HAL
open science

One core dedicated to MPI nonblocking communication progression? A model to assess whether it is worth it

Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Florian Reynier

To cite this version:

Alexandre Denis, Julien Jaeger, Emmanuel Jeannot, Florian Reynier. One core dedicated to MPI nonblocking communication progression? A model to assess whether it is worth it. International Symposium on Cluster, Cloud and Internet Computing (CCGRID), May 2022, Taormina, Italy. hal-03695835

HAL Id: hal-03695835

https://inria.hal.science/hal-03695835v1

Submitted on 15 Jun 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

One core dedicated to MPI nonblocking communication progression? A model to assess whether it is worth it.

Alexandre DENIS*, Julien JAEGER†‡, Emmanuel JEANNOT*, Florian REYNIER*†
*Inria Bordeaux – Sud-Ouest, France
†CEA, DAM, DIF, F-91927, Arpajon, France ‡LIHPC, Université Paris-Saclay, CEA, 91680, Bruyères-le-châtel, France

***Abstract*—Overlapping communications with computation is an efficient way to amortize the cost of communications of an HPC application. To do so, it is possible to utilize MPI nonblocking primitives so that communications run in background alongside computation. However, these mechanisms rely on communications actually making progress in the background, which may not be true for all MPI libraries. Some MPI libraries leverage a core dedicated to communications to ensure communication progression. However, taking a core away from the application for such purpose may have a negative impact on the overall execution time. It may be difficult to know when such dedicated core is actually helpful.**

In this paper, we propose a model for the performance of applications using MPI nonblocking primitives running on top of an MPI library with a dedicated core for communications. This model is used to understand the compromise between computation slowdown due to the communication core not being available for computation, and the communication speed-up thanks to the dedicated core; evaluate whether nonblocking communication is actually obtaining the expected performance in the context of the given application; predict the performance of a given application if ran with a dedicated core.

We describe the performance model and evaluate it on different applications. We compare the predictions of the model with actual executions.

I. INTRODUCTION

On the path to exascale, the cost of communications is an obstacle for scalability of HPC applications. Overlapping communications with computation is an efficient way to amortize this cost: when performing communication at the same time as computation, their cost is close to zero. To do so, applications have to utilize MPI nonblocking operations so that communications run in background alongside computation.

However, these mechanisms rely on communications actually making progress in the background, which may not be true for all MPI libraries [1]. These mechanisms may be present or not, adequate or perfectible, they may have an impact on communication performance or may interfere with computation by stealing CPU cycles.

With the increasing number of cores per node the question of dedicating a core for ensuring a steady progression of communications is raised. Indeed, it could brings good overlap and good overall communication performance, at a (negligible?) expense of one CPU core not usable for computation.

Nonetheless, communication/computation overlap requires explicit support from the application: algorithms have to be adapted to be able to perform computation at the same time as communications, and it may require drastic changes in the way the application manages its communication. However, HPC applications are usually huge codes, that may have been developed and maintained for decades. Their developers often prefer to use the blocking operations and are reluctant to use the nonblocking operations because they are notoriously hard to use [2], they were not taught to use them, and they are known not to overlap so well. Thus they need a tool to evaluate whether it is worth investing time in converting existing applications to nonblocking communications.

In this paper, we propose a model for the performance of hybrid (MPI+X) applications, running with one MPI process per node and a shared-memory parallel paradigm (e.g. OpenMP) using all compute resources on each node, and using MPI nonblocking operations on top of an MPI library with a dedicated core for communications. In short, the contribution of this paper is a model used to:

- understand the compromise between computation slowdown due to the communication core not being available for computation, and the communication speedup thanks to the dedicated core;
- evaluate whether nonblocking communication is actually obtaining the expected performance in the context of the given application;
- predict the maximum performance of a given application if run with a dedicated core;
- predict the maximum performance we would get if a given amount of the application communications would be converted to nonblocking operations.

The rest of this paper is organized as follows. Section II gives an overview of the use of a core dedicated to communication progression. Section III presents our generic performance model for applications of a core dedicated to communications. Section IV explains how to exploit the model in practice. Section V evaluates the model with real applications, and Section VI gives some examples of what the model may be used for. We conclude in Section VII.

II. CONTEXT AND RELATED WORKS

To ensure overlap of communications with computation, HPC application programmers use MPI nonblocking primitives to let communication operations perform in background

while the computation is running on the CPU. However, the MPI specification [3] only guarantees that these primitives will not block; it does not ensure that progression will actually happen in background. The actual behavior depends on the implementation of the MPI library, but generally, progression is poor [1]. If no explicit mechanism for progression is implemented in the MPI library, then progression only occurs inside the calls to the MPI library. Therefore, if a user calls `MPI_Isend`, then performs some computation, and finally calls `MPI_Wait` to check for completion, chances are high that communication will actually *begin* in the `MPI_Wait`, which prevents any tentative of communication and computation overlap.

To ensure actual background progression, it has been proposed to use multi-threaded communications [4], to use a dedicated core [5], [6], [7], to use special tasks or threads for polling in the runtime [8], [9], [10], [11] to improve background progression of communications. Specific work has been done in some applications [2], [12], [13] to utilize nonblocking communications so as to get overlap. The aforementioned solutions are implemented either in the application, in another runtime, or in an MPI library. Our work aims at predicting the performance we would get after such a transformation would be done on the application code or if the MPI library features a progress thread.

In this paper, we use progression in the MPI library itself so as not to modify the applications. The choice for a freely available MPI library with such feature is currently narrow. OpenMPI had an experimental `--enable-progress-thread` configure option a long time ago, that has been dropped in version 1.4. Nowadays, OpenMPI does not allow to have a progress thread in the library. The `MPICH_ASYNC_PROGRESS` feature of MPICH allows to have a progress thread, but does not allow to bind the thread on a dedicated core. A very recent version of MPICH offers this feature, but unfortunately we were unable to test it on our cluster. For MVAPICH, section 5.6 of the user guide of MVAPICH2-X states that its progression engine does not need a dedicated core. The introduction of an additional thread beside a multi-threaded runtime usually causes some performance degradation due to conflicts in thread scheduling. We have shown [14] this degradation happens even with a spare core in case the progress thread is not bound. Thus in this paper we use MadMPI [15] which allows using a progress thread and pinning it to a dedicated core.

