Automatic Code Motion to Extend MPI Nonblocking Overlap Window

Van Man Nguyen[1,2,3,4], Emmanuelle Saillard[2], Julien Jaeger[1,3], Denis Barthou[2,4], and Patrick Carribault[1,3]

[1] CEA, DAM, DIF, F-91297, Arpajon, France
{van-man.nguyen.ocre,patrick.carribault,julien.jaeger}@cea.fr
[2] Inria, Bordeaux, France
{van-man.nguyen,emmanuelle.saillard,denis.barthou}@inria.fr
[3] Laboratoire en Informatique Haute Performance pour le Calcul et la simulation
[4] Bordeaux Institute of Technology, U. of Bordeaux, LaBRI, Bordeaux, France

Abstract. HPC applications rely on a distributed-memory parallel programming model to improve the overall execution time. This leads to spawning multiple processes that need to communicate with each other to make the code progress. But these communications involve overheads caused by network latencies or synchronizations between processes. One possible approach to reduce those overheads is to overlap communications with computations. MPI allows this solution through its nonblocking communication mode: a nonblocking communication is composed of an initialization and a completion call. It is then possible to overlap the communication by inserting computations between these two calls. The use of nonblocking collective calls is however still marginal and adds a new layer of complexity. In this paper we propose an automatic static optimization that (i) transforms blocking MPI communications into their nonblocking counterparts and (ii) performs extensive code motion to increase the size of overlapping intervals between initialization and completion calls. Our method is implemented in LLVM as a compilation pass, and shows promising results on two mini applications.

Keywords: Static Optimization · Message Passing Interface · Nonblocking communications.

1 Introduction

HPC applications (e.g., simulations) run on clusters which sport a mix of shared- and distributed-memory architecture. In this context, the computations are spread over multiple NUMA (non-uniform memory access) nodes that are interconnected using a high speed network. Thus the application needs to perform communications between those nodes to carry out the simulation. However the communications can introduce overheads due to idle times, either because a process is waiting for data another process must send, or because processes not

progressing at the same speed must synchronize. The time waiting on communications is not being spent on progressing the computation. A possible optimization would be to leverage these waiting times by performing computations independent of the communications.

The Message Passing Interface (MPI) defines multiple functions to perform communications over such distributed architectures. Among these operations, the nonblocking ones allow communications to asynchronously progress, thus enabling the overlap of communications by computations. Nonblocking communications are split into 2 distinct calls, one that initializes the exchange, and one that waits for its completion. To achieve overlapping, we have to insert computations, that are independent of the communications, between those calls so they can be performed while the communications are ongoing.

The use of nonblocking collective communications is however marginal. Many legacy codes still prefer blocking communications and the nonblocking form introduces a new complexity: it is up to the developer to make sure that the code does not have any race condition. As statements can be inserted and executed while the communication is ongoing, they can have an influence on the communication buffers. Many prior works proposed techniques to increase overlapping time by looking for specific patterns of code architecture such as producer-consumer loops or by performing basic code motion. In this paper we propose an automatic optimization that transforms blocking MPI calls into their nonblocking counterparts and that optimizes their overlapping potential through extensive code motion. Our contributions are the following :

- Automatic transformation of blocking MPI calls into their nonblocking mode.
- Increase of overlapping possibilities by performing extensive code motion to move apart data dependencies.
- Implementation using a state-of-the-art and widespread compilation framework (LLVM).

Section 2 presents related work on the use of nonblocking communications in optimizing HPC applications. Section 3 introduces a simple motivating example. Section 4 describes the optimization pass and finally, its implementation and the results are the subject of section 5.

2 Related Work

2.1 Asynchronous Communications in Scientific Applications

Many applications rely on nonblocking communications to improve performance on large-scale clusters. But code developers usually perform manual transformations and major redesign of widely-used algorithms to demonstrate the advantages of such nonblocking calls by reducing communication overheads.

