclusively fully specified requests for our simulation.

We generate random requests with the following parameters:

- $0 \leq communicator\_id \leq 100$
- $0 \leq tag \leq 100$
- $0 \leq source \leq 500$

Table I presents the time necessary for performing a single matching operation. For the *success from receive* and *fail from send* paths, the hash-based algorithm shows significantly better performance over the NewMadeleine's approach. For the other

matching paths, obtained experimental results are equivalent. Note that in a no wildcarded context, the *fail from receive* and *success from send* paths require the same operations for both matching algorithms.

Finally, to further compare our approach with NewMadeleine's algorithm, we also integrated the last one into the MPC-MPI. As described previously, we implemented an adapted version of the algorithm and used a fourth hash table instead of a linked list.

IV. Experimental results

This section describes our experimental environment and presents results on some micro-benchmarks and proxy HPC applications. We took the MPC-MPI linked list matching algorithm as a reference and compared it with our constant time solution and with NewMadeleine's approach.

A. Experimental Context

We chose to validate the performance of our approach on some micro-benchmarks and representative proxy applications. We executed each program with each MPI matching algorithm. Afterward we compute their total execution time and highlight the performance improvements of our approach. All benchmarks were run with the MPC-MPI library, which already supports MPI-4.0 Sessions. Therefore, we instrumented each benchmark to obtain a true session context. Essentially, we replaced all MPI_init by MPI_Session_init and set up an appropriate MPI_Info object.

All experiments from this section were run on *Inti*, a cluster hosted at CEA. The AMD partition contains nodes equipped with AMD EPYC 7H12 processors and ConnectX-6 InfiniBand [18] controllers.

B. Micro-benchmarks

To evaluate the behavior of our algorithm in a favorable context, we decided to use the *shuffle* benchmark from the MadMPI benchmark suite [2]. It provides a ping-pong operation with non-blocking MPI point-to-point routines. The server starts by posting *N* receive requests of 1 byte with tags going from *0* to *N-1* (the order in which messages are posted corresponds to the message tag). At the same time, the client posts *N* send requests with tags from *0* to *N-1* but in a random order. Both actors perform an (MPI_Waitall) to wait for all requests to finish. Next, the same operation is performed with reversed roles (i.e. the "pong" action).

Figure 3 highlights the effectiveness of the hash-based matching algorithm in the presence of many point-to-point communications. The X-axis indicates the number of posted requests during a half roundtrip (i.e. *N*). The Y-axis shows the execution time of the ping-pong operation divided by two (i.e. the "one-way" time). The number above the graph highlights the performance improvements of our algorithm over a linked list approach (in blue on the graph) and the NewMadeleine's approach (in orange on the graph). We can notice that in the presence of several simultaneous requests,

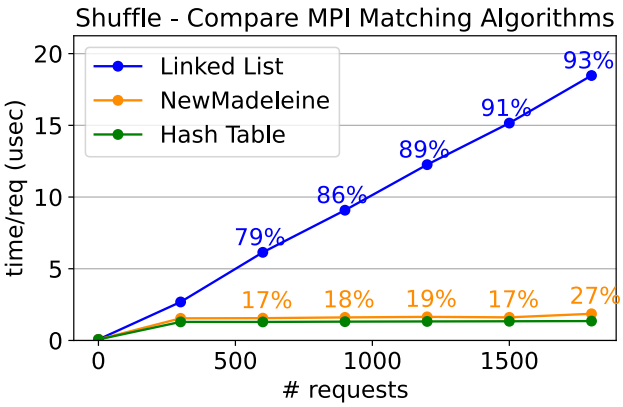


Fig. 3: Shuffle Benchmark - performance improvements of the hint-driven algorithm over MPC-MPI's and NewMadeleine's approaches. (The number above the graphs indicates the performance gain of our approach over the two other algorithms)

the hash-based algorithm performs significantly better than the MPC-MPI's approach. The theoretical complexity of both approaches can explain these results: a constant time algorithm always performs better on a large scale than a linear one. We also can notice that our algorithm and NewMadeleine's algorithm maintain a constant pace.

