# Runtime Optimization of Application Level Communication Patterns

Edgar Gabriel and Shuo Huang
Department of Computer Science , University of Houston ,
Houston, TX, USA
{gabriel, shhuang}@cs.uh.edu

*Abstract*— **This paper introduces the Abstract Data and Communication Library (ADCL). ADCL is an application level communication library aiming at providing the highest possible performance for application level communication operations on a given execution environment. The library provides for each communication pattern a large number of implementations and incorporates a runtime selection logic in order to choose the implementation leading to the highest performance of the application on the current platform. Two different runtime selection algorithms are currently available within ADCL: the library can either apply a brute force search strategy which tests all available implementations of a given communication pattern; alternatively, a heuristic relying on attributes characterizing an implementation has been developed in order to speed up the runtime decision procedure. The paper also evaluates the performance of a finite difference code using ADCL on an AMD Opteron cluster using InfiniBand and Gigabit Ethernet interconnects.**

## I. Introduction

Software development for High Performance Computing systems is currently facing significant challenges, since many of the software technologies applied in the last ten years have reached their limits. The number of applications being capable of efficiently using several thousands of processors or achieving a sustained performance of multiple teraflops is very limited and is usually the result of many person-years of optimizations for a particular platform. These optimizations are however often not portable. As an example, an application optimized for the IBM Blue Gene will very probably perform poorly on a Cray X1 or the Earth Simulator, and will probably even have performance problems on a PC cluster utilizing a commodity network interconnect. Among the problems application developers face are the wide variety of available hardware and software components and their influence on the performance of an application, such as processor type and frequency, number of processor per node and number of cores per processor, characteristics and performance of the network interconnect, or software components such as the operating system, device drivers and communication libraries. Hence, an end-user faces a unique execution environment on each parallel machine he uses. Even experts struggle to fully understand correlations between hardware/software parameters of the execution environment and their effect on the performance of a parallel application.

In the following, we would like to clarify the dilemma of an application developer using a realistic and common example. Consider a regular 3-dimensional finite difference code using an iterative algorithm to solve the resulting system of linear equations. The parallel equation solver consists of three different operations requiring communication: scalar products, vector norms and matrix-vector products. Although the first two operations do have an impact on the scalability of the algorithm, the dominating operation from the communication perspective is the matrix-vector product. The occurring communication pattern for this operation is neighborhood communication, i.e. each process has to exchange data with its six neighboring processes at least once per iteration. Depending on the execution environment and some parameters of the application (e.g. problem size), different implementations for the very same communication pattern can lead to optimal performance. Figure 1 presents the execution times for 200 iterations of the equation solver applied for a steady problem using 32 processes on the same number of processors on a state-of-the-art PC cluster for two different problem sizes ($32 \times 32 \times 32$ and $64 \times 32 \times 32$ mesh points per process) and two different network interconnects (Infiniband and Gigabit Ethernet). The neighborhood communication has been implemented in four different ways, named here *fcfs, fcfs-pack, ordered, overlap*.

The results indicate, that already for this simple testcase on a single platform three different implementations of the neighborhood communication lead to the best performance of this application: the *fcfs* implementation shows the best performance for both problem sizes when using the Infiniband interconnect. This implementation is initiating all required communications simultaneously using asynchronous communication. However, for the Gigabit Ethernet interconnect the *fcfs* approach seems to congest the network. Instead, the implementation which is overlapping communication and computation

(*overlap*), is showing the best performance for the small problem size while the *ordered* algorithm, which limits the number of messages concurrently on the fly, is the fastest implementation for the large problem size for this network interconnect. Thus, the implementation that was considered to be the fastest one over the Infiniband network has turned out to be under certain circumstances the slowest of the available implementations over Gigabit Ethernet. An application developer implementing the neighborhood communication using a particular algorithm will inevitably give up performance on certain platforms.

