Auto-tuning Non-blocking Collective Communication Operations

Youcef Barigou, Vishwanath Venkatesan, and Edgar Gabriel

Department of Computer Science,
University of Houston, Houston, TX 77204-3010, USA.
Email: {ybarigou, venkates, gabriel}@cs.uh.edu

Abstract—Collective operations are widely used in large scale scientific applications, and critical to the scalability of these applications for large process counts. It has also been demonstrated that collective operations have to be carefully tuned for a given platform and application scenario to maximize their performance. Non-blocking collective operations extend the concept of collective operations by offering the additional benefit of being able to overlap communication and computation. This paper presents the automatic run-time tuning of non-blocking collective communication operations, which allows the communication library to choose the best performing implementation for a non-blocking collective operation on a case by case basis. The paper demonstrates that libraries using a single algorithm or implementation for a non-blocking collective operation will inevitably lead to suboptimal performance in many scenarios, and thus validate the necessity for run-time tuning of these operations. The benefits of the approach are further demonstrated for an application kernel using a multi-dimensional Fast Fourier Transform. The results obtained for the application scenario indicate a performance improvement of up to 40% compared to the current state of the art.

I. INTRODUCTION

Collective communication operations have been a key concept used in large scale parallel applications to minimize communication costs. The interface specification of collective operations represent a higher level abstraction for often occurring communication patterns, separating the desired outcome of the group communication from its actual implementation. To highlight the benefits of collective operations, 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. Not surprisingly, developers of 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 [19, 21] or exploiting capabilities of the network adaptors[18].

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. 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 collective operations due to (micro) load imbalances between the processes [7]. In [12] we demonstrated that an implementation which minimized the execution time of a collective operation on one platform and for one particular application scenario can show 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 suboptimal 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 [12] and STAR-MPI [8] 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.

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 due to the fact that application developers do not have to worry about scheduling of messages to avoid deadlock scenarios. However, it is not
necessarily straightforward 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 due to the fact that 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. [16] has however extended non-blocking operations to collective communication as well, and has recently been adopted by the MPI standard [9].

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.

In this paper we present the work performed within the ADCL library to support the automatic run-time tuning of non-blocking collective communication operations. The focus within the context of this paper is on two collective operations, namely non-blocking broadcast and non-blocking all-to-all communication. We demonstrate the variability of the best implementation of a non-blocking collective operation depending on network and application characteristics and the benefits of this approach for an application kernel using a multidimensional Fast Fourier Transform (FFT) on two platforms for multiple process counts and data volumes. The results obtained for the application scenario indicate a performance improvement of up to 40% compared to the current state of the art.

The paper is organized as follows: in section II we discuss the most relevant related work in auto-tuning of HPC applications, focusing on auto-tuning libraries for communication operations. Section III presents the extensions to the ADCL library introduced to support the run-time tuning of non-blocking collective operations. The performance of the auto-tuned non-blocking communication operations is presented in section IV for micro-benchmark as well as an application scenario. Finally, section V summarizes the paper and presents the ongoing work in this area.

II. RELATED WORK

Significant efforts have been devoted to the development of new algorithms to improve the performance of collective operations, including non-blocking collective operations [17, 18]. However, there is to the best of our knowledge no comprehensive study of tuning non-blocking collective operations available as of today. The fundamental problem in the analysis of non-blocking collective operations stems from the fact, that it is virtually impossible to capture in a model such as LogP [6] the ‘visible’ part of the operation.

A number of auto-tuning projects are based on a-priori 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) [24] and the Automatically Tuned Collective Communications (ATCC) [20] 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. These projects face however 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.

Among the projects using runtime tuning techniques in HPC are FFTW [10] and PhiPAC [3], each focusing on one particular operations (Fast Fourier Transform and Matrix Multiply respectively). STAR-MPI [8] provides runtime tuning of collective communication operations using an API similar to MPI. However, none of these projects has so far investigated the challenges and benefits of tuning non-blocking collective operations.

There are a number of generic auto-tuning frameworks such as OpenTuner [1] and Active Harmony [23]. The closest work related to this paper that we are aware of has been performed by the latter project by tuning of 3D Fast Fourier Transforms using non-blocking all-to-all operations [22]. The main difference between the work presented in this paper and our work is, that we focus on the tuning of the non-blocking All-to-all operation, while [22] focuses on tuning the sequence of communication and computation to maximize the overlap.