With the increase of the number of cores per socket, dedicating a core for progression is an elegant solution to address the issue of progression. However such a dedicated core leads to a trade-off. On the one hand, this solution removes a core from the application. One obvious effect is the whole computation of the application having to be executed on $n-1$ cores per compute node. This may induce some overhead, which depends on the number of cores. On the other hand, the MPI runtime may use its dedicated core for communication progression at any time; it is especially important since it has been shown in our previous work [1] that what hinders progression is not a large amount of work to run on a CPU, but instead very small tasks to be scheduled when needed. Having a full dedicated core ensures that the MPI library will be able to schedule these tasks at any time, whenever needed, and should guarantee that progress is maximized. Hence, the goal of this paper is to model this trade-off to assess whether a dedicated core for progression is useful or harmful. This is of paramount importance as the number of cores per socket is expected to continue to increase.

Previous work exists to estimate the gain we would obtain from overlapping communications and computation, which is the same question as ours. They either estimate [16] the potential overlap from an algorithmic point of view, or focus [17] on networks with offloading capabilities and non-threaded applications. However, this is different from hybrid MPI+OpenMP applications case that is the most common way [18] of writing MPI program today and hence, the focus of this study.

III. A MODEL FOR APPLICATION PERFORMANCE WITH A DEDICATED CORE FOR COMMUNICATION PROGRESSION

A. Overview of the performance model

To sort out cases where overlap with dedicated core is beneficial from cases where it is detrimental to performance, we propose a model to predict performance. This model takes into account two phenomena:

- the impact on computation performance of having one less core because the core dedicated to communications is not available for computation anymore. We will study this aspect in Section III-B;
- the impact on communication performance and on overlap of communication and computation of having a dedicated core. It will be described in Section III-C.

The first part about computation is expected to be a performance degradation. The second part, about overlap, is expected to be a performance improvement; it may be negligible if communication may not be overlapable, or if communications constitute a small part of the total application execution time. Then, the computation slowdown may overcome the communication speedup. For this reason, the dedicated core cannot just be an "enabled and forget" feature. The actual challenge is to know whether the degradation is compensated by the gain. To be used in the relevant situations, we must be able to predict its behavior.

Unfortunately, the behavior of HPC applications diverge greatly with regard to computation and communication. Even on the same application, different inputs (parameters, data, etc.) may lead to different performance behavior. Worse, two set of parameters can seem to be similar, showing very close computation time when run without dedicated core. However, with the use of dedicated core, only one set of parameters will give a significant time gain. This is due to the input parameters, which may drastically change communications/computation scheme and proportions.

To cover these cases and be oblivious to input parameters, we will design our model to be *independent from the application*. Hence, the model is the same for all applications, but, of course, the calibration of the model is application dependent. To do so, our model decomposes an application as a sum of specialized parts, each with a different behavior when a dedicated core for communication progression is involved. These parts will be presented in Section III-B and Section III-C.

Thanks to our model, we aim at predicting if the use of a dedicated core for progression is beneficial for an application. To apply our model, we will first run the application on all cores without a progression thread to get a reference time $t_{\text{noprogress}}$. Details on how the exact timings are measured on the different MPI ranks is given in Section IV. This $t_{\text{noprogress}}$ is composed of:

- t_{comp} the time spent by the application in computation;
- t_{MPI} the time spent in the MPI library. This time is itself composed of $t_{\text{MPIoverlapable}}$ for communications that may benefit from overlap (nonblocking point-to-point and collective primitives) and progress in background, and $t_{\text{MPInotoverlapable}}$ for the remaining communications and all MPI management time.

This distinction will help us model and analyze precisely the behavior of applications with and without dedicated core. It is clear that the expected speedup from the use of a dedicated core will be limited to the time spent in the MPI library.

Let $t_{\text{dedicated}}$ be the execution time of the application with a dedicated core for communications. It will diverge from $t_{\text{noprogress}}$ as follows:

- on the t_{comp} part, computation will be slowed down by an overhead t_{overhead};
- on the t_{MPI} part, we expect the nonblocking communication primitives to be fully overlapped and thus the gain should be $t_{\text{MPIoverlapable}}$.

Thus, the general formulation for our model is:

$$t_{\text{dedicated}} = t_{\text{noprogress}} + t_{\text{overhead}} - t_{\text{MPIoverlapable}}$$

This formulation helps to get an intuitive idea of the model. However, instead of computing t_{overhead} and $t_{\text{MPIoverlapable}}$, in the later Sections we will decompose t_{comp} and t_{MPI}.

B. Impact of a dedicated core on computation

The evolution of modern processors exhibits an increase in the number of cores. With more and more cores, it becomes more difficult to optimize applications and fully exploit those CPUs. Thus stealing a core to an application becomes less impacting for computation performances.

1) Study of OpenMP applications scalability: To quantify the loss caused by stealing a core to the application, we ran a study on the scalability of OpenMP threading for several well-known benchmarks:

- from the NAS Parallel benchmarks [19]:
  - BT-MZ: a Block Tri-diagonal solver app;
  - LU-MZ: a Lower-Upper Gauss-Seidel solver app;
  - SP-MZ: a Scalar Penta-diagonal solver app;
- and from the CORAL Benchmarks:
  - MiniMD: a simple parallel molecular dynamics (MD) code [20];
  - MiniFE: a proxy application for unstructured implicit finite element codes [20];
  - Lulesh: the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics [21];
  - Kripke: a simple scalable 3D Sn deterministic particle transport code [22].

