RUPER-LB: Load balancing embarrasingly parallel applications in unpredictable cloud environments

V. Giménez-Alventosa
Instituto de Instrumentacíon para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València, Camí de Vera s/n, 46022, València, Spain
vicent.gimenez@i3m.upv.es

Germán Moltó
Instituto de Instrumentacíon para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València, Camí de Vera s/n, 46022, València, Spain
gmolto@dsic.upv.es

J. Damián Segrelles
Instituto de Instrumentacíon para Imagen Molecular (I3M), Centro mixto CSIC - Universitat Politècnica de València, Camí de Vera s/n, 46022, València, Spain
dquilis@dsic.upv.es

Abstract

The suitability of cloud computing has been studied by several authors to run scientific applications. However, the unpredictable performance fluctuations in these environments hinders the migration of scientific applications to cloud providers. To mitigate these effects, this work presents RUPER-LB, a load balancer for loosely-coupled iterative parallel applications that runs on infrastructures with disparate computing capabilities. The results obtained with a real world simulation software, show the suitability of RUPER-LB to adapt this kind of applications to execution environments with variable performance and highlight the convenience of its adoption.

2012 ACM Subject Classification Software and its engineering → Software libraries and repositories

Keywords and phrases Load balancer, Parallel computing, MPI, Multithreading, Cloud computing

Acknowledgements This study was supported by the program “Ayudas para la contratación de personal investigador en formación de carácter predoctoral, programa VALi+d” under grant number ACIF/2018/148 from the Conselleria d’Educació of the Generalitat Valenciana and the “Fondo Social Europeo” (FSE). The authors would like to thank the Spanish "Ministerio de Economía, Industria y Competitividad" for the project “BigCLOE” with reference number TIN2016-79951-R.

1 Introduction

Since the emerging of cloud computing, several authors have studied its suitability to run scientific applications. The motivation of these studies are the inherent benefits offered by cloud providers. First, cloud computing allows to scale the underlying infrastructure to fit the user needs, eliminating the effects of both under and over provisioning resources. Then, the pay-per-use model provides a cost-effective usage of resources, allowing the users to deploy the required infrastructure and pay for it only during the execution time. Finally, virtualisation provides increased flexibility, since Virtual Machines (VM) can be configured with all the dependencies required by the applications.

However, clouds are not widely used for all kind of scientific applications because they also exhibit some drawbacks. First, cloud providers use a multi-tenant approach to optimise resource usage. This means that the physical processors, disk, memory, etc. where the VM is running can be shared with VMs from another user. This hardware sharing causes a variability on the CPU performance, memory bandwidth, network communications and disk I/O speed, a problem commonly known as *noisy neighbour* [4]. In addition, cloud providers typically offer instance types featuring certain characteristics, such as amount of RAM, number of virtual equivalent CPUs (vCPUs), storage, etc., but the user cannot select the specific hardware characteristics. These vCPUs are not physical cores, but a CPU equivalent unit. Unfortunately, the performance of these vCPUs are highly dependent on the underlying hardware, which produce high performance differences between instances of the same type. All these effects have been widely studied in the bibliography [10, 12, 15, 16] and even methodologies are provided to correctly measure this variability [2].

As a response to the demand of instances with predictable capabilities, some providers such as Amazon Web Services (AWS) offer the option to launch single-tenant instances [1] at the expense of additional costs. However, depending on the application this fee may not be worth. Also, these single-tenant instances ensure that the physical hardware will be used only by VMs from the account owner. However, this does not preclude from suffering noisy neighbour effects among the user's own instances.

Turning to parallel scientific applications, their execution time is usually determined by the slowest process, so an unbalanced situation will delay the entire application. These facts highlight the need for advanced load balancing techniques to adapt scientific applications to the variable performance found on heterogeneous environments. This effort has been done for High Performance Computing (HPC) applications where authors have studied the suitability of cloud computing environments [11] [8] [7]. These studies agree that tightly coupled applications are less suitable for cloud computing, which is reasonable considering the fluctuations reported on network bandwidth. To mitigate the unbalance problem, several load balancing algorithms adapted to cloud environments have been proposed [14] [9]. In addition, we can find studies of techniques for efficient VM deployment [17] [5]. However, this unpredictable variability of the computational capabilities does not only affect tightly coupled processes, but also loosely coupled ones.