C. Applications

Our approach showed encouraging performance improvements on the *shuffle* benchmark. However, it is pretty challenging to reach such a large number of simultaneous requests while executing real HPC applications. Consequently, this section presents results for some HPC proxy applications. The results are depicted on Figure 4.

1) Quicksilver: Quicksilver [19] is a proxy application that approximates the overall application performance of Mercury [20]. That uses the Monte Carlo Method to solve particle transport problems. Its primary goal was to facilitate the refactoring of Mercury. It has become an essential benchmark for the HPC community lately, as it reproduces realistic high-performance patterns.

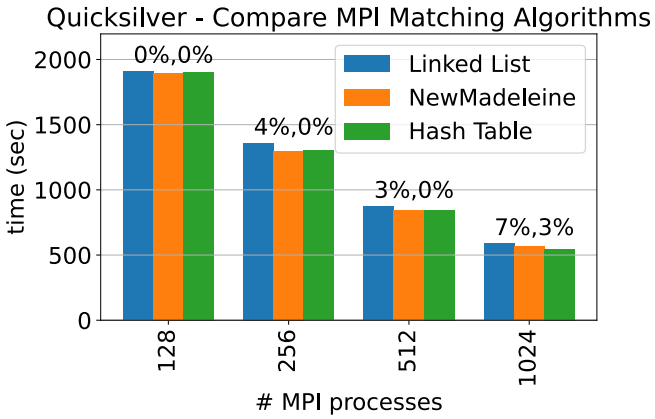
We evaluated our approach on several configurations based on examples from Quicksilver's repository [3]. We observed that the hint-driven algorithm performs at least as well as the linked list approach for all tested inputs. Figure 4a shows results for the *homogeneousProblem_v7_ts.inp* input, which was run with 128, 256, 512, and 1024 MPI processes. The X-axis indicates the number of MPI processes, the Y-axis shows the total execution time of the application with the corresponding algorithm, and the number above the bars presents the performance improvements of our approach over MPC-MPI's (left) and NewMadeleine's (right) algorithms. The presented graph shows encouraging results, especially for 1024 processes.

[2] Available at https://pm2.gitlabpages.inria.fr/releases

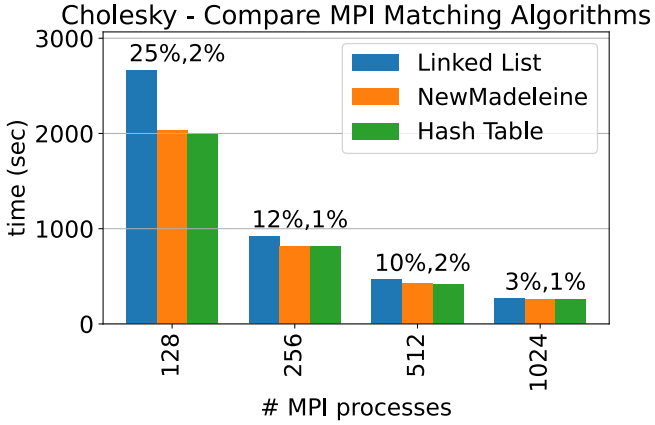
[3] Available at https://github.com/LLNL/Quicksilver/tree/master/Examples

		Matching paths			
		Fail from recv	**Sucess from recv**	**Fail from send**	**Sucess from send**
Operations	**Our algorithm**	1 push back + 1 lookup	1 delete list + 1 lookup	1 push back + 1 lookup	1 delete list + 1 lookup
	NewMadeleine algorithm	1 push back + 1 lookup	4 delete list + 1 lookup	4 push back + 1 lookup	1 delete list + 1 lookup
Time per request (us)	**Our algorithm [HT]**	0.44	0.30	0.44	0.30
	NewMadeleine algorithm [NM]	0.44	0.42	1.25	0.30
	HT execution time improvement (%)	0%	29%	65%	0%

TABLE I: NewMadeleine algorithm vs single hash-table-based algorithm



(a) Quicksilver



(b) Cholesky Factorization

Fig. 4: Proxy HPC Applications - performance improvements of the hint-driven algorithm over MPC-MPI's and New-Madeleine's approaches.(The two numbers above the bars indicate, in order, the gain of our approach gain over MPC-MPI's and New-Madeleine's algorithms)

2) Tiled Cholesky Factorization: In linear algebra, Cholesky factorization is a decomposition of a symmetric positive definite matrix A into the product of a lower matrix L and its transpose L^T (i.e. $A = LL^T$). The matrix is divided into square blocks for the tiled version of this algorithm.