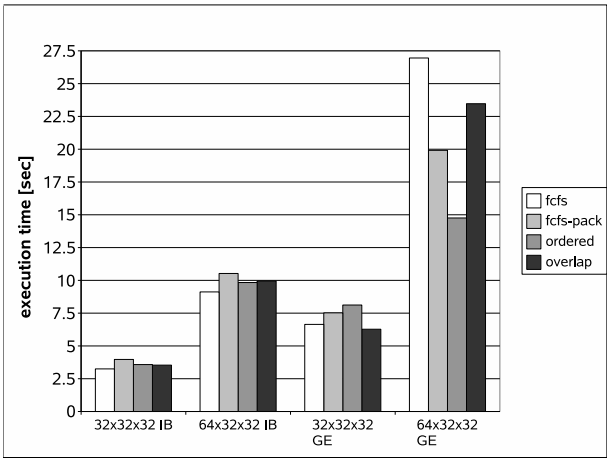


Fig. 1. Comparison of the execution times for various implementations of the neighborhood communication in a regular finite difference code, on 32 processors using Open MPI 1.0.1

In the last couple of years many projects have been dealing with optimizing collective communication operations. Most of the work has focused on collective operations as defined in the MPI specifications, such as in [1]–[7]. While all of them incorporate a certain flexibility with respect to the algorithm used to implement a given collective operation, most of them determine a-priori which algorithm will be applied for a particular message length. Star-MPI [5] is the only approach known to the authors which incorporates a runtime decision logic.

Among the numerical libraries incorporating adaptive techniques are ATLAS [8] and FFTW [9]. The probably most flexible approach is offered by FFTW. This library incorporates runtime optimizations of Fast Fourier Transform (FFT) operations. In order to compute an FFT, the user has to invoke first a 'planner' specifying a problem which has to be solved. The planner measures the actual runtime of many different implementations and selects the fastest one. In case many transforms of the same size are executed in an application, this 'plan' delivers the optimal performance for all subsequent FFTs.

In this paper we present a new application level communication library which enables the runtime optimization of collective operations. The library provides two runtime selection algorithms in order to determine the implementation leading to the best application performance at runtime. We demonstrate the benefits of this approach using a finite difference code over InfiniBand and Gigabit Ethernet. The remainder of the paper is organized as follows: section II presents the main ideas and the main concepts of the Abstract Data and Communication Library (ADCL). Section III presents then a heuristic for the runtime selection logic based on attributes characterizing an implementation. Section IV details performance results and discusses current limitations of the library and of the algorithms used for the runtime selection logic. Finally, section V summarizes the paper and presents the currently ongoing work in this project.

## II. The Abstract Data and Communication Library (ADCL)

ADCL ( Abstract Data and Communication Library) is an application level communication library aiming at providing the highest possible performance for application level communication patterns within a given execution environment. In this context, an application level communication pattern is defined as a repeatedly occurring communication operation incorporating a group of processes. Typical examples are the 2-D or 3-D neighborhood communication occurring in many applications that are based on regular domain decomposition. Although not necessarily implemented using MPI collective operations, these communication patterns are collective by their nature, and usually dominate the time spent in communication in the according applications.

The ADCL API offers high level interfaces of application level collective operations. The high level interfaces are required in order to be able to switch within the library the implementation of the according collective operation without modifying the application itself. Thus, ADCL complements functionality available within MPI. The main objects within the ADCL API are:

1) an `ADCL_Topology` object, which provides a description of the process topology and neighborhood relations within the application. In the example shown below the creation of this object is based on a concept defined in the MPI-1 specification, cartesian communicators.
2) an `ADCL_Vector` object, which specifies the data structure to be used during the communication and the actual data. The user can for example *register* a data structure such as a vector or a matrix with the ADCL library, detailing how many dimensions the object has, the extent of each dimension, the

number of halo-cells, the basic datatype of the object, and the pointer to the data array of the object.
3) an `ADCL_Request` object, which combines the process topology and a vector object, and is thus capable of determining which elements of the vector object have to be sent to which neighboring process.

Another useful feature of the ADCL library is its ability to 'share' performance data between different requests. Two conditions have to be fulfilled in order to share the performance data between two requests: first, they have to use the same process topology; second the vector objects used to construct the requests have to be of identical dimensions.

### A. Runtime selection logic

A key concept of the adaptive communication framework is its ability to select the fastest of the available implementations for a given communication pattern during the regular execution of the application. The approach chosen by ADCL is to use the first $n$ iterations of the application to determine the fastest available implementation. Although some of the tested implementations will deliver a suboptimal performance, this approach avoids a separate 'planner' step.

