Extending PTF's MPI Parameters plugin

Alberto Olmo Hernández

Resumen— Periscope Tuning Framework es una herramienta de sintonización automática que monitoriza aplicaciones reales ejecutadas en supercomputadores con el fin de optimizar alguno de los aspectos que influyan en su rendimiento. Dicha herramienta está compuesta por varios plugins que la dotan de funcionalidad y permiten que pueda ser utilizada en diversas aplicaciones con tipos de optimizaciones diferentes. Lo que se pretende con este trabajo es mejorar e implementar nuevos aspectos dentro del plugin de sintonización de parámetros de la librería MPI o MPI Parameters Plugin.

Palabras clave— MPI, optimización de parámetros, Periscope Tuning Framework, plugin MPI Parameters

Abstract— Periscope Tuning Framework is an automatic tuning tool that monitors real applications executed in supercomputers with the aim of optimizing an specific aspect that influence their performance. This tool is composed by several plugins that make it functional and also allow it to be used by various and different types of programs and with distinctive types of optimizations. The main purpose of this work is to improve and implement new aspects inside the plugin of MPI tuning: MPI Parameters Plugin.

Index Terms— MPI, parameter optimization, Periscope Tuning Framework, MPI Parameters plugin

1 Introduction

Nowadays we can find computer-based applications capable of solving problems that need large quantities of mathematical calculations. Therefore, large computational resources are needed for executing these applications in a reasonable time. In order to obtain acceptable execution times, it is necessary to optimize those applications' performance parameters so as to obtain the maximum efficiency from the platform. Periscope Tuning Framework (PTF) **[1]** is a tool that automatizes this process, offering to the users the combination of parameters that maximizes the application performance. The set of parameters as well as the objective function to optimize are provided by the user (for instance, through a configuration file). PTF includes a structure based on plugins that guides the creation and execution of multiple scenarios to obtain the best combination of parameters. Each one of this scenarios implies the complete or partial execution of the application.

At present, this tool includes the plugins Compiler Flags Selection, DVFS so as to tune those parameters that involve the power consumption, the Master-Worker for applications of this kind, the Parallel Pattern to tune applications that combine traditional multicore with GPUs, and finally, MPI Parameters. This last one is responsible for the optimization of values for a set of parameters related with the MPI library indicated by the user. Its extension and improvement is the main objective of this work.

The very first version of MPI Parameters only allowed distinguishing which was the implementation of MPI that was going to be used: IBM MPI, OpenMPI, Intel MPI or others. However, the problem of this version was that in each of these cases, although it was able to recognize the MPI type, it wasn't capable of differentiate whether the introduced parameter was a parameter from the stated implementation or not as well as its associated values and their correctness.

With this work, it has been achieved an initial analysis of the inputs for the IBM MPI version recognizing the parameters entered and validating them with their associated values.

In addition, as of this analysis, it is possible to detect redundant combinations of parameters. Consequently, the functionality of the plugin has been modified so as to reduce the number of cases to execute and, therefore, cut down on the necessary time to tune the application.

Finally, it should be pointed out that, based on the obtained results by means of tests with different MPI applications, we can affirm that the interface has been enhanced as well as the analysis performance of the MPI Parameters plugin regarding its initial version.

The organization of this article is distributed in the following form: section 2 describes the context in which this project has been developed. Section 3 explains the objectives of this work. Next, section 5 introduces the methodology used. Section 6 describes the different phases through which the project had to pass. Straightaway, section 7 shows the obtained results and, finally, conclusions are detailed in section 8.

2 Context

In order to contextualize this project, some remarkable details for the two most important elements of this work are going to be discussed: Persicope Tuning Framework and how it uses its plugin-based structure.

- *E-mail de contacte: aolmo.uab@gmail.com*
- *Menció realitzada: Ingeniería de Computadores*
- *Treball tutoritzat per: Eduardo César Galobardes (Arquitectura de Computadores y Sistemas Operativos)*
- *Curso 2015 - 2016*

2.1 AutoTune project and Periscope Tuning Framework

The European project AutoTune **[2]** had as main objective to develop an automatic tuning tool for parallel applications. In order to do so, the Persicope **[3]** tool was extended and enhanced and called Periscope Tuning Framework or PTF for short.

That project was coordinated by the *Tunische Universität München* and had the collaboration of the following partners: *CAPS entreprises, University of Vienna, Liebniz Computing Centre, University of Galway* and *Universitat Autònoma de Barcelona.*

PTF is a tool whose aim is to tune a set of parameters or values for parallel applications in an automatic and scalable way for High Perfomance Computing frameworks. PTF is made up by a frontend and a hierarchy of agents of communication and data analysis. This frontend is in charge of starting the application and the agents network. Additionally, it analyzes the set of available processors distributing the application processes amongst the agents hierarchy in order to make them look for inefficiencies in the group of assigned processes.

All processes from the application are bind together with a monitoring system provided by the called Monitoring Request Interface or MRI. The MRI allows the agent nodes to configure the measurements; collect the performance data; and start, destroy or continue with the application execution. Thus, PTF is unique due to the fact that combines the analysis and the tuning of multiple aspects inside an automatic tuning structure allowing the user to:

- Identify the variables to tune,
- Evaluate these variables by reducing the total search time for the tuned versions.
- Be able to cover an extensive and extensible range of tuning aspects thanks to its plugin-based structure.
- Recommend the users on how to improve their code being able to do it manually or automatically.

