Optimizing MPI Runtime Parameter Settings by Using Machine Learning*

Simone Pellegrini, Jie Wang, Thomas Fahringer, and Hans Moritsch

University of Innsbruck – Distributed and Parallel Systems Group
Technikerstr. 21A, 6020 Innsbruck, Austria
{spellegrini,wangjie,tf,hans}@dps.uibk.ac.at

Abstract. Manually tuning MPI runtime parameters is a practice commonly employed to optimise MPI application performance on a specific architecture. However, the best setting for these parameters not only depends on the underlying system but also on the application itself and its input data. This paper introduces a novel approach based on machine learning techniques to estimate the values of MPI runtime parameters that tries to achieve optimal speedup for a target architecture and any unseen input program. The effectiveness of our optimization tool is evaluated against two benchmarks executed on a multi-core SMP machine.

Keywords: MPI, optimization, runtime parameter tuning, multi-core.

1 Introduction

Existing MPI implementations allow the tuning of runtime parameters providing system administrators, end-users and developers the possibility to customise the MPI environment to suit the specific needs of applications, hardware or operating environments. An example is the opportunity to change the semantics of point-to-point communications in relation to the size of the message being transmitted. According to a *threshold* (runtime parameter) value the library can use an *eager* protocol when small messages are exchanged or the more expensive *rendezvous* method for larger messages. A *default* setting, designed to be a good compromise between functionalities and performance, is nevertheless provided by the MPI library to allow easy deployment of MPI applications. Open MPI provides an entire layer – called Modular Component Architecture (MCA) [1] – with the purpose of providing a simple interface to tune the runtime environment. The current development version of Open MPI has several hundred MCA runtime parameters. This large number of configurable parameters makes the manual tuning of the Open MPI environment particularly difficult and challenging.

Considering a subset of tunable runtime parameters, determining the values that optimize the performance of an MPI program on a *target* system is not trivial and can depend on several factors (e.g. number of nodes in a cluster, type

* This work is funded by the Tiroler Zukunftsstiftung under contract nr. P7030-015-024.

of network interconnection and layout and amount of shared cache in a single node). This optimization problem is comparable with the selection of the *optimization sequence* (usually expressed as a vector of flags) performed by modern compilers. As the best sequence that reduces the execution time of a program strongly depends on the underlying architecture, and because the definition of effective cost models is not always feasible, *feedback*-driven *iterative* techniques (also known as *iterative compilation*) have been employed [2]. The number of possible *settings* (or *configurations*) of runtime parameters, and thus the corresponding *optimization space*, is too large to be manually explored. For these reasons, tools have been developed in order to *automate* the process of evaluating the execution of an MPI program considering numerous combinations/settings of these parameters in order to find the one with the best execution time [3]. Even if the process is automated, the exhaustive exploration of the optimization space could be very expensive in terms of time; and it makes sense only if the cost of the optimization phase can be amortised over many runs of the program.

In this paper we introduce a mechanism to estimate *optimal* runtime parameter settings for MPI programs running on any hardware architecture. The approach is based on *machine learning* (ML) techniques which use specific knowledge of the underlying system (acquired during an *off-line* training phase) to build a *predictor* capable of estimating the best setting for a subset of runtime parameters for any unseen MPI input program. A program is described by a set of *features* extracted both *statically* by analysing the source code and *dynamically* by profiling *one* run of the program. Although our approach is general, we focus on the optimization of MPI applications running on a multi-core SMP node as the increased deployment of multi-core systems in clusters makes intra-node optimizations fundamental in the overall application performance [4]. As a result of our work we show that despite the information used to describe a program being rather simple, the estimated optimal settings of the runtime parameters always outperform the *default* one and achieve, on average, around 90% of the available performance improvement. Two ML algorithms (i.e. decision trees and artificial neural networks) are used, and the accuracy of their prediction is evaluated against two different benchmarks: the Jacobi relaxation method [5] and the Integer Sort (IS) benchmark of the Nas Parallel Benchmarks suite [6].

