Incorporating Historic Knowledge into a Self-Optimizing Communication Library for High Performance Computing

Saber Feki and Edgar Gabriel
Parallel Software Technologies Laboratory,
Department of Computer Science,
University of Houston, Houston, TX 77204-3010, USA.
Email: {sfeki, gabriel}@cs.uh.edu

Abstract— **Emerging computing systems have a wide variety of hardware and software components influencing the performance of parallel applications, presenting end-users with a (nearly) unique execution environment on each parallel machine. One of the big challenges of High Performance Computing is therefore to develop portable and efficient codes for any execution environment. The Abstract Data and Communication Library (ADCL) is a self-optimizing runtime communication library aiming at providing the highest possible performance for application level communication operations. The library provides for a given communication pattern a large number of implementations and incorporates a runtime selection logic. This selection aims at adaptively choosing the best performing implementation on the current platform and for the given problem. In this paper, we present a recent enhancement to the library which introduces the capability of utilizing information from previous executions in order to minimize the overhead of the runtime selection logic which mainly stems from testing underperforming implementations. We introduce the notion of similar problems by using a proximity measure for a given operation. The approach is evaluated for the n-dimensional neighborhood communication on different platforms and for a large range of different problems.**

Keywords: self-optimizing communication libraries, historic learning, proximity measures

I. Introduction

Due to the availability of multi-core processors and the omni-presence of gigabit quality networks, parallel programing is increasingly applied in the development of software products in areas such as gaming, or data and image processing. Companies turn towards parallel processing in order to exploit the capabilities of modern micro-processors and have the ability to solve – in the most general sense – larger problems in a shorter time frame. However, achieving good performance for a parallel application is highly challenging due to the wide range of hardware and software components available for of-the-shelf computer systems. As an example, the design of the new multi-core processors of Intel, AMD, SUN or IBM differ significantly in the organization and connectivity of the cores, the cache hierarchies and the I/O capabilities of the processors. It is similarly challenging to deal with the capabilities of the different network interconnects, which don't just include standard Fast and Gigabit Ethernet, but also more advanced technologies such as InfiniBand, iWarp, or 10Gigabit Ethernet. Furthermore, the software stack consisting of the operating system, device drivers, and communication libraries, has a significant influence on the performance achieved by the application. Thus, an application developer has to be aware of the fact, that his code will basically face a unique execution environment for every single machine that it is running on.

Scientific computing is already facing today most of the challenges outlined above. In order to exploit the capabilities of large scale High Performance Computing (HPC) systems, end-users and application developers apply resource and time consuming tuning of individual software components on each platform. 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 [1] or ATCC [2]. The main drawback of these approaches is, that the tuning procedure itself often takes the same or a similar amount of time as running the application itself. Additionally, several factors influencing the performance of the application can only be determined while executing the application. These factors include process placement by the batch scheduler [3], resource utilization due to the fact, that some resources such as the network switch are shared by multiple applications, operating system jitter [4]and application characteristics (e.g., communication volume and frequencies).

There are only very few projects as of today in HPC trying to incorporate runtime adaptation or exposing self-tuning behavior. For example, the Abstract Data and Communication Library (ADCL) [5], [6] allows applications to incorporate self-tuning operations, which are optimized by ADCL while executing the application itself. Thus, application developers avoid expensive pre-tuning steps. The library has been successfully used to tune various communication operations, such as n-dimensional neighborhood communication, parallel matrix-matrix operations and low-level parameters of a message passing library [7]. Although ADCL has shown to deliver close-to-optimal performance for a large number of platforms and network interconnects [8], there are still situations, where the current approach used by ADCL results in a significant overhead due to the tuning procedure. Most notably, applications exposing frequently and dynamically varying problem sizes, e.g. due to adaptive mesh-refinements, would require a very lightweight quick tuning procedure. Similarly, on certain platforms, exploring the performance of a suboptimal communication pattern can add a tremendous penalty to the overall execution time, resulting in a large discrepancy between a hand-tuned and an automatically optimized version of the code.

In this paper, we present an extension to the tuning algorithm and the decision logic of ADCL, which takes performance information from previous executions into account. There are two challenges which complicate this task: first, it is very seldom in real life, that exactly the same problem/problem size is being executed on the same platform twice. Thus, the library has to introduce the notion of and a quantification for *similar* problems. Second, even on a single parallel machine, conditions can vary between different executions significantly. The library therefore has to introduce a 'safety net' which results in dismissing historic performance data under certain conditions. The paper describes the algorithms used in ADCL to incorporate historic knowledge and evaluates the properties of the system for the most popular communication pattern in high performance computing, namely the n-dimensional neighborhood communication.

