ExaQUte

Exascale **Q**uantification of **U**ncertainties for **Te**chnology and Science Simulation

D4.2 Profiling report of the partner's tools, complete with performance suggestions

Document information table

Contract number:	800898
Project acronym:	ExaQUte
Project Coordinator:	CIMNE
Document Responsible Partner:	BSC
Deliverable Type:	Software
Dissemination Level:	Public
Related WP & Task:	WP 4, Task 4.3
Status:	Final version

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No **800898**

Authoring

Prepared by:				
Authors	Partner	Modified Page/Sections	Version	Comments
Ramon Amela	BSC		V0.1	
Rosa M. Badia	BSC		V0.1	
Stanislav Böhm	IT4I		V0.2	
Riccardo Tosi	CIMNE		V0.3	

Change Log

Versions	Modified Page/Sections	Comments
V0.1	First document version	
V0.2	Inputs section 4	
V0.3	Review section 2	

Approval

Aproved by:				
	Name	Partner	Date	OK
Task leader	Rosa M. Badia	BSC	23.05.19	OK
WP leader	Rosa M. Badia	BSC	23.05.19	OK
Coordinator	Riccardo Rossi	CIMNE	30.05.19	OK

Executive summary

This deliverable focuses on the profiling activities developed in the project with the partner's applications. To perform this profiling activities, a couple of benchmarks were defined in collaboration with WP5. The first benchmark is an embarrassingly parallel benchmark that performs a read and then multiple writes of the same object, with the objective of stressing the memory and storage systems and evaluate the overhead when these reads and writes are performed in parallel.

A second benchmark is defined based on the Continuation Multi Level Monte Carlo (C-MLMC) algorithm. While this algorithm is normally executed using multiple levels, for the profiling and performance analysis objectives, the execution of a single level was enough since the forthcoming levels have similar performance characteristics. Additionally, while the simulation tasks can be executed as parallel (multi-threaded tasks), in the benchmark, single threaded tasks were executed to increase the number of simulations to be scheduled and stress the scheduling engines.

A set of experiments based on these two benchmarks have been executed in the MareNostrum 4 supercomputer and using PyCOMPSs as underlying programming model and dynamic scheduler of the tasks involved in the executions.

While the first benchmark was executed several times in a single iteration, the second benchmark was executed in an iterative manner, with cycles of 1) Execution and trace generation; 2) Performance analysis; 3) Improvements. This had enabled to perform several improvements in the benchmark and in the scheduler of PyCOMPSs.

The initial iterations focused on the C-MLMC structure itself, performing re-factors of the code to remove fine grain and sequential tasks and merging them in larger granularity tasks. The next iterations focused on improving the PyCOMPSs scheduler, removing existent bottlenecks and increasing its performance by making the scheduler a multi-threaded engine. While the results can still be improved, we are satisfied with the results since the granularity of the simulations run in this evaluation step are much finer than the one that will be used for the real scenarios.

The deliverable finishes with some recommendations that should be followed along the project in order to obtain good performance in the execution of the project codes.

Table of contents

List of Figures

Nomenclature / Acronym list

Acronym	Meaning
API	Application Programming Interface
ExaQUte	EXAscale Quantification of Uncertainties for Technology and Science Simulation
DAG	Directed Acyclic Graph
FILE_IN	Path to a file passed to a function that is not modified
FILE_INOUT	Path to a file passed to a function that is modified during the call
FILE_OUT	Path to a file passed to a function that is created during the call
HPC	High Performance Computing
IN	Parameter of a function that is not modified
INOUT	Parameter of a function that is modified during the call
OpenMP	Open Multi Processing
MPI	Message Passing Interface
PBS	Portable Batch System
PyCOMPSs	Python binding for COMPS Superscalar
SLURM	Simple Linux Utility for Resource Management

1 Introduction

One of the challenges addressed by the project ExaQUte is the dynamic scheduling of Multi Level Monte Carlo (MLMC) [1–4] workloads aiming at achieving good efficiencies in distributed computing platforms.

In ExaQUte, the orchestration of the computations and the management of the statistical outputs will be done at a very high level employing a Python layer, which will also be in charge of handling any robustness issues (both related to hardware failure or to lack of convergence in the solvers). This Python layer run on top of PyCOMPSs [5, 6], a task-based programming model that enables automatic parallelization and execution of the codes in distributed computing platforms, and on top of HyperLoom [7], a dynamic scheduler also able to run a large number tasks in the same type of platforms.

The objective of ExaQUTE WP4 is the deployment of programming models and related runtimes HyperLoom and PyCOMPSs on the supercomputer infrastructure, their configuration and optimization.

The task 4.3 focuses at the performance analysis of partners' applications through post-portem profiling. This deliverable shows the results obtained in the task that has gone through an incremental process of testing initial algorithms and analysing its performance, proposing improvements and testing again.

Section 2 describe the algorithms that have been used in the experiments. Section 3 describe the results obtained with PyCOMPSs and by doing performance analysis with Extrae and Paraver. Section 4 presents the results obtained with HyperLoom. Finally, section 5 presents some recommendations to follow in the algorithms' development in order to achieve good performance.

2 Experiments description

The goal of the task is to perform profiling and performance analysis of the project applications with the objective of providing feedback to the partners involved in their development.

The profiling and performance analysis of the examples run with PyCOMPSs have been performed with the BSC set of performance tools. More specifically, the Extrae library is used to generate post-mortem tracefiles of the execution of the applications that can be later analysed with the Paraver performance analysis tool[1]. In the case of PyCOMPSs, the runtime is instrumented to automatically generate the traces as an execution option, but Extrae is a generic tool that can be used to generate tracefiles of all kind of applications.

The focus of the analysis has been in special on the task scheduling and how the different bottlenecks that this scheduling can cause. The analysis has been based in a set of experiments performed with different algorithms that represent the codes under development in the project. This section describes the algorithms used in the experiments.

[1]https://tools.bsc.es

2.1 Algorithm description

A first version of the Monte Carlo (MC) algorithm was defined. This algorithm performs a conditional loop until a convergence criteria is met. In the first iteration, an initial set of samples (initial hierarchy) is defined. This set of samples is updated in each iteration, and the number of samples under evaluation in each iteration may change. For each iteration, the Quantity of Interest (QoI) values are evaluated, by throwing for each random value a simulation. When all simulations of the iteration have finished, the statistics, such as expectations and variances of the results, are evaluated. This phase requires a reduction of the results generated by the simulations by accumulating them. Finally the convergence is evaluated to decide if a new iteration is necessary. The description of the sequential MC algorithm is described in Algorithm 1.