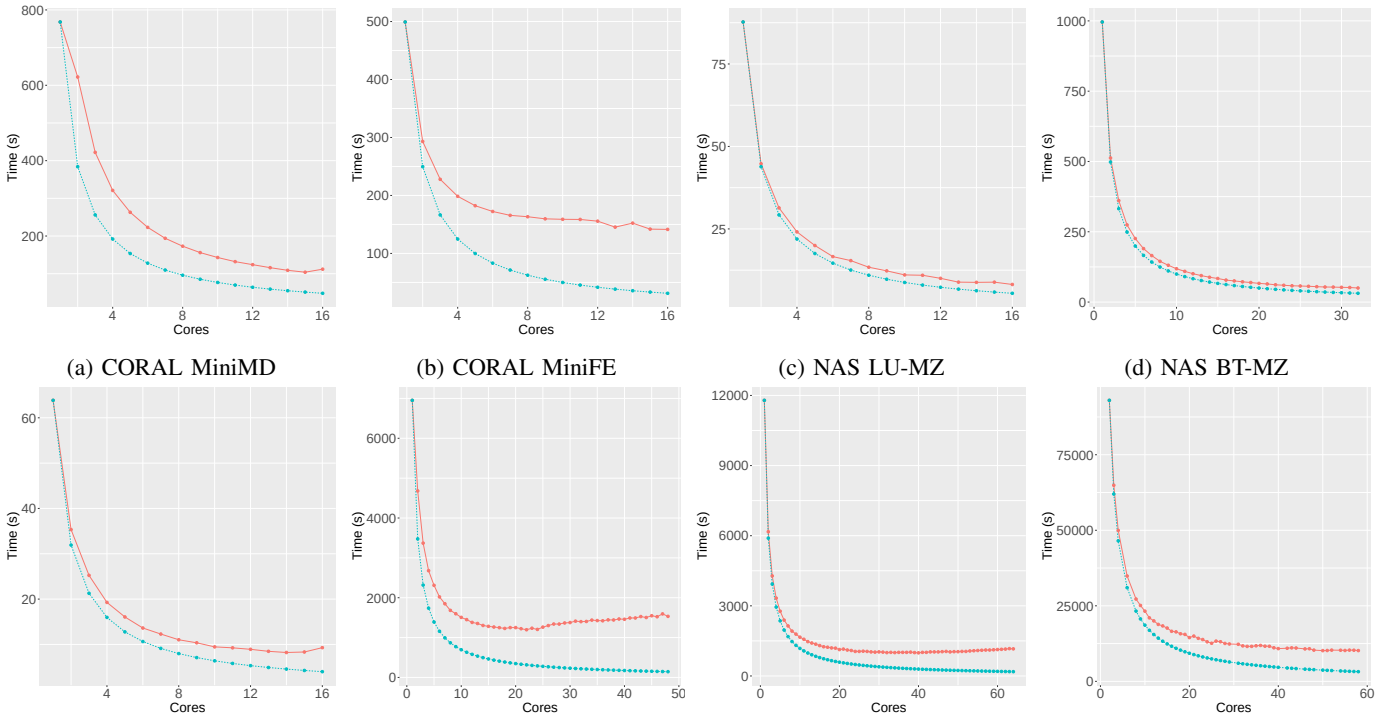
We tested those benchmarks on several test platforms:

- *inti/sandy-bridge*: 234 nodes, with dual-socket Intel Xeon E5-2680, each with 8 cores at 2.7 GHz, equipped with Mellanox QDR Infiniband (used for the LU-MZ, SP-MZ, MiniFE and MiniMD benchmarks),
- *inti/haswell*: 8 nodes, with dual-socket Intel Xeon E5-2698, each with 16 cores at 2.3 GHz, equipped with Mellanox MT27600 (Connect-IB) InfiniBand boards (used for BT-MZ benchmark),
- *inti/skylake*: 32 nodes, with dual-socket Intel Xeon Platinum 8168, each with 24 cores at 2.7 GHz, equipped with Mellanox MT27700 (Connect-IB) InfiniBand boards (used for the Lulesh benchmark),
- *inti/KNL*: 24 nodes, with Intel Xeon Phi 7250, each with 68 cores at 1.4 GHz, equipped with Mellanox EDR InfiniBand boards (used for the Kripke benchmark).

OpenMP scalability results for all benchmarks are displayed in Figure 1. From these graphs, we observe two types of behavior. On the one hand, benchmarks on the top row show the expected behavior: the execution time follows a $1/(\delta \times N_{\text{core}})$ slope (with δ often very close to 1). On the other hand, the first three benchmarks of the bottom row show bad OpenMP scalability, as they all lose performance when the number of threads is greater than a given threshold (specific to each application). We can see on some benchmarks (LU-MZ and Lulesh) that the last point of the curve shows a greater speedup than the previous points. However, since those points are outliers, we consider they do not impact the general slope of the curve.

One can observe that the Kripke benchmark displays the behavior we described in Section III-A: two set of input parameters cause different OpenMP scalability. When executed with small sizes (i.e. x=16 , y=8 and z=8, in Figure 1g), the bad OpenMP scalability causes a slowdown instead of a speedup when large number of threads are used. However, when executed with large sizes (i.e. x=400, y=200, z=200, in Figure 1h), we observe speedup even with a large number of threads, and the scalability follows the $1/N_{\text{core}}$ slope.

We consider that taking a core from the parallel computation will cause slowdown induced by the loss of computational power. As we have seen, if the OpenMP scalability is not good, taking a core from the computation will not cause a slowdown. It may even produce a speedup. Hence, since the impact on computation time is supposed to hinder the speedup gained thanks to the communication progression, we will consider the



(a) CORAL MiniMD (b) CORAL MiniFE (c) NAS LU-MZ (d) NAS BT-MZ

(e) NAS SP-MZ (f) CORAL Lulesh (g) CORAL Kripke - small size (h) CORAL Kripke - large size

Fig. 1: Study of OpenMP scalability on several well-known benchmarks (red plain line) against theoretical linear scaling (blue dotted line)

worst-case scenario for our model: good OpenMP scalability. This case is the one actually producing a slowdown which may prevent a dedicated core for progression to be effective.

As we have seen, most applications with a good OpenMP scalability exposes a $1/(\delta \times N_{\text{core}})$ slope (with δ very close to 1). Hence, in our model, we will approximate the computation overhead induced by the removal of one computational core with the linear equation $1/N_{\text{core}}$.

Though this simple approximation is sufficient for our model, as we will see in Section V, state-of-the-art on the scalability of OpenMP applications [23], [24] can be used to further improve the precision of our model.

2) Modeling manycore cpu impact on thread scalability: The overhead expectation t_{overhead} due to the core stealing impact can be modeled using the computation time (t_{comp}) from no progression execution. With our approximation, we estimate that the new computation time when removing 1 core among N_{core} is:

$$t_{\text{newcomp}} = t_{\text{comp}} \times \frac{N_{\text{core}}}{N_{\text{core}} - 1}$$

The induced slowdown is then computed with:

$$t_{\text{overhead}} = t_{\text{newcomp}} - t_{\text{comp}}$$

This equation can be simplified to:

$$t_{\text{overhead}} = \frac{t_{\text{comp}}}{N_{\text{core}} - 1}$$

For a 16-core node, taking a computational core will cause a $\frac{t_{\text{comp}}}{15}$ slowdown, and on a 64-core node, the slowdown will be $\frac{t_{\text{comp}}}{63}$. Thus, the more core a processor has, the more negligible the slowdown will be.

C. Modeling MPI performance with dedicated core

The second part of the model estimates the decrease of the time spent in MPI functions thanks to a dedicate core.

An MPI distributed application usually includes different MPI calls. These MPI calls will not display the same behavior when a core dedicated for background progression is used. To build an accurate model, we need to classify the MPI functions regarding their response to the use of said dedicated core.

1) Classification of MPI calls: For our model we distinguish four types of MPI calls:

- blocking communication calls
- nonblocking communication initialization calls
- nonblocking communication completion calls (e.g. `MPI_Test`, `MPI_Wait`)
- other MPI calls (e.g. runtime initialization, communicators management, datatypes management, etc.)