The remainder of the paper is organized as follows:Sec. II presents the main concepts and ideas behind ADCL. Sec. III presents the approach taken by ADCL to consider historic knowledge. In sec. IV we evaluate the approach for three different platforms, a large number of different problem sizes and three different versions of the neighborhood communication. In sec. V we discuss some related work, and finally, Sec. VI summarizes our findings and presents the currently ongoing work in that area.

II. The Abstract Data and Communication Library

The Abstract Data and Communication Library (ADCL) enables the creation of self-optimizing applications by allowing an application to register alternative versions of a particular function. From the conceptual perspective, ADCL takes advantage of two characteristics of most scientific applications:

1) *Iterative execution*: most parallel, scientific applications are centered around a large loop, and execute therefore the same code sequence over and over again. Consider for example an application which solves a time-dependent partial differential equation (PDE). These problems are often solved by discretizing the PDE in space and time, and by solving the resulting system of linear equations for each time step. Depending on the application, iteration counts can reach six digit numbers.
2) *Collective execution*: most large scale parallel applications are based on data decomposition, i.e. all processes execute the same code sequence on different data items. Processes are typically also synchronized, i.e. all processes are in the same loop iteration. This synchronization is often required for numerical reasons and is enforced by communication operations.

ADCL uses the initial iterations of the application to determine the fastest available code version. Once performance data on a sufficient number of versions is available, the library makes a decision on which alternative to use throughout the rest of the execution. Two key components of ADCL are the algorithm used to determine, which versions of a particular operation shall be tested, and how to decide efficiently across multiple process on the best performing version. In the following, we give some details on both components.

A. Version Management and Selection

A fundamental assumption within ADCL is, that the library has multiple alternative versions for a particular functionality available to choose from. These alternatives will be stored as different *functions* in the same *function-set*. The number of alternatives can reach from a few, (e.g. the user providing two different version of a parallel matrix-multiply operation) to many millions, in case of the user exploring different values for internal or external parameters, such as buffer sizes, loop unroll depth etc. In case of such large numbers of alternative versions, it is unrealistic to assume, that the library can test all available versions before deciding, which one performs best.

As of today, ADCL incorporates two different strategies for the version selection at runtime. The first version incorporates a simple brute-force search, which evaluates all available alternatives. Clearly, this approach has the limitations outlined above, and we are working on extending this approach by using some early stopping criteria as outlined in [9].

An alternative version selection algorithm is used, if the user annotates the implementations by a set of attributes/attribute values. These attributes are used to reduce the time taken by the runtime selection procedure, by tuning each attribute separately. Given a certain number of measurements conforming that a particular attribute value leads to better performance than versions using other values for the very same attribute, all implementations/versions of the according function-set not having the optimal value for that attribute are being discarded by the library. As has been shown in [5], this can significantly reduce the number of versions to be tested by ADCL, and improve the overall execution time. The main restriction of this approach lies in the assumption, that attributes are not correlated. However, there are a number of algorithms known in the literature to overcome this restriction, e.g. from the experimental design theory the 2^k factorial design [10], [11] algorithms.

B. Collective decision logic

Independent of the version selection approach used by the library, the collective decision logic of ADCL will have to compare performance data of multiple versions gathered on different processes. The challenge lies in the fact, that in the most general case, processes only have access their own performance data and performance data for the same code version might in fact differ significantly across multiple processes. Distributing the performance data of all processes for all versions to all other processes is however not feasible, since the costs for communicating these large volumes of data would often offset the performance benefits achieved by runtime tuning. The approach taken by the library relies therefore on data reduction, i.e. each process provides only a single value for each alternative version of the code section optimized.

