Autotuning of MPI Applications Using PTF

Pre-print version of accepted paper

Anna Sikora
anna.sikora@uab.cat

Eduardo César
eduardo.cesar@uab.cat

Computer Architecture and Operating Systems Dept. Universitat Autònoma de Barcelona. Barcelona, Spain

Isaías Comprés
compresu@in.tum.de

Michael Gerndt
gerndt@in.tum.de

Computer Science Dept. Technische Universität München. München, Germany

ABSTRACT

The main problem when trying to optimize the parameters of libraries, such as MPI, is that there are many parameters that users can configure. Moreover, predicting the behavior of the library for each configuration is non-trivial. This makes it very difficult to select good values for these parameters. This paper proposes a model for autotuning MPI applications. The model is developed to analyze different parameter configurations and is expected to aid users to find the best performance for executing their applications. As part of the AutoTune project, our work is ultimately aiming at extending Periscope to apply automatic tuning to parallel applications. In particular, our objective is to provide a straightforward way of tuning MPI parallel codes. The output of the framework are tuning recommendations that can be integrated into the production version of the code. Experimental tests demonstrate that this methodology could lead to significant performance improvements.

CCS Concepts

•Computing methodologies → Parallel computing methodologies; •Theory of computation → *Parallel algorithms;*

Keywords

Autotuning; MPI; runtime parameters; PTF;

1. INTRODUCTION

MPI is the "de facto" standard for inter-process communication in distributed parallel programs and thus it represents a key factor in the optimization of MPI-based applications. However, a library setup for a specific system might not perform equally in different environments (e.g., different architecture or interconnection network). To increase portability, MPI implementations provide multiple configuration parameters. These parameters are usually set by experienced users with a deep knowledge of a specific MPI application and how it might behave on the target architecture.

The main problem when trying to optimize the parameters of the libraries that handle the communication among processes in parallel applications is that there are many parameters that users can configure, and predicting the behavior of the library for each configuration is non-trivial. This makes it very difficult to select good values for these parameters. The fact that we have so many parameters, several with many possible values, makes it difficult to exhaustively explore all the possible configurations. For example, IBM MPI, Intel MPI, and Open MPI include from more than 50 to more than 150 parameters and, in this set, from ten to several tens of them can influence performance. These are the main motivations for automating the process of testing configurations, and for providing heuristic search algorithms to explore the search space in a reasonable time.

The researchers and the vendors of parallel architectures developed a number of performance analysis (PA) tools that support and partially automate the tuning process. Much research has been dedicated in the last years to the development of auto-tuning strategies and tools to provide the application developer with hints on how to tune their code.

This paper presents a model for autotuning MPI applications that addresses the aforementioned optimizations of MPI applications and is expected to aid users to find the best performance for executing their applications. This work was part of the AutoTune project [11], which extended Periscope [2], an automatic online and distributed performance analysis tool developed by Technische Universität München, with automatic tuning capabilities. Plugins, provided by AutoTune, can use the performance properties and bottlenecks found by Periscope to come up with performance improvements for the application.

The goal of AutoTune is to close the gaps in the application tuning process and thus to simplify the development of efficient parallel programs on a wide range of architectures. Periscope Tuning Framework (PTF) is unique, since it is the first work to *combine analysis and tuning* of *multiple aspects* into an *online* automatic tuning framework. PTF is able to:

- identify tuning variants based on codified expert knowledge;
- evaluate the variants online (i.e., within the same execution of an application), reducing the overall search time for a tuned version;
- address a wide and extensible range of tuning aspects

SEM4HPC'16, May 31 2016, Kyoto, Japan

ISBN 978-1-4503-4351-0/16/05...$15.00

DOI: http://dx.doi.org/10.1145/2916026.2916028

through its plugin-based structure, including energy consumption properties; and

- provide recommendations on how to improve the code, which can be manually or automatically applied.

PTF executes both performance analysis and performance tuning using an online approach. During the execution of the application, the analysis is carried out and the found performance and energy properties are forwarded to the tuning plugins, which determine code alternatives and evaluate the different tuned versions. Finally, detailed recommendations are given to application developers on how to improve their code with respect to performance and energy consumption.

The remainder of this paper is as follows. Section 2 reviews similar works and describes other approaches being used in the autotuning field. Section 3 introduces the functionality of PTF. Next, Section 4 presents the model for autotuning MPI parameters for MPI applications. Section 5 describes the MPI Parameter plugin that is integrated into PTF. In Section 6, a set of experiments is presented to show the improvements that are obtained by using the developed plugin in PTF. Finally, Section 7 concludes the work.

