PTF Master-Worker Plugin User's Guide

PTF Version: 1.1

MW Plugin Version: 1.1

Gertvjola Saveta, Eduardo Cesar, Anna Sikora

13.04.2015

Contents

Chapter 1

Introduction

Most parallel applications are designed and implemented using well known patterns such as master-worker (or Process Farm), pipeline, or divide and conquer. These patterns usually present specific performance problems that could affect any application based on the pattern. Consequently, defining performance models associated to the pattern of the applications can be used to implement generic tuning strategies for wider sets of applications.

This approach has been followed to design and implement the Master-Worker tuning plugin. We have used a performance model to develop a plugin that can be used to automatically tune many MPI master-worker applications, only asking the application developer to identify a few code regions and communication phases.

The master-worker framework is a well-known parallel programming structure because it enables expression, in a natural way, of the behavioral characteristics of a wide range of high-level parallel application patterns. Basically, this framework includes a master process which sends tasks to a set of worker processes, then each worker makes some kind of computation on the received tasks, a computation that generally requires a variable and unpredictable time, and finally it sends back the results to the master.

The performance of master-worker applications mainly depends of two factors. First, it is important to get a balanced computational load among workers; and, second, it is important to decide the appropriate number of workers.

A partition strategy of the set of tasks is used for balancing the load among workers. Instead of distributing the whole set of tasks among workers and then waiting for the results, the master makes a partial distribution of the tasks dividing the original set into portions (called batches) of decreasing size. The idea is to distribute the first of these batches among workers in chunks of (roughly) the same number of tasks. When a worker ends the processing of its assigned chunk the master sends to that worker a new chunk; the process continues until all batches are completely distributed. This way, workers that received tough tasks will not receive more work, and workers that received lighter tasks are employed to do more work. Logically, smaller batches lead to better load balancing but increase the communication overhead, while bigger batches could lead to poorer load balancing and less communication.

In an ideal master-worker application the total execution time would be equal to the sequential execution time divided by the number of workers. Nevertheless, in this case we are assuming that communication is free, the application executes on a dedicated and homogeneous platform, there is a perfect load balancing, and the computation also scales ideally. In this ideal world, any available resource that can be assigned to the application must be assigned since it can be efficiently used to improve the performance of the application. In the real world, however, we can observe that the speedup of the application usually decreases as new resources are assigned to it, indicating a loss in efficiency. Moreover, at some point, assigning more resources to the application produces drops in performance because the introduced costs are bigger than the advantages brought about by the new resources.

Consequently, there are two tuning parameters associated to any master-worker application, a partition factor that determines the appropriate number of tasks of each batch to be sent to the workers, and the best number of workers to be used. Both parameters can be estimated using an analytical approach. Therefore, they can be tuned automatically if the necessary measurements (computation time of the workers and communication cost of tasks) can be obtained.

The Master-Worker plugin can work on instrumented applications using analytical performance models to estimate these performance parameters, but also on uninstrumented applications only for estimating the number of workers.

The combination of values leading to the lowest execution time is provided to the user as an advice.

Chapter 2

Quick Start

2.1 Quick installation

Master-Worker plugin is being installed along with the Periscope Tuning Framework. Please refer to the PTF Installation Guide for a complete description of the installation process.

2.2 Basic configuration - `param_spec.conf`

In order to use the Master-Worker plugin, a set of configuration parameters is required. These parameters are read from a configuration file (by default `param_spec.conf`).

You can start by copying the example configuration file `param_spec.conf` into the folder containing the executable of your application.

```
$PSC_ROOT/templates/param_spec.conf →
    $APP_ROOT/.../param_spec.conf
```

Then, you can edit `param_spec.conf` to adapt the configuration file to your application and architecture characteristics.

The configuration file will be different if the application is uninstrumented or instrumented. In the case of instrumented application, several parameters should be provided for computing the analytical models, while, in the uninstrumented one, only the range for the number of workers should be specified.

2.2.1 Unistrumented Application

```
# Configuration File Start
MWT_BEGIN
NW=<initial-value>:<step>:<final-value>
MWT_END
```

The NW parameter defines the range of workers that should be exhaustively tested using the indicated step.

2.2.2 Instrumented Application

```
# Configuration File Start
MWT_BEGIN
MASTERRECV=<file-id>:<line>
[MASTERRECV=<file-id>:<line>]
WORKERRECV=<file-id>:<line>
[WORKERRECV=<file-id>:<line>]
WORKERFUNC=<file-id>:<initial-line>
NETLATENCY=<value in sec>
NETSPEED=<value in sec/byte>
IMBALANCETHR=<value in %>
TASKSIZE=<value in bytes>
MWT_END
```

- `WORKERRECV` : the parameter defines the line (or lines) in the code of the MPI_Recv call that receives tasks in the workers;
- `WORKERFUNC` : the parameter indicates the line in the code of the function for task processing;
- `MASTERRECV` : the parameter indicates the line (or lines) in the code where the master calls MPI_Recv for receiving data from workers;
- `TASKSIZE` : the parameter indicates the size of each task;
- `NETLATENCY` : the parameter specifies the average library latency for building messages;
- `NETSPEED` : the parameter specifies the average communication time per byte(inverse bandwidth);
- `IMBALANCETHR` : the parameter indicates the maximum acceptable execution time difference (in %) between the fastest and the slowest

worker, beyond which the plugin considers that the application is imbalanced.