Algorithm 1: Sequential MC

```
begin
    given initial hierarchy $N, M$
    while convergence is not True do
        $N \longleftarrow N(N, it)$
        for $i = 0 : N$ do
            $QoI_M \longleftarrow solver(w^{(i)})$
        compute statistics $\mathbb{E}^{\mathrm{MC}}[QoI_M]$, $\mathbb{V}\mathrm{ar}^{\mathrm{MC}}[QoI_M]$, $\cdots$
        estimate convergence
        $it = it + 1$
```

While the previous is the simple version of a Monte Carlo algorithm, in the project the partners will focus their experiments on the Multi Level Monte Carlo and Continuation Multi Level Monte Carlo (C-MLMC) algorithms. MLMC and C-MLMC algorithms basically differ from the sequential MC because they exploit a hierarchy of levels. These levels present an increasing computational cost and an increasing accuracy, and at each iteration we compute the simulation with finer granularity of the models (levels). This enables a more efficient exploration of the design space only simulating for finer levels of granularity the areas of interest. The description of the MLMC and C-MLMC algorithm is described in Algorithm 2.

These are the algorithms (defined in collaboration with WP5) taken as basis to encode the distributed versions which are then profiled and analysed . In both cases the algorithm executes sets of Kratos simulations that generate the QoI values. These values will depend on the application and the function to optimize. Nevertheless, the general schema is the same in both cases. First of all, some simulations are launched. Afterwards, the useful information to compute the QoI is extracted. Afterwards, these extracted values are accumulated in order to obtain the convergence criteria. Finally, a stop criterion is checked in order to know if more simulations are needed to obtain the desired precision.

2.2 Description of the experiments

Considering the nature of the algorithms, the experiments have been designed in order to incrementally check the possible bottlenecks of their distributed execution. More precisely, the following steps have been followed:

Algorithm 2: Standard MLMC/CMLMC

```
begin
    given initial hierarchy $N, M, L$
    for $l = 0 : L$ do
        for $i = 0 : N_l$ do
            generate r.v. $w^{(i,l)}$
            $QoI_{M_l} \longleftarrow solver(w^{(i,l)})$
            $QoI_{M_{l-1}} \longleftarrow solver(w^{(i,l-1)})$
            $Y_l^i = QoI_{M_l} - QoI_{M_{l-1}}$
    compute statistics $\mathbb{E}^{\text{MC}}[Y_l], \mathbb{V}\text{ar}^{\text{MC}}[Y_l], \mathbb{E}^{\text{MLMC}}[QoI_M], \cdots$
    compute parameters $\mathcal{P}$
    $\mathcal{I} \longleftarrow \mathcal{I}(\varepsilon, \varepsilon_0)$ (CMLMC)
begin
    while *convergence is not True* do
        $\varepsilon_{it} \longleftarrow \varepsilon_{it}(it)$ (CMLMC)
        $\varepsilon_{it} \longleftarrow \varepsilon$ (MLMC)
        $L \longleftarrow L(\theta, \varepsilon_{it}, \mathcal{P})$
        $N \longleftarrow N(\phi, \theta, \varepsilon_{it}, \mathbb{V}\text{ar}^{\text{MC}}[Y_l], L)$
        for $l = 0 : L$ do
            for $i = 0 : N_l$ do
                generate r.v. $w^{(i,l)}$
                $QoI_{M_l} \longleftarrow solver(w^{(i,l)})$
                $QoI_{M_{l-1}} \longleftarrow solver(w^{(i,l-1)})$
                $Y_l^i = QoI_{M_l} - QoI_{M_{l-1}}$
        compute $\mathcal{P}$
        compute statistics $\mathbb{E}^{\text{MC}}[Y_l], \mathbb{V}\text{ar}^{\text{MC}}[Y_l], \mathbb{E}^{\text{MLMC}}[QoI_M], \cdots$
        estimate convergence
        $it = it + 1$
```

- Object transference:
  First of all, an experiment has been designed in order to compute the overhead due to sharing a single object along all the worker nodes. In the final application, a lot of concurrent reads on the same model will be done. In this first step, we quantify the capability of the frameworks to make available the model in all the worker nodes in an efficient way.

- First distributed version:
  Afterwards, the first distributed version is encoded. In this step, we demonstrate that the application have enough parallelism to be efficiently executed in a distributed environment with a high amount of resources.

- Optimization of the distributed version:
  In the next steps, the application is profiled in order to detect the problems that limit its performance. At each iteration, one or more problems are detected and solved in order to increase the efficiency of the execution.
  It is at this stage where we explain the procedure followed to detect each one of the improvable behaviors. This is the methodology that should be followed in the future to correctly analyze the application. We give some guidelines to understand how the profiling has been done. Since more implementations will be done, this profiling methodology should be used in order to reach the best possible performance.
  Basically, we analyze the Paraver traces to detect the points in which the resources are idle, analyze the algorithm to detect the reason why this is happening and propose solutions to minimize this effect. Although at this stage all the detected problems have been solved, this work sould be done all along the project with each one of the new implementations added to the framework.

3 Description of PyCOMPSs' experiments

Here we will describe the results of the experiments performed with PyCOMPSs, including screenshots with the Paraver tracefiles obtained with Extrae. Most of the Paraver views shown in the figures are timelines that show the execution of the tasks on the different processors. Each line represent a processor and different colours represent invocations to different tasks types.

All the new implementation and small bug fixes done during the process can be found on the *ExaQUte*'s branch of the COMPSs official github repository [8].

3.1 Platform description

The results presented in this section have been obtained using the MareNostrum IV Supercomputer [9] located at the Barcelona Supercomputing Center (BSC). Its current peak performance is 11.15 Petaflops, ten times more than its previous version, MareNostrum III. The supercomputer is composed by 3,456 nodes, each of them with two Intel®Xeon Platinum 8160 (24 cores at 2,1 GHz each). It has 384.75 TB of main memory, 100Gb Intel®Omni-Path Full-Fat Tree Interconnection, and 14 PB of shared disk storage managed by the Global Parallel File System.

3.2 Concurrent distributed model loading

In this case, a 90 MB model with 1,006,972 elements has been used as input. A first totally sequential execution was done, performing several writes in sequential, each of them requiring a deserialization. When running 10 deserializations in sequential, each one of them takes 7.43 seconds in average with a standard deviation of 0.25.

Next, we have launched a parallel run with 48,000 tasks that build again the object in memory with 100 worker nodes, with a total of 4,800 cores. The total elapsed time has been 127.97 seconds. When inspecting the tasks, we have stated that the time spent inside the user code is 9.87 seconds in average with an standard deviation of 0.24 for the deserialization. When considering only the user's sequential code, there is already a small performance lose. It has to be noted here that there are much more accesses to memory in the parallel case, which will imply a lower memory bandwidth available and more cache misses. In the first test, the program was running alone in a whole node. In the second case, there are 48 concurrent processes running in each one of the nodes.

In addition, each one of the cores is responsible to execute 10 tasks in the ideal case where there is no unbalance at all. The real mean time spent by each one of these cores is 12.97 seconds. There are still 3 seconds of overhead that have not been explained. Figure 1 shows the histogram graph of task duration. In the horizontal axis, there is the time duration of a task. In the vertical axis, the core in which that task has been executed. There is a batch of tasks that last on the order of 4 seconds more than the others. These are the first task executions per node, where all the Python modules are imported and initialized. Nevertheless, this work is done just once. This is why afterwards all the tasks are faster. A red line has been drawn at 12 seconds.

