Best Practice Guide

PTF Version: 1.1

AutoTune Partners

14.04.2015

Contents

1 Introduction

The Periscope Tuning Framework (PTF) is an extension to the existing performance analysis tool Periscope. PTF makes it possible to use profiling information gathered by Periscope and employ it for automatically tuning software applications.

There are five tuning plugins available as part of PTF, which target different tuning problems/techniques and which can be used to improve the performance and energy efficiency of a given application with one or more tuning techniques.

PTF and its associated tuning plugins provide many features to aid users in their effective exploitation. It is therefore useful to understand these features and techniques so as to be able to extract maximum benefit out of PTF.

This Best Practice Guide provides various ways to use each plugin more effectively as well as obtain tuning results more efficiently. In Section 2 of this guide, we explain how PTF fits into the overall tuning lifecycle. In Section 3, we provide several generic best practices applicable for all the plugins as well as the steps required for manual instrumentation. Finally, in Section 4, best practices specific to each of the separate plugins are provided.

2 PTF in the Tuning Cycle

The typical approach to software application tuning begins with selecting the baseline which is a version of the application used as a reference in all the further tuning steps. The process subsequently involves profiling the application to identify the performance bottlenecks of the application. Once identified, these bottlenecks are analyzed and a particular bottleneck is selected for tuning with the aim of improving the performance (e.g., time to solution) of the application. The tuning technique to be used on the bottleneck is identified and applied to the application and the tuning cycle begins again.

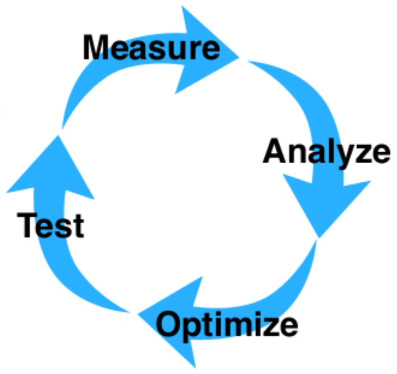


Figure 1: Application tuning cycle

When implemented manually, the tuning process is iterative, cumbersome and often very time consuming. PTF provides tuning plugins which can perform the application tuning automatically and remove this burden of manual tuning from the application developer. The tuning plugins explore the available tuning search space, evaluate different tuning scenarios and return the end results to the users without any human interventions. Along with automatic evaluation, PTF provides smarter ways to explore possible search combinations using various search strategies. For example, in particular cases, PTF can evaluate the optimal combination without complete execution of the application which reduces the tuning times drastically.

Use of automatic tuning contributes towards the overall productivity of software development. Additionally, due to codified expert knowledge in the form of PTF plugins it can be applied many times to different applications by users who may not be experts in application tuning.

3 Best Practice on how to use PTF: A Walkthrough

3.1 PTF best practices: the basics

In this section we will walk through the basics of how best to approach the instrumentation and tuning of applications with PTF in the context of an example application.

To begin our walk-through demonstration on how to best exploit PTF, we first choose to work with the BT-MZ benchmark. The BT benchmark is part of the NPB (NASA Parallel Benchmark) Suite [?] and contains the kernel for a Block Tri-diagonal solver. BT-MZ is a multi-zone version of the BT benchmark which is designed to exploit multiple levels of parallelism in the application. The BT-MZ application is written in FORTRAN with over 4.5k lines of code. The application is a serial application running on a single core, where there is a potential for performance optimization through using a combination of various compiler flags. As an application developer, we typically have some idea of the compiler flags that are important for performance improvements but not necessarily the optimal combination of the these flags. We also know that the incorrect combination of flags may actually degrade the performance of our application. The required manual experimentation of different flag combinations can be time consuming and cumbersome, so we would like a tool to automate the task of selecting the optimal combination of the compiler flags. The Compiler Flags Selection (CFS) plugin from PTF does exactly this and also provides some additional features, which we will walk through in the following sections.

3.1.1 Building the application with the PTF plugin

Using information in the CFS Plugin User's Guide [?] the BT-MZ benchmark application is compiled. From the CFS Plugin User's Guide we learn that instrumentation of the application is required so that PTF can collect performance measurements. In order to collect measurements, the compilation process is tweaked to use the `psc_instrument` command to perform instrumentation while compiling the application, as shown in Figure 2.

```
#---------------------------------------------------------------------------
# This is the fortran compiler used for fortran programs
#---------------------------------------------------------------------------
F77 = mpif90
F77 = psc_instrument -i -v -d -s ../bin/bt-mz.C.x.sir -t user mpif90
# This links fortran programs; usually the same as ${F77}
FLINK   = $(F77)
```

Figure 2: Change made to make.def file to add `psc_instrument` wrapper

In order to test each possible combination of compiler flags, PTF recompiles the application with each combination. PTF is informed about the compilation steps and arguments using the cfs_config.cfg file, as mentioned in the CFS Plugin User's Guide (available on the PTF website). A template for the configuration file is installed along with the PTF installation. A configuration file used for this example can be seen in Figure 3.

```
makefile_path="../";
makefile_flags_var="FFLAGS";
makefile_args="BT-MZ CLASS=C";
application_src_path="../BT-MZ";
selective_file_list="x_solve.f y_solve.f z_solve.f";
make_selective="true";

search_algorithm="exhaustive";

tp "TP_IFORT_OPT" = "-" ["O1", "O2", "O3"];
tp "TP_IFORT_XHOST"  = " " ["-xhost", " "];
```

Figure 3: An example of a configuration file for the CFS plugin.

3.1.2 Running the application with PTF for the first time

Subsequent to this build step, the BT-MZ application is executed using the PTF `psc_frontend` command as shown in the PTF User's Guide [**?**]. After successful execution, the optimal combination of compiler flags along with a summary of results for each combination is reported by the CFS plugin. For the example that we provide here, the search for the optimal combination of flags took approximately 4697 seconds. This overhead is expected because running PTF for this basic example involves recompilation and re-execution of the application with each combination of the selected compiler flags. Sample output of the PTF execution for the BT-MZ example discussed here is shown in Figure 4.

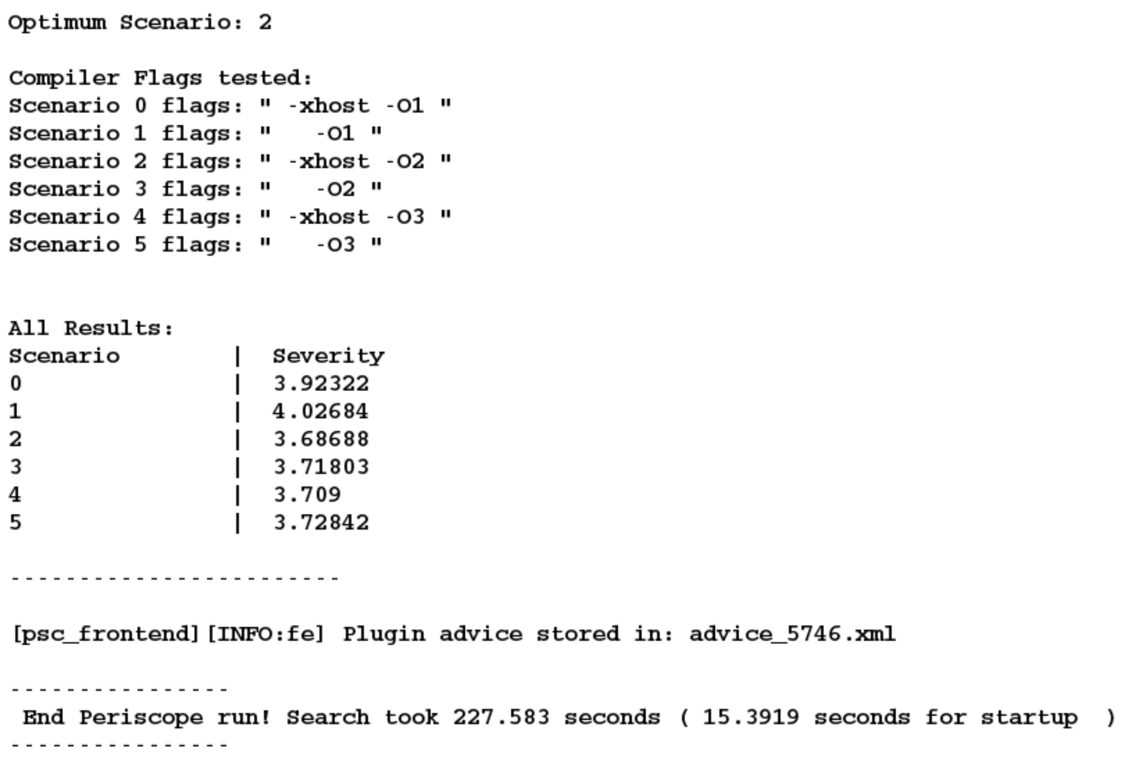
```
Optimum Scenario: 2

Compiler Flags tested:
Scenario 0 flags: "-O1  -xhost "
Scenario 1 flags: "-O1    "
Scenario 2 flags: "-O2  -xhost "
Scenario 3 flags: "-O2    "
Scenario 4 flags: "-O3  -xhost "
Scenario 5 flags: "-O3    "


All Results:
Scenario        |  Severity
0               |  805.32
1               |  837.48
2               |  660.455
3               |  751.322
4               |  676.317
5               |  752.343

-----------------------

[psc_frontend][INFO:fe] Plugin advice stored in: advice_6057.xml

---------------
 End Periscope run! Search took 4697.16 seconds ( 24.3193 seconds for startup  )
---------------
```

Figure 4: Sample output of PTF execution for the BT-MZ benchmark in naive mode.

3.1.3 Reducing the overall execution time using a phase region

The PTF User's Guide explains the concept of a *phase region*. Many HPC applications have a global progress loop, e.g., going through the simulated time steps. The execution of the loop body for a single time step is called a phase. The phase region, i.e., the loop body, can be marked for PTF in the source code as a *user region*. If the phase region is given, PTF will perform all experiments for a single iteration instead of executing the entire loop and thus, significantly reducing the tuning time. If no phase region is marked via a user region, the main program region is used as the default phase region.

In our example, BT-MZ has a main time-stepping loop with a trip count of 200, where the same type and amount of computation was executed for each iteration of the loop. A PTF user region can therefore be placed around the body of this computationally dominant loop. Only two pragma lines are required to be inserted into the BT-MZ source code, representing the start and end of the user region, respectively. For the BT-MZ example, this results in PTF executing only one iteration of the time-stepping loop rather than the full 200 iterations, reducing the execution time of PTF to approximately 227 seconds (relative to 4697 seconds without User Regions). Sample output from a PTF run using the phase region approach is shown in Figure 5.