2. RELATED WORK

The complexity of today's parallel architectures has a significant impact on application performance. In order to avoid wasting energy and money due to low utilization of processors, developers have been investing significant time into tuning their codes. However, tuning implies searching for the best combination of code transformations and parameter settings of the execution environment, which can be fairly complicated. Thus, much research has been dedicated to the areas of performance analysis and auto-tuning.

The explored techniques, similar in approach to ours, can be grouped into the following categories:

- self-tuning libraries for linear algebra and signal processing like ATLAS [19], OSKI [18] and SPIRAL [14];
- tools that automatically analyze alternative compiler optimizations and search for their optimal combination [17, 8, 12, 7];
- auto-tuners that search a space of application-level parameters that are believed to impact the performance of an application [5, 3];
- frameworks that try to combine ideas from all the other groups [16].

Performance analysis and tuning are currently supported via separate tools. AutoTune aims at bridging this gap and integrating support for both steps in a single tuning framework. The fact that tuning the MPI parameters allows for significantly improving the application performance is demonstrated by the development of tuning tools such as mpitune [6] and OPTO [4]. These tools are similar to our approach in the sense of executing the application for testing different parameter combinations. However, they are specific to a particular MPI implementation, while PTF can be applied to any MPI flavour.

Pellegrini et al. [13] propose a machine learning approach to the MPI parameter tuning. This approach is faster than the one followed by PTF because it only runs the application once and then uses a predictive model for determining the parameters. However, PTF can produce more accurate results because it tests every parameter combination on the application itself.

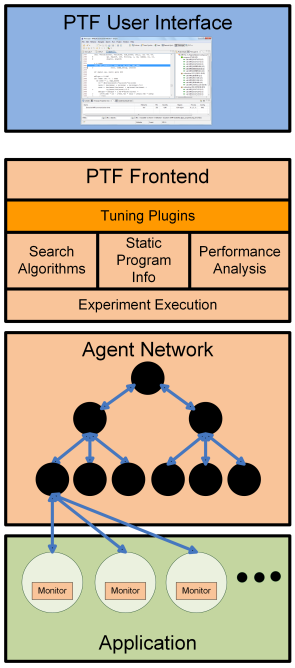

Figure 1: Architecture of the Periscope Tuning Framework.

3. PTF

The Periscope Tuning Framework (PTF) [1] consists of Periscope and the tuning plugins developed in the AutoTune project. It supports tuning applications at design time. The most important novelty of PTF is the close integration of performance analysis and tuning. It enables the plugins to gather detailed performance information during the evaluation of tuning scenarios, to shrink the search space, and to increase the efficiency of the tuning plugins.

The overall architecture of PTF is shown in Figure 1. It consists of the user interface, frontend, analysis agent hierarchy, and monitor that is linked to the application. The user interface allows to inspect performance properties in Eclipse, the frontend triggers performance analysis strategies that are executed by the analysis agents, and the MRI monitor (Monitoring Request Interface) measures performance data required for the automatic identification of performance properties.

Periscope's performance analysis determines information about the execution of an application in the form of *performance properties*. A performance property (e.g., load imbalance, communication, cache misses, redundant computations, etc.) characterizes a specific performance behavior of a program and can be checked by a condition. Conditions are associated with a confidence value (between 0 and 1) indicating the degree of confidence about the existence of a performance property. In addition, for every performance property a severity value is provided that specifies the im-

portance of the property. The higher the severity, the more important or severe a performance property is.

The frontend triggers *performance analysis strategies*, e.g., investigating certain performance properties related to specific programming models such as MPI.

The real analysis is performed by the analysis agents, the leaf nodes of the agent hierarchy. Each agent is responsible for a subset of the MPI processes. It configures the MRI monitor to measure the required performance metrics during an experiment. The experiment is then performed and the measurements are retrieved from the monitor. The analysis agent determines the properties and propagates them through the agent hierarchy to the frontend. PTF supports several analysis strategies, such as single core analysis, MPI analysis, OpenMP analysis, and configurable analysis.

PTF goes beyond automatic performance analysis and allows to automatically tune applications with respect to various aspects. It provides a rich toolbox for implementing *tuning plugins*. They follow a predefined *tuning model* that defines the sequence of operations that all plugins have to implement. The operations are defined by the *Tuning Plugin Interface (TPI)*. Plugins are loaded dynamically and can be provided in source or binary form. The sequence of TPI operations is determined by the frontend. It calls the plugin operations and implements the interface between the plugins and the rest of Periscope.

The plugins typically investigate a number of variants called scenarios to identify optimizations. In this process, analysis information is used to shrink the search space and to improve the search efficiency of plugins.