With this classification, we decompose the total time spent in MPI in the following categories:

$$t_{\text{MPI}} = t_{\text{MPIblocking}} + t_{\text{MPInonblocking}} + t_{\text{MPItest}} + t_{\text{MPIwait}} + t_{\text{MPIother}}$$

with the following definitions:

- $t_{\text{MPIblocking}}$ is the time spent in blocking communication calls;
- $t_{\text{MPInonblocking}}$ is the time spent in nonblocking initialization calls such as `MPI_Isend`, `MPI_Irecv` or `MPI_Iallgather`;
- $t_{\text{MPItest}} + t_{\text{MPIwait}}$, the time spent in active progression functions such as `MPI_Test` and `MPI_Wait`;
- t_{MPIother} is the remaining time spent in the MPI runtime (e.g. initialization, communicators management, datatypes management, etc.).

In the following Sections, we will describe how each category of MPI call will behave when a dedicated core is used for background progression.

2) Impact of dedicated core on each MPI call category:

a) Impact on nonblocking initialization and completion calls: As we presented in Section II, when the MPI library implements no progression mechanisms, then nonblocking communications only progress inside MPI calls. These calls can be nonblocking initialization calls (e.g. `MPI_Isend`, `MPI_Irecv`, `MPI_Iallgather`), completion calls (e.g. `MPI_Test`, `MPI_Wait`), or any MPI calls involved in communications (even blocking primitives).

With a dedicated core, nonblocking communications are expected to progress on said dedicated core, thus not consuming computational power from the other cores. However, even if the communication itself takes place on the dedicated core, the time taken by the actual involved MPI calls is not completely nullified: the requests still have to be initialized and the associated operation registered in the MPI runtime for initialization calls ; the status of the request has to be fetched, with the relevant synchronization, for completion calls.

We define $t_{\text{minMPInonblockinit}}$, $t_{\text{minMPItest}}$ and $t_{\text{minMPIwait}}$ to be the minimum time required to execute a nonblocking initialization call, a test call and a wait call respectively, without performing any progression.

The minimum time for nonblocking initialization calls and `MPI_Test` calls is easy to meet. When an asynchronous progression mechanism is involved, the initialization call job is to just fill in the request argument, then let the progression happen in background. So no extra time is taken for progression. For a `MPI_Test` call, its job is just to test if the associated operation is finished. If not, it will let the background mechanism to progress the operation, and will not take extra time to realize such progression.

However, a call to `MPI_Wait` actually has to *wait* until the operation is done. If the operation is not finished when the call is performed, then it will block until the operation completes. Even if the progression happens in background, it cannot be overlapped by computations and it will not be hidden. Our model aims at predicting if an application will benefit from the use of nonblocking communications with a dedicated core for asynchronous progress. Because of the semantics of the `MPI_Wait` procedure, nonblocking communications by themselves are useful only if there is enough computation to hide the communications. Hence, in our model, we consider the best-case scenario for using nonblocking communications. We assume that, when transformed to use said nonblocking communications, the application exhibits enough computations between an initialization call and its corresponding completion call to completely overlap the communication time. In this case, when using a dedicated core for progression, the execution time for the `MPI_Wait` calls will always be the minimum time.

Thus, if we consider that the application embeds $N_{\text{nonblocking}}$ nonblocking initialization calls, N_{test} `MPI_Test` calls and N_{wait} `MPI_Wait` calls, we model the time for all calls involved in nonblocking communications to be:

$$t_{\text{MPInonblockdedicated}} = N_{\text{nonblocking}} \times t_{\text{minMPInonblock}} + N_{\text{test}} \times t_{\text{minMPItest}} + N_{\text{wait}} \times t_{\text{minMPIwait}}$$

b) Impact on blocking calls: We have seen so far how the nonblocking calls already in the application will behave with a dedicated core. We will now see what would happen if blocking calls would be changed into nonblocking calls to benefit from the use of a dedicated core.

Due to algorithmic constraints, it is expected that even with a large refactoring of the application, not all blocking calls may be changed to nonblocking with overlap. Thus blocking communications in the original application will either remain blocking communications, or be changed in their nonblocking counterparts (hence adding the necessary initialization and completion calls). Both cases will behave differently when a dedicated core is used.

For the first case, one can think that it is trivial, as dedicated core is used to progress nonblocking calls and not blocking calls. This is wrong. As we said in the previous part, blocking calls may help progress nonblocking communications. We decided to decompose the time of a blocking call $t_{\text{MPIblocking}}$ in the time of the actual blocking communication $t_{\text{MPIblockingcom}}$ and the time spend to progress pending nonblocking communications $t_{\text{MPIblockprogress}}$:

$$t_{\text{MPIblocking}} = t_{\text{MPIblockingcom}} + t_{\text{MPIblockprogress}}$$

If a dedicated core is used for progress, then blocking calls will not progress nonblocking communications anymore. Thus, $t_{\text{MPIblockprogress}}$ becomes null, and we have:

$$t_{\text{MPIblockingdedicated}} = t_{\text{MPIblockingcom}}$$

For the second case, as the blocking communication is transformed in a nonblocking communication, its time becomes similar to nonblocking communications. The most direct way to change a blocking communication to its nonblocking counterpart is to call the corresponding initialization call, then `MPI_Wait` as its completion call. Thus, the new time when using a dedicated core for progression is: $t_{\text{MPIblockingtransformed}} = t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}$.

Let us consider an application with N_{blocking} MPI blocking communications, and that a ratio α of these blocking commu-

nications are transformed in their nonblocking counterparts, the new time for these communications are:

$$t_{\text{MPIoldblocking}} = \alpha \times N_{\text{blocking}} \times t_{\text{MPIblockingtransformed}} + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingdedicated}}$$

which can be developed as:

$$t_{\text{MPIoldblocking}} = \alpha \times N_{\text{blocking}} \times (t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}) + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingcom}}$$

c) Impact on the other MPI calls and full MPI model: The last type of MPI calls in our classification is *other*, which are the MPI calls that handle the library and internal structures management, but perform no communications and no progression. These calls are not impacted by the use of a dedicated core for progress, and the time t_{MPIother} remains the same.

