Towards Performance and Portability through Runtime Adaption for High Performance Computing Applications

Edgar Gabriel [†], Saber Feki [†], Katharina Benkert [‡], Michael M. Resch [‡]

[†] Parallel Software Technologies Laboratory,
Department of Computer Science,
University of Houston, Houston, TX, USA
{gabriel, sfeki}@cs.uh.edu

[‡]High Performance Computing Center Stuttgart (HRLS),
University of Stuttgart, Stuttgart, Germany
{benkert, resch }@hlrs.de

March 18, 2008

Abstract

The Abstract Data and Communication Library (ADCL) is an adaptive communication library optimizing application level collective communication operations at runtime. The library provides for a given communication pattern a large number of implementations and incorporates a runtime selection logic in order to choose the implementation leading to the highest performance. In this paper, we demonstrate for the first time, how an application utilizing ADCL is deployed on a wide range of HPC architectures, including an IBM Blue Gene, an NEC SX8, an IBM Power PC cluster using an IBM Federation Switch, an AMD Opteron cluster utilizing an 4xInfiniBand and a Gigabit Ethernet network, and an Intel EM64T cluster using a hierarchical Gigabit Ethernet network with reduced uplink bandwidth. We demonstrate, how different implementations for the three dimensional neighborhood communication lead to the minimal execution time of the application on different architectures. ADCL gives the user the advantage of having to maintain only a single version of the source code and still have the ability to achieve close to optimal performance for the application on all architectures.

1 Introduction and Motivation

Petaflop computer systems are expected to be available within the next year [3]. However, the number of applications capable of efficiently using thousands of processors or achieving a sustained performance of multiple teraflops is limited and 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 a Massively Parallel Processing System such as the IBM Blue Gene [10] will very probably perform poorly on a vector architecture such as the NEC SX8 [11], and vice versa. Among the problems application developers face are the wide variety of available hardware and software components, such as

- processor type and frequency, number of processor per node and number of cores per processor,
- size and performance of the main memory, cache hierarchies,
- network interconnect, network topology, network device drivers,
- operating system, compilers and communication libraries,

and the influence of each of these components on the performance of their applications. Hence, an end-user faces a (nearly) unique execution environment on each parallel machine. Developing code which is portable and has the ability to perform as close to the theoretical peak performance of a given architecture as possible is one of the big challenges of HPC.

In order to exploit the performance of HPC systems, end-users and application developers apply resource and time consuming tuning of individual software components as well as of the application. Certain software components can be tuned for a given platform before the execution of the application. This approach has been taken by several projects such as ATLAS [18] or ATCC [15]. However, several factors influencing the performance of the application can only be determined while executing the application itself. These factors include process placement by the batch scheduler [5], resource utilization due to the fact, that some resources such as the network switch are shared by multiple applications, and application characteristics (e.g., communication volume and frequencies).

This paper presents a software infrastructure allowing for the runtime optimization of communication operations. The Abstract Data and Communication Library (ADCL) [9] gives application developers the ability to express often occurring communication patterns in higher level terms, and optimizes the communication at runtime, adapting to the characteristics of the hardware and the application. In this paper, we demonstrate for the first time, how an application utilizing ADCL is deployed on a wide range of HPC resources, including an (I) IBM Blue Gene, an (II) NEC SX8, an (III) IBM Power PC cluster connected by an IBM Federation Switch, an AMD Opteron cluster utilizing 4x InfiniBand, an (V) AMD Opteron cluster utilizing Gigabit Ethernet, and an (VI) Intel EM64T cluster using a hierarchical Gigabit Ethernet network with reduced uplink bandwidth. We focus on the optimization of a single, however highly relevant communication operation, namely the n-dimensional neighborhood communication. This communication pattern is the dominant communication operation in many applications [14, 16]. We demonstrate, how different implementations for the three dimensional neighborhood communication lead to the minimal execution time of the application on different architectures. ADCL gives the user the advantage of having to maintain only a single version of the source code and still have the ability to achieve close to optimal performance for the application on all architectures. We will also document the penalty of an application, which would have only a single, fixed implementation for that operation.