4. AUTOTUNING OF MPI APPLICATIONS

We have developed an MPI Parameter plugin that aims at automatically optimizing the values of a user selected subset of MPI parameters. Users indicate MPI parameters to be tuned and a range of values to be explored. In addition, users indicate the preferred search strategy (exhaustive, evolutionary) and if an automatic strategy should be used for the eager limit parameter. Then, the plugin generates the search space that is the crossproduct of all possible combinations of values of the parameters and selects scenarios to be evaluated based on the search strategy. Finally, the best results are provided to the users as advice.

Given the amount of MPI parameters that can influence performance and their dependence on the library implementation, it is difficult to find general models to guide the search, although, it can be possible for a reduced set of specific parameters. Consequently, we have decided to give users the possibility of using evolutionary algorithms to heuristically guide the search executing a reasonable number of experiments. In particular, genetic algorithms can be defined as a search heuristic used in optimization and search problems inspired by natural evolution mechanisms such as inheritance, selection, crossover, and mutation.

In addition, for the case of the eager limit parameter in combination with the memory buffer one, we have developed a special analysis strategy that further shrinks the search space for this tuning plugin. These parameters are specially relevant because of their potential influence on the application performance.

4.1 Heuristic Search

Suppose that a user wants to configure the following 5 parameters for an application using IBM MPI on SuperMUC [9]: `eager_limit` (from a few bytes to 64 KB), `buffer_mem` (from 4 KB to 2 GB), `pe_affinity` (yes,no), `task_affinity` (core, cpu), and `polling_interval` (from 1 to 2 billion microseconds). The number of scenarios that will be included in the search space is 480000 (15x80x2x2x100), which is unfeasible to explore exhaustively in a reasonable time, even if the application execution takes only a few seconds.

A promising way of traversing the search space obtaining reasonably good results consists of using heuristic search algorithms, such as genetic or incremental ones. PTF has been enriched with the implementation of different search strategies: the *Generalized Differential Evolution 3 (GDE3)* genetic algorithm [10], individual based on testing parameters incrementally, and probabilistic random search.

In the GDE3 strategy, a population of 10 initial scenarios is randomly generated and executed, then, an iterative process is followed generating new populations by selecting the best five scenarios (those with the smallest execution time) for the next generation (elitism), generating five new scenarios by crossing over the previous population, and introducing mutations with a fixed probability. The number of iterations can be configured, but, generally, a close to optimal solution can be found in less than 30 iterations (generations). In the example presented above, this would mean executing 300 experiments instead of 480,000, which can take a few hours or maybe some days depending on the application execution time. However, this can be an affordable analysis and tuning time for many applications.

The individual strategy iterates through the list of parameters and incrementally adds a new parameter to the already explored set of parameters. Consequently, in each search step it explores the effect of one new tuning parameter in combination with the already processed ones. In the example, this strategy will lead to the execution of only 199 experiments (15+80+2+2+100).

Finally, the random strategy samples the search space for a pre-configured number of scenarios using a parametrizable probability distribution (uniform by default).

4.2 Eager Limit Strategy

Specific analysis strategies can be developed for certain parameters in order to reduce the search space. In particular, we have developed a special analysis strategy for the case of the *eager limit* parameter in combination with the *memory buffer* one. These parameters have been chosen because of their potential influence on the applications performance.

The eager limit parameter allows users to establish the maximum size of messages that will be sent using the eager protocol. This parameter is usually limited by an upper bound by MPI implementations. For example, in the case of the IBM MPI version installed in SuperMUC the maximum is 64 KB, and it can range from a few bytes up to this limit. Because there are many possible values for this parameter, evaluating every value exhaustively can generate many scenarios in the plugin search space.

The eager limit parameter can affect the performance of point to point communications significantly. This type of communication is affected by the actual protocol used in the communication. Sending a message eagerly means that the sender is sure that the receiver has enough buffer space for storing the message, so it simply sends the message, avoiding the hand shake costs of other protocols, such as the

rendezvous protocol. Using the eager protocol may reduce communication time up to 60%, depending on the message size. However, the eager protocol introduces also some disadvantages, such as the necessity of bigger memory buffers, which can negatively affect the application performance, and the potential under-utilization of these buffers if the application traffic consists mostly of messages larger than the set limit.

Because of the performance impact of this parameter, it is worthwhile to define a specific performance property that is related to the optimal values (application dependent) for this parameter. This property allows for a significant reduction of the plugin search space and, as a consequence, the overall tuning time.

To detect this property for MPI applications, 8 new metrics were added to the framework:

- `PSC_MPI_MSG_P2P_THR`: this metric contains the total number of bytes transferred near the eager limit (currently between 1 KB and 64 KB).
- `PSC_MPI_MSG_P2P_TOT`: total number of bytes transferred using the MPI point to point operations.
- `PSC_MPI_MSG_P2P_<2K-64K>`: total number of messages (count) at certain size ranges. The first one contains messages up to 2 KB, while the rest are the counts of messages greater than the previous slot and under the KB value in the metric's name (for example, the 32K metric contains the message count of transfers between 16 KB + 1 and 32 KB).

