Extension of the PTF's MPI Parameters Plugin

Jordi Caballol Carrasco

Resum– Actualment la computació d'altes prestacions és una part important en moltes disciplines. Aquestes aplicacions cada cop són més complexes i, per tant, necessiten més rendiment. Per a intentar aconseguir aquest rendiment és molt important optimitzar i ajustar correctament les aplicacions. Com que aquesta és una tasca bastant complexa existeixen diverses eines pensades per a facilitar-ho.
Periscope Tunning Framework és una eina construïda sobre l'entorn de monitoratge Periscope que ens permet ajustar automàticament diferents aspectes d'aplicacions HPC utilitzant plugins especialitzats en algún d'aquests espectes, com poden ser els paràmetres de MPI o els *flags* del compilador. Un dels plugins que existeixen és el que ens permet ajustar els paràmetres de MPI.
En aquest treball es presenten les millores d'aquest plugin, afegint-hi suport per als paràmetres de Intel MPI, de manera que el plugin sigui capaç de detectar quan li estem passant valors incorrectes així com decidir quan podem estalviar-nos l'execució d'algun cas.
Paraules clau– Intel MPI, optimització de paràmetres, Periscope Tuning Framework, MPI Parameters Plugin

Abstract– Nowadays high performance computing is an important part of some disciplines. These aplicacions are becoming more and more complex, therefore, they need a better performance. In order to try to get this better performance is very important to optimize and adjust correctly the applications. Since it is a very complex task there are some tools designes to make it easier.
Periscope Tuning Framework is a tool built over the Periscope monitoring framework that allows us to automatically adjust some aspects of HPC applications by using plugins that are specialized in one of those aspects, like the MPI parameters or the compiler flags. One of those plugins is the one that allows us to adjust the MPI parameters.
In this work we present the improvements added to this plugin, adding support for the Intel MPI parameters, so the plugin will be able to detect when we are using wrong values and to decide when we can avoid executing some cases.
Keywords– Intel MPI, parameter optimization, Periscope Tuning Framework, MPI Parameters Plugin

1 Introduction

High Performance Computing is essential for research and development in many fields, such as physics, astronomy or engineering. Nowadays high performance applications and the systems they run on are becoming very complex, using heterogeneus technologies in the same application and complex parallel paradigms that require lots of comunications between nodes. Since the capacity of a single node is limited this parallelism is crucial if we want to get a good performance, but at the same time it adds a lot of complexity, in some cases the time spent in comunications is even greater than the time spent in performing calculations. Tuning correctly these applications is vital if we want to minimize the negative effect of the parallelism while keeping its benefits, but due to its inherently complex nature it is becoming a very difficult and incredibly time consuming task.

There are some tools that help the programmers in this task of reducing the amount of time needed to optimize applications, such as MATE [1], Scalasca [2], Periscope [3] and TAU [4]. Some of them, as is the case of *Periscope*, will help us in gathering and analysing data from the executions of our applications so we can determine what optimizations we need to implement, but even with this tool getting good results still requires lots of trial and error, and this means time. The logical evolution of this kind of tools is a tool that not only tells us what problems we have but also gives us some advice on how to solve them.

• E-mail de contacte: jcaballol94@gmail.com
• Menció realitzada: Enginyeria de Computadors
• Treball tutoritzat per: Anna Bàrbara Sikora (Arquitectura de Computadors i Sistemes Operatius)
• Curs 2015/16

This is the case of PTF [5], a tool built over *Periscope*. The idea behing PTF is to automatically execute a set of experiments with different settings and analyse the results of these experiments, then PTF determines which one resulted in the best results according to a specific criteria, like the time of execution or the power consumption.

To get the configurations it will try, PTF uses a set of plugins, one for every supported feature. Currently there are the following plugins supported: Compiler Flags Selection, DVFS, Master-Worker, Parallel Pattern and MPI Parameters. The latter, the MPI Parameters Plugin, is the one we will work with. This plugin is meant to read a list of parameters and values from a configuration file and try combinations of them trying to find the best. Since the first version it supports IBM MPI, Intel MPI and OpenMPI but in a very limited way: it wasn't able to detect whether a given parameter exists or not, nor check if the values we specify for a parameter are valid or not. Also there are some combinations of parameters that are equivalent but the plugin isn't able to determine this so it will execute both, wasting some precious time. The objective of this work is to solve this for the Intel MPI set of parameters.

The remainder of this paper is organized as follows. Section 2 describes the PTF and the MPI parameters plugin, section 3 will define the objectives, in section 4 we will see the state of the art, in section 5 the methodology used is explained, in section 6 we will see how it was implemented, in section 7 we will see the results obtained and in section 8 we will extract some conclusions from the work.

2 Context

In this section, in order to contextualize the developed work, I will introduce the PTF environment and its plugin based structure.

2.1 Plugin Tuning Framework

PTF is a tool focused on static tuning, identifying tuning recomendations from data extracted from application runs.

PTF is developed using a plugin based structure, what makes it easily extensible and allowing the future implementation of both open source and propietary plugins. The mission of the plugins is defining what cases we need to test and prepare the execution of the experiments. Since the number of cases can be very large the plugins can execute some previous data and use expert knowledge to try to reduce the number of cases.

Once we have defined a search space with all the cases, we can search the best one using one of the predefined algorithms or we can use a plugin specific algorithm. Also the seach can be based not only on the execution time but also on power consumption.

In fig. 1 we can see the structure of the framework. The PTF framework executes the plugins and controls the execution of the experiments. To monitor and evaluate the experiments PTF uses Periscope. It allows us to gather the data in runtime and even apply some tuning actions without restarting the application, saving some time.

Also the framework includes an interactive interface based on Eclipse, making it more comfortable to use.