We used a task version of the Tiled Cholesky Factorization previously introduced in [21]. In this application, MPI communications are finely integrated into OpenMP tasks to enhance the asynchronous execution of the application.

Thus, we could run this application entirely with the Multi-Processor Computing runtime.

The Cholesky Factorization algorithm exclusively relies on point-to-point operations triggered by the OpenMP tasks. As the tasks are scheduled in an unpredictable order, receive and send requests are also posted in a shuffled order. Thus, the application presents an irregular communication scheme similar to the *shuffle* benchmark.

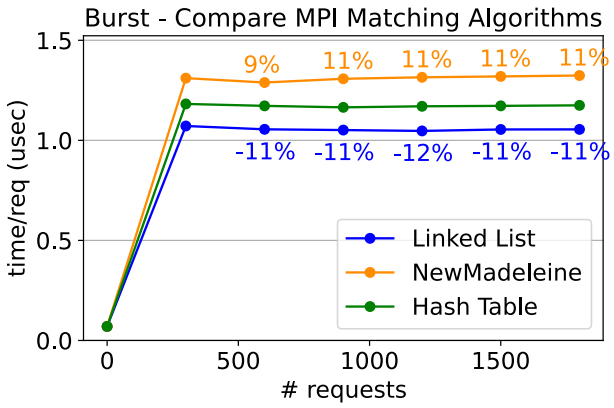
We set the size of the matrix to 131072 and the block size to 256 and performed a strong scaling with MPI processes going exponentially from 128 (1 node) to 1024 (8 nodes). Figure 4b presents the obtained results. As for the Quicksilver graph, the X and Y axes indicate the number of MPI processes and the total execution time. Again, the number above the bars highlights the performance improvements. We observe significant performance improvements principally for runs with a smaller number of MPI processes.

D. Overhead

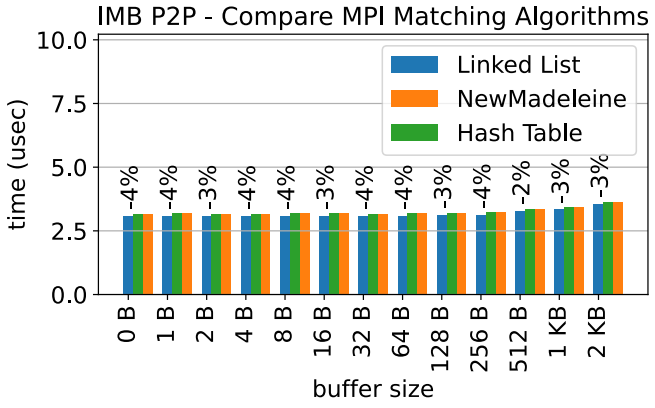
Generally, a lookup in a hash table is less efficient than checking the head of a list (i.e. dereference a pointer). Therefore, we propose to evaluate the overhead of our approach in a context favorable for a linked list implementation: cases where the searched message always corresponds to the head of the list. For this purpose, we use the *burst* benchmark from the MadMPI benchmark suite. It is similar to the *shuffle* benchmark, except that requests with the same tags are posted in the same order. In this configuration, a hash table has a considerable disadvantage.

Figure 5a shows a significant overhead of our approach. The X-axis and Y-axis are identical to the *shuffle* benchmark. The number above and below the graph allows to compare our approach to other matching algorithms. We can notice significant performance improvements over NewMadeleine's approach, but an overhead compared to MPC-MPI's approach.