The rest of the paper is organised as follows. Section 2 depicts the impact of runtime parameters. In Section 3, we introduce our machine learning framework, decision trees and artificial neural networks. Section 4 presents the results and accuracy of the optimal runtime parameter values prediction for the two benchmarks. Section 5 discusses related work, and Section 6 concludes the paper with some brief considerations and an outlook to future work.

2 Impact of Runtime Parameter Settings

The motivation behind our work is that the correct setting of runtime parameters within an application and an architecture can result in a valuable performance gain of the program itself. Exploring the entire optimization space for MPI applications

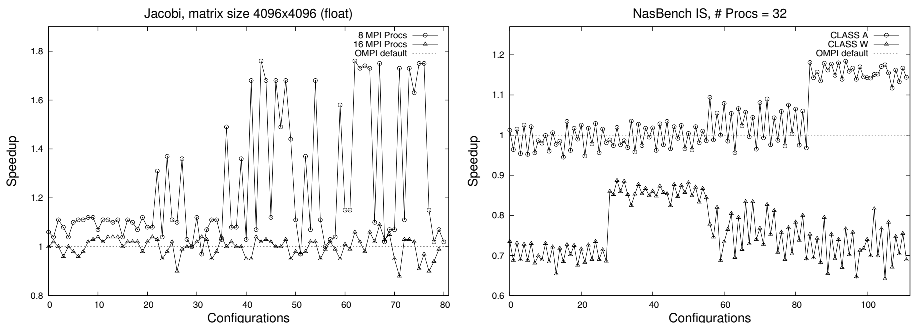

Fig. 1. Variation of the speedup of the Jacobi method (on the left) and IS benchmark (on the right) against combinations of the runtime parameters using different communicator (for Jacobi) and problem (for IS) sizes

requires the execution of the same program using a large number of different configurations which exponentially increases with the amount of runtime parameters considered. Formally, we define a *configuration* (C_j) of a set of n runtime parameters $(p_1, p_2, \ldots, p_n)$ as a vector $C_j = (c_{1,j}, c_{2,j}, \ldots, c_{n,j})$ where $c_{i,j}$ is the value (a buffer size or a threshold/flag value) associated with the parameter p_i.

According to the particular program to optimize and the underlying architecture a *pre-selection* of *interesting* tunable runtime parameters can be done. For example, Open MPI's MCA provides an entire framework for the optimization of collective operations (i.e. `coll`). Although this module offers several parameters to select which kind of algorithm to use or the maximum depth of the tree employed to perform reduction operations, it is not useful if the target application does not contain collective operations.

Fig. 1 depicts the impact of the variation of 4 runtime parameters on the performance of the Jacobi method and IS benchmark. The graph on the left compares the speedup achieved by the Jacobi method (executed with 8 and 16 MPI processes) considering several configurations. As the values of 4 different runtime parameters are varied across configurations, vectors of parameter values have been numbered in order to make a 2-dimensional representation possible. For the Jacobi, configurations that achieve a high speedup using 8 processors turn to be very inefficient when 16 processes are used (and vice-versa). In addition to the number of processes being used, the problem size plays an important role. The graph on the right in Fig. 1 shows the speedup achieved by the IS benchmark using two different problem sizes (i.e. classes `A` and `W`) while varying the configuration of the runtime parameters. None of the tested configurations outperforms the Open MPI default for the small problem size (class `W`). However, for the larger problem size (class `A`) some configurations achieve a performance gain of up to 18%.

2.1 Selection of Runtime Parameters

As the focus of our work is the performance optimization of MPI programs running on a multi-core SMP machine, the optimization space can be reduced by selecting those parameters whose effects are relevant in a shared memory system. Preliminary experiments have been conducted in order to determine the set of parameters with high impact on the performance of MPI application. Four parameters have been selected known to have a significant impact on point-to-point and collective operations:

sm_eager_limit: threshold (in byte) which decides when the MPI library switches from *eager* to *rendezvous* mode (for `Send` and `Receive` operations)
mpi_paffinity_alone: flag used to enable *processor affinity* (if enabled (`1`), each MPI process is exclusively bound to a specific processor core)
coll_sm_control_size: buffer size (in byte) chosen to optimize collective operations (usually sized according to the size of the shared cache)
coll_tuned_use_dynamic_rules: flag which enables *dynamic* optimization of collective operations.