With those metrics the performance analysis strategies provided by PTF detect the new property called `EagerLimitDependency`. When found, it means that the generated point to point traffic is sensible to alterations of the eager limit. In that sense, the performance of the application is dependent on the eager limit setting.

The severity of the `EagerLimitDependency` property is computed based on the fraction of the total MPI point to point traffic that took place near valid eager limit settings. That is simply the division of the `PSC_MPI_MSG_P2P_THR` metric over the `PSC_MPI_MSG_P2P_TOT` metric. The rest of the metrics are embedded in the extra information fields of the new property and can be used by the plugin to detect where exactly the traffic occurred; this extra information is then used to clip the search space and significantly accelerate the search in the dimension of this parameter.

We have developed a specific tuning model for the case of the eager limit parameter in combination with the memory buffer one. The plugin calls a pre-analysis using a configurable analysis strategy to obtain `EagerLimitDependency` property and, with the information provided, it first decides if it is worthy to tune the eager limit parameter. Consequently, the parameter will be included in the tuning space only if the proportion of messages in the valid range of the eager limit is more than 30% of the total number of messages sent by the application. We have considered that for a lower proportion the potential performance improvements would be too small.

Next, if it is worthy to tune the eager limit, the plugin analyzes the number of messages in each range and sets its search space in the limits of the range with the bigger number of messages. For example, Figure 2 shows that the range including more messages is from 4 KB to 16 KB, the plugin will generate a search space from 4 KB to 16 KB with a step of 1 KB. In addition, the plugin will use Expression 1 to calculate the appropriated search space for the memory buffer parameter. In this expression `n` is the number of processes of the application and the eager limit is expressed in KB, so for our example using 1000 processes it would produce a search space for the memory buffer going from 8000 KB to 32000 KB with a step of 1000 KB.

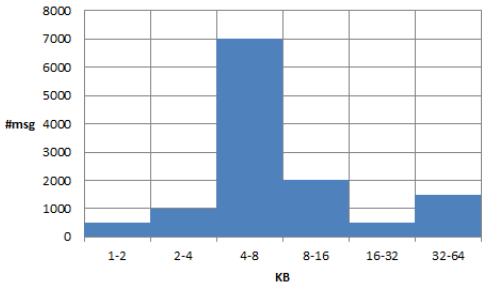

Figure 2: Hypothetical output produced by the `EagerLimitDependency` property.

$$mem\_buff = 2n * max(eager\_limit, 64) \quad (1)$$

5. MPI PARAMETERS PLUGIN

Based on the presented model, we developed the MPI Parameter plugin and integrated it into PTF. The integration with PTF provides the plugin with on-line measurements in the form of high level properties, allowing it to make tuning decisions based on the actual performance of the application. The plugin generates the scenarios to represent specific MPI configurations in the form of tuples of parameter-value pairs (i.e., specific combinations of values for the selected subset of MPI parameters). These scenarios are executed via PTF and evaluated using the resulting properties.

Before the execution of the experiments, the application must be prepared for tuning. In this case, the user should create the configuration file specifying the configuration options for the MPI library and a range of valid values for each of them. In addition, the used MPI implementation is specified, so the plugin will configure it accordingly.

5.1 Configuration File

In the MPI Parameters plugin there is one configuration file where all the parameters for the tuning process can be defined. The plugin uses its own parser and syntax for this file. Users create the configuration file specifying the MPI library parameters to be tuned and a range of valid values for each of them. Depending on the parameter, the valid values may vary; some of them require a Boolean value (e.g., `pe_affinity` (yes,no)), while others need a string of characters (e.g., `task_affinity` (core, cpu)), or a range of integers (e.g., `eager_limit` (from a few bytes to 64 KB), `polling_interval` (from 1 to 2 billion microseconds)). In addition, users can specify the kind of search strategy the plugin should apply, choosing between exhaustive, genetic strategy, individual, or random strategies (`SEARCH= exhaustive`, `gde3`, `individual` or `random`).

5.2 Complete Tuning Flow

Figure 3 shows the workflow of the MPI Parameters plugin. First, a set of MPI parameters and their possible values is obtained from a configuration file and scenarios are created for each possible combination of parameter-value pairs. The plugin then starts to experiment with the scenarios selected by the search strategy and evaluates each of them using an objective function. The plugin finishes when all the selected scenarios are explored or a time limit is reached. Finally, the scenario with the best performance (the combination of values that has given the lowest execution time) is used as a recommendation to the user. This advice can be applied by assigning the values to the corresponding environment variables, for example `set MP_EAGER_LIMIT = 16384` or by passing the value as an option in the `mpirun` command, for example `-eager_limit 16384` for IBM MPI.