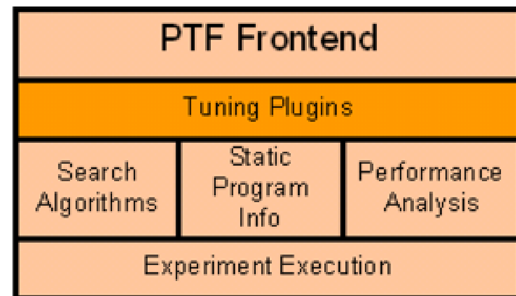

Fig. 1: The architecture of PTF

2.2 The MPI Parameters Plugin

In this work we focus on the MPI Parameters plugin. This plugin is developed for MPI based applications, regardless of the implementation, and it's used to find the best combination of parameters. The idea of the plugin is very simple, first it will read a configuration with the following information:

- ***The MPI implementation:*** It will tell us what implementation we are using, since for every implementation the way we have to pass the parameters is different, and so are the parameters. The plugin fully supports IBM MPI and supports Intel MPI and OpenMPI in a limited way. It also has the option of not specifying the version, and in this case we will pass the parameters as environment variables.
- ***The search algorithm:*** It specifies wich algorithm of searching space we should use.
- ***The parameters:*** In the file we also have to specify the parameters and the values we have to try. In its first version the plugin can read ranges of numbers and lists of strings, and with the implementation of IBM MPI it added the support for strings with parameters, as they were needed for the task_affinity parameter.

Once the plugin has read the file, it processes the values of the required parameters. This includes generating some values from complex parameters, storing the lists of strings (the PTF environment only allows us to define the experiments with ranges, so we have to manage the strings ourselves) and treating a special case: For IBM MPI we can choose the option to set the eager limit automatically. The eager limit is very important because it determines when we will use the rendezvous protocol for sending messages and when we will use the eager protocol instead, and this can lead to a great difference in performance. For this case the plugin will execute a first experiment to measure the size of the messages and use this information to calculate the best size.

Finally the plugin will execute the experiments. In the first version the plugin simply passed the parameters to the environment to execute them but in the more recent version they added an optimization to skip redundant cases. Now

the plugin checks all the parameters to eliminate parameters that are meaningful in this specific situation and then generates a checksum of the remaining parameters. This checksum is searched in a list containing the checksums of all the executed experiments to see if PTF has already executed the same combination and if so it will skip this execution.

This skipping may sound trivial but the PTF environment isn't meant to do it so it is more complex. To do it they get the results of the execution we did and insert a copy of them as if they were the results of the current execution.

3 Objectives

The objective of this work is to extend the MPI parameters plugin to improve the way it processes the Intel MPI parameters, making it more efficient and easier to work with. The initial version of the plugin didn't have any information about the parameters, it only read them from the configuration file and assumed that they were correct without any kind of processing. It has two main drawbacks: if a user introduced a mistake in the configuration file the plugin won't complain, so PTF will not be aware that it isn't executing properly. Also there are some parameters that activate some funcionality while others tune this funcionality, so if the funcionality is deactivated PTF can ignore the other parameter. However, since the plugin doesn't have this information it will try all the combinations of it anyway.

The last semester Alberto Olmo solved the presented problems for the IBM parameters [6], but for Intel MPI and OpenMPI it still works the same way. In this work we will try to implement something similar for the Intel MPI parameters and, if possible, even improve it.

In order to do this we will need to work in two different things: on one hand we will need to rewrite the parser. To implement the parser we will use Flex and Bison [8] like the current parser. Flex and Bison allow us to define a parser divided in two parts: a scanner that reads "words" and a parser that works with the relations between the words.

On the other hand we will need to recognize the parameters and the values we assign to them, so we can check if they exist. Also in the scanner we can perform a first check of the values: for example if we have a parameter that recieves numbers we won't accept strings.

In the parser we will also need to define rules to match the parameters recieved from the scanner. Here we will check that the values are defined with the correct structure and we will save them so the plugin can process them.

Once we have a good parser the other part that will recieve our focus will be preparing the experiments to be executed: since now PTF knows the relations between the parameters it can ignore parameters that have no effect and then see if it has already executed an equivalent case so it can skip the redundant cases, achieving an improvement in performance.

4 State of the art

When it comes to automatically tune the parameters for an Intel MPI application [7], the main application we find is Intel's utility MPItune. MPItune offers us two modes of working: A cluster specific mode and an application specific mode.

- The cluster mode is completely different from the PTF, it executes a serie of benchmarks to find the best parameters for the cluster, with independence of the application being executed. This is very interestig because you just need to run the utility once instead of running it for all the applications.
- The application specific mode is very similar to our plugin. In both the main idea is the same: try some combinations of parameters and select the best of them. However there are some important differences. The main of them the way the parameters are selected: MPItune uses a fixed set of parameters while in PTF the parameters are specified throug a configuration file. In one hand it makes MPItune easier to use, and since the tool is from Intel like the Intel MPI implementation, we can assume that the parameters will be pretty good. On the other hand no one will know our application better than us, so allowing us to choose the parameters can lead to great results, specially if it's done by someone with a lot of experience. Also allowing the user to set the parameters allows him to limit the things TPF is going to try instead of always having to try all of them. Another difference between them is that MPItune always uses execution time as a criteria while PTF allows us to use others like the power consumption.

So as we see we currently have a good tool to help us tunning the Intel MPI parameters, but more than being an alternative to PTF, MPItune is useful in different situations. MPItune is useful to do more general optimizations while PTF is useful for performing more specific tasks.

5 Methodology

The motodology followed during the development of this work was the cascade development, i.e. every step started once the previous one was finished.