In order to detail the algorithm lets assume, that ADCL gather n measurements/data points for each version i on each process j. Let us denote the execution time of the k-th measurement by $t(i,j,k)$. In an initial step, the library removes outliers, i.e. measurements not fulfilling the condition $\mathcal{C} = t(i,j,k) \,|\, t(i,j,k) < b \cdot \min_k t(i,j,k)$, with b being a well defined constant, from the data set. This leads to a filtered subset

$$M^f(i,j) = \{t(i,j,k) \,|\, t(i,j,k) \text{ fulfills } \mathcal{C}\} \qquad (1)$$

of measurements with cardinality $n_f(i,j)$. Then, the performance measurements for each version are analyzed locally on each process and characterized by the local average execution time

$$m(i,j) = \frac{1}{n} \sum_k t(i,j,k) \qquad (2)$$

and its filtered counterpart

$$m_f(i,j) = \frac{1}{n_f} \sum_{k \in M^f(i,j)} t(i,j,k) \qquad (3)$$

as estimates of the mean value. In a global reduction operation, the library determines for each version the maximum average execution time across all processes

$$m(i) = \max_j m(i,j), \qquad (4)$$

$$m_f(i) = \max_j m_f(i,j) \qquad (5)$$

considering all respectively only filtered data (3), and the maximum number of outliers $n_o(i)$ over all processes

$$n_o(i) = \max_j n_o(i,j).$$

This reduction is motivated by a fundamental law parallel computing, which states, that the performance of an (synchronous) application is determined by the slowest process/processor. Finally, the library selects the maximum execution time including or excluding outliers by

$$r(i) = \begin{cases} m_f(i) & \text{if } n_o(i) <= nmax_o \\ m(i) & \text{otherwise} \end{cases}$$

depending on whether the maximum number of outliers is exceeded or not. The algorithm i' fulfilling $r(i') = min_i\, r(i)$ is chosen as the best one. Assuming that the runtime environment produces reproducible performance data over the lifetime of an application, this algorithm is guaranteed to find the fastest of available implementation for the current tuple of {problem size, runtime environment, versions tested} **[?]**.

C. *An example function-set*

In the following, we would like to give a concrete example for a function-set in ADCL by describing the alternative versions for the most relevant communication pattern in parallel computing, namely the n-dimensional neighborhood communication. This communication patterns often occurs for applications relying on data decomposition, where each process owns a rectangular portion of the overall computational domain of equal size. Typically, processes are mapped onto a regular n-dimensional cartesian process topology. Due to the numerical operations performed on the data, a process often needs read access to data items owned by its 'neighboring' processes. A typical solution for this problem is to keep a copy of these data items in addition to the items owned by a process in so-called *ghost cells*. These ghost-cells have to updated frequently, e.g. in every iteration of an iterative solver. A process is not allowed to modify a ghost-cell. Fig. 1 shows an example for nine processes, which are arranged in a 2-D logical process topology, and the resulting messages between the processes when updating the ghost-cells. Ghost cells are depicted in green, while the main computational domain of each process is represented as white boxes. Due to the local structure of the discretization scheme, a processor has to communicate with at most two processes for a 1-D decomposition, four processes for a 2-D decomposition and six processes for a 3-D decomposition.

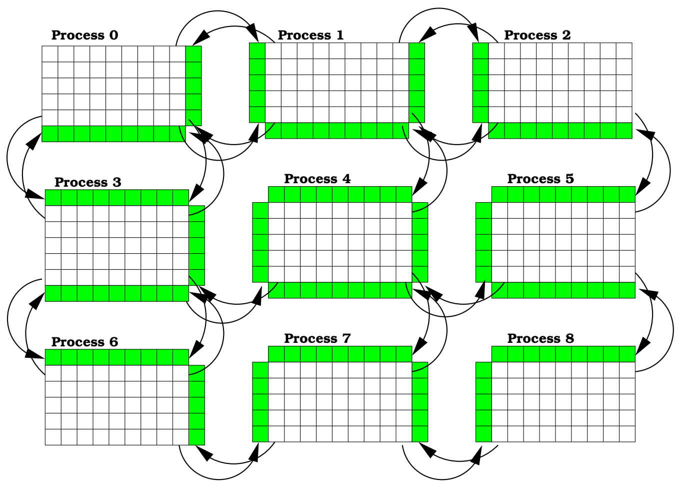

Fig. 1. Data exchange occurring in the 2-D neighborhood communication.

As of today, ADCL uses three attributes in order to characterize a version/implementation of the n-dimensional neighborhood communication:

1) Number of simultaneous communication partners: this attribute characterizes how many communication operations are initiated at once. For neighborhood communication, the currently supported values by ADCL are all (ADCL attribute value `aao`) and one (`pair`). 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, all of them based on the MPI specification [12], [13]. These data transfer primitives 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 `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.

III. Historic Learning

In the following, we would like to present the algorithms used to incorporate historic knowledge to improve the version management and selection algorithm of the library. We start by showing the necessity to improve the algorithm outlined in section II-A using two concrete examples. The algorithms developed to solve those problems can be split into separate sections: a statical analysis of the existing data in the history file, and how to derive decision based on the statistical data.