The goal of the presented results was to see how our approach performs in extreme cases. However, as for the



(a) Burst



(b) IMB PingPong

Fig. 5: Matching algorithm overhead when using a hash table data structure compared to a linked list. (Negative numbers indicate the overhead of our approach over the MPC-MPI's algorithm)

shuffle benchmark, it is almost impossible to have such a situation for real HPC applications. We performed a ping-pong on two different nodes to obtain more realistic results by using the well-known Intel MPI Benchmarks (IMB) [22]. This operation shows the impact of the matching algorithm on communication latency. Figure 5b illustrates the obtained results. The X-axis presents the buffer size, and the Y-axis shows the execution time of an IMB ping-pong operation. The number above each pair of bars indicates the overhead of the hint-driven algorithm compared to a linked list approach. We can observe an insignificant slowdown for the smaller buffer sizes. However, it becomes irrelevant in a real HPC context. We observed a similar overhead for the NewMadeleine algorithm.

We also performed several experiments on some IMB collectives and NAS benchmarks [23] and did notice no overhead.

V. Results verification under AI model

This section presents a solution for identifying the most suitable matching algorithm for a given application. Indeed, the presented results confirmed that no universal solution performs well for all situations. Therefore, we study different matching situations by instrumenting a matching profiler and some micro-benchmarks. We propose a machine learning solution for classifying the MPI matching algorithms. The training data described in this section was built on an another partition of the *Inti* cluster. The last one contains 2 Intel(R) Xeon(R) Platinum 8168 CPUs and ConnectX-4 InfiniBand controllers.

We focus exclusively on the hint-driven and linked-list-based algorithms. In fact, we could not obtain data points where the NewMadeleine's approach performs better, as our solution presents an optimized version of a hash-table-based algorithm. However, our classification models can be extended to other specialized matching algorithms.

A. Matching Profiler

First, we implemented a profiler for gathering data related to the linked-list-based matching algorithm. It collects the total number of attempts for a message matching. All attempts are grouped into two categories: *successful ones*, when a matching message is found, and *unsuccessful ones* when the tail of the list is reached without finding any match. The profiler also provides the number of traversed cells before a success or failure attempt. Failure attempts allow us to know how the list size changes during the execution, and successful attempts allow us to identify matching intensive phases. The last one is a good indicator for detecting the impact of the matching overhead. The matching overhead becomes more critical when the number of traversed cells increases. In this context, a hash-table-based algorithm would have an obvious advantage over a linked list approach.

B. Matching micro-benchmarks

Then, we implemented a simple ping-pong benchmark to highlight the behavior of matching algorithms. Before starting the ping-pong operation, we post N requests into the receive queue of each process. Thus, each process that uses a linked-list-based matching algorithm traverses exactly $N+1$ cells before finding the matching message. This benchmark aims to highlight situations where the hash-table-based algorithm could bring some benefits. Also, it can identify a threshold above which it is not suitable to use a linked-list-based approach.

We executed our ping-pong benchmark with both matching algorithms and measured the execution time. Figure 6 presents obtained results with buffer sizes (X-axis) going from 1 B to 8 KB and for avg_traversed_cells_success (Y-axis) going from 0 to 200. The last one corresponds to the average number of traversed cells before finding a match. Because of the benchmark construction, the two MPI processes traverse the same number of cells. The color map indicates cases where our algorithm implementation performs better. We can notice

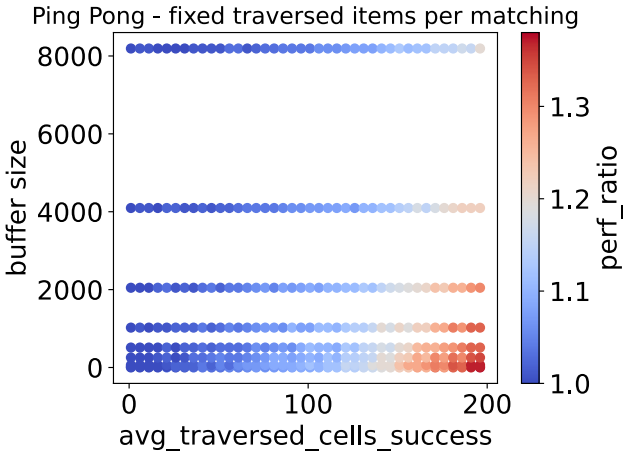


Fig. 6: PingPong Benchmark - analyzing the benefits of using a hash-table-based approach for a ping-pong operation, depending on the buffer size and the average traversed cells success.