Clement *et al.* proposed a sorting algorithm suited for distributed architectures [2]. The algorithm is an adaptation of a partition-based sorting algorithm

that leverages nonblocking calls in order to overlap communications with computations. Although their solution shows potential, it requires balance between the read and write, network, and computing times.

Similarly Kandalla *et al.* implemented the Breadth First Search algorithm with nonblocking neighborhood MPI collective communications [11]. Even if they show a communication overhead improvement up to 70%, the execution time does not improve and sometimes degrades. This might be caused by the additional operations that are needed to partition the problem.

Manually inserting testing points inside the overlapping window, such as calls to the `MPI_Test` function, is an approach taken by some developers to enforce the progression of asynchronous communications. Hoefler *et al.* used this solution to propose an optimization of a conjugate gradient solver [9] using LibNBC [10], a custom library which implements MPI nonblocking collective communications. Song *et al.* developed an algorithm for the 3D Fast Fourier Transform using nonblocking MPI collectives, and pushed this approach further by automatically determining a set of parameters, including the frequency of calls to `MPI_Test`, in order to achieve performance [14].

2.2 Automatic Transformation of MPI codes

On the topic of automatic transformations for MPI, Danalis *et al.* described communication-computation overlapping possibilities including the transformation of blocking calls into their nonblocking counterparts, the decomposition of collective calls into point-to-point ones, the application of code motion, variable cloning, and loop tiling and fission to increase the overlapping window [4]. ASPhALT implements a subset of those optimizations using the open64 source-to-source compiler [3]. It aims at optimizing producer-consumer loops by performing prepush transformations, meaning that it will try to send the data as soon as it is generated so that consumer computation can be performed while the next chunk of data is being produced. The producer-consumer loop is partitioned with an arbitrary size to control the amount of data that is generated, shared and computed.

Guo *et al.* showed how to improve this approach by adding a performance analytical model of the application [7]. With the help of user-added annotations, it predicts performance and decides when the transformation of blocking calls into nonblocking ones becomes worthy. The transformation itself and the code motion are still manually done.

Das *et al.* proposed an approach based on a Wait Graph to sink the completion call of nonblocking communications [5], that is to move it at a later point in the execution. This graph contains information about the control and data flow, enabling them to sink the wait call to the nearest statement that uses a communication buffer.

Petal [1] is a compiler pass implemented within the ROSE[13] compiler that also sinks completion calls to the nearest dependency point. Ahmed *et al.* used an alias analysis to detect whether a statement uses a communication buffer. Their

method transforms nonblocking communications into persistent communications when they are nested inside a loop.

Prior work on the transformation of MPI codes to expose communication-computation overlap possibilities has been mostly focused on a specific scenario such as producer-consumer loops. The attempts at widening the overlap frame have been limited by the nearest sensitive statement. In this paper we propose a solution that performs extensive code transformation and motion so that the size of the overlapping window can be significantly increased.

3 Motivating Example

This section illustrates how our work transforms MPI codes to increase the possibilities of overlapping communications with computations.

```
1    MPI_Alltoall(d1, sendcount, MPI_BYTE, d2,
2        recvcount, MPI_BYTE, MPI_COMM_WORLD);
3    matrix_multiply(a, b, res, matrix_size);
4    touch(d1);
5    matrix_multiply(a2, b2, res2, matrix_size);
```

Listing 1.1: Basic example

The *alltoall* communication at line 1 in Listing 1.1 is blocking. Every MPI process that is involved in the communication has to wait at that statement until their input communication buffers (`d1`) become available again, and until their output communication buffers (`d2`) have received the data from other MPI processes. A possible improvement in this context would be to translate that blocking *alltoall* call into nonblocking calls with an initialization (`MPI_Ialltoall`) and a completion (`MPI_Wait`). We can now move the completion call beyond the first matrix computation, as it is not involved in the communication, and before the function call that accesses the `d1` communication buffer.

```
1    MPI_Request req;
2    MPI_Ialltoall(d1, sendcount, MPI_BYTE, d2,
3         recvcount, MPI_BYTE, MPI_COMM_WORLD, &req);
4    matrix_multiply(a, b, res, matrix_size);
5    MPI_Wait(&req, MPI_STATUS_IGNORE)
6    touch(d1);
7    matrix_multiply(a2, b2, res2, matrix_size);
```