The process was divided in the following 10 tasks:

- ***Gather the necesary information:*** This task consisted in collecting the necesary information, like a book about PTF and it's plugins [9], the Intel MPI reference manual [10] or a tutorial about creating parsers with Flex and Bison [11].
- ***Study the PTF environment:*** This task and the two following tasks consist of studying the information gathered in the first task.
- ***Study the use of Flex and Bison to create parsers.***
- ***Analize the Intel MPI parameters.***
- ***Prepare the environment:*** In this step I prepared a computer and connected it with the SuperMUC supercomputer to be able to work with it.
- ***Implement the parser:*** The firs step in the development was developing the parser.

- ***Test (and fix) the parser:*** Once I had the parser ready it was time to test it and fix the errors it had.
- ***Implement the elimination of redundancies:*** With the parser ready then we implement the optimization.
- ***Test the elimination of redundancies:*** As I did with the parser, once the optimization is implemented we need to test and fix it.
- ***Analize the results:*** Finally we analyze the offects of our changes.

6 DEVELOPMENT

As I explained in the methodology the project was divided in 10 steps, but the implementation itself was divided in two main blocks: the parser and the optimization of removing redundant cases.

To make it more understandable in this section I will divide the parser implementation in three parts:

- ***The scanner***, where we recognize all the parameters and values and check if these values are valid.
- ***The parser***, where we check the structures these values follow and store them acordingly and
- ***Processing the data***, where we use this data to create all the cases that we will need to test and pass them to the application.

Finally to eliminate the redundancies we first remove all the parameters that can be ignored and then use a *checksum* to check if this combination of parameters have been tested before or not.

6.1 The scanner

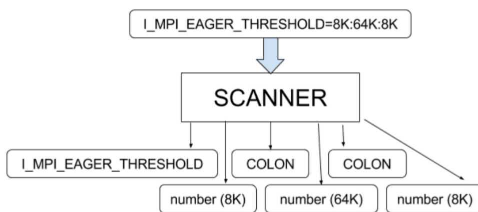

Fig. 2: The scanner

As shown in fig. 2, the scanner reads the text and identifies all the simbols in it. To create the scanner I used Flex, since the existing scanner was created using this tool and I was only expanding it. Flex is a very powerful tool that allows us to perform pattern recognition in the input, like numeric values or specific string (exactly what we need in order to identify the different parameters) and execute C code every time the pattern is matched. This code includes throw errors or pass *tokens* to the parser.

Flex allows us to define our scanner as a state machine, so it will only recognize a pattern if it is in the correct state. Using this feature we can make Flex recognize only the values our parameter can take.

To implement this I first analysed the parameters, searching all the types of inputs we will need to recognize. These are the following inputs:

- ***Intervals:*** An interval of numbers, defined as the first value, the last value and the step between the values.
- ***Yes/no values:*** Values to activate or deactivate something. They can be: yes, enable, on, 1, no, disable, off or 0.
- ***I_MPI_SHM_LMT:*** This parameter is like a yes/no, but instead of taking the positive values it takes "shm".
- ***I_MPI_DAPL_DYNAMIC_CONNECTION_MODE:*** This parameter can take two values: "reject" or "disconnect".
- ***I_MPI_TCP_POLLING_MODE:*** This parameter can take the values "poll" and "epoll" and for epoll we can specify "epoll:edge" or "epoll:level".
- ***I_MPI_OFA_RAIL_SCHEDULER:*** This one takes the following values: "round_robin", "first_rail" and "process_bind".
- ***I_MPI_FABRICS:*** With this parameter we define the fabrics we are going to use. These fabrics can be: "shm", "dapl", "tcp", "tmi" or "ofa". We can also specify two fabrics, one for intranode comunications and one for internode comunications, using a colon: *intranode:internode*.
- ***I_MPI_ADJUST_REDUCE_SEGMENT:*** For this parameter we recieve a list of possible options delimited by double quotes. These options can be a default segment size (then we recieve a number) or a size for a specific algorithm (written as *algid:block_size*).
- ***Adjust:*** This family of parameters is the most complex one: Like the *adjust reduce segment* case here we recieve lists delimited by double quotes and we recive the *algid:data* structure, but in this case the data we recieve isn't just a number, it can be te size of the messages (a number), or the size and number of processes (*size@n_procs*). Also both the size and the number of processes can be ranges (*ex. 1024-2014*).

I've defined a state for each one of these inputs, so when the scanner reads a parameter name it passes the corresponding token to the parser and switch the state the the proper one, once the scanner is in the state of an input type it will only recognize and pass to the parser the symbols that can appear in this kind of input (for instance if it recieves a range it will recognize numbers and colons) and throw an error otherwise. It returns to the original state when it finds an end of line so it can recognize the next parameter name.

Once the list of inputs is defined we can now create the rules for every parameter we need to parse. All the rules follow the same structure. For example:

```
<VARS_INTEL>(I_MPI_EAGER_THRESHOLD|
i_mpi_eager_threshold)({equals})
{
  BEGIN(PARSE_INTERVAL);
  return I_MPI_EAGER_THRESHOLD;
}
```

<VARS_INTEL> means that the rule will only be matched if the scanner is in the state VARS_INTEL (that

is parsing Intel parameters). Then we define the rule, in this case we check for the word i_mpi_eager_threshold both uppercase and lowercase followed by "=". Then we define the code that will be executed if the rule is matched: first we change the state to the state of reading the values (in this case the parameter recieves an interval) and then the scanner passes the token to the parser.