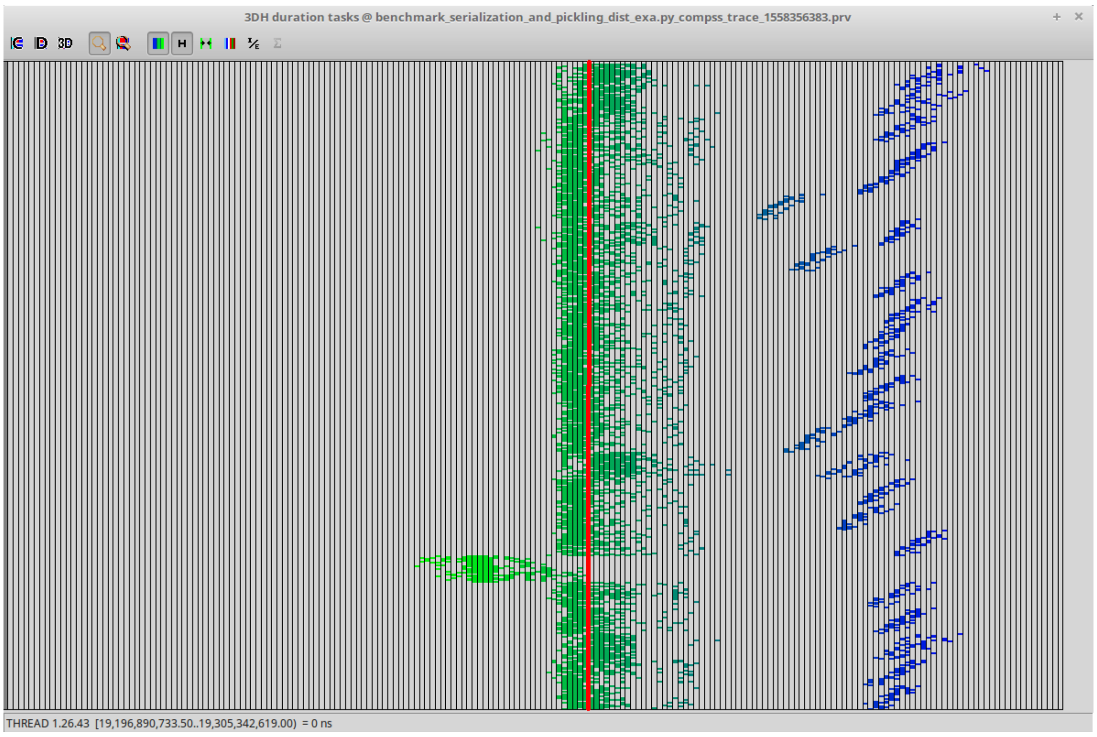

3DH duration tasks @ benchmark_serialization_and_pickling_dist_exa.py_compss_trace_1558356383.prv
THREAD 1.26.43 [19,196,890,733.50..19,305,342,619.00) = 0 ns

Figure 1: Histogram of task duration

Indeed, this is the mean task duration once the longer tasks are discarded. We can so consider that the difference between 9.87 and 12 is due to the transference and task initialization time. Indeed, this can be seen in the execution trace shown in Figure 2. The first dark red segment corresponds to the imports due to the initialization in each node. The light red corresponds to transference time and task creation. Finally, the white color corresponds to the actual user code execution time.

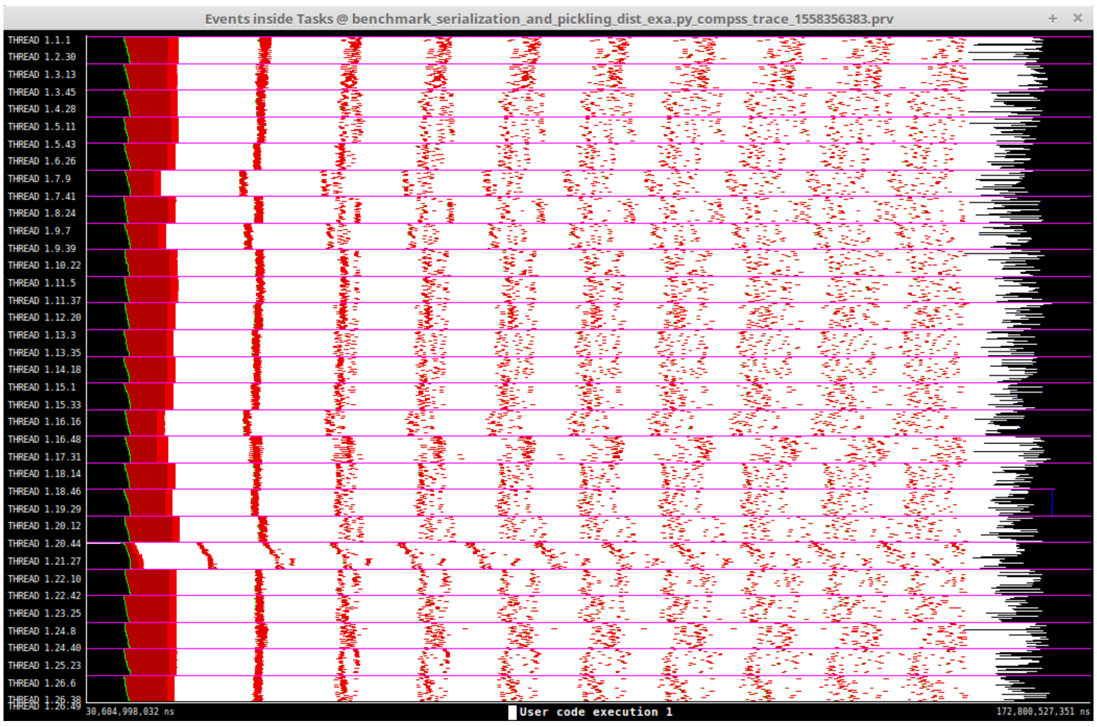


Figure 2: Events inside task view of the serialization benchmark execution with 25 worker nodes

We consider the numbers obtained to be more than reasonable taking into account the benefit of running in parallel.

3.3 First distributed version

Once verified that PyCOMPSs was capable to load the model concurrently with a more than acceptable performance, the first Montecarlo and Multilevel Montecarlo versions have been coded and analyzed in order to guide the future developments. The task-graph corresponding to the first distributed Multilevel Montecarlo is shown on Figure 3. In this graph, nodes represent tasks' invocations and edges represent actual data-dependences between the tasks. These dependences are detected by the PyCOMPSs runtime at execution time. First of all, there is a first batch of tasks corresponding to the Kratos simulations. Afte each of these tasks, there is a task responsible to extract the useful information from the simulation results. Next, another task adds the current result to the convergence criteria. Finally, once all the partial results has been added, a task checks if new executions should be launched. More precisely, in this example two iterations are performed. In addition, each one of the tasks updating the convergence criterion is responsible to add a given result into a certain level. This is why we can see as many dependency pipelines as levels considered.