2.2 The PTF plugins

With the purpose of being able to cover the largest number of possible applications and implementations either MPI, OpenMP or others, Periscope Tuning Framework consists of a system based on plugins that provides it of the necessary functions so as to deal with these different implementations. In order to make a plugin for PTF, the tool supplies the developer with an interface called IPlugin that contains all the methods that will be called by PTF and that must be implemented based on the necessities of the plugin.

Initially, each one of these plugins reads from a file created by the user where the desired parameters and values can be specified. The introduced configuration directly depends on the plugin type and must contain a an initial sentence so as to point out the parser where the introduction of parameters is starting, and another ending sentence to indicate the final of it.

Due to the fact that it can be a large amount of different executable combinations, PTF provides the users with several algorithms based on heuristics to reduce the search space. Amongst these, the *individual* and the *GDE3* should be outlined. In case of not specifying any of them in the configuration file, the selected one by default would be the *exhaustive*, which executes all possible combinations of parameters. *Figure 1* shows the PTF's state diagram with all the methods that are called in each moment and must be provided and implemented by the plugin's programmer.

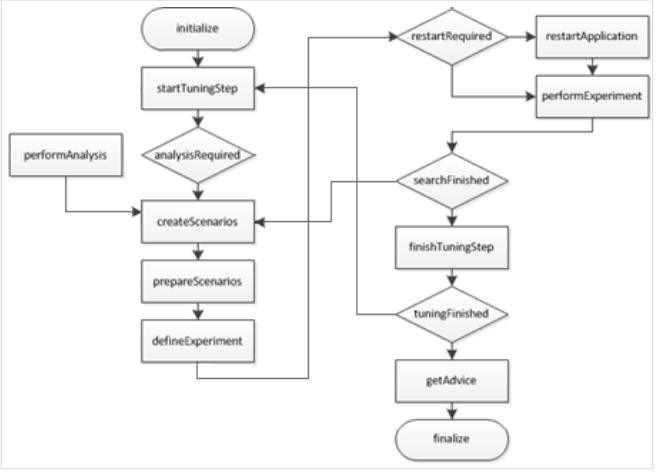


Figure 1

In order to adapt the last MPI Parameters plugin version to the objectives of this project, we have introduced significant modifications in the following methods *from figure 1:*

- initialize(): In this function, the configuration files are read and all basic variables and structures are initialized.
- createScenarios(): In this method the scenarios are created based on the specified parameters by the user. In addition the search space is also created.
- prepareScenarios(): This method chooses an element (variant) of the search space made in the last function depending on the search algorithm used. Basically, it creates the set of *parameter* = *value* that will be passed to the application to generate the scenario that will be executed next.
- restartRequired(): Through this function, we can indicate PTF whether it is necessary to restart the application or not.

2.3 The MPI Parameters plugin

MPI or Message Passing Interface **[4]** is the *de facto* standard for parallel applications based on message passing. The different implementations of the MPI specification include a significant number of parameters that can affect the performance of the communications in the application. Consequently, the optimization of these parameters can have a significant impact in the applications based on this system. Nonetheless, the large number of parameters (between some tens and more than 100 depending on the implementation), as well as the wide range of values that some parameters can adopt, make

the manual tuning process tedious and extensive.

The aim of the MPI Parameters plugin is the optimization of the set of values introduced by the user by using the configuration file. The integration with PTF will provide the plugin with real time information that is used to make necessary tuning decisions to optimize the efficiency of the application. Nevertheless, in spite of performing it in an automatic way, the number of parameters and the set of values that each one of them can have, make the search space rather hard to deal with.

Because of this, additionally of being able to use the search strategies provided by PTF, MPI Parameters is capable of reducing the search space by its own plugin analysis. In this sense, the *auto_eager_limit* parameter was implemented. This, once specified in the configuration file, can internally reduce the search space for the *eager_limit* and *buffer_mem* parameters.

A possible configuration example for tuning an application that uses IBM MPI could be the following one:

```
MPIPO_BEGIN IBM
SEARCH = gde3;
use_bulk_xfer = yes, no;
bulk_min_msg_size = 1024:4096:1024;
wait_mode = nopoll, poll, sleep, yield;
css_interrupt = yes, no;
pe_affinity = yes, no;
task_affinity = core, cpu, mcm;
MPIPO_END
```

In this example, there appear some typical parameters from the Plugin. For instance, the SEARCH one allows indicating the search strategy to be used. Additionally a ranged parameter appears: *bulk_min_msg_size.* Finally, there also appears two other possible parameters *task_affinity* a and *pe_affinity* whose inputs are a list of values.

3 Objectives

The main objective of this project was to extend and improve the initial capacities of the MPI Parameters tuning plugin from the Periscope Tunig Framework tool. In particular, for the parameters of the IBM MPI implementation. It should be pointed out that in the original version of the plugin the user could introduce IBM MPI parameters in the configuration file already, however, their management appeared to have some important deficiencies. Firstly, some combination of values could not be introduced. Secondly, the syntactical analyser accepted any parameter name as valid. Finally, there wasn't a proper validation of the assigned values.