6.2 The parser

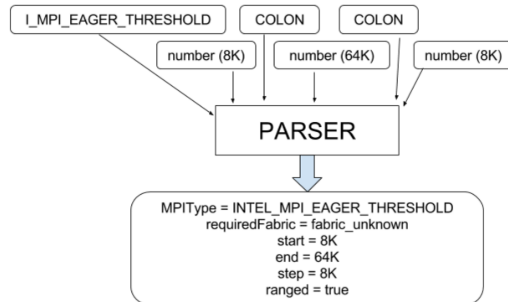

Fig. 3: The parser

As shown in figure 3, the parser gets the symbols from the scanner and stores a structure with all the information about the parameter. For the implementation of the parser we use Bison. With Bison we can use rules to match the different structures we will find, like in Flex we can define code that will be executed when the rule is matched. The parser will use this code to store the values it will recieve from the scanner in the corresponding data structures.

As in the scanner we will define two kinds of rules: the rules to process the parameters and the rules to parse the values of those parameters.

The rules to parse the names of the parameters follow the same idea as in the scanner. Following the example we used in the scanner section for the parameter i_mpi_eager_threshold we have the following rule:

```
intel_statement:
      ...
  | I_MPI_EAGER_THRESHOLD
    {
      setIntelMPIType(
        (mpiType)
          INTEL_MPI_EAGER_THRESHOLD,
        "I_MPI_EAGER_THRESHOLD",
        (intelMpiFabric)fabric_unknown);
    }
    intel_getValues;
      ...
```

So we define the rule *intel_statement* with a case for every parameter. All the cases are almost identic: if the parser recieves a specifiec token it executes the code between brackets. This code only contains a call to the function *setIntelMPIType*.

The function *setMPIType* creates a structure with the information of the parameter and adds it to the vector where we store all the data. For the intel parameters I have defined the function *setIntelMPIType* that does the same but it also stores the fabric we need to use so the parameter is valid (for example if we aren't using shared memory we will ignore all the parameters related to shared memory), so in the case of Intel Parameters we are saving the type of the parameter, the actual name of the parameter (it is the name we will use to pass the parameter to the application) and the fabric. In this example the parametrer is common to all fabrics so we pass *fabric_unknown*.

The next statement, *intel_getValues*, is the rule we expect to find following the token. If the parser recieves something that doesn't match the rule it will throw an error. For this parameter we use the rule to get a range. The rules I have defined to read the different values are:

- ***intel_getValues:*** This is the rule to read numeric values. This rule defines a list of *intel_value* separated by commas. The rule *intel_value* reads a value or a range and passes it to the function *treatInterval* to store it. This function simply stores the range using three values: the beginning, the end and the step. If no step is defined we use 1 by default and if it's a single value instead of a range we set the start and the end to the same number.
- ***intel_getStringList:*** This is used to read parameters that use strings as values (like yes, no, reject, disconnect, etc.) It's simply a comma separated list and for every value it calls *PushValue* to store it. Since there are multiple ways of passing yes or no (like enable, on, 1, etc.) in the function *PushValue* it changes all the positive values for "1" and all the negatives for "0", so then the plugin can easily remove the equivalent values.
- ***intel_getFabrics:*** This rule is for parsing the parameter *I_MPI_FABRICS*. It defines two cases: we can recieve only a fabric, or we can recieve the two fabrics separated by a colon. In the firs case it will store it as an intranode fabric, leaving the internode as unknown, while in the second one we will store the first as intranode and the latter as internode.
- ***intel_getSegmentsLists:*** To read the values of the parameter *I_MPI_ADJUST_REDUCE_SEGMENT* we define a comma separated list of segment lists.

  The rule to read the segments lists define a segment list delimited using quotes. When we find the first quote we call the function *ClearSegmentsList* to clear the current list so we can start saving the new one. After we find the second quote we call *PushSegmentsList* to store the list.
  The segment lists are simply a list comma separated segments. These segments can be a default segment (a single number) or an algorithm id and the value for this algorithm (in the format *alg_id:value*). To store them in the current list we just call *PushDefaultSegment* for the default value or *PushSegment* for the rest.
- ***intel_getAlgorithmsLists:*** This is the rule used for the *I_MPI_ADJUST* family of parameters. Like for the segment lists we define a comma separated list of algorithm lists and every list is delimited using quotes. We use *ClearAdjustAlgList* and *PushAdjustAlgList* the same way to clear and store the lists we find.
  To define the algorithms we have two cases: a default

algorithm and an algorithm with a list of conditions. In both cases when we read the first value it is the algorithm id, so we store it using *NewAlgorithm*. If we then find a semicolon it means that it was a default algorithm so we start reading the next, but if we find a colon we need to read the list of conditions, that is a comma separated list. For every condition in the list we call *NewParameter* to add a blank condition to the list and then we read it to fill the condition. The conditions can have the following structures: *range, range-range, range@range*, or *range-range@range-range*. To store them we just call *ParameterInterval* or *SingleValueParameter* for every range and if we find the "@" we store its position using *SetProcStart*.

6.3 Processing the data

Since the interface PTF provides us to define the cases we want to try only accepts ranges, we need to create some way to express all the other values. In the original implementation of the plugin they defined a very simple way to pass the strings to the application. They created a list with all the strings and passed to the PTF the range of the indexs to reference the strings. For example if we have three strings we pass the range from 0 to 2, with the 0 refering the first one, 1 refering the second and so on.

For the Intel MPI parameters we also have some other kinds of values, like multiple ranges or the MPI_ADJUST parameters, so we need to find a way to pass them. The solution I used is very simple: it creates a string for every value of the parameter and then it passes the generated strings to the application by using the former method.

Also when processing the data it checks if the required fabric of a parameter matches the fabrics specified in the configuration file and, if not, it ignores the parameter.