Some projects incorporate runtime adaptation for particular operations for HPC applications. The best known example is FFTW [8]. The FFTW library optimizes Fast Fourier Transform (FFT) operations. 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. Thus, FFTW makes the runtime optimization upfront in the planner step, which does not perform any useful work. On the other hand, ADCL integrated the runtime selection logic into the regular execution of the applications. This is especially important, since the ADCL approach enables the library to restart the runtime selection logic in case significant deviations from the original performance, e.g. due to changing network conditions have been observed.

Another approach is presented by the Star-MPI [6] project. This library incorporates runtime optimization of collective operations similarly to ADCL. There are however two main differences between Star-MPI and ADCL: first, Star-MPI focuses only on collective operations as defined in the MPI standard, while ADCL provides a broad framework which can be (and has already been applied) for a wider range of problems. Second, unlike ADCL, Star-MPI has only a single runtime decision logic, namely a brute force search, whereas one of the main research focuses of ADCL is to develop alternative runtime decision algorithms in order to speed up the runtime decision logic, e.g. as required for adaptive or irregular applications.

The remainder of the paper is organized as follows: in section 2 we present the concept of ADCL. Section 3 presents the application scenario and the results achieved using ADCL for this application on all platforms mentioned above. Finally, section 4 summarizes the paper and presents the ongoing work.

2 Technical Concept

ADCL is an application level communication library aiming at providing the highest possible performance for application level communication patterns, such as the n-D neighborhood communication, within 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.

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

- `ADCL_Topology`: provides a description of the process topology and neighborhood relations within the application.
- `ADCL_Vector`: specifies the data structures to be used during the communication. 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.
- `ADCL_Function`: each ADCL function is the equivalent to an actual implementation of a particular communication pattern.
- `ADCL_Fnctset`: a collection of ADCL functions providing the same functionality. ADCL provides pre-defined function sets, such as for neighborhood communication (`ADCL_FNCTSET_NEIGHBORHOOD`) or circular shift operations (`ADCL_FNCTSET_SHIFT`). The user can however also register its own functions in order to utilize the ADCL runtime selection logic.
- `ADCL_Attribute`: abstraction for a particular characteristic of a function/implementation. Each attribute is represented by the set of possible values for this characteristic.
- `ADCL_Attrset`: a collection of ADCL attributes
- `ADCL_Request`: combines a process topology, a function-set and a vector object. The application can initiate a communication by starting a particular ADCL request.

Figure 1 gives a simple example for an ADCL code, using a 2-D neighborhood communication on a 2-D process topology.

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. The approach chosen for ADCL is to use the first iterations of the application to determine the fastest available implementation. Although some of the tested implementations may not deliver the optimal performance, this approach avoids a separate 'planner' step which does not contribute at all to the solution of the problem and ignores other influences on the communication process such as work imbalances. A planner also has the disadvantage that once a decision is made, it is kept for the whole execution time, even if the networking conditions change significantly compared to the initial evaluation. With ADCL on the other hand, one could easily use a monitoring interface, which restarts the runtime decision logic whenever necessary. During the first iterations, ADCL will call each/some implementation(s) multiple times in order to determine the fastest implementation from a given set of functions. Each process keeps track of the execution times of all implementations in a data array which is attached to the according `ADCL_Request`. After all implementations have been tested the required number of times, the measurements are analyzed mainly locally with a statistical method [1]. Then, depending on the evaluation method used, the method judged fastest is chosen.

ADCL incorporates as of today two separate 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 the evaluation and learning phase can take a significant amount of time. In order to speed up the selection logic, an alternative runtime heuristic based on the attributes characterizing an implementation has been developed [9]. 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. Therefore, the algorithm tries to determine 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 according attribute and thus shrinks the list of

```
double vector[...][...];
ADCL_Vector vec;
ADCL_Topology topo;
ADCL_Request request;

/* Generate a 2-D process topology */
MPI_Cart_create (MPI_COMM_WORLD, 2, cart_dims, periods, 0, &cart_comm);
ADCL_Topology_create (cart_comm, &topo );

/* Register a 2D vector with ADCL */
ADCL_Vector_register (ndims, vec_dims, NUM_HALO_CELLS, MPI_DOUBLE, vector, &vec);

/* Combine description of data structure and process topology */
ADCL_Request_create (vec, topo, ADCL_FNCTSET_NEIGHBORHOOD, &request );

/* Main application loop */
for (i=0; i<NIT; i++ ) {
   ...
   /* Initiate neighborhood communication */
   ADCL_Request_start (request );
   ...
}
```

