ON COMMUNICATION-COMPUTATION OVERLAP IN HIGH-PERFORMANCE COMPUTING

A Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Youcef Barigou

May 2016

ON COMMUNICATION-COMPUTATION OVERLAP IN HIGH-PERFORMANCE COMPUTING

Youcef Barigou

APPROVED:

Dr. Edgar Gabriel, Chairman
Dept. of Computer Science, University of Houston

Dr. Jaspal Subhlok
Dept. of Computer Science, University of Houston

Dr. Shishir Shah
Dept. of Computer Science, University of Houston

Dr. Lars Grabow
Dept. of Chemical & Bio-molecular Engineering,
University of Houston

Dean, College of Natural Sciences and Mathematics

Acknowledgments

I would like to express my special appreciation and thanks to my advisor, Dr. Edgar Gabriel, for being an outstanding mentor for me, for encouraging my research and for allowing me to grow as a research scientist, as well as for the precious advice and help on both research and career.

I would also like to thank my committee members, Dr. Jaspal Subhlok, Dr. Shishir Shah, and Dr. Lars Grabow for serving as my committee members, and for the highly constructive comments and suggestions.

A special thanks to my family: my parents, my brother, my sister, my brother-in-law and my uncle for supporting me both financially and spiritually throughout this degree, and my life in general.

Finally, I would like to thank anyone who contributed and helped me either directly or indirectly throughout this achievement.

ON COMMUNICATION-COMPUTATION OVERLAP IN HIGH-PERFORMANCE COMPUTING

A Dissertation Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

By

Youcef Barigou

May 2016

Abstract

The number of compute nodes and cores per node have increased many fold on high end computer systems over the last decade. For a parallel application to scale to tens or even hundreds of thousands of processes, all non-computing related operations have to be kept at an absolute minimum, including communication operations.

Non-blocking-collective operations extend the concept of collective operations by offering the additional benefit of being able to overlap communication and computation. However, it has been demonstrated that collective operations have to be carefully tuned for a given platform and application scenario to maximize their performance. Also, using non-blocking-collective operations in real-world applications is non-trivial. Application codes often have to be restructured significantly in order to maximize the communication-computation overlap.

The goal of this dissertation is to optimize non-blocking collective-communication operations and facilitate their utilization at the end-user level. This is achieved by an automatic run-time tuning of non-blocking collective-communication operations, which allows the communication library to maximize communication-computation overlap, and choose the best performing implementation for a non-blocking-collective operation on a case by case basis. Specifically, an approach to maximize the communication-computation overlap for hybrid OpenMP/MPI applications is developed. It leverages automatic parallelization by extending existing concepts to utilize non-blocking-collective operations. It also integrates the run-time auto-tuning techniques of non-blocking-collective operations, optimizing both, the algorithms used for the non-blocking-collective operations as well as location and frequency of accompanying progress-function calls.

Contents

List of Figures

Chapter 1

Introduction

Carrying out simultaneous calculations to solve large computations using multiple-compute resources is the central theme of parallel computing. In order to execute a computation in parallel, it needs to be broken down into independent sub problems and then distributed across available resources. In general, there are two kinds of resources: shared memory and distributed memory. This chapter summarizes the research presented in this dissertation by describing research challenges and goals, and presenting the outline of this dissertation.

1.1 Motivation

High-Performance Computing (HPC) is nowadays involved in different fields of science and industry like physics simulation, off-line movie rendering, weather forecasting, video encoding, etc. by performing large computations on supercomputers. In fact, these fields have an increasing need for computing power in order to solve

complex problems that would take a very long time on regular computers. For example, simulations often require a trade off between accuracy and compute time, i.e. in order to achieve more accurate results, more data has to be processed and therefore, more compute power is needed. Computer architectures can be classified into three categories: Single Instruction Multiple Data (also SIMD) architectures, which are capable of executing the same operation on different elements of a vector in one clock cycle (e.g. vector architectures such as Cray-1); Single Instruction Multiple Threads (or SIMT) architectures which are massively parallel architectures composed of a multitude of less-powerful processors that execute the same operations on a large amount of data with the same parameters (e.g. Graphics Processing Units or GPUs); and finally, Multiple Instructions Multiple Data (or MIMD) architectures, which represent the most general computing architecture. In MIMD architectures, different processors can execute different operations on different data. A very common MIMD architecture is clusters, which are 'supercomputers' composed of a number of multi-core computers (called nodes) interconnected through a fast network (e.g. ethernet or Infiniband).

Among the benefits of using Parallel Computing is the accumulation of resources such as the memory (which is often limited on regular computers) and the possibility of running multiple tasks at the same time. However, the most common and natural way for a scientist to write a program for a large scientific problem is such that it can be executed only on one processor (ie. a single sequential task) instead of multiple processors at the same time. These programs are usually parallelized afterwards

either by programmers who manually rewrite these programs to port them on parallel architectures or by using automatic parallelizers that take as input the initial sequential version and automatically generate a parallel version of the program. Parallelizing a program can be very complex because it is a non deterministic process, i.e. for a single program, a large number of more or less efficient parallel versions can be generated. The best parallelization strategy depends on numerous factors, for example the targeted architecture and the availability of software libraries. A parallel program consists of multiple sub-programs (or tasks) that are executed on different processors in parallel. Since these tasks are executed independently from each other, they sometimes need to communicate. For example, if a processor needs an intermediate result in order to carry on its calculations, and this result is calculated in another task by a different processor, then the processor that needs the data has to wait for the other processor to either explicitly provide the data directly through message passing, which is common in multi-processing (or distributed memory programming, cf. 1.1.1); or indirectly by storing the data in a memory space that both processors can have access to, which is common in multi-threading (or shared-memory programming, cf. 1.1.1). In the case of clusters, if the two processors are located in the same node, then the communication can occur within the node itself without going explicitly through the network. However, if the communication is between two processors from different nodes, then it has to be carried out by the network.

1.1.1 Multi-threading vs. Multi-processing

As mentioned in the previous paragraph, multi-tasking can be implemented in two ways: multi-threading or multi-processing. Multi-threading and multi-processing involve running multiple threads and multiple processes, respectively, either on single processor or multiple processor systems. In the case of single processor systems, the actual observance of the computer doing multiple tasks at the same time is an illusion. Reason is that a software scheduler performs time-slicing on single CPU architectures. In this case, only a single task is executing at any given time, but the scheduler is switching between tasks fast enough so that the user never notices that there are multiple threads or processes contending for the same CPU resource. However, multi-threading or multi-processing on multiple processors reduces the time-slicing. The time-slicing effect is still there, because a modern OS could have hundreds of threads or processes contending for two or more processors, unless it is a one-to-one relationship in the number of threads or processes to the number of processing cores available (in which case it is a true parallel execution). So at some point, a thread or a process will have to stop for another thread or process to start on a CPU that the two or more threads or processes are sharing. This is again handled by the scheduler of the operating system.

With a multi-processors system, separate executions can happen at the same time, unlike with the uni-processor system. If multiple threads or processes are being run, and are executing a highly parallel task, then more cores can be used, for better performance. Moreover, if the parallel task requires a sequential component, a thread or process will be waiting for the result from another thread or process before

it can continue. In this case, some type of barrier or synchronization can be utilized so that the threads or processes that need to be idle are not spinning using CPU time, and only the threads or processes that need to run are contending for CPU resources. On the other hand, multi-threading or multi-processing does not necessarily mean that a single thread or process uses more than one processor simultaneously.

The main difference between multi-threading and multi-processing is that threads (of the same process) run in a shared address space, while processes run in separate address spaces. Consequently, if different threads are generated by the same process, then data communication between these threads is (in general) implicitly performed through the memory that they share, which makes the communication faster. However, this requires more complex programming in order to synchronize the threads and avoid concurrent writings to the same memory address for example. On the other hand, if two different processes need to communicate data, data has to be explicitly sent/received by the two processes (using sockets or streams for example), which is more costly than using a shared memory space, especially if the communication has to travel to an external network support. In the case of clusters, although the memory is shared by all the cores within a node, it is possible to run multiple processes on the same node. Each will have its own virtual address space, so multi-processing can be used within a single node as well.

Yet another possibility is to use multi-threading that takes advantage of multiple cores on a single node and multi-processing for exploiting many nodes. This is sometimes referred to as hybrid programming. It allows the programmer to map the parallelism that exists in the program onto the characteristics of the machine.

To sum up, both paradigms have advantages and disadvantages. Therefore, efficient parallelization on clusters is usually achieved using hybrid programming: multi-processing eliminates most needs for synchronization primitives and code is usually straightforward; multi-threading is lightweight (low-memory footprint) and avoids communication overhead.

1.1.2 Parallel Programming Languages and Libraries

Different programming languages, libraries, Application Programming Interfaces (APIs), and parallel programming models have been created for programming parallel computers. For shared memory architectures, where programming languages communicate by manipulating shared memory variables, POSIX Threads [81] and OpenMP [24] are two of the most widely used shared memory APIs. Whereas for distributed memory architectures, the Message Passing Interface [42, 66] (MPI) is the most widely used API.

MPI

MPI, designed in 1993-94, is a standard that defines a library of functions that can be used with C, C ++ and Fortran. It uses distributed-memory programming by message passing. It has become a de facto standard for communication nodes running multi-processing programs on distributed memory systems. MPI was written to achieve good performance on both massively parallel machines with shared memory as well as clusters of heterogeneous distributed memory systems. It is available on many hardware and operating systems. Thus, MPI has the advantage over older

message passing libraries to be highly portable (because MPI has been implemented on almost all memory architectures). Since 2013, a new version of MPI is available , MPI-3 [35], which provides powerful additional features.

Point-to-point vs. collective-communications Point-to-point communications allow two processes within the same communicator to exchange data (scalar, array, or derived type). A communicator is a set of processes that can communicate with each other and both processes can communicate only if they are in the same communicator. The corresponding functions are `MPI_Send`, `MPI_Recv` and `MPI_Sendrecv`.

On the other hand, collective-communications in MPI involve all processes of a communicator. It is possible to send the same data to all processes (`MPI_Bcast`), distribute an array between all processes (`MPI_Scatter`), or to perform a reduction operation (e.g. addition) where each process will be involved (`MPI_Reduce`).

Collective-communication operations have been a key concept used in large-scale parallel applications to minimize communication costs. The interface specification of collective operations represents a higher-level abstraction for often occurring communication patterns, separating the desired outcome of the group communication from its actual implementation. Hence, they allow for numerous optimizations internally at different levels:

- **Software level optimization**: different algorithms can provide different performances. To highlight this benefit, consider for example the costs of a broadcast operation when using a simple, linear algorithm ($O(N)$) – as is often the case in applications not using collective-communication interfaces – vs. an implementation of a broadcast operation using a binary tree internally

($O(log(N))$), with N being the number of processes. For tens or hundreds of thousands of processes, the difference between these two broadcast operations will be enormous and will contribute significantly towards the scalability of the overall application. Figure 1.1 shows the estimated execution times of a linear and a binary tree broadcast algorithm of 1 MB message length using QDR InfiniBand parameters using the Hockney's model [45].

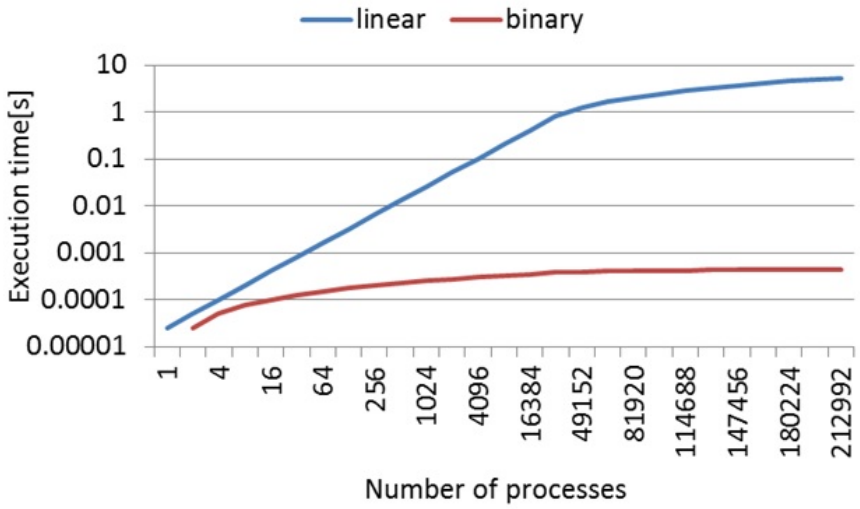

Figure 1.1: Estimated execution time of a broadcast operation of 1 MB message length using QDR InfiniBand parameters.

- **Hardware level optimization**: takes into account the hardware topology by minimizing the utilization of the lowest performing network connections. One can also consider the fact that some networking cards have built-in support for some collective operations. In fact, collective operations can be offloaded into some specialized networking cards, which enables the processes involved

in collective operations to perform other useful work simultaneously.

Collective operations are therefore essential for scalability of applications at large-process counts and they simplify code maintenance as well as readability (one function call instead of a user-level implementation).

Blocking vs. Non-blocking communication Point-to-point communication operations in MPI can be either blocking (synchronous) or non-blocking (asynchronous). A blocking communication operation blocks the processes involved until the operation completes. Specifically, the function calls `MPI_Send`, `MPI_Recv` do not return until the communication is finished. On the other hand, a non-blocking communication operation only initiates the operation. Progression and completion of a non-blocking operation have to be enforced separately using functions such as: `MPI_Test` for progression, and `MPI_Wait` for completion (does not return until the communication is finished). An `MPI_Request` object uniquely identifies a currently ongoing non-blocking operation. Asynchronous message passing allows more parallelism. Since a process does not block, it can do some computation while the message is in transit. In the case of receive, a process can express its interest in receiving messages on multiple ports simultaneously. In a synchronous system, such parallelism can be achieved by forking a separate process (or thread) for each concurrent operation, but this approach incurs the cost of extra process management.

Likewise, collective operations can also be either blocking or non-blocking.

Another key aspect of non-blocking operations is progression. As mentioned previously in this subsection, non-blocking point-to-point operations in MPI require

progression and completion function calls in order to ensure that the non-blocking operation is progressing or finishing, respectively. Consequently, non-blocking-collective operations need a mechanism to continue execution in the background. There are three ways to ensure progression (and completion) of a non-blocking-collective operation:

- **Using a progress thread**: each process spawns a separate thread that executes the non-blocking operation in a blocking manner. This approach has the drawback that the number of threads can be large (e.g. 1000 non-blocking send operations to different processes).

- **Offloading to the hardware**: some networking cards include hardware support for collective operations which allows them to be executed in the background (apart from the processes), and therefore enables communication and computation overlap since the processes are not blocked. However, portability is restricted in this case due to the need of specialized hardware.

- **Using a progress engine**: by regularly invoking progress/completion functions (similarly to `MPI_Test`/`MPI_Wait`) at the user-level, the MPI library makes the progress of the underlying non-blocking point-to-point operations that constitute the non-blocking-collective operation.

As of today, using a progress engine is the most widely used option because making an MPI library thread safe is highly challenging.

1.2 Research Challenges

1.2.1 Scalability to Large Process Counts

The number of compute nodes and cores per node have increased many fold on high end computer systems over the last decade. For a parallel application to scale to tens or even hundreds of thousands of processes, all non-computing related operations have to be kept at an absolute minimum, including communication operations. MPI is, as of today, the most widely utilized communication paradigm for parallel applications in high-performance computing. It includes collective-communication operations that express communication patterns using higher level abstractions, and to use scalable algorithms and/or hardware support for their implementation.

However, some communication-intensive collective operations are still very costly in terms of performance, and can entail dramatic scalability limitations depending on both the application and the hardware support. In order to corroborate this statement, the scalability of two common scenarios in high-performance computing, involving a collective operation each is evaluated below.

The scalability of an application is defined by its speedup. Speedup is a metric for relative performance enhancement when executing a parallel task. Established by Amdahl's law [44], the speedup of a given application for p processes is the fraction of the execution time of the application by one process to the execution time of the same application by p processes. Ideal speedup (or linear speedup) is therefore equal to p. When running an application with linear speedup, doubling the number of processors doubles the speed. As this is ideal, it is considered very good scalability.

A method to estimate the theoretical performance of parallel applications is the use of parallel models, rather than running experiments. One basic formula to estimate the time it takes to send a message through the network is $L + \frac{m}{B}$, where:

- L is the latency of the network, i.e. the time it takes for a single byte to travel the network.
- B is the bandwidth of the network, i.e. how many bytes per second can travel the network.
- m is the message length (in bytes).

This formula is also called Hockney's model [45] (where network congestion and concurrency of messages are not considered).

In the following, Hockney's model was used to theoretically evaluate the scalability of two common high-performance computing scenarios. The first scenario consisted of an application that performed a 2D Cartesian stencil collective-communication operation, and the second one consisted of an application that performed an all-to-all collective-communication operation. In order to emulate typical high-performance computing behaviors, each collective operation was performed within a loop that encompasses computations.

Let

- $S_{stencil}$ be the speedup of the 2D Cartesian stencil application.
- $S_{alltoall}$ be the speedup of the all-to-all application.
- N be the total problem size.

- N_{it} be the total number of iterations.
- P be the number of processes.
- S be the size of the data to be communicated per process.
- L be the latency of the network.
- B be the bandwidth of the network.
- T_{comp} be the total computation time.
- T_{comm} be the total communication time.
- T_{1proc} be the total time needed for one process to execute the application.
- T_P be the total time needed for p processes to execute the application.

2D Cartesian stencil communication

As depicted in figure 1.2, each process in a 2D Cartesian stencil communication has four communication partners (except the ones at the boundaries), where four sends and four receives occur sequentially.

As mentioned previously, the speedup is determined by

$$S_{stencil\_2D} = \frac{T_{1proc}}{T_{nprocs}} = \frac{T_{1proc}}{T_{comp} + T_{comm}} \tag{1.1}$$

There is no communication in the case of a single process, therefore

$$T_{comp} = \frac{T_{1proc}}{P} \tag{1.2}$$

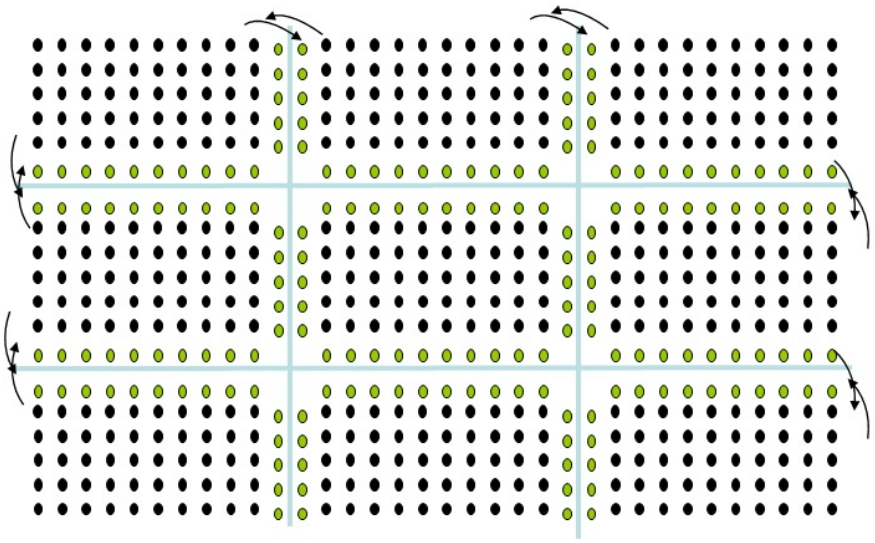

Figure 1.2: Example of 2D Cartesian stencil communication with nine processes (Black dots: private data, Green dots: communicated data).

Each process performs one send and one receive with four neighboring processes in every iteration. According to Hockney's model, the total communication time of the parallel execution is

$$T_{comm} = 2 * 4 * N_{it} \left[L + \frac{S}{B} \right] \tag{1.3}$$

The size of data to be communicated by each process in a 2D Cartesian stencil communication is

$$S = \sqrt{\frac{N}{P}} \tag{1.4}$$

Considering 1.1, 1.2, 1.3, and 1.4, the final speedup formula is

$$S_{stencil\_2D} = \frac{T_{1proc}}{\frac{T_{1proc}}{P} + 8N_{it}\left[L + \frac{\sqrt{N}}{B\sqrt{P}}\right]} \tag{1.5}$$

All-to-all Collective-Communication Operation

In an all-to-all communication operation, all pairs of processes communicate with each other, where one send and one receive operation occurs sequentially.

The speedup is determined by

$$S_{all-to-all} = \frac{T_{1proc}}{T_{nprocs}} = \frac{T_{1proc}}{T_{comp} + T_{comm}} \tag{1.6}$$

There is no communication in the case of a single process, therefore

$$T_{comp} = \frac{T_{1proc}}{P} \tag{1.7}$$

Each process performs one send and one receive with all the other processes in every iteration. According to Hockney's model, the total communication time of the parallel execution is

$$T_{comm} = 2 * N_{it}P\left[L + \frac{S}{B}\right] \tag{1.8}$$

The size of data to be communicated by each process in an all-to-all communication is

$$S = \frac{N}{P^2} \tag{1.9}$$

Considering 1.6, 1.7, 1.8, and 1.9, the final speedup formula is

$$S_{all-to-all} = \frac{T_{1proc}}{\frac{T_{1proc}}{P} + 2N_{it}\left[PL + \frac{N}{BP}\right]} \tag{1.10}$$

Results and Discussion

In order to illustrate the final speedup formulas 1.5 and 1.10, let us assume the following (realistic) values for the previous parameters, where two different network platforms - namely Gigabit Ethernet and QDR Infiniband - are considered:

$N =$ 10 GB.

$T_{1proc} =$ 1000 seconds.

$N_{it} =$ 500 (one collective operation every 2 seconds).

$L =$ $50\mu s$ for Gigabit Ethernet, and $1\mu s$ for QDR InfiniBand.

$B =$ 0.125 GB/s for Ethernet, and 40 GB/s for QDR InfiniBand.

Figures 1.3 - 1.4 display the corresponding speedups of the 2D Cartesian stencil and all-to-all collective operations, respectively, on each network platform.

The graphs indicate that communication costs limit the speedups, moderately for 2D Cartesian stencil communication, which however, gets worse when increasing the number of operations per second, and dramatically for all-to-all communication. Hence, although collective-communication operations reduce communication costs compared to user-level trivial algorithms (often used by applications that do not use collective operations), they induce high performance costs with large process counts, and represent a major factor towards the scalability of parallel applications.

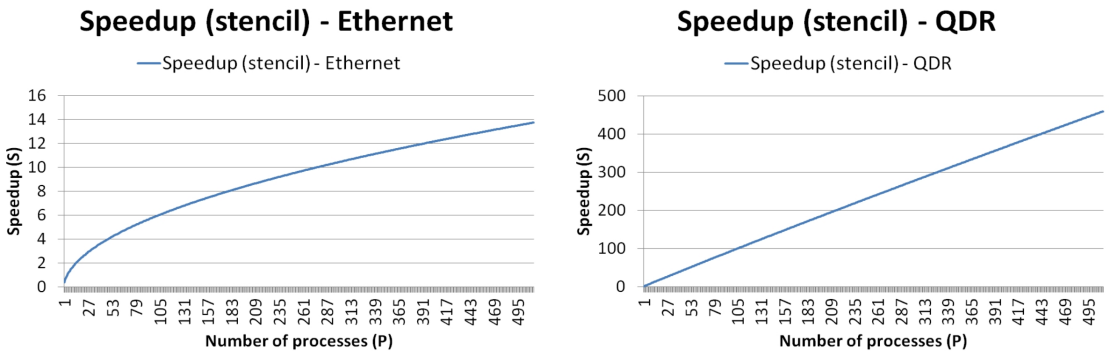

Figure 1.3: Speedup of 2D Cartesian stencil communication using Gigabit Ethernet (left) and QDR InfiniBand (right).

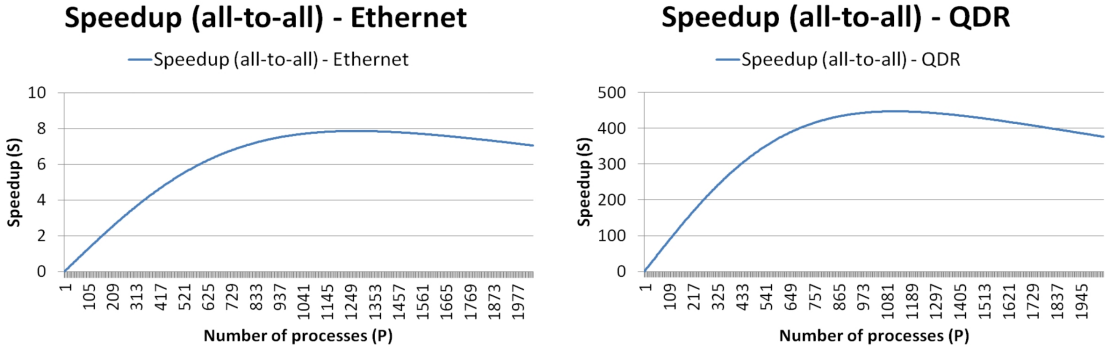

Figure 1.4: Speedup of all-to-all communication using Gigabit Ethernet (left) and QDR InfiniBand (right).