The aim of this project is to enhance the plugin efficiency thanks to the recognition of the parameters that the user can introduce and offering a better interface by using on-screen messages that would inform about the lexical and syntactical errors that could be made in the introduction of values and parameters.

Having said that, the first task to deal with was called *Study of the PTF environment*, which consisted of controlling that, in the case the user indicated that the IBM MPI implementation is being used, there can only appear parameters from this implementation and their related values in the configuration file. It was decided to use the lexicographical analyzers generator Flex and the parser generator Bison **[5].** These tools were chosen since the last version of the plugin, as well as the rest of the plugins from PTF, used these programs. Besides, creating a parser from scratch would have only slowed down the implementation of the main objectives of this work, even preventing them from being realized.

The use of Flex and Bison in the project propitiated that some resources had to be designated to the study of the programming languages grammar definition so as to be able to understand the basis that shape this field. In this way, it would be much easier to carry out the task of creating a lexical analyzer and a syntactical analyzer in the proper manner.

Once understood the operation of these by means of tutorials **[6][7]** contended in the task *Study of the definition of the grammar in programming languages,* the next task was called *Study of the programming structure for PTF plugins* and consisted of correctly designing the specification interface for the IBM MPI parameters taking into account that the new version of the syntactical analyzer ought to be compatible with the last one, so as to let the user introduce any other kind of parameters for the Intel MPI, OpenMPI or other MPI implementations.

In order to achieve this, some resources were needed to make an initial analysis of the MPI Parameters plugin internal operation (these tasks were called *MPI Parameters Plugin analysis* and *Interface design of the IBM MPI parameters specification*) and the structure that formed it, through the use of the documentation of Periscope Tuning Framework generated by Doxygen.

Finally, it was necessary to adapt each one of the obtained parameters from the parser to understandable variables for the plugin as a final objective (contained in the task called *Introduction of new functionalities to the MPI Parameters Plugin)*. With this new functionality it was achieved to treat one parameter by one and reject those redundant or not necessary. This also permitted enhancing the plugin's efficiency.

4 State of the art

There are other tools for the MPI parameter tuning process. This fact shows that the tuning of MPI parameters is important so as to improve the performance of an application based in this field. The two current and most related MPI tools to the MPI Parameters plugin idea are OTPO **[8]** and MPITune from Intel **[9].**

OTPO or Open Tool for Parameter Optimization is a tool designed to help in the optimization process of MCA (Modular Component Architecture) parameters from OpenMPI and whose purpose is to explore the effect of these parameters in different architectures and with different configurations. OTPO, similarly to the MPI Parameters plugin, will accept a list of user-specified parameters with their respective values. For each one of

the possible combinations, OTPO will execute an MPI task that will be under study by a benchmark (currently it only owns three different ones), to finally provide the best combination of the whole set based on the execution times. This tool is still under development and because of that the results offered to the user are not interpreted and visually well formatted. Therefore, the differences between OTPO and MPI Parameters are clear; OTPO only accepts tuning parameters from the OpenMPI library whereas MPI Parameters supports any kind of MPI implementation, furthermore, it gives the possibility to choose the search algorithm to the user so as to let them control the number of executions in an approximated way. Finally, MPI Parameters carries out the experiments by using the given application and not benchmarks.

MPITune from Intel is a parameter tuning tool that looks for the best combination of parameters based on the executed time. It contains the *–fast* option that lightens the process by using the result of previous executions of MPITune located in the output file. In the same way as OTPO, MPITune is limited since it only accepts configuration parameters from the Intel MPI type differently to MPI Parameters.

As it can be noticed, with the new improvements proposed in this project for MPI Parameters the distance between it and OTPO and MPITune increases since it demonstrates to be the most versatile option to choose.

5 METHODOLOGY

The initial methodology used for the tasks *Study of the PTF environment, Study of the definition of the grammar in programming languages, Study of the programming structure* and *MPI Parameters plugin analysis* consisted in gathering information by means of books, guides, tutorials and internet **[10][11][12]** with the purpose of acquiring the necessary experience so as to realize the following tasks.

Nonetheless, for the tasks *Interface design of the IBM MPI parameters specification* and *Introduction of the new functionalities to the MPI Parameters Plugin* the followed methodology was used:

1. Evaluation of the problem to solve and search for the possible solutions by using the mentioned references.
2. Search for the best solution and creation of the first prototype.
3. Making of one solution based on the prototype.
4. Testing of the final solution.

It should be pointed out that, as a part of the methodology, each time a change was implemented, it was also added to the version control file where the version was also included if the change was important enough.

6 DEVELOPMENT

In order to be able to understand how the current project was developed, it is necessary to mention that was divided in three phases carried out in order. The first one of this phases consisted in the implementation of the syntactical analyser of the configuration file. So as to achieve this, the lexicographical analyzer generator Flex and the parser generator Bison were required. Once this was done, the next step was to classify these data to make the plugin have them stored and later use them.

A *checksym* method was also implemented in order to be able to distinguish those repeated parameters and enhance the analysis and tuning performance. This checksum is used to identify each one of the sentences that are going to be added to the command line and rejects the repeated ones. For this, the function transforms each sentence in a numeric value and then stores it in only if it doesn't find the same value in the array so as not to run the same command more than once. Finally, the necessary methods were also modified to make use of this new interface as well as optimizing the efficiency of the plugin itself.

