Nornir Documentation

Release 1.0.0

Daniele De Sensi

Jan 25, 2021

USER GUIDE

CHAPTER

ONE

INTRODUCTION

Nornir is a runtime support, that can be used to control performance and/or power consumption of parallel applications. The user can specify performance and power consumption requiremnets, and Nornir will enforce them by selecting the appropriate amount of resources to allocate to the application (e.g. number of cores, clock frequency, threads' mapping, etc…). The application will be monitored throughout its entire execution to provide such guarantees even in presence of workload fluctuations or phase changes. Nornir also allows the user to easily integrate custom decision policies inside the framework. For more information, please refer to the documentation.

CHAPTER

TWO

CONTRIBUTIONS

Nornir has been mainly developed by Daniele De Sensi (d.desensi.software@gmail.com).

The following people contributed to Nornir:

- Daniele De Sensi (d.desensi.software@gmail.com): Main developer
- Federico Umani: PredictorSMT predictor.

If you would like to contribute to Nornir development, for example by adding new algorithms, please refer to the documentation.

2.1 Building and Installing

First of all, download Nornir:

```
$ git clone git://github.com/DanieleDeSensi/nornir.git
$ cd nornir
```

To compile Nornir:

```
$ mkdir build
$ cd build
$ cmake ../
$ make
```

Then, you can install it:

```
$ make install
```

To install it into a non-default directory *dir*, simply specify the *-DCMAKE_INSTALL_PREFIX=dir* when calling *cmake*.

2.1.1 Configuring the System

Some operations performed by Nornir are hardware-specific (e.g. reading the system energy consumption, dynamically chaning the clock frequency, etc..). To perform such operations Nornir relies on the Mammut library, and you may want to refer to the Mammut documentation for more information. However, these are the main points to consider:

- To change the frequency, Mammut assumes that the *acpi_cpufreq* driver is used for scaling the clock frequency. Mammut currently does not support the *intel_pstate* driver. If your machine is using the *intel_pstate* driver, you should switch to *acpi_cpufreq*

- On Intel and AMD processors, you should load the *msr* kernel module to allow Mammut to read the energy consumption. For PowerPC and ARM processors please refer to the Mammut documentation.
- Changing the freqeuency and reading the energy usually require *sudo* rights, and the easiest option would be to just run your application with *sudo*. If this is not possible, Mammut supports the msr-safe library. If you want to use msr-safe, please contact me and I will provide you the MSR registers whitelist.

2.2 Interaction between Nornir and the application

Nornir can be used in different ways depending on how your application is implemented and on how you want the Nornir runtime to interact with your application. This decision influence how application performance is measured, what type of performance requirements you can express, and which actuators Nornir can use (e.g. in some cases it may be able to dynamically add/remove threads from the application). Namely:

- If you can't or don't want to modify your application:
  - If your application is an OpenMP application, you can express performance requirements in terms of OpenMP tasks (or OpenMP loop iterations) executed per time unit. For more details refer to the *OpenMP* section.
  - If your application is not using OpenMP, you can express performance requirements in terms of assembler instructions executed per time unit. For more details refer to the *Black-Box* section.
- If you can modify your application:
  - If your application is a FastFlow application, you can express performance requirements in terms of FastFlow tasks executed per time unit. For more details refer to the FastFlow section.
  - If your application is a C/C++ application, you need to add two instrumentation calls in the main loop of your application. In this case you can express performance requirements in terms of loop iterations executed per time unit. For more details refer to the *Instrumentation* section.

2.2.1 OpenMP

First of all, when compiling Nornir you should specify the *-DENABLE_OMP=ON* flag in when calling *cmake*. Then, let's assume that your application is called *myOMPApp*, and that you usually run it with the following command:

```
$ myOMPApp p1 p2
```

Then, if you want to control its performance and/or power consumption, you can run it in this way:

```
$ nornir_openmp "myOMPApp p1 p2" config.xml
```

config.xml is a file containing your requirements and other configuration parameters. A full description is provided in the *Requirements and Configuration* section.

Warning: You may need to run this command with *sudo* to allow Nornir to read the energy consumption and scale the clock frequency.