1.2.2 Optimization of Collective Operations

Communication libraries have invested enormous efforts over the last two decades to optimize collective-communication, developing a wide range of solutions such as hardware topology-aware algorithms [62, 78] or exploiting capabilities of the network adaptors [58]. However, implementations of collective-communication operations often require careful tuning in order to reach the desired performance. The challenge in optimizing collective operation stems from the fact that both the hardware/system utilized as well as the application characteristics have to be taken into account. From the system level perspective, factors influencing the performance of collective operations include networking technology and network topology, but also processor type and organization, memory hierarchies, operating system versions, device drivers, and communication libraries. In fact, one can argue that every parallel computer is (nearly) unique. Similarly, different parallel applications expose very different communication characteristics, such as frequency of data-transfer operations, volume of data-transfer operations, and process-arrival patterns, i.e. the time difference on how processes enter a collective operation due to (micro) load imbalances between the processes [30]. In [39], it was demonstrated that an implementation which minimizes the execution time of a collective operation on one platform and for one particular application scenario showed very poor performance on a different platform or even on the same platform but for different number of processes or application scenarios. Relying on one particular, hard-coded implementation of a collective operation will inevitably lead to sub-optimal performance in many scenarios.

One possible approach to deal with the large number of factors influencing the

performance of collective operations is to use auto-tuning techniques. Libraries such as ADCL [39] and STAR-MPI [31] have shown that collective operations which have the ability to adapt to the current-execution environment can significantly boost the performance of the application. The concept of these libraries is based on measuring the execution time of alternative implementations of a collective operation while executing the application itself, and use the version leading to the lowest execution time for the remainder of the application. Thus, they don't require extrapolation of performance results obtained from executing a benchmark for a limited number of test-cases and process counts, but optimize that particular application scenario on the set of nodes allocated for the current execution.

However, performance-based tuning of collective-communication operations does not necessarily entail versatile optimization. In fact, in order to assess the optimality of different implementations of collective operations, critical criteria other than performance could also be considered (e.g. cache-friendliness, energy efficiency, etc.).

1.2.3 Overlapping Communication and Computation

To further reduce the costs of communication operations, application developers often try to overlap communication and compute operations by using non-blocking operations. Non-blocking operations also simplify the code because application developers do not have to worry about scheduling of messages to avoid deadlock scenarios. However, it is not necessarily straight forward to obtain the desired overlap of communication and computation: ensuring that non-blocking communication

continues execution 'in the background' can be challenging, especially for single-threaded communication libraries. In addition, applications using non-blocking operations typically have a larger-memory footprint compared to application using blocking communication because separate buffers have to be used to support simultaneously ongoing communication and computation. Non-blocking operations have mostly been restricted to point-to-point operations. Work by Hoefler et.al [50] has, however, extended non-blocking operations to collective-communication as well, and has recently been adopted by the MPI standard [35].

Optimizing non-blocking collective-communication operations is, however, even more challenging than their blocking counterparts, first and foremost due to the fact that the actual time spent in the communication operation can not be directly measured. This violates a fundamental requirement of auto-tuning libraries, namely to have accurate and reliable measurements for the alternative implementations.

Furthermore, the performance of non-blocking operations is closely tied to their ability to ensure progress in the background, especially in the case of single threaded MPI libraries. In order to progress non-blocking operations, the application has to regularly invoke the progress engine of the MPI library [49]. For this scenario, auto-tuning offers the unique opportunity to optimize the number and frequency of progress calls: too few progress calls will not achieve the required overlapping between communication and compute operations. Too many progress calls can lead to an unnecessary overhead.

1.2.4 Code Generation

Although using non-blocking-communication operations provides an efficient way of improving the scalability of parallel applications, they complicate the code and can become very complex to apply. For example, when the size of the data that has to be communicated in a given iteration is calculated in an overlapping code region of the previous iteration. On top of that, finding code regions to overlap is actually not always trivial. Some algorithms usually require sophisticated transformations that are highly dependent on the corresponding computational problem. Ultimately, even if non-blocking-collective operations are auto-tuned, they can still entail the following complex steps towards being efficiently integrated into code:

- Detecting computation regions that overlap non-blocking-collective operations.
- Placing progress-function calls at appropriate locations.
- Increasing communication-computation overlap through automatic pipe-lining and tiling of non-blocking-collective operations (e.g. by applying double-buffering techniques).
- Avoiding concurrent data accesses.
- Co-tuning multiple non-blocking-collective operations and communication for performance optimization.

This requires high user expertise and knowledge about software libraries, hardware platforms, and the objective application itself. To circumvent this tedious effort, optimization-aware automatic code generation techniques could be used.

1.3 Research Goals

The main goal of our research is to dynamically optimize the performance of non-blocking collective-communication operations in nested loops that operate on (large) static problem sizes, along with improving their utilization in the context of distributed memory computing, as follows:

- Dynamic auto-tuning of non-blocking collective-communication operations by using, at run-time, initial nested-loop iterations to supply key statistical data in order to evaluate and auto-tune involved collective-communication operations on the fly. After the auto-tuning phase is accomplished, the remaining iterations are executed under the potential improvement of the auto-tuning phase. Ultimately, auto-tuning is performed in conjunction with actual execution. This is achieved through an efficient and user-friendly auto-tuning strategy that accurately measures the performance of different implementations of non-blocking-collective operations. An analysis of the factors that impact their performance (e.g. the frequency of progress function calls, the number of processes, the amount of communicated data, and the hardware platform) is performed. Micro-benchmarks will be used in order to theoretically assess the significance of these factors. The benefits of this approach will be demonstrated with real world application programs.

- On a broader scope, co-tuning combinations of blocking and non-blocking collective-communication operations, as well as computations at run-time in order to extricate the potential interference between them. Auto-tuning these

combinations as a whole could lead to a different outcome and performance improvement rather than auto-tuning them individually. This is achieved mainly by using an abstract and user-adjustable mechanism to properly evaluate the run-time performance of such combinations for the sake of auto-tuning.

- Dynamic auto-tuning of the frequency-of-progress function calls for non-blocking collective-communication operations, which is twofold: either adjusting a predefined distribution of frequencies to the statistical data collected throughout run-time auto-tuning, then finding the frequency that leads to the optimal performance; or spotting the ineffective progress function calls (i.e. they do not actually progress the non-blocking-collective operation, hence create a performance overhead), then skipping them in the subsequent part of the execution.

- Automatically generating parallelized versions from basic sequential codes that use, efficiently, non-blocking collective-communication operations. One way to achieve this challenging goal is by enhancing an abstract state-of-the-art C to MPI automatic parallelizer that is limited to straightforward usage of blocking-collective operations. This allows the user to not only bypass complex steps towards integrating non-blocking collective-communication operations into code, but also benefit from elaborate optimization, through leveraging automatic code generation as well as auto-tuning of non-blocking collective-communication operations. Three main sub-goals are targeted: (i) Applicability, which is the scope of HPC applications that the approach can be applied to (ii) Portability, which represents the breadth of architectures that the approach targets (iii) Effectiveness, which represents the potential amount of

communication-computation overlap that the approach produces.

The performance of different applications and micro-benchmarks will be evaluated on a variety of architectures in order to assess the performance benefits from the points listed above.

1.4 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents a literature review of projects applying static and dynamic optimizations on high-performance computing applications, and projects on collective algorithms in general (including blocking and non-blocking). It also points out the contributions and the shortcomings of each approach. Chapter 3 describes a strategy of dynamic auto-tuning of non-blocking collective-communication operations based on accurate instrumentation and analysis of different factors that affect performance. Chapter 4 presents a generic approach founded upon a model-based parallel code generator and auto-tuning techniques to optimize the communication-computation overlap. Finally, chapter 5 summarizes the scientific contributions of this work and presents the future perspectives in this research area.

Chapter 2

Related work

This chapter discusses the most relevant related work in auto-tuning of HPC applications focusing on auto-tuning libraries for communication operations. Auto-tuning projects can be characterized as either static or dynamic. Static tuning, which consists of optimizing applications or code sequences before run-time, provides programs that can not alter their behavior during execution. On the other hand, dynamic tuning designate run-time-driven optimization, and therefore produces programs that have the ability to adapt while the execution is ongoing. Additionally, this chapter reviews state-of-the-art projects on non-blocking collective-communication operations and collective algorithms in general. Furthermore, a section will be dedicated to projects involving collective-communication operations within the context of automatic parallelization.

2.1 Auto-tuning

2.1.1 Static Auto-tuning

A number of auto-tuning projects are based on an apriori tuning step which evaluates the performance of different versions for the same operation for various message lengths and process counts. Among the best known projects representing this approach are the Automatically Tuned Linear Algebra Software (ATLAS) [88] and the Automatically Tuned Collective-Communications (ATCC) [75, 84] frameworks. Both projects use an extensive configuration step to determine the best performing implementation from a given pool of available algorithms on a given platform. Other projects along the line of static tuning use performance estimation through mathematical models of different communication algorithms in order to predict the optimal algorithm for future execution.

ATLAS

Automatically Tuned Linear Algebra Software (ATLAS) is an optimized software library for linear algebra [88]. It provides automatically tuned implementations of the Basic Linear Algebra Subprograms (BLAS) APIs [60]. It is based on three fundamental approaches for computer-aided optimization that are part of Automated Empirical Optimization of Software (AEOS):

- Parameterization: functions have high-level parameters the space of which can be explored in order to be used for different optimization factors such as tile (or blocking) size.

- Multiple implementation: functions can be implemented in different equivalent approaches that yield distinct performances.
- Code generation: code generators (or compilers) incorporate knowledge about the systems on which the programs they produce are executed, which leads to various system-related parameters that impact the performance.

During an initial configuration step, ATLAS determines (through an exhaustive search) the optimal parameters to use with the best performing implementation on a particular platform using a specific compiler for BLAS.

ATCC

The Automatically Tuned Collective-Communications (ATCC) [75, 84] optimizes collective-communication operations as defined in the MPI specification. It conducts experiments on a given platform involving a set of predefined algorithms for collective-communication operations. In a configuration step, ATCC considers different aspects of the platform (network and communication layers, memory model, etc.), and selects the algorithms that perform best for each collective-communication operation. Likewise, it chooses the optimal buffer size for a given message size according to the same platform.

PEAK

PEAK [71] is a compile-time auto-tuning tool that aims at reducing the cost of exponential optimization search spaces by trading-off tuning speed and tuned program performance. PEAK uses the feedback-directed optimization-orchestration technique

as follows: it applies combinations of different code optimizations through an algorithm called *Combined Elimination* that explores the optimization space, partially executes the optimized code in order to evaluate its performance, and finally identifies the sections of the program that can be candidate for auto-tuning. Afterwards, it instruments the source code with an auto-tuning driver that rates the versions generated under the previous optimization combinations based on execution times until it finds the best combination.

AutoTune

AutoTune [41] is an auto-tuning project based on the Periscope performance analysis tool [9]. It consists of a framework called Periscope Tuning Framework (PTF) that uses Periscope to automate the tuning process of applications. The framework identifies recommendations for later production runs of an application through experimental runs. PTF includes automated performance analysis strategies for various different parallel programming models. The strategies are based on formalized performance properties specifying typical performance bottlenecks, the metrics required for their detection, as well as the severity of the bottlenecks. Auto-tuning in PTF involves the exploration of tuning parameter spaces (e.g. compiler flags, MPI runtime parameters, etc.) to run performance analysis strategies first and use codified expert knowledge to shrink the tuning space based on the resulting performance properties.

Model-based Auto-tuning

Model-based auto-tuning involves the use of mathematical formulas to calculate the performance of different algorithms. The performance is therefore predicted instead

of being evaluated experimentally. One basic formula to estimate the time it takes to send a message through the network is $L + \frac{m}{B}$ (also called Hockney's model [45]), where:

- L is the latency of the network, i.e. the time it takes for a single byte to travel the network.
- B is the bandwidth of the network, i.e. how many bytes per second can travel the network.
- m is the message length (in bytes).

Congestion can not be modeled using this model. Another model, the LogP model [23], describes a network in terms of latency, L, overhead, o, gap per message, g, and number of nodes involved in communication, P. The time to send a message between two nodes according to LogP model is $L + 2o$. LogP assumes that only constant-size, small messages are communicated between the nodes. In this model, the network allows transmission of at most bL/gc messages simultaneously.

LogGP [3] is an extension of the LogP model that additionally allows for large messages by introducing the gap per byte parameter, G. LogGP model predicts the time to send a message of size m between two nodes as $L + 2o + (m1)G$. In both LogP and LogGP model, the sender is able to initiate a new message after time g.

Stencil Auto-tuning

Stencil codes [77] are commonly used in scientific computing, notably in solving partial differential equations, image processing, etc. Jacobi kernels and Gauss-Seidel

kernels are examples of stencil codes. In general, stencil codes are nested loops that update multidimensional arrays multiple times. At each time, all array elements (except boundaries) are updated using neighboring elements in a fixed pattern (called stencil). This regularity of data accesses makes stencil codes suitable for SIMD architectures (since the same instructions are applied to different array elements). Kaushik et al. [26] presented a auto-tuning technique for stencil codes. Based on a pool of parameter-based optimizations (hierarchical blocking, prefetching, etc.), they use a code generator that produced optimized multi-threaded code variants to evaluate the performance of these optimizations on different platforms. A benchmark was then used to determine the optimal parameters for each optimization on a given platform through a combination of explicit search for global maxima and heuristics for constraining the search space.

Limitations of Static-tuning

These projects face a number of fundamental drawbacks. Most notable, the tuning procedure itself can be fairly time consuming and will still only cover a subset of the relevant parameter space with respect to the number of processes or message lengths being used by applications. Consequently, significant additional time and energy costs are spent on tuning these libraries instead of performing useful work on solving actual compute-intensive problems. For example, statically tuning MPI collective operations on thousands of nodes is a colossal job considering the amount of factors and parameters that have to be taken into account in order to tune MPI efficiently. Furthermore, several features influencing the performance of the application can only be determined while executing the application itself, since the performance

depends on factors such as process placement [29], resource utilization (e.g. multiple parallel jobs running simultaneously) and application characteristics (e.g. message sizes depending on input data).

Model-based auto-tuning projects, although using sophisticated formulas to predict collective-communication performances, are fundamentally limited. First of all, since they are simplifications of how machines perform operations in real-life, they do not include the intrinsic hazards of hardware and software platforms (e.g. operating system jitters leading to a slow-down of a subset of processes utilized by the parallel job [74]), and therefore do not hold for atypical scenarios. Secondly, it is not always possible to determine some parameters of communication models. As an example, there is no approach, as of today, that can reasonably estimate the receive overhead in the LogGP model [48]. Lastly, it is hard to model complex scenarios involving irregular sequences of computation and communication.

2.1.2 Dynamic Auto-tuning

Among the projects using run-time tuning techniques in HPC are FFTW [36] and PHiPAC [13], each focusing on one particular operations (Fast Fourier Transform and Matrix Multiply, respectively). STAR-MPI [31] provides run-time tuning of collective-communication operations using an API similar to MPI. There are also a number of generic auto-tuning frameworks such as OpenTuner [4] and Active Harmony [83].

FFTW

The FFTW library [36] optimizes sequential and parallel Fast Fourier Transform (FFT) operations. To compute an FFT, the application invokes first a 'planner' step specifying the FFT problem size to be solved. Depending on an argument passed by the application to the planner routine, the library measures the actual run time of many different implementations and selects the fastest one. Since the tuning procedure is in a separate step and initiated just once by the user, the library does not perform any useful work and removes the possibility of re-initiating the decision procedure in case of changing conditions. In fact, by making the run-time optimization upfront in the planner step, the library can not restart the selection logic in case significant deviations from the original performance, due to changing network conditions for example, have been observed. FFTW includes a feature called *Wisdom* that implements the notion of historic learning. It is achieved by gathering data from previous executions into a file and using them at future executions.

However, the wisdom concept in FFTW can be applied only between problems of the same size, i.e. it does not consider the notion of similarity between problems of different sizes. Also, it does not take into account the characteristics of the platforms on which the historic knowledge has been collected, which can lead to inaccurate outcome of the planner step.

PHiPAC

The Portable High Performance Ansi C (PHiPAC) [13] is a methodology that optimizes, among others, BLAS matrix-matrix operations. PHiPAC circumvents the

effort of manually tuning software performance by producing parameterized code generators. The parameters are tied to the machine performance. Code is generated based on a pool of coding suggestions such as loop unrolling, explicit removal of unnecessary dependencies in code blocks, and the use of machine-effective C constructs. For a given code generator, search scripts are also designed to find the best set of parameters for a given architecture/compiler.

However, the dynamic tuning aspect of PHiPAC includes only adjustment of cache blocking size according to various criteria for the matrix-multiply code generator. Most of the tuning procedure is done offline, taking a long time to find optimal parameters for code generators using naive search algorithms. Moreover, the best algorithm may strongly depend on the inputs. For example, for sparse matrix operations, the best algorithm depends on the sparsity pattern of the matrix but must be done in part at run-time, when the user's input data is available. Sometimes the tuning space is so large that it is even impractical to tune it exhaustively offline. This requires clever machine learning techniques to search faster. Finally, sometimes the search space includes changes to the software, and changes to the hardware as well, e.g. when one is trying to build specialized hardware to solve a problem, making the search space larger still.

ADAPT

A vast body of research for code optimizations is available in the field of compilers [5, 19, 27, 70, 85]. As a matter of fact, static-program optimizations are not efficient because information about the program input and the target architecture are not always known at compile-time. Therefore, a solution with dynamic program

optimization is necessary. Similarly to PHiPAC, ADAPT [86] is an auto-tuning solution that applies, dynamically however, optimizations such as loop distribution, loop unrolling, loop tiling, and other compiler optimizations to some code intervals. The most important code intervals (called hot-spots) are replaced by if-then-else statements to decide, at run-time, whether to use default code sections or automatically generated and optimized variants of code sequences. The selection of the best performing code variant is performed by a combination of pruning and sampling mechanisms. ADAPT includes a feature that invokes code generation and compilation of more code variants while the execution is ongoing. The overhead of code generation and compilation is hidden because they are performed concurrently with the execution. The resulting code variants can then be dynamically loaded and used within the same execution.

The main drawback of ADAPT is that it is applicable only to sequential and threaded applications and does not support optimization of applications running on distributed-memory architectures.

SALSA

The Self-Adapting Large-scale Solver Architecture (SALSA) [28] is a linear and non-linear system solver auto-tuner. Using the characteristics of the matrix of the system, SALSA estimates the best solver to use based on a knowledge data base. Among the characteristics that SALSA uses for choosing the right solver are structural properties of the matrix such as the maximum and minimum number of non-zero elements per row, matrix norms (e.g. Frobenius-norm), and spectral properties. The decision process is carried out by means of machine learning algorithms (such as boosting

algorithms and alternating decision trees [12]), and has been trained using a large set of matrices from various application domains. The decision algorithm is capable of handling missing features for the prediction, for example, when some norms are considered too expensive to be computed at run-time.

One of the disadvantages of SALSA is that the majority of the relevant matrix properties, such as conditioning number, and eigenvalues, are usually very compute-intensive, and can themselves generate another complexity level of optimization and auto-tuning; unless if such calculations are performed off-line. This would not be possible if the matrix is sparse and available at run-time only. Another notable disadvantage is that, even though a large number of simulations on different computer platforms/architectures have been used to supply the predictive system with accurate description of the relationship between matrix properties and solver performance, the resulting predictive system is still static and must be updated/retrained with more recent examples/matrices in order to stay relevant and to incorporate new data (for new matrices, cases, models, for new computer architectures, etc.), as well as algorithmic improvements and implementation enhancements to the linear solver.

IPA

In contrast to SALSA, the Intelligent Performance Assistant (IPA) [67] is a fully dynamic tool to assist the optimization of linear (only) solvers by tuning their parameters during execution. It starts with a large set of parameters based on available solver options and finds the best combinations. The parameters are continuously adapted during the execution using run-time performance measurements (e.g. execution time) and a genetic algorithm to guide the search.

The main advantage of IPA is that the auto-tuning is performed in conjunction with actual execution of the linear solver. Therefore, the run-time selection of the parameters of the solver contributes towards the solution of the linear system. Yet, this project faces a critical limitation: the choice of the solver is arbitrary and is done by the end-user. The end-user has to explicitly run IPA-tuned solvers individually in order to determine the best solver (in other words, IPA does not perform inter-solver but only intra-solver optimization).

OSKI

The Optimized Sparse Kernel Interface (OSKI) [87] is a collection of low-level primitives that provide automatically tuned computational kernels on sparse matrices, for use by solver libraries and applications. The kernels include sparse matrix-vector multiply (SpMV) and sparse triangular solve (SpTS), among others. Tuning in this project refers to the process of selecting the data structure and code transformations that lead to the fastest implementation of a kernel, given a machine and matrix. Tuning is performed at run-time since the matrix may be unknown until then. The interesting aspect of OSKI is that it can be user guided by reflecting the need for and cost of run-time tuning. The user explicitly triggers the tuning procedure at run-time when needed, or if the cost of tuning can be amortized by application characteristics known only at the user-level. It also supports self-profiling, which allows a transparent monitoring of all operations performed on a given matrix, and then use this information in deciding how aggressively to tune. Self-profiling enables the OSKI library to guess whether tuning will be profitable. Furthermore, OSKI allows for user inspection and control of the tuning process. To help the user reduce the

cost of tuning, it provides two mechanisms that allow both to guide and to see the results of the tuning process. First, the user may provide explicit hints about the workload (e.g., the number of SpMVs) and the kind of structure they believe the matrix possesses (e.g., uniform blocks of size $3 * 3$, or diagonals). Second, the user may retrieve string-based summaries of what tuning transformations and other performance optimizations have been applied to a given matrix. Thus, a user may see and save these results for re-application on future problems (matrices) which they believe have similar structure to a previously tuned matrix. Moreover, a user may select and apply transformations manually.

However, relying on the user guidance in many aspects of the auto-tuning procedure can be counterproductive. In fact, if the user does not have enough expertise and understanding of both the objective application and the OSKI library itself, then they can not determine the locations within the code sequence where the tuning procedure can be invoked efficiently. The outcome performance could therefore be penalized instead. Moreover, OSKI devolves the task of historic learning to the user. The user has to manually keep track of formerly tuned problems' characteristics and their corresponding tuning effectiveness, then apply unsophisticated similarity assessment between these problems and future problems in order to enhance the auto-tuning procedure.

STAR-MPI

STAR-MPI [31] incorporates run-time optimization of collective-communication operations providing a similar API as defined in the MPI specifications [64, 65]. Each operation has a repository of algorithms available. Using an empirical approach, the

library performs dynamic tuning of each collective operation by determining the performance of all available algorithms in the repository and selects the most efficient one. STAR-MPI tunes different instances for each operation separately. In order to achieve this goal, the prototypes of the STAR-MPI collective operations have been extended by an additional argument, namely an integer value uniquely identifying each call-site. This is however hidden from applications by using pre-processor directives to redirect the MPI calls to STAR-MPI. In order to minimize the overhead during the empirical testing phase of different implementations, STAR-MPI introduces the notion of 'grouping' algorithms. Initially, the library compares a single implementation from each group. After the winner group has been determined, the library evaluates all other available implementations within the winner group.

Hartmann et al. [43] presented a similar project that consisted of an adaptive extension library on top of MPI to optimize the performance of specific collective-communication operations in two phases. The first phase, called the configuration phase, determined for each collective operation the implementation leading to the best performance improvement for a given execution environment (hardware platform and MPI implementation). It is basically a static step which is performed only once for the given execution environment before running any real application. The second phase is the dynamic component that is activated for each of the collective MPI operations called by the application. The execution phase dynamically selects the most efficient implementation of the collective operation for the specified message size and the number of processors participating in the operation. Using the extension library, performance improvements can be achieved by an orthogonal organization of

the processes in 2D or 3D meshes and by decomposing the collective-communication operations into several consecutive phases of MPI communication.