6.1 Lexicographical analyzer and syntactical analyzer

Flex is a tool that allows to create lexicographical analyzers from a set of rules in order to be able to read and classify the content of a file using symbols or tokens. This process depends on the set of criteria established by the user. An example of this in natural language is shown below:

"Pablo's house"
Personal name + genitive + common name

The sentence "Pablo's house" can be morphologically decomposed in: Personal name + genitive + common name. In order to let the lexicographical analyser determine each word's type and assign them a category (token), we ought to define the rules that make them distinguishable.

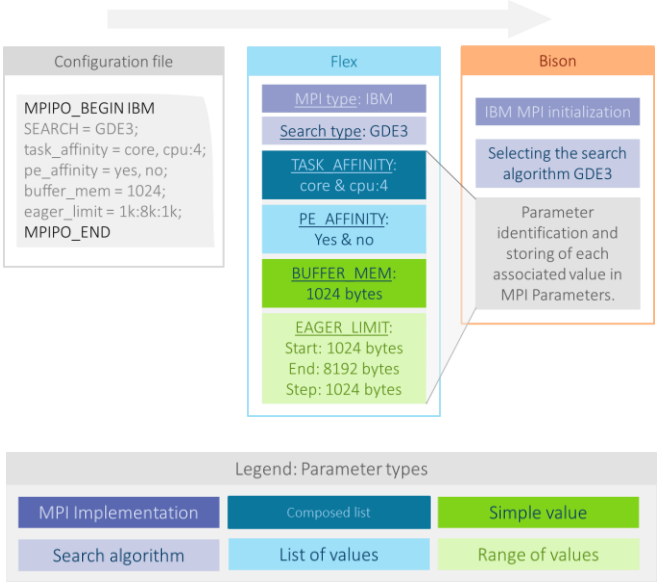
The current rules from the generated file with Flex of MPI Parameters, allow differentiating:

- MPI implementations: this is the chosen MPI type that will be used in the file and that can be either OpenMPI, Intel MPI, IBM MPI or any other implementation.[1]
- Parameters: these are the MPI implementation variables that we can use in order to tune the application.
- Values: the values are formed by integer numbers, character strings, as for instance *yes* or *no,* or by integer numbers and a prefix *K, M* or *G* to represent Kilo, Mega or Giga respectively. However, there is a conversion from this values to Bytes.
- Intervals: The intervals consist of a range of type *start : end : step,* where *start* and *end* contain the initial and final values of the interval. In addition, the *step* will represent the jump between each iteration. This value is set to 1 by default.
- Lists: These are a set of values separated by commas that will be processed in the given order.

[1] Nonetheless, this Project is centred in the treatment of the possible values only for the IBM MPI implementation.

- Composed lists: These are special lists where some elements can look like *string : value*. This case will only happen for the *task_affinity* parameter.
- End of line and syntactical errors in the parameters' introduction. If there appears this kind of error, the execution will stop.
- Start and end of the file: This is achieved by using the strings: *MPIPO_BEGIN* and *MPIPO_END*

Next figure shows the different phases through which the data flows from the moment they are written in the configuration file until Bison's generated analyzer recognizes the formed structures.



Initially, as it can be noticed, the user introduces the parameters that want to optimize by using the configuration file. Then, these parameters and their values are morphologically analyzed. As a requirement, these ought to be introduced between *MPIPO_BEGIN* and *MPIPO_END*. The name of the MPI implementation that will be used must appear after the *MPIPO_BEGIN* statement. This could be IBM, OpenMPI or Intel. If the user leaves this blank the plugin understands that the user is using another type of MPI implementation.

Once the MPI specification that will appear in the file is known, the lexicographical analyzer will be able to recognize the parameters that should appear from those that shouldn't. As an example, those that are incorrectly written would be discarded. This will also allow the plugin to know in advance which are the set of values that a parameter can accept and provide error messages to the user in case of not introducing a correct one. For instance, the parameter *pe_affinity* owns a value set of only *yes* and *no*, because of this, if we included '123', the execution would stop and show an error that would look like:

```
[ERROR]: value '123' not recognized on
                line 2
```

In case that the introduced parameters were valid, the lexicographical analyzer generated with Flex would form the token list which would be communicated to the grammatical analyzer. That is to say that, if we take as an example the parameter *task_affinity = core, cpu:4* firstly, the lexicographical analyzer will recognize it as a *composed list* and, secondly, it will transform it to the following formation of tokens:

MP_TASK_AFFINITY - CORE - COMMA - CPU - COLON - INT_VALUE

The very first element is the one that will denominate the parameter. The following tokens are the representation of the introduced string, now transformed into tokens that the syntactical analyzer will be able to understand. In case it is necessary to store a *string* or *integer* value, the two special tokens *STRING_VALUE* and *INT_VALUE* can be used, their behavior is similar to a variable whose value can be accessed. Finally the lexicographical analyzer will send to the grammatical one the lists of parsed tokens in order to recognize the grammatical structures that could appear and act accordingly.

6.2 Parsed data classification

Once all tokens are gathered, the grammatical analyzer will store them in the shape of values that the plugin will understand. In order to achieve this, PTF relies on an interface called IPlugin. This interface must be adapted to the necessities of each plugin and the methods from *figure 1* ought to be implemented.