Considering a threshold value and a buffer size in the range of $[2^9, \ldots, 2^{28}]$ (a total of 20 values if power of 2 values are considered) and 2 possible values for the flags, the number of possible configurations generated by the variation of these 4 parameters is $20^2 * 2^2 = 1600$. The use of automatic iterative techniques for finding the best configuration in this four-dimensional space for an application with an execution time of 5 minutes would take around 5.5 days.

3 Using Machine Learning to Estimate Optimal Configurations

The challenge is to develop a predictive model that analyses any MPI *input program*, and according to the gained knowledge of how the *target* architecture behaves, determines the *values* – for a set of pre-defined runtime parameters – which achieve *optimal* speedup. As the prediction depends on the characteristics of the input program and its input data, we define five *program features* (see Table 1), extracted both *statically* and *dynamically*, which allow MPI programs to be *classified*. Furthermore, the target system has to be described in order to obtain a configuration of runtime parameters which is optimal for the underlying architecture. The behaviour of a system is nevertheless non-trivial to model as several variables (e.g. number of cores, amount of private/shared cache, memory and network latency/bandwidth) must be considered; additionally, even with the complete knowledge of the system, interactions between components can not be described by simple analytical models.

To solve this problem, we use machine learning (ML) techniques; the main idea is illustrated in Fig. 2. A set of training programs (described by *program features*) is executed on the *target architecture* using several configurations of the selected runtime parameters (see Section 2.1). Each program of the training set is executed against a representative set of parameter configurations; the execution time is measured and *compared* with the one obtained using the Open MPI's *default* parameter values (*baseline*). The resulting *speedup*, together with the program features and the current configuration, is stored and used to train a

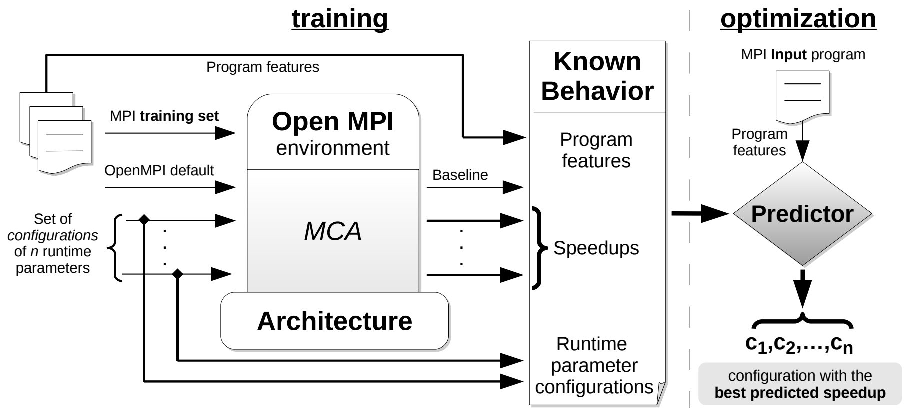

Fig. 2. Training and use of the predictor

predictor. This phase, also known as *training* or *learning* phase, is executed *off-line* and it must be repeated every time the target architecture changes (or when a prediction for a different system is needed). During the optimization phase, when an *unseen* MPI *input program* is presented to the predictor, its program features are extracted and used to determine the configuration which optimizes the execution time of the *input program*.

We use two standard machine learning techniques, *Decision Trees* [7] and *Artificial Neural Networks* [8]. The construction of the two ML-based predictors is compared and their accuracy is evaluated using two different benchmarks (i.e. Jacobi method and IS). This section continues by describing the three key phases of our approach: (*i*) MPI program characterisation, (*ii*) generation of training data and (*iii*) prediction model construction.

3.1 Characterising MPI Programs

A program is described by a set of *features* which is extracted both *statically* and *dynamically*. As the runtime parameters only affect the communication between processes, a program is described by *analysing* its MPI statements. The *communication pattern*, amount of *exchanged data* and the *communicator size* are extracted from any MPI program by instrumenting and tracing its execution *once*. Table 1 displays the features used to characterise the behaviour of MPI programs.

