D3.8 – Final release of adaptive runtime systems

WP3: Development Environment

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exa-scale computing, software and simulation

Due date:	M38
Submission date:	30/11/2014
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	KTH
Version:	1.0
Status	Final
Author(s):	Xavier Aguilar (KTH), Michael Schliephake (KTH)
Reviewer(s)	Weronika Filinger (UEDIN), Ulf Schiller (UCL)

Dissemination level	
<PU/PP/RE/CO>	*PU*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	14/11/2014	First version of the deliverable	Aguilar, Schliephake (KTH)
0.2	27/11/2014	First revision of the deliverable	Aguilar, Schliephake (KTH)
1.0	28/11/2014	Final version for submission	Lorna Smith (UEDIN)

Table of Contents

1 Executive Summary

In the CRESTA project, part of the effort was devoted to designing different frameworks in the CRESTA development environment of exascale applications. This deliverable is a software deliverable, delivering a final release of the runtime environment. In this associated report, we present the installation instructions for the runtime system environment developed within the CRESTA project.

The runtime environment developed within this project has been shown to benefit applications on the medium- to large- scale and represents a significant step on the software's roadmap towards the utilisation with applications on the scale of future exascale platforms.

2 Introduction

2.1 Purpose

The purpose of this deliverable is to describe the runtime system that has been produced in the CRESTA project. The software is on the CRESTA Subversion repository. The purposes of this associated document are:

- to motivate the purpose of the software and its relevance to co-design briefly in section 3
- to describe the installation and compilation of the runtime system in section 4, and
- to give a short overview of the runtime system software developed during the CRESTA project in section 5.

2.2 Glossary of Acronyms

D	Deliverable
EC	European Commission
EPCC	Edinburgh Parallel Computing Centre
PM	Project Manager
WP	Work Package
Mon-C	Monitoring component of the runtime system

3 Purpose and relevance of the runtime system

3.1 Rationale for the development of a runtime system

The rationale for the development of this runtime system has already been described in the design document of the runtime system (deliverable D3.2 [7]) and was summarised in deliverable D3.7 [6]. For convenience we reproduce the text from D3.7 here.

Massive parallel computing is a major driving force in computational science and scientific discovery and the systems are permanently getting larger and more complex. Future exascale systems will be composed of hundreds of thousands of cores and will have complex designs that are likely to use heterogeneous technologies. It will, therefore, be a challenging task to achieve good application and system performance. In addition, the increasing complexity of these machines will also increase the complexity of the applications and operating systems.

These new kinds of heterogeneous systems pose new challenges in the development and porting of applications, and require significant effort to achieve the systems peak capability. Human experts who optimise and port applications for these systems need to be complemented with intelligent software tools providing support in a transparent and automated way.

In order to achieve good performance, typically highly system-specific features have to be exploited, which often means that best practices in programming and software development have to be relaxed and the resulting code is difficult to port to different systems. Runtime systems help to build portable applications for a broad range of HPC infrastructures in a modular way. The heterogeneous character of recent hardware as well as the parallel program's highly dynamic behaviour not known before their execution require runtime systems to take into consideration the hardware topology as well as monitoring information of the on-going program execution. The runtime system consists therefore of a resource manager, a library for runtime administration of parallel applications, and a performance monitoring and analysis tool. The design is based on a task-oriented programming model.

Requirements. One of the hardest requirements in the development of simulation applications is their adaptation to different computer systems due to the varying technical parameters that have a huge influence on the numerical performance: cache and memory hierarchies, the number of cores per CPU, the number of sockets per node, and the characteristics of the interconnect network.

Today, optimisations are typically implemented directly in the code. The necessary effort to do this will grow immensely in the future due to the increasing heterogeneity and diversity of HPC computer systems. A runtime system must aim to improve the performance portability that can be achieved with one certain implementation.

An important requirement for tool development is the reuse of existing application codes often implemented in Fortran or C. The introduction of new software tools should allow its incremental adoption, keeping the need for reimplementation or adaptation of existing code to a minimum. A further requirement connected to the previous one is the wish that software tools support an adaptive use of best practices, which otherwise would not be applied due to prohibitive implementation effort. Finally, given that hybrid programming models gain more importance, the runtime system may not prevent the use of parallelisation technologies that it does not address itself.