Listing 1.2: Optimized version of Listing 1.1

In prior work, the calls would be hoisted or sunk to the first statement that reads or writes to a communication buffer, depending on the call, as presented in Listing 1.2. However there are statements beyond the first dependency that are independent of the MPI call. Moving those statements along with the nonblocking call will further increase the overlapping window. Applied to the previous example, it results in the code in Listing 1.3. In this paper we propose a method to perform such code motion to increase the possibilities of overlapping communications with computations by identifying such boundaries and by displacing them further. In the previous code snippet, the completion and the `touch` calls are moved beyond the second matrix computation as well, leading to a wider overlapping window.

```
1    MPI_Request req;
2    MPI_Ialltoall(d1, sendcount, MPI_BYTE, d2,
3         recvcount, MPI_BYTE, MPI_COMM_WORLD, &req);
4    matrix_multiply(a, b, res, matrix_size);
5    matrix_multiply(a2, b2, res2, matrix_size);
6    MPI_Wait(&req, MPI_STATUS_IGNORE)
7    touch(d1);
```

Listing 1.3: Optimized version of Listing 1.1 with extensive code motion

4 Maximizing Communication-computation Overlap

As defined in the standard, a nonblocking MPI communication is composed of two calls: an initialization and a completion call. This form enables the overlap of communications with computations by inserting statements between these two calls. In order to avoid race conditions, those statements should not modify the communication buffers. As suggested by prior work, it is possible to perform multiple code transformations such as loop fission or sinking the wait to the nearest dependent statement to enlarge the overlapping frame. To go one step further, we propose to move not only the initialization call but also the statements that contribute to the values used in this call, and the same for the completion call and the statements that depend on it. Defining these backward and forward slices [15] of computations associated to the MPI calls, and their insertion points, is the heart of our contribution in order to increase the size of the overlapping window.

4.1 Finding slices and insertion point

The principle of the method is to automatically determine for any data-exchange based point-to-point and collective MPI call all statements that it depends on

(the backward slice for that call) and all statements that depend on it (the forward slice). These slices correspond to a sequence of statements connected by dependencies. In this work, the scope of these slices is limited to statements that are in the same control-flow structure: same function, same loop and same if-then-else construct. To find the slices and the insertion points, we specifically rely on the Control-Flow Graph (CFG) of the function. It is a directed graph where the vertices are basic blocks (BB). A basic block is a sequence of instructions (or statements) that have to be executed in a specific order. When the first instruction of a given BB has been executed, then the following instructions in that BB must be executed in that order. One can only enter a BB through its first instruction, and leave it through its last. The edges are the execution paths between the basic blocks.

For every point-to-point and collective communication, we consider their communication buffers and we iteratively build their backward slice. Starting from the MPI call, the CFG is backwardly visited and each statement that belongs to the use-def or def-use chains of the MPI call is collected. Each statement that uses or defines a communication buffer and all statements that it depends on are taken into account. Thus, those chains are iteratively scanned, allowing the slice to capture indirect dependencies as well. The iterative method stops when leaving the if-then-else, for loop, or function, surrounding the MPI call, or when the next statement to put in the list is in another control structure. The collected sequence of statements correspond to the backward slice, and the place in the CFG where the iterative method stops to the insertion point for this slice and for the initialization call. The same applies for the forward slice, moving forward in the CFG from the MPI call.

Algorithm 1 describes this code transformation for MPI communications, for the specific case of the initialization call insertion.