III. CONCEPT

A. The Abstract Data and Communication Library

The Abstract Data and Communication Library (ADCL) [12] is an auto-tuning library for collective communication operations. ADCL provides for each supported operation a large number of implementations. Transparent to the user, the library incorporates a runtime selection logic in order to choose the implementation leading to the highest performance. For this, ADCL switches internally during the first iterations of the applications the implementation used 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 runtime selection logic chooses the implementation leading to the lowest overall execution time, and uses this version for the rest of the execution.

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 version 2.0 of the library, high-level interfaces have been introduced 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 actual communication operation can be initiated using ADCL Request start (blocking execution) or ADCL Request init / ADCL Request wait for non-blocking operations. Note that it would be possible to extend the concept of ADCL to non-persistent interfaces for many collective operations, but not for all. Since ADCL relies on the repetitive execution of the same operation, an implementation for non-persistent collective operations would have to rely on hashing the arguments of the function call to correctly identify a particular operation. This is however not possible for the vector version of the MPI collective operations, since each process only knows its own arguments, and its impossible to reliably identify an operation based on the local arguments. Hence, ADCL decided to support only persistent operations at this point in time. In addition, ADCL also provides low-level interfaces which can be used by end-user applications to register its own functionality for auto-tuning using ADCL and thus make use of the ADCL selection logic, statistical filtering etc.

ADCL incorporates multiple runtime selection algorithms. A brute force search strategy evaluates all available implementations before choosing which implementation 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 runtime heuristic based on attributes characterizing an implementation has been developed [13]. 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 [4]. 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 runtime selection is however mostly useful for very large parameter spaces such as described in [5], and is therefore omitted for the rest of this paper.

B. LibNBC

LibNBC [16] is a library providing non-blocking collective communication operations and is used in most MPI libraries as the basis for the implementation of the new non-blocking collective communication operations defined in version three of the MPI specification. 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 (locally) 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 algorithms.

C. Auto-tuning Non-blocking Communication Operations

As discussed in section II, 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 however highly challenging for two reasons. First, the actual time spent in the communication operation can not be directly measured due to the fact that 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 [15] if the MPI library does not utilize a separate thread for this purpose. Auto-tuning offers for this scenario the 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 an 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 describing a particular characteristic of an implementation. Typical attributes used in existing function-sets are the algorithm used to implement the operation (e.g. linear, binary tree, etc.), 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 discontiguous data (e.g. pack/unpack, derived data types, etc.). Note, that a large number of attributes and attribute values allow for a more fine 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.

In the following, we discuss the solution to the timing problem outlined above, give a description of the non-blocking function-sets in ADCL along with the interface developed to tackle the progress problem.
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);
}

Fig. 1. Code sample using the ADCL High Level API.

D. 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 allows to transform the problem of tuning a collective operation into the tuning of a larger code section, which can include both communication and compute operations [2]. The user (or an automated tool, which is not discussed in this paper) 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 for the current function executed from the function set which has been associated with the timer object. The following code sample shows a non-blocking all-to-all operation using the ADCL syntax.

E. Non-blocking Function-sets

Non-blocking operations separate the initiation and the completion of the operation, allowing to overlap 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 we restrict the non-blocking collective operations to be restricted to non-overlapping sub-communicators.

A newly introduced ADCL progress function triggers the progress engine of the LibNBC library and thus the underlying MPI implementation. As discussed in [16], 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 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 installations at the moment do not operate using a progress thread. Thus, calling into 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 then in turn calls the MPI library.

For non-blocking collective operations, each function implementing a non-blocking operation is represented by a particular LibNBC schedule. For this, we converted existing implementations of the MPI_Bcast, MPI_Reduce, MPI_Allgather, and MPI_Alltoall operations in Open MPI [11] to a LibNBC schedule. Within the context of this paper, 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, and a pair-wise exchange algorithm. We choose at the moment not to provide further versions of this operation, although a further distinction based on data transfer primitives (i.e. Put/Get vs. Isend/Irecv) could be added later on.

IV. Performance Evaluation

In the following section we evaluate the impact of auto-tuning non-blocking collective communication operations. 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.
The second cluster is the whale cluster, which consists of 64 nodes with two quad-core AMD Opteron (Barcelona) processors, 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. We refer to the platform in that case as whale-tcp. The 1.6 series of Open MPI [11] was used in all instances.