2.3 Running Master-Worker plugin

The Master-Worker runs as a plugin within the Periscope Tuning Framework. It can be started using `psc_frontend` (see also *PTF User's Guide*) by setting the `tune` flag to `masterworker`.

```
--tune=masterworker
```

For an uninstrumented application, the user should execute:

```
psc_frontend --apprun="./Application" --tune=masterworker --uninstrumented
```

This will start the measurements and the master-worker tuning strategy for an uninstrumented application.

Please note that, in the uninstrumented mode, the execution time is measured as the wall clock time of the system command which executes the application. This means that reliable results can be achieved only if the execution time of the application is not too small.

By default, the PTF runs on instrumented applications. For an instrumented application the user should execute:

```
psc_frontend --apprun="./Application" --tune=masterworker
```

This will start the measurements and the master-worker tuning strategy for an instrumented application.

2.4 Execution results

Upon successful completion, the Master-Worker plugin displays at the standard output the list of all parameters combination (scenarios) that were used in the search along with the corresponding execution times (severity). It also outputs the scenario with the best execution time.

For example, the following is the output of the above call to `psc_frontend` for an uninstrumented master-worker application using different numbers of workers:

```
 Optimum Scenario:0
NumberOfWorkers Value:  2

All Results:
-----------------------

Scenario | Severity | Tuning Parameters

0 11.077821 Name:  NumberOfWorkers Value:  2
1 11.078959 Name:  NumberOfWorkers Value:  4
2 11.080354 Name:  NumberOfWorkers Value:  6
```

and, the following is an example of the output for an instrumented master-worker application:

```
Found best Scenario:  5
TuningParameter:  0 Name:  PartitionFactor Value:  55
TuningParameter:  1 Name:  NumberOfWorkers Value:  327

All Results (9):
Scenario | Severity Tuning Parameters
0 | 0.11575 Name:  PartitionFactor Value:  50 Name:  NumberOfWorkers
Value:  269
1 | 0.110449 Name:  PartitionFactor Value:  50 Name:  NumberOfWorkers
Value:  298
2 | 0.107365 Name:  PartitionFactor Value:  50 Name:  NumberOfWorkers
Value:  327
3 | 0.114166 Name:  PartitionFactor Value:  55 Name:  NumberOfWorkers
Value:  269
4 | 0.108855 Name:  PartitionFactor Value:  55 Name:  NumberOfWorkers
Value:  298
5 | 0.105826 Name:  PartitionFactor Value:  55 Name:  NumberOfWorkers
Value:  327
6 | 0.11606 Name:  PartitionFactor Value:  60 Name:  NumberOfWorkers
Value:  269
7 | 0.107335 Name:  PartitionFactor Value:  60 Name:  NumberOfWorkers
Value:  298
8 | 0.108243 Name:  PartitionFactor Value:  60 Name:  NumberOfWorkers
Value:  327
```

Chapter 3

Master-Worker Autotuning Approach

The Master-Worker plugin follows the general PTF plugin approach (see also *PTF User's Guide*).

3.1 Tuning parameter

In case of an uninstrumented application, the only parameter that can be tuned is the number of workers. The user should specify the desire number of workers to be tuned in the configuration file. However, for an instrumented application the plugin predicts and tunes the number of workers and the partition factor. For being able to predict the user have to specify the tuning parameters depending on his apllication. The parameters asked to be specified are: the `WORKERRECV` parameter defines the line (or lines) in the code of the MPI_Recv call that receives tasks in the workers; the `WORKERFUNC` parameter indicates the line in the code of the function for task processing; the `MASTERRECV` parameter indicates the line (or lines) in the code where the master calls MPI_Recv for receiving data from workers; the `TASKSIZE` parameter indicates the size of each task; `NETLATENCY` and `NETSPEED` parameters are used for specifying the average library latency for building messages and the average communication time per byte (inverse bandwidth); finally, the `IMBALANCETHR` parameter indicates the maximum acceptable execution time difference (in %) between the fastest and the slowest worker, beyond which the plugin considers that the application is imbalanced.

3.2 Search strategy

In order to find the best tuning of an application, a search through the tuning space has to be performed. For an uninstrumented application, the Master-Worker plugin uses an exhaustive search strategy. For an instrumented one, the Master-Worker plugin uses prediction models for determining the most promising values of the partition factor and number of workers generating a reduced search space around these values.

3.3 Tuning scenario

Tuning scenarios are being generated at run time and the performance of the application is being evaluated for each of these scenarios. In the Master-Worker plugin, one scenario represents the number of workers (uninstrumented application) or the number of workers and partition factor (instrumented application).