A. Motivating Examples

The first problematic scenario has been observed on a cluster (*cacau*), which consists of 200 nodes, each node equipped with a dual processor Intel EM64T processor at the High Performance Computing Center in Stuttgart, Germany. For our analysis we used 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. Although this network configuration is rarely found in large-scale HPC centers, it is not uncommon to have a hierarchical Gigabit Ethernet network in low-cost clusters found at many institutions.

ADCL managed to find in all test-cases the best performing implementation for the 3-dimensional neighborhood communication for a particular application. However, comparing the performance of the ADCL version of the code to a hand-tuned version shows, that the automatically tuned version has an overhead of 72% in the final execution time. A detailed analysis revealed, that the overhead stems from the fact that the runtime selection logic has to test versions of that function set which show extremly poor performance on this network. The difference in the execution time between the best and the worst performing version for 700 iterations of that code could be as high as 110 seconds vs. 210 seconds.

Second, a large class of HPC applications show a dynamic behavior with respect to the problem sizes used. As an example, simulations of the air-flow around the wings of an airplane require vastly different computational meshes depending on the wing-type, distance to the wing, air temperature etc. Modern codes do not create therefore a uniform mesh to compute the flow, but have local error criteria which are used to refine the computational mesh upon demand. This leads to a computational behavior, where the problem size is limited for a certain number of iterations in the code, before it might be refined/modified again. Using the version management and selection algorithm outlined in section II-A, ADCL would very often not finish the evaluation of the available code versions for a given problem size before the computational mesh and the problem size would be changed by the code again.

B. Pre-requisits for Historic Learning

The main goal of this paper is to develop algorithms which enhance the ADCL library with the capability of making a faster but still accurate /decision by reusing the knowledge learned from previous executions. A fundamental requirement to incorporate historic knowledge into the ADCL decision logic is to store information gained in one run into a history file such that the according information can be accessed and re-used by subsequent executions. ADCL has therefore been extended by a history file, which is stored in ADCL specific directory `.adcl`. Once a particular function-set has been evaluated for a given problem size, ADCl stores problem characteristics as well as the performance data for that problem. The history file itself is written in an XML format and can be displayed in a user friendly way in a web-browser by linking the XML file to an XSL file.

Characterizing the problem consists of the function-set which has been optimized/explored, the process topology used by the application, the problem sizes. set. The function-set identifies the operation which has been optimized, such as n-dimensional neighborhood communication. Furthermore, it also identifies the collection of different versions which can be explored, and optionally the attributes and attribute-values for each version. The process topology contains the information about the number of processes used and the logic relation between the processes. As an example, processes might be logically organized in a cartesian topology, where each process has a left/right/upper/lower neighbor. As of today, we require that the topology information between two problems is identical in order to explore similar entries in the history file, i.e. the application has to utilize the same function-set on the same number of process. We plan to relax however this requirement in the future, since the number of processes is very often less relevant than the topology itself, e.g. a 2-D cartesian process topology exposes nearly identical characteristics from the communication perspective for 100 or for 1000 processes. The problem size is a function-set specific characterization of the dimensions utilized by the application. For the neighborhood communication, the problem size is described by the length of the messages used to communicate with each of the neighboring processes. The focus of the algorithms presented in the subsequent subsections will be on identifying similar problems based on varying problem sizes, which will require a meaningful distance measure between two problem sizes for a given function set.

Another item in the history file will contain a graph describing the network topology between the processes. The connection between the processes can be characterized by the network interconnect utilized (InfiniBand, Gigabit Ethernet, etc.) and some simple characteristics such as network latency and bandwidth. This approach to characterize a network in a compact fashion has been explored e.g. in the *carto* framework of the Open MPI library [**?**]. This graph is not required for the second scenario described in section III-A, where we assume to have frequently changing problem size within the same execution, but would be required for introducing portability of the history files across different platforms. This feature is however not available as of today.

The performance data for a given function-set and problem size contains of the fastest implementation found in the function-set and the optimal attributes values characterizing this particular implementation. Furthermore, the history file also includes the performance data for all other versions tested. Although this might sound like large quantities of data which have to be stored, each code version tested contains in reality only a single floating-point data item in the history file.