First, we build the backward slice of the call. In order to walk through the CFG from statement to statement, we extend the notion of dominance and post-dominance from BB to statements. A statement s_1 dominates a statement s_2 if s_1 belongs to a BB dominating the BB of s_2, or if s_1 precedes s_2 in the sequence of a BB.

We stop iterating over the statements once a suitable insertion point has been found for the initialization call. If needed, we allocate a new `MPI_Request` and create a new call site that will initialize the communication. That new nonblocking call site will use the same argument list as the blocking version, at which we append the request. Those newly created instructions are added at the insertion point. The correctness of an insertion point for the initialization call is defined by the function STMT_IMMOVABLE_INIT, and described in Section 4.2.

We operate the same way for the completion call by visiting the subsequent statements, starting at the MPI call site. Once we find a suitable insertion point for the completion call, we insert the `MPI_Wait()` call, using the `MPI_Request` that has been created for the corresponding initialization call, or the `MPI_Request` from the pre-existing nonblocking communication, as its argument.

Algorithm 1 Finding an insertion point for the initialization call

```
procedure INSERT_MPI_INIT_CALL(function)
Require: List of MPI communications called in function
Ensure: MPI nonblocking init calls are inserted along with their dependencies at valid locations.
    for all mpi_call ∈ function do
        list_stmt_init ← ∅
        V ← get_dependencies(mpi_call)    ▷ Build the list of statements upon which the MPI call depends using use-def and def-use chains.
        stmt ← mpi_call.get_stmt()
        while stmt_immovable_init(stmt, mpi_call.get_stmt(), V) = false do
            stmt ← immediate dominator(stmt)
            if stmt ∈ V then
                list_stmt_init ← list_stmt_init ∪ {stmt}
        insert_init ← stmt
        Move statements from list_stmt_init to the point of the code where stmt is the immediate dominator, and insert the init call

procedure STMT_IMMOVABLE_INIT(stmt, call_stmt, V)
Ensure: True if stmt is a valid insertion point
    if stmt is the first statement of the function then return true
    for all tstmt between stmt and its immediate dominator do
        if tstmt ∈ V then return true
    if call_stmt is between stmt and its immediate post-dominator then return true
    if stmt is a MPI procedure and stmt ≠ call_stmt then return true
    return false
```

Finally, the original blocking call is removed from the function. If the communication was already nonblocking, then the original call is simply moved to the first insertion point.

4.2 Defining a suitable insertion point

For each MPI communication, the insertion point for the initialization or the completion call is the statement after which we will move the initialization, or before which we will move the completion call. The specific case for the initialization is displayed in the STMT_IMMOVABLE_INIT function of Algorithm 1.

A statement is an insertion point if :

- The statement is the first statement of the current function.
- There is a control flow dependency.
- The statement is an MPI call. This constraint prevents from undoing previous transformations and from having different sequences of MPI collective calls, while allowing multiple pending nonblocking calls. Moreover according to the standard, it is not allowed to execute MPI functions beyond the boundaries defined by calls such as `MPI_Init` and `MPI_Finalize`.

In the literature another condition would also be a suitable insertion point:

– There is a data dependency between the call and the current statement.

This condition is limiting the size of the overlapping interval. While it is necessary to not overlap such data dependencies to keep the correctness of the program, other statements beyond this first dependency might be completely independent of the MPI call. In such case it can be useful to not stop at this first data dependency, and to add it to the list of statements that will be moved around, along with the insertion of the initialization or the completion calls when a stronger condition is reached.

This limitation is the reason why this condition is not taken into account in our work. In the following section, we will describe how we deal with such data dependencies.