The way it transforma all the values into strings is the following:

- ***INTEL_MPI_SHM_CACHE_BYPASS_THRESHOLDS:*** This parameter can accept 2 or 4 ranges, so it checks if the number of ranges it has is correct and then it iterates over all the possible combinations creating a string for each one.
- ***INTEL_MPI_ADJUST_REDUCE_SEGMENT:*** For this parameter it has a list of lists of algorithms. Every list of algorithms can contain a single default size for all algorithms or a list of algorithms with their corresponding size. To create the strings it iterates over the list of lists and for every list if it's a default value it just prints the value. Else it iterates over all the algorithms and prints them with the format *alg_id:size*.
- ***INTEL_MPI_ADJUST:*** This family of parameters has the most complex values, so the way of processign them is also the most complex. To create the strings it will need an auxiliar structure, so the first thing it will do is to fill this structure. This structure is just a tree with a first level with all the algorithms, a second level with all the conditions for each algorithm and a third level with a range in every leaf. For example, if the values we have read are: "1:1:5,1024;2:10-15@1-3:4;" we create the tree shown in figure 4:

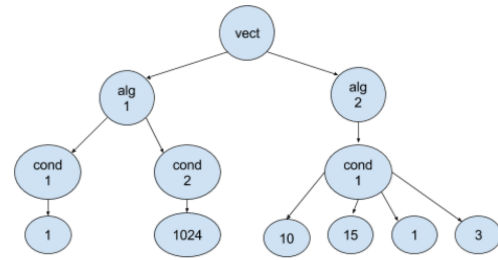

Fig. 4: Example of a tree

Once it has filled the tree we need to keep updating its values untill we've generated all the possible combinations. To do this it will increment the last range (the rightmost one in the figure) by adding the defined increment. If after incrementing a range it becomes greater than the limit, it removes it from the tree and increments the previous range. If it removes all the ranges from a condition, it will remove the condition and, if it removes all the conditions of an algorithm, it will also remove the algorithm. This is repeated until it increments a range that doesn't pass the limit. Finally it will refill the tree again with the initial values of the removed ranges.

By repeating this process we will reach a point where we will remove the whole three, when this happens it means that we have finished.

6.4 Elimination of equivalent combinations

The elimination of the redundant combinations of parameters is pretty straightforward. For every combination of values it has to try, it will first search for parameters that render others useless, for example disabling the fboxes make pointless fixing the size of those boxes. If it finds them it will search for the parameter thats useless and remove it. Once it has removed all the values it generates a checksum and searches it in a list with the checksums of all the executed combinations. If the checksum is in the list it can skip this execution since it means that it has already executed an identical one.

The parameters it will search for are:

- ***I_MPI_SHM_CELL_SIZE:*** If we set this parameter the value of I_MPI_INTRANODE_EAGER_THRESHOLD is set to the same value, so we can ignore its settings.
- ***I_MPI_SHM_CACHE_BYPASS:*** If we disable the cache bypass we can ignore its size, so we remove the parameter I_MPI_SHM_CACHE_BYPASS_THRESHOLDS.
- ***I_MPI_SHM_FBOX:*** It's the same as with the cache bypass, if we are not using fboxes we don't need to set their size so we ignore I_MPI_SHM_FBOX_SIZE.
- ***I_MPI_SSHM:*** Enabling this sets the intranode eager threshold to 262144, so we can ignore the parameter I_MPI_INTRANODE_EAGER_THRESHOLD.

7 RESULTS

In this section we'll se all the changes introduced to the application and some tests to see them working. The main changes we have introduced are:

In the parser:

- The main thing we have added to the parser is the capacity to recognize Intel MPI parameters, so if we make a mistake writing them the parser will complain.
- In addition to recognizing the name of the parameter the parser can now check if the values we assign to it are correct or not.
- Also we can now recognize and process new kinds of data, including the MPI_ADJUST family of parameters and the intel fabrics.
- The parser now knows what fabric is required for every parameter, and can ignore the parameters that are not of the fabrics we are using.

In the plugin's body:

- We added to the plugin the capacity of removing useless parameters and skipping redundant combinations of parameters.

7.1 Testing the parser

To test the parser I created a simple program so instead of executing the whole PTF it will only execute the parser and print all the values it will generate for every parameter. This way testing the parser takes much less time.

The first thing we'll test is recognizing the names of the parameters, to do that I will simply misspell the name of some parameters and see if it complains. For example if we write *I_MPI_SHM_LMT_BUFFER_PUM* instead of *I_MPI_SHM_LMT_BUFFER_NUM* we get the following error:

```
[ERROR]: sentence 'I_MPI_SHM_LMT_BUFFER_PUM'
not recognized on line: 4
```

The next thing I tested is checking the correctness of the values. I tested it the same way: passing invalid data to the parameters. Following with the previous example if we pass a string to *I_MPI_SHM_LMT_BUFFER_NUM* we get:

```
[ERROR]: value 'enable' not recognized
on line: 4
```

And the same happens if we pass a range to a parameter that expects a string.

```
[ERROR]: value '1024' not recognized on
line: 3 this must be a yes/no value
```

With strings it will fail not only if we pass a range but also if we use a string that doesn't match the parameter:

```
[ERROR]: value 'yes' not recognized on
line: 3 this must be a polling mode
```

There are some ranges that only accept some values, like multiples of 64 or powers of 2, this is also checked but in a latter stage, when we are generating the values for each parameter. For example the parameter I_MPI_SHM_FBOX_SIZE only accepts mutiples of 64 so if we pass something different we get:

```
Ignored I_MPI_SHM_FBOX_SIZE: Value not
multiple of 64
```

Finally we want to check if it also discards the parameters of unused fabrics, so we set the fabrics to TCP and use a parameter of SHM:

```
Ignored I_MPI_SHM_FBOX_SIZE: Not of the
specified fabric
```