Figure 1: Code sample for a parallel application using a regular 2-D neighborhood communication based on ADCL and MPI.

available implementations. For the n-dimensional neighborhood communication, the library currently characterizes each implementation by three attributes: (I) the number of simultaneous communication partners for each process (e.g., *pair:* pair-wise, *aao:* all-to-all), the data transfer primitives used (e.g., synchronous send, asynchronous send, one-sided communication) and the handling of non-contiguous data (e.g., *ddt:* derived data types, *pack:* pack/unpack).

Table 1 lists all implementations available in ADCL for the neighborhood communication, and the according attribute values. 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 `MPI_Send/Recv` 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. For the sake of clarity, we will skip the implementations using one-sided, and work with the twelve remaining implementations.

3 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 solver as often applied in scientific application. The software used in this section solves a set of linear equations that stem from discretization of a partial differential equation PDE using center differences. The parallel implementation subdivides the computational domain into subdomains of equal size. The processes are mapped onto a regular three-dimensional cartesian topology. Due to 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.

In order to evaluate the 'correct' decision of the runtime selection logic, we additionally executed the same application using one single implementation at a time by 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

Name of the implementation	No. of simult. comm. partners	Handling of noncont. msgs.	Data transfer primitive
IsendIrecv_aao	aao	ddt	MPI_Isend/Irecv/Waitall
IsendIrecv_pair	pair	ddt	MPI_Isend/Irecv/Waitall
SendIrecv_aao	aao	ddt	MPI_Send/Irecv/Waitall
SendIrecv_pair	pair	ddt	MPI_Send/Irecv/Wait
IsendIrecv_aao_pack	aao	pack	MPI_Isend/Irecv/Waitall
IsendIrecv_pair_pack	pair	pack	MPI_Isend/Irecv/Waitall
SendIrecv_aao_pack	aao	pack	MPI_Send/Irecv/Waitall
SendIrecv_pair_pack	pair	pack	MPI_Send/Irecv/Wait
SendRecv_pair	pair	ddt	MPI_Send/Recv
Sendrecv_pair	pair	ddt	MPI_Sendrecv
SendRecv_pair_pack	pair	pack	MPI_Send/Recv
Sendrecv_pair_pack	pair	pack	MPI_Sendrecv
WinfecePut_aao	aao	ddt	MPI_Put/MPI_Win_fence
WinfeceGet_aao	aao	ddt	MPI_Get/MPI_Win_fence
PostStartPut_aao	aao	ddt	MPI_Put/MPI_Win_post/start
PostStartGet_aao	aao	ddt	MPI_Get/MPI_Win_post/start
WinfecePut_pair	pair	ddt	MPI_Put/MPI_Win_fence
WinfeceGet_pair	pair	ddt	MPI_Get/MPI_Win_fence
PostStartPut_pair	pair	ddt	MPI_Put/MPI_Win_post/start
PostStartGet_pair	pair	ddt	MPI_Get/MPI_Win_post/start

Table 1: Currently available implementations of regular n-dimensional communication within ADCL and the according attributes

batch scheduler allocation and thus had the same node assignments. We will refer to these measurements as verification runs throughout the rest of this section. During each verification run, the execution times of 700 iterations per implementation were stored and subsequently averaged over all three runs. Depending on the machine, we have executed three different problem sizes, namely a small test case with $32 \times 32 \times 32$ mesh points per process, a medium test case with $64 \times 32 \times 32$ mesh points per process, and a large test case with $64 \times 64 \times 32$ mesh point per process.

Tests have been executed on six platforms all in all:

1. **IBM Blue Gene**: The Blue Gene system at the San Diego Supercomputing Center consists of 3,072 compute nodes with 6,144 processors. Each node consists of two PowerPC processors that run at 700 MHz and share 512 MB of memory. All compute nodes are connected by two high-speed networks: a 3-D torus for point-to-point message passing and a global tree for collective message passing. We have run tests using 128, 256 and 512 processes.