In the first version all the accumulations at the end of the algorithm were done one by one following a sequential schema. This can be shown in the chains of 3 different tasks

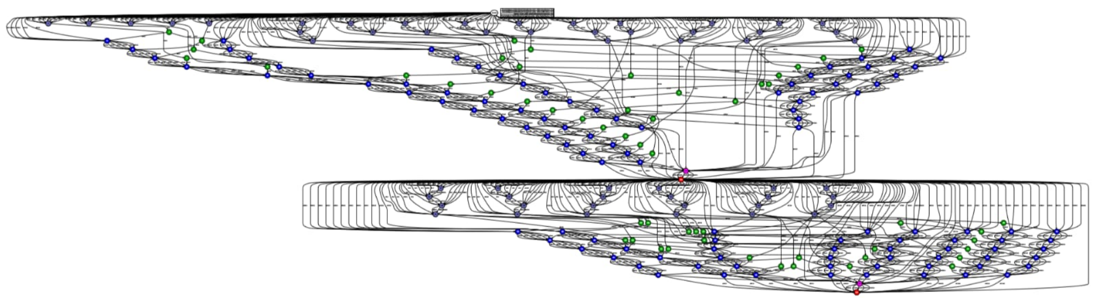


Figure 3: Initial Multilevel Montecarlo dependency graph

that are created in the dependency graph below the simulations. Once demonstrated that both the Multilevel and standard Montecarlo algorithm were working in a distributed environment, we focused in the single level algorithm. The strategy followed was to reduce the complexity of the problem with the objective of isolating the potential problems in the system. Indeed, we have been launching the Multilevel code considering just one level. This way, even if we are considering the most simple case, we are already running the Multilevel code. This makes smaller the change from single to several levels of the algorithm and allows an easier debug phase of the code.

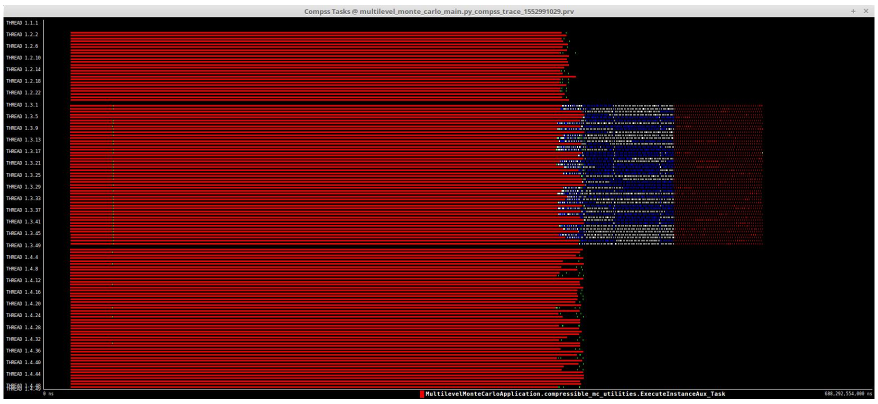
Regarding the approach followed, an important aspect to take into account with regard the previous subsections and the results in which they are based [10], is that the first four moments considered to compute the convergence can be expressed as a combination of the following parameters:

- $S_1 = \sum_i QoI_i$
- $S_2 = \sum_i QoI_i^2$
- $S_3 = \sum_i QoI_i^3$
- $S_4 = \sum_i QoI_i^4$

This is indeed an embarrassingly parallel operation depending on the simulation results that can be performed in whichever order [4].

Figure 4 helps understand how this fact affects to the final performance. In this case, a single iteration involving 2,000 simulations is executed in 2 worker nodes (48 cores each) and 24 worker cores in the master node is shown. Indeed, since all the values are added one after the other, there is an almost sequential portion of code executed once all the simulations (represented in red in the image) have been computed.

At this point, it is important to emphasize that all the profiling executions are done using a single CPU for each of the simulations. This is done this way since we focus on maximizing the concurrency level, and in this way we can detect the maximum amount of issues when using a given amount of resources. But we can opt, for instance, for launching an execution with 1,000 simulations, using 16 worker nodes and assigning 4 cores to each one of the simulations, and the obtained execution trace is the one shown in Figure 5. Even if the sequential part remains the same, it can start as soon as the first simulations finish. This is possible thanks to the dynamic scheduling performed by the PyCOMPSs runtime.


(a) Complete initial Multilevel Montecarlo execution trace

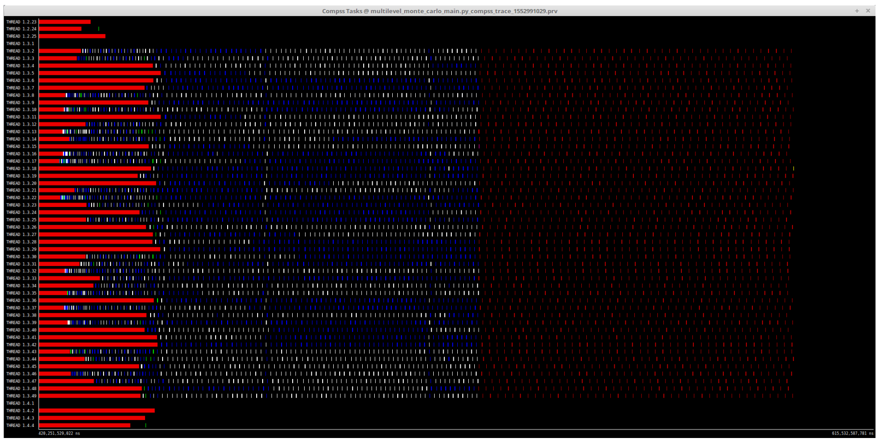

(b) Initial Multilevel Montecarlo execution trace sequential portion

Figure 4: Initial Multilevel Montecarlo execution trace

Since it has been demonstrated that keeping one core per simulation is the best way to discover the Workflow/PyCOMPSs limitations, we have kept this strategy in the following subsections. Indeed, it has revealed as a really successful strategy that has enabled us to increase the scalability while using a relatively small amount of resources.

3.4 Distributed version with grouped convergence extraction

The first aspect that has been improved is the treatment of the simulations' results after the simulation, processing in each task groups of data with size greater than one. This way, a single task can extract the results of several simulations and accumulate them into the convergence criteria. Considering that the overhead of creating and scheduling a task is at least constant, this mechanism highly reduces the amount of overhead. This reduces the overhead from a time proportional to the number of simulations to the number of simulations divided by the amount of extractions done in a single task. For example, with 100 extractions per task and 20,000 simulations, this overhead is only added 200 times instead of 20,000.