In the initial version of MPI Parameters, all the parsed data was stored in order by using *strings* that, afterwards would form the command line to be executed due to the fact that no specific parameters were detected.

In the proposed solution of this work, it has been necessary to include:

- A new variable that stores the specific type of the parameter. For instance *pe_affinity, wait_mode* or *bulk_min_msg_size.* Each one of the parameters now accepted for the IBM MPI implementation are provided in the attached table in the appendix of this article.
- A new structure that will store the data from the *task_affinity* parameter and called *TaskAffinity*. This will contain Boolean variables whose function will be to indicate whether some of the possible values have been included or not, for instance, CPU or Core. Furthermore it will also keep the integer values associated to them. Additionally, an array that stores *TaskAffinity* elements has also been created. In order to correctly generate the command line, these elements will be extracted one by one and converted into strings in the *prepareScenarios()* function from the plugin.
- The functions that deal with the parameters and that had to be created are:

- *isAutoEagerLimit():* This function checks whether the user has introduced the *auto_eager_limit* parameter or not.
- *deleteEagerLimitAndBufferMem():* This method will delete the *eager_limit* and *buffer_mem* parameters if the user has introduced the *auto_eager_limit* parameter in the same configuration file.
- *deleteConfParameter():* This function allows the plugin to delete any specified parameter amongst those introduced in the configuration file.
- *transformTaskAffinityToStringValues():* This method handles the generation of a *string* vector filled with all the *TaskAffinity* values transformed into *strings.* Afterwards, this will allow the plugin to generate the command line.

6.3 Plugin's functions modification

In the context of creating the plugin, once all the data from the configuration file is extracted, it is also classified and stored in an array of strings. From this moment on, the parameters can be accessed from the plugin so as to reject those unnecessary for the execution.

Taking advantage of this new functionality, the pe_affinity and use_bulk_xfer parameters, which can only take yes/no values, are assessed during the execution in order to determine whether task_affinity and bulk_min_msg_size parameters must be taken into consideration for a specific scenario or not.

In addition, a checksum method has also been introduced. This acts as follows: as each parameter is associated with its corresponding prefix and value, for instance –task_affinity: cpu:4; a signature is applied and this sentence is stored as a unique integer value that will be stored in an array of integers. Straightaway, before being stored, the method checks whether this value is already in the vector or not (this would imply that the sentence is repeated). If signature is found the parameter combination is rejected since it is a redundant one. Thus, the MPI Parameter plugin's efficiency is improved as it is shown in the section 7.2 Real executions in FSSIM application.

Some functions from the plugin had to be modified and adapted in order to achieve these new functionalities. Below, there is an explanation of the included changes in each one of them:

- initialize(): If there exist variables that must be initialized, this initialization should be done in this function. Additionally, a new functionality has been added that allows the plugin to know if the *auto_eager_limit* parameter has been added. If so, this will affect the stated configuration for the *eager_limit* and *buffer_mem* parameters by rejecting their values because these values will be automatically computed. Furthermore, the commented treatment for the *task_affinity* values will be also done here. In case of finding this parameter, its values will be kept in a *TaskAffinity* type vector so as to let the *prepareScenarios()* function use it afterwards.
- createScenarios(): In this function the range of values for *eager_limit* and *buffer_mem* are automatically defined in case the parameter *auto_eager_limit* appears in the configuration file. Besides, the number of Bytes to communicate by the application as well as the message weights are also treated in this method. The values (of type *start : end : step*) for *eager_limit* and *buffer_mem* will depend on the proportion of messages of certain sizes with regard to the total number of messages. Additionally, an error design was corrected. This error changed the step value depending on the associated range to that parameters, thereby, the number of possible cases was not reduced. The adopted solution has been to homogenize the step by assigning a constant value of 1024 Bytes without taking into account the range.
- prepareScenarios(): In this method, the new checksum is used and the reject process of the unnecessary parameters is carried out. In addition, there also appears the creation of several *flags* by associating the set of parameters to an specific value. These *flags* are preceded by different sentences depending on the MPI implementation "-" for IBM "—mca" for OpenMPI, "-genv" for Intel and "export" for any other types of MPI then, each introduced parameter is adapted to the MPI implementation selected by the user. In case of the IBM MPI (purpose of this work) each parameter from the configuration file is stored in form of variables.
- restartRequired(): This function will determine whether a restart of the execution must be done or not. In case that *prepareScenarios()* had encountered a repetition of a command line, its associated scenario would be marked as empty, provoking to return *false* and to not restart the application in the moment of calling this method.

7 RESULTS DISCUSSION

Below, we summarize all the changes introduced in each one of the three parts in which this project has been divided:

The improvements for the base file that Flex will use to generate the syntactical analyzer have been:

I. The initial file has been restructured so as to represent in an organized way its different sections.
II. The improvement to detect the lexicographical errors before sending the encountered data to the parser has been implemented. Previously, any type of parameter was accepted as valid. Now the line where the error appears is shown as well as the error that provokes it.
III. New recognition patterns have been added. These make the code comprehension easier and improve the symbol recognition capacity. As an example, in order to represent the symbol "=" we now must write *({equals}).* This can capture not only the "=" but also as many tabulations and spaces surrounding it as

needed. Previously, if it was preceded or followed by any spaces or tabs the "=" symbol was not recognized. This enhancement has also been made for the "; " ":" and ","symbols.