2. **NEC SX8**: the installation at the HLRS consists of 72 nodes, each node having 8 vector processors of 16 Gflops peak (2Ghz) and 128 GB of main memory. The nodes are interconnected by an IXS switch. Each node can send and receive with 16 GB/s in each direction. We have executed tests with 16, 32 and 64 processes.

3. **DataStar p655**: the p655 partition of DataStar cluster at San Diego Supercomputing Center has 272 8-way compute nodes – 176 nodes with 1.5-GHz Power4+ CPUs and 16 GB of memory, and 96 with 1.7 GHz Power4+ CPUs and 32 GB of memory. The nodes are connected by an IBM high-speed Federation switch. The tests executed and presented in this paper include 64, 128, 256 and 512 processes runs.

4. **SharkIB**: this cluster consists of 24 nodes, each node having a dual core AMD Opteron processor and 2 GB of main memory. The nodes are interconnected by a 4xInfiniBand network. We present results for 32 and 48 processes on 16 respectively 24 nodes.

Architecture	No. of proc.	Problem size/Proc	Best implementation
DataStar p655	64	64x32x32	SendIrecv_aao
		64x64x32	SendIrecv_aao
	128	64x32x32	SendIrecv_pair_pack
		64x64x32	IsendIrecv_aao
	256	64x32x32	IsendIrecv_aao
		64x64x32	IsendIrecv_aao
	512	64x32x32	SendIrecv_pair_pack
		64x64x32	IsendIrecv_aao
IBM Blue Gene	128	64x32x32	IsendIrecv_aao_pack
		64x64x32	SendIrecv_pair_pack
	256	64x32x32	SendIrecv_pair_pack
		64x64x32	IsendIrecv_aao_pack
	512	64x32x32	SendIrecv_pair_pack
		64x64x32	IsendIrecv_aao_pack
NEC SX8	16	64x32x32	Sendrecv_pair
		64x64x32	Sendrecv_pair
	32	64x32x32	SendIrecv_pair_pack
		64x64x32	SendIrecv_pair_pack
	64	64x32x32	Sendrecv_pair_pack
		64x64x32	SendIrecv_pair_pack
SharkIB	32	32x32x32	IsendIrecv_aao
		64x64x32	SendIrecv_pair_pack
	48	32x32x32	SendIrecv_aao
		64x64x32	IsendIrecv_aao
SharkGE	32	32x32x32	IsendIrecv_aao
		64x64x32	IsendIrecv_aao_pack
	48	32x32x32	IsendIrecv_aao
		64x64x32	Sendrecv_pair
CacauGE	64	64x32x32	SendRecv_pair_pack

Table 2: Fastest implementation for each platform and problem size as determined by the verification runs.

5. **SharkGE**: This is same cluster as described in the previous bullet, using however a Gigabit Ethernet network interconnect between the nodes.

6. **CacauGE**: Cacau is a 200 node, dual processor Intel EM64T cluster at the HLRS. Although the main network of the cluster is a 4xInfiniBand interconnect, we used for the subsequent analysis the secondary network of the cluster, namely a hierarchical Gigabit Ethernet network. This network consists of six 48-port switches which are used to connect the nodes, each 48-port switch has four links to the upper level 24 port Gigabit Ethernet switch. Thus, this network has a 12:1 blocking factor. We have executed tests using 64 processors on 64 nodes in order to ensure that communication between the processes has to use two or more of the 48-port switches.

Table 2 summarizes for each platform and problem size the implementation of the neighborhood communication which lead to the overall best performance. The results have been determined by using the average of three independent runs. In the 29 different test cases presented here, seven out of the twelve implementations available in ADCL for that communication pattern turn out to lead to the minimal execution time of the application. Most notably, there are changes for the best performing implementation depending on both, the number of processors and the problem size per process for basically all platforms tested.

In the following, we would like to quantify the penalty an application would pay by not using the optimal implementation for a given problem size. Since the focus of this paper is on cross-platform analysis, we detail two scenarios. First, for each platform we choose the largest problem on the largest number of processors that we ran, and evaluate what the performance penalty would be to use an implementation, which has been determined to be the winner on any of the other platforms. In table 3, each row represents the performance penalty of the application running on that platform when using the implementation determined to be the winner on the platform showed in the column. As an example, the element in the first row, third column shows the performance penalty for the application running the large problem size on the Datastar cluster using the 'winner' function determined by the SX8 using 64 processes.