Now we know that the parser will throw an error if we input wrong values. It's time to check the values it gets when we do it correctly. The first cases we will try are the ranges and the lists of strings, since they were already supported by the original version of the parser and should still work. In this example I use the values

```
I_MPI_SHM_FBOX_SIZE=64:256:64;
I_MPI_TCP_POLLING_MODE=poll, epoll:level,
    epoll:edge;
```

and it generates:

```
Parameter id: 0
Parameter name: I_MPI_SHM_FBOX_SIZE
Parameter range: 64:256:64

-genv I_MPI_SHM_FBOX_SIZE 64
-genv I_MPI_SHM_FBOX_SIZE 128
-genv I_MPI_SHM_FBOX_SIZE 192
-genv I_MPI_SHM_FBOX_SIZE 256

Parameter id: 1
Parameter name: I_MPI_TCP_POLLING_MODE
Parameter range: 0:2:1

-genv I_MPI_TCP_POLLING_MODE epoll:edge
-genv I_MPI_TCP_POLLING_MODE epoll:level
-genv I_MPI_TCP_POLLING_MODE poll
```

Now it's time to test the new kinds of values. First I will test *I_MPI_SHM_CACHE_BYPASS_THRESHOLDS* and *I_MPI_ADJUST_REDUCE_SEGMENT* since they are simpler. We input the following values:

```
Parameter id: 0
I_MPI_SHM_CACHE_BYPASS_THRESHOLDS=
    1:2,3:4,5,1024:2048:512;
I_MPI_ADJUST_REDUCE_SEGMENT=
    "1:1024,2:2048;", "1:10,2:20,3:30;";
```

And the result is:

```
Parameter id: 0
Parameter name:
  I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
Parameter range: 0:11:1

-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  1,3,5,1024
```

```
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  1,3,5,1536
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  1,3,5,2048
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  1,4,5,1024
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  1,4,5,1536
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  1,4,5,2048
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  2,3,5,1024
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  2,3,5,1536
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  2,3,5,2048
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  2,4,5,1024
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  2,4,5,1536
-genv I_MPI_SHM_CACHE_BYPASS_THRESHOLDS
  2,4,5,2048

Parameter id: 1
Parameter name:
  I_MPI_ADJUST_REDUCE_SEGMENT
Parameter range: 0:1:1

-genv I_MPI_ADJUST_REDUCE_SEGMENT
  1:1024,2:2048
-genv I_MPI_ADJUST_REDUCE_SEGMENT
  1:10,2:20,3:30
```

Finally I tested the I_MPI_ADJUST parameter. The values used are:

```
I_MPI_ADJUST_REDUCE = "1:1024@18:20,
  2:1:2-3:4; 3;", "1:1-2@3:4-4:5; 2;";
```

And we get the following values:

```
Parameter id: 0
Parameter name: I_MPI_ADJUST_REDUCE
Parameter range: 0:9:1

-genv I_MPI_ADJUST_REDUCE 1:1024@18,
    2-3;3
-genv I_MPI_ADJUST_REDUCE 1:1024@18,
    2-4;3
-genv I_MPI_ADJUST_REDUCE 1:1024@19,
    2-3;3
-genv I_MPI_ADJUST_REDUCE 1:1024@19,
    2-4;3
-genv I_MPI_ADJUST_REDUCE 1:1024@20,
    2-3;3
-genv I_MPI_ADJUST_REDUCE 1:1024@20,
    2-4;3
-genv I_MPI_ADJUST_REDUCE 1:1-2@3-4;2
-genv I_MPI_ADJUST_REDUCE 1:1-2@3-5;2
-genv I_MPI_ADJUST_REDUCE 1:1-2@4-4;2
-genv I_MPI_ADJUST_REDUCE 1:1-2@4-5;2
```

7.2 Testing the elimination of redundancies

With the parser working there's only one thing left to test: the elimination of redundancies. To test it I picked two parameters that can result in redundant combinations, these parameters are:

```
I_MPI_SHM_FBOX = yes, no;
I_MPI_SHM_FBOX_SIZE = 8K:64K:8K;
```

Here we have 8 values for the size that should be executed twice, one time with fbox to yes and other with no. When executing it with the original version, without the optimization, it gives us the following results:

```
Found best scenario: 3
Parameter combination:

-genv I_MPI_FABRICS shm:dapl
-genv I_MPI_SHM_FBOX 0
-genv I_MPI_SHM_FBOX_SIZE 32768

All Results:
```

I skip the results of the experiments because they are too long.

```
[psc_frontend][INFO:fe] Plugin advice
  stored in: advice_31696.xml

----------------
 End Periscope run! Search took
  419.493 seconds ( 17.2518 seconds
  for startup  )
```

But when we use the new version it returns the following results:

```
Found best scenario: 0
Parameter combination:

-genv I_MPI_FABRICS shm:dapl
-genv I_MPI_SHM_FBOX 0
-genv I_MPI_SHM_FBOX_SIZE 8192

All Results:
Scenario  |   Runtime    | Flags
0  |  3.054260 |
  -genv I_MPI_FABRICS shm:dapl
  -genv I_MPI_SHM_FBOX 0
  -genv I_MPI_SHM_FBOX_SIZE 8192
1  |  3.054260 |
  -genv I_MPI_FABRICS shm:dapl
  -genv I_MPI_SHM_FBOX 0
  -genv I_MPI_SHM_FBOX_SIZE 16384
```

I skip the rest of the results of the experiments because they are too long.

```
[psc_frontend][INFO:fe] Plugin advice
  stored in: advice_7879.xml

----------------
 End Periscope run! Search took
  233.606 seconds ( 17.2598 seconds
  for startup  )
```