While different application classes put different requirements on runtime support, we focus on numerical simulation applications. Typical requirements of numerical simulations are:

- Integration of data and task parallelism,
- Use of multi-level parallelism in the algorithm design,

- Development of algorithms with a high degree of parallel executable tasks, which have a moderate size, can be created very quickly, and avoid global communication operations,
- Usage of multi-threading, asynchronous communication and one-sided communication,
- Consideration of the increasing depth of the memory hierarchy,
- Optimised scheduling and mapping taking into account chip-architectures, memory hierarchies, internal communication abilities, etc. to provide a higher degree of parallelism and decrease memory and communication bandwidth usage.

3.2 Relevance of the runtime system

In co-design terms, the development of the runtime system benefited significantly from access and input from the real simulation applications in the project. This is an example of the final software tools being of a better quality than they would otherwise have been without the applications. In particular, the application NEK5000 provided input and direction as to how scalable communication could be implemented. A module for fast collective communication operations has been developed based on the existing implementation in NEK5000. These functions will not only be used in the runtime system itself. This component will be used by simulation programs for neuroinformatics as well as quantum chemistry, as these need a significant reduction in communication latency as well. Hence the final software is of benefit to other applications beyond NEK5000.

The runtime system itself provided the expected performance improvements of about 10% measured in wallclock time for tested applications. These have been measured in tests up to several hundred nodes[6]. This means that the software can be used for practical purposes in medium- to large- scale simulation applications. This is also a significant milestone on the software's roadmap towards utilisation on exascale scale applications. The next stage in the software's roadmap to exascale is to improve the distributed mapping and scheduling algorithms as well as further improvements of the task model.

4 Runtime System

4.1 Download and installation

The software is available from the CRESTA software repository hosted at EPCC. Its URL is

https://svn.ecdf.ed.ac.uk/repo/ph/cresta/wp3/runtime

The archive file there can be downloaded and uncompressed on the local system. The software package consists of several modules that are located in different directories of the file tree. The compilation will be done in two steps. The monitoring component will be compiled first. The second step is the compilation of the runtime administration component.

4.2 Installation of the Runtime Monitoring Component (Mon-C)

The monitoring component is located in the subdirectory `mon-c` of the file tree provided in the archive file.

Installing the monitoring component follows the normal procedure of configuring the package with "configure CFLAGS=-DHAVE_RDTSC", building it with "make" and installing it with "make install". In most cases that would be sufficient, however, some systems may require the use of different flags in the configuring process.

Example of *configure* in a Cray XE6 system installing the basic functionalities:

```
./configure --prefix=/opt/monC CC=cc F77=ftn CFLAGS=-DHAVE_RDTSC
```

More configuration options can be obtained by executing:

```
./configure --help
```

If Hardware Performance Counters are available in the target system through the PAPI interface [3], it is recommended to configure the monitoring package to use them by adding the flag "—with-papi" in the configure process:

```
./configure --with-papi=/opt/papi/4.3/ --prefix=/opt/monC CC=cc
F77=ftn CFLAGS=-DHAVE_RDTSC
```

Afterwards, the user has to add the flag "-DHAVE_PAPI" to the runtime compilation flags.

4.3 Installation of the Runtime Administration Component (Rta-C)

The monitoring component is located in the subdirectory `rta-c` of the file tree provided in the archive file.

The component itself consists of several sub-components located in subdirectories of `rta-c`. The list of components is

devtools	General tools used in Rta-C
mampicl	Implementation of latency-optimised communication routines
rts	Routines for ask management and relocation

All sub-components can be configured and compiled separately if needed. Normally, this is not needed.

The configuration will be done by adaptation of a file named `custom.mk` that is contained in the directory of each sub-component. Definitions of symbols that control the compilation can be made here. Each of the `custom.mk` files in the sub-components includes again another file `custom.mk` from its parent directory. The parent directory is the directory `rta-c`, i.e. the main directory of the Rta-C component. General settings for all subcomponents can be defined at a central place in that way.