Thus, when putting together all parts of the MPI model and considering that a ratio α of blocking calls will be changed between the initial version of the application and the version we want to model, the MPI time when using a dedicated core is:

$$t_{\text{MPIdedicated}} = t_{\text{MPInonblockdedicated}} + t_{\text{MPIoldblocking}} + t_{\text{MPIother}}$$

which can be developed in:

$$t_{\text{MPIdedicated}} = N_{\text{nonblocking}} \times t_{\text{minMPInonblock}} + N_{\text{test}} \times t_{\text{minMPItest}} + N_{\text{wait}} \times t_{\text{minMPIwait}} + \alpha \times N_{\text{blocking}} \times (t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}) + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingcom}} + t_{\text{MPIother}}$$

D. Global model with MPI and computation

The final model is the combination of the estimated computation slowdown modeled in Section III-B2, and the communication evolution modeled in Section III-C:

$$t_{\text{dedicated}} = t_{\text{newcomp}} + t_{\text{MPIdedicated}}$$

which can be developed in:

$$t_{\text{dedicated}} = t_{\text{comp}} \times \frac{N_{\text{core}}}{N_{\text{core}} - 1} + N_{\text{nonblocking}} \times t_{\text{minMPInonblock}} + N_{\text{test}} \times t_{\text{minMPItest}} + N_{\text{wait}} \times t_{\text{minMPIwait}} + \alpha \times N_{\text{blocking}} \times (t_{\text{minMPInonblock}} + t_{\text{minMPIwait}}) + (1-\alpha) \times N_{\text{blocking}} \times t_{\text{MPIblockingcom}} + t_{\text{MPIother}}$$

The good cases where the dedicated core brings some improvement in the execution time should have a gain on the MPI part (colored part in the model) higher than the overhead estimated in the computation part (black part in the model).

IV. Gathering applications data to use the model

The model is parameterized with timings for each part of the original application. These timings need to be measured on an execution of the original application. We used a tool that wraps MPI calls, called *mpiP* [25], which is a light-weight profiling library for MPI applications. It generates logs gathering time spent in each MPI functions, number of calls, message sizes with minimum, maximum and average value. Information are collected for each MPI rank. This tool is generic enough to gather information on any tested application. We have to approximate some terms of our model since profiling does not give directly the exact value we would need in the model.

First, we have to approximate the minimum time for nonblocking initialization calls, `MPI_Test` and `MPI_Wait` calls. To do so, we consider the minimum measured time of each type of call when executing the application. Even when no dedicated core is used for progress, completion calls may not perform progress. When a completion call is used on a request, the associated operation may already be done, because it has been progressed by other previous MPI calls (blocking calls, or completion calls for other operations). If there is no other pending operation, then the completion call will just check the status of the request, and its timing will be the minimum. We assume this case happens at least once in an MPI run for nonblocking initialization, `MPI_Test` and `MPI_Wait` calls. Thus we take the minimum measured time for each of these as the respective timings for $t_{\text{minMPInonblock}}$, $t_{\text{minMPItest}}$ and $t_{\text{minMPIwait}}$.

Second, we do not know which part of the time spend in a blocking call actually relates to the execution of the associated blocking operation, or if some of the measured time corresponds to progression of pending nonblocking operations. For this reason, we consider that the time measured for each blocking call is fully dedicated to the associated operation, and that $t_{\text{MPIblockprogress}}$ is always zero.

These approximations may cause some discrepancies between our model forecasts and measured runs, but we will show that although we have kept the model simple and made some approximations, it remains accurate enough for our purpose.

V. Evaluation of the model

In this section, we evaluate how close to reality the model we propose is, and check that it is good enough for the question we are trying to answer. We run the applications presented in Section III-B1 on various data sets and number of nodes; it is especially important to use various data sets since the behavior of some applications depends on inputs. Each run is performed in hybrid MPI+OpenMP mode, with one MPI process per node, and each compute resources of a node used by the OpenMP threads for computation.

As seen in Section II, OpenMPI or MPICH do not provide the feature we aim to model. Hence, we performed our tests with the MPI implementation *MadMPI*, as it allows activating and deactivating asynchronous progression and use of a dedicated core [15]. However, the model makes no asumption

about the MPI library and should work with any MPI library with a core dedicated to progression. The machines used are *inti/sandy-bridge* presented in Section III-B1. Though it provides a lesser number of cores that *inti/skylake*, hence taking a core away from OpeMP should cause a greater slowdown, this machine offers a greater number of nodes, which allows testing more realistic MPI configurations.

To evaluate the model, we compare its output to the real results obtained when running the applications with a dedicated core. This evaluation is limited to applications that already leverage nonblocking communications. Since we only run unmodified applications, in this section we have $\alpha = 0$.

Each experiment was run several times. The number of repetitions depends on the execution time of one run, with at least ten repetitions. We present the median value of these runs. Each combination of applications/parameters were run with two configurations. Both configurations were run with the same number of repetitions.

- First, we collect data to calibrate our model. We run the application without a dedicated core, and without any progression mechanisms for communications. This run is similar to what a user obtains with a regular MPI library without progression. We measure $t_{\text{noprogress}}$ and we gather all the input parameters needed by the MPI model as explained in Section III-C, with the help of *mpiP*. We used bash scripts to get the data necessary to compute our predicted t_{model} with an *R* script building our model presented in Section III-D.
- The second run uses one dedicated core mapped on the first logical core (e.g. core #0). We configure the OpenMP runtime to use $N_{\text{core}} - 1$ threads and they are bound on all other available cores (1 to $N_{\text{core}} - 1$). All progression mechanisms are enabled. We also use *mpiP* in this run to have the same measurement overhead. This execution gives us the real value for $t_{\text{dedicated}}$.

The closer t_{model} is to $t_{\text{dedicated}}$, the better the model is.

We run the model on three applications: BT-MZ from the NAS Parallel Benchmark, and Kripke and MiniMD from the CORAL benchmarks.

The results are shown in Figure 2 for Kripke, Figure 3 for NAS BT-MZ, and Figure 4 for MiniMD. On these figures, the x-axis corresponds to executions on various data sets, and various number of nodes for BT-MZ and MiniMD. The y-axis shows the performance represented as a speedup compared to the basic execution without dedicated core. For each configuration, we display two values. The red cross corresponds to the real execution with a dedicated core, defined as $s_{\text{dedicated}} = \frac{t_{\text{noprogress}}}{t_{\text{dedicated}}}$; the green bullet represents the performance predicted by the model, defined as $s_{\text{model}} = \frac{t_{\text{noprogress}}}{t_{\text{model}}}$. Thus, the closer the red cross is from the green bullet, the more accurate our model is.