For instrumented applications the plugin performs two tuning steps. The first for tuning the partition factor and the second for estimating the number of workers. When the prediction is done, the plugin creates for each tuning parameter three values. The predicted value, and two expanding the predicted value +-10% to calculate the estimated error. In total the plugin will create 9 scenarios. Scenarios are created at the end of the second tuning step. Each tuning step predicts the best value for a tuning parameter and then a set of scenarios are created nearby the predicted values.

3.4 Tuning action

Applying a scenario to the application means, in the case of the Master-Worker plugin, re-executing the application with a different number of workers (uninstrumented application) or a different number of workers and partition factor (instrumented application).

3.4.1 Uninstrumented application

For the unisntrumented application the plugin will go through one tuning step. This step is to re-execute the application with a different number of workers.

3.4.2 Instrumented application

On the other hand, instrumented applications require two tuning steps. Pre-analysis is required for both tuning steps of the plugin.

- The Periscope pre-analysis is used to optimize the partition factor. The `ExecTime` and `MPITime` properties are used to obtain the average execution time of the workers and total number of tasks processed. In addition, the plugin obtains the execution time for the worker that took longer and the one that took less time to process the assigned tasks. With this information it can decide if the application is balanced or not. If the difference between the slowest and the fastest workers is over a given threshold, a simulator is triggered to estimate the partition factor that would lead to the most balanced distribution of the load.
- Then, fixing the partition factor to the value obtained in the first tuning step and calling a new pre-analysis for the same properties as in tuning step 1, the plugin gets the information needed to estimate the appropriate number of workers, i.e., the total number of bytes transferred and the total execution time of all application workers. This number of workers is calculated with the provided mathematical model. Finally, the plugin creates one scenario with the two estimated parameters, and 7 extra scenarios introducing a 10% variation on the number of workers and the partition factor.

Thus, the tuning action is to re-execute the application.

Chapter 4

Configuration

4.1 param_spec.conf file

All configuration settings for the Master-Worker plugin are read at execution time from the configuration file. The default name of the configuration file is:

param_spec.conf

Otherwise, an environment variable (PSC_PARAM_SPEC_FILE) is used to specify the name of the configuration file.

4.2 Master-Worker tuning parameters

The tuning parameters for the Master-Worker plugin are defined in the param_spec.conf file as follows:

4.2.1 Uninstrumented Application

```
# Configuration File Start
MWT_BEGIN
NW=<initial-value>:<step>:<final-value>
MWT_END
```

The NW parameter defines the range of workers that should be exhaustively tested using the indicated step.

4.2.2 Instrumented Application

```
# Configuration File Start
MWT_BEGIN
MASTERRECV=<file-id>:<line>
[MASTERRECV=<file-id>:<line>]
WORKERRECV=<file-id>:<line>
[WORKERRECV=<file-id>:<line>]
WORKERFUNC=<file-id>:<initial-line>
NETLATENCY=<value in sec>
NETSPEED=<value in sec/byte>
IMBALANCETHR=<value in %>
TASKSIZE=<value in bytes>
MWT_END
```

- `WORKERRECV` : the parameter defines the line (or lines) in the code of the MPI_Recv call that receives tasks in the workers;
- `WORKERFUNC` : the parameter indicates the line in the code of the function for task processing;
- `MASTERRECV` : the parameter indicates the line (or lines) in the code where the master calls MPI_Recv for receiving data from workers;
- `TASKSIZE` : the parameter indicates the size of each task;
- `NETLATENCY` : the parameter specifies the average library latency for building messages;
- `NETSPEED` : the parameter specifies the average communication time per byte(inverse bandwidth);
- `IMBALANCETHR` : the parameter indicates the maximum acceptable execution time difference (in %) between the fastest and the slowest worker, beyond which the plugin considers that the application is imbalanced.

It is worth noticing that for setting `NETLATENCY` and `NETSPEED` the user may use the specifications for the target hardware or run a program for measuring the MPI performance, such as ping-pong or mpptest.

4.3 Improved tuning time

The only way to guide the Master-Worker plugin for reducing the tuning time is to instrument the application.

In this case, the tuning has two phases, in the first, it runs a Periscope pre-analysis of the application using a partition factor of 1 (all tasks are evenly distributed among the workers at the beginning of the iteration) and measures the differences among the workers' execution times. If the difference between the fastest and the slowest worker is bigger than the threshold indicated by the user then the plugin simulates the execution of the application using different partition factors, choosing the one that leads to the best results. If this difference is bellow the threshold the partition factor remains equal to 1.

In the second phase, the plugin executes a second pre-analysis of the application only if the partition factor has been changed for taking new measurements on the balanced application. During this phase an analytical model is used to estimate the appropriate number of workers.

Next, only nine scenarios are created using the estimated values and variations (±10%) of them.

Finally, the advice given to the user consists of the partition factor and number of workers that led to the best execution time.

Chapter 5

How To Use the Tuning Advice

The best combination of values, in our case the partition factor and the number of workers, is selected based on the execution-time metric and provided to the user as a recommendation. Moreover, all scenarios created are shown to verify the advice. The advice about the partition factor can be used changing the corresponding variable in the source code of the application or passing the value as an application command-line parameter. The advice about the number of workers must be used in the `mpirun` command through `-np` option.