A. Results using Micro-Benchmarks

For the first set of tests, a micro-benchmark has been developed which allows to evaluate 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 being invoked. The compute operation is thus 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 runtime selection logic in ADCL, and are referred to as the 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 runtime 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 thus had the same node assignments. Tests have been executed for 32, 128 and 256 processes, (the latter on crill only) for 1KB and 2 MB message length for ADCL_Ibcast and 1KB and 128KB per process-pair for ADCL_Ialltoall; where the execution times correspond to the execution of the entire loop. During each verification run, the benchmark measures the time it takes to execute between 1,000 iterations (for long message lengths, i.e. 128KB and 2 MB) and 10,000 iterations (for short message lengths, i.e. 1KB).

Figure 2 shows two representative verification runs obtained for ADCL_Ialltoall on whale using 128 processes with 128KB message length per process pair and 50s 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.

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, we define the "correct winner function" as an implementation of the operation which 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 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 are 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 thus introduces an overhead compared to using the 'optimal' implementation only. However, the additional costs due to the learning phase are not relevant for long running applications. Ultimately, the results of the verification runs show that the concepts and interfaces developed within this work allow to tune non-blocking collective operations at run time.

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

a) Influence of the network characteristics: The result shown in this paragraph highlight the influence of the network interconnect on the best performing implementation of a non-blocking collective operation. Fig. 3 depicts the results obtained for ADCL_Ialltoall tests executed with 32 processes, 128KB message length per pair of processes, and 50s computational time. The results shown in the right part of fig. 3 are using the whale cluster using the InfiniBand network, while the tests shown in the left part of fig. 3 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 application and hardware parameters except for the network are identical. Specifically, the implementation using the linear algorithm shows the best performance for the whale cluster in multiple instances (for 5, 10 and 100 progress calls), but does very poorly on whale-tcp and is typically the worst choice on that platform. The results for ADCL_Ibcast are very similar, but omitted here due to space limitations.

b) Influence of the communication volumes: The tests shown in fig. 4 demonstrate the influence of the message length on the optimal implementation. These two figures show as an example the results obtained on the crill cluster for 256 processes using 1KB and 128KB message length for 10s of compute time for the ADCL_Ialltoall operation. Results obtained with ADCL_Ibcast were similar, but are not shown here for brevity. The dissemination algorithm is the best choice for the 1KB message length, but the worst choice for the 128KB scenario. On the other hand, the linear and pairwise algorithm perform poorly in the 1KB scenario, but very well on the 128KB scenario. Thus, our tests demonstrate that the message length has a significant influence on the choice for the best implementation.

c) Influence of the number of processes used: To demonstrate the effects of the number of process on the implementation leading to a minimal execution time, consider the results shown in fig. 5 which were obtained on the whale cluster for 1KB message length and 10s compute time for the non-blocking All-to-all operation. The difference between the two graphs is the number of processes used, namely 32 in the first case, and 128 in the second case. We see once again a large variability on which algorithm performs well depending on the number of processes used, with linear and pairwise performing poorly for 32 processes and very good for 128 processes on this platform, while the dissemination algorithm based implementation of ADCL_Ialltoall performed well for 32 processes but poorly for 128 processes. This general trend has been confirmed in numerous tests for both All-to-all and Ibcast.

d) Influence of the number of progress calls: For most scenarios, the ability to overlap communication and computation increases with increasing number of progress calls. For many application scenarios, the number of progress calls that can be inserted is given by the code structure, e.g. 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. Fig. 6 demonstrates a scenario where increasing the number of progress calls does in fact reduce the performance of the benchmark.

Finally, one should also note that the number of progress calls also has influence on the algorithm leading to 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. Fig. 7 shows a scenario observed on the crill cluster, in which the pairwise algorithm delivers the best performance of the Ialltoall operation in case that only a single progress call could be inserted into the code sequence, while the linear algorithm does typically best if more than just one progress call could be executed.

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.

B. Results using an Application Kernel

In the following, we present the benefits of using run-time tuning for non-blocking collective operations with an application kernel, which uses a three dimensional Fast Fourier Transform (FFT). The application benchmark was adopted by Hoefler et al. to utilize non-blocking collective all-to-all operations [14], with multiple different versions being available to implement the multi-dimensional FFT. Specifically, the sequence of calculations and communications are implemented in a pipelined, tiled, windowed, and window-tiled manner. 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 our case, we considered the default tile size [14] of the benchmark which is set to 10 for the tiled and window-tiled implementations, and a window size of 3 for the windowed and window-tiled implementations.