Active Harmony

Active Harmony [83] is an automated run-time performance tuning system based on a server-client model used mainly for tuning the parameters of a cluster-based web server and some scientific simulations. The tuning system helps programs adapt themselves to the execution environment to achieve better performance. It includes two main features [21]: first, it has the capability to prioritize the parameters to be tuned to focus on those that are critical for the performance. However, this parameter prioritization is done using a stand alone tool, not while running and tuning the application. Moreover, the algorithm used there is very simplistic and based on the assumption of independent parameters. The authors suggest to use an experimental factorial design algorithm [21] as an alternative when the assumption is not verified. Second, it uses the experience learned in the previous program executions to speed up the tuning procedure. Similarly, the algorithm used is also simple and based on a proximity measure using the Euclidean distance. It starts by characterizing the current problem and then searches for the closest one in the history data available. If a similar configuration is not found in the history data, active harmony will start a regular tuning procedure based on a Nelder-Mead simplex algorithm.

ADCL

The Abstract Data and Communnication Library (ADCL) [38, 40] is an auto-tuning library for collective-communication operations. Since ADCL is the basis for much

of the work done in this dissertation, it is described in more detail than the previous projects to clearly define the starting point of this work.

ADCL provides for each supported operation a large number of implementations. Transparent to the user, the library incorporates a run-time selection logic in order to choose the implementation leading to the highest performance. For this, ADCL switches the implementation used internally during the first iterations of the application in order to determine the fastest version for the current scenario. After each/some implementation(s) (depending on the selection logic) have been tested a specified number of times, the run-time selection logic chooses the implementation leading to the shortest overall execution time, and uses this version for the rest of the execution.

A key concept of ADCL is its ability to select the fastest of the available implementations for a given communication pattern during the regular execution of the application. From the conceptual perspective, ADCL takes advantage of two characteristics of most scientific applications, iterative and collective executions. Iterative execution refers to the fact that most parallel, scientific applications are centered around a large loop, and execute therefore the same code sequence over and over again. Consider for example an application which solves a time-dependent partial differential equation (PDE). These problems are often solved by discretizing the PDE in space and time, and by solving the resulting system of linear equations for each time step. Depending on the application, iteration counts can reach six digit numbers. Collective execution on the other hand refers to the fact that most large scale parallel applications are based on data decomposition, i.e. all processes execute the

same code sequence on different data items. Processes are typically also synchronized, i.e. all processes are in the same loop iteration. This synchronization is often required for numerical reasons and is enforced by communication operations.

Among the communication operations currently supported by ADCL are the most widely used collective operations, such as Cartesian neighborhood communication, All-gather, All-to-all, and All-reduce operations. In addition, ADCL extends the concept of persistent point-to-point operations supported by the MPI specification to collective operations. Collective operations can be referenced can be referenced by an `ADCL_Request` handle. The actual communication operation can be initiated using `ADCL_Request_start` for blocking execution or `ADCL_Request_init`, and `ADCL_Request_wait` for non-blocking operations. ADCL also provides low-level interfaces which can be used by end-user application to its own functionality for auto-tuning using ADCL and thus make use of the ADCL selection logic, statistical filtering, etc.

ADCL incorporates multiple run-time selection algorithms. A brute force search strategy evaluates all available implementations before choosing the implementation that leads to the best performance. While this approach guarantees to find the best performing implementation on a given platform, it has the drawback that testing all implementations can take a significant amount of time. In order to speed up the selection logic, an alternative run-time heuristic based on attributes characterizing an implementation has been developed [40]. The heuristic is based on the assumption that the fastest implementation for a given problem size on a given platform is also the implementation having optimal values for the attributes. Therefore, the

algorithm determines the optimal value for each attribute used to characterize an implementation. Once the optimal value for an attribute has been found, the library removes all implementations not having the required value for the corresponding attribute and thus shrinks the list of available implementations. The third selection algorithm is based on the 2^k factorial design algorithm [17]. This version of the selection logic also operates on the attributes characterizing the implementations. However, it supports the pruning of the search space for correlated parameters, while the heuristic presented above implicitly assumes that the attributes are not correlated. The 2^k factorial design-based run-time selection is however mostly useful for very large parameter spaces such as described in [20].

Independently of the version selection approach used by ADCL, the collective decision logic of ADCL has to compare performance data of multiple functions gathered on different processes. In the most general case, processes only have access to their own performance data, and performance data for the same code version might differ significantly across multiple processes. Distributing the performance data of all processes, for all versions, to other processes is not feasible since the costs for communicating these large volumes of data would often offset the performance benefits achieved by run-time tuning. Therefore, the approach taken by ADCL relies on data reduction, i.e. each process provides only a single value for each alternative version of the code. First, it locally filters the measurements by detecting outliers in order to obtain reliable estimates of local average execution times within each process. Afterwards, the library executes a global reduction operation that determines the maximum of: raw average execution times, filtered average execution times, and

proportions of outliers. The reduction is motivated by a fundamental law in parallel computing, which states that the performance of an application is determined by the slowest process. Finally, considering whether the number of outliers exceeded a certain threshold or not, the library estimates the maximum overall execution time of each version [10].

In contrast to the other dynamic-tuning libraries, ADCL integrates the runtime selection logic into the regular execution of the applications, and thus does not waste any compute resources. ADCL also contributes towards the solution of the application and extends the concept of process grouping (of STAR-MPI for example) by introducing the formal notion of attributes. This allows grouping of implementations/code versions based on more than just one criterion. ADCL is a general framework that can be applied for a wide variety of operations compared to PHiPAC, ADAPT, SALSA, and IPA, which target specific types of problems and/or software/hardware platforms. In contrast with FFTW and Active Harmony, ADCL incorporates alternative selection algorithms to choose the best performing implementation, and applies sophisticated heuristics that take into consideration the dependencies between attributes. Finally, ADCL includes a historic learning feature that automatically takes advantage of previously optimized problems (as opposed to OSKI where it is done explicitly by the end-user).

Given the advantages of ADCL, the work presented in this dissertation will be based on this library and integrated to it, in order to address the challenges listed before.

2.2 Collective Operations

As the gap in performance between the processors and the memory systems continue to grow in high-performance computing, the communication component of an application will dictate the overall application performance and scalability. Therefore, it is useful to abstract often occurring communication operations across cores as collective-communication operations. Collective-communication operations separate desired data movement from actual implementation and allow for numerous optimizations internally at algorithmic, software, and hardware levels. They also simplify code maintenance and readability and reduce communication costs compared to (trivial) algorithms often used by applications that do not use collective operations. Yet, their performance cost are not negligible and are at the heart of the high-performance computing research field. Significant efforts have been devoted to the development of new algorithms to improve the performance of collective operations, including non-blocking-collective operations as follows.

2.2.1 Blocking-Collective Operations

In the last couple of years, important research on optimizing blocking collective-communication operations has been done; especially at the algorithmic level. Nishtala et al. [68, 69] focused on auto-tuning blocking collective-communication operations on multi-core systems. In particular, they highlight how loosening the synchronization semantics of a collective can lead to a more efficient use of the memory system. The most common algorithmic representation of collective operations is trees

that include synchronization between their pairs of communicating nodes. The actual sequence of communication operations in a collective operation can therefore be implemented as various types of trees (binary, binomial, etc.), that thereby involve performance specific synchronization schemes. Nishtala et al. [68, 69] exposed the synchronization requirements for a collective operation in the interface so the collective operation and the synchronization can be optimized together. They selected, among different combinations of algorithms (trees) and synchronization schemes, the one that provides the best performance for a given collective operation.

Sack et al. [78] optimized blocking-collective algorithms through non-minimal communication. Minimal communication, as opposed to non-minimal communication, refers to collective algorithms where no process sends or receives more than the minimum amount of data, such as all-gather and reduce-scatter collective algorithms. With the addition of a small amount of redundant communication, the authors performed non-minimal topology-aware reciprocal algorithms (topology-aware means that the physical distance and latency between processes within the network is taken into account at higher levels for performance purposes). Their algorithms were non-minimal, which adds a flexibility to reorder the stages of communication to run multiple operations in parallel on a multi-port network. This therefore decreased the overall execution time of the collective operation. An extra stage of communication with a relatively small overhead was used to restore the correct data order. They claimed that collective algorithms need not preserve the correct ordering as long as the misordering is universal and the correct ordering is restored at the end.

Thakur and Gropp [82] presented an approach to improve the performance of

blocking collective-communication operations for clusters connected by switched networks. The approach used multiple algorithms for each collective operation depending on the message length and number of processes, with the goal of minimizing latency for short messages and minimizing bandwidth for long messages. The cost of the collective-communication algorithms was estimated in terms of latency and bandwidth based on a model similar to Hockney's (described previously) and Van de Geijn [79]. Using this model, the message length and process count cutoff points for switching between algorithms were determined. This involves experimental evaluation of the model's parameters (i.e. latency and bandwidth), considering they may be different for different networks/machines. Also, the authors assumed flat communication models, i.e. the communication cost of any pair of processes is the same.

Similarly, Li et al. [61] optimized blocking collective-communication operations based on performance models for clusters with NUMA [63] nodes. The authors make use of shared memory for intra-node communication, since direct memory access reduces buffer copying and system resource overhead, and provides faster synchronization between threads. Several algorithms for collective-communication operations were developed by the authors, and dominate depending on the message length. The authors focused on the all-reduce collective operation by designing a hierarchy-aware algorithms to reduce inter-socket data transfer, utilize shared cache levels to reduce data transfer latency, and adopt nested loop optimizations (such as strip mining) in order to improve the cache efficiency when the data size exceeds the

capacity of the cache. The authors demonstrated how different algorithms of all-reduce provide the best performances depending on the message length; tree-based all-reduce is best for short message lengths while tiled-reduce followed by a broadcast is best for long message lengths.

In general, despite the considerable efforts that have been spent on optimizing blocking collective-communication operations, the latter face a fundamental drawback: a blocking-collective operation is finished on a given process as soon as its part of the overall communication is done and the communication buffer can be accessed. This does not indicate that other processes have completed, or for that matter even started the collective operation. Since most algorithms introduce synchronization due to data dependencies (it is obvious that every process has to wait for the root process in a broadcast operation), the application waiting in blocking-collective calls results from this pseudo-synchronization (or process-skewing) effect [54, 73]. This effect is inherent in blocking-collective operations and can not be avoided, the reason being that common sources of process-skewing and load imbalance are not easily measurable, because they result (partly) from system noise as demonstrated theoretically [2] and practically [55, 73].

2.2.2 Non-Blocking-Collective Operations

Non-blocking-collective operations have been recently standardized by the Message Passing Interface (MPI) Forum. However, efficient designs offered by the MPI communication run-times are likely to be the key factors that drive their adoption. Non-blocking-collective operations perform the pseudo-synchronizing collective operation

in the background and allow some limited asynchronism and load imbalance between processes compared to blocking-collective operations. Yet, properly applying non-blocking-collective operations to real-world applications turn out to be non-trivial because the code must often be restructured significantly to take full advantage of non-blocking-collective operations.

LibNBC

LibNBC [50] is a software-based implementation of a non-blocking interface for MPI collective operations. Based on non-blocking point-to-point operations, LibNBC is used to overlap collective-communication patterns with computation and enables the user to move communication and the necessary synchronization in the background and use parts of the original communication time to perform useful computation. The central concept in LibNBC's design is the collective operation schedule. During initialization of the operation, each process records its part of the collective operation in a local schedule. A schedule contains, among others, non-blocking send and receive operations and a local synchronization object referred to as a 'barrier'. A barrier in a schedule has the semantics that all operations before the barrier have to be finished before any of the operations after the barrier can be started. The execution of a schedule is non-blocking and the state of the operation is kept as a pointer to a position in the schedule. With send, receive, and barrier, one can express many collective-communication operations.

The progression of non-blocking collective-communication operations in LibNBC is twofold [49]: either spawning a separate thread for each process to execute the non-blocking operation in a blocking manner in the background, which scales poorly;

or invoking the LibNBC progress/completion functions (`NBC_Test`/`NBC_Wait`) at the user-level.

Because of its portability and flexibility, LibNBC will be used as a framework for the implementation part of non-blocking collective-communication operations in the work presented in this dissertation.

Thread-based Approaches

In order to maximize the overlap of communication and computation, many thread-based solutions for non-blocking-collective operations have been proposed [47, 49, 56]. They rely on spare cores in each node to progress non-blocking-collectives. In [56], the solution relied on only one thread per node to execute the non-blocking-collective operations on behalf of the application processes. However, these designs required additional memory resources, and involved expensive copy operation between the processes and the threads dedicated to perform the non-blocking-collective operation. Such factors limit the overall performance and scalability benefits associated with using non-blocking-collectives in MPI. Moreover, the processes that share the compute resources with the dedicated threads are therefore delayed, which leads to an overall load imbalance. The load imbalance is even higher on clusters that run a large number of processes per node, where the task-list of the non-blocking-collective operation per node becomes larger.

Hardware-based Approaches

Kandalla et. al [57–59] and Inozemtsev et al. [51] focused on hiding the latency of collective operations by offloading them to the networking hardware (mostly InfiniBand)

in order to overlap communication and computation using independent progression of communication. However, since these methods required network hardware support for asynchronous communication, their portability was restricted. They also currently have several performance and scalability limitations. The size of the memory and the processing speed of the network hardware support were generally limited compared to those of CPUs. If the task-list was too large, delegating the progression of non-blocking-collective operations to the network hardware support could lead to memory overhead as demonstrated in Inozemtsev et al. [51]. The initial stage of the offloading process was blocking at the process level, and each particular collective operation in a specific communicator required separate memory resources. Consequently, hardware-based approaches do not scale with the increase of number of concurrent non-blocking operations. This is due to applications involving multiple concurrent non-blocking communication operations, communication-intensive non-blocking-collective operations that entail default large amounts of non-blocking point-to-point operations, very large messages communicated in short segments, large process counts, or a combination of these factors.

Optimization of Non-Blocking-Collective Operations

Song et al. [80] performed tuning of 3D Fast Fourier Transforms using non-blocking all-to-all collective-communication operation. Their work, however, focused on tuning the parameters of the 3D Fast Fourier Transform in order to maximize the overlap of communication and computation. Their auto-tuning approach was limited to the context of 3D Fast Fourier Transform and did not auto-tune the non-blocking all-to-all collective operation itself. The fundamental problem in the analysis of

non-blocking-collective operations stems from the fact, that it is virtually impossible to capture the 'visible' part of the operation. For the same reason, it is impractical to statically auto-tune non-blocking-collective operations. In other words, the proceedings of non-blocking-collective operations are irregular and therefore unpredictable, reason being they are shaped at the end-user level (they overlap arbitrary end-user computations, and progress through end-user function calls, like `MPI_Test`, as well). It is therefore impossible to produce a consistent model (LogP for instance) for non-blocking-collective operations, or to estimate their performance before actual execution. Optimizing non-blocking collective-communication operations therefore requires dynamic auto-tuning, and there is, to the best of our knowledge, no comprehensive study of tuning non-blocking collective-communication operations available as of today.

Chapter 3

Auto-tuning Non-blocking Collective-Communication Operations

As discussed in chapter 2, auto-tuning projects typically measure the execution time of alternative implementations of a collective operation and use the version leading to the lowest execution time. Extending this concept to auto-tune non-blocking-collective operations is challenging for two reasons. First, the actual time spent in the communication operation can not be directly measured because the communication operation is only partially visible from the application perspective. This violates a fundamental requirement of auto-tuning libraries, namely to have accurate and reproducible measurements of the alternative versions of an operation. Second, the performance of non-blocking operations is closely tied to their ability to ensure progress outside of the MPI library. In order to progress non-blocking operations, the

application has to regularly invoke the progress engine of the MPI library [49] if the MPI library does not utilize a separate thread for this purpose. Auto-tuning offers a unique opportunity to optimize the number and frequency of progress calls: too few progress calls will not achieve the desired overlapping between communication and compute operations; too many progress calls can lead to unnecessary overhead.

In ADCL terminology, a communication operation supported by ADCL is referred to as a function-set, while a particular implementation of the operation is a function. An ADCL function-set can contain an `attribute-set`, which is a collection of attributes, each attribute describes a particular characteristic of an implementation. Typical attributes found in existing function-sets are used to implement the operation (e.g. linear, binary tree), data transfer primitives used (e.g. blocking point-to-point, non-blocking point-to-point, one-sided get, one-sided put), or the method used to handle dis-contiguous data e.g. pack/unpack, derived data types. Note, that a large number of attributes and attribute values allow for a finer grained characterization of the available implementations, but will also increase the number of functions in the function-set and thus the duration of the initial tuning phase.

Besides that, high-level interfaces have been introduced in version 2.0 of the library for most collective operations. These interfaces represent persistent collective-communication operations (extending on the concept of persistent point-to-point operations supported by the MPI specification) and can be referenced by an `ADCL_Request` handle. The ADCL request structure stores internally a pointer to the current function used for evaluation. This minimizes the overhead introduced by ADCL, since in the vast majority of the cases it is sufficient to call the current function pointer,

and increment a counter to keep track of the number of times the function has been called. If a function has been executed a certain number of times, the ADCL logic internally switches the function pointer to the next function in the function-set. After all functions have been evaluated, the selection logic is activated to determine the fastest implementation.

In the following, the solution to the timing problem outlined above is discussed, and a description of the non-blocking function-sets in ADCL, along with the interface developed to tackle the progress problem, is given.

3.1 Timing of Non-blocking Operations

To solve the timing problem described above, non-blocking operations have to decouple the actual measurement of a communication operation from the function call. In ADCL, this has been achieved through the introduction of a *timer object*. An `ADCL_Timer` transforms the problem of tuning a collective operation into the tuning of a larger code section, which can include both communication and compute operations [11]. The user (or an automated tool, which is not discussed in this dissertation) can start and stop the timer at strategically important locations in the code, e.g. the beginning and the end of the main compute loop. The time between starting and stopping a timer will be stored as the execution time. The following code sample shows a non-blocking all-to-all operation using the ADCL syntax.

```
ADCL_Request req;
ADCL_Timer timer;

// Initialize non-blocking persistent
// collective operation
ADCL_Ialltoall_init ( sbuf, scount, sdat,
         rbuf, rcount, rdat, comm, &req);
// Associate request with a timer object
ADCL_Timer_create ( req, &timer);

//Main application loop
for (i=0; i<MAXIT; i++ ) {
   //Start timer
   ADCL_Timer_start (timer);
   ...
   // Start communication operation
   ADCL_Request_init (req);
   ...
   // Wait for completion
   ADCL_Request_wait (req);
   ...
   // Stop timer
   ADCL_Timer_end (timer);
}
```

Figure 3.1: Code sample using the ADCL High Level API.

3.2 Non-blocking Function-sets

Non-blocking operations separate the initiation and the completion of the operation, overlapping the communication operation with computation, I/O, or other communication operations. In the most generic case, ADCL will have to store two functions per operations, namely for initializing an operation (the `init` function) and for completion (the `wait` function). The completion operation might often be generic, e.g. the equivalent of `MPI_Wait`. Note, that conceptually the wait function pointer is simply set to NULL for blocking-collective operations. In fact, any blocking collective-communication operation could be represented technically as a non-blocking operation by setting the `init` function pointer only – a fact that is later used in the evaluation section – as long as it is guaranteed that the non-blocking-collective operations are restricted to non-overlapping sub-communicators, which otherwise could lead to a deadlock.

For non-blocking-collective operations, each function implementing a non-blocking operation is represented by a particular LibNBC schedule. For this, existing implementations of the `MPI_Bcast`, `MPI_Reduce`, `MPI_Allgather`, and `MPI_Alltoall` operations in Open MPI [37] are converted to a LibNBC schedule. Within the context of this chapter, the focus is on non-blocking broadcast and non-blocking all-to-all operations. For the Ibcast operation a description based on two attributes has been chosen: the fan-out parameter of the broadcast tree, and the segment size internally used by the implementation. The default function-set for Ibcast contains therefore 21 functions, with the fan-out value ranging from 1 (which is a chain algorithm) to 5

(i.e. each parent process has five children), a special value 0 used for the linear algorithm (effectively representing an infinite number of children), and a value of N used for the binomial algorithm. For each fan-out value, a function with segment sizes of 32 KB, 64 KB and 128 KB is provided, leading to the $7 \times 3 = 21$ implementations.

For the Ialltoall operation, three different algorithms have been implemented, namely the linear algorithm, a dissemination algorithm [18], and a pair-wise exchange algorithm. Further distinction based on data transfer primitives (i.e. Put/Get vs. Isend/Irecv) could be added later.

A newly introduced ADCL progress functions triggers the progress engine of the LibNBC library and the underlying MPI implementation. As discussed in [50], there are fundamentally two mechanisms on how to ensure that non-blocking communication operations are performed asynchronously. A library can either utilize a separate thread to ensure progress of the communication operations, or call regularly into the MPI library in a non-blocking manner to probe for pending operations and ensure the continuation of the communication operation. Although MPI libraries are currently in the process of improving their multi-threading support, most production-level MPI installation do not operate using a progress thread. Thus, calling the MPI library (e.g. using a function such as `MPI_Test`) is the most portable way to ensure progression of non-blocking operations in general. Using the ADCL progress function and the ADCL request handle associated with it, ADCL can trigger the LibNBC progress function, which calls the MPI library.

3.3 Performance Evaluation

The following section evaluates the impact of auto-tuning non-blocking collective-communication operations. First the execution environment is described, followed by the results obtained with micro-benchmarks and an application kernel based on a 3-D Fast Fourier Transform (FFT).

3.3.1 Experimental Platforms

Two clusters have been used in the subsequent tests, both located at the University of Houston. The first is the *crill* cluster, which consists of 16 nodes with four 12-core AMD Opteron (Magny Cours) processor cores each (48 cores per node, 768 cores total) and 64 GB of main memory per node. Each node further has two 4x DDR InfiniBand HCAs, providing similar bandwidth to 4xQDR InfiniBand for large messages and message striping.

The second cluster is the *whale* cluster, which consists of 64 nodes with two quad-core AMD Opteron (Barcelona) processor, 16 GB of main memory and a single DDR InfiniBand HCA per node. In addition, some tests were performed on this cluster using the Gigabit Ethernet network interconnect. The platform is referred to as *whale-tcp* in that case. The 1.6 series of Open MPI [37] was used in all instances.

3.3.2 Results using Micro-Benchmarks

For the first set of tests, a micro-benchmark has been developed which evaluates the ability to overlap collective-communication operations with compute operations. The benchmark executes a loop a configurable number of times, in each loop iteration

initiating the non-blocking collective-communication operation, executing a compute operation, and calling the completion function. The time spent in the compute operation is provided as an input argument to the benchmark, and is typically set to a value that is larger or equal to the costs of communication operation. Thus, the time observed in the benchmark should ideally be equal to the time spent in the computation, if the communication library is able to completely overlap the communication with computation. The benchmark further takes an argument for the number of times the ADCL progress function is invoked. The compute operation is split into equal chunks such that the time spent in one instance of the compute operation is equal to *compute_time_per_iteration* divided by *num_progress_calls*.

The initial set of tests focused on the correctness of the run-time selection logic in ADCL, and is referred to as verification runs. For this, the same benchmark scenario is executed using a single implementation of the `ADCL_Ibcast` or `ADCL_Ialltoall` function-set, circumventing the run-time selection logic of ADCL. In order to make the conditions as comparable as possible, the reference data was produced within the same batch scheduler allocation and had the same node assignments. During each verification run, the benchmark measures the time it takes to execute between 1000 and $10,000$ iterations, depending on the problem size. Tests have been executed for 32, 128 and 256 processes, (the latter on *crill* only) for 1 KB and 2 MB message length for `ADCL_Ibcast` and 1 KB and 128 KB per process-pair for `ADCL_Ialltoall`.

Figure 3.2 shows two representative verification runs obtained for `ADCL_Ialltoall` on *whale* using 128 processes with 128 KB message length per process pair and 50 s compute time overall and on *crill* using 256 processes for the same message length

and compute time. The figures detail the result obtained for each implementation separately, as well as for ADCL using the brute-force search and the attributes based heuristic.

Furthermore, tests have been executed for various numbers of progress calls during the compute operation to advance the non-blocking-collective operation. Figure 3.3 show some results obtained for the IBcast verification runs, namely on the *crill* cluster using 256 processes with 2 MB message length and 50 s compute time, and on the *whale* cluster using 128 processes with 2 MB message length and 100 s compute time. Due to the large number of functions in the function-set, a slightly modified representation of the results is used for these test-cases. Rather than presenting the execution time obtained for every single implementation (which would be spread out), the minimum and maximum execution times obtained across the available implementations, and the execution time obtained for the same test cases using the two ADCL selection logics are presented instead.