IV. It can now recognize and distinguish the different ways through which the user can introduce the values for parameters, that is to say that some value inputs like *yes, no* are recognized as well as the *start : end : step* range. Additionally, the introduction of Bytes can be followed by K (Kilo), M (Mega) o G (Giga). Finally the specific *task_affinity* structure can be recognized too.

The introduced changes for the base file used by Bison in order to generate the grammatical analyzer have been:

I. General restructuration of the file. It has been divided into documented sections that make the structure more user-friendly and easier to understand. The parsing zones for the IBM MPI parameters and the future OpenMPI and Intel MPI and functions are also delimited.
II. Parser implementation (grammar) of the whole lexical found by the lexicographical analyzer.
III. Inclusion of all the necessary tokens to parse and manage each one of the parameters for the IBM MPI implementation when chosen in the configuration file.
IV. Inclusion of the function that treats all found intervals of type *start : end : step* where *start* is the initial value of the sequence, *end* the final value and *step* the jumping value. In addition, these three values can also be specified in Bytes, Kbytes, Mbytes or Gbytes only by adding the prefix at the end of each value. The function is responsible for transforming these values into Bytes before storing them in *long* format in the plugin variables.
V. Implementation of the function that takes one by one the IBM MPI parameters from the configuration file and assigns them to an *enum* variable that will be understandable for the plugin.
VI. Treatement of the values from the new data structure *TaskAffinity* implemented in the MPI Parameters plugin.
VII. Binding of the last implementation of the grammatical analyzer with the new one in order to prevent compatibility problems.

Finally, the following improvements have been implemented in the plugin's code:

I. Creation of the new *TaskAffinity* structure where all the obtained values from the *task_affinity* parameter are stored. In the *initialize()* function, the parameters are received in the shape of a vector of *TaskAffinity* elements. Afterwards, each one of this is converted to *string* format so as to correctly generate the command line.
II. Control of the *auto_eager_limit* parameter. Now if it is found, the values for *eager_limit* and *buffer_mem* are rejected if present in the configuration file.
III. Analysis of the parameters whose range value is *yes, no.* If they appear now, an analysis of those that will be affected is performed. This will optimize the plugin's execution since the number of possible cases to run is reduced.
IV. Implementation of a checksum method whose main objective is to detect those redundant combination of parameters by using a numeric signature that will identify each one of them and let the plugin know whether there is a repeated one or not.

7.1 Test executions

Some tests were held over a simple application during the realization of the new modifications of MPI Parameters. This program executed the sum of the *n* first natural numbers stored in a vector of the same position length. The user provided the *n* value as well as the number of processes. Once defined, the program divides the vector in equal parts so as to compute the calculations in a parallel way. Finally, when all the processes obtain their own final results they send to their father process their results by using MPI in order to make the last addition.

The following file was created with the objective of testing how lexical errors are managed in the new plugin version. It includes several errors that passed undetected in the original version of the plugin.

```
MPIPO_BEGIN IBM
eagers_limit = no;
# 'eagers' debería ser 'eager'.
task_affinity = yes;
# 'yes' no es un valor válido.
use_bulk_xfer = 1234;
# Solo debe aceptar 'yes' o 'no'.
pe_affinity=YES,NO;
# Estos valores deberían ser aceptados
# aunque estén escritos en mayúsculas.
MPIPO_END
```

Note: In the commentaries (starting with "#") each one of the sentences is explained.

The results obtained by executing the first version of MPI Parameters with the application test and with the configuration file stated above were the following ones:

- The *eagers_limit* parameter was accepted and was not thought as an invalid parameter. The execution kept running and the plugin executed the application with the following command line:

```
Restart command: ./fssim ./input/Fish […]
App flags: -eagers_limit no
```

  Consequently, the time of this execution was lost since the parameter was not a valid one.
- The *task_affinity* error is not detected by the parser, so the experiment is generated producing the following error:

```
ERROR: 0031-308
     Invalid value for -task_affinity: no
```

- For the *use_bulk_xfer = 1234* case, the plugin not only wasn't able to distinguish that *use_bulk_xfer* can only keep *yes/no* values but also the experiment is executed with the mistaken values.
- In the case of *pe_affinity = YES,NO* the values were correctly accepted although those weren't transformed into lowercases.

The very same combination of parameters in the new version of the plugin generated the following results:

- For the *eagers_limit* case there appeared the following error provided by the parser and before excuting it:

```
[ERROR]: parameter 'eagers_limit' not
recognized on line: 2
```

- For the *task_affinity = yes case* there appears the following error message:

```
[ERROR]: value 'yes' not recognized for
TASK_AFFINITY on line: 4
```

- In the case of the parameter and value *use_bulk_xfer = 1234* the following error was displayed:

```
[ERROR]: value '1234' not recognized on
line: 6. This must be a yes/no value.
```

As it can be seen, the new implementation of the plugin allows to distinguish the error type in each case as well as to show message with the values that must be introduced (only for values).