4.3 Displacing the dependencies to achieve overlap

While traversing the CFG, we visit every statement until a valid insertion point, defined in Section 4.2, is found either by going from immediate dominator to immediate dominator as shown in Algorithm 1 for the initialization, or by going from immediate postdominator to immediate postdominator for the completion call. In the meantime, every visited statement that belongs to the slice, thus every visited statement that use or define an argument of the MPI call, will be enqueued rather than being considered as an insertion point, as explained in the previous section. Those statements will need to be moved to the insertion location to keep the dependencies and to prevent race conditions that could be caused by the introduction of nonblocking communications. A queue is used to store those statements to ensure that the order in which they were visited can be reproduced.

When a suitable insertion point has been found for the initialization or completion call, we dequeue the instructions at that location while ensuring that the execution order of those statements is kept. In the case of the initialization call, the newly created `MPI_Request` (if necessary), and the nonblocking call are inserted after dequeuing all the dependent statements. In the case of the completion call, the call is inserted before dequeuing the other statements. This way, the order between the dependencies is kept.

5 Implementation and Experimental results

5.1 Implementation using LLVM

Algorithm 1 is implemented as a compilation pass in the LLVM compiler [12]: the code is represented as an intermediate representation (IR) which allows us to be completely independent of the source language. The only language-related information we need to consider is the representation of the MPI calls in the parsed language, to be able to correctly capture them. LLVM defines many

analysis passes whose results can be reused in other optimizations and user-defined passes. These passes provide us the list of loops, the dominator and post-dominator trees for a given function, and the use-def and def-use chains of each value. The pass is applied on selected files of an MPI application using the LLVM `opt` tool.

5.2 Experimental Results

All measurements are performed on a supercomputer based on Intel Sandybridge processors. This partition is composed of 3,360 cores, each one having 4,000 Mo of memory, distributed over 210 nodes. The nodes are interconnected using infiniBand. We used the OpenMPI installed by default on this environment, which is based on version `2.0.4`.

Our method is evaluated by measuring the duration of each newly created overlapping window for the motivating example presented in Listing 1.1 and for two mini applications from the Mantevo project [8]: miniMD and miniFE. For blocking calls that have been transformed into their nonblocking form we insert the time measurement functions immediately below the initialization and above its associated completion call : we measure the execution time of the statements that are inside the overlapping window. For example in Listing 1.3, the first reading would be placed after the `MPI_Ialltoall` between lines 3 and 4, and the second before the `MPI_Wait` between lines 5 and 6, thus measuring the duration taken by the two matrix computations that makes the overlapping window up.

All results are collected per process and averaged. We measure the effectiveness of our method by comparing non-iterative transformations (related work, denoted as `basic`) with extensive code motion (our method, denoted as `extended`). A wide overlapping window means a communication-computation overlap possibility.
Each version of all the codes was run to ensure numerical results remained valid with each transformation.

The example in Listing 1.1 is a slightly modified version of a benchmark designed to measure the performance of nonblocking MPI calls, specifically their ability to asynchronously progress communications [6]. This example helps verify the correctness and performance of the transformations. The matrix size in the `matrix_multiply` call is set so that the function takes a user-defined duration to complete, 2,500 microseconds in our runs.

Our optimization pass successfully translated the blocking *alltoall* call into its nonblocking counterparts and the completion call was sunk below the second matrix computation. Table 1 shows the duration of the overlapping window measured for Listings 1.2 and 1.3. The result confirms what we statically observed on the IR with an overlapping window of 4,803 microseconds, which roughly corresponds to the overlapping of both matrix computations when performing extensive code motion. Similarly when using a basic code motion technique the observed duration of the overlapping interval is at 2,406 microseconds, corresponding to the execution of one matrix operation.