The algorithm used within ADCL to determine the fastest available implementation relies on a brute force search strategy. This approach tests all available implementations multiple times. Each process keeps track of the execution time(s) of each implementation in an data array which is attached to the according `ADCL_Request`. After all implementations have been tested, all processes have to agree collectively on the implementation which will be used for the rest of the application. The according algorithm consists of four steps:

1) Filtering of the execution times in order to exclude incidental outliers. Right now, an outlier is defined as a value which is $x$ times higher than the minimal value measured for the very same implementation. However, a measurement is only considered to be an outlier and thus removed from performance data of the according implementation, if the fraction of number of outliers to total number of available measurements for the same implementation does not exceed a certain threshold. In case the number of outliers generated by an implementations is non negligible, the library assumes this to be a property of the implementation on the current execution environment.
2) Each process determines the average execution time for each implementation using the filtered list of measurements.
3) All processes determine collectively for each implementation the maximum average execution time across all processes using an `MPI_Allreduce`.
4) Each process determines individually which implementation has the lowest maximum average execution time.

Assuming that the runtime environment produces reproducible performance data over the lifetime of an application, the brute force search is guaranteed to find the fastest of available implementation for the current tuple of {problem size, runtime environment }. Furthermore, the algorithm requires only a single (additional) collective operation during the entire runtime decision procedure.

The major drawback of this approach is the time it might take to determine the fastest implementation. According to our experience on various platforms, the library requires between 10 and 30 measurements per implementation in order to have reliable performance data. Taking into account that the library might have to test up to twenty different implementations, up to 600 iterations might be required before the runtime selection logic comes up with a final decision. Although the capability of ADCL to share performance data between multiple requests speeds up the decision procedure in real-world applications, adaptive applications with varying problem sizes would require a significantly faster procedure in order for ADCL to become useful for this class of applications.

## III. A Runtime Selection Logic Based on Performance Hypothesis

Any implementation of a collective communication operation has certain implicit requirements to the hardware and software environment in order to achieve the expected performance. As of today, ADCL uses three attributes in order to characterize an implementation:

1) Number of simultaneous communication partners: this attribute characterizes how many communication operation are initiated at once. The currently supported values by ADCL are all ( ADCL attribute value `aao`) and one (`pair`). Please note, that for other communication patterns such as a broadcast operation, this attribute might characterize whether the broadcast is implemented using a binary tree or a flat tree. This parameter is typically bound by the network/switch.
2) Handling of non-contiguous messages: supported values are MPI derived data types (`ddt`) and pack/unpack (`pack`). The optimal value for this parameter will depend on the MPI library and some hardware characteristics.

3) Data transfer primitive: a total of eight different data transfer primitives are available in ADCL as of today, which can be categorized as either blocking communication (e.g. `MPI_Send`, `MPI_Recv`), non-blocking/asynchronous communication (e.g. `MPI_Isend`, `MPI_Irecv`), or one-sided operations (e.g. `MPI_Put`, `MPI_Get`). Which data transfer primitive will deliver the best performance depends on the implementation of the according function in the MPI library and potentially some hardware support (e.g. for one-sided communication).

Please note, that not all combinations of attributes can really lead to feasible implementations. As an example, implementations using a blocking data transfer primitives such as SendRecv can not be applied for implementations having more than one simultaneous communication partner. Therefore, a total of 20 implementations are currently available within ADCL for the n-dimensional neighborhood communication. Further attributes such as the capability of the library/environment to overlap communication and computation will be added in the near future.

In order to speed up the selection logic, an alternative runtime heuristic based on the attributes characterizing an implementation has been developed. The heuristic is based on the assumption, that the fastest implementation for a given problem size on a given execution environment is also the implementation having 'optimal' values for the attributes in the given scenario. Therefore, the algorithm tries to determine the optimal value for each attribute used to characterize an implementation. The heuristic guided by performance hypothesis is pruning the search space by assuming an optimal solution is comprised of optimal subcomponents. Once the optimal value for an attribute has been found, the library removes all implementations not having the required value for the according attribute and thus shrinks the list of available implementations.