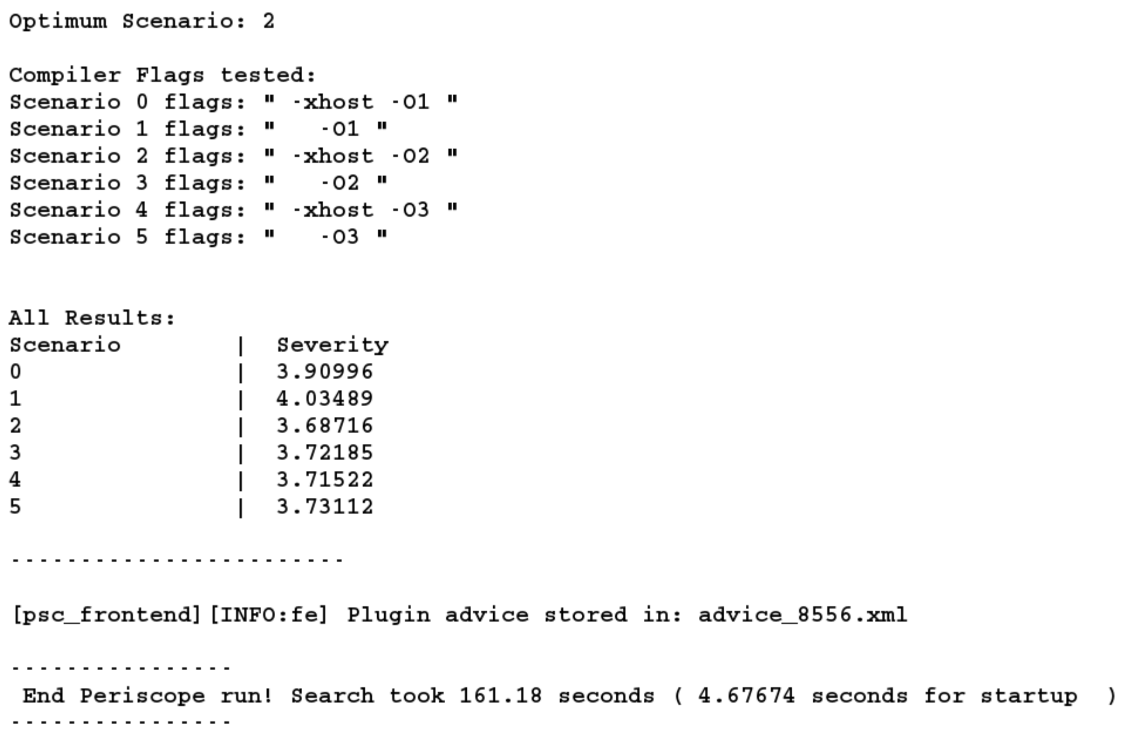


```
Optimum Scenario: 2

Compiler Flags tested:
Scenario 0 flags: " -xhost -O1 "
Scenario 1 flags: "   -O1 "
Scenario 2 flags: " -xhost -O2 "
Scenario 3 flags: "   -O2 "
Scenario 4 flags: " -xhost -O3 "
Scenario 5 flags: "   -O3 "


All Results:
Scenario        |  Severity
0               |  3.92322
1               |  4.02684
2               |  3.68688
3               |  3.71803
4               |  3.709
5               |  3.72842

------------------------

 [psc_frontend][INFO:fe] Plugin advice stored in: advice_5746.xml

----------------
 End Periscope run! Search took 227.583 seconds ( 15.3919 seconds for startup  )
----------------
```

Figure 5: CFS output for the BT-MZ application with a marked phase region and selective make enabled.

3.1.4 Reducing initialization time

The PTF User's Guide provides information on the "fast starter" flag which can reduce the amount of time spent in the initialization of PTF. For example, by passing `--starter=FastInteractive` to the `psc_frontend` command, the total execution time for PTF can be reduced further to approximately 161 seconds for the BT-MZ example described here (reducing the time spent in initialization time from approximately 20 seconds to 5 seconds. Sample output showing the effect of `--starter=FastInteractive` can be seen in Figure 6.



```
Optimum Scenario: 2

Compiler Flags tested:
Scenario 0 flags: " -xhost -O1 "
Scenario 1 flags: "   -O1 "
Scenario 2 flags: " -xhost -O2 "
Scenario 3 flags: "   -O2 "
Scenario 4 flags: " -xhost -O3 "
Scenario 5 flags: "   -O3 "


All Results:
Scenario        |  Severity
0               |  3.90996
1               |  4.03489
2               |  3.68716
3               |  3.72185
4               |  3.71522
5               |  3.73112

----------------------

[psc_frontend][INFO:fe] Plugin advice stored in: advice_8556.xml

---------------
 End Periscope run! Search took 161.18 seconds ( 4.67674 seconds for startup  )
---------------
```

Figure 6: Sample output for the BT-MZ appliation showing the reduction in time spent in initialization of PTF by using the FastInteractive flag.

3.2 Manual instrumentation

Manual instrumentation is an important technique to be exploited when dealing with codes that use complex language constructs and that are consequently difficult to instrument using the built-in `psc_instrument` command (it is often useful for C++ codes). In our second use case we will manually instrument a C++ application and run it using the CFS plugin. The following subsections will walk through each step involved in the manual instrumentation of the code and its significance. We instrument code in order to inform PTF about computationally intensive parts of a given code so as to steer the focus of PTF tuning efforts. These steps mainly involve identifying the computationally intensive parts of the code through profiling, and then marking them with specific PTF library calls . The build procedure is subsequently updated to link against the PTF libraries to make these functions available. Finally, a `SIR` file is created to convey manual instrumentation information to PTF.

3.2.1 Application background

By way of demonstrating best practices on manual instrumentation we will focus on the C++ LULESH (**L**ivermore **U**nstructured **L**agrangian **E**xplicit **S**hock **H**ydrodynamics) [**?**] code. LULESH is one of the proxy applications proposed by Lawrence Livermore National Lab (LLNL) as part of co-design efforts to address exascale challenges. By its nature, LULESH is a highly simplified application, hard-coded to only solve a simple Sedov blast problem with analytic answers. It represents the numerical algorithms, data motion, and programming style typical in HPC applications written in C or C++.

3.2.2 Finding the phase region

To identify the computationally intensive parts of LULESH we will profile the code using gprof by adding the `-pg` compiler flag. This requires updating the Makefile of the application. After executing the newly compiled binary we obtain the profiling data in the form of a binary file named `gmon.out`. Using a tool such `gprof` allows us see the amount of time spent in each function of our application. Figure 7 shows partial output from the gprof profile of LULESH running in sequential mode, showing that the majority of time during execution was spent in the `LagrangeLeapFrog` function.

```
-----------------------------------------------
                0.00    5.31     100/100         main [1]
[2]    100.0    0.00    5.31     100         LagrangeLeapFrog(Domain&) [2]
                0.01    3.68     100/100         LagrangeElements(Domain&, int) [3]
                0.99    0.00     100/100         CalcElemVolumeDerivative(double*,
```

Figure 7: Partial output from `gprof` showing hotspot functions in LULESH.

Inspecting the source code reveals that the `LagrangeLeapFrog` function is called within an iterative time-stepping loop (a so-called“ phase region”). Figure 8 shows a segment of LULESH source code from the `lulesh.cc` file, containing the while loop which iterates over a given number of time-steps.

```
   // BEGIN timestep to solution */
#if USE_MPI
   double start = MPI_Wtime();
#else
   timeval start;
   gettimeofday(&start, NULL) ;
#endif

   while((locDom->time() < locDom->stoptime()) && (locDom->cycle() < opts.its)) {

      TimeIncrement(*locDom) ;
      LagrangeLeapFrog(*locDom) ;

      if ((opts.showProg != 0) && (opts.quiet == 0) && (myRank == 0)) {
         printf("cycle = %d, time = %e, dt=%e\n",
                locDom->cycle(), double(locDom->time()), double(locDom->deltatime()) ) ;
      }
   }

   // Use reduced max elapsed time
   double elapsed_time;
#if USE_MPI
   elapsed_time = MPI_Wtime() - start;
#else
```

Figure 8: The compute intensive while loop interating over the timesteps and calling the `LagrangeLeapFrog` function in LULESH.

At this stage, we will convey this information about the computationally intensive part of the code to PTF by placing the user region around this segment of the code.

3.2.3 Marking the PTF region in the source code

To start with the instrumentation we need to mark the beginning and end of the application. Figure 9 shows how we mark the beginning of an application using the PTF region, and Figure 10 shows the end of PTF region.

The following guidelines need to be followed when marking the start of a PTF region in the example code we work with here (LULESH):

- Firstly, we have to include a header file `mrimonitor.h`.
- Then we need to find the `main` function of the application and declare some of PTF-specific variables.
- Next we call the `startMonLib()` function to start a monitoring library.
- We then insert a call to the `startRegion()` function with the third argument as the line number where the call to start a monitoring library is made.

- Finally, we put the content of the entire `main` function into the code block surrounded by the curly braces.

```
2692
2693 int main(int argc, char *argv[])
2694 {
2695 //Periscope specific
2696  int psc_ret_val;
2697  int pscOldTaskId=0;
2698  startMonLib();
2699
2700  //Marking start of main region
2701  startRegion(1,1,2698,0,-1);
```

Figure 9: The `main` function is marked with the start of a PTF region.

The following guidelines need to be followed while marking the end of a PTF region.

- Assign a zero value to `psc_ret_val` variable.
- Make a call to `endRegion()` function with the same arguments as that of the `startRegion()` function.
- Stop the monitoring library by calling `stopMonLib()` function.
- Place these calls before the `MPI_Finalize()` call.

```
    psc_ret_val = 0;
  }
  //Marking end of main Region
  endRegion(1,1,2698,0,-1);
  //Stopping the monitoring library
  stopMonLib();

#if USE_MPI
    MPI_Finalize() ;
#endif

  return (psc_ret_val);
}
```

Figure 10: The end of the PTF region is marked with in the `main` function.