The set of program features considered in this paper is relatively small as our goal is to evaluate the suitability of machine learning techniques for the estimation of MPI runtime parameter values. Furthermore, as the training set is designed in accordance with program features (see Section 3.2), this characterisation enables a fast training phase. In Section 4 we show that even if the program characterisation is rather simple, the predicted configurations of the runtime parameters achieve, on average, 90% of the available performance improvement.

Table 1. MPI program features

`coll_ratio`	ratio between the amount of collective operations and the total number of communication operations
`coll_data`	average amount of data exchanged in collective operations
`p2p_ratio`	ratio between the amount of point-to-point operations and the total number of communication operations
`p2p_data`	average amount of data exchanged in point-to-point operations
`comm_size`	number of processes involved in the communication

3.2 Generating Training Data

According to the defined program features, we designed a training set of micro-benchmarks with the purpose of generating training data. The training-set is composed of three micro-benchmarks: `p2p`, `coll` and `mixed`.

`p2p`: measures the execution time of a *synchronous* `send/recv` operation between two processes using different message sizes

`coll`: measures the execution time of a *collective* operation, `alltoall`, varying the communicator and message sizes

`mixed`: mixes point-to-point and collective operations varying the message and communicator sizes

The three training programs have been executed on our target architecture using several configurations of the 4 selected runtime parameters. Each configuration was executed 10 times, and the mean value of the measured execution time has been used to calculate the speedup. The resulting set of training data contains around 3000 instances.

3.3 Predictor Model Construction

The training data is stored in a repository (also called *known behaviour* in Fig. 2), and each entry is encoded as a vector of 10 values:

$$\{(f_{1,j}, \ldots, f_{5,j}), (c_{1,k}, \ldots, c_{4,k}), \textit{speedup}\}$$

Each vector represents the *speedup* achieved by one of the training programs whose i-th feature values $f_{i,j}$ and was executed using $c_{i,k}$ as the value of the i-th runtime parameter. The known behaviour is used to build two predictor models based on different machine learning algorithms.

Decision Trees. We use `REPTree`, a fast *tree learner* which uses reduced-error pruning and can build a regression tree [7], to train a predictor from the repository data. The speedups, together with program features ($f_{i,j}$) and runtime parameter ($c_{i,k}$) values are used to model the input/output relationship in the form of *if-then* rules. The constructed decision tree is able to *estimate* the speedup of a program (described by its feature F_i) running with a specific configuration C_j of the runtime parameters. A formal representation can be given as follows: let dt be the decision tree model, the predicted speedup $S = dt(F_i, C_j)$. For a specific input program F_x, the best configuration of the runtime parameters C_{best} would be the one with the highest predicted speedup $S_{max} = dt(F_x, C_{best})$.

Artificial Neural networks. *Artificial Neural Networks* (ANNs) [8] are a class of machine learning models that can map a set of input values to a set of target values [9]. We use ANNs because are robust to noise and they have been successfully used in modeling both linear and non-linear *regression problems* [9]. As opposed to decision tree model, ANNs are trained only using the configurations of runtime parameters with the best speedup for each distinct vector of program features. Its input/output relationship can be described as follows: let ann be the ANN model, $C_{best} = ann(F_x)$. A three-layer feed-forward back-propagation network [8] is used and the ANN structure with the best performance for our problem is chosen as below. The hidden layer of the network contains 10 neurons and the the transfer function is *hyperbolic tangent sigmoid*, for the output layer the *logarithmic sigmoid* function has been used.

4 Experimental Results

In this section we evaluate the accuracy of the two predictors based respectively on decision trees and neural networks for the Jacobi method and IS benchmark. To run the experiments we used a 32-cores Sun X4600 M2 server with AMD Opteron 8356 (Barcelona) processors. In this system there are 8 sockets with one processor each. Each chip contains, there are four cores – with private L1 and L2 cache (512 KB) – which share an L3 cache of 2 MB. The system runs CentOS version 5 (kernel 2.6.18) 64 bits and the Open MPI version used is 1.2.6 (compiled with gcc-4.1.2). We use Weka's implementation of `REPTree` [7] and Matlab's Neural Network Toolbox [10] for training the ANN-based model.