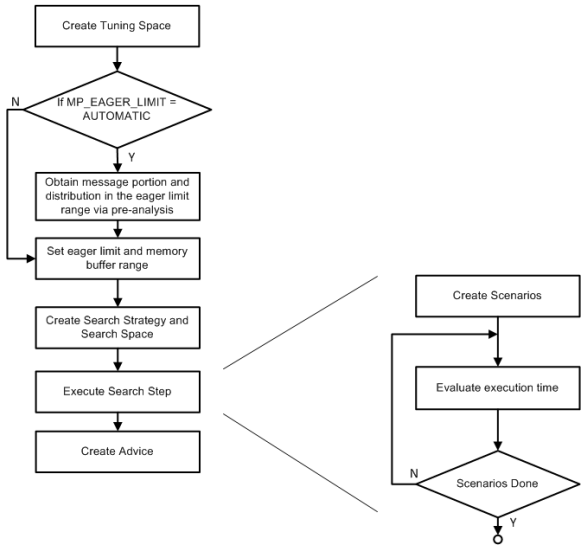

Figure 3: MPI Parameters plugin flowchart.

6. EVALUATION

To perform the experimentation on the plugin, we selected 10 IBM MPI parameters that can significantly influence the application's performance. They are the following:

- `eager_limit`: The eager limit is a threshold that controls which messages are sent using the eager protocol instead of the slower rendezvous one. It can be set between 0 and 64 KB (default 64 KB).

- `buffer_mem`: This parameter is closely related to the eager limit, given that messages sent eagerly must be stored in buffers, whichs size is indicated by this parameter. Consequently, increasing the `eager_limit` and `buffer_mem` could reduce communication time at the cost of reserving more memory for the MPI library, which could negatively affect application performance. The maximum value of this parameter is 64 MB (it is set accordingly to the number of application processes and the `eager_limit` value).

- `use_bulk_xfer`: If set to yes (the default), this transparently causes portions of the user's virtual address space to be pinned and mapped to a communications adapter. This causes the low level communication protocol to use Remote Direct Memory Access to copy data from the send buffer to the receive buffer as part of the MPI receive.

- `bulk_min_msg_size`: Contiguous messages with data lengths greater than or equal to the value of this parameter will use the bulk transfer, while messages that are smaller than the value, or are non-contiguous, will use packet mode transfer. This parameter has effect only when `use_bulk_xfer` is set to yes. Again, there is a tradeoff between use of memory and faster data transfers. Its values can be in the range 4 KB to 2 GB (default 64 KB).

- `task_affinity`: Tasks of a parallel application can be allocated to a single core (CORE - the default), several cores (CORE:n), a whole processor (CPU), several processors (CPU:n), or over the processors of a node in a round-robin fashion (MCM). This parameter is useful for assigning resources for hybrid applications and for managing which tasks are closer among them or the amount of memory assigned to each task.

- `pe_affinity`: Determines whether Load Leveler or the MPI environment determine the task scheduling. In the latter case, which is the default, MPI will use the value specified by the `task_affinity` parameter.

- `cc_scratch_buf`: If set to yes, this parameter uses the fastest collective communication algorithm even if it requires the allocation of more scratch buffer space (default yes).

- `wait_mode`: Used to specify how a thread or task behaves when it discovers it is blocked, waiting for a message to arrive. It can be set to poll (default) or nopoll.

- `css_interrupt`: Used to specify whether or not arriving packets generate interrupts. It is recommended to set this parameter to yes when `wait_mode` is set to nopoll (default no).

- `polling_interval`: Used to specify a fixed period of time to interrupt the nopoll wait (wait mode set to nopoll and css interrupt set to no). Its values can be in the range 1 to 2 billion micro seconds (default 400000).

In addition, we also selected a set of Intel MPI parameters covering approximately the same features described before in order to demonstrate the functionality of the plugin over different MPI implementations.

6.1 Settings

To test the plugin we have used the FSSIM application [15], which is a biological simulator that models the movement of large fish schools. It uses individual oriented simulation to recreate the patterns in which actual fish schools move. FSSIM uses a distributed cluster-based approach to parallelize the simulation and follows an SPMD model to divide the workload among the processes. We ran PTF with the MPI Parameters plugin on the FSSIM application, using a medium workload of 64K individuals and 64 threads and a big workload of 256K individuals and 256 threads.

In addition, we have also tested the plugin with the NAS Parallel Benchmarks (NPB 3.2) executing class C of Conjugate Gradiant (CG) and Integer Sort (IS) on 16 cores.