3.2.4 Marking a user region in the source code

A user region can be used to mark the phase region of the application. It then identifies a part of the code where we want PTF to focus it's optimization efforts. User region pragmas are placed around computationally intensive code sections. In the case of a computationally intensive loop traversing a high trip count, performance of a single iteration of the loop for a particular optimization can often be representive of the overall performance of an application for that optimization. When such a loop is marked as a User Region, PTF executes the loop for only a single iteration to measure the performance improvements.

```
while((locDom->time() < locDom->stoptime()) && (locDom->cycle() < opts.its)) {

  startRegion(27,1,2700,0,-1);
  {
      TimeIncrement(*locDom) ;
      LagrangeLeapFrog(*locDom) ;
      if ((opts.showProg != 0) && (opts.quiet == 0) && (myRank == 0)) {
        printf("cycle = %d, time = %e, dt=%e\n",
                       locDom->cycle(), double(locDom->time()),
                       double(locDom->deltatime()) ) ;
      }
  }
  endRegion(27,1,2700,0,-1);

 }
```

Figure 11: Instrumented version of the timestep loop.

Figure 11 shows the updated time-stepping loop. The following guidelines need to be followed when marking code with a user region.

- Call `startRegion()` and `endRegion()` functions at the beginning and end of the loop. Make a note that both the function calls are inside the loop.
- The third parameter in the function is the line number of the statement before the outer `startRegion()` call.
- The user region marking the phase region entails a collective synchronization. PTF configures application monitoring in `startRegion()` and `endRegion()`. This configuration leads to a barrier synchronization.

3.2.5 Updating makefile to link against PTF libraries

When instrumenting the code we include the `mrimonitor.h` header file and insert function calls `startRegion` and `endRegion()` to mark the region to be instrumented. While compiling this manually instrumented code we need to inform the compiler about the location of the header file to include it, and also inform the linker about the name and location of the libraries containing the necessary function calls.

```
PSC_CXXFLAGS =  -I$(HOME)/install/Periscope/include/
MRI_LDFLAGS = -L$(HOME)/install/Periscope/lib -lmrimon -lmpiprofiler \
                -lpscreg -lpscutil -lqualexpr -lm
PAPI_LDFLAGS = -L/lrz/sys/tools/papi/5.0.0/lib -lpapi
PSC_LDFLAGS = $(MRI_LDFLAGS) $(PAPI_LDFLAGS)

#Default build suggestions with OpenMP for g++
CXXFLAGS = -g -O3 -fopenmp -I. $(PSC_CXXFLAGS) $(DEVEL_INC) -Wall
LDFLAGS = -g -O3 -fopenmp $(PSC_LDFLAGS)
```

Figure 12: Updated `Makefile` to find PTF header files and link the PTF libraries.

In Figure 12 we show an updated `Makefile` for LULESH. Relevant changes include -

- The path for the PTF headers is provided to compiler by adding to `CXXFLAGS`.
- The path and name of the PTF libraries are provided using `LDFLAGS` variable.
- Additional libraries are pointed to by aggregating them into the `PSC_LDFLAGS` variable.

3.2.6 Generating a `SIR` File

By way of a `SIR` file we can inform PTF about the outer and User Regions generated during the manual or automatic instrumentation, with an example `SIR` file shown in Figure 13 .

```
<?xml version="1.0" encoding="UTF-8"?>
<sir xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http:
//www.lrr.in.tum.de/Periscope" xsi:schemaLocation="http://www.lrr.in.tu
m.de/Periscope psc_properties.xsd" language="C">
<!-- ./prep/lulesh.sir -->
  <unit type="program" name="int main(int, char **) C" id="1-2698">
    <position startLine="2698" endLine="2823">
      <file name="./prep/lulesh.c"/>
    </position>
    <codeRegion type="userRegion" name="" id="1-2700">
      <position startLine="2700" endLine="2824">
        <file name="./prep/lulesh.c"/>
      </position>
    </codeRegion>
  </unit>
</sir>
```

Figure 13: `SIR` file

To generate a `SIR` file one can follow template or any existing `SIR` file and update it using the guidelines below to suit a given application.

- Update the `id` field of a unit tag, to 1-(line number of a line where the call to `startMonLib()` is made). The id of a region is composed of the two numbers, you can choose both but unique.
- The `position` tag under the unit tag should contain `startLine` as a line number specified in the outer tag after 1 and `endLine` should be the line where the outer region ends.
- The `file` tag should contain the value `name` in the `./<file_name>.c`. The name of the file containing the outer region should go here.
- The `codeRegion` should have `id` as 1-(line number of a line where the call to `startRegion()` is made) and `type` should be `userRegion`.
- The `position` and `file` tag should follow the same format as the `position` tag used above.

3.2.7 Executing the instrumented application with PTF

Once the instrumentation is completed, the execution of the application is carried out in the same way as of an application automatically instrumented with the `psc_instrument` command. In this case, we use the following commands:

```
module load gcc ace/6.1 papi
module load boost/1.45 gcc mpi.intel

psc_frontend --apprun="./lulesh2.0" --sir="./lulesh.sir"
             --mpinumprocs=8 --tune=compilerflags
```

The `module load` commands configure an environment on the SuperMUC system with paths for the necessary libraries and the `psc_frontend` command is used to start the tuning running the CFS plugin. Here, we have used the same configuration file `cfs_config.cfg` as that used for the BT-MZ use case.

3.2.8 Improving the accuracy of PTF results

In the case of an instrumented applications with a single iteration of a computationally intensive loop executing for a short amount of time, tuning results may not be consistent. This is mainly due to OS noise or the resolution of the instrumentation. In such cases we can modify the code by increasing the number of iterations to be instrumented in order to obtain consistent results. This can also be achieved by loop splitting. Figure 14 shows the version of LULESH code shown in Figure11 after splitting the loop into two loops. In the new version, a user region is placed around the inner loop which runs for 1000 iterations.

```
           startRegion(27,1,2700,0,-1);
           {
              for(int innerCounter=0; innerCounter<1000; innerCounter++)
              {
                 TimeIncrement(*locDom) ;
                 LagrangeLeapFrog(*locDom) ;
                 if ((opts.showProg != 0) && (opts.quiet == 0) && (myRank == 0)) {
                         printf("cycle = %d, time = %e, dt=%e\n",
                                 locDom->cycle(), double(locDom->time()),
                                 double(locDom->deltatime()) ) ;
                 }
              }
           }
           endRegion(27,1,2700,0,-1);
    }
```

Figure 14: Splitting the loop to execute more iterations in the User Region in LULESH.

3.2.9 Results for LULESH

- **CFS Plugin**

  Figure 15 demonstrates that the combination of the compiler flags suggested by PTF results in 9.5% improvement in the overall execution time of LULESH.

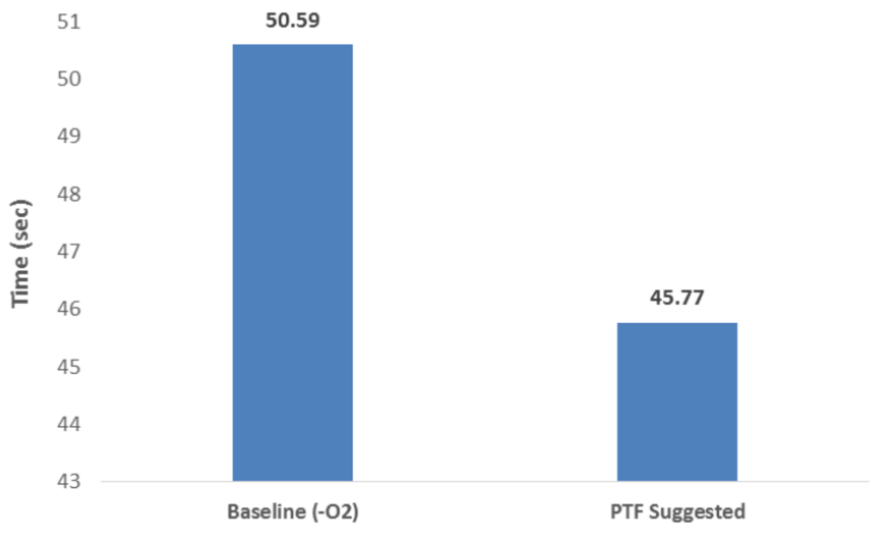


Figure 15: Execution time with the optimal compiler flags suggested by PTF and the baseline reading taken without using PTF.

3.2.10 Additional tips

- It is recommended to add a delay of one or more executions of the phase region in the `psc_frontend` command using the `--delay` flag to avoid the warm up phase of the application.
- Check the reproducibility of the measurements.

4 Best Practice on how to use the PTF Tuning Plugins

In the previous section we focused on walking through two examples of how to best employ PTF on sample applications with the CFS plugin. In this section we provide an overview of the features of each of the PTF plugins which can help to employ PTF and the plugins more effectively as part of a typical PTF tuning workflow.

4.1 Compiler Flags Selection Plugin

By way of guiding the user on how to best exploit the CFS plugin, we will focus on two applications from the CESAR suite of exascale proxy applications [**?**], NEKBONE and MOCFE. The NEKBONE proxy application is a thermal hydraulics simulation code for reactor simulations that solves the Poisson equation, Helmholz equation, as well as other differential equations. It is an MPI-based FORTRAN application consisting of 13 files. For the example discussed here, the application was executed with 8 processes and 50 elements per process, a polynomial order of 10, and without the multi-grid preconditioner (i.e. 'Example 3' in the provided source code).

The MOCFE application simulates the main procedures in a 3D method of characteristics (MOC) code for numerical solution of the steady state neutron transport equation. 3D-MOC features heterogeneous geometry capability, high degree of accuracy, and potential for scalability. It is a FORTRAN 90 MPI application consisting of 36 files. We have instrumented the inner loop of `Method_FGMRES.F90` as a phase_region of the application. For the example cases described here MOCFE was executed with 16 processes and a Krylov iteration size of 60.

4.1.1 Using different search algorithms

The CFS plugin searches for the best possible combination of compiler flags from a list of flags provided in the `cfs_config.cfg` file of the plugin. It uses one of five search algorithms, i.e., *Exhaustive Search*, *Random Search*, *Individual Search*, *Genetic Search*, and *Machine Learning-based Random Search* for selecting the best combination of flags. In general, the Individual Search strategy provides good results and is recommended as a good starting point. The Genetic Search and Random Search algorithms can be used in combination with Machine Learning to perform a broader search.

The search algorithms of the CFS plugin are described below:

- *Exhaustive Search (ES):* The plugin compiles and executes the application with all combinations of the specified compiler flags.
- *Random Search (RS):* The plugin randomly selects combinations of flags.
- *Individual (IS):* This algorithm adds-up each compiler flag one after the other from the tuning parameter list in the order used in the specification file.