The files custom.mk will be included into makefiles during the compilation process. They can therefore contain everything that is allowed according to the syntax of makefiles – symbols, functions, rule definitions etc. Information about possible definitions and typical settings are given in the provided example files. In most cases, only specific definitions should be needed in case the system will be linked together with certain tools having specific requirements on the compilation. To this end, the symbols `CCFLAGS_USER`, `CXXFLAGS_USER`, and `LDFLAGS_USER` can be defined. They will be used to extend the definitions of the typically present symbols `CFLAGS`, `CXXFLAGS`, and `LDFLAGS`. The introduction of these symbols minimises the risk of unwanted inference with predefined default options from software environment modules or other systems and user-specific environment configurations. Furthermore, the compilation of modules with the MPI compilers uses the definitions from the additional modules `MPICFLAGS_USER`, `MPICXXFLAGS_USER`, and `MPILDFLAGS_USER`.

Prerequisites for the compilation:

- The names of the compilers have to be defined in environment variables. These variables are `CC` for the C-compiler, `CXX` for the C++-compiler, `MPICC` for the MPI-C-compiler and `MPICXX` for the MPI-C++-compiler. C modules can be compiled also with the C++ compiler if desired.
- The library Scotch [4] has to be provided on the system. Two symbols have to be defined in the `custom.mk` file of Rta-C – `SCOTCH_INCLUDE`, which provides the path of the directory with the included files and `SCOTCH_LIB`, which provides the path of the directory with the compiled library.

Having done the configuration, the compilation could be done entering every sub-directory and executing the make command. While this will be used when developing within Rta-CA, a one-step compilation is provided in the sub-directory `build`. The makefile there will recursively compile all sub-components of Rta-C at once.

The compilation can be performed for an arbitrary number of different configurations. The makefiles have been prepared for two configurations: `release` and `debug`. The outcome from the compilation for these configurations is a library for normal use respective libraries for debugging purposes. The configuration `release` will be used as default, the compilation for `debug` will be triggered by the symbol definition `DEBUG`, for example by the command "`make DEBUG=1`".

The compilation for a certain configuration ensures that the object and library files as well as other intermediate data of the compilation product will be saved in a subdirectory of the component with the name of the configuration. Configuration names other than release and debug will be defined with the symbols `MK_CONFIG`. For example "`make MK_CONFIG=spec-conf-1`". The compilation process can be customised for a certain configuration then again in the `custom.mk` files where it is possible to provide conditional definitions based on the `MK_CONFIG` setting.

The library of the compiled Rta-C that can be linked to applications and the header files containing the external definitions will be available after the compilation in the subdirectories `lib` respective `include`.

5 A short overview of the runtime system

This section provides an excerpt from the deliverable D3.7 that presented the CRESTA development environment. The excerpt here reproduces the descriptions of the components of the runtime system. More details about the approach, programming model, and performance measurements can be found in Deliverable D3.7.[6]

5.1.1 User interface of the runtime system

The user interface consists of five elements that will be used by an application developer. These comprise the definition of computational tasks, support for the automatic re-mapping of distributed arrays as well as user-defined data that define the state of a computational task, a control function to perform dynamic load-balancing, and the management of MPI communicators. This interface has been designed with simplicity in mind in order to allow its convenient introduction into existing simulation codes. Furthermore, the current implementation of the user interface to the runtime system, focusing on the dynamic gradual improvement of task mappings and load-balance, allows developers to optionally deactivate automatic load-balancing if a certain computer system cannot be well supported for some reason. This can be compared to a parallelisation similar to the OpenMP approach that allows compiling a program optionally with OpenMP support.

The definition of tasks: A task is identified by a key. The possibility to define tasks hierarchically leads naturally to a tree structure of tasks. Examples of task keys are `"task_1"` or `"task_1/subtask_a"`. A number of parameters for a task can be specified in a structure `task_params`. These comprise the number of allocated processors, estimations of computational work and communication, and a callback function that is used to serialize respective de-serialize the task status. The following function can be used for the definition of the task tree.

```
task_tree *define_task(task_tree *parent, int n,
                       string *keys, task_parms *children);
```

One call to `define_task()` adds the number of `n` sub-tasks to the parent task that are allowed to run in parallel. Several calls to this function define a sequence of tasks. Arbitrary task trees may be constructed in that way.

The beginning and end of the execution of a certain task can be registered in the code by calls to the functions `begin_task(key)` resp. `end_task(key)`.