C. Analysis of Historic Data

This algorithm 1 perform an analysis of the historic knowledge base and compute a statistical distance threshold to decide on problem sizes similarities. The starting point of the algorithm are a number of entries in the history file for a given function-set (`NbOfPBSizes`). For the sake of clarity, we consider for the rest of the paper only the entries for the function-set of interest, and ignore that the history file might contain other entries as well. The library defines the relative maximum tolerable performance penalty compared to the best performing implementation as p_{max}. The algorithm determines than for each entry in the history file the *top cluster*, i.e. the group of code versions whose execution time is within $p_{max}\%$ of the execution time of the version which achieved the best performance for that problem size.

The main loop of the algorithm determines now for each entry `PSR` the distance to all other entries `PST`, `PST=1..NbOfPbSizes`, `PST` $\neq$ `PSR` in the history file, and whether the winner version of an entry `PST` is in the top cluster of the entry `PSR`. If this is the case, a flag in a boolean array is set to *true* for the entry `PST`. Once all entries for a given problem size have been evaluated, we define the maximum distance for the problem `PSR` as the distance to the last entry in the boolean array, until which *all* entries are marked as *true*. To clarify the definition of the maximum distance in this context, please consider a boolean array with the entries (*true, true, false, true, false*). In this case, the maximum distance is the distance between the entry `PSR` and the problem size which lead to the second entry in the boolean array, since this is the element until which the winner of all entries delivered an acceptable decision for the problem size `PSR`.

Algorithm 1 AnalyseHistoricData(p_{max})

```
Require: Execution time of each implementation for different
  problem sizes and the corresponding winners.
  {Determine the versions within $p_{max}$% of the best one for
  each problem size (PS).}
  for PS=1 to NbOfPbSizes do
    ClusterTopVersions( PS, $p_{max}$ )
  end for
  {Check whether the winner of the problem size reference
  (PSR) is in the top versions of test problem size (PST)}
  for PSR=1 to NbOfPbSizes do
    for PST=1 to NbOfPbSizes do
      if PSR $\neq$ PST then
        Distance(PSR, PST)$\leftarrow$CompDist(PSR, PST)
        if Winner( PSR ) $\in$ TopVersions( PST ) then
          IsSimilar( PSR, PST ) $\leftarrow$ 1
        else
          IsSimilar( PSR, PST ) $\leftarrow$ 0
        end if
      end if
    end for
    $D_{max}$(PSR,p)$\leftarrow$CompMaxDist(Distance,IsSimilar)
  end for
```

Please note, that the algorithm automatically chooses the maximum distance to be zero in case a problem does not have any 'related' problems in the history file. Determining the distance between two problem sizes is function-set specific. For the n-dimensional neighborhood communication we use as of now the standard euclidean distance using the message length of the data items to be transfered to each neighboring process as the components.

The result of this analysis is the maximum distance for each problem size in the history file. This quantifies for a given problem size, that using a winner version of any problem sizes within the maximum distance leads to an acceptable performance for the problem size of interest, acceptable being defined as not more than $p_{max}\%$ above the optimal performance. As of today, the algorithm outlined above is run in a post-mortem analysis step. The maximum distance for each entry in the history file is stored along with all the other information outlined in the previous subsection. We envision these calculations however to be performed in a long-term every time a new entry is being added to the history file.

D. Version Management and Selection Algorithms using Historic Data

The algorithm as outlined above has one major restriction: assuming that ADCL has to execute a problem size for which no entry is available in the history file, algorithm 1 does not provide yet any hints on how to choose the code version to be used for the given function set. However, it is straight forward to extend this algorithm to provide a good starting point for a new problem size. We suggest to alternative approaches: given a new problem size `PSNew`, algorithm 2 *PredictFromClosest* predicts the code version by taking the winner version of the closest problem size in the history file, if `PSNew` is within the maximum distance determined for the closest problem size.

Alternatively, the algorithm 3 *PredictFromSimilar* suggests to use the winner of a weighted majority vote from the similar problem sizes available in the history file, considering again only the problem sizes for which `PSNew` is within their defined maximum distance as a result of algorithm 1.