We executed a large number of tests on both whale and crill, omitted whale-tcp however due to the low performance and high execution time of these test-cases over Gigabit Ethernet. Instead, we executed several tests on an IBM Bluegene/P located at KAUST Supercomputing Laboratory. 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. We are omitting here the last two arguments for the sake of clarity, we would like to note however, that 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.

e) Comparison to LibNBC: In the first set of tests, the performance obtained with the ADCL versions of the code is compared to the versions using non-blocking communication based on the LibNBC library. Our results indicate that ADCL outperforms the LibNBC version in the vast majority of the test cases, for all four patterns of the 3-D FFT operation. Specifically, out of 393 tests performed, ADCL reduced the execution time compared to the LibNBC version in 74% of the test cases. Fig. 9 shows some of the results obtained on the crill cluster using 160 and 500 processes. In the vast majority of the remaining 24% of the cases, the performance of LibNBC and ADCL was on par, in very few scenarios, e.g. such as two scenarios using 160 processes shown in fig. 9. The performance benefits can be explained by the fact, that LibNBC only supports a single implementation of the non-blocking all-to-all operation by default, namely the linear algorithms. Analyzing the scenarios where LibNBC outperformed ADCL revealed that 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 in this case once again stems from the overhead of testing the suboptimal algorithms during the learning phase of ADCL.

f) Comparison to blocking operations: In the next set of tests, we added a version of the 3-D FFT operation which utilized a blocking MPI_Alltoall operation for the communication, thus not attempting to overlap computation and communication. The results obtained on whale with the different versions of the FFT application kernel using MPI, LibNBC and ADCL are displayed in fig. 10 and fig. 12.

While ADCL outperforms once again 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, we modified the ADCL_Ialltoall function-set to also include blocking operations. As mentioned in section III-C, 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 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, and there are no simultaneous non-blocking collective operations with partially overlapping communicators (see, page 221 in MPI-3 [9] for an example). 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, we separated the timing of the selection phase of ADCL from the rest of the execution, and made a similar modification to the MPI version in order to measure the same number of iterations in both scenarios. The graphs in fig. 11 and fig. 12 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 neglecting the performance benefits. Nevertheless, for applications that involve larger number of iterations, the cost of the learning phase of ADCL is less relevant and incorporating the blocking All-to-all algorithms into the non-blocking Ialltoall function-set could be advantageous.

To summarize the findings of this subsection, auto-tuning non-blocking collective operations showed in the vast majority of the test cases executed significant performance benefits for the application kernel. In the few instances where non-blocking operations did not show initially benefits compared to blocking operations, ADCL was still able to provide performance on par with the best performing implementation found once the decision logic has finished the evaluation, which for longer running applications would still be beneficial. In addition, this provides also an interesting aspect of the historic learning feature of ADCL, which allows to transfer information across different executions of ADCL, the goal being to shorten the time spent in the tuning phase.

V. Conclusions

This paper presented the work 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 the paper is the development 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 demonstrate, that a library providing a single algorithm or implementation of a non-blocking collective operation will inevitably be suboptimal in many scenarios. Our analysis shows 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, we evaluated the benefits of our approach using an application kernel. The results obtained for the application scenario indicate a performance
Fig. 9. Execution time of various patterns used for the 3D FFT operation using LibNBC and ADCL on crill for 160 processes (right) and 500 processes (left).

Fig. 10. Execution time of various patterns used for the 3D FFT operation using LibNBC, ADCL and MPI for 160 processes (right) and 358 processes (left).

Fig. 11. Execution time of various patterns used for the 3D FFT operation using the modified ADCL function-set and MPI on whale for 160 processes (right) and 358 process (left).

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

This work can be extended in multiple directions. One of the interesting features not yet explored in this work is the ability of the ADCL timer object to co-tune multiple operations simultaneously, since the algorithmic choice for one non-blocking operation could have an effect on the performance of another operation. Furthermore, we plan to extend the investigation presented in this paper on deciding at runtime whether to use a blocking or a non-blocking version of a collective operation.

Acknowledgments. Partial support for this work was provided by the National Science Foundation’s Computer Systems Research program under Award No. CNS-0846002 and CRI-0958464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We would like to thank the KAUST Supercomputing Laboratory for giving us access to their IBM BlueGene/P.
using the modified ADCL function-set and MPI on Bluegene/P

Fig. 12. Execution time of various patterns used for the 3D FFT operation

REFERENCES


