H2020 FETHPC-1-2014

An Exascale Programming, Multi-objective Optimisation and Resilience Management Environment Based on Nested Recursive Parallelism

Project Number 671603

D5.3 – On-Demand, On-Line Monitoring Infrastructure (b)

WP5: Cross layer resilience and online analysis for non-functional parameters

Version: 1.0
Author(s): Xavier Aguilar (KTH)
Date: 17/05/18

Due date:	PM32
Submission date:	
Project start date:	01/10/2015
Project duration:	36 months
Deliverable lead organization	KTH
Version:	1.0
Status	Draft
Author(s):	Xavier Aguilar (KTH)
Reviewer(s)	Philipp Gschwandter (UIBK) Kiril Dichev (QUB)

Dissemination level	
PU	*Public*

Disclaimer

This deliverable has been prepared by the responsible Work Package of the Project in accordance with the Consortium Agreement and the Grant Agreement Nr 671603. It solely reflects the opinion of the parties to such agreements on a collective basis in the context of the Project and to the extent foreseen in such agreements.

Acknowledgements

The work presented in this document has been conducted in the context of the EU Horizon 2020. AllScale is a 36-month project that started on October 1st, 2015 and is funded by the European Commission.

The partners in the project are UNIVERSITÄT INNSBRUCK (UIBK), FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN NÜRNBERG (FAU), THE QUEEN'S UNIVERSITY OF BELFAST (QUB), KUNGLIGA TEKNISKA HÖGSKOLAN (KTH), NUMERICAL MECHANICS APPLICATIONS INTERNATIONAL SA (NUMECA), IBM IRELAND LIMITED (IBM).

The content of this document is the result of extensive discussions within the AllScale Consortium as a whole.

More information

Public AllScale reports and other information pertaining to the project are available through the AllScale public Web site under http://www.allscale.eu.

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	18/04/18	First draft	Xavier Aguilar
0.2	08/05/18	Added PG feedback	XA, PG
0.3	14/05/18	Added KD feedback	XA, PG, KD
1.0	17/05/18	Final version	XA, PG, KD

Table of Contents

Index of Figures

Executive Summary

This document describes the status and structure of the first prototype of the AllScale Performance Monitoring Component (D5.3). This is a **source code deliverable**.

The document contains instructions on how to obtain and build the prototype as well as a description of its implementation.

1 Introduction

Deliverable D5.3 is a source code deliverable that extends D5.2 and provides the full prototype of the AllScale Performance Monitoring Component. This prototype captures performance data for AllScale Work Items, which are defined in D4.1 and implemented in D4.2. The monitoring infrastructure also provides a holistic view of the performance of the whole application while it runs. In other words, it provides access to the performance data for the whole distributed application during its execution time. Thereby, the AllScale Scheduler can use performance data in its decision making process when scheduling tasks. In addition, the monitoring infrastructure generates several performance profiles that can be used by AllScale users and developers to analyze their applications after program execution.

One of the major differences between the full prototype here presented and the one in D5.2 is its sophisticated implementation to reduce its overhead. While in the first prototype presented in D5.2 each thread processed its own performance data and updated it in a central data structure, the full prototype utilizes a producer-consumer model with double buffering. Thereby, the thread contention and the overhead introduced in the application are reduced considerably. This new design and implementation allowed us for example to reduce the overhead in iPIC3D, one of the pilot codes in the project, from around 80% in the worst case scenario to only 4.5%.

In addition to the more sophisticated implementation, the full prototype includes new metrics and features. For example, new architectures supported for energy monitoring (x86_64 and Cray), sampled metrics at a node level, new performance plots, and support for database storage of the collected data.

The remainder of this document will provide instructions on how to build the prototype as well as more details of the new features implemented.

2 Build Instructions

The AllScale Monitoring Component is embedded within the AllScale runtime on top of HPX, therefore we need to install HPX first. Information on how to install HPX can be found here:
http://stellar-group.github.io/hpx/docs/html/hpx/manual/build_system.html

In addition to the installing instructions that can be found in the previous link, HPX has to be built with the flag –DHPX_WITH_THREAD_IDLE_RATES=On.

If the user wants to collect PAPI counters, PAPI needs to be installed too:
http://icl.cs.utk.edu/papi/

The source code of the prototype can be currently found in github at this URL:
https://github.com/allscale/allscale_runtime

Before compiling the full prototype together with the AllScale Runtime (standard installation), the AllScale API has to be installed too. The AllScale API can be downloaded from github at the following address:
https://github.com/allscale/allscale_api

Once the sources have been obtained either via the Git SCM or a downloadable snapshot, the build process is configured using CMake, which interfaces with the native build tools of the platform used. The monitoring component builds together with the AllScale runtime as well as its accompanying examples.