To summarize the results obtained with the verification runs, ADCL chooses in the vast majority of the cases the correct function as the 'winner'. Within this context, the 'correct winner function' is defined as an implementation of the operation that achieves either the best performance for the test case when executed without the ADCL decision logic, or is very close to the best performance (within 5%). Especially for the Ibcast operation, which has 21 possible implementations available, there is typically a small group of functions delivering very similar performance. Based on this definition, the ADCL brute force search made in 90% of the test-cases the correct decisions out of the 324 verification runs that have been performed overall, and the

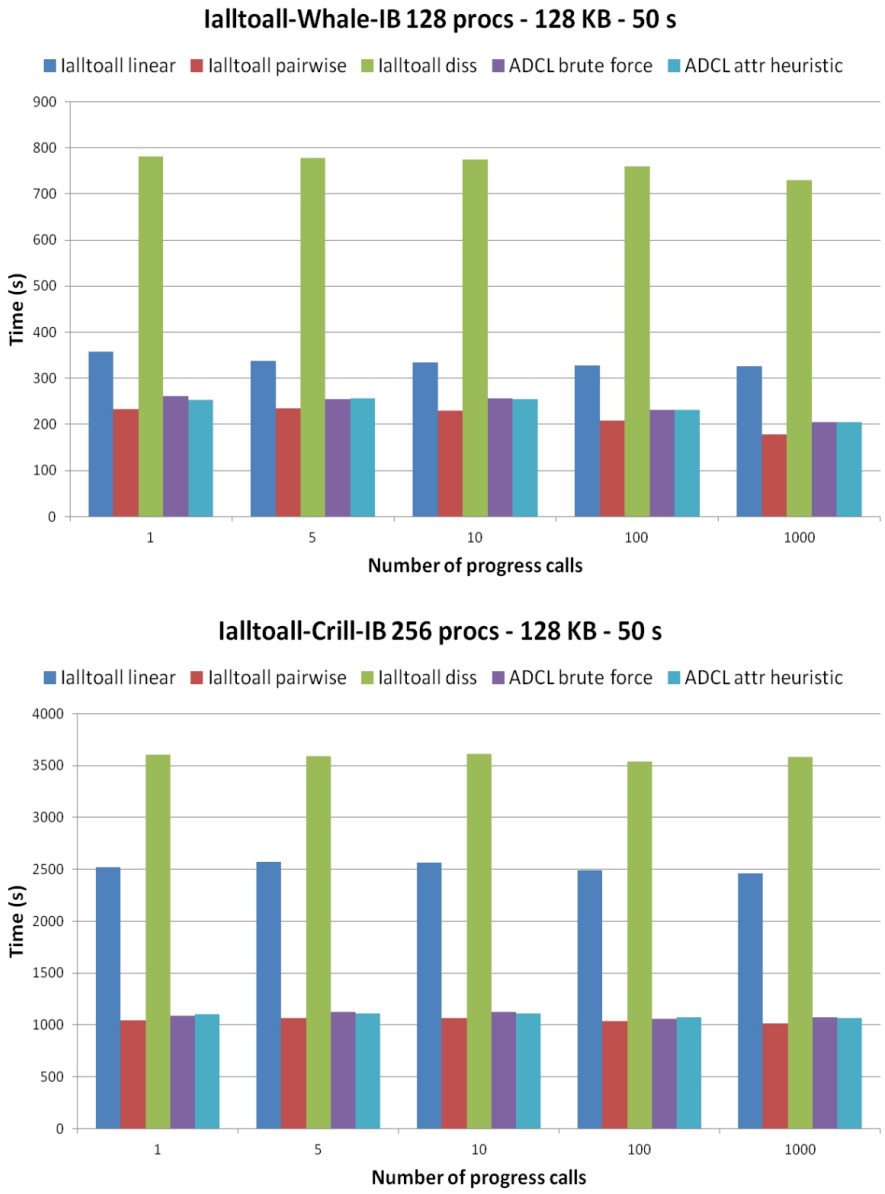

Figure 3.2: Execution time of the Ialltoall verification runs executed for 128 KB message length and 50 s compute time with 128 processes on *whale* (top) and 256 processes on *crill* (bottom).

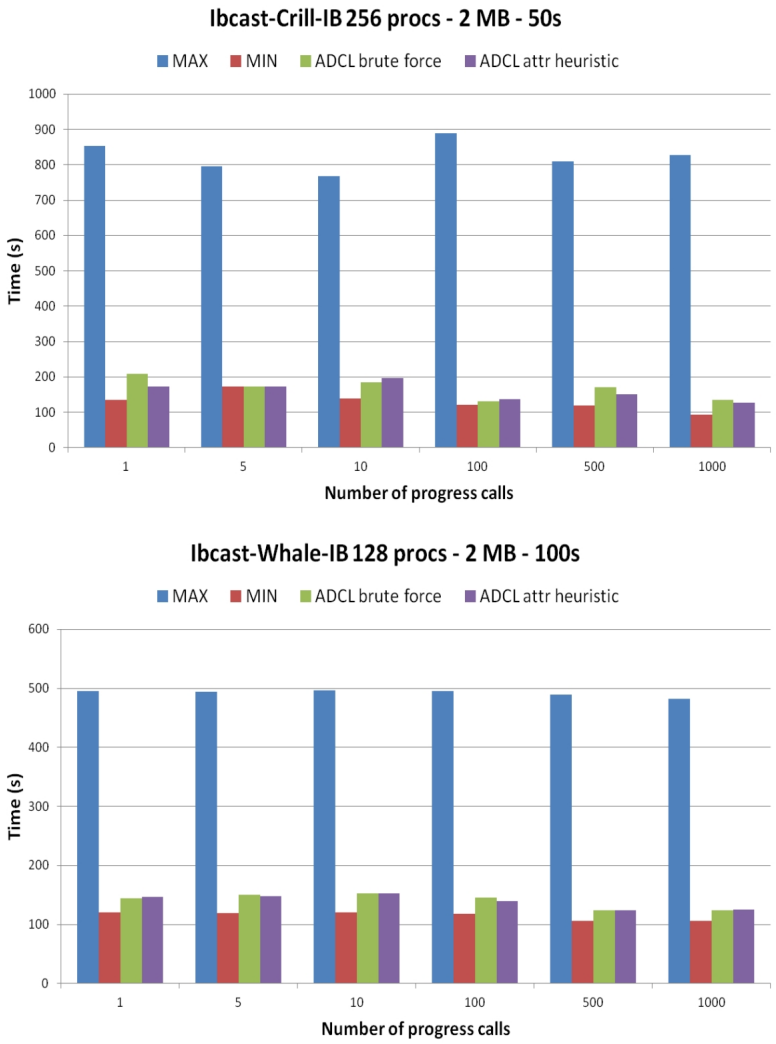

Figure 3.3: Execution time of the Ibcast verification runs executed on *crill* with 256 processes, 2 MB message length and 50 s compute time (top) and on *whale* with 128 processes, 2 MB message length and 100 s compute time (bottom).

attribute-based search heuristic in 92% of the test-cases. The few cases where ADCL made a suboptimal decision typically involved having a larger number of data outliers during the evaluation phase of ADCL, due to external influences from the Operating System or other applications.

The execution times obtained using ADCL were usually slightly higher than the minimum execution time obtained with a fixed implementation. The reason for this is that ADCL also has to evaluate during the learning phase implementations of the operation that turn out to be suboptimal, and introduced an overhead compared to using the 'optimal' implementation only. However, the additional costs of the learning phase are not relevant for long running applications. Ultimately, the results of the verification runs showed that the concepts and interfaces developed within this work tuned non-blocking-collective operations at run-time.

In the following, particular aspects of the tests were analyzed in order to understand the parameters influencing the performance of the non-blocking-collective operations.

Influence of the network characteristics The results shown in this paragraph highlight the influence of the network interconnect on the best performing implementation for the `ADCL_Ialltoall` operation. Figure 3.4 depicts the results obtained for tests executed with 32 processes, 128 KB message length per pair of processes, and 50 s computational time. However, the results shown in the bottom part of figure 3.4 are using the *whale* cluster with one DDR InfiniBand networking card, while the tests shown in the top part of figure 3.4 use the same cluster with Gigabit Ethernet network interconnect. The graphs highlight, that different implementations of the

Ialltoall operation show very different behavior for these two platforms, although all parameters, except for the network, are identical. Specifically, the implementation using the linear algorithm showed the best performance for the *whale* cluster in multiple instances (for 5, 10 and 100 progress calls), but did very poorly on *whale-tcp* and is typically the worst choice on that platform.

Influence of the communication volumes The set of tests shown in figure 3.5 demonstrate the influence of the message length on the optimal algorithm. These two figures show the results obtained on the *crill* cluster for 256 processes using 1 KB and 128 KB message length for 10 s of compute time. The dissemination algorithm is the best choice for the 1 KB message length, but the worst choice for the 128 KB scenario. On the other hand, the linear and pairwise algorithm perform poorly in the 1 KB scenario, but very well on the 128 KB scenario.

Influence of the number of processes used Figure 3.6 show the results obtained on the *whale* cluster for 1 KB message length and 10 s compute time. The difference between the two graphs is the number of processes used, namely 32 in the first case, and 128 in the second case. Once again, there was a large variability on which algorithm performed well depending on the number of processes used, with linear and pairwise performed poorly for 32 processes and very good for 128 processes on this platform, while the dissemination-algorithm based implementation of Ialltoall performed well for 32 processes but poorly for 128 processes.

Influence of the number of progress calls All results shown so far in figures 3.2 - 3.6 demonstrated a dependence of the overall execution time on the number

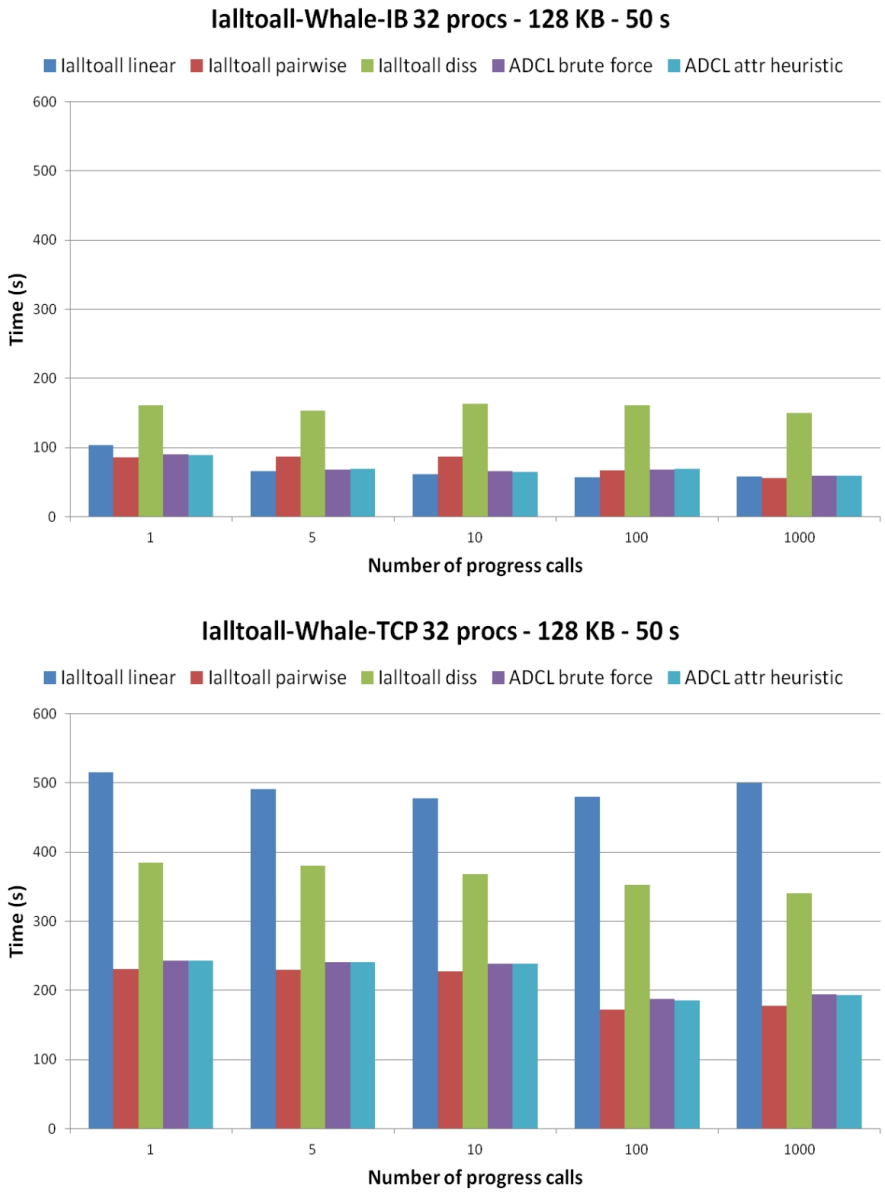

Figure 3.4: Comparison of the execution times of various implementations of the Ialltoall operation using 32 processes, 128 KB message length and 50 s compute time using the *whale* cluster (bottom) and *whale-tcp* (top).

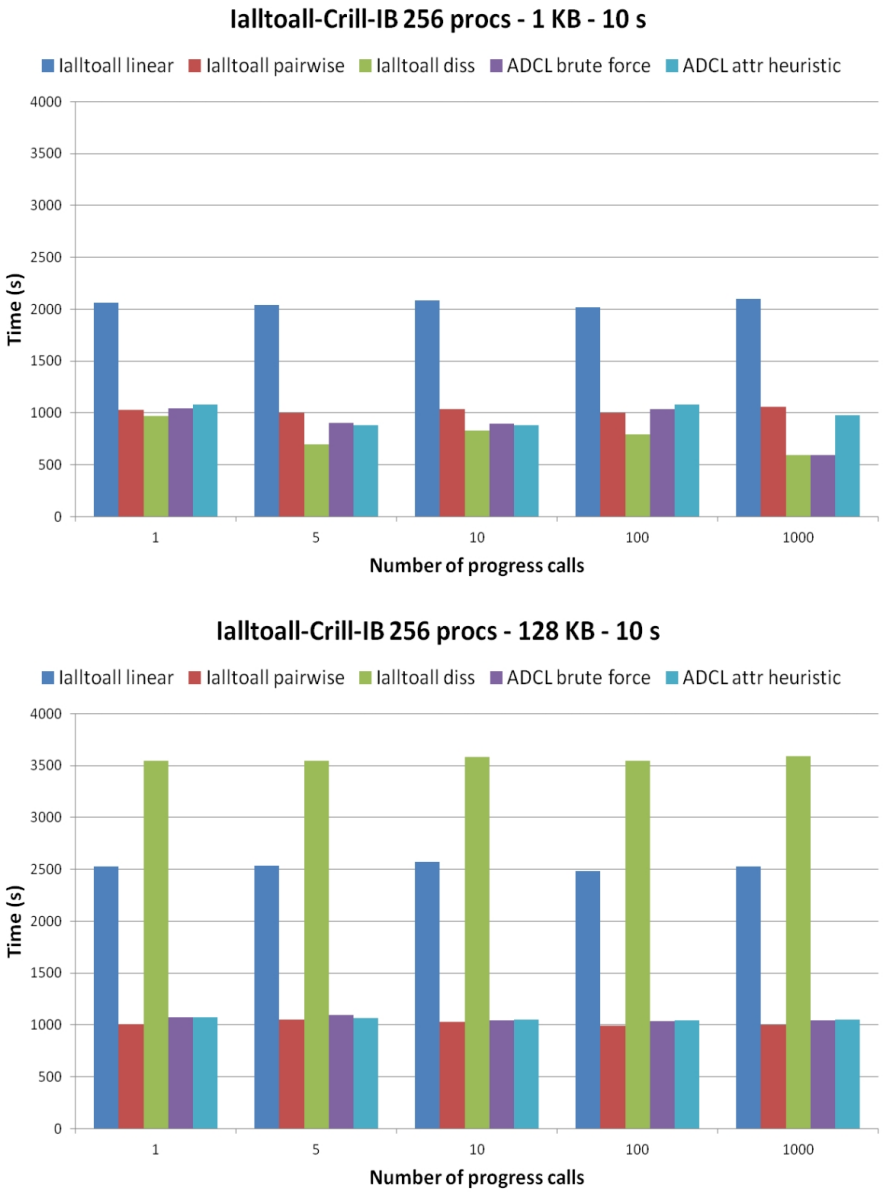

Figure 3.5: Comparison of the execution times of various implementations of the Ialltoall operation using 256 processes on the *crill* cluster, 10 s compute time for 1 KB message (bottom) and 128 KB messages (top).

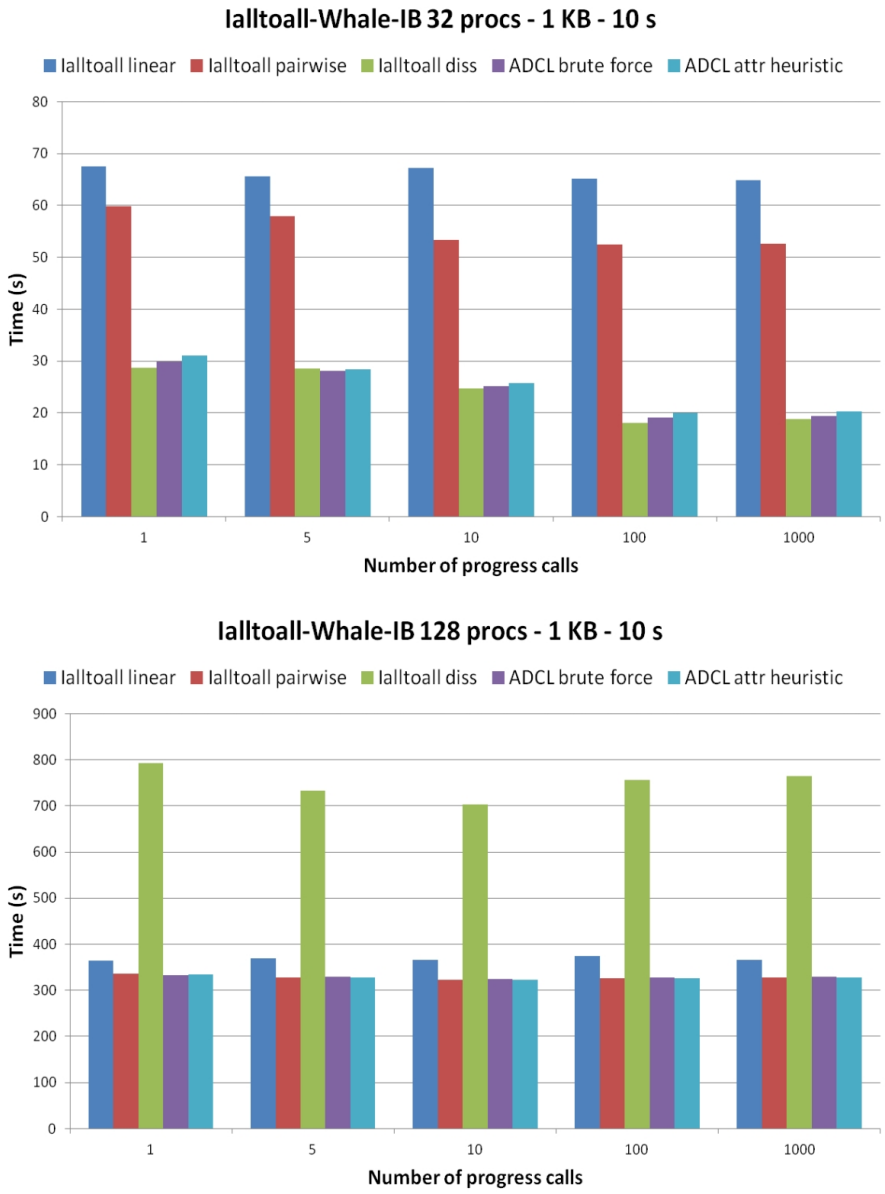

Figure 3.6: Comparison of the execution times of various implementations of the Ialltoall operation on the *whale* cluster, using 1 KB message length, 10 s compute time, 32 processes (bottom) and 128 processes (top).

of progress calls being made. Typically, the ability to overlap communication and computation increased with increasing numbers of progress calls. For many application scenarios, the number of progress calls that can be inserted is given by the code structure, e.g. such as in-between subsequent calls to a library function which is utilized as a black box. However, in case the entire code is under the control of the application developer, an arbitrary number of progress calls could be introduced into the code. Figure 3.7 demonstrates a scenario in which too many progress calls reduce the performance of the benchmark.

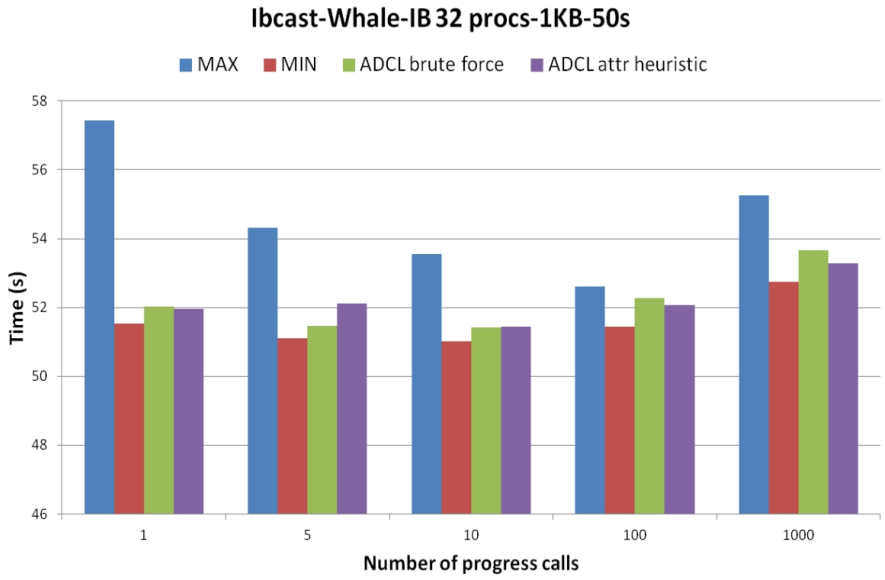

Figure 3.7: Influence of the number of progress calls on the execution time of the micro-benchmark for the Ibcast operation on the *whale* cluster using 32 processes, 1 KB message length and 50 s compute time.

Finally, one should also note that the number of progress calls also has influence on the algorithm leading the best performance. Some algorithms require fewer

progress calls due to the lower number of steps/messages involved, while algorithms splitting up the overall communication volume into more sub-steps, i.e. into more cycles in the LibNBC schedule, require typically a larger number of progress calls to overlap communication and computation. Figure 3.8 shows a scenario observed on the *crill* cluster, in which the pairwise algorithm delivers the best performance of the Ialltoall operation where only a single progress call could be inserted into the code sequence, while the linear algorithm does best if more than one progress call is executed.

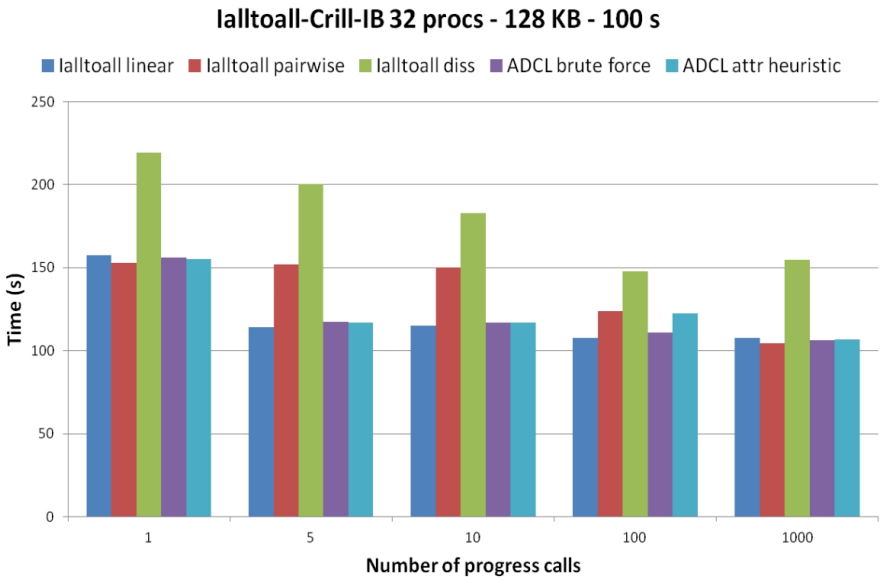

Figure 3.8: Influence of the number of progress calls on the optimal algorithm.