Algorithm 2 PredictFromClosest(PSNew, D_{max}, p)

```
{Find the closest problem size and its winner}
$D_{min}$ ← Distance(PSNew, PS1)
PredictedWinner ← Winner(PS1)
for PS=1 to NbOfPbSizes do
    D ← Distance(PSNew, PS)
    if D $\leq D_{max}$(PS,p) & D $\leq D_{min}$ then
        $D_{min}$ ← Distance(PSNew, PS)
        PredictedWinner ← Winner(PS)
    end if
end for
{Estimate the execution time of the best version for the new problem size by interpolation of the execution times of the closest problem size }
EstExecTime ← EstExecTime(PredictedWinner)
{The estimated maximum execution time is the predicted execution time plus p%}
MaxExecTime ← EstExecTime*(1+p/100)
return {PredictedWinner, MaxExecTime}
```

In both algorithms, we can furthermore estimate the execution time of the predicted winner version by interpolating the data available of the related problem sizes involved in the prediction process. This predicted execution time can be used as a safety net. In case the measured execution time of the predicted winner function deviates more than $p_{max}\%$ from the predicted execution time, the library dismisses the historic data base and starts an optimization run for `PSNew`.

IV. Evaluation

A. Prediction accuracy

In the following, we analyze the prediction accuracy using both prediction algorithms; the one using the closest problem size as a reference for the decision and the one based on a weighted majority vote of similar problem sizes. We explored our prediction techniques for three different communication patterns namely the 1D, 2D and 3D neighborhood communication. For each communication pattern, we execute a simulation code for a large set of problem sizes :

- 50 problem sizes for 1D neighborhood communication from 24 to 128 data item per process.
- 55 problem sizes for 2D neighborhood communication from 32x32 to 72x72 mesh point per process.
- 60 problem sizes for 3D neighborhood communication from 32x32x32 to 64x64x64 mesh point per process.

For each problem size, we evaluate the execution time of 5000 iterations of n-dimensional neighborhood communication for each available implementation/version in ADCL. We identify each time the winner version chosen by the ADCL brute force selection logic as well. We run the prediction algorithms to make a decision on a winner version for each problem size based on the data of all other problem sizes which we consider then as our historic knowledge base.
In our evaluation, we used 16, 32 and 48 processes of an Opteron cluster using either InfiniBand or Gigabit Ethernet as a network interconnect. For each scenario: communication pattern and network interconnect, we compute the number of problem sizes which are well predicted using a given prodiction algorithm for a given acceptable performance window. In the figures 2 to 7, we compare the prediction accuracy of our two prediction algorithms for different values of the acceptable performance window expressed as the percentage of performance to the best version.
Analyzing those results, we come up with the following findings:

- The prediction algorithms have a good prediction rate more than 90% for an acceptable performance window of 10% in most of the cases. This validate the correctness of our approach for different scenarios. For the few problem sizes where the historic knowledge base provide a bad predicted winner, the library will be able to detect it

Algorithm 3 PredictFromSimilar(PSNew, D_{max}, p)

```
Initialization: Confidance[NbOfVersions] ← 0
for PS=1 to NbOfPbSizes do
    {Find the closest problem sizes}
    if Distance(PSNew, PS) $\leq D_{max}$(PS,p) then
        Confidance(Winner(PS))← Confidance(Winner(PS)) + 1/Distance(PSNew,PS))
    end if
end for
PredictedWinner ← Versions( Max(Confidance) )
{Estimate the execution time of the best version for the new problem size by interpolation of thr execution times of similar problem sizes }
EstExecTime ← EstExecTime(PredictedWinner)
{The estimated maximum execution time is the predicted execution time plus p%}
MaxExecTime ← EstExecTime*(1+p/100)
return {PredictedWinner, MaxExecTime}
```

using the predicted execution time and will then ignore the historic data base and starts an new optimization using one of the selection algorithms of ADCL.

- The prediction rate is increasing when we increase the acceptable performance window. This is predictable since we allow then more flexibility in term of performance acceptance.
- The prediction rate is better with smaller number of processes, however, it is still good for large number of processes. As an example, for the 2D pattern over Infiniband interconnect (figure 4), for an acceptable performance window of 10%, the prediction rate is around 95% when using 16 or 32 processes and around 85% when using 48 processes.
- In most of the scenarios, the weighted majority vote technique is performing better than the one based on the closest problem size in term of prediction. We can see that very easily in the 1D case for 32 and 48 processes (figures 2and 3).