Figure 6 shows the execution trace of a single iteration of a Montecarlo algorithm with 16,000 simulations and 16 worker nodes (768 cores). Although difficult to see in the current view, is not only that there are less tasks that are serialized at the end of the execution to the reduction performed, but also that the COMPSs runtime is able to

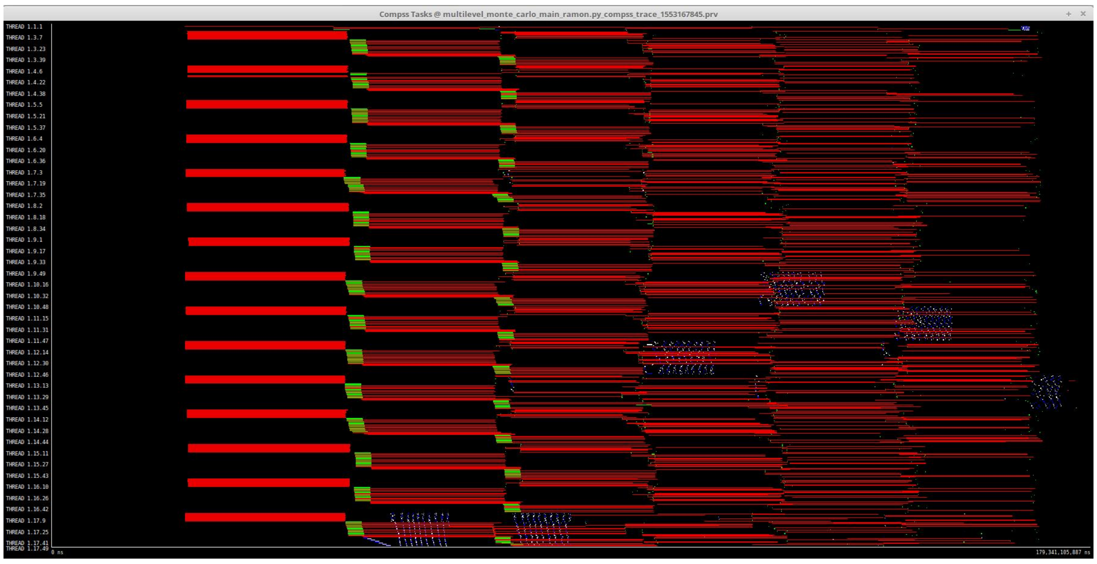


Figure 5: Initial Multilevel Montecarlo execution trace with OpenMP

interleave them between the simulations, almost eliminating the sequential part at the end.

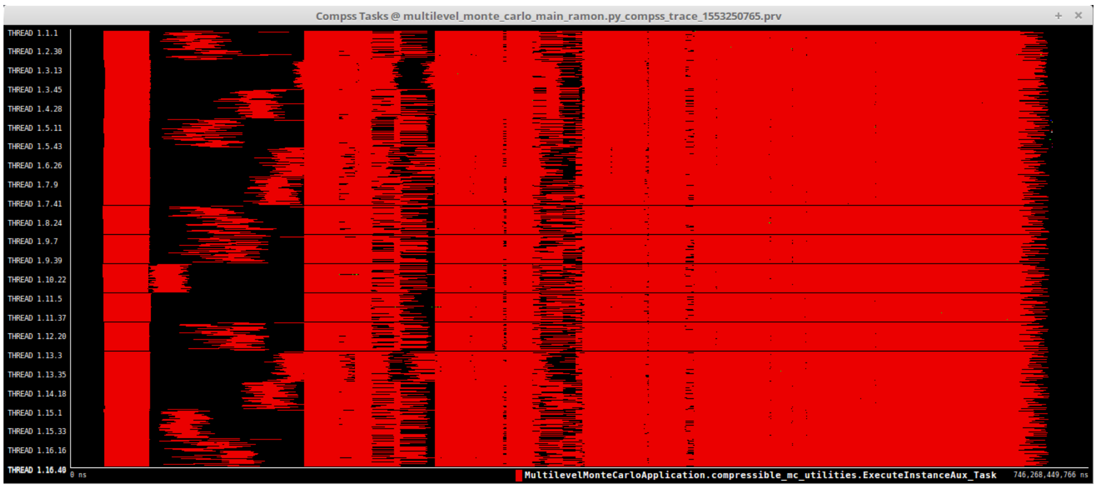


Figure 6: Improved Montecarlo execution trace

For an execution with 8 times more simulations, this tail at the end of the execution to compute the convergence criterias has disappeared (it is partially overlapped with the simulations' executions). Nevertheless, solving this problem has raised another one: it seems that the scheduler struggles at the beginning of the execution and has minor problems afterwards (black holes after the first iteration of simulation tasks).

Analysing the runtime behaviour with more detail, we observed that the issue appears while the algorithm is creating tasks at the beginning of the execution, due to the large amount of tasks generated in a short period of time (16,070). While this is not a problem with larger task granularities or with smaller number of nodes, when the runtime is stressed with large number of tasks and large number of cores to schedule tasks this problem becomes more important. Figure 7 shows a view of the COMPSs runtime threads. Different colors represent different states of the runtime. More specifically, when the second thread is represented in purple, the runtime is creating tasks. We can see that this period corresponds with the initial part of the execution where the issue is present.

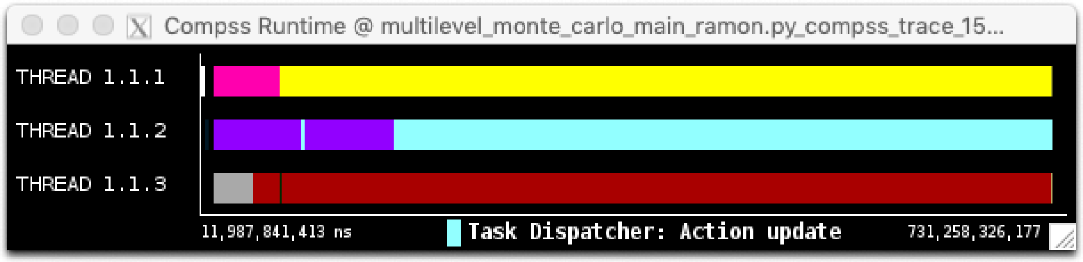


Figure 7: COMPSs runtime view

3.5 Distributed version with the scheduling improvements

3.5.1 Problem diagnosis

On the first hand, the current scheduler has been studied in order to detect where it could be improved. COMPSs has several scheduling policies available. The focus has been put in the `ready` scheduler, which only takes into account the tasks that have already been freed from dependencies, ant therefore are ready to be executed at that moment. Figure 8 shows the code corresponding to the starting point at this stage. This function is called every time that a tasks frees an execution slot.

There are three main things that could be improved:

- Keep track of idling worker nodes
  The scheduler evaluates the possibility of executing every free task in all the resources, verifying the available slots for each one of them. Hence, the complexity increases linearly with the amount of available resources. In addition, the scheduler tries to execute all the available tasks even when all the available resources are busy.