To summarize the analysis, run-time adaptation is a key to achieve good performance for non-blocking-collective operations due to the many factors influencing the choice of the optimal implementation, including network characteristics, number of

processes, communication volumes, and number of progress calls being made.

3.3.3 Results using an Application Kernel

In the following, the benefits of using run-time tuning for non-blocking-collective operations with an application kernel are presented, with the use of a three dimensional Fast Fourier Transform (FFT). The application benchmark was adopted by Hoefler et.al. [46] to utilize non-blocking-collective all-to-all operations, with multiple different versions available to implement the multi-dimensional FFT. Specifically, the sequence of calculations and communications can be implemented in a pipelined, tiled, windowed, and window-tiled manner. Since the three first implementations are special cases of the window-tiled implementation, this version explains the occurring operation. As depicted in figure 3.9, the computation and communication occurring the multi-dimensional FFT is subdivided into tiles in order to overlap communication and computation. Increasing the number of tiles used allows to coarsen the computational aspect of the multi-dimensional FFT, while windowing manages multiple outstanding communication operations.

The pipelined implementation has a window size equal to 2, i.e. it is using two alternating buffers, and a tile size equal to 1. The tiled implementation has a window size equal to 2 and a tile size larger than 1. The windowed implementation has a window size greater than 2 and a tile size equal to 1. Finally, the window-tiled implementation has a window size greater than 2 and a tile size greater than 1. In the following, the default tile size [46] of the benchmark is considered. It is set to 10 for the tiled and window-tiled implementations, and a window size of 3 for the

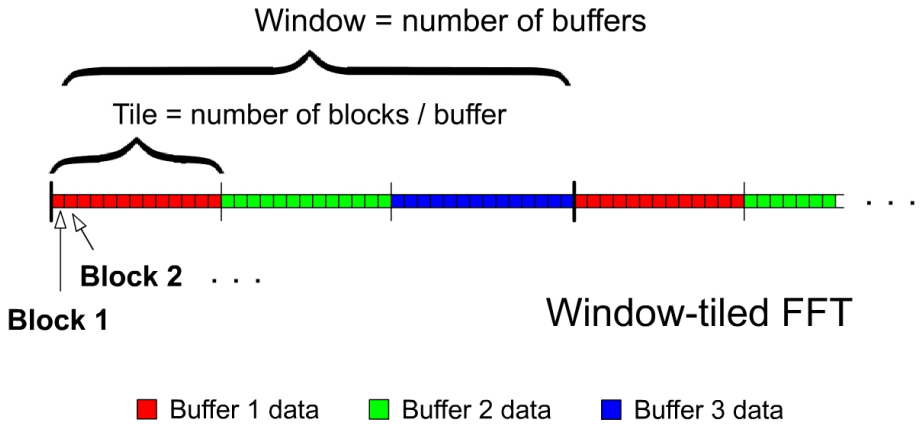

Figure 3.9: Representation of the window-tiled pattern used for multi-dimensional FFT operations.

windowed and window-tiled implementations.

A large number of tests on both *whale* and *crill* were executed. However, due to low performance and high execution times of *whale-tcp* over Gigabit Ethernet, these test-cases were omitted. Instead, several tests on an IBM Bluegene/P located at KAUST Supercomputing Laboratory were executed. Process counts of 160, 358, 500 and 1024 (Bluegene/P only) have been used for different problem sizes, number of planes, and number of progress calls. For the sake of clarity, the last two arguments are omitted. Altering those parameters did not change any of the fundamental observations discussed. Each version of the 3D FFT was executed 350 times iteratively on randomly generated data using ADCL and LibNBC. The output data of the test-cases was verified for correctness in multiple instances.

Comparison to LibNBC The first set tests the performance of the ADCL versions of the code compared to the versions using non-blocking communication based

on the LibNBC library. The results shown in figure 3.10, were obtained on the *crill* cluster using 160 and 500 processes. The results demonstrate that ADCL outperforms LibNBC for all four versions of the 3-D FFT operation by selecting the best non-blocking Ialltoall implementation, with the exception of two scenarios using 160 processes. The performance benefits can be explained by the support of only a single implementation of the non-blocking all-to-all operation by default in LibNBC, namely the linear algorithm. Analyzing the scenarios where LibNBC outperformed ADCL revealed the selection logic of ADCL determined the same linear algorithm to be optimal as provided by LibNBC. The difference in the execution time between the LibNBC and the ADCL version stemmed from the overhead of testing the other algorithms during the learning phase of ADCL.

Comparison to blocking operations In the next set of tests, a version of the 3D-FFT operation which utilizes a blocking `MPI_Alltoall` operation for the communication was added. The results obtained on *whale* with the different versions of the FFT application kernel using MPI, LibNBC and ADCL are displayed in figure 3.11 and figure 3.13.

ADCL outperformed LibNBC in the vast majority of the cases, it is interesting to note, that in some cases such as using 358 processes on whale and 1024 processes on the BlueGene, the versions using the blocking `MPI_Alltoall` operation outperformed all non-blocking versions. To understand the effects that lead to this behavior, the `ADCL_Ialltoall` function-set has been modified to also include blocking operations. As mentioned in section 3.2, it is possible to include a blocking operation as part of a non-blocking function-set by simply not utilizing the `wait` function pointer in the

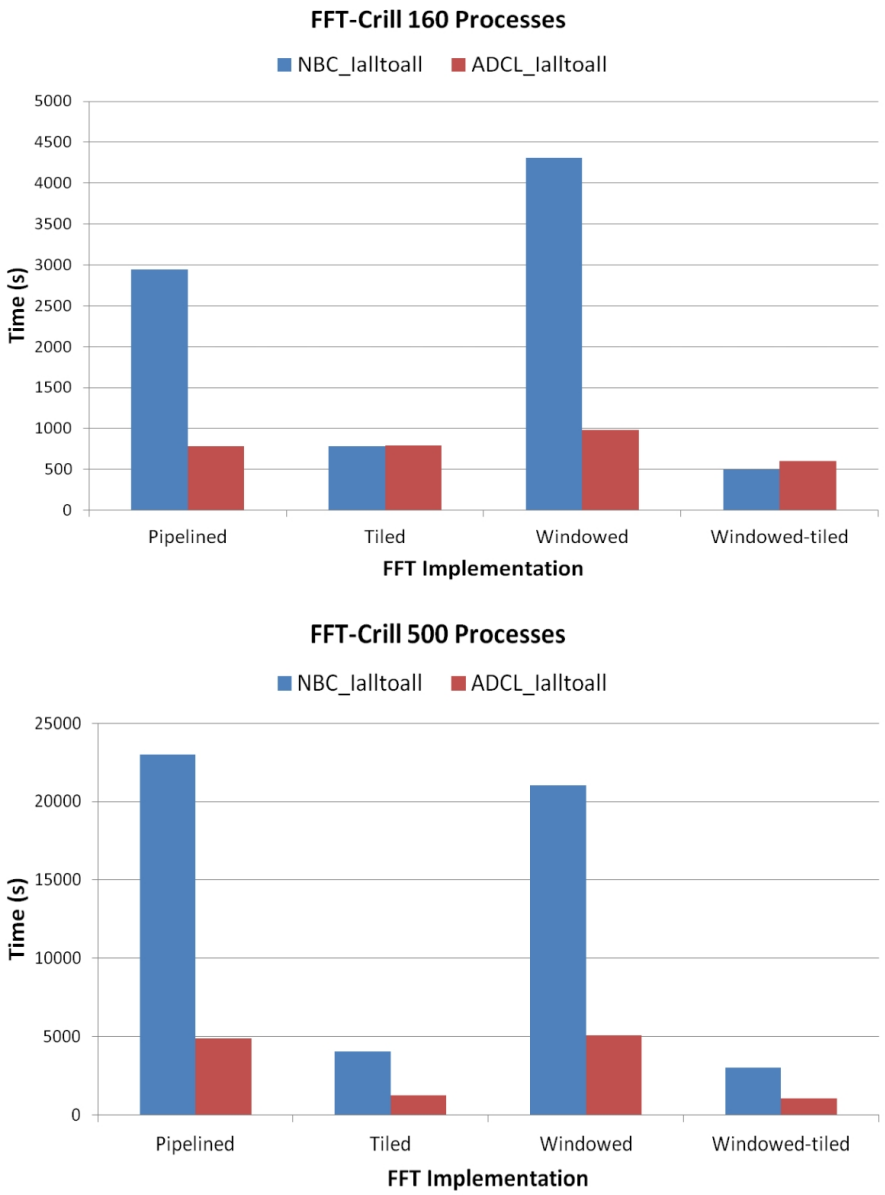

Figure 3.10: Execution time of various patterns used for the 3D FFT operation using LibNBC and ADCL on *crill* for 160 processes (bottom) and 500 processes (top).

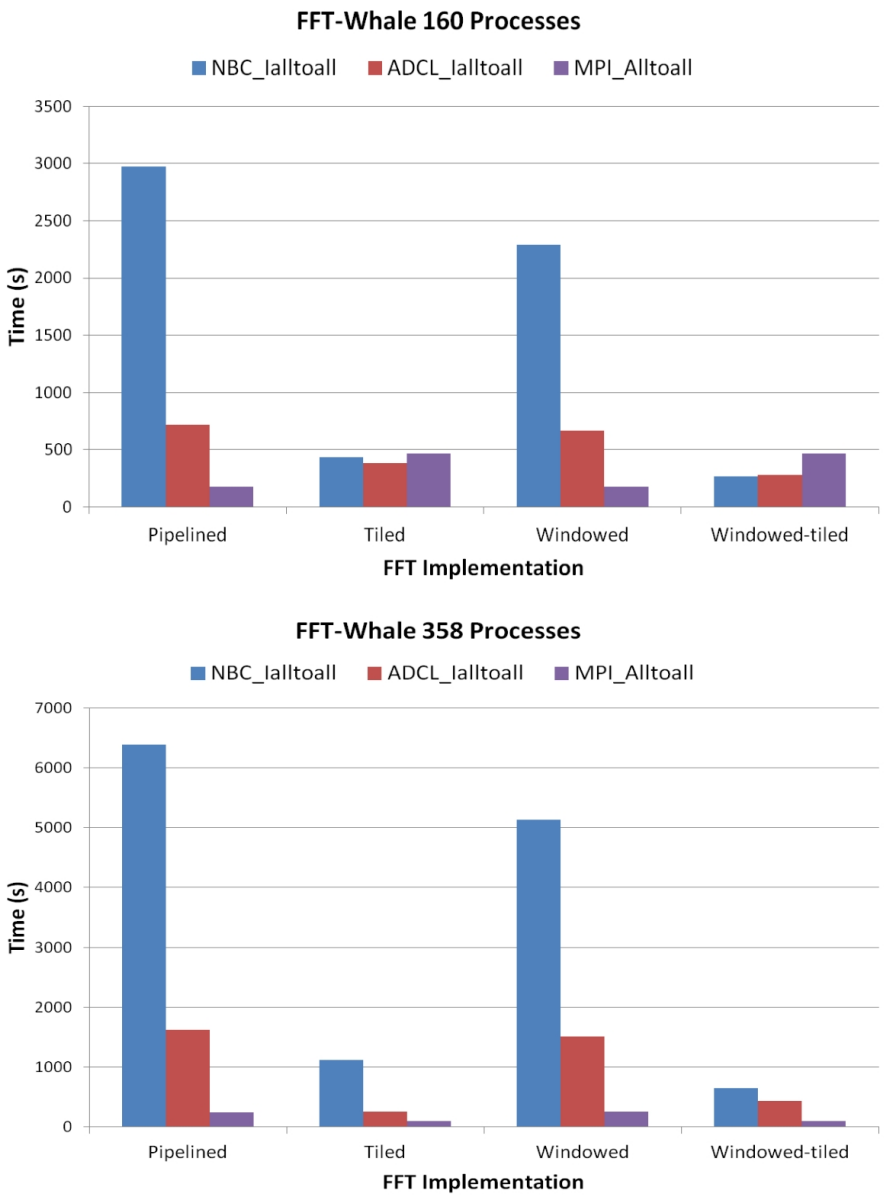

Figure 3.11: Execution time of various patterns used for the 3D FFT operation using LibNBC, ADCL and MPI for 160 processes (bottom) and 358 processes (top).

ADCL request. This is not a problem from the conceptual perspective as long as the order of non-blocking-collective operations posted is identical on all processes. This modified version of the `ADCL_Ialltoall` function-set would ultimately also select whether a particular code sequence benefits from utilizing a non-blocking operation, or would be better off using a blocking version.

The results obtained with the modified `ADCL_Ialltoall` function-set reveal, that the blocking version using `MPI_Alltoall` still provides better performance than the ADCL function-set in some instances. However, the ADCL selection logic did select in 13 out of 16 test cases analyzed on whale a non-blocking implementation as the winner. Consequently, the timing of the selection phase of ADCL is separated from the rest of the execution, and a similar modification to the MPI version is made in order to measure the same number of iterations in both scenarios. The graphs in figure 3.12 and figure 3.13 show the performance of ADCL versus MPI, for both the overall execution time, and the execution time excluding the learning phase. This break down of the execution time reveals, that the ADCL based version does in fact outperform in many instances the blocking `MPI_Alltoall`-based implementation, but due to the larger number of functions that have to be evaluated in the extended `ADCL_Ialltoall` function-set, the additional costs of the learning phase are negating the performance benefits. Nevertheless, the 3D FFT implementations were executed only 350 times each for the sake of simplicity. In general, high-performance computing applications (such as scientific simulations) involve much larger loop nests [52, 76], in which the learning phase of ADCL would be less effective while incorporating the blocking all-to-all algorithms into the non-blocking Ialltoall function-set would be

more advantageous.

To summarize the findings of this subsection, auto-tuning non-blocking-collective operations showed significant performance benefits of the application kernel in the vast majority of the cases. In the few instances where non-blocking operations did not show benefits compared to blocking operations, ADCL was still able to provide performance on par with the best performing implementation found once the decision logic had finished the evaluation, which for longer running applications would still be beneficial.

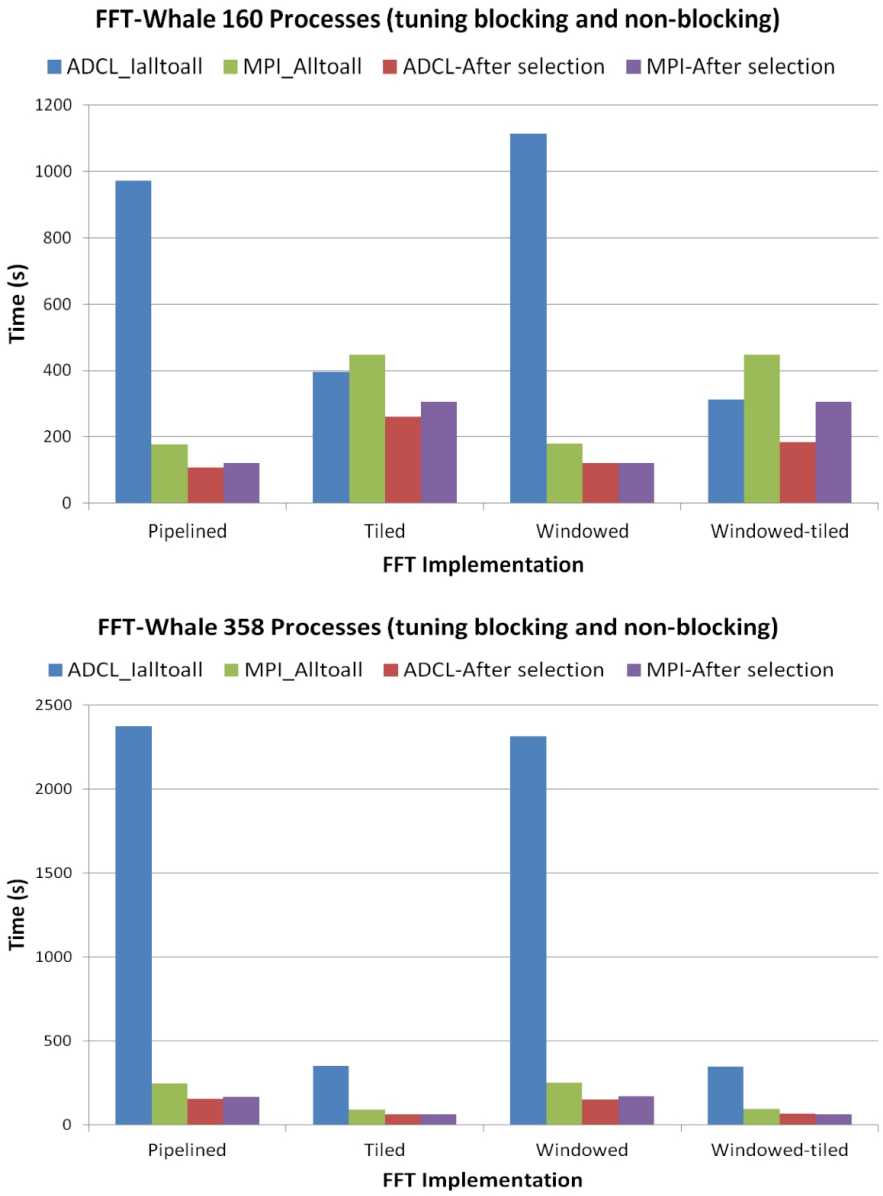

Figure 3.12: Execution time of various patterns used for the 3D FFT operation using the modified ADCL function-set and MPI on *whale* for 160 processes (bottom) and 358 process (top).

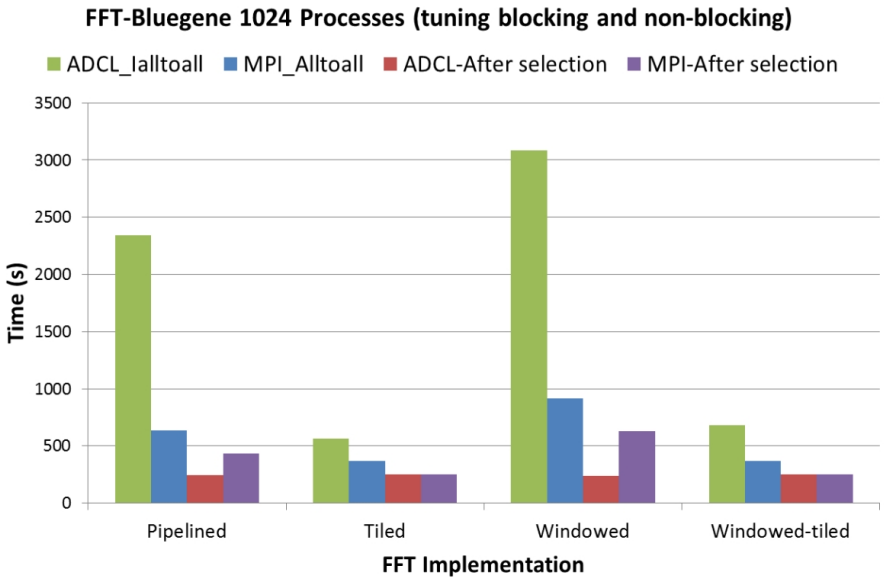

Figure 3.13: Execution time of various patterns used for the 3D FFT operation using the modified ADCL function-set and MPI on *Bluegene/P* for 1024 processes.

Chapter 4

Maximizing Communication-Computation Overlap through Automatic Parallelization

This chapter presents an approach that consists of enhancing an abstract state-of-the-art C to C+MPI+OpenMP automatic code parallelizer that is initially limited to partial and straightforward usage of blocking-collective operations. The approach bypasses the complex steps that integrate NBCs into parallel code, and leverages run-time auto-tuning techniques. Four application benchmarks were used to demonstrate the efficiency and versatility of this approach on two different platforms. The results indicate significant performance improvements in all test scenarios. The resulting parallel applications achieved a performance improvement in the range of $32 - 43\%$

compared to the version using blocking communication-operations, and up to 95% of the maximum theoretical communication-computation overlap identified for each scenario.

The chapter is organized as follows: section 4.1 recalls and summarizes the challenges of using non-blocking-collective operations. Section 4.3 introduces the main concept of the approach, i.e. the extension of an automatic parallel-code generator to maximize communication-computation overlap by integrating an auto-tuner for non-blocking-collective operations, and finally section 4.4 presents the performance results obtained with the approach.

4.1 Factors Affecting Communication-Computation Overlap

Overlapping communication and computation is generally considered a powerful concept for scaling applications for a large number of processes. Multiple factors affect the communication-computation overlap. Although many of the arguments listed in this section also apply to non-blocking individual-communication operations, the discussion focuses on non-blocking-collective operations.

4.1.1 Overlapping Code Sections

Non-blocking-collective (NBC) operations consist of three phases: initialization, progression, and completion. Communication-computation overlap occurs between the

initialization and the completion of an NBC. If an NBC can be executed along another code section and wait for completion before the communicated data is required, then the code section is called an overlapping code region (or section).

In order to hide the costs of NBCs, the user has to find overlapping code sections that comprise sufficient amounts of computations to cover the time spent in the corresponding NBCs. This can be very complex, especially if the computational part is scattered with data-dependencies. To circumvent this problem, minor manual code changes can be applied, using techniques such as double-buffering and look-ahead techniques [46, 50].

If the computational part of the overlapping sections is larger than the time spent in the corresponding NBCs, the scalability of the parallel application will often be limited. This is caused by the fact that larger sub-problems prevent finer-grained parallelism, which simultaneously uses more compute resources. In the ideal scenario, the time spent in computations should be equal to the time spent in the corresponding NBCs. This is often dependent on the size of the sub-problems, commonly referred to as block size, and adjusting the block size is an essential factor in achieving satisfactory communication-computation overlap.

Consequently, optimal-overlapping code sections highly depends on the parallelization strategy applied initially.

4.1.2 Underlying Algorithms

Another challenge in optimizing NBCs stems from the impact that both the hardware and system being utilized, as well as the application characteristics have on

the performance of NBCs. From the system level perspective, factors include networking technology and network topology, processor type and organization, memory hierarchies, operating system versions, device drivers, and communication libraries. In fact, one can argue that every parallel computer is (nearly) unique. Similarly, different parallel applications expose very different communication characteristics, such as frequency of data transfers, volume of data transfer operations, and process arrival patterns (i.e. the time difference on how processes enter a collective operation due to - micro - load imbalances between the processes [30]).

NBCs can be implemented in different equivalent ways (for example, a non-blocking broadcast can be implemented using a linear algorithm, a binary tree algorithm, etc.), such NBCs have to be carefully tuned for a given platform and application to maximize their performance [6]. Using a single algorithm for a NBC leads to sub-optimal performances.

4.1.3 Progression

Progression is another important aspect of non-blocking operations. The performance of NBCs is closely tied to progress in the background, especially in the case of single threaded MPI libraries. There are three ways to ensure background progression:

- **Using a progress thread**: each process spawns a separate thread that executes the non-blocking operation in a blocking manner. This approach has the drawback that the number of threads can be large (e.g. 1000 non-blocking send operations to different processes).

- **Offloading to the hardware**: some networking cards include hardware support for collective operations which allows execution in the background (apart from the processes), and enables communication and computation overlap since the processes are not blocked. However, portability is restricted due to the need of specialized hardware.

- **Using a progress engine**: by regularly invoking progress/completion functions (similarly to `MPI_Test`) at the user-level, the MPI library advances the underlying non-blocking point-to-point operations that constitute the non-blocking-collective operation.

As of today, using a progress engine is the most widely used option because making an MPI library thread safe is highly challenging. The application has to regularly invoke the progress engine of the MPI library [49].

Depending on the application, the locality and frequency of progress-function calls play a significant role on the overall performance. In fact, the overlap of communication and computation may be impaired if, for example, a non-blocking-collective operation is delayed awaiting for a progress-function call to trigger the next steps of the operation. On the other hand, since a progress-function call usually involves intrinsic and performance-costly MPI routines (e.g. `MPI_Test`), it can create a performance overhead if called too often [6]. Computational overlapping code sections have to be flexible enough for progress-function calls to be inserted at optimal locations within the code sequence.

4.1.4 Overlapping Multiple NBCs

Concurrent - or even partially concurrent - NBCs can potentially interfere with one another, especially if they operate on overlapping communicators that use the same network interconnect. Interfering NBCs have to be optimized as a whole, because the outcome of a local (separate) optimization is not the same outcome that a global optimization would provide.

In summary, NBCs provide a promising way to improve the scalability of parallel applications, their utilization is challenging. The developer has to follow a complex repetitive cycle to efficiently integrate NBCs into parallel codes:

- Apply the parallelization strategy to produce overlapping code regions.
- Select the algorithm that provides the optimal sequence of underlying point-to-point operations.
- Place progress-function calls at optimal locations within overlapping regions.
- Enforce data-dependencies and avoid deadlocks (in case of overlapping NBCs).
- Tune interfering (overlapping) NBCs as a whole (co-tuning).