two important tendencies from our results: first, the hash-based algorithm offers significant performance improvements when several requests are in the queue; second, this advantage becomes insignificant for larger buffer sizes since the latency becomes less than the time needed for transferring a large buffer.

C. Binary Classification

Based on observations from previous sections, we decided to instrument a machine learning model for predicting which algorithm is the most suitable for a given application. Indeed, our goal is to identify situations where our hash-based approach can bring significant performance improvements.

For this purpose, we have built a dataset based on our ping-pong implementation and some NAS and IMB benchmarks. We collected data on the average number of traversed cells before a match, buffer sizes and execution time for each configuration. Then we set up three simple models: a logistic regression, a support vector machine (SVM), and a multilayer perceptron (MLP). As all these models have a fixed input size we need to adapt our collected data. The avg_traversed_cells_success metric is computed individually for each MPI process, and obviously, the number of processes can change for different runs.

Consequently, we decided to compose the input of our models from statics data on the avg_traversed_cells_success metric and buffer sizes. The input comprises the following items for both metrics: mean, standard deviation, variance, and some quartiles. The output is composed of two classes corresponding to our two matching algorithms. We consider the hash-based algorithm performing better if it decreases the total execution time by at least 5%.

We took care to build a balanced dataset and split it into a training and a testing part. All three models showed encouraging results on the benchmark-based testing data, obtaining an accuracy between 99% and 100%. Also, we tested the MLP model (i.e. trained exclusively on micro-benchmarks) on some Cholesky and Quicksilver samples. We obtained satisfactory results for the Cholesky application: an accuracy of 75% and a well-balanced *confusion matrix*. However, our model could not provide the same performance on Quicksilver as we got several numbers of *false negatives* for a 60% of accuracy. Indeed, Quicksilver has a specific communication pattern that changes during the execution, and it is difficult to reproduce its behavior only with micro-benchmarks. We integrated the Cholesky samples into the training dataset to improve our model. Thereby we achieved an 87% accuracy on the Quicksilver dataset, but still with a dominant number of *false negatives*. We believe our model can be improved by adding more significant data related to real HPC communication patterns.

VI. Conclusion

This paper studies possible MPI matching benefits in a no *any_source*, no *any_tag* context. Our goal is to raise users' awareness about the importance of providing helpful information to the communication library.

We proposed a hint-driven constant-time matching algorithm and integrated it into the MPC-MPI library thanks to the new MPI-4.0 Sessions. We showed how it could improve the overall execution time of some real HPC applications by up to 25%. Indeed, we proved the effectiveness of our approach in a real implementation of MPI Sessions and realistic scenarios.

Also, we tested our algorithm in some extreme cases (i.e. in the presence of many simultaneous requests) and highlighted its weak points. We have shown that no universal algorithm is suitable for all applications and we have proposed integrating several matching implementations into the communication libraries. Hence, a user could choose the most suitable one for its application. To assist the user with his choice, we have implemented a matching profiler and built some classical machine learning models, which have shown encouraging results.

One notable contribution of this paper is that we focus our analysis on the matching algorithms rather than different communication libraries. Indeed, all experiments were performed using the same communication library but with different matching implementations. Thus, we brought out possible performance improvements which are exclusively due to the matching algorithms.

In the presented work, we addressed the performance of three matching algorithms, but several other implementations exist. Therefore, this work should be extended to cover all possible configurations and matching algorithms. Also, we would like to improve our solution for identifying the most efficient algorithm. Indeed, we believe that adding more proxy applications with different communication patterns can significantly enhance our approach.