- Parallelize the scheduling process
  The scheduling process is done in a single thread. Hence, all the tasks are treated sequentially.

- Avoid computing the ordered list with the actions to be performed in each resource each time that a slot is freed
  The scheduler maintains an ordered list of actions to be performed, which is updated very often. A score of the suitability to execute each task in each resource is evaluated. Assuming that the scores associated to each pair `resource - task` do not vary all along the execution, this ordered list could be maintained between calls to the scheduler. This can avoid reordering the list each time as long as the amount of available tasks does not change. For big executions, this list can contain tens of thousands of tasks. Thus, this fact cannot be overlooked.

```java
private <T extends WorkerResourceDescription> void tryToLaunchFreeActions(
    List<AllocatableAction> dataFreeActions, List<AllocatableAction>
    resourceFreeActions,
    List<AllocatableAction> blockedCandidates, ResourceScheduler<T> resource) {

   // Try to launch all the data free actions and the resource free actions
   PriorityQueue<ObjectValue<AllocatableAction>> executableActions = new
    PriorityQueue<>();
   for (AllocatableAction freeAction : dataFreeActions) {
       Score actionScore = generateActionScore(freeAction);
       Score fullScore = freeAction.schedulingScore(resource, actionScore);
       ObjectValue<AllocatableAction> obj = new ObjectValue<>(freeAction,fullScore);
       executableActions.add(obj);
   }

   for (AllocatableAction freeAction : resourceFreeActions) {
       Score actionScore = generateActionScore(freeAction);
       Score fullScore = freeAction.schedulingScore(resource, actionScore);
       ObjectValue<AllocatableAction> obj = new ObjectValue<>(freeAction,fullScore);
       if (!executableActions.contains(obj)) {
           executableActions.add(obj);
       }
   }

   while (!executableActions.isEmpty()) {
       ObjectValue<AllocatableAction> obj = executableActions.poll();
       AllocatableAction freeAction = obj.getObject();
       Score actionScore = obj.getScore();

       // LOGGER.debug("Trying to launch action " + freeAction);
       try {
           scheduleAction(freeAction, actionScore);
           tryToLaunch(freeAction);
       } catch (BlockedActionException e) {
            blockedCandidates.add(freeAction);
        }
   }
}

```

Figure 8: Original COMPSs ready scheduler

3.5.2 Implementation proposed

Considering the problems detected in the previous section, a solution has been proposed that both includes a multi-threaded treatment that keeps the ordered lists and a control of the available resources in order to stop the scheduling operations once all the resources are busy.

Figure 9 shows how the main `while` has been changed and Figure 10 shows the chosen mechanism to asynchronously update the scheduler structures. This should be enough to briefly understand the basis on which the full implementation is based.

The implemented solution is based in three main ideas:

- Keep track of the available workers in a HashMap
  This data structure has been chosen in order to guarantee a constant complexity access to this information which highly reduces the original overhead.

- Keep a different list for each one of the resources
  This fact allows to store a list with the priority order of the available tasks for each one of the resources considering the chosen policy. The current implemented policies are FIFO, LIFO, data locality and load balancing (in case of equal data locality, tasks are sent to the workers with less workload).

- Spawn threads to update the scheduling structures
  Since each one of the resources has its own priority queue,its update can be done

```java

Future<?> lastToken = this.resourceTokens.get(resource);
if (lastToken != null) {
    try {
        lastToken.get();
    } catch (InterruptedException | ExecutionException e) {
        e.printStackTrace();
        LOGGER.fatal("Unexpected thread interruption");
        ErrorManager.fatal("Unexpected thread interruption");
    }
}
this.resourceTokens.put(resource, null);

Iterator<ObjectValue<AllocatableAction>> executableActionsIterator =
    this.unassignedReadyActions.get(resource).iterator();
HashSet<ObjectValue<AllocatableAction>> objectValueToErase =
    new HashSet<ObjectValue<AllocatableAction>>();
while (executableActionsIterator.hasNext() && !this.availableWorkers.isEmpty()) {
    ObjectValue<AllocatableAction> obj = executableActionsIterator.next();
    AllocatableAction freeAction = obj.getObject();
    try {
        if (Tracer.isActivated()) {
            Tracer.emitEvent(Tracer.Event.TRY_TO_SCHEDULE.getId(),
                Tracer.Event.TRY_TO_SCHEDULE.getType());
        }
        freeAction.tryToSchedule(obj.getScore(), this.availableWorkers);
        if (Tracer.isActivated()) {
            Tracer.emitEvent(Tracer.EVENT_END,
                Tracer.Event.TRY_TO_SCHEDULE.getType());
        }
        ResourceScheduler<? extends WorkerResourceDescription> assignedResource =
            freeAction.getAssignedResource();
        tryToLaunch(freeAction);
        if (!assignedResource.canRunSomething()) {
            this.availableWorkers.remove(assignedResource);
        }
        objectValueToErase.add(obj);
    } catch (BlockedActionException e) {
        ...
    } catch (UnassignedActionException e) {
        ...
    }
}

```

Figure 9: Asynchronous scheduling structures update

```java
private Runnable createAddRunnable(
final Map.Entry<ResourceScheduler<?>,
    TreeSet<ObjectValue<AllocatableAction>>> currentEntry, final AllocatableAction
    action, final Future<?> token) {
    Runnable addRunnable = new Runnable() {
        public void run() {
            if (token != null) {
                try {
                    token.get();
                } catch (InterruptedException | ExecutionException e) {
                    e.printStackTrace();
                    LOGGER.fatal("Unexpected thread interruption");
                    ErrorManager.fatal("Unexpected thread interruption");
                }
            }
            addActionToResource(currentEntry, action);
        }
    };
    return addRunnable;
}

```

Figure 10: Asynchronous scheduling structures update

asynchronously and just wait for the result in case a certain resource frees a slot to perform some computations.

In order to achieve the desired behavior without making the code too complicated, a strategy based on tokens has been followed. This way, each time that a list must be modified, the thread in charge of this modification waits for the token corresponding to the last modification for each one of the lists to be generated. In addition, each time that a modification is added to the thread scheduler, the token is updated so the next time that a new thread is spawned it waits for the previous modifying thread to finish. Afterwards, when the queue needs to be accessed, the main thread waits for the token corresponding to the last modification to be generated. This way, it is guaranteed that the queue contains all the modifications needed until the moment and that they have been done in the same order that they were requested.

These modifications are basically erasing tasks that are already running on another resource and adding tasks that has been freed from the last resource scheduling. This way, the modification of the scheduling structures is removed from the scheduling critical path.

3.5.3 Obtained results

With these improvements, we have been able to launch the execution shown on Figure 11. In this case, we have 20 worker nodes (960 cores) and 49,380 simulations. For this case, the amount of simulations is kept almost constant while the concurrence degree (and the amount of resources to be managed) has been increased. Although the issue at the beginning of the executions remain (the black hole after the first set of simulations), the general behavior is much better than the obtained with the previous version of the scheduler.