These require a high user expertise on a multitude of software libraries, hardware platforms, and the objective application itself.

4.2 Measuring Communication-Computation Overlap Performance Benefit

To evaluate the amount of communication-computation overlap achieved throughout the execution of a parallel application, multiple metrics are defined in this subsection. Given the overall execution time of a code region, T_{exec} (which also consists of the time spent in communication operations), T_{comm}, and the time spent in computation operations, T_{comp}, let's define:

- **Potential Overlap Ratio** (POR): POR is the maximum (theoretical) amount of time to be saved in case of a 'perfect' communication-computation overlap. Therefore,

$$POR = \frac{min\left(T_{comp}, T_{comm}\right)}{T_{comp} + T_{comm}} \tag{4.1}$$

- **Actual Overlap Ratio** (AOR): AOR is the relative amount of time saved during the actual execution. Therefore,

$$AOR = \frac{|T_{comp} + T_{comm} - T_{exec}|}{T_{comp} + T_{comm}} \tag{4.2}$$

- **Relative Overlap Ratio** (ROR): ROR is the relative amount of overlap achieved during the actual execution.

$$ROR = \frac{AOR}{POR} = \frac{|T_{comp} + T_{comm} - T_{exec}|}{min\left(T_{comp}, T_{comm}\right)} \tag{4.3}$$

The ROR represents an appropriate evaluation measure by determining the proportion of the observed overlap to the theoretical overlap that is later adopted in the evaluation section of this chapter.

4.3 Concept

The main goal of this work is to optimize the communication-computation overlap starting from the earliest stage - sequential codes - based on a parallel model. Using a parallel model enables a regular basis to be generic, and provides a consistent control over the four main factors (described in 4.1) that affect the communication-computation overlap in the early stages of parallelization.

Nevertheless, it is beyond the scope of this chapter to design and implement a new model-based parallel-code generator that supports the goals of this chapter from scratch. In fact, it represents a very complex and challenging domain in HPC and requires enormous effort. For this reason, we investigated different state-of-the-art parallel models and corresponding parallel-code generators, and found that PLUTO [15, 16] performs closest to the goals of this chapter.

4.3.1 PLUTO

PLUTO [15, 16] is a parallel-code generator based on the Polyhedral Model [32–34]. The Polyhedral Model is an abstract representation of compute-intensive affine-loop nests. It performs polyhedral transformations, such as tiling and skewing [7], to generate coarse-grained parallel codes. The skewing transformation shifts the data-dependencies by skewing the grid in appropriate directions. As a consequence, the tiling transformation can be applied without violating data-dependencies and in a more flexible and efficient way by carving the grid into independent blocks. Tiling optimizes both the spacial and temporal localities of the data. Tiling improves data

locality by dividing the grid into coalesced blocks of cells (or elements) and by causing disparate redundant data accesses to be close to each other within the produced tiles. If a tile is small enough to fit into the compute unit's cache, then the performance overhead can dramatically diminish due to these redundant data accesses. The core transformation framework of PLUTO finds affine transformations for efficient tiling. Hybrid codes including MPI and OpenMP instructions (for scalability on heterogeneous architectures) can be automatically generated from sequential C program sections.

Stencil codes are a class of iterative kernels which update array elements by fixed patterns, called stencils. They are very common in the context of scientific and engineering applications, such as computational fluid dynamics, solving partial differential equations, image processing, and cellular automata. Since computing time and memory consumption grow linearly with the number of array elements, parallel implementations of stencil codes are of paramount importance to research. This is challenging since the computations are tightly coupled (cell updates depend on neighboring cells) and most stencil codes are memory bound, i.e. the ratio of memory accesses to calculations is high.

Most of sequential stencil codes correspond to compute-intensive affine-loop nests. This type of loop nests is called Static Control Parts (SCoP [8]), which represents the category of sequential programs that applies to PLUTO.

PLUTO is the only model-based parallel-code generator that we are aware of that targets such a large category of scientific problems (generally stencil codes), and that generates hybrid codes through MPI and OpenMP. They are the most common and

portable programming models in shared and distributed memory architectures.

The objective is to provide: (i) an optimal computation-communication overlap (effectiveness), (ii) on the largest possible set of HPC applications (applicability), and, (iii) on as many architectures as possible (portability).

By choosing PLUTO, both applicability (because of stencil codes) and portability (hybrid code generation based on MPI and OpenMP) are maintained. Effectiveness, depends on the intrinsic potential of communication-computation overlap that PLUTO may provide. This potential is explored in the remaining part of this section.

Parallelization Process in PLUTO

In the following, a brief illustration of the loop-nest parallelization process in PLUTO is given, starting from a simple sequential code. Discussion is restricted to the aspects of PLUTO that are most relevant to this chapter. For other details, please refer to [14, 25]. Let us consider the sequential code in figure 4.1.

```
#define N ...
#define T ...
...
//Main application loop
for (t=0; t<T; t++ ) {
   for (i=0; i<N; i++ ) {
      a[i]= a[i-1] + a[i] + a[i+1];
   }
}
...
```

Figure 4.1: Illustrative sequential code sample.

The loop nest is composed of a temporal dimension 't' and a spacial dimension 'i'. Both dimensions form a 2-D grid. In order to compute the grid in parallel, PLUTO applies skewing and tiling polyhedral transformations that create tiles. The skewing transformation is required in order to produce data-independent tiles. The tiles are evenly distributed across the MPI processes. Figure 4.2 displays this general scheme: the tile distribution is done over two MPI processes ('gray' for process P1, and 'blue' for process P2). P2 starts its part of the execution as soon as a second independent tile occurs, which is the moment depicted in the figure after the two tiles at the bottom left are computed by P1.

In the next step, P1 requires data in order to compute boundary elements. The required data being located at P2, P2 has to explicitly communicate it to P1. This is achieved by the MPI part of the code generation in PLUTO. After that, the two processes keep advancing throughout the grid following the same logic, i.e. alternating communication operations and intra-tile compute-operations.

At this point, the two processes execute two tiles in parallel each, step by step. This local parallelism is achieved by the OpenMP part of the code generation in PLUTO. As a consequence of the skewing polyhedral transformation, the processes swipe the grid towards a skewed tangent in order to be able to execute alternating tiles in parallel. This oblique progression divides the execution into three parts: ramp-up, full-throttle and ramp-down.

In an ideal scenario, during the ramp-up phase, the number of processes involved increases until the maximum number of processes is reached. After that, the number of tiles per process starts to increase until the full-throttle phase is reached. During

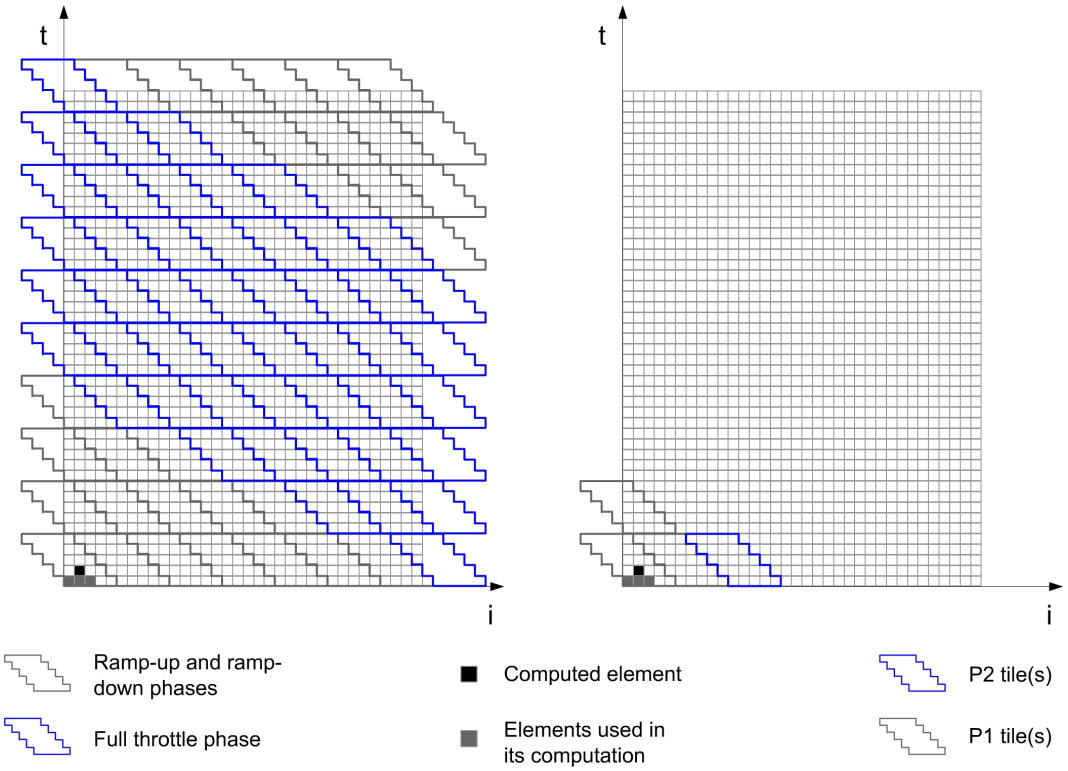

Figure 4.2: Parallelization of 1-D Jacobi stencil code in PLUTO displaying the different stages of the loop tiling transformation, as well as the stencil pattern.

the full-throttle phase, the number of tiles executed in parallel, per process, reaches its maximum. From a global point of view, each process is assigned a range of tiles in an incremental fashion. Within each process, parallelism is achieved through OpenMP directives where the first loop iterates over the tiles that belong to the process (inter-tile loop). The remaining inner loops iterate within the tile itself (intra-tile loops). After the full-throttle phase is finished, the ramp-down phase represents a 'mirrored' equivalent of the ramp-up phase that computes the remainder of the grid.

The communication pattern between processes varies throughout the three phases. An `MPI_Alltoall` operation is first executed in order to exchange necessary meta-data. The meta-data consist of the amount of data that needs to be exchanged between each pair of processes. Next, the actual exchange of data occurs through a series of non-blocking `MPI_Isend` and `MPI_Irecv` operations, accordingly, followed by a global `MPI_Waitall`. This sequence of `MPI_Alltoall`, `MPI_Isend`, `MPI_Irecv`, and `MPI_Waitall` mimics a neighborhood communication. No computations occur in-between the `MPI_Alltoall` and the `MPI_Waitall` operations, which means that the pseudo-neighborhood communication is blocking. In addition, intermediate packing and unpacking routines are used to generate the message buffers.

The code in figure 4.3 depicts the general structure of the output codes generated by PLUTO.

Enabling Communication-Computation Overlap in PLUTO

Hiding the Alltoall operation The first step towards the goal of this chapter is to incorporate double-buffering and look-ahead methods in PLUTO in order to

```
// Preliminary declarations (buffers, ...)
...
for (...) { // Main generic code loop
  #pragma omp parallel for ...
  for (...) { // Iterates over process tiles
    for (...) { // First dimension in tile
      for (...) { // Second dimension in tile
        .
        // Computations
        .
      }
    }
  }
  ...
  // Data packing for MPI pseudo-neighborhood communication
  MPI_Alltoall(...); // Pseudo-neighborhood meta-communication
  ...
  for(...){
    MPI_Isend(...); // Pseudo-neighborhood send to each neighbor
    MPI_Irecv(...); // Pseudo-neighborhood receive from each neighbor
  }
  MPI_Waitall();
  // Unpacking exchanged data
}
```

Figure 4.3: General structure of a PLUTO generic hybrid code.

automatically transform the blocking `MPI_Alltoall` operation into a non-blocking equivalent (`MPI_Ialltoall`) operation. For this, the Ialltoall operation of the next iteration is launched in advance during the current iteration, after making a copy of the meta-buffers. These buffers are involved in the Ialltoall operation of the current iteration during the previous iteration (recurrently). After that, the computational block is executed while progressing the Ialltoall operation of the next iteration in the background through `MPI_Test` function calls. After the computational block is finished, the Ialltoall operation of the next iteration is completed and the meta-buffers

of the current iteration are restored.

Hiding the pseudo-neighborhood communication-operation The only reason that `MPI_Waitall` is called at the end of the communication block (which therefore blocks the pseudo-neighborhood operation) is that the next iteration's tiles are part of the OpenMP block which requires the subsequently exchanged data. This block is executed immediately afterwards. In order to hide the pseudo-neighborhood communication, a loop fission transform is applied on the OpenMP block to split it into two separate compute-blocks. Only the inter-tile loop is split into two separate analogous loops. The first inter-tile loop operates on the process' inner tiles, excluding every intra-process neighboring tile (see fig. 4.2). This way, the inter-tile loops are guaranteed to operate on data-independent tiles, and are assured to safely overlap the pseudo-neighborhood communication-operation. The `MPI_Waitall` operation is executed afterwards. The unpacking routines are called thereafter. The second OpenMP inter-tile loop is then able to operate, safely (since the pseudo-neighborhood communication is over at this point), on the tiles that require the exchanged data. The first OpenMP block includes progress calls for both the Ialltoall and Isend / Irecv operations (`MPI_Test` and `MPI_Testall`, respectively). The second OpenMP block, however, contains only `MPI_Test` calls (since the pseudo-neighborhood operation is finished at this point).

```
// Preliminary declarations (buffers, ...)
...
for (...) { // Main code loop
  // Backup meta-buffers of current iteration
  ...
  MPI_Ialltoall(...); // Pseudo-neighborhood meta-communication of next iteration
  #pragma omp parallel for ...
  for (...) { // Iterates over data-independent process tiles
      for (...) { // First dimension in tile
        ...
      }
    ...
  }
  MPI_Waitall(...);
  // Unpacking exchanged data
  ...
  #pragma omp parallel for ...
  for (...) { // Iterates over data-dependent process tiles
      for (...) { // First dimension in tile
        ...
      }
    ...
  }
  MPI_Wait(...); // Completion of Ialltoall operation
  // Restore meta-buffers of current iteration
  ...
  // Packing routines for MPI pseudo-neighborhood communication
  for(...){
    MPI_Isend(...); // Pseudo-neighborhood send to each neighbor
    MPI_Irecv(...); // Pseudo-neighborhood receive from each neighbor
  }
}
```

Figure 4.4: The new structure for generic hybrid code which enables communication-computation overlap.

Maximizing Communication-Computation Overlap

Now that the communication-computation overlap is enabled, the next step was to maximize it, considering the four main factors that affect the communication-computation overlap detailed in section 4.1.

Overlapping code regions The maximal overlapping code region is the two compute-intensive OpenMP blocks. The first block overlaps both the Ialltoall and the pseudo-neighborhood communication, and the second one overlaps with the rest of the Ialltoall operation only.

Underlying algorithms The Ialltoall communication in PLUTO operates on small amounts of data, called process-pairs data counts. The underlying algorithm has a larger impact on performance. This sensitivity is due to the high ratio of latency to communication length.

The idea is to use the Abstract Data and Communication Library (ADCL 2.1.2) to auto-tune the Ialltoall operation. However, this imposes a limitation since tuning in ADCL is done at run-time, and a learning phase requires multiple iterations where the communication and computational patterns have to remain consistent. Because of this, the ramp-up and ramp-down sections have to be excluded from the approach. This is not a major restriction since the full-throttle phase is the dominant phase for long running applications. Figure 4.2 clearly depicts this, where the 'blue' (dark) tiles represent the full-throttle corresponding tiles. As a matter of fact, the main dimension upon which PLUTO performs the skewing and tiling transformations is the temporal dimension. If the total number of time iterations ('t' in figure 4.2)

increases, the full-throttle section gets longer, while the ramp-up, and ramp-down sections remain static.

The Timer object, which measures the time spent in a code section and upon which the auto-tuning decision of ADCL is made (c.f. 2.1.2), has to cover both the Ialltoall and the corresponding-computational-overlapping code region. The code in figure 4.5 displays the new generic code structure (both ADCL and MPI progress function calls are excluded for simplicity).

Progression Progression is critical in hiding non-blocking-collective operations in the background. An insufficient number of progress calls leads to longer running NBCs, which decreases the communication-computation overlap. Too many progress calls create a performance overhead [6]. Finding the optimum frequency of progress calls is challenging. To solve this challenge, the frequency of progress calls is auto-tuned. Since the compute part of the application consists of multiple nested loops, the progress calls being inserted at each loop level are evaluated. A progress function located at the innermost loop will lead to a significantly higher number of progress calls than a progress function being located in the outermost loop.

Using ADCL, a depth attribute tests the different Ialltoall implementations at different depths of progress calls. The depth that provides the best performance within the timer bounds is selected for the rest of the application (more specifically, for the rest of the full-throttle section).

```
// Preliminary declarations (buffers, ...)
ADCL_Request ...;
ADCL_Timer timer;
...
// Initialize non-blocking persistent collective operation
ADCL_Ialltoall_init (...);
...
for (...) { // Main code loop
  // Backup meta-buffers of current iteration
  ...
  //Start timer
  ADCL_Timer_start (timer);
  // Start meta-communication operation of next iteration
  ADCL_Request_init (...);
  #pragma omp parallel for ...
  for (...) { // Iterates over data-independent process tiles
      for (...) { // First dimension in tile
        ...
      }
    ...
  }
  MPI_Waitall(...);
  // Unpacking exchanged data
  ...
  #pragma omp parallel for ...
  for (...) { // Iterates over data-dependent process tiles
      for (...) { // First dimension in tile
        ...
      }
    ...
  }
  // Wait for completion of meta-communication
  ADCL_Request_wait (...);
  // Stop timer
  ADCL_Timer_end (timer);
  // Restore meta-buffers of current iteration
  ...
  // Packing routines for MPI pseudo-neighborhood communication
  ...
```

```
  for(...){
    MPI_Isend(...); // Pseudo-neighborhood send to each neighbor
    MPI_Irecv(...); // Pseudo-neighborhood receive from each neighbor
  }
}
```

Figure 4.5: The new structure for generic hybrid code, the communication-computation overlap + auto-tuning.

Overlapping NBCs If the input sequential code operates on multiple grids, the generated code will have multiple communication blocks. ADCL has the ability to attach one common timer object to different requests, which simultaneously co-tunes multiple requests at the same time. Therefore, if multiple Ialltoall operations are generated alongside multiple pseudo-neighborhood communication operations, the solution is to combine all the operations through a common timer object. The same auto-tuning logic described previously applies here as well. Additionally, ADCL includes a user-configurable meta-parameter to indicate whether two similar requests should be merged and auto-tuned as a single request, or whether they should be distinguished. Two requests are similar if all their parameters (e.g. collective operation, message length, etc.) are the same, except possibly their buffers. This meta-parameter, along with other useful meta-parameters, is defined in a local ADCL configuration file. Switching 'off' the request-merging feature produces larger pools of implementations and, as a consequence, achieves a finer optimization. In fact, if all the requests are distinguishable, ADCL considers every combination of implementation as a new and distinct implementation. However, this has an impact on the length of the selection phase. Switching the request-merging feature 'on' would

shrink the search space.

4.3.2 Low-Level Aspects

Ialltoall and Auto-Tuning in ADCL

By default, the Ialltoall collective operation in ADCL has three different algorithms (or implementations), namely: linear, pairwise, and dissemination. ADCL provides different search algorithms in order to regulate the accuracy of the selection phase versus its length. A brute-force algorithm allows ADCL to test every implementation in order to determine the fastest one. It is the most accurate but the slowest search algorithm in ADCL. Besides that, a number of iterations is required per implementation during the selection phase. ADCL uses these iterations to accurately assess the performance of these implementations, eliminating outliers. 30 iterations per implementation are recommended [11], which brings the selection phase cost up to 90 iterations. On the other hand, the maximum depth of progress calls ranges from 3 (for 2-D input sequential codes) to 4 (for 3-D input sequential code). One extra depth of 'zero' signifies a blocking implementation, allowing no progress calls to be performed. Consequently, the number of total implementations of the Ialltoall collective operation in ADCL ranges from 12 (3 implementations * 4 depths) to 15 (3 implementations * 5 depths). This leads to a selection phase cost of 360 to 450 iterations. If two requests are co-tuned by the same timer object, with the request-merging feature disabled in ADCL, then the total cost of the common selection phase becomes 4320 $(12 * 12 * 30)$ to 6750 $(15 * 15 * 30)$ iterations.

If the main application loop is long, a brute-force algorithm is recommended.

Otherwise, other heuristic-based search algorithms that shrink the search space can be used. Merging similar requests in ADCL shrinks the search space, as explained previously in 4.3.1. The desired search algorithm, as well as the number of iterations per implementation are also defined in the same local ADCL configuration file mentioned in 4.3.1.

Automatic ADCL configuration As discussed above, ADCL comprises different configuration meta-parameters (or simply parameters) that alter the selection phase according to different aspects. The main aspect that the configuration parameters are designed to tune is the balance between accuracy and performance overhead. Considering these parameters (e.g. request merging, search algorithm), the question that arises is how to determine the configuration that maximizes the ratio of accuracy to overhead, before run-time ?

The idea is to incorporate the ADCL configuration in the parallelization process in PLUTO. Based on some problem characteristics (e.g. the spacial and temporal dimensions, and the maximum number of merge-able ADCL requests), one can estimate the configuration's overhead by the ratio of the total number of iterations needed by the ADCL selection phase, to the total number of iterations that involve ADCL. In the context of PLUTO, this can be achieved before run-time, through the following steps:

1. The configuration that provides the maximal accuracy is brute force, no request-merging, and 30 iterations per implementation.

2. The number of iterations needed by this configuration (360 to 450 for a single

request, 4320 to 6750 for two co-tuned but non-merged requests, etc. - c.f. 4.3.2 -).

3. Calculate the number of iterations required by the ramp-up and ramp-down phases.

4. Subtract the latter from the total number of iterations, obtaining the number of iterations of the full-throttle phase (which is the only part that involves ADCL).

5. Divide the total number of iterations needed by the ADCL selection phase (obtained in (2)), by the total number of iterations that involve ADCL (obtained in (4)).

As a result, if the overhead estimate is above a predefined threshold (15% by default, user-configurable), then the initial configuration is gradually modified in order to reduce the overhead estimate, until a satisfactory overhead estimate is met (below the threshold). The first alteration consists in enabling the request-merging feature, in case co-tuned requests are detected. If not, this step is skipped. However, if the request-merging gets enabled, then the steps (2) and (5) - only - are repeated. If the new ratio is still above the predefined threshold, then the next alteration consists of decreasing the number of iterations per implementation by 5 (from 30 to 25).

The same process repeats, until the number of iterations per implementation drops to 15. If the overhead threshold is still not met, then there is no point decreasing the number of iterations per implementation any further, since a lower number increases the risk of inconsistency of the ADCL auto-tuning engine due to

outliers [10]. Instead, the attribute-based heuristic is chosen as a search algorithm instead of brute force. At this point, there is no possible way to estimate the overhead of the selection phase in ADCL, since the performance cost (or the number of iterations needed) of the attribute-based heuristic is non-deterministic.

Since the initial configuration is the most accurate causes the configuration to continue downgrading until a satisfactory overhead estimate is met. Hence, it constitutes a convergent iterative optimization process that, when incorporated in PLUTO, optimizes the configuration of ADCL automatically before run-time.

Blocking alternatives A depth of 'zero' as the optimal placement for the progress function means that there is no possibility of communication-computation overlap in the given scenario. Changing the tile size plays a significant role to balance the communication-computation ratio. To prevent additional overhead, the communication operation is switched back to blocking algorithms. This is done after the ADCL selection phase is finished, thereby making the selection phase of ADCL the only source of potential overhead. It leads to a blocking Alltoall operation that is optimized using the same principles within ADCL. In fact, ADCL has the ability to optimize the same collective operation considering both blocking and non-blocking implementations at the same time [6]. By registering a new request through `ADCL_Alltoall_init()` instead of `ADCL_Ialltoall_init()`, both blocking implementations as well as the 'zero' depth non-blocking implementations are auto-tuned.

The performance price incurred is restricted to the auto-tuning of the blocking Alltoall operation. This selection does not involve tuning the progress depth. In

fact, the new pool of implementations contains the three previous non-blocking implementations of the Ialltoall collective operation corresponding to the 'zero' depth, alongside eight additional (default) Alltoall algorithms. The latter are, in contrary, blocking at the outset. The new total number of implementations is eleven, given the default recommended 30 iterations per implementation, the total cost of the new selection phase adds up to 330 iterations, slightly shorter than the cost of the non-blocking selection phase (360 to 450 as calculated previously in this sub-section).