- *Genetic Search (GS):* GS searches for the combination of compiler flags based on the GDE3 genetic algorithm.
- *Machine learning based Random Search (MLRS):* MLRS uses the Tuning Database of PTF, which is a collection of CFS tuning results for different applications, to model and predict the suitable compiler flags for the current application. The database, in our case, was filled with the NAS parallel benchmark experiments.

For the example cases discussed here, we can run all of these algorithms (apart from ES due to the huge number of scenarios) on the NEKBONE application. In the case of the NEKBONE application, the body of the progress loop in the NEKBONE application was marked as a User Region and we also use the following CFS plugin specific features: i) the *Selective Make* option of the CFS plugin to recompile only compute intensive files instead of the entire application and ii) the *Remote Make* option to perform the compilation on the login nodes of SuperMUC, the supercomputer at LRZ that we run our use cases on. For the use-cases described here the compiler flags were pre-defined in the `cfs_config.cfg` file of the CFS plugin.

We focus on the following FORTRAN compiler flags of Intel's `ifort` compiler when auto-tuning NEKBONE using the CFS plugin:

```
tp "TP_IFORT_OPT" = "-" ["O2", "O3", "O4"];
tp "TP_IFORT_XHOST"  = " " ["-xhost", " "];
tp "TP_IFORT_UNROLL" = " " ["-unroll", " "];
tp "TP_IFORT_VERSION" = " " ["-opt-multi-version-aggressive", " "];
tp "TP_IFORT_FMA" = " " ["-fma", " "];
tp "TP_IFORT_INLINE" = " " ["-finline-functions","-fno-inline-functions"];
tp "TP_IFORT_PREFETCH" = "-opt-prefetch=" [1,4,1];
tp "TP_IFORT_UNROLL" = "-unroll" [1,16,4];
tp "TP_IFORT_OPTBLOCK" = "-opt-block-factor=" [1,3,1];
tp "TP_IFORT_STREAM" = " " ["-opt-streaming-stores always",
          "-opt-streaming-stores never", "-opt-streaming-stores auto"];
tp "TP_IFORT_IP" = " " ["-ip", " "];
```

Figure 16 compares the execution times for the different search algorithms for the NEKBONE application for a given number of experiments.

It can be seen that all three search algorithms converge to approximately the same execution time for the instrumented application (0.38 seconds). While the IS algorithm results in the best execution time after only five experiments, the RS algorithm requires 22 experiments and the GS algorithm requires 80 experiments to obtain the shortest execution time for the instrumented application, thus demonstrating the potential advantages of some search algorithms over others when using the CFS plugin. As mentioned, the Individual Search strategy generally provides good results and is recommended as a good starting point when employing the CFS plugin.

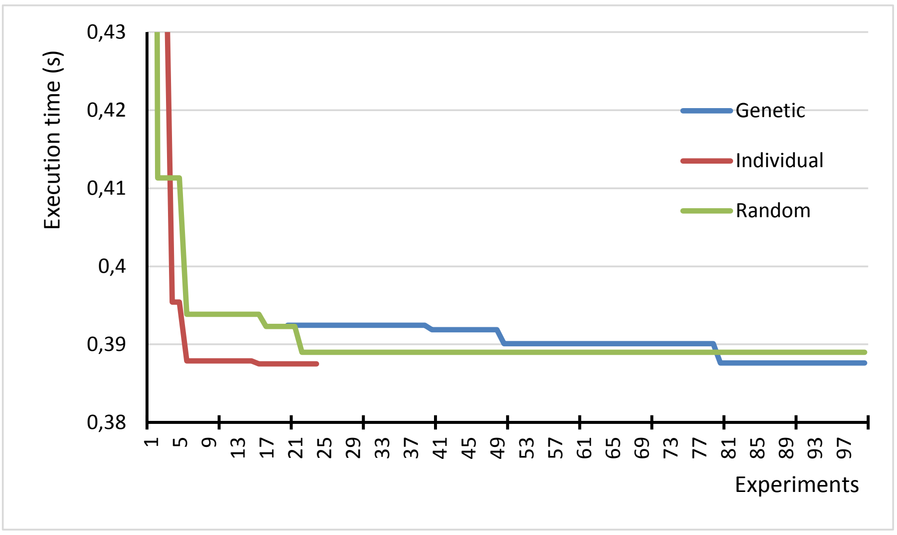


Figure 16: Comparison of IS, GS, and RS for NEKBONE.

	Full Compile		Selective Compile	
Code	#Files	Time (s)	#Files	Time (s)
MOCFE	36	32.12	4	4.503
NEKBONE	13	45.16	1	2.22

Table 1: Compilation time for the whole application relative to compilation time selecting only the most significant application files.

4.1.2 Reducing the compilation time for selected flag combinations

The *Selective Make* feature of the CFS plugin allows users to reduce the compilation time for a selected combination of flags by recompiling only significant target application files, i.e., files that account for most of the execution time. To demonstrate the impact of this feature here, we apply the CFS plugin (with Individual Search) to the NEKBONE and MOCFE applications . As can be seen from Table 1, the compilation time for the applications differs significantly between recompiling the whole application and recompiling only the significant files using Selective Make. Indeed, for applications composed of many files the Selective Make feature generally leads to a significant reduction in the overhead of searching through CFS parameter spaces and is recommended to CFS plugin users to improve productivity.

Table 2 compares the results running MOCFE and NEKBONE with the full compilation and the compilation using Selective Make. The results demonstrate that selective

	Full Compile		Selective Compile	
Code	Search (s)	Exec (s)	Search (s)	Exec (s)
MOCFE	1498	8.4	855	8.3
NEKBONE	1950	0.37	914	0.38

Table 2: Comparison of search time and resultant execution time for selective compilation.

Scenario	Total	x_solve	y_solve	z_solve	compute_rhs
0	4.04	1.14	1.16	1.26	0.42
1	4.03	1.13	1.15	1.25	0.42
2	4.02	1.13	1.15	1.25	0.42
3	3.94	1.10	1.12	1.20	0.44
4	3.93	1.10	1.13	1.19	0.44

Table 3: Tuning results for individual subroutines in NPB BT-MZ.

compilation leads to a significant reduction of the search time while the results vary only slightly, since the most time consuming routines were recompiled anyway.

4.1.3 Measuring significant routines

The CFS plugin also allows users to determine the best flag combinations per file. This is based on the identification of routines that take up the majority of computation time in each file selected for selective compilation, where the effect of the compiler flags on the execution of these routines is used to identify the file-specific best flag combination. To demonstrate this feature, we focus on the BT-MZ application as used in section 2 of this guide and run the application with 4 MPI processes and problem size C from the BT-MZ test cases. We use Machine Learning based Random Search with 5 samples. The results presented in Table 3 were achieved for the five samples:

The following flags were tested in the scenarios:

1. `-O3 -xhost -fma -fno-inline-functions -opt-prefetch=4 -unroll13 -opt-block-factor=1 -opt-streaming-stores never`
2. `-O3 -xhost -fma -finline-functions -opt-prefetch=4 -unroll13 -opt-block-factor=1 -opt-streaming-stores auto`
3. `-O3 -xhost -fma -finline-functions -opt-prefetch=3 -unroll13 -opt-block-factor=3 -opt-streaming-stores auto`
4. `-O3 -xhost -fma -finline-functions -opt-prefetch=1 -unroll1 -opt-block-factor=3 -opt-streaming-stores always`

5. `-O3 -xhost -fma -fno-inline-functions -opt-prefetch=2 -unroll1 -opt-block-factor=3 -opt-streaming-stores always -ip`

Subroutines `x_solve`, `y_solve`, and `z_solve` are very similar and the best configurations were found to be scenario 4 for `x_solve` and `z_solve`. A very small time difference was reported for scenarios 3 and 4 for `y_solve`. These scenarios are almost identical and thus we can summarize that all three routines work well with scenario 4. For subroutine `compute_rhs` a more significant difference can be seen for scenarios 0-2 and 3-4. Therefore, the plugin recommends to compile file `rhs.f` with the flag combination from scenario 1. The global optimum is scenario 4 since the all three `solve` routines are almost three times as time consuming as `compute_rhs`.

4.1.4 Remote Make

Finally, compute nodes of HPC systems often have a light-weight version of Linux and may not have all the necessary files to recompile the application. In order to recompile the application with the different flags combinations, the CFS plugin supports remote building of the application. The compilation is then run on, for example, a login node via `ssh`. Please consult the CFS Plugin User's Guide on how to configure the CFS plugin to use `remote make`.

4.2 Parallel Patterns Plugin

By way of guiding the user on how to best exploit the Parallel Patterns plugin, we will focus on the FaceDetect application [**?**]. The FaceDetect application uses an image-processing pipeline built on top of the PEPPHER framework. Within PEPPHER, pipeline patterns are expressed using annotated while-loops that comprise calls to multi-architectural components. The high-level FaceDetect code is shown in Figure 17. The application uses a four stage pipeline to perform detection of human faces on a stream of images.

The Parallel Patterns plugin for PTF supports automatic performance tuning of high-level pipeline patterns built on top of the PEPPHER framework. The plugin searches for the best combination of pattern-specific parameters, parameters exposed by the runtime system, and machine-specific parameters such that execution is optimized for a given workload and target architecture.

The plugin for Parallel Patterns allows for the restriction of a potentially large search space via:

- tuning range specifications for stage replication factors and stage buffer sizes by means of source code directives
- tuning range specifications for all tuning parameters via configuration file
- performance analysis to focus tuning on the most time-consuming stage

```
#pragma pph pipeline
while ( inputstream >> file ) {
    ReadImage ( file , image );
    #pragma pph stage replicate(?)  buffer(?)
    ResizeAndColorConvert ( image, oimage );
    #pragma pph stage replicate(?)  buffer(?)
    DetectFaces( oimage );
    WriteFaceDetectedImage ( file , oimage );
}
```

Figure 17: A pipeline pattern for face detection in a stream of images. The ”?” symbols within the annotations indicate that the tuning ranges for `ResizeAndColorConvert` and `DetectFaces` stages should be automatically determined.