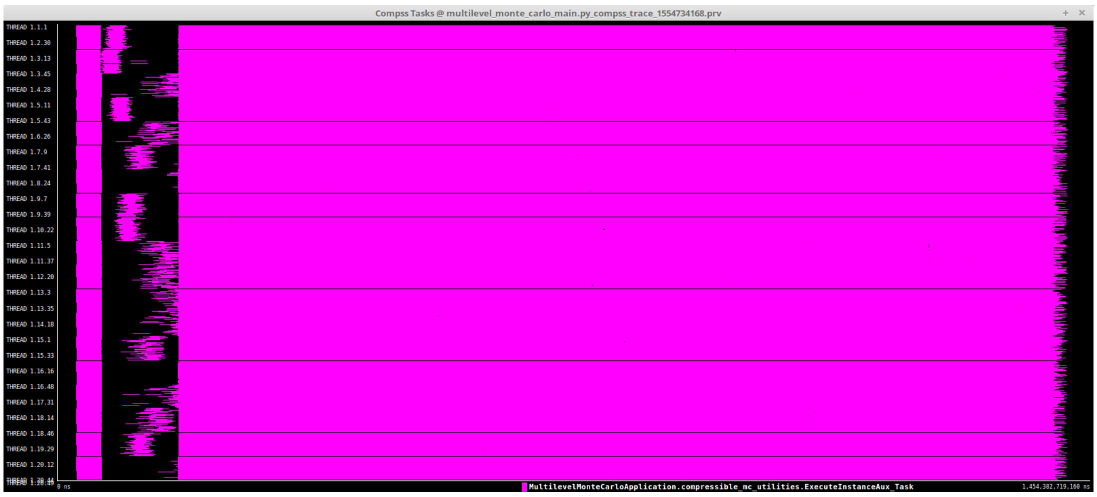


Figure 11: Improved Montecarlo execution trace with the first scheduling improvements

3.6 Distributed version with dynamic convergence checking

Since the obtained results at this stage were already good enough, we have decided to implement a code improvement while still looking after the scheduling issue at the beginning of the execution.

In fact, each one of the Montecarlo iterations was done in an efficient way. Nevertheless, the convergence was checked once all the simulations of a single batch were finished. The computation of the stop criterion must wait until all the simulations of a whole batch have finished in order to avoid having a biased result. Launching the batches one by one can lead to resource wasting in case another iteration is needed since no new simulations are started until the convergence has been computed. This is not the case for a single level execution because all the simulations take approximately the same time. Nevertheless, in the Multilevel scenario we could end up in a case where the most long and demanding simulation is launched at the end with some resources and all the other blocked waiting for this simulation to finish. Hence, we have started working in the following actions in parallel:

- Adapt the Montecarlo code in order to launch several batches of executions
  This has been done in a way that the convergence is checked when the batches in an incremental way once the different simulations in a batch starts to finish. Nevertheless, the resources are filled with the executions of the following batches in order to increase the resource efficiency in case the convergence is not reached.

- Improve the scheduling behavior
  Since the bad behavior was found at the beginning of the execution, we have checked all the task registration process and we have found some parts of the runtime code that could be improved. Despite the modifications shown in the previous subsection, in this case we have not changed the global architecture of the scheduler. Instead, we have done some minor modifications that have apparently solved the problem.

Once the previous improvements have been implemented, they have been tested in a production environment. More precisely, a first execution with 25 worker nodes and 130,000 simulations. The execution trace is shown in Figure 12. Considering the starting point, this could be considered as a very good execution.

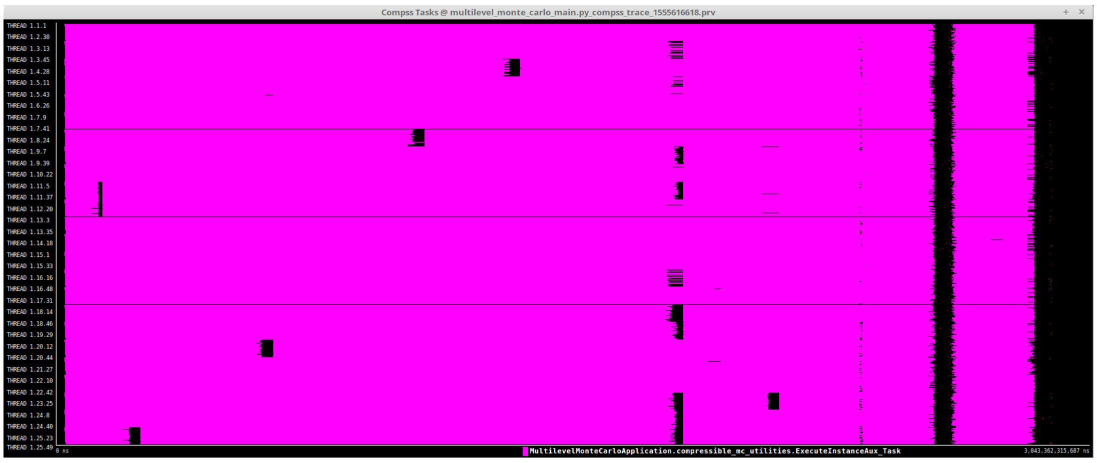


Figure 12: Improved Montecarlo execution trace with all the scheduling improvements with 130,000 simulations and 25 worker nodes

Afterwards, the same execution with more available resources has been done. More precisely, Figure 13 shows an execution with 50 worker nodes and Figure 14 shows an execution with 100 worker nodes. It seems clear that the scheduling system struggles with this amount of resources and available tasks. We can then fix the limit of concurrency at this stage of the development to 25 worker nodes, that is 1200 cores, for simulations lasting about 25 seconds.

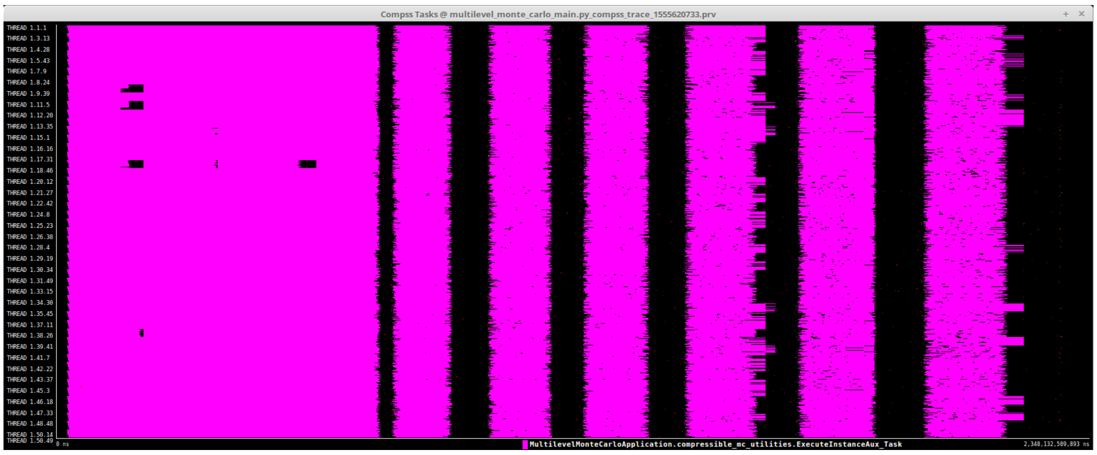

Compss Tasks @ multilevel_monte_carlo_main.py_compss_trace_1555620733.prv
MultilevelMonteCarloApplication.compressible_mc_utilities.ExecuteInstanceAux_Task