Besides that, note that selecting the maximum depth by ADCL does not necessarily mean that the ideal progression has been met. In fact, the characteristics of the application - mainly the dimensions of the problem to parallelize - can limit the frequency of progress calls if very few `for` loops are generated in the OpenMP blocks. If the highest frequency is selected by ADCL, then there may be a gap between the selected frequency and the 'theoretically' optimal (ideal) frequency. This optimal frequency is unknown (since there is no further inner `for` loops available), which makes estimating the gap impossible, even at run-time. One solution is to change the tile size to enable a uniform progression pattern, through a single optimal frequency of progress calls. This primarily boils down to granularity, and a more proper tile size becomes the main aspect.

Tile Size Calculation

Tile size plays a significant role in the granularity of the parallelization, as well as the potential progression of the entailed non-blocking operations. Though PLUTO is fully automatic, the option to specify the tile size manually is provided. The tile size affects the scalability of the application as follows: the full-throttle phase (during

which the number of tiles that can be executed in parallel is maximal) puts a limit on the the maximum number of processes that can be used. If the number of processes is larger than the number of tiles, a subset of processes will have only one tile to compute at full-throttle, and the rest of the processes remains idle. The number of tile per process during the full-throttle phase is referred to as tile-occupancy, and is proportional to the problem size. A minimal tile-occupancy of 2 is required in order for the MPI processes to compute at least one tile while the other tile is involved in inter-process communication. This inverse correlation between the tile size and tile-occupancy represents a critical higher-bound (in terms of tile size) in order to ensure communication-computation overlap.

On the other hand, the global ratio of communication to compute-operations is correlated to the data exchanged between the pairs of neighboring processes, and more specifically, their corresponding adjacent tiles. In general, the amount of communicated data is of one dimension less than the amount of computed elements. In fact, axiomatically, the intersection of two adjacent volumetric tiles is a superficial space, and the intersection of two adjacent superficial tiles is a linear space, and so forth. This suggests that a larger tile increases the ratio of communication to compute-operations (offering more computations to hide the communication cost).

As a consequence, the optimal tile size is the largest tile size for which the tile-occupancy is not less than 2 for each process. Yet, for a single sequential code, PLUTO outputs a single hybrid code that is reusable with an arbitrary number of processes and problem sizes, and arbitrary total-time iterations. However, it corresponds to a single input tile size that is statically defined by the user as an input

to PLUTO, alongside the initial sequential code. Since, the number of processes and the spacial and temporal dimensions are unknown during the parallelization process while the tile size is required beforehand, makes it impractical to determine the optimal tile size. This fundamental limit requires the definition of the number of processes as well as the spacial dimensions to be shifted upstream, i.e. before the parallelization process. The only penalty paid is the non-re-usability of the output hybrid code if the number of processes and/or the spacial dimensions change.

Furthermore, one of the major benefits of tiling is to improve the data locality (spacial and temporal) for faster data accesses (c.f. 4.1). It gives the tile size a significant importance regarding the computational performance. Cache friendliness is a good example: a tile that fits into the highest CPU cache level allows faster calculations compared to a larger tile that would produce cache misses. For this reason, tile sizes floored to powers of 2 are privileged to void the padding for cache-friendliness [53]. However, if the tile size is a single-digit per dimension (e.g. $7*7*7$), then flooring to powers of 2 ($4*4*4$ in this case) is discarded since other performance factors become predominant when small tile sizes are used [22].

4.4 Experimental Evaluation

In the following section, the impact of the approach described in this chapter on optimizing the communication-computation overlap using the ADCL-PLUTO version is evaluated. First, the execution environment is described, followed by detailed results obtained with four different application benchmarks, and two distinct hardware platforms.

4.4.1 Experimental Platforms

Two clusters located at the University of Houston have been used in the subsequent tests, namely *Opuntia* and *Crill*. *Opuntia* consists of 92 nodes with a total of 1,740 processor cores of 2.8 GHz Intel Xeon E5-2680v2, 1.83 TB of total main memory and a 56 Gb/s Ethernet interconnect.

The second cluster, *Crill*, consists of 16 nodes with four 2.2 GHz 12-core AMD Opteron (Magny Cours) processor cores each (48 cores per node, 768 cores total) and 64 GB of main memory per node. Each node further has two 4x DDR InfiniBand Host Channel Adapters (HCA). The 1.8 series of Open MPI [37] was used in all instances, along with the 0.11 version of PLUTO [1] and the 2.0 version of ADCL [72].

4.4.2 Application Benchmarks

Four application benchmarks were used to assess the communication-computation overlap achieved by the approach presented in this chapter. The four benchmarks are: FDTD-1D, Jacobi-1D, Seidel-2D and ADI-2D. The four benchmarks were developed by Saday et. al. [1].

FDTD-1D (Finite-Difference Time-Domain) is a numerical analysis technique used for modeling computational electrodynamics. FDTD-1D performs computations on two different one-dimensional buffers, which represents the spacial dimension. The computations are repeated a certain number of times, which represents the temporal dimension. Jacobi-1D uses relaxation to find discretized solutions to differential equations. It performs computations over a temporal dimension as well, but on only a single one-dimensional buffer. A second temporary buffer can be used

for optimization purposes, but is not required.

Seidel-2D is an iterative method used to solve systems of linear equations. It performs computations on a single two-dimensional buffer, along with a temporal dimension. ADI-2D (Alternating Direction Implicit) is a finite difference method for solving parabolic, hyperbolic and elliptic partial differential equations. It perform computations on two two-dimensional buffers along with a temporal dimension, a third two-dimensional buffer is used for reading purposes only.

The four application benchmarks were chosen for the sake of representativeness. They can be ordered from the most memory-intensive benchmark to the most compute-intensive one based on the ratio of communication to compute-operations that each benchmark involves:

- FDTD-1D: two one-dimensional buffers (most memory-intensive)
- Jacobi-1D: one one-dimensional buffer (more memory-intensive)
- ADI-2D: two two-dimensional buffers (more compute-intensive)
- Seidel-2D: one two-dimensional buffer (most compute-intensive)

The higher the number of buffers, the more memory movements are required compared to the amount of compute-operations. In this case, the code is classified as more memory-intensive. On the other hand, the higher the dimension of the buffer(s), the more compute-operations are required compared to the number of memory accesses. In this case, the code is considered to be more compute-intensive (c.f. 4.3.2).

Benchmark	Process counts	Time iterations	Space size	% of ramp-up & -down	% of ADCL Selection	Tile-occupancy	Optimal tile size
FDTD-1D	128	$200K$	$20M$	3.86%	11.28%	4	$512 * 512$
Jacobi-1D	128	$200K$	$20M$	5.74%	0.95%	4	$512 * 512$
ADI-2D	128	$40K$	$4K * 4K$	9.16%	26.54%	3	$5 * 5 * 5$
Seidel-2D	128	$40K$	$4K * 4K$	9.17%	2.95%	3	$5 * 5 * 5$

Table 4.1: Benchmark parameter values and subsequent performance indicators on *Opuntia*. ($K = thousand$, $M = million$)

The four application benchmarks therefore constitute an impartial base-set in order to evaluate the communication-computation overlap achieved by the ADCL-PLUTO version. Each application benchmark is accompanied by a set of performance evaluation parameters. The performance evaluation parameters are: process counts, time iterations (total length of the temporal dimension), and space size (total length, surface, or volume of the spacial dimension(s)).

Based on the experimental platforms described in 4.4.1, the process counts chosen for the experimental evaluation were: 64, 128 and 256 for both *Opuntia* and *Crill*. For the sake of clarity, results are only presented for the 128 process test case on *Opuntia* and 256 processes on *Crill* initially. Other results do not change any of (nor add to) the fundamental observations discussed in this section, and are therefore presented subsequently.

4.4.3 Experimental Results

In the first set of tests, the automatic ADCL configuration described in section 4.3.2 was disabled. Table 4.1 shows the values of the different corresponding performance evaluation parameters per benchmark for *Opuntia*. Table 4.2 shows the values for the same parameters for test cases executed on *Crill*.

Benchmark	Process counts	Time iterations	Space size	% of ramp-up & -down	% of ADCL Selection	Tile-occupancy	Optimal tile size
FDTD-1D	256	$200K$	$20M$	3.86%	5.64%	4	$256 * 256$
Jacobi-1D	256	$200K$	$20M$	5.73%	0.47%	4	$256 * 256$
ADI-2D	256	$40K$	$4K * 4K$	7.32%	17.34%	3	$3 * 3 * 3$
Seidel-2D	256	$40K$	$4K * 4K$	7.33%	1.73%	3	$3 * 3 * 3$

Table 4.2: Benchmark parameter values and subsequent performance indicators on *Crill*. ($K = thousand$, $M = million$)

The first three columns show the number of processes used, the number of time-step iterations (temporal dimension), and the size of the problem space (spacial dimensions). Temporal dimensions and spacial dimensions were chosen to give reasonable execution times, as well as tolerable memory space utilized by the main data structures. Column four displays the relative number of iterations spent in the ramp-up and ramp-down phases (combined) (recall that no communication-computation overlap is achieved during these two phases, c.f. 4.3.1). This value is essential for the heuristic used in the automatic ADCL configuration phase detailed in 4.3.2. The iterations mentioned here are the iterations of the output-parallel application, and not the input-sequential benchmark. Column five contains the relative performance cost of the ADCL selection phase (in terms of iterations). Column six shows the tile-occupancy and the last column displays the corresponding tile-size.

Each benchmark has been executed three times on randomly generated data, reporting the average of those three executions (double precision numbers have been used, along with result checking). After each execution, the maximum time spent across all processes is recorded. The executions were achieved within the same batch scheduler allocation and thus had the same node assignments per benchmark for the sake of consistency, in accordance with the parameter values exposed in tables 4.1

and 4.2.

Initially, the execution is performed with no communication-computation overlap, using the default PLUTO parallel-code generation. The default parallel versions incorporate components that measure the computational time, as well as the communication time, separately. They also allow to measure the total time, as well as other secondary measures (such as the total communication volume across all processes). The sum of both the communication time and computational time therefore represents the theoretical performance achieved with no communication-computation overlap, and is used as a base-line to assess the performance of the ADCL-PLUTO version. The same time-measuring components are subsequently used by the ADCL-PLUTO version to measure the total time spent in the communication-computation section. For the sake of fairness, the other (common) parts of the generated code are excluded because of irrelevance.

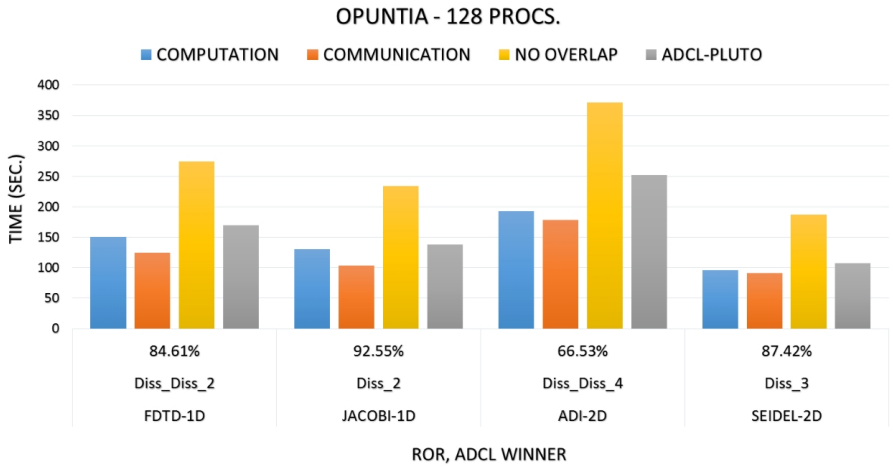

Figure 4.6: Execution times of the four benchmarks on *Opuntia* using 128 processes (automatic ADCL configuration **disabled**).

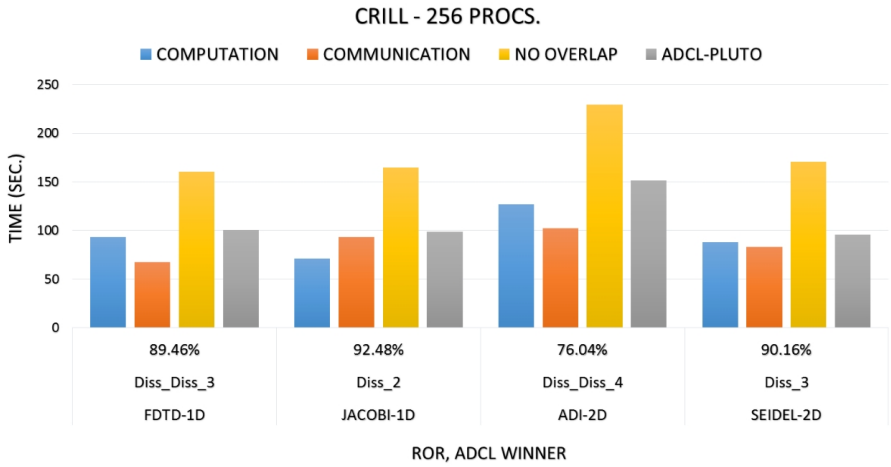

Figure 4.7: Execution times of the four benchmarks on *Crill* using 256 processes (automatic ADCL configuration **disabled**).

Figures 4.6 and 4.7 display the results on *Opuntia* and *Crill*, respectively. The graphs are structured as follows: each set of four bars indicates the execution time spent in each benchmark (denoted on the x-axis) from left to right showing the time spent in compute-operations only (T_{comp}), in communication operations only (T_{comm}), compute- and communication- operations without overlap ($T_{comp} + T_{comm}$), and compute- and communication- operations using the ADCL-PLUTO version. The x-axis also displays the ROR values (c.f. section 4.2) for each benchmark, as well as the outcome of the ADCL auto-tuning phase in terms of the fastest Alltoall algorithm (winner), and the best progress call depth described in 4.3.1. The format is the following: Winner_Depth for the cases where only one ADCL request was auto-tuned, and Winner1_Winner2_Depth for the cases where two ADCL requests were co-tuned. The different Alltoall algorithms for the winner part are abbreviated as *Lin* for linear algorithm, *Pair* for pairwise exchange, and *Diss* for dissemination

algorithm.

The results indicate that the ADCL-PLUTO version achieved a high communication-computation overlap in most cases, producing overall performance improvements ranging from 32% (corresponding to ADI-2D on *Opuntia*) to 43% (corresponding to Seidel-2D on *Crill*); together with *ROR* values approximating 90% for FDTD-1D, Jacobi-1D, and Seidel-2D. However, they expose lower *ROR* values for ADI-2D on both platforms, namely around 66% on *Opuntia*, and 76% on *Crill*. This can be explained by the higher proportion of the number of iterations spent in the selection phase of ADCL (as shown in tables 4.1 and 4.2) compared to the other benchmarks.

To solve this problem, the same set of tests was executed enabling the automatic ADCL configuration feature. The results displayed in figures 4.8 and 4.9 show that the *ROR* values for ADI-2D increased considerably: around 89% on *Opuntia*, and 91% on *Crill*). The minimal overall performance improvement across the four benchmarks also increased to 37% (corresponding to FDTD-1D on *Crill*) compared to 32% obtained without automatic ADCL configuration. Note, that for all the other three benchmarks, the results remain unchanged, since the initial ADCL configuration is retained as outcome of the configuration's heuristic.

This improvement is due to tri-dimensional calculations that ADI-2D performs on two separate buffers, which generates two independent ADCL requests. It is therefore required to enable the ADCL request-merging feature for this specific benchmark. Since both requests are auto-tuned as a single request, the maximum proportion of iterations needed by ADCL to finish the selection phase of the ADI-2D benchmark decreases significantly, as shown in table 4.3.

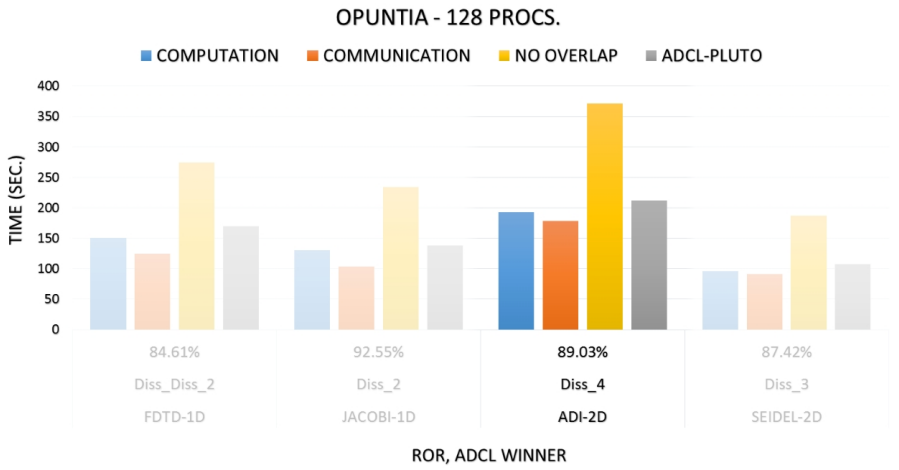

Figure 4.8: Execution times of the four benchmarks on *Opuntia* using 128 processes (automatic ADCL configuration **enabled**).

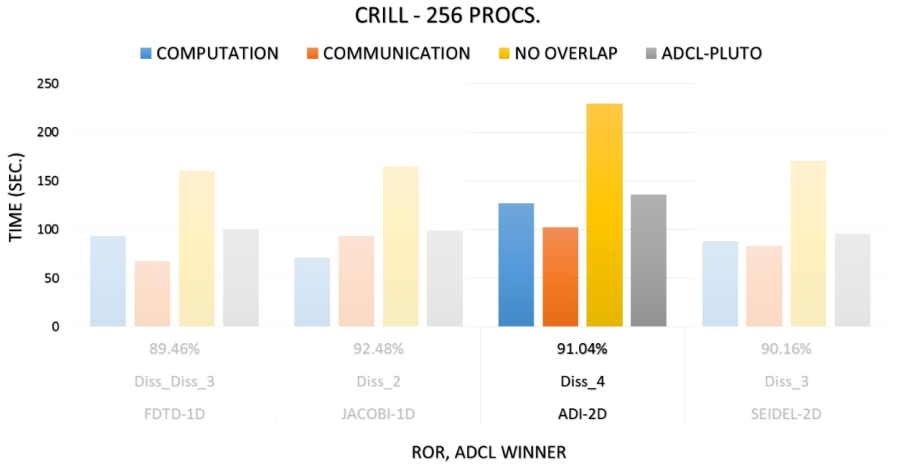

Figure 4.9: Execution times of the four benchmarks on *Crill* using 256 processes (automatic ADCL configuration **enabled**).

Platform	% of ADCL Selection without auto. config.	% of ADCL Selection with auto. config.
Opuntia	26.54%	1.77%
Crill	17.34%	1.16%

Table 4.3: Comparison of the relative performance cost of the ADCL selection phase for ADI-2D with vs. without the automatic ADCL configuration phase.

The values in the left column represent the proportion of iterations needed by ADCL to finish the selection phase of the ADI-2D benchmark, with no automatic ADCL configuration. This means that these values was generated based on the initial (default) ADCL configuration that aims at maximizing the accuracy of the selection phase, regardless of the incurred overhead. Since the threshold of the proportion of the selection phase is set to 15% by default (c.f. 4.3.2), the request merging feature was consequently triggered for ADI-2D, and produced the new proportion displayed in the right column of table 4.3.

FDTD-1D and Seidel-2D being the most memory-intensive and the most compute-intensive benchmarks, respectively (c.f. 4.4.2), exhibit the lowest ROR values. On the other hand, Jacobi-1D and ADI-1D, which are less memory-intensive and less compute-intensive, respectively, exhibit the highest ROR values.

All four benchmarks depicted the dissemination algorithm of the Ialltoall collective operation as winner of the auto-tuning phase. The Alltoall operation in the ADCL-PLUTO version consists of exchanging meta-counters between the processes that are involved in the subsequent pseudo-neighborhood communication (c.f. 4.3.1), and the dissemination algorithm is more adequate for exchanging short messages compared to its other counterparts.

Figures 4.10, 4.11, 4.12, 4.13, 4.14, and 4.15 show the results of the remaining

process counts on both platforms.

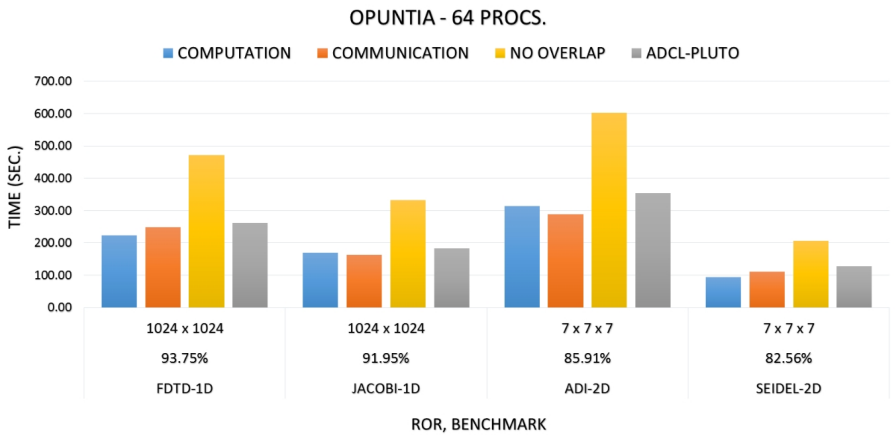

Figure 4.10: Execution times of the four benchmarks on *Opuntia* using 64 processes (automatic ADCL configuration enabled).

4.4.4 Effect of the Tile Size

In order to verify the appropriateness of the tile size calculated in 4.3.2, measurements with enforced and arbitrary tile sizes were run. The results obtained are displayed in figures 4.16 and 4.17.

The x-axis displays the different tile sizes including the optimal tile sizes (underlined in 'red') in the middle, for each benchmark. The results show that the balance between compute-operations and communication-operations in terms of performance cost is altered when the tile size varies, even with a constant problem size. If the tile size decreases below the optimal size, then the tile-occupancy increases, which increases the ratio of communication-operations to compute-operations. This effect is observed on both *Opuntia* and *Crill*, and is even more visible with memory-intensive

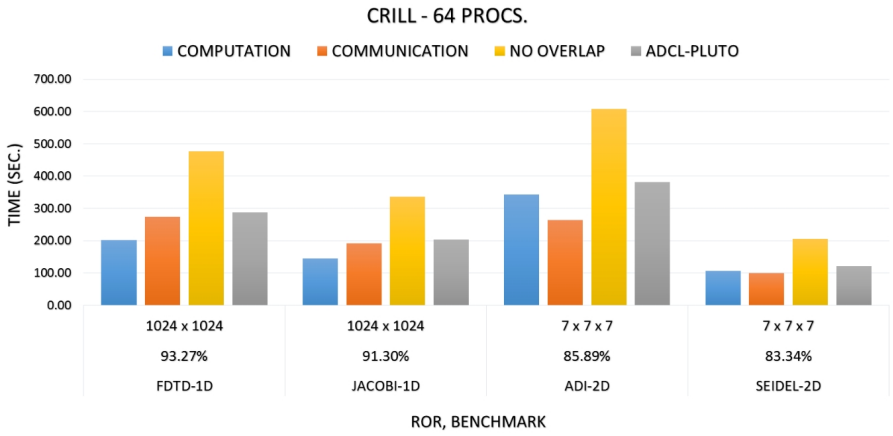

Figure 4.11: Execution times of the four benchmarks on *Crill* using 64 processes (automatic ADCL configuration enabled).

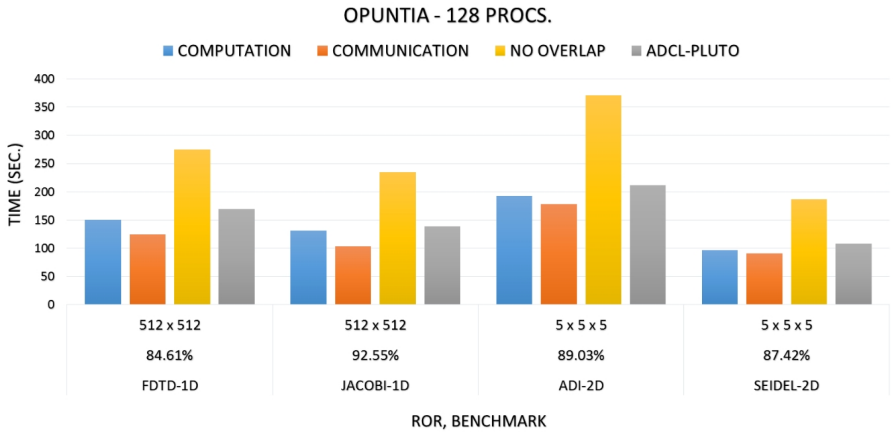

Figure 4.12: Execution times of the four benchmarks on *Opuntia* using 128 processes (automatic ADCL configuration enabled).