Depending on the number of parameters to be tuned and the range of values they can take, the potential size of the search space can be huge. As it is not feasible to test all parameters combination exhaustively, we have used the exhaustive search strategy for a very limited configuration of the parameters and the heuristic search strategies for a more complete configuration.

We have prepared the following configuration files for different searches, where (`SEARCH = gde3`, `individual`, or `random`, by default `exhaustive`) :

- exhaustive search on IBM MPI

```
MPI_PIPO BEGIN ibm
eager_limit=1024:8192:65536;
use_bulk_xfer=yes,no;
bulk_min_msg_size=4096:32768:1048576;
cc_scratch_buf=yes,no;
MPI_PIPO END
```

- heuristic search on IBM MPI

```
MPI_PIPO BEGIN ibm
SEARCH=gde3;
eager_limit=1024:1024:65536;
buffer_mem=131072:131072:8388608;
use_bulk_xfer=yes,no;
bulk_min_msg_size=4096:4096:1048576;
task_affinity=CORE,MCM;
pe_affinity=yes,no;
cc_scratch_buf=yes,no;
wait_mode=nopoll,poll;
css_interrupt=yes,no;
polling_interval=100000:10000:1000000;
MPI_PIPO END
```

6.2 Automatic tuning results

The exhaustive search strategy, tested only for the medium size case of FSSIM, generated all possible combinations of the selected parameters and values, 1024 for IBM MPI in our example. Each scenario was executed and the one with the smallest wall time was chosen as the best one. PTF produced the results shown in Figure 4. The worst execution times are obtained for small values of the `eager_limit` ($<$ 17KB) in combination with the `use_bulk_xfer` set to yes.

The best combination corresponds to scenario 940, with an execution time of 2.77 sec, which is more than 1.5 times better than the time for the execution using the parameters' default values (4.32 sec). The drawback is that for finding the optimum scenario in the set of tested ones, PTF needed 26819 sec (almost 7.5 hours). It is worth noticing that many scenarios lead to similarly good solutions (around 2.8 sec). Consequently, we expect that an heuristic search will find one of these solutions significantly reducing the search time.

Both cases of FSSIM (64K and 256K individuals) have been executed using the three heuristics offered by PTF, namely GDE3 (genetic), individual and random.

Figure 7 shows the results for the 6 combinations and Table 1 compares the execution time of the application with the default parameters and the best cases found by each heuristic. In all cases, a good combination of parameters is

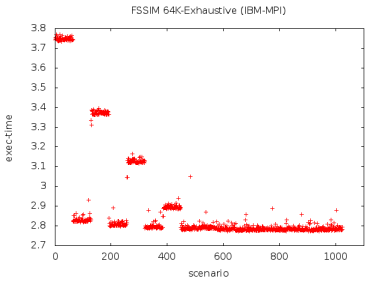

Figure 4: Execution of FSSIM using the MPI Parameters plugin with exhaustive search and IBM MPI.

found, obtaining a performance improvement of 1.5 and 2.1 for the 64K and 256K individuals simulations respectively.

However, the reasons that led to these improvements were different in each case. For the 64K individuals simulation the application requires less memory and sends smaller messages than in the 256K individuals case. Consequently, the 64K individuals simulation is more sensitive to the values of the `eager_limit` and `buffer_mem` parameters. Actually, the three search algorithms found solutions where the `eager_limit` was set to 40 KB approximately and the `buffer_mem` was big enough to ensure that each process would be able to send up to 3 messages eagerly at a time to any other process. The worst execution times shown in the figure correspond to small values of the `eager_limit` parameter.

This conclusion is reinforced when using the automatic eager limit strategy explained in Section 4.2. In this case, the plugin determines that more of 90% of the bytes communicated by the application were sent in messages of less than 64 KB, which means that the eager limit is a very significant parameter for the application performance.

In this case, the genetic search strategy (GDE3) executed 420 different scenarios (41 generations) in 11025 sec (approximately 3 hours), which reduced the analysis time by a factor of 2.5 with respect to the exhaustive search. The individual strategy executed 119 scenarios in 3127 sec (approximately 52 min), which reduced the analysis time by a factor of 3.5 with respect to the GDE3 strategy. Finally, the random strategy executed 100 scenarios in 2642 sec (approximately 44 min), reducing the analysis time an extra 15%.

In addition, the FSSIM 64K individuals simulation was also tuned for Intel MPI. Figure 6 shows the results for this experiment and Table 1 includes the execution time for the best parameter configuration found using GDE3. The results are similar to the previous ones, in particular, the best execution times are obtained for a `intranode_eager_threshold` value of 40KB approximately. It is worth noticing that in this case GDE3 only needed to execute 150 scenarios (14 generations) because the results convergence was faster than for IBM MPI.