Support for re-mapping of tasks: It is necessary to transport the state of a computational task between processes in order to move tasks during the runtime. The application developer has to provide a function that can serialise resp. de-serialise the state of a computational task. The state of a task will be serialised in the owner process of a task, transported to the destination process, and finally de-serialised there. A ready-to-use convenience implementation is provided for the transport of arrays, which are one of the most frequently-used data structures in numerical simulations. Arrays can be registered at their owning computational tasks and will be handled by the runtime system automatically. This avoids repeating coding tasks of serialisation for the developer as well as allowing optimised handling of memory allocations. The prototype for callbacks is defined as

```
int (*task_serialisation_cb)(int opcode, void *buffer);
```

The value of `opcode` defines whether the requested operation is a serialisation or a de-serialisation, and `buffer` is used to store the serialised data to write or read from. The registration of arrays at their owning tasks will be done with the function

```
void register_array(task_tree *owner, void *array,
                    int dtype, int n, int *dims);
```

which specifies the dimensions of the array and the type of its elements.

Support for the management of MPI communicators: The runtime system supports the re-mapping and execution of computational tasks by means of MPI, whereas the choice of the inner parallelisation technique for multi-processor tasks is under control of the application developer. This would lead to the need of maintaining a directory of task mappings onto MPI ranks in order to perform communication between the owning processes of computational tasks in need of message-passing. The design choice for the runtime system was to avoid explicit bookkeeping. MPI communicators will be used for that instead. Initially when setting up the calculation in typical numerical simulations, processes determine their communication partners rank-wise. These ranks are defined often as global properties of the MPI processes and updated occasionally, for example when re-distributions of data occur. In a program running under the control of the runtime system, however, the rank numbers of the communication partners become part of the state of computational tasks. They will be moved together with the other data defining the state of a task and used in all subsequent communication operations until an update is required due to re-distributions of data initiated by the simulation application itself. It is the responsibility of the runtime system to provide an MPI communicator to the application that reflects updated mappings of computational tasks onto MPI processes after their re-distribution. This functionality has been implemented by means of communicator management functions as provided by the MPI-2 standard. From an application developer's point of view, the programmer defines a so-called load-balancing context that connects the group of a certain MPI communicator with a sub-tree of the task tree. Load-balancing will then be performed amongst the participating MPI processes of the context's communicator. The load-balancing context is defined by using the function

```
MPI_Comm *define_lb_context(MPI_Comm comm,
                            task_tree *root_task);
```

The function returns an MPI communicator for communication operations using the previously defined rank numbers of partner processes.

Developer control of the load-balancing process: The runtime system monitors the execution of a parallel program. It is necessary from time to time to hand over the control to the runtime system. A new task mapping is then calculated based on the previous monitoring. The callbacks specified during the definition of tasks will be activated for the serialisation and deserialization of tasks, and the runtime system manages the transport of these data between the processes. Finally, a new MPI communicator reflecting the new task distribution will be created and returned to the application for subsequent use in communication between the computational tasks. The user triggers these activities at suitable points in time by calling the following function, which also returns the new MPI communicator to the application.

```
void perform_load_balancing(MPI_Comm *comm);
```

5.1.2 Software architecture

The runtime system consists of three main components: a runtime administration component (Rta-C) schedules tasks and monitors their execution status; a monitoring component (Mon-C) provides information on the hardware utilisation, which is for scheduling decisions as well as to complement potentially incomplete or imprecise resource requirement specifications; and finally a performance analysis component (Pan-C) that analyses recorded monitoring data to provide more sophisticated hints for application control, beyond the capabilities of single run monitoring (see figure 1). Implementations of the components Rta-C and Mon-C have been realised within CRESTA. The performance analysis component (Pan-C) as well as the data storage in the performance database will be developed in future projects.