An implementation is characterized by $N$ attributes. Each attribute has $n(i), i = 1, N$ possible values. The library assumes that the optimal value $k(j)_{opt}$ for an attribute $j$ has been found, if $reqconf(j), i = 1, N$ measurements confirm this hypothesis. In order to be able to deduct from a set of measurements towards the optimal value of a single attribute, the library only compares the execution times of implementations whose attributes differ only in the according attribute.

To clarify this approach, please assume that we want to determine the best value for the second attribute. We assume, that this attribute can have all-in-all three distinct values, e.g. 1, 2, and 3. Assuming that we have to deal with four attributes characterizing an implementation, the library collects first the performance data of the implementations with the attribute values as shown in Table I.

| | Impl. 1 | Impl. 2 | Impl. 3 |
|---|---|---|---|
| Value for attr. 1 | x | x | x |
| Value for attr. 2 | 1 | 2 | 3 |
| Value for attr. 3 | y | y | y |
| Value for attr. 4 | z | z | z |

TABLE I
ATTRIBUTE VALUES FOR THREE HYPOTHETICAL IMPLEMENTATIONS OF A COLLECTIVE OPERATION.

Since the values of all attributes except for the second one are being constant we assume that any performance differences between the four implementations can be denoted to the second attribute. The library determines collectively across all processes which of the four implementations has the lowest average execution time, using the same approach as outlined in the brute force section. If we assume as an example, that the implementation with the attribute values $[x, 3, y, z]$ has the lowest average execution time, the library would develop for the second attribute the hypothesis that 3 is its optimal value on this platform for the given problem size. At this point, only one set of measurement confirms the hypothesis that 3 is the optimal value for the second attribute. Thus, the confidence value in this hypothesis is set to 1. Typically, a hypothesis has to be confirmed by more than one set of measurements before ADCL considers this hypothesis to be probably correct. Thus, an additional set of measurements with differing (but constant) values for one of the other attributes has to be gathered, e.g by using $y + 1$ as the value for the third attribute.

If the new set of measurements confirms the result of the previous set, the confidence value for the hypothesis is increased. If another attribute value is determined for this set of measurements to be the best one, the confidence value for the original performance hypothesis is decreased and the hypothesis potentially even removed/nullified, if the confidence value reaches zero. Please note, that if the measurements do not converge towards an optimal value for an attribute, no implementation will be removed based on this attribute. Once a hypothesis reaches however the required number of confirmations, the library removes all implementations which have not the optimal value for the according attribute and thus shrinks the list of implementations which have to be evaluated.

### A. Optimizations for multi-attribute scenarios

Compared to the brute force search outlined in section II-A, the approach relying on performance hypothesis can significantly reduce the number of implementations which have to be tested and thus speed up the

overall decision procedure. However, the algorithm also contains some inefficiencies compared to the brute force search. As an example, the runtime decision algorithm based on the brute force search requires only a single collective communication operation. The approach using performance hypotheses requires an `MPI_Allreduce` operation every time the performance data of a set of implementations is available and the data needs to be evaluated in order to determine the best value for an attribute.

Please note furthermore, that in the example shown in the last subsection, performance data for the implementations having the attributes $\{x, 1, y, z\}, \{x, 2, y, z\}, \{x, 3, y, z\}$, and $\{x, 1, y + 1, z\}, \{x, 2, y + 1, z\}, \{x, 3, y + 1, z\}$, might be available at a certain point in time. Since we were only interested in the second attribute, the data of the first three implementations and of the second three implementations have been compared in order to determine the optimal value for the second attribute. However, in case the third attribute has only two possible values, it would also make sense to compare $\{x, 1, y, z\}$ with $\{x, 1, y + 1, z\}$, $\{x, 2, y, z\}$ with $\{x, 2, y + 1, z\}$, and so on. The problem however is, that the runtime selection logic might have removed already one/some of the according implementation(s) if the confidence value for the first or the second attribute has reached the required confidence value.