Note that it returned a different scenario, but in both cases the parameter I_MPI_SHM_FBOX is disabled (set to 0) so both are equivalent. We can see that in the two scenarios I've put in the example we have this parameter disabled and the execution time is exactly the same, this is because when skipping a case we assign it the results of the equivalent case we have executed. We can also see that the parameters have been skipped because it is displayed in runtime:

```
[psc_frontend][INFO:fe] Starting
  application
  -genv I_MPI_FABRICS shm:dapl
  -genv I_MPI_SHM_FBOX 0
  ./fssim ./input/Fish_64k.dat
  output 1.25 1.30 5 10 using 64
  MPI procs and 1 OpenMP threads...
[psc_frontend][INFO:fe] Starting
  agents network...
[psc_frontend][INFO:fe] Scenario pool
  not empty, still searching...
Ignoring fbox size: fbox disabled
Command added:
  -genv I_MPI_FABRICS shm:dapl
  -genv I_MPI_SHM_FBOX 0
Ignoring combination...
Ignoring fbox size: fbox disabled
Command added:
  -genv I_MPI_FABRICS shm:dapl
  -genv I_MPI_SHM_FBOX 0
Ignoring combination...
```

Once we've seen that the optimization is working properly (it is skipping the parameters and the result is the same as without the optimization) it's time to see the effects it has on the performance. In this case we can see that without the optimization it took 419.493 seconds while with the optimization it just took 233.606 seconds, this difference of time makes sense because with the original version we executed 16 experiments (of values of fbox size by to values of fbox) while with the execution with the optimization we only execute 9 experiments, the 8 with the fbox enabled and only one with the fbox disabled, so it's executing a bit more than the half of the experiments and takes a bit more that the half of time.

Obviously this case is not representative of the overall improvement since all the parameters we used were affected by the optimization, while in a real case we will have cases that are not affected by it, but it makes clear that skipping cases is a lot faster than executing them, so the effects of the optimization will be good.

8 Conclusions

Finally we can extract some conclusions from this work.

- Recognizing and verifying the Intel parameters, throwing an error if the parameter is incorrect.
- Checking the values of the parameters to make shure that they are valid.
- Added support to the MPI_ADJUST family of parameters, that were impossible to set up correctly using the supported values in the initial version.
- Ignoring parameters that don't affect the fabrics we are using.
- Making the optimization of removing redundant case also available to Intel Parameters.
- Improved the checksum algorithm used in the optimization, now it takes into acount the position of the letters so cases like *I_MPI_EAGER_THRESHOLD=1024;* and *I_MPI_EAGER_THRESHOLD=2410;* don't give the same result.

That said it's time to discuss some ways to continue improving the plugin. The most obvious one is adding the same support we now have for IBM MPI and Intel MPI to OpenMPI. Another thing that probably can be improved is the generation of the values for the MPI_ADJUST parameters, since it probably can be done in a simpler and faster way. Finally, as currently the automatic eager threshold setting is used for IBM, we can work on adapting this setting for Intel. This isn't done yet because in Intel we have more than one threshold to set, so it would need a more complex analysis of the application, but it is with no doubt an interesting feature to implement.

References

[1] A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque, Mate: Monitoring, analysis and tuning environment for parallel and distributed applications: Research articles, Concurr. Comput. : Pract. Exper. 19 (2007), no. 11, 1517–1531.

[2] Markus Geimer, Felix Wolf, Brian J. N. Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr, The Scalasca performance toolset architecture, Concurrency and Computation: Practice and Experience 22 (2010), no. 6, 702–719.

[3] M. Gerndt and M. Ott, Automatic performance analysis with periscope, Concurr. Comput. : Pract. Exper. 22 (2010), no. 6, 736–748.

[4] Sameer S. Shende and Allen D. Malony, The tau parallel performance system, Int. J. High Perform. Comput. Appl. 20 (2006), no. 2, 287–311.

[5] http://periscope.in.tum.de/

[6] https://ddd.uab.cat/pub/tfg/2016/tfg_39376/Extending_PTFs_MPI_Parameters_plugin_-_Alberto_Olmo.pdf

[7] https://software.intel.com/en-us/node/535547

[8] http://web.iitd.ac.in/~sumeet/flex__bison.pdf

[9] Automatic Tuning of HPC Applications - The Periscope Tuning Framework, by Michael Gerndt, Siegfried Benkner and Eduardo Cesar.

[10] https://software.intel.com/enus/articles/intelmpilibrarydocumentation

[11] http://aquamentus.com/flex_bison.html

Apèndix

In this appendix I'll put a list of all the Intel MPI parameters that are now supported by the plugin and the values they accept:

A General parameters

I_MPI_FABRICS	A fabric or two (an intranode and an internode one).
I_MPI_EAGER_THRESHOLD	A range.
I_MPI_INTRANODE_EAGER_THRESHOLD	A range.
I_MPI_INTRANODE_DIRECT_COPY	A yes/no value.
I_MPI_SPIN_COUNT	A range.
I_MPI_SCALABLE_OPTIMIZATION	A yes/no value.
I_MPI_WAIT_MODE	A yes/no value.
I_MPI_DYNAMIC_CONNECTION	A yes/no value.

B SHM parameters

I_MPI_SHM_CACHE_BYPASS	A yes/no value.
I_MPI_SHM_CACHE_BYPASS_THRESHOLDS	A range. Only valid if I_MPI_SHM_CACHE_BYPASS is enabled.
I_MPI_SHM_FBOX	A yes/no value.
I_MPI_SHM_FBOX_SIZE	A range, but only with values multiple of 64. Only valid if I_MPI_SHM_FBOX is enabled.
I_MPI_SHM_CELL_NUM	A range.
I_MPI_SHM_CELL_SIZE	A range. It also sets I_MPI_INTRANODE_EAGER_THRESHOLD.
I_MPI_SHM_LMT	It accepts shm, disable, no, 0 and off.
I_MPI_SHM_LMT_BUFFER_NUM	A range.
I_MPI_SHM_LMT_BUFFER_SIZE	A range.
I_MPI_SSHM	A yes/no value. I also sets I_MPI_INTRANODE_EAGER_THRESHOLD
I_MPI_SSHM_BUFFER_NUM	A range.
I_MPI_SSHM_BUFFER_SIZE	A range.
I_MPI_SSHM_DYNAMIC_CONNECTION	A yes/no value.
I_MPI_SHM_BYPASS	A yes/no value.
I_MPI_SHM_SPIN_COUNT	A range.