MPI Call	File	Line	Interval duration basic (μs)	Interval duration extended (μs)
`MPI_Alltoall`	`bench.c`	28	2,406.57	4,803.15

Table 1: Overlapping window duration for the motivating example

MiniMD simulates molecular dynamics using the Lennard-Jones potential or the Embedded Atom Model (EAM). It is a simpler version of LAMMPS and is written in about 5000 lines of `C++` code. We used version `1.2`, the EAM force and a problem of size 128^3. The benchmark is deployed over 8 nodes, using 15 cores on each node. Applied to each file of the benchmark, our pass transformed 57 MPI calls. Out of those 57 calls, 30 were executed during the run. The most significant transformations are shown in Table 2, the 24 remaining transformations have an overlapping window that is too narrow to expose any potential gain for asynchronous progression. The `MPI_Allreduce` called in thermo.cpp shows the bigger overlapping interval when applying extensive code motion.

MPI Call	File	Line	Interval duration basic (μs)	Interval duration extended (μs)
`MPI_Allreduce`	`thermo.cpp`	133	0.05	65.84
`MPI_Bcast`	`force_eam.cpp`	524	41.59	54.34
`MPI_Bcast`	`force_eam.cpp`	525	32.53	42.51
`MPI_Bcast`	`force_eam.cpp`	526	25.66	35.37
`MPI_Bcast`	`force_eam.cpp`	527	16.71	18.31
`MPI_Bcast`	`force_eam.cpp`	528	9.40	10.09
Max. MPI Call Overlap			125.94	226.46

Table 2: Most significant overlapping window duration for miniMD

MiniFE aims at approximating an unstructured implicit finite element application using fewer than 8000 lines of code in `C++`. We used version `2.0` and as with miniMD, measurements use the reference benchmark and a problem of size 1024^3. It is also run on 8 sandy nodes using 15 cores on each. Our pass found and transformed 37 MPI calls. Out of those 37 calls, 22 were detected at runtime and only 3 of them had a significant overlapping window in either the basic or the extensive case. The duration of their overlapping interval is shown in table 3. The basic approach is unable to expose any overlapping potential. Using extensive code motion, we successfully created an overlapping window of 4 milliseconds.

5.3 Discussion

In this section we chose to display the duration of the overlap windows instead of the actual execution time of each program. Success in hiding the commu-

MPI call	File	Line	Interval duration basic (μs)	Interval duration extended (μs)
`MPI_Allreduce`	`SparseMatrix_functions.hpp`	313	0.11	4193
`MPI_Bcast`	`utils.cpp`	92	0.51	166
`MPI_Allreduce`	`make_local_matrix.cpp`	216	0.22	1.41
Max. MPI Call Overlap			0.84	4360.41

Table 3: Most significant overlapping window duration for miniFE

nication times of MPI nonblocking calls heavily depends on the MPI runtime implementation and on how efficient it is in conducting asynchronous progression. Nonblocking MPI communications are also often more time consuming than their blocking counterparts, mainly due to the progression mechanism. For these reasons the performance gain one can achieve depends more on the quality of the MPI implementation than on the quality of the transformation method.

As our work focuses on increasing the size of the overlap windows, it is clearer to display the duration of these intervals. Their duration does not depend on the quality of the MPI implementation, and allows to clearly show the benefits of our method when compared to state-of-art.

It is also necessary to note that our optimization pass only detects dependencies that can be resolved through the semantics of the code. As a consequence, it is not able to properly capture statements or calls that have no data dependencies, yet that have an implicit relationship with the communication, such as probes to measure the communication time.

6 Conclusion

In this paper we propose a method to automatically perform extensive code motion in order to increase overlapping opportunities for nonblocking MPI communications. Our algorithm builds on and improves state-of-the-art methods to transform all blocking communications of a program into nonblocking operations. While previous work only moves apart the nonblocking calls to the first instruction they depend on, we use code motion to further extend computation-communication overlaps. Our method was implemented as a pass in the LLVM compiler and successfully tested on two miniapplications.

In future work, we will aim at improving the support for already existing nonblocking communications. In the current implementation, only initialization calls are moved, because we did not yet succeed in matching existing completion calls (`MPI_Test*()` and `MPI_Wait*()`) to their corresponding initialization calls. Thus, the code motion misses information to capture all necessary data dependencies to ensure the validity of the insertion point. Being able to link the completion calls to their respective initialization calls will allow moving both calls to increase overlap possibilities. Another limitation of our approach is the

analysis being intraprocedual. Pushing the boundaries of the analysis beyond the current function would further improve overlap possibilities.