In order to overcome these two limitations, the runtime selection algorithm using the performance hypotheses has been extended such that the decision procedure can be delayed. In this case, the sequence of implementations whose performance data has to be compared is being 'queued' until a certain number of comparisons have to be executed. Thus, a single allreduce operations can be used to execute multiple comparisons and the same measurements can be used potentially to determine the optimal values for multiple attributes. In the current implementation of ADCL, the decision procedure for attribute $j$ is delayed until $reqconf(j)$ set of measurements are available, since this earliest point at which ADCL can perform any actions on behalf of attribute $j$.

## IV. Performance Evaluation

In the following, we will analyze the effect of using different implementations for the neighborhood communication on the performance of a parallel, iterative equation solver. The software used in this analysis solves a partial differential equation (PDE), respectively the set of linear equations obtained by discretization of the PDE using center differences. To partition the data among the processors, the parallel implementation subdivides the computational domain into rectangular subdomains of equal size. Thus, each processors holds the data of the corresponding subdomain. The processes are mapped onto a regular three-dimensional mesh. Due to the local structure of the discretization scheme, a processor has to communicate with at most six processors to perform a matrix-vector product. For the subsequent analysis the code has been modified such that it makes use of the ADCL library, i.e. the sections of the source code which established the 3-D process topology and the neighborhood communication routines have been exchanged by the according ADCL counterparts.

Since most MPI libraries do not show performance advantages for MPI put/get operations compared to two-sided communication on a typical PC cluster and in order to simplify our analysis, we have configured ADCL for the following tests without the one-sided data transfer primitives. This leaves twelve implementations for the 3-D neighborhood communication for the runtime selection logic to choose from. The number of tests required to evaluate an implementation has been set to 30.

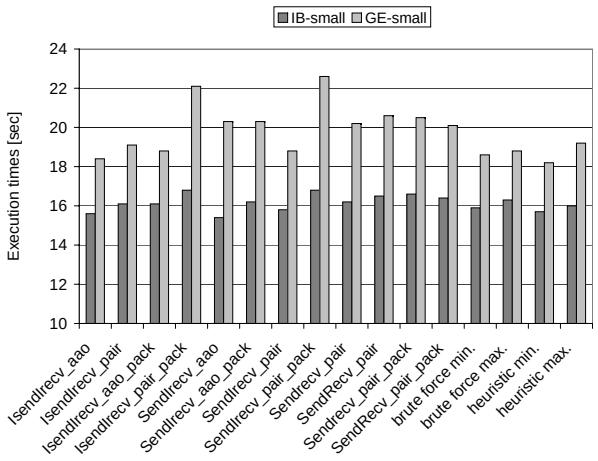


Fig. 2. Execution times for the small problem size using 16 processes for 500 iterations.

The cluster used for the following measurements consists of 24 nodes, each having a single dual-core 2.2 GHz Opteron processor. The nodes are connected by an 4x InfiniBand interconnect and an Gigabit Ethernet switch. We show results for 16 and 32 process testcases using 16 nodes for both scenarios, and three different application problem sizes. The small problem size is configured such that each process holds $32 \times 32 \times 32$ mesh points, the medium problem size assigns $64 \times 32 \times 32$ mesh points to each process while the large configuration doubles the number of mesh points per process once again to $64 \times 64 \times 32$. The execution times presented in this section are the overall execution time for 500 iterations of the iterative solver. Timings are given in seconds. The communication library used throughout the tests was

Open MPI version 1.1.1 [10]. Using Open MPI specific runtime flags we could test separately the performance of the code over InfiniBand as well as over GEthernet.

First, we would like to evaluate whether ADCL makes the correct decisions regarding the implementation chosen when using the runtime decision logic. For this, we measured the execution times for each of the available implementations independently of the ADCL runtime selection logic for all three problem sizes and both network interconnects. Each test has been repeated between up to 9 times. In the figures 2, 3 and 4 we show the best execution time achieved for each implementation. When using the runtime selection logic of ADCL, we report the best and the worst execution times obtained for both runtime selection algorithms.