2.2.2 Black-Box

First of all, when compiling Nornir you should specify the *-DENABLE_BLACKBOX=ON* flag in when calling *cmake*. Then, let's assume that your application is called *myApp*, and that you usually run it with the following command:

```
$ myApp p1 p2
```

Then, if you want to control its performance and/or power consumption, you can run it in this way:

```
$ nornir_blackbox "myOMPApp p1 p2" config.xml
```

config.xml is a file containing your requirements and other configuration parameters. A full description is provided in the *Requirements and Configuration* section.

Warning: You may need to run this command with *sudo* to allow Nornir to read the energy consumption and scale the clock frequency.

2.2.3 FastFlow

If your application is implemented using a FastFlow *farm*, you can express requirements in terms of tasks processed per time unit. You should convert your farm so that it can be controlled by the Nornir runtime support. The process is straightforward and involves the following steps:

- Add *#include <nornir/nornir.hpp>* before including any FastFlow header.
- The classes describing the Emitter, Workers and Collector of the farm must extend *nornir::AdaptiveNode* instead of *ff::ff_node*. *svc_init* and *svc_end* are now called every time that the number of threads is changed by Nornir (in FastFlow they were called only once at the application start and end). Accordingly, if those operations need to be performed only once, you should ensure that by using some support functions.
- If the application wants to be aware of the changes in the number of threads (e.g. to redistribute internal data), the nodes can implement the *notifyRethreading* virtual method.
- The farm needs to be passed as a parameter to the Nornir manager, which will take care of the execution, providing the performance and/or power consumption required, as shown in the following code snippet:

```
// Create emitter, worker, and collector and add to the farm 'farm'.
nornir::Parameters config;
config.contractType = CONTRACT_PERF_THROUGHPUT;
config.requiredBandwidth = 40;
nornir::ManagerFarm<> manager(&farm, config); // Create nornir manager.
manager.start(); // Start farm.
manager.join(); // Wait for farm end.
```

In this snippet, we shown how it is possible to manage an already existing farm, by requiring a minimum performance of 40 tasks processed per second. More details on the parameters can be found in the the *Requirements and Configuration* section. A full working example can be found here.

When compiling your application, you should add `-lnornir` flag to the linker options.

Warning: You may need to run your application with *sudo* to allow Nornir to read the energy consumption and scale the clock frequency.

2.2.4 Instrumentation

If your application is a C/C++ application, you can modify it by adding a few instrumentation calls, which will allow Nornir to measure the actual application performance. After identifying the main loop of your application, you can add the instrumentation calls in the following way:

C

C++

```
#include <nornir/nornir.h>

....
NornirInstrumenter* instr = nornir_instrumenter_create("config.xml");

while(...){ // This is the main loop
  nornir_instrumenter_begin(instr);
  ... // Loop core
  nornir_instrumenter_end(instr);
}
nornir_instrumenter_destroy(instr);
```

```
#include <nornir/nornir.hpp>

....
nornir::Instrumenter* instr = new nornir::Instrumenter("config.xml", 1, NULL, true);

while(...){ // This is the main loop
  instr->begin();
  ... // Loop core
  instr->end();
}
delete instr;
```

config.xml is a file containing your requirements and other configuration parameters. A full description is provided in the *Requirements and Configuration* section. This example assumes that the main loop is only executed by one thread. If this is not the case (i.e. the instrumentation calls are called by more simultaneously), you need to specify which thread is calling the instrumentation calls in the following way (*nThreads* is the number of threads and *tid* is the identifier of a thread):

C

C++

```
#include <nornir/nornir.h>

....
NornirInstrumenter* instr = nornir_instrumenter_create_with_threads("config.xml",␣
↪nThreads);

while(...){ // This is the main loop
  nornir_instrumenter_begin_with_threads(instr, tid);
  ... // Loop core
  nornir_instrumenter_end_with_threads(instr, tid);
}
nornir_instrumenter_destroy(instr);
```

```
#include <nornir/nornir.hpp>

....
nornir::Instrumenter* instr = new nornir::Instrumenter("config.xml", nThreads, NULL,␣
↪true);

while(...){ // This is the main loop
  instr->begin(tid);
  ... // Loop core
  instr->end(tid);
}
delete instr;
```

When compiling your application, you should add `-lnornir` flag to the linker options. A full working example can be found here.