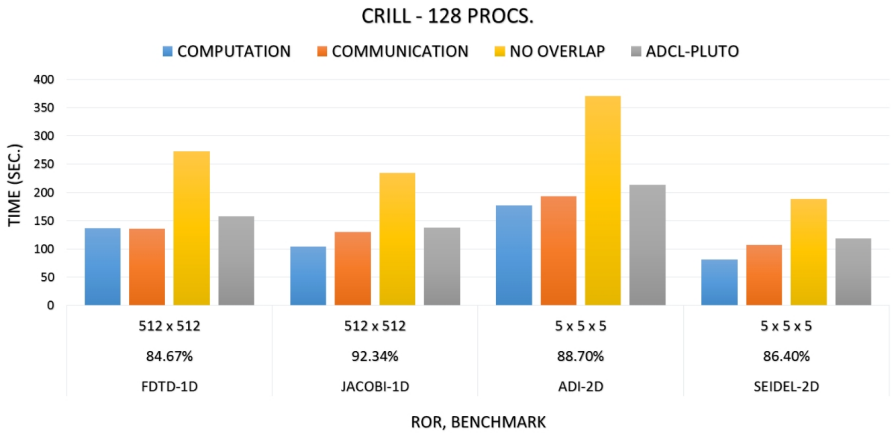

Figure 4.13: Execution times of the four benchmarks on *Crill* using 128 processes (automatic ADCL configuration enabled).

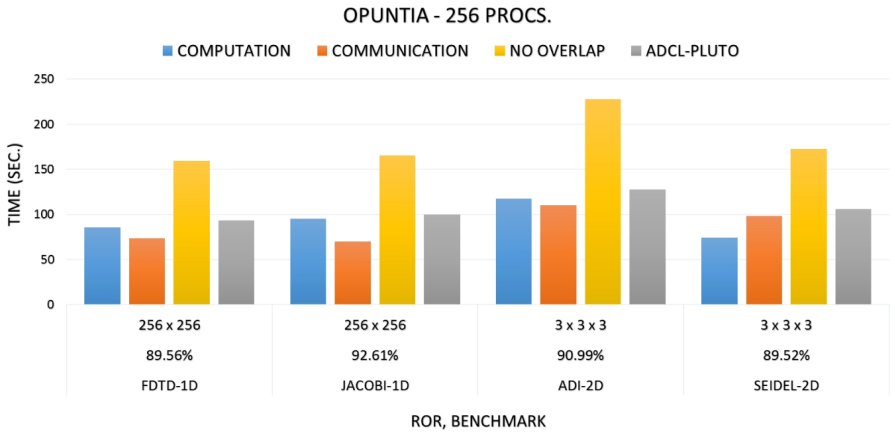

Figure 4.14: Execution times of the four benchmarks on *Opuntia* using 256 processes (automatic ADCL configuration enabled).

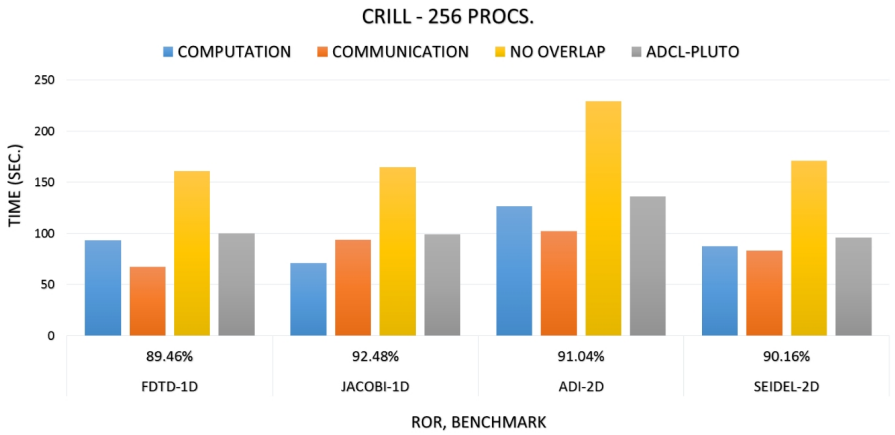

Figure 4.15: Execution times of the four benchmarks on *Crill* using 256 processes (automatic ADCL configuration enabled).

benchmarks.

On the other hand, increasing the tile size drops the tile-occupancy below 2. As a consequence, a subset of the processes becomes unable to overlap its part of the pseudo-neighborhood-communication with the compute-operations. When the tile-occupancy drops below 1, part of the processes becomes completely idle, with no available tile to compute.

The results demonstrate that the ADCL-PLUTO version optimizes execution times, even with enforced and sub-optimal tile sizes, through the automatic ADCL configuration heuristic, as well as the incorporated blocking alternative feature.

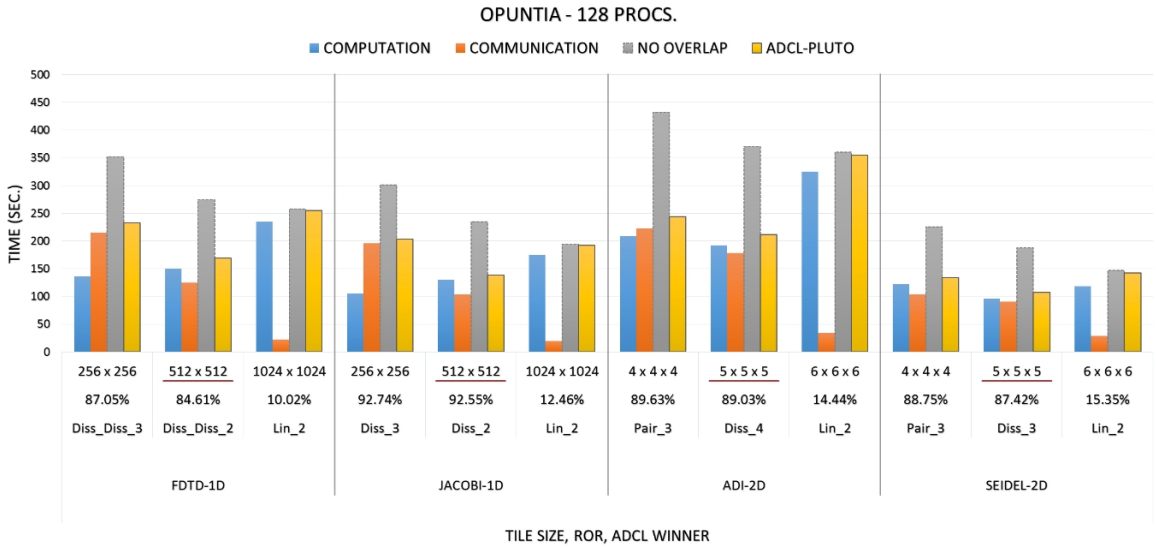

Figure 4.16: Execution times of the four benchmarks on *Opuntia* with additional tile sizes.

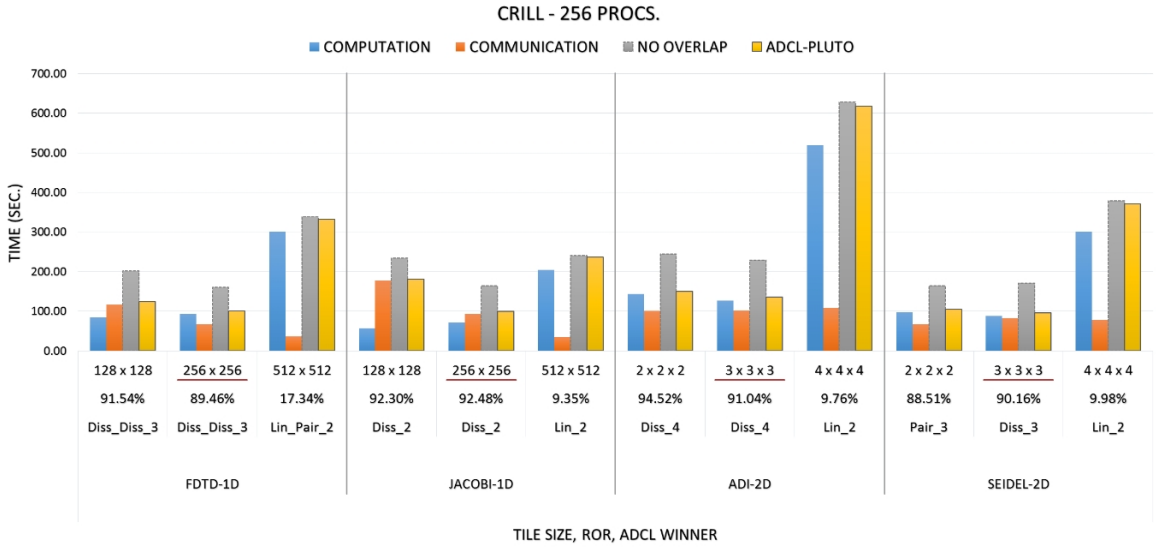

Figure 4.17: Execution times of the four benchmarks on *Crill* with additional tile sizes.

Chapter 5

Conclusions and future work

The work presented in this dissertation was, first, performed within the Abstract Data and Communication Library (ADCL) to support the automatic run-time tuning of non-blocking-collective communication operations. The main contribution of this initial part of the work is the development of solutions which solve the challenges of auto-tuning non-blocking-collective communication, including the timing of the operations, creating a library of algorithms/implementations, parametrization of the implementations, and handling of the progress problem. The results demonstrated, that a library providing a single algorithm or implementation of a non-blocking-collective operation will inevitable be sub-optimal in many scenarios. An analysis showed that the network interconnect as well as application characteristics such as data volumes, number of processes, or the number of progress calls have to be taken into consideration to minimize the execution time of a non-blocking-collective operation. Furthermore, the benefits of the approach were evaluated using an application kernel. The results obtained for the application scenario indicated a

performance improvement of up to 40% compared to the current state of the art.

Next, an approach to automatically optimize the communication-computation overlap was presented. The approach is based on ADCL-PLUTO, an extension of the PLUTO automatic parallelizer incorporating support for non-blocking communication. The main challenges for achieving good overlap of communication and compute operations were identified, followed by the inherent challenges and conceptual extensions necessary in integrating ADCL, which allows to optimize both the underlying algorithms used for the non-blocking-collective operations as well as location and frequency of accompanying progress function calls. Furthermore, guidelines for determining optimal tile sizes to maximize the overlap between communication and compute operations have been developed, along with a heuristic guiding the search algorithm of the ADCL auto-tuning library depending on the number of iterations executed by the application.

The approach has been evaluated with four application benchmarks on two different platforms and multiple process counts. The results indicated significant performance improvements in virtually all test cases. The parallel applications using the ADCL-PLUTO software developed as part of this work achieved a performance improvement in the range of $32 - 43\%$ compared to the version using blocking communication operations, and achieved a relative overlap ratio (ROR) of up to 95% of the maximum theoretical communication-computation overlap identified for each scenario.

There are multiple avenues on how to extend the work presented in this dissertation. First, the neighborhood communication operation could also be converted

to use the new MPI-3 collective operations (`MPI_Neighbor_alltoall`), thus allowing to auto-tune this sequence of communication operations similarly to the Alltoall operation. Second, historic learning methods that already exist in ADCL might be used to perform the auto-tuning during the ramp-up and ramp-down phases as well. Historic learning allows ADCL to utilize knowledge/information gathered in previous executions, avoiding the learning phase and thus potentially reducing the overhead generated in the learning phase. Finally, the solution could be applied and tested with more applications that fit the general pattern supported by PLUTO.

Both ADCL and PLUTO are open source software packages available for download on their corresponding web pages. The modified ADCL-PLUTO version is available for download on the authors webpages.

Bibliography

[1] PLUTO: An automatic parallelizer and locality optimizer for affine loop nests. `http://pluto-compiler.sourceforge.net/`.

[2] S. Agarwal, R. Garg, and N. Vishnoi. The impact of noise on the scaling of collectives: A theoretical approach. In *Proceedings of the 12th Annual IEEE International Conference on High Performance Computing*, 2005.

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. In *Proceedings of the seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105. ACM Press, 1995.

[4] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe. Opentuner: An extensible framework for program autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM.

[5] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the jalapeno jvm. In *ACM SIGPLAN Notices*, page 4765, 2000.

[6] Y. Barigou, V. Venkatesan, and E. Gabriel. Auto-tuning non-blocking collective communication operations. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 1204 – 1213, 2015.

[7] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Springer*, volume 2958, pages 209–225, 2003.

[8] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the*

19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction, pages 283–303, 2010.

[9] S. Benedict, V. Petkov, and M. Gerndt. Periscope: An online-based distributed performance analysis tool. In *Proceedings of the 3rd International Workshop on Parallel Tools for High Performance Computing, 2009, ZIH, Dresden*, pages 1–16. Springer Berlin Heidelberg, 2010.

[10] K. Benkert, E. Gabriel, and M. M. Resch. Outlier Detection in Performance Data of Parallel Applications. In *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing*, 2008.

[11] K. Benkert, E. Gabriel, and S. Roller. Timing Collective Communications in an Empirical Optimization Framework. In *Proceedings of The Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering (PARENG)*. Civil Comp Press, 2011.

[12] S. Bhowmick, V. Eijkhout, Y. Freund, E. Fuentes, and D. Keyes. Application of Machine Learning in Selecting Sparse Linear Solver. 2006.

[13] J. Bilmes, K. Asanovic, C. Chin, and J. Demmel. Optimizing matrix multiply using PHIPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*, Vienna, Austra, 1997.

[14] U. Bondhugula. Compiling affine loop nests for distributed-memory parallel architectures. In *ACM/IEEE Supercomputing (SC '13)*, 2010.

[15] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International Conference on Compiler Construction (ETAPS CC)*, 2008.

[16] U. Bondhugula, A. Hartono, J. Ramanujan, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *ACM SIGPLAN Programming Languages Design and Implementation (PLDI)*, 2008.

[17] G. E. P. Box, W. G. Hunter, J. S. Hunter, and W. G. Hunter. *Statistics for Experimenters: An Introduction to Design, Data Analysis, and Model Building*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, Inc., 1978.

[18] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithn for all-to-all communications in multiport message-passing systems. volume 8, pages 1143–1156, 1997.

[19] M. Burcea and M. Voss. A runtime optimization system for OpenMP. In *WOMPAT*, pages 42–53, 2003.

[20] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki. A tool for optimizing runtime parameters of open mpi. In *Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 210–217, Berlin, Heidelberg, 2008. Springer-Verlag.

[21] I.-H. Chung and J. K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 30. IEEE Computer Society, 2004.

[22] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (PLDI)*, volume 30, pages 279–290, 1995.

[23] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12. ACM Press, 1993.

[24] L. Dagum and R. Menon. OpenMP: A Proposed Industry Standard API for Shared Memory Programming, 1997.

[25] R. Dathathri, C. G, T. Ramashekar, and U. Bondhugula. Generating efficient data movement code for heterogeneous architectures with distributed-memory. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 375–386, 2013.

[26] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, number 4. IEEE Press Piscataway, NJ, USA, 2008.

[27] R. K. Dybvig, M. Leone, and D. Wise. The Dynamo Project: Dynamic optimization via staged compilation. `http://www.cs.indiana.edu/proglang/dynamo/`.

[28] V. Eijkhout, E. Fuentes, T. Eidson, and J. Dongarra. The component structure of a self-adapting numerical software system. volume 33, 2005.

[29] J. J. Evans, C. S. Hood, and W. Gropp. Exploring the relationship between parallel application run-time variability and network performance in clusters. In *28th Annual IEEE Conference on Local Computer Networks (LCN 2003)*, pages 538–547, 2003.

[30] A. Faraj, P. Patarasuk, and X. Yuan. A Study of Process Arrival Patterns for MPI Collective Operations. Springer, 2008.

[31] A. Faraj, X. Yuan, and D. Lowenthal. Star-mpi: self tuned adaptive routines for mpi collective operations. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 199–208, New York, NY, USA, 2006. ACM Press.

[32] P. Feautrier. Dataflow analysis of scalar and array references. In *International Journal of Parallel Programming*, volume 20, pages 23–53, 1991.

[33] P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. In *International Journal of Parallel Programming*, volume 21, pages 313–348, 1992.

[34] P. Feautrier. Some efficient solutions to the affine scheduling problem. part ii. multidimensional time. In *International Journal of Parallel Programming*, volume 21, pages 389–420, 1992.

[35] M. Forum. Mpi: A message passing interface standard version 3.0. `http://www.mpi-forum.org/`.

[36] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. volume 93, pages 216–231, 2005.

[37] 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 of the 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, 2004.

[38] E. Gabriel, S. Feki, K. Benkert, and M. M. Resch. Towards Performance and Portability through Runtime Adaption for High Performance Computing Applications. In *International Supercomputing Conference*, Dresden, Germany, 2008.

[39] E. Gabriel, S. Feki, K. Benkert, and M. M. Resch. Towards Performance Portability through Runtime Adaption for High Performance Computing Applications. volume 22, 2010.

[40] E. Gabriel and S. Huang. Runtime optimization of application level communication patterns. In *12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, Long Beach, CA, USA, 2007.

[41] M. Gerndt. Automatic online tuning. `http://www.autotune-project.eu/`.

[42] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI-The Complete Reference*, volume 2. The MIT Press, 1998.

[43] O. Hartmann, M. Kuhnemann, T. Rauber, and G. Runger. An adaptive extension library for improving collective communication operations. volume 10, page 11731194, 2008.

[44] M. D. Hill and M. R. Marty. Amdahls law in the multicore era. volume 41, pages 33 – 38. IEEE Computer Society, 2008.

[45] R. Hockney. The communication challenge for MPP: Intel Paragon and Meiko CS-2. volume 20, pages 389–398, 1994.

[46] T. Hoefler, P. Gottschling, and A. Lumsdaine. Brief Announcement: Leveraging Non-blocking Collective Communication in High-performance Applications. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'08*, pages 113–115. Association for Computing Machinery (ACM), 2008. (short paper).

[47] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine. A case for standard non-blocking collective operations. In *Proceedings of the 14th European PVM/MPI Users Group Meeting*, pages 125–134. Springer Berlin Heidelberg, 2007.

[48] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnect Networks. In *Proceedings of the IPDPS*. IEEE, 2007.

[49] T. Hoefler and A. Lumsdaine. Message Progression in Parallel Computing - To Thread or not to Thread? In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2008.

[50] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proc. of the 2007 Intl. Conf. on High Perf. Comp., Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, 2007.

[51] G. Inozemtsev and A. Afsahi. Designing an offloaded nonblocking mpi allgather collective using core-direct. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER)*, pages 477 – 485. IEEE, 2012.

[52] T. Ishiyama, K. Nitadori, and J. Makino. 4.45 pflops astrophysical n-body simulation on k computer: the gravitational trillion-body problem. In *Proceedings of the 2012 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE/ACM, 2012.

[53] K. Ishizaka, M. Obata, and H. Kasahara. Cache optimization for coarse grain task parallel processing using inter-array padding. In *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC), Springer*, volume 2958, pages 64–76, 2003.

[54] K. Iskra, P. Beckman, K. Yoshii, and S. Coghlan. The influence of operating systems on the performance of collective operations at extreme scale. In *Proceedings of Cluster Computing*. IEEE International Conference, 2006.

[55] T. Jones, S. Dawson, RobNeely, W. T. Jr., L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts. Improving the scalability of parallel jobs by adding parallel awareness to the operating system. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing*, page 10, 2003.

[56] K. Kandalla, H. Subramoni, K. Tomko, D. Pekurovsky, and D. Panda. A novel functional partitioning approach to design high-performance mpi-3 non-blocking alltoallv collective on multi-core systems. In *Proceedings of the 42nd International Conference on Parallel Processing (ICPP)*, pages 611 – 620. IEEE, 2013.

[57] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, B. de Supinski, and D. Panda. Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers. In *IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*, 2012.

[58] K. C. Kandalla, H. Subramoni, K. A. Tomko, D. Pekurovsky, S. Sur, and D. K. Panda. High-performance and scalable non-blocking all-to-all with collective

offload on infiniband clusters: a study with parallel 3d fft. volume 26, pages 237–246, 2011.

[59] K. C. Kandalla, H. Subramoni, J. Vienne, S. P. Raikar, K. Tomko, S. Sur, and D. K. Panda. Designing non-blocking broadcast with collective offload on infiniband clusters: A case study with HPL. In *IEEE 19th Annual Symposium on High Performance Interconnects (HOTI)*, 2011.

[60] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. volume 5, pages 308–323, 1979.

[61] S. Li, T. Hoefler, and M. Snir. Numa-aware shared-memory collective communication for mpi. In *Proceedings of HPDC*, 2013.

[62] T. Ma, G. Bosilca, A. Bouteiller, and J. J. Dongarra. Kernel-assisted and topology-aware mpi collective communications on multicore/many-core platforms. volume 73, pages 1000–1010, 2013.

[63] N. Manchanda and K. Anand. Non-uniform memory access (numa). 2010.

[64] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, 1995. `http://www.mpi-forum.org`.

[65] Message Passing Interface Forum. *MPI-2.2: Extensions to the Message Passing Interface*, 2009. `http://www.mpi-forum.org`.

[66] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. 2009. http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[67] I. Mishev, N. Fedorova, S. Terekhov, B. Beckner, A. Usadi, M. Ray, and O. Diyankov. Adaptive control for solver performance optimization in reservoir simulation. In *Proceedings of the 11th European Conference on the Mathematics of Oil Recovery*, 2008.

[68] R. Nishtala. *Automatically Tuning Collective Communication for One-Sided Programming Models*. PhD thesis, EECS Department, University of California, Berkeley, 2009.

[69] R. Nishtala and K. A. Yelick. Optimizing collective communication on multicores. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, page 18, 2008.

[70] The OpenUH Compiler Project. `http://www.cs.uh.edu/~openuh`, 2011.

[71] Z. Pan and R. Eigenmann. Peaka fast and effective performance tuning system via compiler optimization orchestration. volume 30, 2008.

[72] Parallel Software Technologies Laboratory. *ADCL: Abstract Data and Communication Library, User Level API Functions*, 2007. http://pstl.cs.uh.edu/projects/adcl-spec.pdf.

[73] F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: Achieving optimal performance on the 8, 192 processors of asci q. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing*, page 55. ACM, 2003.

[74] F. Petrini, D. J. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[75] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. volume 10, pages 127–143, 2007.

[76] A. Rahimian, I. Lashuk, S. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society Press, 2010.

[77] G. Roth, G. Roth, J. Mellor-crummey, J. Mellor-crummey, K. Kennedy, K. Kennedy, R. G. Brickner, and R. G. Brickner. Compiling stencils in high performance fortran. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–20. ACM New York, NY, USA, 1997.

[78] P. Sack and W. Gropp. Faster topology-aware collective algorithms through non-minimal communication. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 45–54, New York, NY, USA, 2012. ACM.

[79] M. Shroff and R. A. van de Geijn. Mpi collective communication benchmark.

[80] S. Song and J. K. Hollingsworth. Designing and Auto-Tuning Parallel 3-D FFT for Computation-Communication Overlap. In *Proceedings of the 19th ACM*

Symposium on Principles and Practices for Parallel Programming (PPOPP'14), 2014.

[81] W. R. Stevens and S. A. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2005.

[82] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In J. Dongarra, D. Laforenza, and S. Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 2840 in LNCS, pages 257–267. Springer Verlag, 2003. 10th European PVM/MPI User's Group Meeting, Venice, Italy.

[83] A. Tiwari, J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame. Auto-tuning full applications: A case study. volume 25, pages 286–294, Thousand Oaks, CA, USA, 2011. Sage Publications, Inc.

[84] S. Vadhiyar, G. Fagg, and J. Dongarra. Towards an Accurate Model for Collective Communications. In *Proceedings of International Conference on Computational Science (ICCS 2001)*, San Francisco, USA, 2001.

[85] B. Venners. The Hotspot Virtual Machine. First Published in Developer.com, 1998.

[86] M. J. Voss and R. Eigenmann. ADAPT: Automated De-coupled Adaptive Program Transformation. In *International Conference on Parallel Processing*, page 163, 2000.

[87] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. volume 16, 2005.

[88] R. C. Whaley and A. Petite. Minimizing development and maintenance costs in supporting persistently optimized blas. volume 35, pages 101–121, 2005.