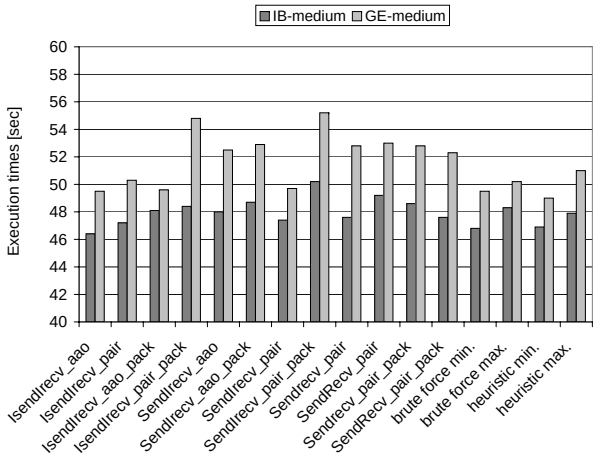


Fig. 3. Execution times for the medium problem size using 16 processes for 500 iterations.

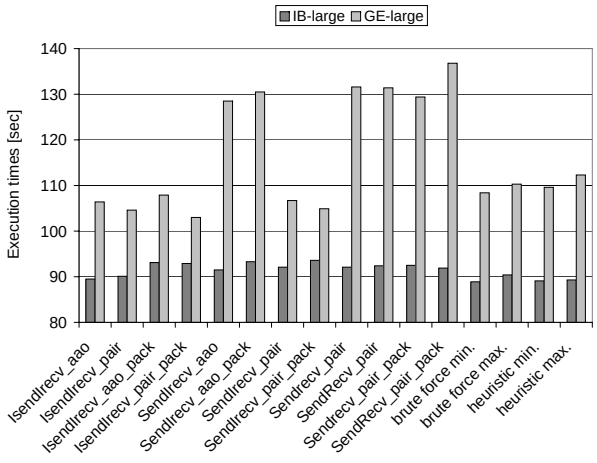


Fig. 4. Execution times for the large problem size using 16 processes for 500 iterations.

An initial observation revelead by fig. 2, 3 and 4 is, that the sensitivity of the execution time on the implementation of the 3-D neighborhood communication is increasing with the problem sizes. Furthermore, the application is more sensitive to the implementation of the neighborhood communication when using the Gigabit Ethernet interconnect.

Over InfiniBand the application showed for the small problem size in most cases the best performance when using `SendIrecv_aao` as the implementation for the neighborhood communication. When using the ADCL, both runtime selection algorithms determined in all of our tests the same implementation to be the fastest one. For the medium and large problem sizes, `IsendIrecv_aao` showed the best performance according to our measurements. While for the large problem size once again both runtime selection algorithms agreed on the best implementation to be `IsendIrecv_aao` as well, for the medium problem size the algorithm using the performance hypotheses concluded twice that a different implementation might be optimal: in one case, it chose `SendIrecv_aao` in the other case it decided to use `IsendIrecv_aao_pack`. Before analyzing the deviating results for the medium problem size, we would like to detail the sequence of events for the optimal scenarios which are represented in these measurements by the small and the large problem size.

As described in section II, each implementation is currently characterized by three attributes. The runtime heuristic using these attributes values evaluates in the first stage the performance of four of the available implementations: `IsendIrecv_aao`, `IsendIrecv_pair`, `IsendIrecv_aao_pack`, `IsendIrecv_pair_pack`. Using the delayed decision approach described in III-A the library can compare the performance data of the first two and the second two implementations in order to determine the best value for attribute one (`aao` vs. `pair`) and of the first and the third as well as the second and the fourth implementation for the second attribute (`ddt` vs. `pack`). Since the required confidence value for both attributes is 2, the library can remove under optimal circumstances after this first stage many of the available implementations. For the 16 process testcases presented above, the measurements confirm that an implementation initiating all messages at once is performing better than implementations communicating only to a single process at a time, and derived datatypes perform better in this scenario than pack/unpack. Thus, the heuristic reduces after the first stage the number of available implementations from 12 to 2. In the final step, the performance of `IsendIrecv_aao` and `SendIrecv_aao` are being compared in order to determine the best value for the the last parameter, namely the preferred transfer primitive.