Kripke was run with 52 different data sets, depicted in Figure 2. We distinguish two types of behavior: cases where the dedicated core brings a significant speedup (> 1.1), and

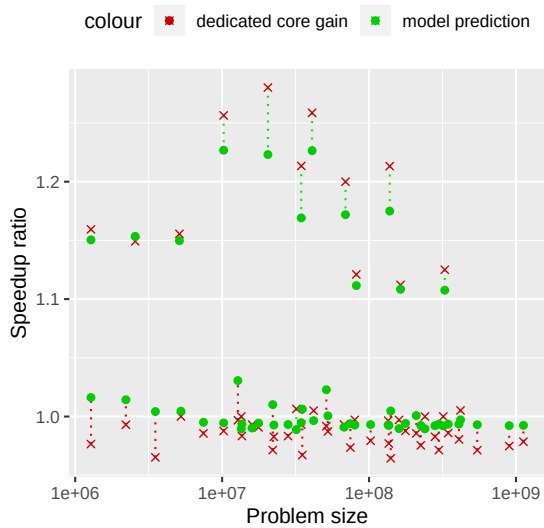


Fig. 2: Comparison of the model prediction and the dedicated core case (Speedup to the time with no progression) for the Kripke application for different problem sizes on *inti/sandy-bridge*.

cases where the dedicated core does not bring any benefit (speedup below or very close to 1). This difference is explained by the MPI behavior of the code. For each input problem size, the run does not use the same part of the code. In the beneficial cases, the run spends a lot of time in nonblocking communications, whereas for the other cases, it is negligible. Note that beneficial cases are not grouped together, so it may be hard for a user to know specifically which case is a good candidate for running with a dedicated core. We observe that our model accurately predicts the behavior of the application for each data set. For all cases where the model predicts a significant speedup, it is confirmed by the experimental execution. And, for all cases where the predicted speedup is close to 1, we see this exact behavior with the experimental run.

For NAS BT-MZ (Figure 3) and MiniMD (Figure 4), the results are less identical. On NAS BT-MZ, the results for class C and D on 16 nodes, the predictions from the model are correct — the dedicated core brings no gain. However, for class D on 128 nodes, the model predicts a speedup higher than 3 where the reality is a mere 0.96. This is due to the communication scheme. Our model makes the assumption that nonblocking operations are always overlapped by computation. Unfortunately, in this application the nonblocking operations are not designed to overlap communications and computation; they are used to overlap multiple communications. Indeed, the code calls a series of `MPI_Isend` and `MPI_Irecv` with a final `MPI_Waitall`, and no computation in between.

For MiniMD we observe that the first two predictions are correct and the third is much too optimistic . For the first two cases, the execution with a dedicated core is slower than without (see Fig 1a) and the model is accurate. For the last cases,

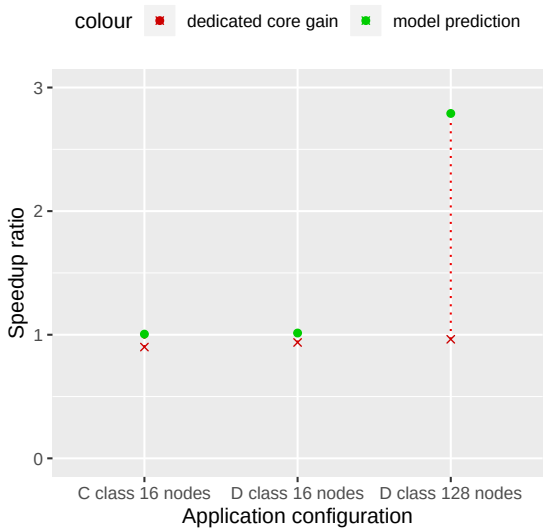


Fig. 3: Comparison of the model prediction and the dedicated core case (Speedup to the time with no progression) for NAS BT-MZ for different configurations on *inti/sandy-bridge*.

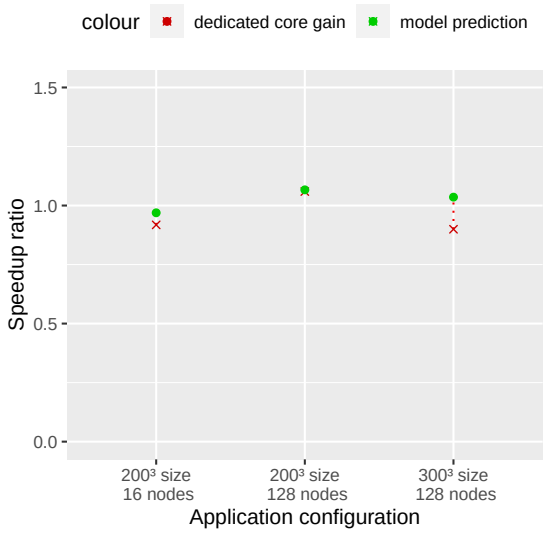


Fig. 4: Comparison of the model prediction and the dedicated core case (Speedup to the time with no progression) MiniMD for different configurations on *inti/sandy-bridge*.

when looking at the detailed values for all parameters of the model and compare them to the real execution, we observe an increased time spent in `MPI_Wait` with dedicated core (4,19 s with no progression against 8,69 s with dedicated core). This behavior is sometimes observed when there is an interaction between the dedicated core and the `MPI_Wait`: they try to progress simultaneously. In those cases, the progression is not just done by the dedicated core and hence the gain is less than expected by the model.

We have shown here that the proposed model successfully predicts performance of a hybrid MPI application using a core

	Case #1	Case #2
$t_{\text{noprogress}}$	147	155
t_{MPI}	73.4	8.3
t_{model}	119.82	156.75
$t_{\text{dedicated}}$	117	154
N_{core}	16	16
t_{comp}	73.6	145.7
$N_{\text{nonblocking}}$	480000	96
$t_{\text{minMPInonblock}}$	$2.14e-05$	$2.16e-05$
N_{test}	1119810	109679
$t_{\text{minMPItest}}$	$2.05e-05$	$2.2e-05$
N_{wait}	10000	2
$t_{\text{minMPIwait}}$	$3.2e-05$	$7.19e-05$
$t_{\text{MPIblocking}}$	11.7	4.98
t_{MPIother}	0.0	0.34

TABLE I: Values of model parameters for two sets of parameters of Kripke (times in seconds).

dedicated to communication when the hypotheses are fulfilled. It exhibits a very good precision in this case, as seen with the Kripke application.

However, the user should be wary of using it blindly. It may give wrong results when hypotheses are not fulfilled: when no computation is executed at the same time as nonblocking communications, like in NAS BT-MZ; when the progression is still performed in `MPI_Wait` like in MiniMD, and when the OpenMP scalability is too far from linear.

In conclusion, the model is strong enough so as to be used to predict performance of dedicated core, but the user should always check all terms of the model and not only rely blindly on the total. These differences due to the approximation made in the model can help application developers find suboptimal behavior related to the use of nonblocking communications in their applications.

VI. Using the Model

In this section, we will detail how to use the model to understand the compromise between computation slowdown and the communication speedup, to evaluate the effectiveness of nonblocking communication usage, and to predict the maximum performance we would get if blocking communications were converted to nonblocking.