4.2.1 Default behavior

The total number of tuning scenarios in the search space is created as a crossproduct of the value ranges of all available tuning parameters. By default, for each of the tuning parameters the tuning range of possible values is determined by the system as follows:

- for each stage replication factor that has been annotated with the "?" in the high-level code, the range of the stage replication factor is set to [1:(*max_execution_units - number_of_pipeline_stages - 1*):1]
- for each buffer size, that has been annotated with the "?" in the high-level code, the range of the buffer size is set to [*max_execution_units*:*max_execution_units**3:*max_execution_units*]
- the NCPUS range is set to [1:*max_cpu_cores*:1]
- the NGPUS range is set to [0:*max_gpus*:1]
- the SCHEDULING_POLICY is set to two scheduling policies EAGER and HEFT[1]

These default values are based on the runtime system requirements and on the application testing experience. To minimize oversubscription, the total number of stages plus stage replicas is set to not exceed the number of execution units on the system. The buffer size influences the memory footprint of the application. For each stage, the buffer size is set to have minimum equal to the replication factor of that stage, and to have maximum of that value multiplied by 3.

Tuning the pipeline on the system with many execution units may require an exploration of a huge search space. For instance, on the system with 16 CPU cores, and 4 GPUs, a full exhaustive search for the FaceDetect application would require an evaluation of 17*3*17*3*16*5*2 = 416,160 scenarios. In Table 4 we summarize the possible values for each tuning parameter in this setup.

Tuning parameter	**Possible values**
`ResizeAndColorConvert` Replication Factor	1, 2, 3,.., 17
`ResizeAndColorConvert` Buffer Size	20, 40, 60
`DetectFaces` Replication Factor	1, 2, 3,.., 17
`DetectFaces` Buffer Size	20, 40, 60
Number of CPU cores	1, 2, 3,.., 16
Number of GPUs	0, 1, 2, 3, 4
Scheduling Policy	"EAGER", "HEFT"

Table 4: Tuning parameters and their values on the PHIA system for the setup with no restrictions.

The following three subsections demonstrate how to efficiently restrict the search space, and yet test majority of the performance-relevant scenarios.

[1]HEFT (Heterogeneous Earliest Finish Time) considers inter-component data dependencies and schedules components to workers taking into account the current system load, available component implementation variants, and historical execution profiles, with the goal of minimizing overall execution time by favoring implementations variants with the lowest expected execution time. EAGER is a simply greedy scheduler.

4.2.2 Tuning range specification via directives

The tuning range source code directives allow users to steer the tuning process of the plugin by specifying values for the stage replication factors and buffer sizes. Careful specification of the tuning ranges for such parameters in the high-level code may reduce the total number of scenarios that need to be tested.

For the FaceDetect application tuning, we used these directives to provide tuning ranges for the replication factors of the two middle stages in the pipeline. As shown in Figure 18, the user specified tuning range starts with the minimum value of 1, which is incremented by 4 until it reaches the maximum value of 17, resulting in a total of 5 values for each tuning parameter. The minimum value of 1 will reveal if incrementing the replication factor for the certain stage has any effect on the performance. For the maximum value, it is usually good to keep the default of *max_execution_units - number_of_pipeline_stages - 1*. This will ensure that oversubscription is avoided. Finally, a good guideline for the increment is actual number of GPUs in the system, since the runtime system needs one CPU core for each GPU card it uses.

```
#pragma pph pipeline
while ( inputstream >> file ) {
    ReadImage ( file , image );
    #pragma pph stage replicate(1:17:4)
    ResizeAndColorConvert ( image, oimage );
    #pragma pph stage replicate(1:17:4)
    DetectFaces( oimage );
    WriteFaceDetectedImage ( file , oimage );
}
```

Figure 18: User-provided tuning hints. For the middle stages the tuning ranges for the stage replication factor is specified in the form `(min:max:step)`.

By specifying tuning ranges for stage replication factors as shown in Figure 18, the search space is reduced from 416,160 to 36,000 scenarios.

4.2.3 Tuning range specification via configuration file

Another option for enabling users to restrict the search space is by specifying tuning ranges in the SIR file for some or all tuning parameters (including NCPUS, NGPUS). In the configuration file, it is a good idea to provide tuning ranges for machine-specific tuning parameters, i.e., NCPUS and NGPUS. For instance, NCPUS parameter may be restricted in a way that the minimum number of CPU cores to be used for execution is equal to the number of pipeline stages, and the maximum is equal to the number of available CPU cores with an increment of 4 (see Figure 19). As a consequence only 4 values for the NCPUS tuning parameter are considered, further restricting the search space to 9,000 scenarios. The tuning parameters and corresponding values that describe this restricted search space are summarized in Table 5.

```
...
  <selector tuningActionType="VAR" tuningActionName="NCPUS" min="4" max="16" step="4"/>
...
```

Figure 19: In order to influence the tuning process the SIR file may be altered manually by changing the corresponding *min max* and *step* values for the desired tuning parameter.

Tuning parameter	**Possible values**
`ResizeAndColorConvert` Replication Factor	1, 5, 9, 13, 17
`ResizeAndColorConvert` Buffer Size	20, 40, 60
`DetectFaces` Replication Factor	1, 5, 9, 13, 17
`DetectFaces` Buffer Size	20, 40, 60
Number of CPU cores	4, 8, 12, 16
Number of GPUs	0, 1, 2, 3, 4
Scheduling Policy	“EAGER”, “HEFT”

Table 5: Tuning parameters and restricted values on the PHIA system.

4.2.4 Focused tuning via performance analysis

Focused tuning of pipeline patterns via PTF pre-analysis may be enabled via `--vpattern-focused` command-line switch. In this mode, the plugin uses PTF's pre-analysis to detect the most performance-demanding stage in the pipeline. Once it has been detected, the tuning efforts are focused on the stage replication factor and buffer size of that stage.

The focused tuning can be used without any additional user-provided hints. In the standalone mode it reduces the search space from initial 416,160 to 8,100. A combination of focused tuning with user-provided tuning ranges may be used to further decrease the search space. For the FaceDetect application the stage replication factor for the most performance-demanding stage (`DetectFaces`), is automatically set to [1:17:1], while for all other stages tuning of stage replication factors (and buffer sizes) is omitted. As a result, the resulting scenario pool contains only 600 scenarios, significantly reduced from the initial total of 416,160 scenarios.

Figure 20 shows the effect of the different plugin features on the size of the search space. The first blue bar shows the number of scenarios when source code directives were used to restrict the search space from 416,160 to 36,000. The second blue bar represents additional restriction of the search space by adjusting tuning ranges directly in the configuration file. The third blue bar shows the number scenarios when only focused tuning via pre-analysis was used. Finally, the fourth blue bar shows the number of the scenarios when all features were used.

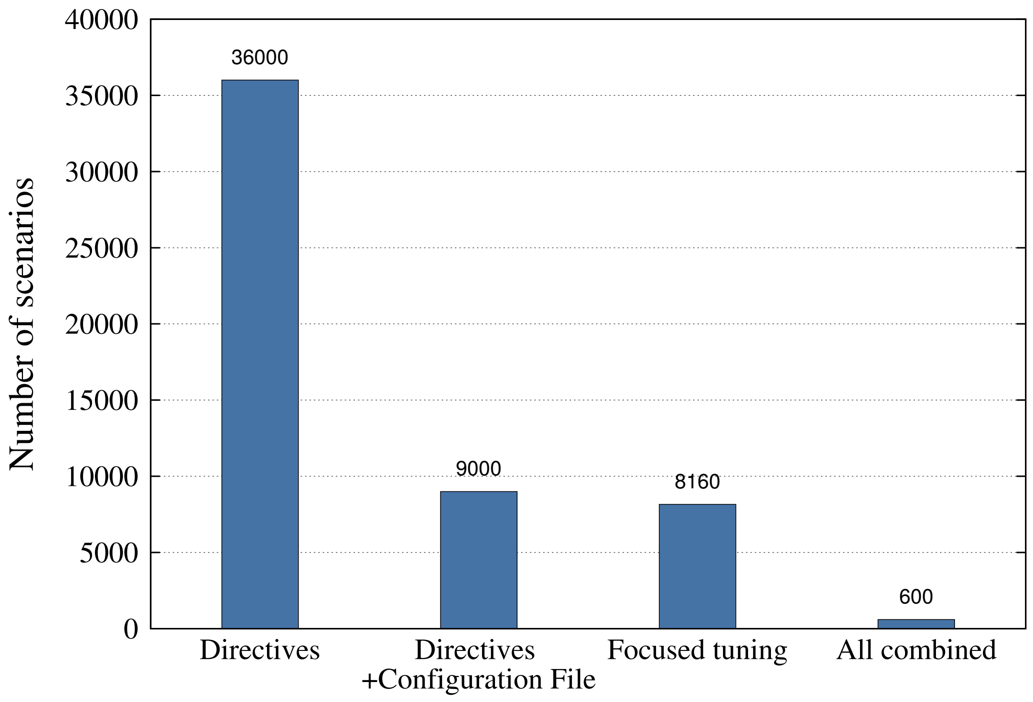


Figure 20: Restricting the search space for the FaceDetect application using different plugin features of the Parallel Patterns plugin.

4.2.5 Additional tips

- Stage Replication Factors

  Stage replication factors determine the number of stage instances executed in parallel and therefore influence the degree of potential parallelism during application execution. Each stage replica will execute in an additional thread. To avoid oversubscription, the total number of stages plus stage replicas should not exceed the number of execution units on the system. On the other hand, if the replication factors are too low, the performance might be limited, so a good guideline is to increase stage replication factors for the stages that have higher performance demands.

- Machine-specific Parameters

  The machine-specific parameters (NCPUS and NGPUS) are closely related to stage replication factors. The effective usage of system execution units may be hindered if the stage replication factors are not sufficiently high. On the other hand, if the total number of available execution units is small, the replication factor will have no effect.

4.3 Dynamic Voltage Frequency Scaling (DVFS) Plugin

As an example of how to best employ the features of the DVFS plugin we have chosen the SeisSol application [?] as our use case here. The SeisSol application is developed by the Department of Earth and Environmental Sciences at the Ludwig-Maximilian University. It is a MPI application written in Fortran 90. The application is used for simulating realistic earthquake scenarios, accounting for a variety of geophysical processes that affect the propagation of seismic waves, e.g. viscoelastic attenuation, strong material heterogeneities and anisotropy. SeisSol contains a computation kernel which simulates the specified number of time steps.