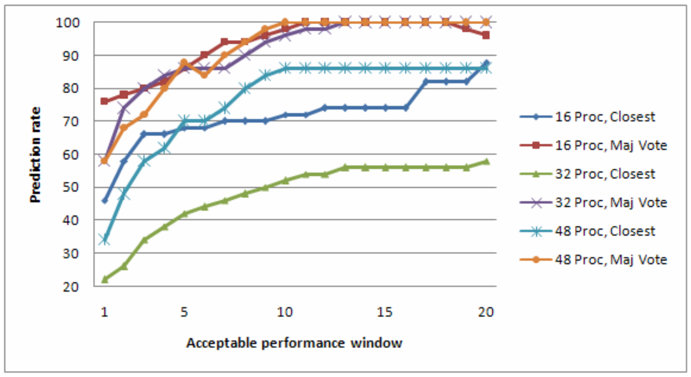

Fig. 3. 1D neighborhood communication over Gigabit Ethernet

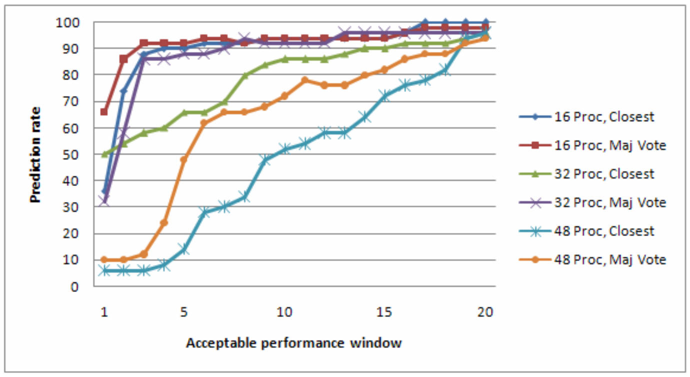

Fig. 2. 1D neighborhood communication over infiniband

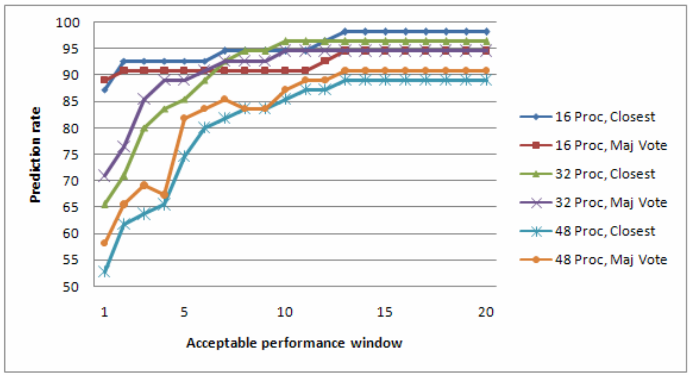

Fig. 4. 2D neighborhood communication over infiniband

B. Performance benefit of historic learning

In this second part of our evaluation, we evaluate the performance benefit of using the historic learning capability introduced within ADCL. The results obtained in [8] indicate that the performance of the application using ADCL is close to the performance when using the best implementation on that platform. However, there is a slight overhead when using ADCL due to the fact, that we are also executing non-optimal versions during the selection phase. The performance of ADCL can further be improved by using our new approach introduced in this paper.

In this part we used a real scientific application consisting of a parallel, iterative solver 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 performing 3D neighborhood communications. We evaluate the execution time of this application using the two existing selection algorithms already available within ADCL: the brute force search and the heuristic based algorithm and using the historic knowledge base. We used in those measurements 32 processes of the same cluster over Gigabit Ethernet interconnect.

Figure 8 shows, that the heuristic based selection logic outperforms the brute force method, but can even be improved dramatically by using the historic learning capability within ADCL.