A. Understand the computation-slowdown/communication-speedup ratio

As we have seen for Kripke in Figure 2, the application exhibits two types of behavior with a dedicated core: either a significant speedup, or no gain at all. The beneficial cases are spread along the x-axis, showing that it does not depend on the problem sizes but instead on the communication scheme.

To understand this specific behavior, we analyze all the different timings gathered in the basic run displayed in table I, and the real value for $t_{\text{dedicated}}$ in addition. The case #1 is an example of successful use of dedicated core with 20% speedup; case #2 is typical of situations where the dedicated core does not bring performance gain. In both cases, the prediction of the model is correct with an error less than 2%.

Kripke features some calls to `MPI_Isend` and `MPI_Irecv`; they are progressed using `MPI_Testany` between computation phases. The communication is ended by a final call to `MPI_Waitall`. We observe in the table the main differences between the two runs is $N_{\text{nonblocking}}$: the first case uses a lot of nonblocking operations while the second has very few of them. It should be noticed that both cases execute approximately in the same duration, even though once decomposed, the timing details are very different.

In the first case, the application uses enough nonblocking operations for the communication speedup to overcome the computation slowdown. On the contrary, in the second case, the small number of nonblocking operations does not successfully counterbalance the computation slowdown. With our model, an application developer can better understand the dynamic behavior of its code, and know the cases where the nonblocking operations are used. The real impacting factor is the communication scheme and the amount of communications compared to computation.

B. Evaluate the effectiveness of nonblocking communication

An application may have everything theoretically to gain performance in terms of MPI overlappable communication and computation, and structurally never put computation and nonblocking communication in parallel. This is the case for NAS BT-MZ. As we have seen in Figure 3, on 128 nodes the model is too optimistic. This is due to the nonblocking operations being used not to overlap with computation, but to overlap multiple communications. Hence, even if globally there would be enough computation to overlap all nonblocking communications, it is not placed at the same time as the nonblocking communication.

If an application already uses nonblocking communications, the model may be used to diagnose pathological cases: by comparing an actual run and the prediction of the model, we can check whether overlap performs as expected or not, whether the computation inserted between the nonblocking operation and the corresponding wait is long enough or not. If the actual run is far from the prediction, there may be room for improvement in the organization of communications and computations.

We have observed the symmetrical case, where the application performs actually better with a dedicated core than predicted by the model. It is especially the case with Kripke, on the points of Figure 2 with a speedup higher than 1.1. When looking at the detailed parameters we measured, we observe that with the dedicated core, N_{test} drops compared to the reference run. The model assumes that `MPI_Test` will be shorter with dedicated core, because it will not have to make communication progress in the call itself. But, in addition, N_{test} is decreased, because the communication finished earlier. A more progression-compliant version of Kripke should remove the calls to `MPI_Test` and let the dedicated core do all the progress work in the background. However, this requires a refactoring of the application. The lesson learned is that we may gain more than the sole cost of communication progression. In addition, we may gain the cost of all tests scattered throughout the code.

	Case #1	Case #2	Case #3
number of nodes	16	128	128
problem size	200^3	200^3	300^3
$t_{\text{noprogress}}$	112	17.9	57.3
t_{MPI}	23.4	8.17	22.4
t_{model}	98	12.61	54.9
$t_{\text{dedicated}}$	122	16.9	63.7
N_{core}	16	16	16
t_{comp}	94.55	9.73	34.9
N_{blocking}	5863	5863	263
$t_{\text{minMPInonblock}}$	$2.50e-05$	$7.49e-05$	$1.03e-05$
N_{wait}	5862	5862	5862
$t_{\text{minMPIwait}}$	$5.33e-06$	$5.47e-06$	$5.59e-06$
$t_{\text{MPIblocking}}$	19.16	4.99	0.41
t_{MPIother}	2.20	1.42	17.8

TABLE II: Values of model parameters for three sets of parameters of MiniMD (times in seconds).

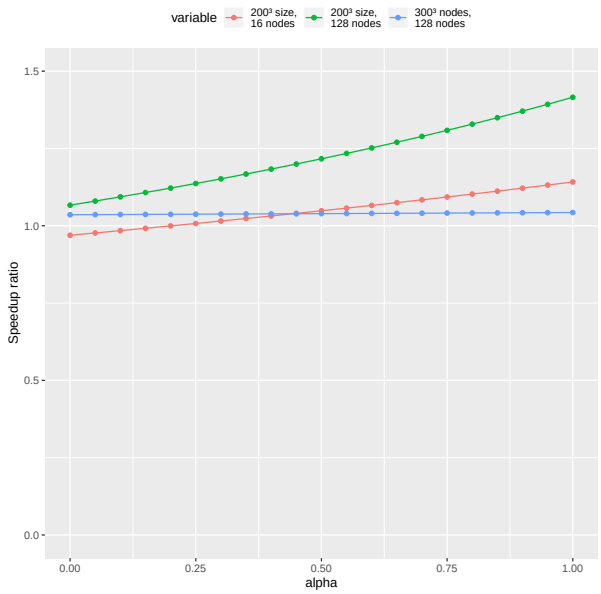


Fig. 5: Evolution of predicted speedup relative to the proportion of blocking call converted to nonblocking for MiniMD

C. Potential gain of transforming blocking call to nonblocking

Finally, the model is able to predict the potential gain of transforming blocking operations to nonblocking ones. As described in Section III-C2, the model can take into account the behavior of an application if a ratio α of blocking operations are changed to their nonblocking counterparts.

We apply the model on MiniMD, on the three configurations used in Section V. MiniMD uses mostly blocking calls, with very few nonblocking operations. The figures for the three configurations are gathered in table II. The predicted speedups for varying α for each configuration are displayed in Figure 5. A ratio $\alpha = 0$ is equivalent to the original unmodified application. A value $\alpha = 1$ means that all blocking communications

are transformed to their nonblocking counterparts. However, the ability to transform blocking to nonblocking calls depends on having available data-independent computation between the call and the wait. For most applications, $\alpha = 1$ is not reachable; we need to study the full range of values for α.

The results for the first and third sets of parameters (red and blue line) show that, even in the unlikely event of being able to convert all communications to nonblocking, the expected gain is poor. Values shown in table II reveal that the number of blocking calls N_{blocking} is low and the time spent in blocking communication $t_{\text{MPIblocking}}$ is low compared to the 1/16th overhead on computation. In the first case, for realistic values of α, no gain may be expected. In the third case (blue line), the model predicts a very small gain for any value of α. However, we know that there would be actually no gain at all, considering that we have already seen in Section V that our model is optimistic and overestimate by 12% the potential gain for this precise application.