4.3.1 Extending the search space

The objective of the DVFS plugin is to tune the energy consumption of an application. The DVFS plugin can be configured with the number of frequencies that the plugin should use for the search space. The models predict one frequency according to the tuning objective. The number of neighboring frequencies (neighbors of the predicted frequencies) to be analyzed can be set. Environment variable PSC_FREQ_NEIGHBORS is used for this purpose. The value of this variable specifies the number of neighbors to the right and to the left of the predicted frequency. The maximum value for this variable is limited to 7 neighbors. If the value exceeds 7 neighbors or if it is negative then the plugin will use the default value of 1 neighbor on each side. The user is advised to use a higher number of neighbors when the execution time does not play a role, i.e., when the tuning is completed in an acceptable time-frame for the user. If it is suspected that the model delivers inaccurate predictions, a higher number of frequencies can help mitigate this problem.

Figure 21 shows the sample output of the DVFS plugin for the SeisSol application with PSC_FREQ_NEIGHBORS=3. The tuning objective was to minimize the Energy Delay Product (EDP).

```
Found Optimum Scenario:3          Frequency: 2700
Search Path:
Scenario  | Governor       | Freq (MHz)    | Energy (J)    | Runtime (s)   | EDP
----------+----------------+---------------+---------------+---------------+-----------
0         | Userspace      | 2400          | 2681.000      | 13.023        | 34914.931
1         | Userspace      | 2500          | 2762.000      | 12.611        | 34831.306
2         | Userspace      | 2600          | 2841.000      | 12.065        | 34275.813
3         | Userspace      | 2700          | 2940.000      | 11.624        | 34174.854
```

Figure 21: Output of the DVFS plugin for inspecting the predicted and three neighbor frequencies.

The output shows that the three neighbors below the predicted CPU frequency of 2.7 GHz are measured compared to the default of one neighbor as shown in Figure23. Since 2.7GHz is the highest CPU frequency supported by the Sandy Bridge processors of SuperMUC, only the three lower neighboring frequencies are evaluated.

4.3.2 Selection of phase regions

There are several methods to prepare the application selected for energy tuning. Many scientific codes have a time stepping loop, also known as a phase region. The most common method is to target the phase region in a code for tuning. The phase region is typically the region that consumes the most amount of time. Instrumenting the phase region consists of simply inserting the appropriate directives as follows:

- Fortran:
  ```
  DO i = 1, 10
  !$ MON USER REGION
  kernel...
  !$ MON END USER REGION
  ENDDO !Instrumented var
  ```

- C/C++
  ```
  for(int i = 0; i <10; i++)
  {
  #pragma start_user_region
  kernel...
  #pragma end_user_region
  }
  ```

Not every application has an outermost iterative loop with a time stepping scheme that can be used as a phase region. Either the main region can be used as phase region, resulting in an automatic restart of the application for each experiment, or some other region can be marked a phase region to restrict the tuning to this region.

PTF user region can also be used to mark code regions also for other purposes then as a phase region. If a single user region is given, this is assumed to be used as phase region in PTF. If multiple user regions are given, the phase region can be identified by an input argument to the PTF frontend. The argument has the format `--phase=‘‘<file id >:<region first line >’’`. For example:

```
psc\_frontend --apprun="./Seissolxx PAR.par" --mpinumprocs=32
              --tune=dvfs --phase="31:809" --sir=SeisSol.sir
```

If there is no phase region use the command as you would normally do.

Note that there can only be one phase region within the source code and it should be the outer most loop of all other instrumented regions. The presence of phase regions ensure faster automatic tuning, as each experiment is carried out by running one iteration (as opposed to running the entire application per experiment). It is important not to have significant variations of time among iterations, otherwise the comparisons between experiments will produce wrong results. If the iterations at the phase region are suspected to have variations among them, then do not use a phase region. The loop of the phase region should have at least $2m+1$ iterations, where m is the number of frequencies chosen. One iteration is for the pre-analysis, and $2m$ other iterations for the experiments.

4.3.3 Multiple tuning objectives

There are three models used in this plugin to predict the best frequency for a given objective, i.e., one for each of the quantities *Time*, *Power*, and *Energy*. The rest of the models are derived from these three main models and are used for the tuning objectives. Energy is modeled directly and this is the default model of the DVFS Plugin. Nevertheless, there is an energy model derived from the product of modeled power and time separately (export the environment variable "export PSC_DVFS_MODEL=2" before running). Variations are not significant between these two energy models and the user is encouraged to use any of the two when tuning for energy consumption.

There are several tuning objectives that can be chosen. You can optimize the energy consumption, the energy delay product, tune for power capping, optimize the total cost of ownership, or tune the energy consumption for an allowed performance degradation. The choice of the metric depends on the needs of the user. Here are some examples:

- Consider optimizing the energy consumption when other costs related to time are minimal compared to energy costs. (PSC_DVFS_MODEL is either 1 or 2)
- The energy delay product provides a strong emphasis on time (export PSC_DVFS_MODEL=3). The formula has time squared times power.
- Consider using the total cost of ownership when you need to balance the influence of power-related costs and time-related costs (export PSC_DVFS_MODEL=4).

  The Total Cost of Ownership (TCO) was used with the following cost formula (given per compute node):

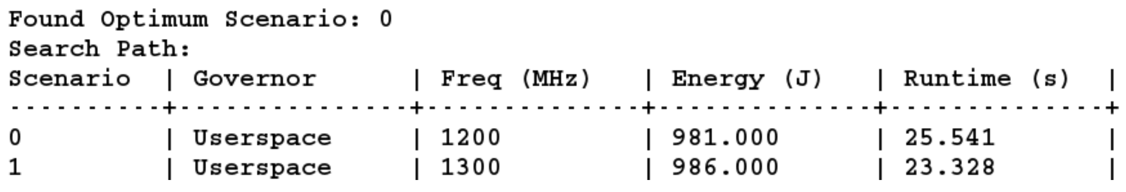
  $$TCO = a \cdot P \cdot t + b \cdot t, \tag{1}$$

  where P is the average power in watts, t the time in seconds, a the cost of the energy (in the plugin $a = 2.7 \cdot 10^{-4}[EUR/J]$), b the cost of personnel and other fixed costs amortized over the lifespan of the HPC system (in the plugin $b = 0.1115[EUR/s]$).
- Power capping is used to limit the power and avoid trespassing a system-wide power limit which can generate more costs as electric companies penalize strong

oscillations (export PSC_DVFS_MODEL=5). The power capping tuning objective uses a maximum power limit of 110 W per node.

- Use PSC_DVFS_MODEL=6 when the increase of performance should be greater than an increase in energy consumption in order to choose higher frequencies and power capping is not needed.
- Use PSC_DVFS_MODEL=7 when the increase of performance should be greater than the increase in power. This case is more restrictive than the previous policy (selected using PSC_DVFS_MODEL=6) as it considers power instead of energy.
- Use PSC_DVFS_MODEL=8 when power capping considerations are important. The frequency used will be chosen such that the power used is below the power threshold or have significant performance increase with respect to the nominal frequency 2.0 GHz.
- Use PSC_DVFS_MODEL=9 when the performance degradation with respect to the nominal frequency should be no more than 10%.

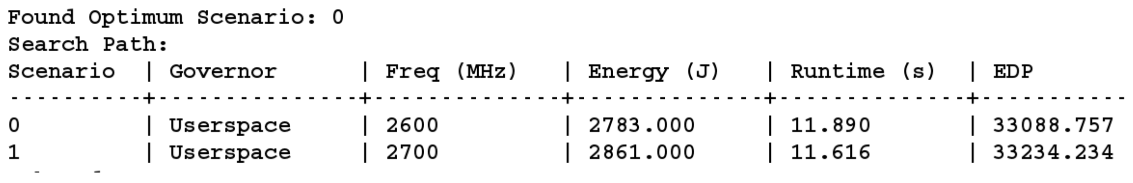
Figure 22 shows the execution of the SeisSol application using a default energy model PSC_DVFS_MODEL=1 and thus tuning for minimum energy usage.



```
Found Optimum Scenario: 0
Search Path:
Scenario  | Governor       | Freq (MHz)    | Energy (J)    | Runtime (s)   |
----------+----------------+---------------+---------------+---------------+
0         | Userspace      | 1200          | 981.000       | 25.541        |
1         | Userspace      | 1300          | 986.000       | 23.328        |
```

Figure 22: Output of the DVFS plugin for energy tuning for SeisSol.

The results show that executing SeisSol at 1.2 GHz would minimize the energy consumed during the execution. When choosing the Energy Delay Product as tuning objective instead of consumed energy, the plugin determines 2.6 GHz as the optimal frequency (Figure 23).



```
Found Optimum Scenario: 0
Search Path:
Scenario  | Governor       | Freq (MHz)    | Energy (J)    | Runtime (s)   | EDP
----------+----------------+---------------+---------------+---------------+-----------
0         | Userspace      | 2600          | 2783.000      | 11.890        | 33088.757
1         | Userspace      | 2700          | 2861.000      | 11.616        | 33234.234
```

Figure 23: Output of the DVFS plugin for tuning the Energy Delay Product for SeisSol.

4.3.4 Multiple application regions

The DVFS plugin can not only determine a global best frequency but also individual best frequencies for different regions of the application. These regions have to be suited for energy tuning, i.e., they have to be coarse enough to amortize for the overhead of changing the processor frequency.

To demonstrate this feature, three regions were marked in SeisSol for energy tuning. Table 6. The two inner regions group similar computations and are included in the phase region (region in calc_seissol.f90). (In the table each region is given an identification number.)

File	Region First Line	Region ID
calc_seissol.f90	489	1
galerkin3d_tetra.f90	193	2
galerkin3d_tetra.f90	289	3

Table 6: Instrumented regions in SeisSol.

Tables 7, 8, and 9 show the results of for the SeisSol application when run with the tuning objective that optimizes for energy consumption. Regardless of the tuning objective, the output of the plugin always displays the energy consumption, the runtime, Energy Delay Product (EDP), Energy Delay squared Product (ED^2P), the average power, and Total Cost of Ownership (TCO). Time and energy (and indirectly average power) are measured and reported on in the output.

Frequency [GHz]	Energy $[J]$	Runtime $[s]$	EDP $[J \cdot s]$	EDDP $[J \cdot s^2]$	Average Power [Watt]	TCO [Euro]
1.6	1132	18.196	20597.4	374782.4	62.2	2.3341
1.7	1143	17.075	19516.6	333244.2	66.9	2.2121
1.8	1172	16.230	19021.3	308712.3	72.2	2.1257

Table 7: Energy figures for Region 1.