- Finally, for the parameter *pe_affinity = YES,NO* we did not obtain any error message and, in addition, a correct conversion of the values to lowercase was stored.

7.2 Real executions in FSSIM application

In order to correctly test the plugin, it was compulsory to test the enhanced version of MPI Parameters with a real application. Because of this, the FSSIM biological simulator application **[13]** that models the movement of big shoals of fishes was used. The configuration used for the test executions selecting the *individual* search algorithm was:

```
MPIPO_BEGIN IBM
SEARCH = individual;
pe_affinity = yes, no;
task_affinity = core, cpu:4, mcm;
use_bulk_xfer = yes, no;
bulk_min_msg_size = 512k, 2M, 100M, 1G;
eager_limit = 2M, 16M;
wait_mode = nopoll, poll, sleep, yield;
css_interrupt = yes, no;
hints_filtered = yes, no;
MPIPO_END
```

First, all parameters were accepted by the plugin without any problem (none of them was written in an incorrect way). Likewise, we could check the proper functioning of the *use_bulk_xfer* and *pe_affinity flag* parameters. These canceled the execution of *bulk_min_msg_size* and *task_affinity* respectively when their values were *no*. Below, the output values from the execution are shown, demonstrating the function of these *flag* parameters. When PTF assigns to them a *yes* value in the *prepareScenarios()* function, they don't limit the execution of their associated parameters in any moment as it can be seen:

```
Restart command: ./fssim ./input/Fish […]
App flags: -pe_affinity yes
                -task_affinity core
```

```
Restart command: ./fssim ./input/Fish […]
App flags: -use_bulk_xfer yes
               -bulk_min_msg_size 524288
```

Nonetheless, when the values of these *flags* is *no*, the results are the following:

```
Restart command: ./fssim ./input/Fish […]
App flags: -pe_affinity no _
```

```
Restart command: ./fssim ./input/Fish […]
App flags: -use_bulk_xfer no
```

In addition, we have verified that effectively, MPI Parameters now manages to recognize the new *task_affinity* structure generating it:

```
Restart command: ./fssim ./input/Fish […]
App flags: -task_affinity cpu:4 _
```

This demonstrates that the command lines are correctly created for this parameter with values that look like "*-parameter string : value*".
In the same execution, it can also be seen how the conversion from *K*, *M* and *G* to Bytes is done as it can be shown in the next output message:

- For the *bulk_min_msg_size = 512k* case we obtain:

```
Restart command: ./fssim ./input/Fish […]
App flags:-bulk_min_msg_size 524288
```

- For the same case but now with *2M* we get:

```
Restart command: ./fssim ./input/Fish […]
App flags: -bulk_min_msg_size 2097152
```

- And for the *1G* case it is shown the following:

```
Restart command: ./fssim ./input/Fish […]
App flags: -bulk_min_msg_size 1073741824
```

Finally, PTF is able to determine through the *individual* search strategy that the best combination of parameters is:

```
-task_affinity core -pe_affinity yes
   -use_bulk_xfer no -buffer_mem 524288
```

If we calculate the average time of execution from 159 results with 12 lasting approximately 24 seconds, 6 of them 82 seconds and the remaining 129, 13 seconds, we obtain an average time of 19,04 seconds. It should be pointed out that, thanks to using the *individual* search algorithm, the total number executions is only 28 instead of the 1536 that would suppose not using it. Additionally, 5 of these cases were not executed due to the elimination of unnecessary parameters and the checksum function, representing a reduction of 95,02 seconds in the total execution time.

If the GDE3 search algorithm is now used together with the same input parameters, the best combination appears to be:

```
-task_affinity core -pe_affinity no
-wait_mode poll -css_interrupt no
-eager_limit 1024
-bulk_min_msg_size 1073741824
-use_bulk_xfer no -hints_filtered no
-buffer_mem 1073741824
```

By using this algorithm, 159 experiments are generated. In addition, due to the use of the *flags* and the checksum function, we manage to dismiss 28 additional cases, reducing the time of execution (taking into account that each execution lasts 19,04 on average) in 533 seconds. Finally, an execution with the search algorithm *exhaustive,* would run each one of the possible combinations. Taking into account that there exist 1536 different combinations in the given configuration file and that the average time of one execution is 19,04 seconds, the execution time would be 29245 seconds. Nevertheless, by using the new functionalities of MPI Parameteres, the number of cases is reduced to 704, avoiding executing 832 unnecessary or repeated combinations. This supposes a reduction of 15841 seconds in the total execution time.

8 Conclusions

Finally, we can conclude that this Project has contributed to improve the MPI Parameters plugin from the PTF tool and its parsing data files generated by the programs Flex and Bison thanks to the achievement of the following objectives:

- Making a verification of the introduced parameters and showing the error messages before launching the application with a wrong parameter.
- Adding the possibility of introducing the prefixes *K, M* and *G* in order to liven up the introduction of values expressed in Bytes.
- Manage to treat parameters and values from the IBM MPI implementation in the plugin independently by using variables to store them.
- Delete those unnecessary parameters for the executions by means of the flags *pe_affinity* and *use_bulk_xfer.*
- Implementing a checksum *method* that avoids the execution of repeated tests and thus improving the general performance.
- Increasing the range of possible values for the *task_affinity* parameter from the IBM MPI implementation.
- Create a structure that contains the values of the new *task_affinity* parameter.
- Control that if the *auto_eager_limit* parameter is introduced the values introduced for *buffer_mem* and *eager_limit* are redundant since a value will be computed for them inside the plugin.
- Correct the design error in *createScenarios()* that changed the step value for *eager_limit* and *buffer_mem.*

Lastly, it ought to be pointed out that the files used by the syntactical analyzer Flex and the parser generator Bison have been completely restructured so as to facilitate other future MPI implementations such as Intel MPI or OpenMPI.