	DS512l	BG512l	SX64l	SharkIB48l	SharkGE48l	CGE64m
DS512l	0.00%	2.42%	3.32%	0.00%	1.74%	7.03%
BG512l	1.84%	0.00%	0.31%	1.84%	2.57%	12.25%
SX64l	1.56%	0.79%	0.00%	1.56%	9.86%	58.99%
SharkIB48l	0.00%	1.33%	0.47%	0.00%	0.34%	4.37%
SharkGE48l	66.90%	79.60%	59.54%	66.90%	0.00%	2.54%
CGE64m	90.02%	88.51%	14.22%	90.02%	33.95%	0.00%

Table 3: Cross architecture performance comparison showing the overhead in percentage of using the fastest implementation determined by a different architecture.

The most remarkable result of table 3 is, that the winner functions of the SharkGE and CacauGE cause significant performance penalties on the high performance interconnects used on Datastar, IBM Blue Gene and NEC SX 8. The performance penalty ranges from 1.74% up to 58.99%. Vice versa, the implementation chosen by these machines lead to singificant performance penalties on SharkGE and CacauGE, increasing the execution time up to 90% compared to their fastest implementation.

In the second scenario, we focus on only three architectures, namely Datastar, IBM Blue Gene and the NEC SX8. We analyze the execution times for the medium problem size for all available number of processes. The results are presented using same format as in table 3. Although the performance penalty for many scenarios shown in table 4 are negligible, there are notable exceptions. As an example, applying the winner function of the 64 processes case on Datastar onto the 128 processes case on Blue Gene leads to a 5% increase in the execution time of that simulation. Similarly, the best performing implementation in the 256 processes test case on the Blue Gene would lead to a 11.49% increase in the execution time for the 256 processes test case on the Datastar architecture. Last but not least, the implementation leading to the best performance on the 64 processes test case on the SX8 would lead to a performance penalty of more than 15% on the same architecture for the same problem size per processes but for the 32 processes test case.

	DS64m	DS128m	DS256m	DS512m	BG128m	BG256m	BG512m	SX16m	SX32m	SX64m
DS64m	0.00%	3.20%	0.25%	3.20%	3.20%	3.20%	3.20%	1.52%	3.20%	3.61%
DS128m	0.21%	0.00%	0.09%	0.00%	0.00%	0.00%	0.00%	1.53%	0.00%	3.41%
DS256m	0.19%	11.49%	0.00%	11.49%	11.49%	11.49%	11.49%	2.36%	11.49%	3.72%
DS512m	0.88%	0.00%	0.14%	0.00%	0.00%	0.00%	0.00%	3.02%	0.00%	5.13%
BG128m	5.34%	0.00%	0.14%	0.00%	0.00%	0.00%	0.00%	4.69%	0.00%	4.93%
BG256m	4.17%	0.00%	0.12%	0.00%	0.00%	0.00%	0.00%	3.91%	0.00%	4.10%
BG512m	4.40%	0.00%	1.71%	0.00%	0.00%	0.00%	0.00%	3.27%	0.00%	2.74%
SX16m	0.09%	1.02%	0.46%	1.02%	1.02%	1.02%	1.02%	0.00%	1.02%	3.43%
SX32m	1.40%	0.00%	1.32%	0.00%	0.00%	0.00%	0.00%	2.28%	0.00%	15.79%
SX64m	1.30%	0.98%	2.79%	0.98%	0.98%	0.98%	0.98%	7.43%	0.98%	0.00%

Table 4: Cross Architecture performance comparison showing the overhead in percentage of using the fastest implementation determined by a different architecture for a constant problem size.

As of now, we have documented the fact, that the performance of an application does depend on the implementation of the neighborhood communication, the hardware architecture, the application problem size, and the number of processes. In the following, we would like to show that using the ADCL runtime selection logic leads in most cases to a close-to-optimal performance. Table 5 documents the average overhead of the application when using ADCL compared to the performance of the application using the fastest implementation for that particular scenario. Table 5 distinguishes between the two runtime selection logics of ADCL, namely the brute force search strategy, and the parameter based optimizations.