The code is organized as follows:

- <project root>
  - allscale – all headers needed to interface with the runtime system
  - src – The source files that contain the implementation of the monitoring infrastructure and the runtime system
  - example – Examples showing the use of the runtime system
  - tests – Unit tests to ensure a functional runtime system
  - tools – Several scripts to perform different tasks such as generating reports and plots
  - CMakeLists.txt – cmake build script
  - README.md – short summary and usage hints

For other non-standard installations of the monitoring prototype, for instance with database support, the corresponding dependencies have to be installed beforehand as well: MongoDB, MongoDB-C and C++ drivers, etc.

3 Component Description

3.1 HPX Hooks Used by the Component

The prototype uses the final extended list of hooks presented in D4.3. Old hooks such as *Work Item Execution Start* have been refined into *Split Work Item Execution Start* and *Process Work Item execution Start*, thereby allowing the Monitor Component to have greater control on the granularity of the performance data that is being collected. With these new hooks, the current full prototype can collect information for all Work Items, or just for splittable Work Items, or for the Work Items that are the leaves of the treeture, i.e. Process Work Items.

When one of these actions is intercepted, the monitoring component takes control and performs the proper tasks needed to collect the performance data. Each thread collects its own data and stores it in a local buffer. Once this buffer is full, the data is sent using a double buffering scheme to a dedicated consumer thread that processes such performance data and updates the data structures per process accordingly.

3.2 Metrics Collected by the Component

The prototype also extends the lists of metrics that can be collected and accessed dynamically while the application runs. In addition to the metrics defined in D5.2, the new prototype supports the collection of power and energy data in Power8, Cray XC, and x64 architectures.

The prototype also provides means to collect sampled metrics at a process level. The sampling interval is defined by the monitor component but can be changed by the user using the environment variable SAMPLING_INTERVAL (in milliseconds). The sampled metrics include Work Item throughput (tasks/ms) and thread idle rates.

3.3 Online Performance Introspection

The prototype provides a global view of the performance of the application. In other words, it allows the scheduler to access the performance data of any locality (process) running in a remote node. Therefore, the prototype extends the API described in D5.2 with the corresponding remote counterpart calls.

In addition, it includes new calls to access some new metrics included in the full prototype (sampled metrics, power metrics, etc.). The Introspection API contains the following calls (remote counterparts have the same name but with the suffix *_remote* added at the end):

```
// Returns the exclusive time for a work item with ID w_id
double get_exclusive_time(std::string w_id);

// Returns the inclusive time for a work item with ID w_id
double get_inclusive_time(std::string w_id);

// Returns the average exclusive time for a work item with name w_name
double get_average_exclusive_time(std::string w_name);

// Returns the minimum exclusive time for a work item with name w_name
double get_minimum_exclusive_time(std::string w_name);

// Returns the maximum exclusive time for a work item with name w_name
double get_maximum_exclusive_time(std::string w_name);

// Returns the maximum exclusive time for a work item with name w_name
double get_maximum_exclusive_time(std::string w_name);

// Returns the mean exclusive execution time for all the children of work item
// with ID w_id
double get_children_mean_time(std::string w_id);
```

```
// Returns the standard deviation of the exclusive execution time for all the
// children of work item with ID w_id
double get_children_SD_time(std::string w_id);

// Returns the average exclusive execution time for the last num_work_items
// executed
double get_avg_work_item_times(std::uint_32t num_work_items);

// Returns the standard deviation in the execution time for the last
// num_work_items executed
double get_SD_work_item_times(std::uint_32t num_work_items);

// Returns the idle rate for the last sampling interval in the current locality
double get_idle_rate();

// Returns the locality throughput in tasks/ms
double get_throughput();

// Returns PAPI counters for a work item with ID w_id
double *get_papi_counters(std::string w_id);

// Returns the PAPI counter with name c_name for the work item
// with ID w_id
double get_papi_counter(std::string w_id, std::string c_name);
```

These Introspection calls return 0 in case the work item does not exist or it has not finished its execution yet.

Regarding energy, the API varies depending on the architecture used. For example, the call *get_instant_energy()* in the Cray XC40 architecture returns the current energy consumed by the whole node, whereas in a Linux x64 with Intel RAPL, it returns the current energy of the processor (DRAM not included). Therefore, the user is advised to look into the source code of the respective module in case he/she wants to access energy introspection data.

3.4 Support for DB storage

By default, the performance data is not kept on disk after program execution, i.e. the performance data is mainly collected to be used in real-time by the scheduler during its decision making process. However, the full prototype also provides support to use a distributed database (MongoDB) for permanent storage of such performance data. Furthermore, by using MongoDB we can also make our performance data resilient.