On the other hand, for the 256K individuals simulation the analysis results indicate that the `eager_limit` must be bigger as the messages are also bigger, the buffer_mem should be smaller because the application requires more memory for the simulation, and the application is sensitive to the value of task_affinity (pinning). Actually, all search strategies found solutions where the `eager_limit` was set to more than 60 KB and, in addition, the worst execution times are obtained

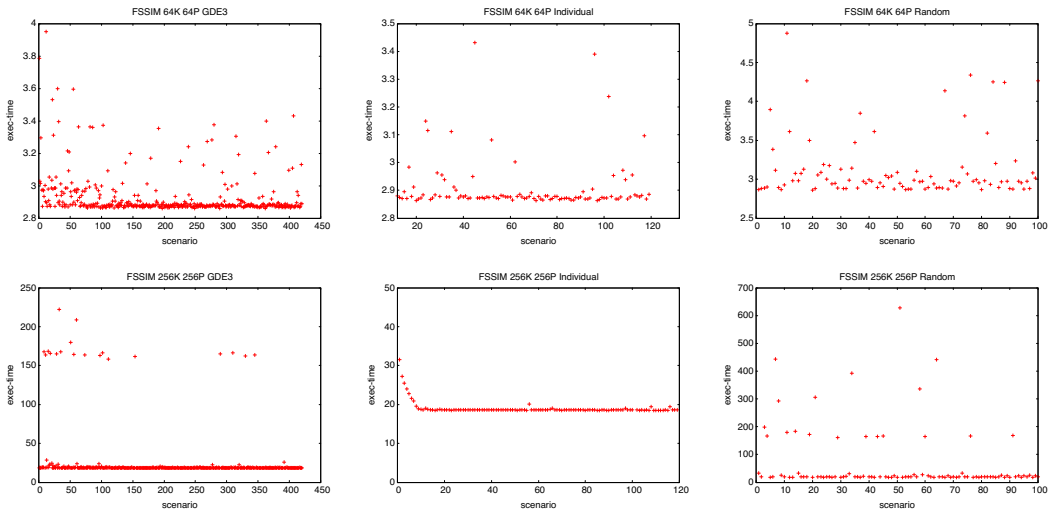

Figure 5: Execution of FSSIM using the MPI Parameters plugin with GDE3, individual and random search and IBM MPI for 64 K and 256 K individuals.

for the round robin assignment of tasks (MCM) and small values of the `eager_limit` parameter.

	Exec. time 64K	Exec. time 256K
default params	4.32	39.74
GDE3	2.86	18.5
individual	2.86	18.48
random	2.86	18.48
Intel GDE3	3.03	-

Table 1: Execution time of FSSIM using default values and the best cases found by each heuristic.

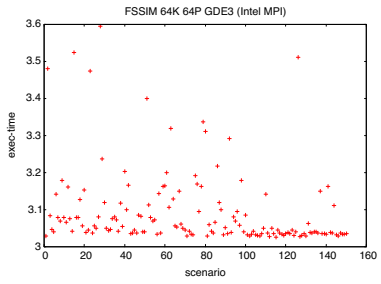

Figure 6: Execution of FSSIM using the MPI Parameters plugin with GDE3 search and Intel MPI.

Finally, for demonstrating the effectivity of the plugin on other applications, it was also tested using the NAS benchmarks CG and IS on 16 cores, IBM MPI and random search. Figure 7 shows the obtained results. The plugin determined that the default values of the parameters included in the configuration file led to the best execution time in the case of CG, which is not surprising because usually the MPI parameters default values are determined using these benchmarks (among others). However, the plugin determined that significantly reducing the value of the `eager_limit` led to an improvement of more than 10% in the case of IS.

7. CONCLUSION

PTF is a framework that allows easy development of tuning plugins. This plugins explore search space and find the best variants of tuning parameters based on performance information resulting from PTF's analysis strategies. Different variants are explored by running experiments that return measurements for each variant. The main advantage of PTF comes from combined performance analysis and tuning. By applying performance analysis, PTF can shrink the search space by reducing the amount of tuned regions and by reducing a range of values for a tuning parameter. This paper presents the MPI Parameters plugin, which automatically optimizes the values of a user selected subset of MPI configuration parameters. The plugin uses different strategies for guiding the search depending on the search space size and user's specification. If the search space generated by the crossproduct of the specified parameters is small, the plugin may perform an exhaustive search to find the best combination of values. If not, users can indicate an heuristic search strategy (GDE3, individual or random) to guide the search executing a reasonable number of experiments. In addition, a specific tuning strategy has been developed for the `eager_limit` parameter because of the impact of this parameter on the application performance. This strategy analyzes the sizes of the messages interchanged by the application to determine if it is worthy to tune this parameter, and focus the search in a reduced set of meaningful values.