C DAPL parameters

I_MPI_DAPL_TRANSLATION_CACHE	A yes/no value.
I_MPI_DAPL_TRANSLATION_CACHE_AVL_TREE	A yes/no value.
I_MPI_DAPL_DIRECT_COPY_THRESHOLD	A range.
I_MPI_DAPL_EAGER_MESSAGE_AGGREGATION	A yes/no value.
I_MPI_DAPL_DYNAMIC_CONNECTION_MODE	Reject and disconnect.
I_MPI_DAPL_SCALABLE_PROGRESS	A yes/no value.
I_MPI_DAPL_BUFFER_NUM	A range.
I_MPI_DAPL_BUFFER_SIZE	A range.
I_MPI_DAPL_RNDV_BUFFER_ALIGNMENT	A range. The values should be powers of 2.
I_MPI_DAPL_RDMA_RNDV_WRITE	A yes/no value.
I_MPI_DAPL_CHECK_MAX_RDMA_SIZE	A yes/no value.
I_MPI_DAPL_MAX_MSG_SIZE	A range.
I_MPI_DAPL_CONN_EVD_SIZE	A range.
I_MPI_DAPL_SR_THRESHOLD	A range.
I_MPI_DAPL_SR_BUF_NUM	A range.
I_MPI_DAPL_RDMA_WRITE_IMM	A yes/no value.
I_MPI_DAPL_DESIRED_STATIC_CONNECTIONS_NUM	A range.

D DAPL-UD PARAMETERS

I_MPI_DAPL_UD	A yes/no value.
I_MPI_DAPL_UD_DIRECT_COPY_THRESHOLD	A range.
I_MPI_DAPL_UD_RECV_BUFFER_NUM	A range.
I_MPI_DAPL_UD_ACK_SEND_POOL_SIZE	A range.
I_MPI_DAPL_UD_ACK_RECV_POOL_SIZE	A range-
I_MPI_DAPL_UD_TRANSLATION_CACHE	A yes/no value.
I_MPI_DAPL_UD_TRANSLATION_CACHE_AVL_TREE	A yes/no value.
I_MPI_DAPL_UD_REQ_EVD_SIZE	A range.
I_MPI_DAPL_UD_CONN_EVD_SIZE	A range.
I_MPI_DAPL_UD_RECV_EVD_SIZE	A range.
I_MPI_DAPL_UD_RNDV_MAX_BLOCK_LEN	A range.
I_MPI_DAPL_UD_RNDV_BUFFER_ALIGNMENT	A range. The values should be powers of two.
I_MPI_DAPL_UD_RNDV_COPY_ALIGNMENT_THRESHOLD	A range. The values should be powers of two.
I_MPI_DAPL_UD_RNDV_DYNAMIC_CONNECTION	A yes/no value.
I_MPI_DAPL_UD_EAGER_DYNAMIC_CONNECTION	A yes/no value.
I_MPI_DAPL_UD_DESIRED_STATIC_CONNECTIONS_NUM	A range.
I_MPI_DAPL_UD_RDMA_MIXED	A yes/no value.
I_MPI_DAPL_UD_MAX_RDMA_SIZE	A range.
I_MPI_DAPL_UD_MAX_RDMA_DTOS	A range.

E TCP PARAMETERS

I_MPI_TCP_BUFFER_SIZE	A range.
I_MPI_TCP_POLLING_MODE	Poll, epoll, epoll:edge and epoll:level.

F OFA PARAMETERS

I_MPI_OFA_NUM_ADAPTERS	A range.
I_MPI_OFA_NUM_PORTS	A range.
I_MPI_OFA_NUM_RDMA_CONNECTIONS	A range.
I_MPI_OFA_SWITCHING_TO_RDMA	A range.
I_MPI_OFA_RAIL_SCHEDULER	ROUND_ROBIN, FIRST_RAIL and PROCESS_BIND.
I_MPI_OFA_TRANSLATION_CACHE	A yes/no value.
I_MPI_OFA_TRANSLATION_CACHE_AVL_TREE	A yes/no value.
I_MPI_OFA_USE_XRC	A yes/no value.
I_MPI_OFA_DYNAMIC_QPS	A yes/no value.
I_MPI_OFA_PACKET_SIZE	A range.

G COLLECTIVE OPERATION CONTROL

All the collective operation control parameters use the MPI_ADJUST style values except for the I_MPI_ADJUST_REDUCE_SEGMENT that has it's own values. The parameters are:

I_MPI_ADJUST_ALLGATHER
I_MPI_ADJUST_ALLGATHERV
I_MPI_ADJUST_ALLREDUCE
I_MPI_ADJUST_ALLTOALL
I_MPI_ADJUST_ALLTOALLV
I_MPI_ADJUST_ALLTOALLW
I_MPI_ADJUST_BARRIER
I_MPI_ADJUST_BCAST
I_MPI_ADJUST_EXSCAN
I_MPI_ADJUST_GATHER I_MPI_ADJUST_GATHERV
I_MPI_ADJUST_REDUCE_SCATTER

I_MPI_ADJUST_REDUCE
I_MPI_ADJUST_SCAN
I_MPI_ADJUST_SCATTER
I_MPI_ADJUST_SCATTERV
I_MPI_ADJUST_REDUCE_SEGMENT