4.1 Jacobi Relaxation Method

The considered Jacobi implementation uses a 2-dimensional matrix split where the rows are exchanged synchronously between neighbour processes. At the end of each iteration, a *reduce* operation is performed to calculate the residual error. The values of the program features associated to the Jacobi implementation are the following. As in each iteration a process performs 4 point-to-point operations (i.e. 2 sends and 2 receives) and only one collective operation (i.e. reduce) `coll_ratio` and `p2p_ratio` can be statically evaluated respectively as $4/5 = 0.8$ and $1/5 = 0.2$ (which means that 80% of the communication time is spent in point-to-point operations and the remaining 20% in collective ones). The value `p2p_data` is the size of the exchanged row and thus depends on the problem size. The Jacobi method has been executed considering different matrix sizes and varying the number of MPI processes (8, 16 and 32).

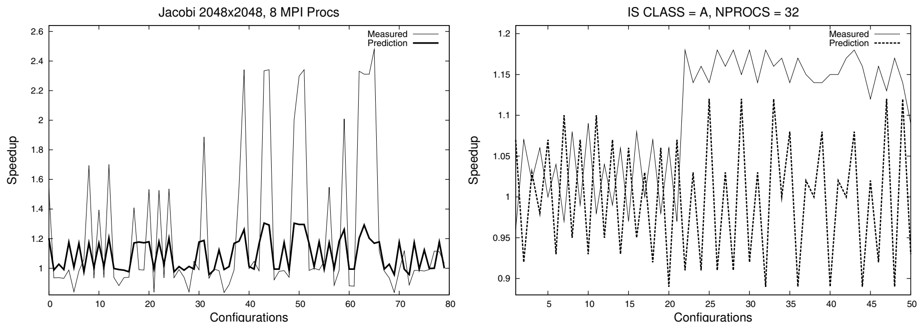

Fig. 3. Predicted and measured speedup using `REPTree`

The result of the predictor based on decision trees is depicted in Fig. 3. The graph on the left compares the predicted and measured speedup for the Jacobi method using 8 processes (because of space limits predictions for 16 and 32 processes are not shown). As stated in the previous sections, the decision tree-based predictor is capable of estimating the speedup of a program running with a specific configuration of the runtime parameters. In order to obtain the *best setting*, the predictor is queried several times using different values of the runtime parameters. The configuration with the highest predicted speedup contains the best runtime parameter settings for the Jacobi method on the target architecture. For example, in Fig. 3 the highest predicted speedup is 1.305 which corresponds to configuration number 46 (the respective values of the runtime parameters are {`262144,1,262144,0`}); its measured speedup is 2.3. The best possible speedup achieved by Jacobi is for configuration number 68 (i.e. {`32768,1,2097152,1`}), with a value of 2.48. Therefore, in this case, the trained decision tree predicts a configuration that gives 95% of the maximum possible performance improvement. The structure of the decision tree built from training data is depicted in Fig. 4. Because of space limit, the model has been simplified reducing the depth from 11 to 3. The impact of a specific feature/parameter on the value of speedup (represented in leafs) depends from its depth in the tree.

The results of the neural network-based predictor are depicted in Table 2. The table shows Jacobi's features vector followed by the configuration of the runtime parameters that gives the best speedup (for the particular feature vector) followed by the predicted best configuration with the respective *measured* speedup. The speedup achieved by the predicted configurations achieve on average 94% of the available speedup.