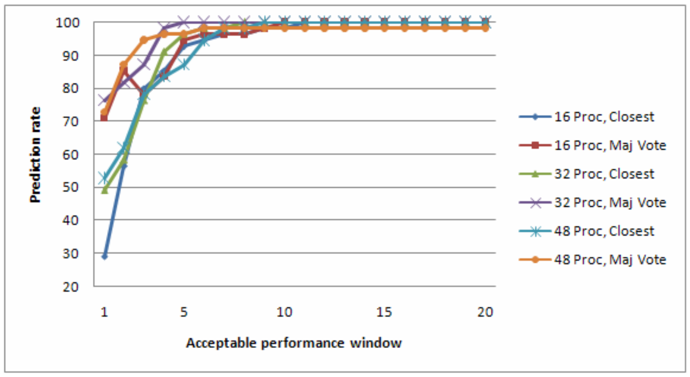

Fig. 5. 2D neighborhood communication over Gigabit Ethernet

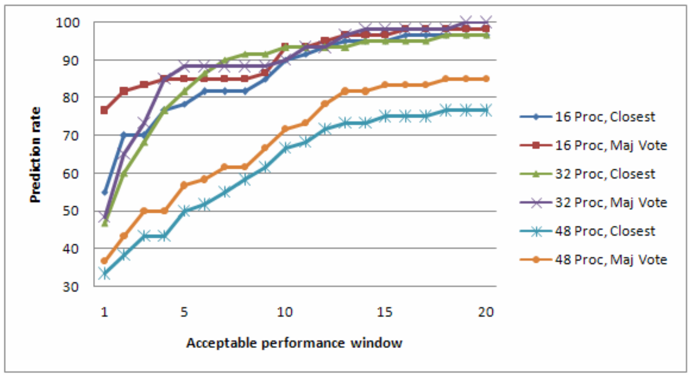

Fig. 7. 3D neighborhood communication over Gigabit Ethernet

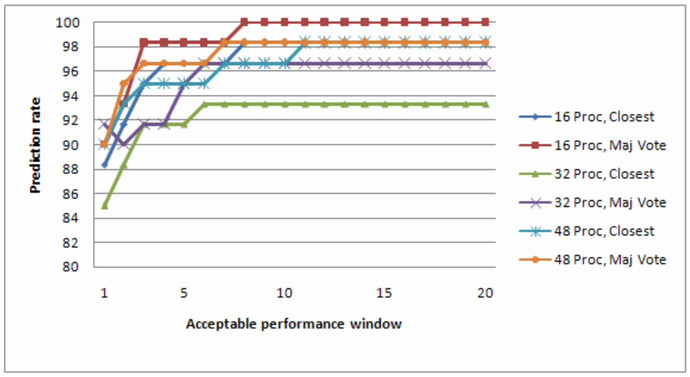

Fig. 6. 3D neighborhood communication over infiniband

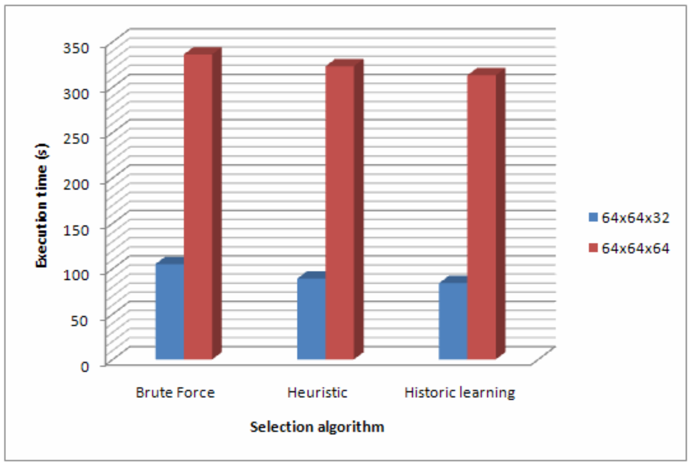

Fig. 8. Historic learning performance benefit

V. Related Work

VI. Conclusion

Acknowledgments

This research was funded by a gift from the Silicon Valley Community Foundation, on behalf of the Cisco Collaborative Research Initiative of Cisco Systems.

References

[1] 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.

[2] J. Pjesivac-Grbovic, G. Bosilca, G. E. Fagg, T. Angskun, and J. J. Dongarra, “MPI Collective Algorithm Selection and Quadtree Encoding,” *Parallel Computing*, vol. 33, pp. 613–623, 2007.

[3] 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)*, October 2003, pp. 538–547.

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

[5] E. Gabriel and S. 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*, 2007, p. 185.

[6] E. Gabriel, S. Feki, K. Benkert, and M. Chaarawi, “The Abstract Data and Communication Library,” *accepted for publication in Journal of Algorithms and Computational Technology*, p. t.b.d., 2008.

[7] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki, “An Open Tool for Parameter Optimization,” in *accepted for publication at the EuroPVM/MPI conference*, Dublin, Ireland, September 2008.

[8] E. Gabriel, S. Feki, K. Benkert, and M. M. Resch, “Towards Performance and Portability through Runtime Adaption for High Performance Computing Applications,” in *accepted for publication at the International Supercomputing Conference*, Dresden, Germany, June 2008.

[9] R. Vuduc, J. W. Demmel, and J. A. Bilmes, “Statistical Models for Empirical Search-Based Performance Tuning,” *International Journal for*

High Performance Computing Applications, vol. 18, no. 1, pp. 65–94, 2004.

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

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

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

[13] ——, *MPI-2: Extensions to the Message Passing Interface*, July 1997, http://www.mpi-forum.org/.