References

1. Ahmed, H., Skjellum, A., Bangalore, P., Pirkelbauer, P.: Transforming Blocking MPI Collectives to Non-Blocking and Persistent Operations. In: Proceedings of the 24th European MPI Users' Group Meeting. pp. 1–11 (2017)
2. Clement, M.J., Quinn, M.J.: Overlapping Computations, Communications and I/O in Parallel Sorting. Journal of Parallel and Distributed Computing **28**(2), 162–172 (1995)
3. Danalis, A., Pollock, L., Swany, M.: Automatic MPI Application Transformation with ASPhALT. In: 2007 IEEE International Parallel and Distributed Processing Symposium. pp. 1–8. IEEE (2007)
4. Danalis, A., Pollock, L., Swany, M., Cavazos, J.: MPI-Aware Compiler Optimizations for Improving Communication–Computation Overlap. In: Proceedings of the 23rd international conference on Supercomputing. pp. 316–325 (2009)
5. Das, D., Gupta, M., Ravindran, R., Shivani, W., Sivakeshava, P., Uppal, R.: Compiler–Controlled Extraction of Computation–Communication Overlap in MPI Applications. In: 2008 IEEE International Symposium on Parallel and Distributed Processing. pp. 1–8. IEEE (2008)
6. Denis, A., Trahay, F.: MPI Overlap: Benchmark and Analysis. In: 2016 45th International Conference on Parallel Processing (ICPP). pp. 258–267 (2016)
7. Guo, J., Yi, Q., Meng, J., Zhang, J., Balaji, P.: Compiler–Assisted Overlapping of Communication and Computation in MPI Applications. In: 2016 IEEE International Conference on Cluster Computing (CLUSTER). pp. 60–69. IEEE (2016)
8. Heroux, M.A., Doerfler, D.W., Crozier, P.S., Willenbring, J.M., Edwards, H.C., Williams, A., Rajan, M., Keiter, E.R., Thornquist, H.K., Numrich, R.W.: Improving Performance via Mini-Applications. Sandia National Laboratories, Tech. Rep. SAND2009-5574 **3** (2009)
9. Hoefler, T., Gottschling, P., Rehm, W., Lumsdaine, A.: Optimizing a Conjugate Gradient Solver with Non-Blocking Collective Operations. In: European Parallel Virtual Machine/Message Passing Interface Users Group Meeting. pp. 374–382. Springer (2006)
10. Hoefler, T., Lumsdaine, A.: Design, Implementation, and Usage of LibNBC. Tech. rep., Open Systems Lab, Indiana University (Aug 2006)
11. Kandalla, K., Buluç, A., Subramoni, H., Tomko, K., Vienne, J., Oliker, L., Panda, D.K.: Can Network-Offload Based Non-Blocking Neighborhood MPI Collectives Improve Communication Overheads of Irregular Graph Algorithms? In: 2012 IEEE International Conference on Cluster Computing Workshops. pp. 222–230. IEEE (2012)
12. Lattner, C., Adve, V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In: International Symposium on Code Generation and Optimization, 2004. CGO 2004. pp. 75–86. IEEE (2004)
13. Quinlan, D.: ROSE: Compiler Support for Object–Oriented Frameworks. Parallel Processing Letters **10**(02n03), 215–226 (2000)
14. Song, S., Hollingsworth, J.K.: Computation–Communication Overlap and Parameter Auto-Tuning for Scalable Pparallel 3-D FFT. Journal of computational science **14**, 38–50 (2016)

15. Weiser, M.: Program Slicing. In: Proceedings of the 5th International Conference on Software Engineering. p. 439449. ICSE 81, IEEE Press (1981)