Architecture	No. of proc.	Problem size/Proc	ADCL Brute force search	ADCL attribute based search
DataStar p655	64	64x32x32	0.38%	-0.16%
		64x64x32	0.25%	-0.06%
	128	64x32x32	0.68%	0.28%
		64x64x32	0.90%	0.54%
	256	64x32x32	1.24%	1.21%
		64x64x32	0.62%	0.31%
	512	64x32x32	2.68%	0.91%
		64x64x32	1.32%	0.95%
IBM Blue Gene	128	64x32x32	0.63%	0.38%
		64x64x32	0.49%	0.17%
	256	64x32x32	0.62%	0.23%
		64x64x32	0.39%	0.11%
	512	64x32x32	0.46%	0.15%
		64x64x32	0.36%	0.21%
NEC SX8	16	64x32x32	1.67%	2.13%
		64x64x32	1.35%	0.68%
	32	64x32x32	1.98%	1.54%
		64x64x32	0.54%	0.65%
	64	64x32x32	2.60%	0.59%
		64x64x32	1.50%	2.40%
SharkIB	32	32x32x32	2.54%	0.57%
		64x64x32	0.38%	0.02%
	48	32x32x32	1.68%	1.99%
		64x64x32	0.65%	0.56%
SharkGE	32	32x32x32	3.03%	1.03%
		64x64x32	0.15%	0.82%
	48	32x32x32	6.70%	4.20%
		64x64x32	4.07%	26.37%
CacauGE	64	64x32x32	72.70%	-

Table 5: Overhead of the application using both ADCL runtime selection algorithms relative to the minimal execution time determined in the verification runs.

The main result of table 5 is that the execution time of the application when using ADCL with its runtime adaption features is in fact very close the optimal performance determined in the verification runs. The overhead introduced by ADCL is in the vast majority of the test cases below 1%. This (minimal) overhead stems from two facts: first, during the initial iterations of the application, ADCL evaluates some of the implementations, which show a suboptimal performance on that platform for that particular problem size. Second, ADCL incorporates a distributed decision algorithm in oder for all processes to agree on the same implementation as the 'winner'. This distributed decision algorithm requires one allreduce operation per implementation. Furthermore, the attribute based search strategy shows in virtually all test scenarios a lower overhead due to the reduced number of implementations being tested.

There are two notable exception to the results above: firstly, for the hierarchical Gigabit Ethernet network used on Cacau [1], and secondly, for the 48 processes test case on SharkGE when using the attributes based search strategy and the large problem size. In the first problem scenario, despite of the fact that the ADCL runtime selection logic did determine in all three runs the correct implementation, the performance penalty for using a suboptimal implementation during the learning phase turned out to be so tremendous, that the overall execution time of the application has increased by 72%. This result only highglits the necessity for additional, improved runtime selection algorithms which can further reduce the time required to determine the fastest algorithm.

[1]Data for the attribute based search strategy missing as of now. This data will be added for the final version of the paper.

For the second problem scenario, a more detailed analysis shows, that in two out of the three runs which have been used to calculate the average overhead in table 5, the parameter based search strategy did reveal a very good performance, showing only a minor overhead compared to the optimal execution time. However, in the third run, the system seemed to face some perturbations which lead to a wrong decision by the ADCL runtime selection logic. Using a suboptimal implementation for that test case resulted in a significant overhead. ADCL as of today relies on the fact, that the data gathered during the training phase is representative for the overall execution. In case this assumption turns out to be wrong, as in happened in the third run, the runtime selection logic will make a suboptimal decision. In order to handle this scenario, ADCL will be extended by a monitoring subsystem in the near future. In case the performance data of the 'winner' implementation deviates significantly from the performance data gathered during the learning phase, ADCL will be able to re-start the runtime selection logic and thus correct an erroneous decision. However, this component is not yet available as of today.