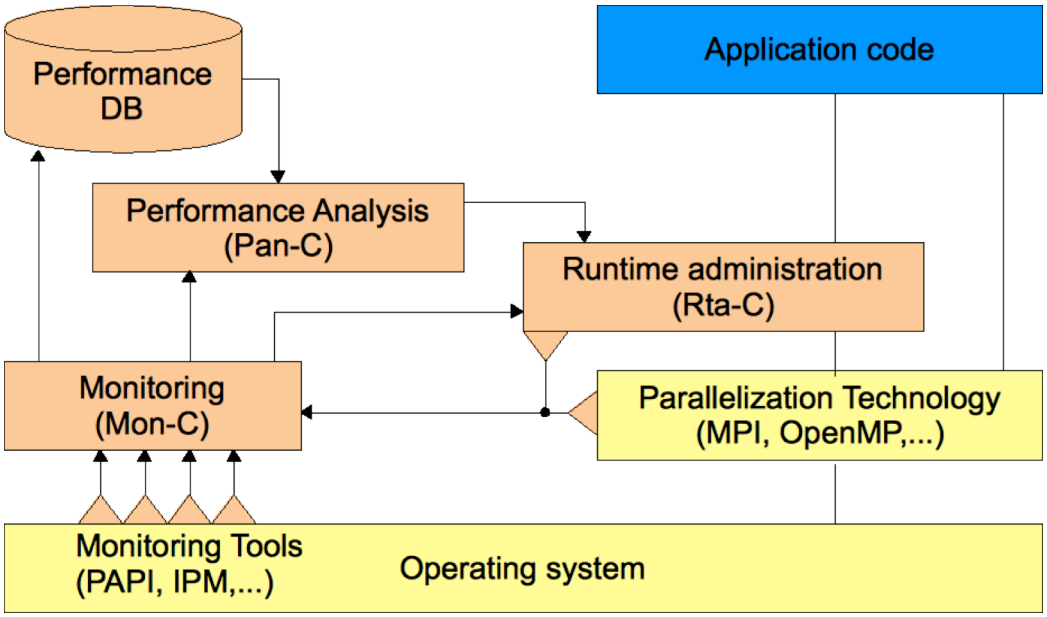

Figure 1: Components of the runtime-system

5.1.3 Runtime administration component (Rta-C)

Rta-C provides the user API allowing the definition of computational tasks as well as control of the load-balancing execution. It maintains internally the task tree as well as the hardware model. This component receives monitoring data from the monitoring component (Mon-C). Furthermore, it comprises the mapping algorithm as well as the functionality for moving tasks.

Rta-C creates the task tree within each process from the task definitions provided by the application. The cost estimates for computational work and communication volumes provided in the task definitions will be used for the calculation of task mappings on platforms that do not have the capability to monitor these parameters during the execution. Otherwise, these values will be replaced by data acquired by Mon-C as described below.

The hardware model is either provided as a static graph with weighted nodes and edges representing computing and communication capabilities, or constructed dynamically during the program execution. The latter is done by measuring communication capabilities during the runtime of the parallel program. The advantage of this approach is that the real communication performance of the nodes allocated to a batch job is determined at the moment of the measurement. Influences from a concrete load on an HPC system as well as effects of dynamic routing configurations can be taken into account in this way. Even occasional updates of the hardware model are possible during long-running simulations.

Mon-C is clocked by the start and end markers of computational tasks. It provides at least timing information about the execution of the computational tasks. Counter values of executed floating-point operations and MPI communications will be provided if available on the platform. Rta-C maintains a record of these measurements. This is for the time being a moving average value of a configurable number of time steps. The monitoring data are used to update the task definitions and provide in that way an up-to-date picture of the workload during the recent time steps.

The re-mapping of tasks in order to improve the load-balancing can be triggered by the application explicitly or automatically when a certain degree of load-imbalance has been reached. The wall clock time per time step is used as metric of load-imbalance.

The task mapping is calculated in a two-phase process. The implementation of this functionality has been based on the library SCOTCH, which provides extensible algorithms for graph partitioning and mapping. The mapping of tasks to compute nodes is defined in the first phase. Compute nodes are represented for this calculation in the hardware graph as single nodes with a heavier weight according to the number of cores per node. The results of this calculation are task groups that will be assigned to one node. Afterwards, the mapping of task groups onto the different cores of a node is decided during the second phase. These calculations will be done in parallel on each node. The final task mapping is then distributed in order to allow the reconfiguration of the MPI communicator used in the load-balancing context.