An obvious question when comparing the performance of the brute search algorithm to the performance hypotheses based approach is, why the latter one does not show a significantly lower execution time compared to the brute force search for the small and the large problem size over InfiniBand, taking into account that the number of implementations tested are significantly lower in the second approach. The answer to this question is twofold: first, the overhead of testing a 'non-optimal' implementation is not dramatic over InfiniBand; second, despite of the optimization presented in section III-A the heuristic still uses at least one additional collective operation during the decision procedure, eliminating some of the performance gained by not testing a sub-optimal implementation. We expect however, that with increasing number of attributes, attribute values and implementations the advantages of the approach using performance hypotheses will become more dominant.

For the testcases using the medium problem size in which the runtime selection algorithm using the heuristic did not select `IsendIrecv_aao` as the fastest implementation, the heuristic did in fact not achieve the required confidence values for the first two attributes after the first stage. Thus, the runtime selection logic compared the performance achieved by all implementations and concluded that another implementation performed better at this point in time. The data dumped out by the library during the decision procedure does indicate, that the application experienced some perturbations. Figure 5 shows the execution times of each individual instance of a neighborhood communication measured when using the runtime selection logic based on the performance hypotheses. The first 360 iterations shown are used for determining the best implementation, while the following iterations are using the 'fastest' implementation determined by the library. Although the data shows a significant number of outliers, the majority of the measurements are located close to the minimal execution time measured by any implementation. Thus, while the outcome in these cases are not what we expected, the data indicates that the decision based on the behavior of the machine at this point in time is justifiable.

The results using the Gigabit Ethernet interface show for the 16 processes testcases a similar behavior as over InfiniBand: in the majority of the tests both runtime selection algorithms select the implementation which our external measurements determined to be the fastest one for that scenario. Even for the test cases where the runtime selection logic comes to a different conclusion than what we expected based on our external measurement, the library manages to avoid the implementations which result in a significantly lower performance. This is especially evident for the large problem size as shown in fig. 4.

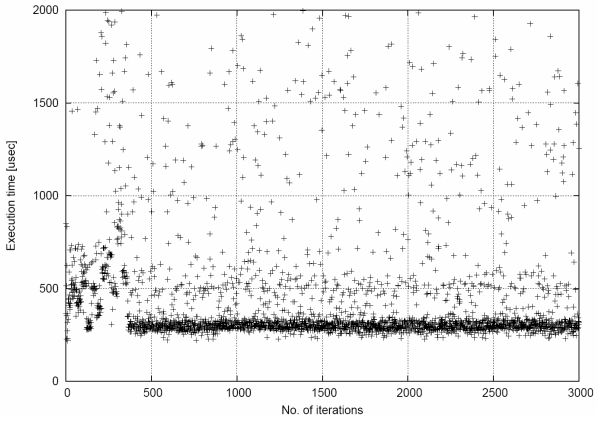


Fig. 5. Execution times of the neighborhood communication when using the performance hypotheses approach for the medium problem size of InfiniBand. The data shown was gathered in process 0.

### A. *Influence of the data filtering parameters*

Since the 32 processes testcases do not reveal any significant new revelations over InfiniBand, we would like to focus in this subsection on the results achieved by the large testcase using 32 processes over Gigabit Ethernet. As shown in fig. 6, the network/switch behaves in this case differently than in the 16 processes testcase: the fastest implementations determined outside of the runtime selection logic suddenly have the parameter values `pair` and `pack` compared to `aao` and `ddt` for the previous results. Since the switch seems to get congested by the communication produced in this scenario, all measurements show a significantly larger variation then in the previous scenarios. Thus, we show for all implementations the best, the worst and the average execution time.

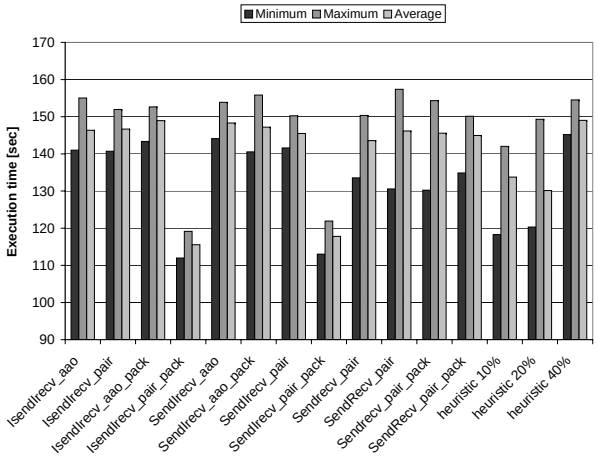


Fig. 6. Execution times for the large problem size on 32 processes using the Gigabit Ethernet interconnect