Acknowledgments

The research leading to these results has received funding from the EU FP7 project AutoTune, no. 288038, and the EU Horizon 2020 project RADEX no. 671657. This work has also been partially supported by MINECO-Spain under contract TIN2014-53234-C2-1-R and GenCat-DIUiE(GRR) 2014-SGR-576.

8. REFERENCES

[1] Autotune. Periscope Tuning Framework, March, 2016. http://periscope.in.tum.de.

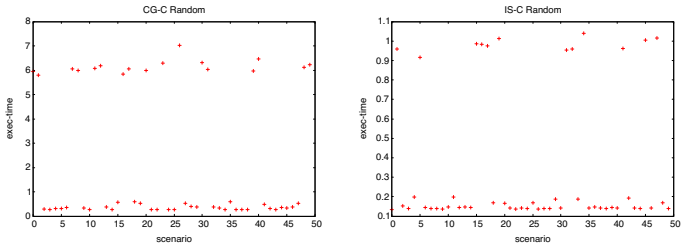

Figure 7: Execution of the CG and IS benchmarks using MPI Parameters plugin with random search and IBM MPI.

[2] S. Benedict, V. Petkov, and M. Gerndt. PERISCOPE: An Online-Based Distributed Performance Analysis Tool. In M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 1–16. Springer Berlin Heidelberg, 2010. 10.1007/978-3-642-11261-4-1.

[3] P. Caymes-Scutari, A. Morajko, T. Margalef, and E. Luque. Scalable dynamic Monitoring, Analysis and Tuning Environment for parallel applications. *J. Parallel Distrib. Comput.*, 70(4):330–337, 2010.

[4] M. Chaarawi, J. M. Squyres, E. Gabriel, and S. Feki. A Tool for Optimizing Runtime Parameters of Open MPI. In A. L. Lastovetsky, M. T. Kechadi, and J. Dongarra, editors, *PVM/MPI*, volume 5205 of *Lecture Notes in Computer Science*, pages 210–217. Springer, 2008.

[5] I.-H. Chung and J. K. Hollingsworth. Using Information from Prior Runs to Improve Automated Tuning Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 30–, Washington, DC, USA, 2004. IEEE Computer Society.

[6] I. Corporation. Intel®MPI Library. Reference Manual for Linux* OS, 2003-2014.

[7] G. Fursin, Y. Kashnikov, A. Memon, Z. Chamski, O. Temam, M. Namolaru, E. Yom-Tov, B. Mendelson, A. Zaks, E. Courtois, F. Bodin, P. Barnard, E. Ashton, E. Bonilla, J. Thomson, C. Williams, and M. O'Boyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International journal of parallel programming*, 39(3):296–327, 2011.

[8] M. Haneda, P. Knijnenburg, and H. Wijshoff. Automatic selection of compiler options using non-parametric inferential statistics. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 123–132, Sept 2005.

[9] IBM MPI: An MPI implementation for SuperMUC https://www.lrz.de/services/software/parallel/mpi/ibmmpi/.

[10] S. Kukkonen and J. Lampinen. GDE3: The third evolution step of generalized differential evolution. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 1, pages 443–450. IEEE, 2005.

[11] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin, and F. Bodin. AutoTune: A Plugin-Driven Approach to the Automatic Tuning of Parallel Applications. In P. Manninen and P. Öster, editors, *Applied Parallel and Scientific Computing*, volume 7782 of *Lecture Notes in Computer Science*, pages 328–342. Springer Berlin Heidelberg, 2013.

[12] Z. Pan and R. Eigenmann. Fast and Effective Orchestration of Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 319–332, Washington, DC, USA, 2006. IEEE Computer Society.

[13] S. Pellegrini, J. Wang, T. Fahringer, and H. Moritsch. Optimizing MPI Runtime Parameter Settings by Using Machine Learning. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting, Espoo, Finland, September 7-10, 2009. Proceedings*, pages 196–206, 2009.

[14] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. Padua, M. Veloso, and R. W. Johnson. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, Feb. 2004.

[15] R. Solar, R. Suppi, and E. Luque. High performance distributed cluster-based individual-oriented fish school simulation. In *ICCS*, pages 76–85, 2011.

[16] A. Tiwari, C. Chen, J. Chame, M. Hall, and J. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12, May 2009.

[17] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August. Compiler Optimization-space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society.

[18] R. Vuduc, J. W. Demmel, and K. A. Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521, 2005.

[19] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3 – 35, 2001. New Trends in High Performance Computing.