In order to use the database module included in the full prototype, the prototype needs to be configured in CMake with the variables

ALLSCALE_WITH_MONGODB=ON, and MONGOCXX_DIR and MONGOC_DIR pointing to the proper MongoDB C++ and C driver installations respectively.

The use of the distributed database is still experimental, and further experimentation and performance measurements while using the database in large runs are ongoing work.

3.5 Performance Reports at the End of a Run

The first prototype already provides basic performance profiles for quick performance evaluation, for example, text profiles or work item dependency graphs. The full prototype includes also heat maps for the new sampled metrics, i.e. idle rate, task throughput, and energy, thereby, allowing the quick exploration of the evolution and the load balance of the system during execution time. Figure 1 shows the idle rate metric. X-axis contains the samples taken by the Monitoring Component, and the Y-axis the localities (processes). The plot is coloured with a gradient from white to red, white being low value and red high value. It can be seen in the picture that in this case, there is one node (locality 1) much more loaded than the other (the execution had one locality per node).

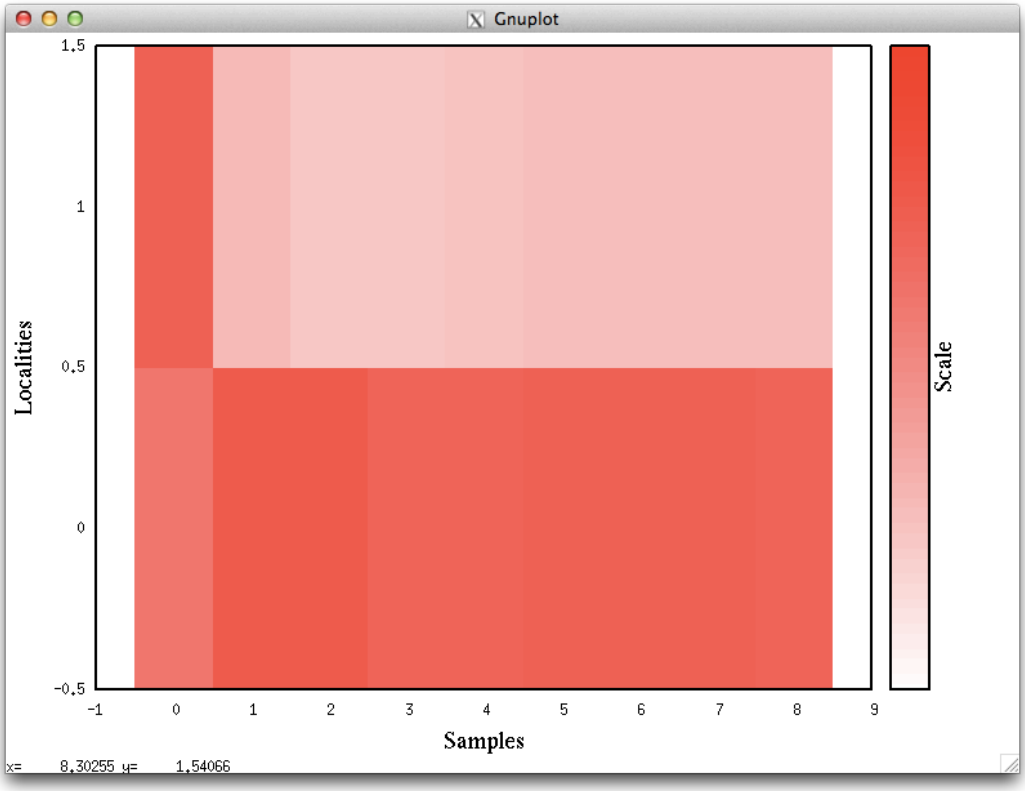

Figure 1: Idle rate heat map. X-axis is samples, and Y-axis localities. The graph is colored from white to red, white being low value and red high value.

4 Summary and Future Work

The full prototype presented in this deliverable is a fully functional feature-complete monitoring infrastructure that allows performance monitoring and introspection in real-time. The prototype provides a global view in real-time across nodes of the performance of the application to the AllScale Scheduler to help in its decision making process when scheduling tasks. In addition, the prototype can provide post-mortem profiles and plots for quick exploration of the performance data collected.

New opportunities of improvement of the full prototype are the addition of more advanced post-mortem reports. For example, traces where the execution of Work Items is mapped in the trace, recording accordingly whether a Work Item migrates across threads during its lifetime. Another interesting field to explore would be the use of the collected data to build performance models of the runtime in order to further improve the execution orchestrated by the AllScale Scheduler.