Figure 13: Improved Montecarlo execution trace with all the scheduling improvements with 130000 simulations and 50 worker nodes

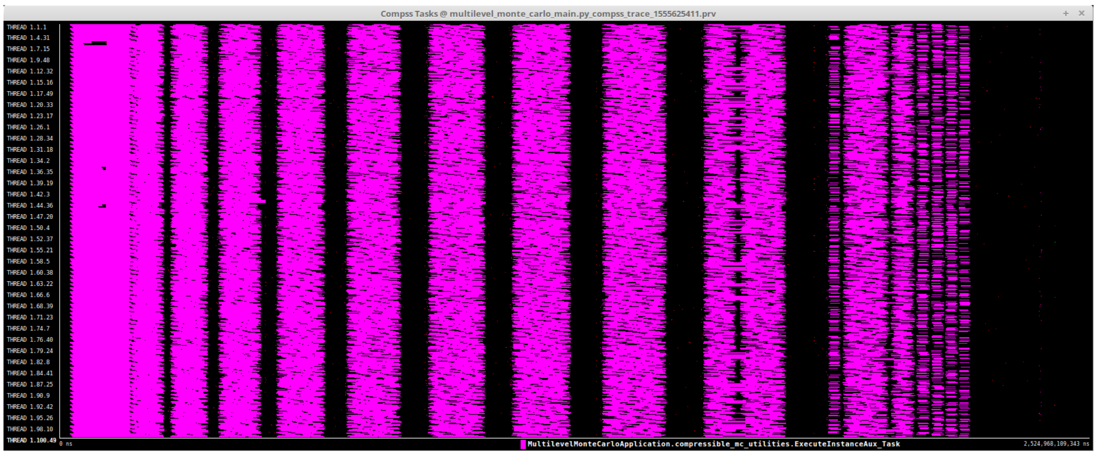

Compss Tasks @ multilevel_monte_carlo_main.py_compss_trace_1555625411.prv
MultilevelMonteCarloApplication.compressible_mc_utilities.ExecuteInstanceAux_Task

Figure 14: Improved Montecarlo execution trace with all the scheduling improvements with 130000 simulations and 100 worker nodes

It is important to realize that executions of only 25 seconds are really fine grain tasks when taking into account the amount of concurrency level and the amount of spawned tasks.

4 Description of experiments performed with HyperLoom

(it4i)

Describe here the results of the experiments performed with HyperLoom.

4.1 Platform description

TODO: Describe computing platform (Salomon, others?)

The Salomon cluster consists of 1009 compute nodes, totaling 24192 compute cores with 129 TB RAM and giving over 2 PFLOP/s theoretical peak performance. Each node is a powerful x86-64 computer, equipped with 24 cores, and at least 128 GB RAM. Nodes are interconnected through a 7D Enhanced hypercube InfiniBand network and are equipped with Intel Xeon E5-2680v3 processors. The Salomon cluster consists of 576 nodes without accelerators, and 432 nodes equipped with Intel Xeon Phi MIC accelerators. The cluster runs with a CentOS Linux operating system, which is compatible with the RedHat Linux family.

4.2 Experiments description

Due to technical problems, the results of performance at Hyperloom are not available at the time of submitting this deliverable D4.2. An addenda to that deliverable, including the results of the performace of the partners' software tools at Hyperloom, will be sent to the Commision at the shortest possible notice.

5 Performance recommendations

The aim of this section is to summarize the methodology that should be followed all along the project in order to maximize the obtained performance.

First of all, it is really important to detect tasks that are executed in a sequential way in order to find alternatives that increase the parallelism degree of the applications. Although this is not possible in all the cases, the most part of times it is indeed possible by doing a refactor in the code.

Next, the task execution time should be profiled in order to verify that it fits the both the programming model needs regarding the magnitude of the overheads and the amount of the requested resources.

Finally, all the modifications in the code should be tested executing as big executions as possible with one core per task in order to check the programming model capabilities to adapt to this new functionalities. This mechanism has revealed as the fastest way to check the scalability of a given program.

References

[1] Michael B Giles. Multilevel Monte Carlo path simulation. *Operations Research*, 56 (3):607–617, 2008.

[2] Michael B Giles. Multilevel Monte Carlo methods. *Acta Numerica*, 24:259–328, 2015.

[3] K Andrew Cliffe, Mike B Giles, Robert Scheichl, and Aretha L Teckentrup. Multilevel monte carlo methods and applications to elliptic pdes with random coefficients. *Computing and Visualization in Science*, 14(1):3, 2011.

[4] M. Pisaroni, S. Krumscheid, and F. Nobile. Quantifying uncertain system outputs via the multilevel Monte Carlo method – Part I: Central moment estimation. MATHICSE Technical Report 23.2017, École Polytechnique Fédérale de Lausanne, 2017.

[5] Ramon Amela, Cristian Ramon-Cortes, Jorge Ejarque, Javier Conejero, and Rosa M Badia. Enabling python to execute efficiently in heterogeneous distributed infras-

tructures with pycompss. In *Proceedings of the 7th Workshop on Python for High-Performance and Scientific Computing*, page 1. ACM, 2017.

[6] Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Álvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M Badia. Servicess: An interoperable programming framework for the cloud. *Journal of grid computing*, 12(1):67–91, 2014.

[7] Vojtěch Cima, Stanislav Böhm, Jan Martinovič, Jiří Dvorskỳ, Kateřina Janurová, Tom Vander Aa, Thomas J Ashby, and Vladimir Chupakhin. Hyperloom: A platform for defining and executing scientific pipelines in distributed environments. In *Proceedings of the 9th Workshop and 7th Workshop on Parallel Programming and RunTime Management Techniques for Manycore Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 1–6. ACM, 2018.

[8] BSC. Compss' exaqute branch. `https://github.com/bsc-wdc/compss/tree/exaQUte`, 2019.

[9] Barcelona Supercomputing Center (BSC). MareNostrum IV Technical Information, 2018. URL `https://www.bsc.es/marenostrum/marenostrum/technical-information`.

[10] M. Pisaroni, F. Nobile, and P. Leyland. A continuation multi level monte carlo (c-mlmc) method for uncertainty quantification in compressible inviscid aerodynamics. *Computer Methods in Applied Mechanics and Engineering*, 326:20 – 50, 2017.