4.2 Integer Sort (IS)

The characterisation of the IS benchmark has been performed by instrumenting the source code and tracing one run for each problem size (i.e. class `A` and `B`). The core routine of the benchmark contains calls to `MPI_Alltoallv` and `MPI_Allreduce` operations. As our classification makes no distinction between

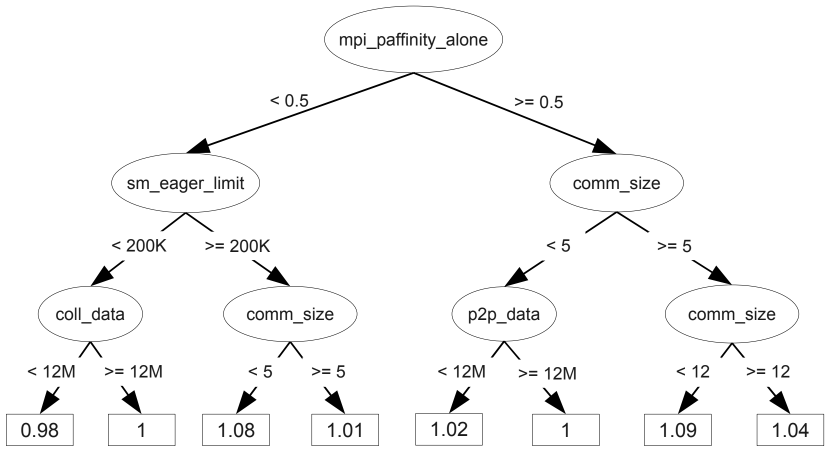

Fig. 4. Decision tree derived from the training data (`depth = 3`)

the two, the program has been characterised with `coll_ratio` = 1 (i.e. 100% of collective operations). The amount of data exchanged by the `MPI_alltoallv` operation depends on both the problem and the communicator size.

The result of the prediction based on `REPTree` for problem size `A` using 32 processes is depicted on the right of Fig. 3. For this benchmark, the predicted best configuration achieves around 84% of the maximum speedup. The `mpi_paffinity_alone` parameter value plays a key role in this benchmark, configurations with the flag value set to 0 always have better performance compared to the ones where the flag is activated (this behaviour can be observed in the graph comparing the speedup of two consecutive configurations). However, the predictor fails to correctly estimate the effect of the parameter on the speedup as the configurations where the affinity flag is equal to 1 are predicted to be faster. The other runtime parameter that clearly has effects on the speedup is the `sm_eager_limit` threshold. As the considered problem size requires to exchange around 32 KB of data among the processes, setting a value of 32 KB for the threshold (from configuration number 22) forces the library use the eager protocol instead of the more expensive rendezvous. This effect is correctly estimated by the predictor as the returned best configuration is {`32768,1,2048,1`}.

In table 2, the accuracy of the ANN-based predictor is depicted. As for decision trees, the prediction of the affinity flag is wrong and the accuracy of the two predictors is comparable. The wrong prediction of the affinity bit can be explained considering the workload of the IS benchmark. As intra-chip communication is faster than inter-chip communication in our test system and because of the nature of the training set (based on point-to-point and collective operations), the trained predictor always turns on the affinity (see predicted configurations in Table 2). However, in computation-bound scenarios, affinity scheduling can cause performance degradation due to the limited amount of shared cache and available memory bandwidth per chip.

Table 2. Best measured and predicted configurations using ANN

Jacobi Method				
Feature Vector	Best conf.	**speedup**	Predicted conf.	**speedup**
0.2,1,0.8,1024, 8	32768,1, 512,1	**3.83**	65536,1, 32768,1	**3.74**
0.2,1,0.8,1024,16	262144,1, 512,0	**1.09**	2048,1, 512,0	**1.02**
0.2,1,0.8,2048, 8	32768,1,2097152,1	**2.48**	65536,1, 32768,0	**2.36**
0.2,1,0.8,2048,16	262144,0, 262144,0	**1.09**	262144,1, 512,0	**1.03**
0.2,1,0.8,4096, 8	32768,1, 512,0	**1.76**	32768,1, 32768,0	**1.57**
0.2,1,0.8,4096,16	262144,1, 262144,0	**1.09**	512,1,2097152,0	**1.03**
Integer Sort Benchmark				
Feature Vector	Best conf.	**speedup**	Predicted conf.	**speedup**
1, 262400,0,0,32	32768,0,512,1	**1.18**	32768,1,32768,0	**1.14**
1,1024000,0,0,32	32768,0,512,1	**1.07**	65536,0, 8192,0	**1.03**

5 Related Work

Most of the work about optimization of MPI applications focuses on *collective operations*. STAR-MPI (Self Tuned Adaptive Routines for MPI collective operations) [11], provides several implementations of the collective operation routines and it *dynamically* selects the best performing algorithm (using an empirical technique) for the application and the specific platform. Jelena et al. [12] also address the same problem by using machine learning techniques (decision trees). Compared to our work, these approaches are less general as the optimization is only limited to collective operations. Furthermore, the significant overhead introduced by the dynamic optimization environment makes these tools not suitable for short-running MPI programs.