9 Thanks to

I would like to express my gratitude to my mentor Eduardo César and his unconditional help and enormous patience during this project. Without him I wouldn't have been able to achieve the initial objectives.

10 BIBLIOGRAPHY

[1] AutoTune Book | PTF. (2016). [online] Available at:http://periscope.in.tum.de/wp-content/ uploads/2015/03/autotune_book1.pdf [Accessed 10 Feb. 2016].

[2] Autotune-project.eu, (n.d.). *Automatic Online Tuning*. [Online] Available at: http:// www. autotune-project.eu/ [Accessed 6 Feb. 2016].

[3] Periscope.in.tum.de, *Periscope | Periscope Tuning Framework*. [Online] Available at: http://periscope.in.tum.de/?page_id=7 [Accessed 6 Feb. 2016].

[4] Computing.llnl.gov, *Message Passing Interface (MPI)*. [Online] Available at: https://computing.llnl.gov/tutorials/mpi/ [Accessed 6 Feb. 2016].

[5] J. Levine, *flex & bison*, O'Reilly, 2009. [Online] Available at: http:// web.iitd.ac.in/ ~sumeet/ flex__bison.pdf [Accessed 6 Feb. 2016].

[6] Vern Paxson *Flex versión 2.5,* ed. 2.5 Abril 1995 [Online]. Available at: http://es.tldp.org /Manuales-LuCAS/FLEX/flex-es-2.5.pdf. [Accessed 6 Feb. 2016].

[7] Charles Donnelly and Richard Stallman, *Bison*: FSF 23 January 2015 [Online] Available at: http://www.gnu.org/software/bison/manual/ bison.pdf. [Accessed 6 Feb. 2016].

[8] Open-mpi.org, *Open Tool for Parameter Optimization (OTPO)*. [Online] Available at: http://www.open-mpi.org/projects/otpo/ [Accessed 6 Feb. 2016].

[9] Software.intel.com, *Application Specific Tuning | Intel® MPITune*. [Online]. https:// software.intel.com/en-us/node/528814 [Accessed 6 Feb. 2016].

[10] Roland C. Backhouse, *Syntax of Programming Languages Theory and Practice.* Edinburgh, Scotland: PHI.

[11] R. Bader, *IBM MPI: An MPI implementation for SuperMUC* [Online]. Available at: https://www.lrz.de/services/software/parallel /mpi/ibmmpi/. [Accessed 6 Feb. 2016].

[12] *IBM MPI: An MPI implementation for SuperMUC* [Online]. Available at: https:// www.lrz.de/ services/software/parallel/mpi/intelmpi/. [Accessed 6 Feb. 2016].

[13] Google Books, (2016). *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*. [Online] Disponible en la web abreviada: http://bit.ly/1KAC14V [Accessed 6 Feb. 2016].

APPENDIX

In this appendix is shown the parameters table now supported by MPI Parameters and only available when the MPI implementation chosen for MPI is IBM MPI.

Type	Variable	Command	Accepted input values
Pinning of Tasks and Threads	MP_PE_AFFINITY	*-pe_affinity*	*Yes, No*
	MP_TASK_AFFINITY	*-task_affinity*	*CPU, CORE, MCM*
	MP_USE_BULK_XFER	*-use_bulk_xfer*	*Yes, No*
Tuning	MP_BULK_MIN_MSG_SIZE	*-bulk_min_msg_size*	*Bytes*
	MP_BUFFER_MEM	*-buffer_mem*	*Bytes*
	MP_EAGER_LIMIT	*-eager_limit*	*Bytes*
	MP_SINGLE_THREAD	*-single_thread*	*Yes, No*
	MP_WAIT_MODE	*-wait_mode*	*NOPOLL, POLL, SLEEP, YIELD*
	MP_POLLING_INTERVAL	*-polling_interval*	*Bytes*
	MP_CSS_INTERRUPT	*-css_interrupt*	*Yes, No*
Advanced Tuning Parameters	MP_HINTS_FILTERED	*-hints_filtered*	*Yes, No*
	MP_MSG_ENVELOPE_BUF	*-msg_envelope_buf*	*Número entero*
	MP_RETRANSMIT_INTERVAL	*-retransmit_interval*	*Número entero*
	MP_THREAD_STACKSIZE	*-thread_stacksize*	*Bytes*
	MP_ACK_THRESH	*-ack_thresh*	*Número entero*
	MP_IO_BUFFER_SIZE	*-io_buffer_size*	*Bytes*
	MP_IO_ERRLOG	*-io_errlog*	*Yes, No*
	MP_REXMIT_BUF_SIZE	*-rexmit_buf_size*	*Bytes*
	MP_REXMIT_BUF_CNT	*-rexmit_buf_cnt*	*Número entero*