ACKNOWLEDGMENT

Present results have been obtained within the frame of DIGIT, Contractual Research Laboratory between CEA, CEA/DAM-Île de France center, and the URCA.

REFERENCES

[1] K. Ferreira, R. E. Grant, M. J. Levenhagen, S. Levy, and T. Groves, "Hardware MPI message matching: Insights into MPI matching behavior to inform design," *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, p. e5150, 2020. e5150 cpe.5150.

[2] M. G. F. Dosanjh, W. Schonbein, R. E. Grant, P. G. Bridges, S. M. Gazimirsaeed, and A. Afsahi, "Fuzzy Matching: Hardware Accelerated MPI Communication Middleware," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 210–220, 2019.

[3] R. Brightwell, K. Pedretti, and K. Ferreira, "Instrumentation and Analysis of MPI Queue Times on the SeaStar High-Performance Network," in *2008 Proceedings of 17th International Conference on Computer Communications and Networks*, pp. 1–7, 2008.

[4] M. Moraru, A. Roussel, M. Pérache, H. Taboada, C. Jaillet, and M. Krajecki, "Benefits of MPI Sessions for GPU MPI applications," in *EuroMPI '21 - 28th European MPI Users' Group Meeting*, (Leibniz, Germany), Sept. 2021.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, (Budapest, Hungary), pp. 97–104, September 2004.

[6] 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), pp. 480–491, Springer International Publishing, 2018.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[8] K. S. Hemmert, K. D. Underwood, and A. Rodrigues, "An architecture to perform NIC based MPI matching," in *2007 IEEE International Conference on Cluster Computing*, pp. 211–221, 2007.

[9] Q. Xiong, A. Skjellum, and M. C. Herbordt, "Accelerating MPI Message Matching through FPGA Offload," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 191–1914, 2018.

[10] S. Derradji, T. Palfer-Sollier, J. Panziera, A. Poudes, and F. W. Atos, "The BXI Interconnect Architecture," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pp. 18–25, Aug 2015.

[11] W. P. Marts, M. G. F. Dosanjh, W. Schonbein, R. E. Grant, and P. G. Bridges, "MPI Tag Matching Performance on ConnectX and ARM," EuroMPI '19, (New York, NY, USA), Association for Computing Machinery, 2019.

[12] S. M. Ghazimirsaeed and A. Afsahi, "Accelerating MPI Message Matching by a Data Clustering Strategy," 2017.

[13] A. Denis, "Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 371–380, 2019.

[14] L. L. N. Laboratory, "CORAL-2 benchmarks." https://asc.llnl.gov/coral-2-benchmarks. 2022.

[15] 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)*, pp. 477–484, 2014.

[16] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A Unified Parallel Runtime for Clusters of NUMA Machines," in *Euro-Par 2008 – Parallel Processing* (E. Luque, T. Margalef, and D. Benítez, eds.), (Berlin, Heidelberg), pp. 78–88, Springer Berlin Heidelberg, 2008.

[17] M. Pérache, P. Carribault, and H. Jourdren, "MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (M. Ropo, J. Westerholm, and J. Dongarra, eds.), (Berlin, Heidelberg), pp. 94–103, Springer Berlin Heidelberg, 2009.

[18] R. Buyya, T. Cortes, and H. Jin, *An Introduction to the InfiniBand Architecture*, pp. 616–632. 2002.

[19] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, and M. J. O'Brien, "Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 866–873, 2017.

[20] "Mercury web site." https://wci.llnl.gov/simulation/computer-codes/mercury. 2022.

[21] J. Schuchart, K. Tsugane, J. Gracia, and M. Sato, "The Impact of Taskyield on the Design of Tasks Communicating Through MPI," in *Evolving OpenMP for Evolving Architectures* (B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, eds.), (Cham), pp. 3–17, Springer International Publishing, 2018.

[22] I. Corporation, "Intel MPI Benchmarks." https://software.intel.com/content/www/us/en/develop/documentation/imb-user-guide/top.html. 2022.

[23] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The nas parallel benchmarks," *Int. J. High Perform. Comput. Appl.*, vol. 5, p. 63–73, sep 1991.