Program example for the definition of tasks and the activation of the load balancing during the program execution. This example is taken from a molecular dynamics simulation that implements a linked cell algorithm. Each MPI process is responsible for the calculations needed in a certain sub-domain. The load-balancing will be used to exchange tasks between processes in order to achieve equal computational load that implies similar numbers of memory accesses, which is the limiting factor in this memory bound application.

The following function shows the initialisation of a few computational tasks. The task “timeIntegr” embraces three sub-tasks “compF”, “compX”, “compV” used for the computation of forces between particles as well as their speed and location. Furthermore, we initialise a load balancing context. The returned MPI communicator uses the task mapping in order to provide the application communication possibilities without keeping track of the task mapping itself.

```
task_tree *timeIntegr, *compF, *compX, *compV;
MPI_Comm  simm_comm;

int (*task_serialisation_cb)(int opcode, void *buffer)
{

    if ( opcode == RECV_TASK ) {
        // convert buffer content into local task data
    }

    else if ( opcode == SEND_TASK ) {

        // convert local task data into buffer content
    }

    return 0;
}

void moduleInit_md_simulation()
{

    task_parms parms;

    parms.send_task_cb = &move_cb;
    parms.recv_task_cb = &move_cb;

    timeIntegr = define_task(NULL, 1, "Time Integration, NULL);

    compF = define_task(timeIntegr, 1, "Compute F", &parms);
    compX = define_task(timeIntegr, 1, "Compute X", &parms);
    compV = define_task(timeIntegr, 1, "Compute V", &parms);

    simm_comm = *define_lb_context(MPI_COMM_WORLD, timeIntegration);

}
```

The next code fragment shows the use of the previously defined tasks. Here, only the start and end of tasks has to be marked by calls to the corresponding functions. Time and performance measurements are done in the background.

```
real timeIntegration_LC(MD_Sim *sim, real t, real t_end, real delta_t)

{
```

```
    if (t >= t_end)
        return t;

    World *world = sim->world;
    Cell *grid = world->sd->grid;

    if (sim->initialized == 0)
    {
        compF_LC(grid, world->sd);
        sim->initialized = !sim->initialized;
    }

    begin_task(timeIntegration);
    while (t < t_end)
    {
        t += delta_t;

        begin_task(compX);
        compX_LC(grid, world->sd, delta_t);
        end_task(compX);


        begin_task(compF);
        compF_LC(grid,world->sd);
        end_task(compF);

        begin_task(compV);
        compV_LC(grid, world->sd, delta_t);
        end_task(compV);

        compoutStatistic_base(p, N, t);
    }
    end_task(timeIntegration);

    return t;
}
```

This function implements one time step. The hierarchical structure of the tasks can be recognised easily. The runtime system aggregates the performance measurements over a certain user-defined number of time steps and uses the collected information for the re-mapping of tasks. This can be seen in the implementation of the calling function that manages the timestepping. Tasks will be re-mapped every LOADBAL_ INTERVAL time steps. During this re-mapping, the callback function defined above will be called in order to serialise and de-serialise the task data that needs to be moved. This serialisation is completely free for the application and can be adapted as needed. The activation of the load balancing will also update the load balancing context, i.e. the MPI communicator used by the tasks. Therefore, it is important for applications not to duplicate this handle respectively to keep copies synchronised with the output of the function perform_load_balancing.

```
#define LOADBAL_INTERVAL 100

void doSimulationRun(MD_Sim *sim)
{
    World *world = sim->world;

    log_msg("PROGRESS: Starting time integration\n");
    t_end_global = fminl(world->t_end, t_end_global);

    outputResults_LC(BaseName, 0, sim->t_cur, sim);
    saveCheckpoint(CheckpointName, 0, sim);
    mapProcesses(sim);

    for (int i = 0; i < world->n_out; i++)
    {
        real t_new, t_next_end;
        if (sim->t_cur >= t_end_global) break;
        t_next_end = fminl(t_end_global, world->t_out[i]);

        if (sim->t_cur >= t_next_end) continue;
```

```
        if (i%LOADBAL_INTERVAL == 0)
            perform_laod_balancing(&simm_comm);

        t_new = timeIntegration_LC(sim, sim->t_cur, t_next_end,
                                   world->delta_t);

        outputResults_LC(BaseName, i + 1, t_new, sim);
        sim->t_cur = t_new;

        saveCheckpoint(CheckpointName, i + 1, sim);

    }

    log_msg("PROGRESS: Finished time integration\n");

}
```