Frequency [GHz]	Energy [J]	Runtime [s]	EDP [J· s]	EDDP $[Js^2]$	Average Power [Watt]	TCO [Euro]
1.6	185	1.545	285.8	441.6	119.7	0.2222
1.7	185	1.548	286.4	443.5	119.5	0.2226
1.8	186	1.547	287.8	445.4	120.2	0.2227

Table 8: Energy figures for Region 2.

Frequency [GHz]	Energy [J]	Runtime [s]	EDP [J· s]	EDDP [Js^2]	Average Power [Watt]	TCO [Euro]
1.6	912	16.2	14805.2	240345.1	56.2	2.0560
1.7	928	15.3	14152.7	215837.8	60.8	1.9507
1.8	971	14.6	14136.7	205814.7	66.7	1.8852

Table 9: Energy figures for Region 3.

These results show that using 1.6 GHz optimizes the energy consumption of SeisSol for all three regions. The energy savings per region with respect to the worst case scenario are shown in Table 10.

Region ID	Energy Savings
1	3.4%
2	0.5%
3	6.1%

Table 10: Energy savings per region.

The plugin can be configured to use a tuning objective of either EDP, power capping, or TCO. The EDP search space is the same as the one shown in Tables 7, 8, and 9. The results for the case of EDP point to an optimization when using 1.8 GHz for the outer loop (Region 1) and Region 3. The frequency that tunes EDP for Region 2 is 1.6 GHz.

The results of the tested scenarios for TCO are shown in Table 11. The minimum TCO is achieved by setting 2.7 GHz and costs are reduced by 2.3%.

Frequency [GHz]	TCO [Euro]
2.6	1.7438
2.7	1.7055

Table 11: DVFS output for Region 1 for TCO.

4.3.5 Automatic implementation of the advice

The output of the DVFS plugin is not only printed to standard out, but also given in a special formatted file for the enopt library. The instrumented application can be linked exclusively with enopt (without PTF) for production runs. In this case, the enopt library will read the region-specific settings from the output file and automatically enforce them at runtime.

4.4 Master-Worker Plugin

As an example of how to best exploit the Master-Worker plugin, we will focus on the Modprimes application as our use case here. The performance of master-worker type applications such as Modprimes depends on two main factors. Firstly, it is important to achieve a balanced computational load distributed among workers; and, secondly, it is important to decide the appropriate number of workers.

4.4.1 Model-based reduction of search space

Load balancing and the number of workers can often present huge parameter spaces to search through and so the Master-Worker plugin sets about reducing a potentially huge search space to only nine scenarios using analytical models for estimating the partition factor and number of workers. In the case of the partition factor, the plugin computes the number of tasks processed by the application and the mean execution time each worker dedicates to process a task. Using this information, in combination with user provided parameters (network latency, bandwidth, and task size), the plugin estimates the partition factor that results in the most balanced workload distribution. In the case of finding the optimal number of workers, the plugin measures both the total number of bytes communicated between workers and the time each worker spends doing computation. With this information (as well as user-provided parameters), the Master-Worker plugin estimates the optimal number of workers for which the execution time is minimized.

The Modprimes application loads a set of random numbers from a file and distributes them among the available workers which each use a brute force algorithm to determine which of the set of numbers are prime numbers. To demonstrate how to best exploit some of the features of the Master-Worker plugin, we will use this application with an input set of 64,000 long integer numbers. For this input data set, the total number of bytes communicated is 512KB and the total time spent by the workers executing the brute force algorithm is approximately 3.25s. Starting the application with 10 workers, the partition factor estimation algorithm will compute the values shown in Table 12. Based on this estimation, the plugin will choose 0.65 as the best partition factor.

Next, the plugin estimates that the optimal number of workers for this application is 188. Finally, the plugin will generate nine different scenarios consisting of the estimated partition factor, the number of workers and small variations ($\pm10\%$) of these values. Table 13 shows the nine scenarios generated for the Modprimes example and the total execution time for each. Based on the results for the example here, the plugin advises the user to use a partition factor of 0.65 and 206 workers.

Assuming a range of 0.10-1.00 (with increments of size 0.5) for the partition factor and a range of 100-500 (with increments of size 4) for the number of workers, exhaustive search of both parameters would have led to the execution of 14,400 scenarios. Even for an example with a very small workload as the one that has been used here, this would have represented several hours of PTF execution time. Figure 24 shows the execution time

Partition Factor	Estimated Execution Time
0.10	0.33023
0.15	0.32983
0.20	0.32964
0.25	0.32951
0.30	0.32942
0.35	0.32935
0.40	0.32931
0.45	0.32928
0.50	0.32924
0.55	0.32922
0.60	0.32918
0.65	0.32917
0.70	0.32947
0.75	0.32940
0.80	0.33380
0.85	0.33700
0.90	0.33689
0.95	0.33143

Table 12: Estimation of the application execution time for different values of the partition factor.

Scenario	Execution Time	Partition Factor	Number of Workers
0	0.0591	0.59	170
1	0.0539	0.59	188
2	0.0523	0.59	206
3	0.0610	0.65	170
4	0.0544	0.65	188
5	0.0510	0.65	206
6	0.0582	0.71	170
7	0.0780	0.71	188
8	0.0553	0.71	206

Table 13: Results for the plugin generated scenarios.

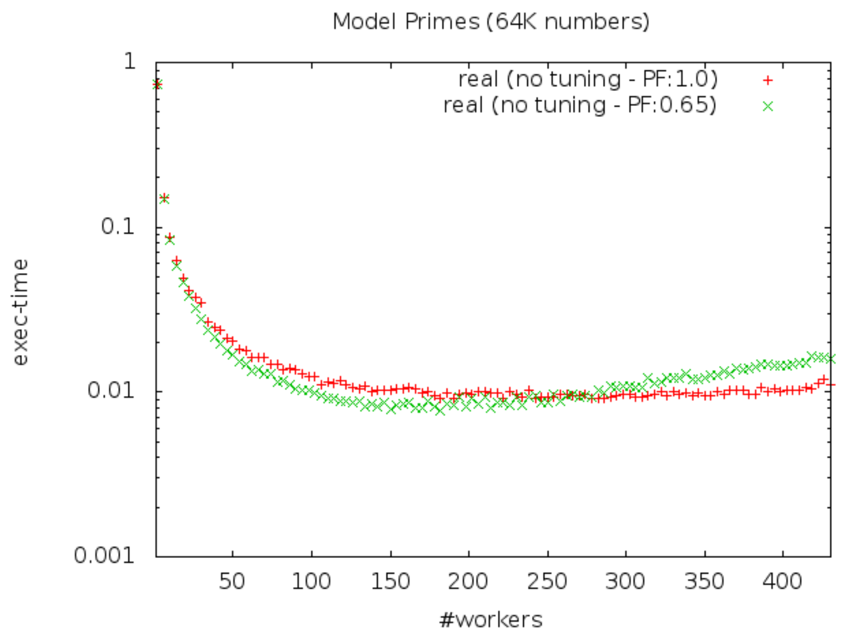


Figure 24: Real application execution time for partition factor values 1.0 and 0.65.

of the application using a partition factor of 1.0 and 0.65 in the range 4 to 430 workers. It can be seen that using a partition factor of 0.65 reduces the execution time of the application and that the best time is obtained for 186 workers.

4.4.2 Uninstrumented applications supported for number of workers

The Master-Worker plugin cannot use analytical models for estimating the partition factor and number of workers if the application is not instrumented or it does not include a partition factor variable in its source code. However, the plugin includes the possibility of tuning only the number of workers using an exhaustive search strategy. In this case, the user can just specify the range of workers and the step of the search. For example, the forest fire propagation application S2F2M [**?**] does not include a mechanism for load balancing (partition factor). In this case, we specify the range of 32 to 192 workers with step 4. The results obtained by PTF are shown in Table 14, where it can be seen that the plugin advises the user to choose 176 workers (scenario 36).

Scenario	Execution Time	Number of Workers
0	53.376	32
1	47.975	36
2	43.410	40
3	39.664	44
4	36.662	48
5	34.372	52
...	...	...
27	19.196	140
28	19.428	144
29	17.815	148
30	17.361	152
31	18.008	156
32	17.069	160
33	17.520	164
34	17.940	168
35	17.374	172
36	15.983	176
37	17.337	180
38	17.213	184
39	16.546	188
40	17.181	192

Table 14: Results for the plugin generated scenarios.

4.5 MPI Parameters Plugin

In order to guide the user on how best to exploit certain features of the MPI Parameters Plugin we will focus on the the Fish School Simulator application (FSSIM) [**?**].

4.5.1 Multiple MPI flavors

The MPI Parameters plugin offers some degree of support for three different implementations of MPI: IBM MPI, Intel MPI and OpenMPI. In these cases, the keywords `ibm`, `intel`, or `openmpi` should appear at the beginning of the configuration file. Then, the plugin interprets that the user is specifying command line options and modifies the application execution command accordingly to the syntax established by the specific implementation.

Assuming that the user has included the parameter for tuning the eager limit in the configuration file, the MPI Parameters plugin will, for example, generate the following command line in each case:

- IBM MPI: mpiexec -n 64 executable -eager_limit 16384
- Intel MPI: mpiexec -genv I_MPI_EAGER_THRESHOLD 16384 -n 64 executable
- OpenMPI: mpiexec -mca osc_pt2pt_eager_limit 16384 -n 64 executable

In any other case, the plugin interprets that the parameters indicated by the user are environment variables and produces the corresponding export commands before re-executing the application. In addition, three example configuration files are provided along with the MPI Parameters plugin, one for each supported implementation (Intel MPI, IBM MPI and OpenMPI). They can be used as a quite complete starting point for tuning a rich set of MPI parameters. However, the user can also modify them (eliminating, adding or changing parameters) with the objective of fitting them to a particular application.