The only work in literature that is comparable with our approach is OPTO [3]. OTPO systematically tests large numbers of combinations of Open MPI's runtime parameters *for common communication patterns* and *performance metrics* to determine the *best* set for a specific benchmark under a given platform. Differently from our approach, OPTO needs several runs of the application in order to find the optimal set of the runtime parameters and the search process is helped by configuration files provided by the user which specify the combinations and the parameter values that should be tested by the tool.

6 Conclusions and Future Work

This paper presents a novel approach to optimize MPI runtime environments for any application running on a target architecture. We build two predictors (based on machine learning techniques) capable of estimating the values, of a subset of Open MPI runtime parameters, which optimize the performance of unseen input programs for the underlying architecture. In contrast to existing optimization tools, our predictor needs only one run of the input program and no additional knowledge has to be provided by the user. No runtime overhead is introduced as the model is built and trained *off-line*. Experiments demonstrate that the predicted optimal settings of runtime parameter for different programs achieve on average 90% of the maximum performance gain.

Future work will consider a better characterisation of MPI programs to improve the quality of the prediction (e.g. by introducing the computation-to-communication ratio to predict the affinity flag). Furthermore, we will apply our predictor to other hardware configurations (e.g. clusters of SMPs).

References

1. Open MPI: Modular Component Architecture, `http://www.open-mpi.org`
2. Bodin, F., Kisuki, T., Knijnenburg, P., O'Boyle, M., Rohou, E.: Iterative Compilation in a Non-Linear Optimisation Space. In: Proceedings of the Workshop on Profile Directed Feedback-Compilation (October 1998)
3. Chaarawi, M., Squyres, J.M., Gabriel, E., Feki, S.: A tool for optimizing runtime parameters of open mpi. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 210–217. Springer, Heidelberg (2008)
4. Chai, L., Lai, P., Jin, H.W., Panda, D.K.: Designing an efficient kernel-level and user-level hybrid approach for mpi intra-node communication on multi-core systems. In: ICPP 2008: Proceedings of the 2008 37th International Conference on Parallel Processing, Washington, DC, USA. IEEE Computer Society Press, Los Alamitos (2008)
5. Meijerink, J.A., Vorst, H.A.v.d.: An iterative solution method for linear systems of which the coefficient matrix is a symmetric m-matrix. Mathematics of Computation 31(137), 148–162 (1977)
6. der Wijngaart, R.F.V.: Nas Parallel Benchmarks Version 2.4. Technical Report NAS-02-007, Computer Science Corporation NASA Advanced Supercomputing (NAS) Division (October 2002)
7. Ian, H., Witten, E.F.: Data Mining: Practical Machine Learning Tools and Techniques. Elsevier, Amsterdam (2005)
8. Bishop, C.M.: Neural Networks for Pattern Recognition. Oxford University Press, Oxford (1996)
9. Ipek, E., Supinski, B.R., Schulz, M., Mckee, S.A.: An Approach to Performance Prediction for Parallel Applications. In: Euro-Par Parallel Processing (2005)
10. Matlab: Neural Network Toolbox, `http://www.mathworks.com/products/neuralnet/`
11. Faraj, A., Yuan, X., Lowenthal, D.: Star-mpi: self tuned adaptive routines for mpi collective operations. In: ICS 2006: Proceedings of the 20th annual international conference on Supercomputing, pp. 199–208. ACM, New York (2006)
12. Fagg, G.E., Angskun, T., Bosilca, G., Dongarra, J.J.: Decision trees and mpi collective algorithm selection problem (2006)