5.1.4 Monitoring component (Mon-C)

The monitoring component in the runtime system uses the Integrated Performance Monitoring (IPM) tool [1] to capture the performance behaviour of MPI applications. IPM provides reports on several program events introducing minimum overhead. Such events can be MPI operations, Posix-I/O file operations, CUDA, or OpenMP events among others. IPM has been widely used by HPC centres such as NERSC to collect more then 310K batch profiles in the past 6 years.

WP3 has extended the IPM monitoring tool with the Performance Introspection API (PIA) [2] to provide online feedback to the runtime system as the application runs. This API is designed to be a simple and lightweight interface written in C that can be used from C, C++, and Fortran. The Performance Introspection API provides for each process a local view of its own performance behaviour through the access to two different data entities, *user-defined code regions* and *activities*.

User-defined regions are measurement intervals defined by the runtime system within the application, for instance, tasks, functions, or blocks of code. These delimited regions can be nested and are annotated in the source code with the routine *ipm_region*. For each one of these regions the associated performance data is fixed and includes performance metrics such as wall clock time of the region, MPI time, number of executed instances for that region, and hardware performance counters. As all these metrics are accumulated during program execution, the amount of memory needed to store them is small, in the order of a few kilobytes. The following code listing shows how to use the Performance Introspection API to access the total time, MPI time, and number of executed instances for a defined region called *foo*.

```
void foo( )

{
  // Defining region start
  ipm_region( IPM_START, "foo");

  // Do whatever here

  // Defining region end
  ipm_region( IPM_END, "foo");

}

int main( int argc, char *argv[])
{

   pia_regid_t id;       // Stores region ID
   pia_regdata_t data;   // Stores region data

   foo( );

   // Obtain region ID
   id = pia_find_region_by_name("foo");
```

```
    // Obtain performance data for that region
    pia_get_region_data(&data, id);

    fprintf( stderr, "%f Walltime\n", data.wtime);
    fprintf( stderr, "%f MPI time\n", data.mtime);
    fprintf( stderr, "%d times executed\n", data.count);

}
```

The other entity the runtime can access using the Performance Introspection API is activities. Activities are statistics associated with certain program events such as MPI calls, Posix-IO calls, or OpenMP phases. For instance, the runtime can consult the activity MPI_Recv, obtaining the total number of times the call has been executed, total time inside the call, maximum and minimum execution time, or number of bytes received for the whole run or for a certain defined region. Activities are accessed through their activity ID as shown in the following code snippet:

```
// Activity name
chat *act1 = "MPI_Send";

// Activity ID
pia_act_t id;

// Activity data
pia_actdata_t data;

// Acces the data
pia_init_activity_data(&adata);
id = pia_find_activity_by_name(act1);
pia_get_activity_data(&data, id);

fprintf(stderr,
        "MPI_Send happened %d times and "
        "transferred %d bytes, adata.ncalls, adata.nbytes);
```

6 References

[1] *The Integrated Performance Monitoring tool (IPM).* www.ipm2.org

[2] Xavier Aguilar and Erwin Laure and Karl Fürlinger: *Online Performance Introspection with IPM.* In *High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on*. IEEE, 2013.

[3] *The Performance Application Programming Interface (PAPI)* http://icl.cs.utk.edu/papi/

[4] Francoise Pellegrini and Jean Roman: *SCOTCH - A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs.* In *Proceedings of HPCN'96*, Brussels, Belgium. LNCS 1067, pages 493-498. Springer, April 1996.

[5] Michael Schliephake and Xavier Aguilar and Erwin Laure: *Design and Implementation of a Runtime System for Parallel Numerical Simulations on Large-Scale Clusters.* In Procedia Computer Science, Volume 4, *Proceedings of the International Conference on Computational Science*, ICCS 2011, 2011, Pages 2105-2114.

[6] *Frameworks for Exascale Applications.* Project CRESTA, Deliverable D3.7 (2014).

[7] *Adaptive runtime-support design document.* Project CRESTA, Deliverable D3.2 (2012).