Warning: You may need to run your application with *sudo* to allow Nornir to read the energy consumption and scale the clock frequency.

2.3 Requirements and Configuration

Different configuration parameters can be provided to Nornir, specifying the type and value of the requirements, the type of resources that Nornir must reconfigure and other additional parameters. The following requirements can be specified:

- **powerConsumption**: The maximum allowed power consumption.
- **throughput**: The minimum required throughput in terms of iterations/tasks/instructions processed per second.
- **executionTime**: The maximum required completion time.
- **expectedTasksNumber**: The number of iterations/tasks/instructions that will be executed by the application.
- **latency**: The maximum latency per iteration/task (NOT SUPPORTED AT THE MOMENT).
- **minUtilization**: The minimum allowed utilization (in the queueing theory sense), between 0 and 100 (default = 80.0).
- **maxUtilization**: The maximum allowed utilization (in the queueing theory sense), between 0 and 100 (default = 90.0).

If specified programmatically (e.g. in the FastFlow case), such parameters must be specified as fields of the *Parameters* object. If specified through an XML file, they must be specified under <*requirements*> XML tag, as shown in the following example, where the user asks Nornir to enforce the most performing configuration with a maximum power consumption of 50 watts:

```
<?xml version="1.0" encoding="UTF-8"?>
<nornirParameters>
  <requirements>
    <throughput>MAX</throughput>
    <powerConsumption>50</powerConsumption>
  </requirements>
</nornirParameters>
```

Other parameters allow the user to specify which hardware knobs Nornir must use:

- **knobCoresEnabled**: Allows Nornir to find the best amount of cores to allocate to the application (default = true).
- **knobMappingEnabled**: Allows Nornir to find the best allocation of threads on cores (default = true).
- **knobFrequencyEnabled**: Allows Nornir to find the best clock frequency (default = true).
- **knobClkModEnabled**: Allows Nornir to find the best clock modulation value (default = false).
- **knobHyperthreadingEnabled**: Allows Nornir to find the best SMT level (default = false).

Nornir provides different algorithms for deciding how many resources to use according to the application characteristics. The algorithm can be specified through the *strategySelection* parameters, which can assume one of the following values:

- **LEARNING**: Applies an online learning algorithm. The algorithm is described in: *"A Reconfiguration Algorithm for Power-Aware Parallel Applications"* - De Sensi, Daniele and Torquati, Massimo and Danelutto, Marco. This algorithm can use different models to predict performance and power consumption of the application (see *strategyPredictionPerformance* and *strategyPredictionPower* parameters). To use some of these prediction models you must specify the *-DENABLE_MLPACK=ON*, *-DENABLE_GSL=ON* or -DENABLE_ARMADILLO=ON* when running *cmake*.
- **ANALYTICAL**: Applies a simple analytical model. The algorithm is described in: *"Energy driven adaptivity in stream parallel computations"* - Danelutto, Marco and De Sensi, Daniele and Torquati, Massimo
- **ANALYTICAL_FULL**: Applies a simple analytical model (to predict both power and performance). The algorithm is described in: *"Application-Aware Power Capping using Nornir"* - De Sensi, Daniele and Danelutto, Marco
- **FULLSEARCH**: Tries all the configurations in order to find the best one.
- **LIMARTINEZ**: Applies the algorithm described in: *"Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors"* - Jian Li and Jose F. Martınez
- **LEO**: Applies the algorithm described in: *"A Probabilistic Graphical Model-based Approach for Minimizing Energy Under Performance Constraints"* - Mishra, Nikita and Zhang, Huazhe and Lafferty, John D. and Hoffmann, Henry
- **HMP_NELDERMEAD**: Nelder-Mead algorithm starting from a predicted optimal (for HMP systems).
- **RAPL**: Uses RAPL power capper to set power consumption requirements.
- **MANUAL_CLI**: Best configuration is manually chosen by using the selector-manual command line interface.
- **MANUAL_WEB**: Best configuration is manually chosen by using the selector-manual web interface.

It is worth mentioning that not all the algorithms support all the type of requirements and/or harware actuators. For more information and for documentation on additional parameters please refer to the parameters.hpp header.