Due to the large variance in the performance achieved by any implementation, both runtime selection algorithms fail in the initial settings to determine the implementation delivering the best performance. A closer analysis of the data revealed, that the problem is rooted in the data filtering approach of ADCL. The default setting within the library accepts up to 40% of outliers. As a result of this setting, many implementations could remove too many potential outliers and thus an implementation, which in fact showed a very unreliable performance behavior was chosen by the algorithms. Figure 6 shows the performance of the runtime selection logic using the heuristic for different settings of the parameter within ADCL, which defines the number of accepted outliers. While 10% outliers seem to be too strict for this environment, accepting 40% of outliers definitely leads to choosing the wrong implementations. The best results were achieved when the threshold for the number of accepted outliers has been set to 20%. In this case, both runtime selection algorithms identified in the majority of the experiments the 'correct' implementation.

## V. Summary

This paper presents the Adaptive Data and Communication Library (ADCL). The key feature of ADCL is its capability to choose at runtime the best performing implementation for a particular collective operation. We introduce in this paper furthermore attributes characterizing the actual implementation of an application level collective operation and detail a runtime decision heuristic relying on these attributes. The library has been evaluated using a finite-difference code using InfiniBand and Gigabit Ethernet network interconnects. The results indicate, that in an environment producing repeatable performance data, the runtime decision logic based on the performance hypotheses is determining the 'fastest' implementation reliable. The library is also capable to handle moderate perturbations, since it incorporates a (simplistic) data filtering approach. In case of a less reliable environment, we showed the dependence of the library on the parameters of the data filtering.

The currently ongoing work within ADCL focuses on three aspects: first, the number of available implementations are being extended by introducing new attribute values. This includes additional implementation such as topology-aware implementations of the neighborhood communication as presented in [11] as well as using additional transfer primitives such as persistent request operations. Second, initial measurements in less reliable environments than the ones used throughout this paper indicate, that the filtering of the measurements has a high impact on the decision made in the following steps. Thus, we are working on extending the filtering routines within ADCL to use cluster analysis techniques. Third, we are also working on extending our attribute model characterizing implementation to include sub-classes of attributes. As an example, the data transfer primitives consists today of two-sided and one-sided operation. According to our experience on current platforms, one-sided operations are either significantly faster or significantly slower than regular send/recv operations. Thus, by testing a single implementation which uses one-sided operations, we could conclude on the performance of a whole class of implementations.

## References


[1] R. Thakur and W. Gropp, "Improving the performance of collective operations in MPICH," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. LNCS, J. Dongarra, D. Laforenza, and S. Orlando, Eds., no. 2840. Springer Verlag, 2003, pp. 257–267, 10th European PVM/MPI User's Group Meeting, Venice, Italy.

[2] 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.

[3] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPI's collective communication operations for clustered wide area systems," *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, vol. 34, no. 8, pp. 131–140, May 1999.

[4] R. Rabenseifner and J. L. Träff, "More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems," in *Proceedings of EuroPVM/MPI*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2004, pp. 36–46.

[5] 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*. New York, NY, USA: ACM Press, 2006, pp. 199–208.

[6] B. Palmer and J. Nieplocha, "Efficient Algorithms for Ghost Cells Updates on Two Classes of MPP Architectures," in *Proceedings of the 14th IASTED international conference on Parallel and Distributed Computing and Systems (PDCS)*, Cambridge, USA, 2002.

[7] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[8] R. C. Whaley and A. Petite, "Minimizing development and maintenance costs in supporting persistently optimized blas," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, 2005.

[9] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of IEEE*, vol. 93, no. 2, pp. 216–231, 2005.

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

[11] R. Keller, E. Gabriel, B. Krammer, M. S. Mller, and M. M. Resch, "Efficient execution of MPI applications on the grid: porting and optimization issues'," *Journal of Grid Computing*, vol. 1, no. 2, pp. 133–149, 2003.