The second set of parameters (green line) is the same as the first, except for the higher number of nodes. The predictions of the model, however, are very different, with a predicted significant performance improvement, even for moderate values of α. With the higher number of nodes compared to the first case, the computation time is lower, and thus the communication time gets a higher proportion of the total time, which is enough to compensate for the slowdown cause by one less core for computation with a dedicated core.

The slope of each case plot is the most interesting metric as it represents the efficiency of blocking to nonblocking transformation. The first and the second cases have a positive slope since the time spent in blocking communications $t_{\text{MPIblocking}}$ is much larger than what is lost in computation speed with a dedicated core; in contrast, the third case has a quasi-neutral slope. Even with t_{MPI} representing a large part of the total time, the proportion of $t_{\text{MPIblocking}}$ is negligible. As a consequence, α has a very low impact on the potential gain and the transformation is not efficient. Thus for every case, we have to evaluate the potential gain by running the model with the parameters from the application.

This model is thus able to discriminate the cases where a transformed application will be effective or not. Taking into account the base point of the α plot, it can show the current state of the dedicated core effect on the application. At last, looking at the slope for used case of the application will show the impact of the transformation on the performances using a dedicated core.

VII. Conclusion

Overlapping communications with computation is an efficient way to amortize the cost of communications. To do so, application programmers are supposed to use nonblocking MPI primitives. However, MPI libraries seldom exhibit actual progression without dedicating any core to communication progression, and as a consequence, application programmers seldom make the effort to use nonblocking communications.

In this paper, we have studied the compromise of dedicating one core to communication progression, the overhead it causes on computation, and the gain it brings to communication progression. We have proposed a model that is able to estimate the performance an application would get with a core dedicated to communication, based only on performance figures from a run without dedicated core. We have shown the validity of the model and explained its limits. We have used the model to explain the performance of an application that uses overlap, to check the effectiveness of overlap, and to predict the performance an application would get if converted to use nonblocking communications.

For future work, we aim at refining our model. First, the computation is currently modeled using a linear scaling model. For a better fit, the use of an application-specific model would provide better predictions. Also, we considered so far that there is enough computation to overlap communications, which is not always the case. We should identify whether computations may actually be overlapped or not. We aim to also model persistent operations. On the gathering of initial timings, we may rely on runtime instrumentation to have more precise and more focused timings. Finally, actually modifying applications to use nonblocking communication would validate the transformation part of the model.

References

[1] 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

[2] S. Paul, M. Araya, J. Mellor-Crummey, and D. Hohl, "Performance analysis and optimization of a hybrid seismic imaging application," *Procedia Computer Science*, vol. 80, pp. 8–18, 06 2016.

[3] MPI Forum, "MPI: A message-passing interface standard – version 4.0," Jun. 2021.

[4] M. Jiayin, S. Bo, Y. Wu, and Y. Guangwen, "Overlapping communication and computation in MPI by multithreading." 01 2006, pp. 52–57.

[5] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?" in *2008 IEEE International Conference on Cluster Computing*, Sep. 2008, pp. 213–222.

[6] M. Si, A. Peña, P. Balaji, M. Takagi, and Y. Ishikawa, "MT-MPI: multithreaded MPI for many-core environments," 06 2014.

[7] M. Miwa and K. Nakashima, "Progression of MPI non-blocking collective operations using hyper-threading," 03 2015, pp. 163–171.

[8] D. Buettner, J.-T. Acquaviva, and J. Weidendorfer, "Real asynchronous MPI communication in hybrid codes through OpenMP communication tasks," 12 2013, pp. 208–215.

[9] M. Wittmann, G. Hager, T. Zeiser, and G. Wellein, "Asynchronous MPI for the masses," 02 2013.

[10] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein, "Prospects for truly asynchronous communication with pure MPI and hybrid MPI/OpenMP on current supercomputing platforms," 01 2011.

[11] A. Ruhela, H. Subramoni, S. Chakraborty, M. Bayatpour, P. Kousha, and D. Panda, "Efficient asynchronous communication progress for MPI without dedicated resources," 09 2018, pp. 1–11.

[12] J. H. Göbbert, H. Iliev, C. Ansorge, and H. Pitsch, "Overlapping of communication and computation in nb3dfft for 3d fast fourier transformations," 02 2017, pp. 151–159.

[13] T. Straatsma and D. Chavarría-Miranda, "On eliminating synchronous communication in molecular simulations to improve scalability," *Computer Physics Communications*, vol. 184, pp. 2634–2640, 12 2013.

[14] A. Denis, J. Jaeger, E. Jeannot, and F. Reynier, "Experiments for Assessing Computation/Communication Overlap of MPI Nonblocking Collectives," Inria, Research Report RR-9367, Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-03012097

[15] 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

[16] J. C. Sancho, K. J. Barker, D. J. Kerbyson, and K. Davis, "Quantifying the potential benefit of overlapping communication and computation in large-scale scientific applications," in *SC'06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. IEEE, 2006, pp. 17–17.

[17] R. Brightwell, R. Riesen, and K. Underwood, "Analyzing the impact of overlap, offload, and independent progress for message passing interface applications." *IJHPCA*, vol. 19, pp. 103–117, 01 2005.

[18] A. Hori, E. Jeannot, G. Bosilca, T. Ogura, B. Gerofi, J. Yin, and Y. Ishikawa, "An international survey on MPI users," *Parallel Comput.*, vol. 108, p. 102853, 2021. [Online]. Available: https://doi.org/10.1016/j.parco.2021.102853

[19] H. J. R.F. Van Der Wijngaart, "Nas parallel benchmarks, multi-zone versions," NASA Ames Research Center, Moffett Field, CA, Tech. Rep., 2003.

[20] P. Crozier, H. Thornquist, R. Numrich, A. Williams, H. Edwards, E. Keiter, M. Rajan, J. Willenbring, D. Doerfler, and M. Heroux, "Improving performance via mini-applications," 01 2009.

[21] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Tech. Rep. LLNL-TR-641973, August 2013.

[22] A. J. Kunen, T. S. Bailey, and P. N. Brown, "Kripke - a massively parallel transport mini-app," 6 2015. [Online]. Available: https://www.osti.gov/biblio/1229802

[23] A. Daumen, P. Carribault, F. Trahay, and G. Thomas, "Scalomp: Analyzing the scalability of openmp applications," in *OpenMP: Conquering the Full Hardware Spectrum*, X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, Eds. Springer International Publishing, 2019, pp. 36–49.

[24] C. Iwainsky, S. Shudler, A. Calotoiu, A. Strube, M. Knobloch, C. Bischof, and F. Wolf, "How many threads will be too many? on the scalability of openmp implementations," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 451–463.

[25] J. Vetter and C. Chambreau, "mpip: Lightweight, scalable mpi profiling," *URL: http://www. llnl. gov/CASC/mpiP*, 01 2005.