The three configuration files included with the PTF release are the following:

1. IBM MPI

```
MPIPO_BEGIN ibm
eager_limit=4096:2048:65560;
buffer_mem=8388608:2097152:134217728;
use_bulk_xfer=yes,no;
bulk_min_msg_size=4096:4096:1048576;
pe_affinity=yes,no;
cc_scratch_buf=yes,no;
wait_mode=nopoll,poll;
css_interrupt=yes,no;
```

```
polling_interval=100000:10000:1000000;
SEARCH=gde3;
MPIPO_END
```

2. Intel MPI

```
MPIPO_BEGIN intel
I_MPI_EAGER_THRESHOLD=4096:2048:65560;
I_MPI_INTRANODE_EAGER_THRESHOLD=4096:2048:65560;
I_MPI_SHM_LMT=shm,direct,no;
I_MPI_SPIN_COUNT=1:2:500;
I_MPI_SCALABLE_OPTIMIZATION=yes,no;
I_MPI_WAIT_MODE=yes,no;
I_MPI_USE_DYNAMIC_CONNECTIONS=yes,no;
I_MPI_SHM_FBOX=yes,no;
I_MPI_SHM_FBOX_SIZE=2048:512:65472;
I_MPI_SHM_CELL_NUM=64:4:256;
I_MPI_SHM_CELL_SIZE=2048:1024:65472;
SEARCH=gde3;
MPIPO_END
```

3. OpenMPI

```
MPIPO_BEGIN openmpi
mpi_paffinity_alone=0,1;
btl_openib_eager_limit=1024:2048:65560;
btl_openib_free_list_num=2:4:128;
btl_openib_use_eager_rdma=0,1;
btl_openib_eager_rdma_num=1:2:32;
btl_sm_eager_limit=1024:2048:65560;
btl_sm_num_fifos=1:1:10;
btl_sm_fifo_size=2048:512:65472;
btl_sm_free_list_num=2:4:128;
MPIPO_END
```

4.5.2 Genetic search

PTF has been enriched with the implementation of a search strategy based on the *Generalized Differential Evolution 3 (GDE3)* genetic algorithm. In this strategy a population of ten initial scenarios is randomly generated and executed, then, an iterative process is followed, generating new populations by selecting the best five scenarios (those with the smallest execution time) for the next generation (elitism), generating five new scenarios by crossing over the previous population (crossover), and introducing mutations with a

```
MPI_PIPO BEGIN ibm
eager_limit=4096:15366:65560;
use_bulk_xfer=yes,no;
bulk_min_msg_size=4096:4096:1048576;
task_affinity=CORE,MCM;
pe_affinity=yes,no;
cc_scratch_buf=yes,no;
MPI_PIPO END
```

Table 15: Configuration file for the exhaustive search.

```
MPIPO_BEGIN ibm
eager_limit=4096:2048:65560;
buffer_mem=8388608:2097152:134217728;
use_bulk_xfer=yes,no;
bulk_min_msg_size=4096:4096:1048576;
cc_scratch_buf=yes,no;
wait_mode=nopoll,poll;
css_interrupt=yes,no;
polling_interval=100000:10000:1000000;
task_affinity=CORE,MCM;
pe_affinity=yes,no;
SEARCH=gde3;
MPI_PIPO END
```

Table 16: Configuration file for the GDE3 search.

fixed probability (mutation). The number of iterations can be configured, but, generally, a close to optimal solution can be found in less than 30 iterations (generations).

For our use cases we have used a medium size population of fishes (64K), to avoid long executions, and have run the MPI application on 64 compute cores. In addition, given that it is not possible to test a large set of parameters combinations exhaustively in a reasonable time, we have used the exhaustive search strategy for a very limited configuration (see Table 15) of the parameters, and the heuristic search strategy for a more complete configuration (see Table 16).

The exhaustive search strategy generates all possible combinations of the selected parameters and values, 320 in our example. Each scenario is executed and the one with the smallest wall time is chosen as the best one. The best combination corresponds to scenario 217: `use_bulk_xfer` yes, `bulk_min_msg_size` 796672, `cc_scratch_buf` no, `task_affinity` CORE, `eager_limit` 50194, `pe_affinity` yes. The execution time for this scenario was 2.8 sec, which is 1.6 times better than the time for the execution using the parameters default values (4.4 sec). The drawback is that for finding the optimal scenario in the set of tested ones, PTF needed 21505.2 sec (almost 6 hours) for the use case here.

The genetic search strategy (GDE3) can also be used for the same tuning space, but can even handle much larger tuning spaces consisting of significantly more parameters and possible values. For the use case we provide here, it executes 20 different scenarios in 1138.65 sec (approximately 20 min), which reduces the analysis time by a factor of 19 with respect to the exhaustive search. The best scenario in this case was found to be number 14: `use_bulk_xfer` no, `bulk_min_msg_size` 988789, `css_interrupt` no, `wait_mode` poll, `polling_interval` 636027, `cc_scratch_buf` no, `task_affinity` CORE, `buffer_mem` 102205313, `eager_limit` 59434, `pe_affinity` yes. The execution time for this scenario was 2.8 sec, again significantly better than the one obtained with the default values and now the search time has also been significantly reduced. These results demonstrate that using the genetic search strategy can lead to results that are almost as good as the ones produced by the exhaustive search in only a fraction of the time needed for the latter.

4.5.3 Eager-limit parameter strategy

The MPI Parameters plugin includes an automatic strategy for determining if it is worthy to include the eager limit and memory buffer parameters into the plugin search space. Moreover, if it determines that it is worthy to include these parameters in the search, it will automatically generate the range of values that should be searched, trying to shrink this range in order to reduce the size of the search space. For using this strategy, the user only has to specify in the configuration file the option `AUTO_EAGER_LIMIT=<eager-limit parameter>,<buffer-mem parameter>`. In our case, using IBM MPI, the configuration option is `AUTO_EAGER_LIMIT=eager_limit, buffer_mem;`.

Here we test the plugin using this configuration option, where the FSSIM application was executed with a medium size population of 64K fishes and on 64 cores. The plugin determines that it is worthy to include the `eager_limit` and `buffer_mem` parameters into the search space because the total number of bytes communicated in point-to-point messages smaller than 64Kb was found to be greater than 30% of the total number of bytes communicated in point-to-point messages. In addition, the plugin determines that the search range for the eager limit parameter should be between 1Kb and 32Kb because more than 80% of the messages sent eagerly were in this range. Finally, the plugin uses the values determined for the eager limit and expression 2 to compute the range for the buffer mem parameter range (128Kb to 4Mb).

$$mem\_buff = 2n * max(eager\_limit, 64) \tag{2}$$

Figures 25 and 26 clearly show that increasing the eager limit for this application up to approximately 30Kb produces significant performance improvements, and that beyond this size no extra gains are obtained. This demonstrates that the eager limit strategy is successfully identifying the range of values that should be explored by the user for finding the best value for this parameter.

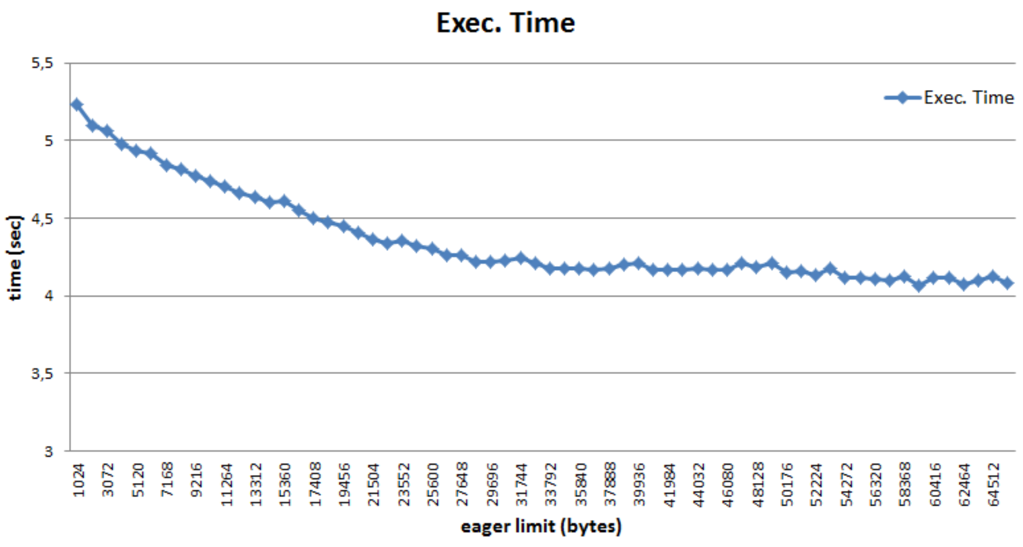


Figure 25: FSSIM execution time for different values of the eager limit and memory buffer parameters (IBM MPI).

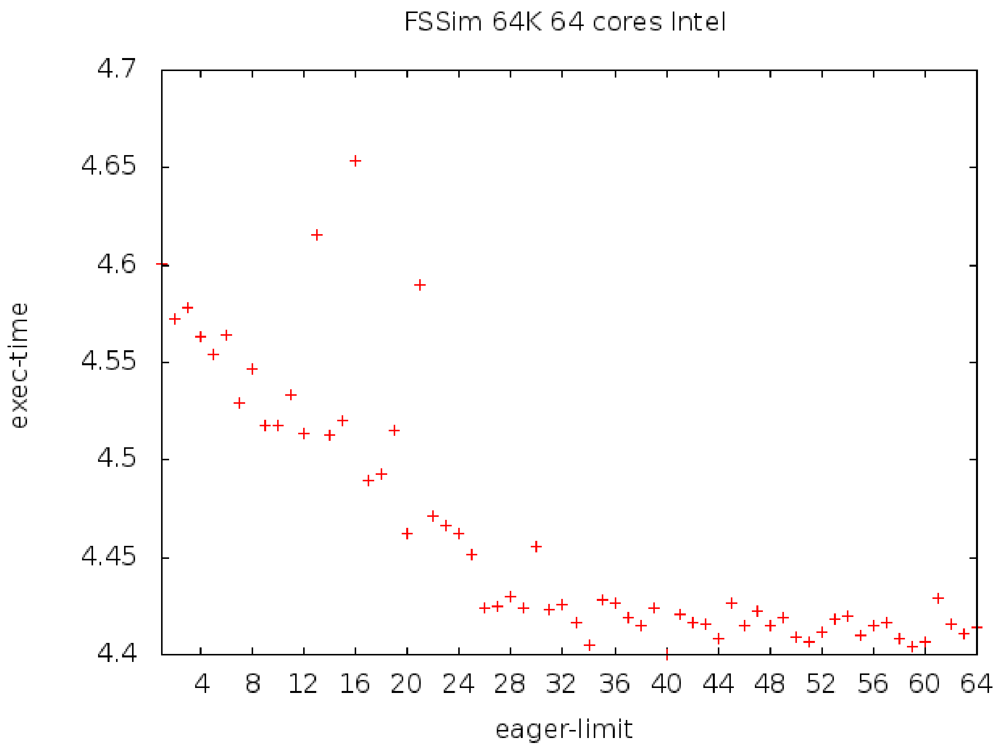


Figure 26: FSSIM execution time for different values of the eager limit and memory buffers parameters (Intel MPI).