4 Summary and Outlook

In this paper, the performance of an application using the Abstract Data and Communication Library (ADCL) for the 3-dimensional neighborhood communication on different architectures has been evaluated. ADCL incorporates a large number of implementations for this operations, and provides a runtime selection logic in order determine the fastest implementation for a given execution environment and application during the execution. The results presented in this paper show that different platforms and problem sizes lead in fact to very different implementations for the neighborhood communication being optimal. Furthermore, the paper documents the performance penalty when using a suboptimal implementation. ADCL delivers in the vast majority of the analyzed scenarios a performance which is close to the optimal performance determined by manual verification runs. However, ADCL offers two main advantages to the end user compared to manual tuning: first, the user does not have to maintain different version of his source code in order to have a performant implementation for different platforms. Second, he does not have to execute time-consuming pre-execution tuning of the application for a particular platform.

The currently ongoing work in ADCL includes multiple areas. ADCL is currently being extended to incorporate historic learning capabilities in order to improve the starting point of the search procedure for a given platform. In a long term, we plan to include further runtime selection algorithms such as including early stopping criteria [17] or 2^k factorial design algorithms [13] for very large parameter spaces.

Additionally to the application used in this paper, ADCL has also been used successfully to optimize parallel matrix-matrix operations [12] and is the basis for an Open MPI specific parameter optimization tool [7]. Furthermore, two real-world applications at the HLRS in Stuttgart are currently being modified to use ADCL, namely a Supernova simulation [2] and a particle simulation using Lattice Boltzman Algorithms [4].

Acknowledgements

This research was funded in part by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems and in part by the National Science Foundation through TeraGrid resources at San Diego Supercomputing Center. We would also like to thank the NEC Corporation of America for the resources and support in porting ADCL to the SX architecture.

References

[1] Katharina Benkert, Edgar Gabriel, and Michael M. Resch. Outlier Detection in Performance Data of Parallel Applications. In *accepted for publications in the PDESC 2008 workshop*, page t.b.d., 2008.

[2] R. Buras, M. Rampp, H.T. Janka, and K. Kifonidis. Two-dimensional hydrodynamic core-collapse supernova simulations with spectral neutrino transport. Numerical method and results for a 15 Mo. star. *Astron. Astrophysics*, 447:1049–1092, 2006.

[3] Ohio Supercomputing Center. Ohio Supercomputer Center Hosts Jack Dongarra, Renowned High Performance Computing Expert. http://www.osc.edu/press/releases/2007/dongarra.shtml, 2007.

[4] Shiyi Chen and Gary D. Doolen. Lattice Boltzmann method for fluid flows. *ARFM*, 30:329–364, 1998.

[5] J. J. Evans, C. S. Hood, and W. D. Gropp. Exploring the Relationship Between Parallel Application Run-Time Variability and Network Performance. In *Proceedings of the Workshop on High-Speed Local Networks (HSLN), IEEE Conference on Local Computer Networks (LCN)*, pages 538– 547, October 2003.

[6] Ahmad Faraj, Xin Yuan, and David 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.

[7] Open Tool for Parameter Optimization. http://www.open-mpi.org/otpo, 2007.

[8] Matteo Frigo and Steven G. Johnson. The Design and Implementation of FFTW3. *Proceedings of IEEE*, 93(2):216–231, 2005.

[9] Edgar Gabriel and Shuo Huang. Runtime optimization of application level communication patterns. In *Proceedings of the 2007 International Parallel and Distributed Processing Symposium, 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, page 185, 2007.

[10] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2/3):195–212, 2005.

[11] NEC High Performance Computing Group. NEC SX series. http://www.nec.de/hpc/.

[12] Shuo Huang. Applying Adaptive Software Technologies for Scientific Applications. Master Thesis, Department of Computer Science, University of Houston, 2007.

[13] R. K. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[14] D. Kerbyson and K. Barker. Automatic identification of application communication patterns via templates. In *Proc. 18th International Conference on Parallel and Distributed Computing Systems (PDCS-2005)*, Las Vegas, NV, Sep 2005.

[15] Jelena Pjesivac-Grbovic, George Bosilca, Graham E. Fagg, Thara Angskun, and Jack J. Dongarra. MPI Collective Algorithm Selection and Quadtree Encoding. *Parallel Computing*, t.b.d:t.b.d, 2007.

[16] T. Tabe and Q. Stout. The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan, Nov 1999.

[17] Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. *International Journal for High Performance Computing Applications*, 18(1):65–94, 2004.

[18] R. C. Whaley and A. Petite. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, 2005.