Loosely coupled applications neither require a continuous communication nor synchronisation points, like HPC applications. For instance, most of the load balancing algorithms designed for HPC involve an unnecessary overhead for these applications due the amount of synchronisation points and communications involved. On the other hand, classic load balancing algorithms used on heterogeneous systems, which rely on previous knowledge of the underlying performance [3], are not suitable for these environments due the unpredictable performance fluctuations.

To address these problems, we present RUPER-LB (Runtime Unpredictable Performance Load Balancer) a load balancing algorithm for loosely coupled applications running on environments with unpredictable performance variability with both multi-process and multi-thread balance. RUPER-LB is provided as open-source code under the GPLv3 license and can be download from **Blinded**. For assessment purposes, RUPER-LB was used to balance PenRed [6] simulations, which is a radiation transport simulation framework focused on medical applications with MPI and multithreading built-in parallelism.

2 Materials and Methods

RUPER-LB focuses on parallel iterative applications such as Monte-Carlo simulations, iterative solvers or multi-parametric analysis. These applications must comply with the following restrictions:

Firstly, the application must be split in tasks. During the execution of these tasks, the application should not require any communication or synchronisation point among the executing threads or processes. Nevertheless, if communications are required, their overhead

Table 1 Worker (left) and task (right) object states.

Variable	Description
I_n	Assigned iterations
started	Flags the task start
finished	Flags the task end
I_d	Number of finished iterations
t_r	Last report timestamp
t_i	Task start timestamp
m	Velocity measures vector

Variable	Description
I_n	Number of iterations to do
w	Vector of *worker* objects
t_0	Task start timestamp
t_{pc}	Last checkpoint timestamp
Δt_{pc}	Time between checkpoints
started	Flags task start
finished	Flags task finish
t_{min}	Balance time threshold
ds_{max}	Maximum speed deviation

on the task performance should be negligible. If these assumptions are not accomplished, RUPER-LB can still be used but an HPC-like load balancing algorithm may achieve better results in terms of makespan.

Secondly, the application should measure its speed at runtime. Thus, RUPER-LB assumes that the application behaves like an iterative process, whose speed is measured in iterations per second. The number of iterations to process by each thread and process should be allowed to be changed at runtime. Notice that RUPER-LB neither requires an homogeneous computational cost for the iterations nor a previous balanced distribution among threads.

PenRed, the selected code to test the presented algorithm, satisfies these required assumptions. In this code, tasks correspond to each particle source defined by the user. Each generated primary particle and all its secondaries will be considered as a single history, which corresponds to one iteration. Finally the number of histories to simulate by each thread and process can be changed at runtime.

2.1 Multi-threading balance

Some multi-threading applications employ the involved threads in an unbalanced way. For example, assigning I/O operations or network communications to a specific thread. Also, the computational cost of the iterations that constitute the process could be heterogeneous, or some thread could use accelerated hardware like a GPGPU. Both situations will produce variable unbalances on thread speeds, measured in iterations per second. Also, it is not feasible in a Cloud to know which computational resources are being shared with other VMs running on the same physical hardware and, therefore, how their workload pattern will change during the execution. This fact could increase the unbalance produced by previous effects. Thus, we need to balance the workload between the threads of a single process dynamically. This section describes how this local load balancing is performed.

The workload distribution, i.e. the number of iterations assigned to each thread, is handled by two components implemented as classes in an object oriented programming (OOP) language. These are the *tasks* and the *workers*, which represent a single task and the threads executing the task respectively. Also, the execution could involve more than one task, each of them having its own workers. Figure 1 top shows the basic balance schema for single process executions, where each thread is assigned to a single worker of the active task. The basic states of both components are listed in table 1.

Basically, each worker reports periodically the number of completed iterations to the *task* object. This is done using the *report* method, whose code is shown in figure 2 left. In this

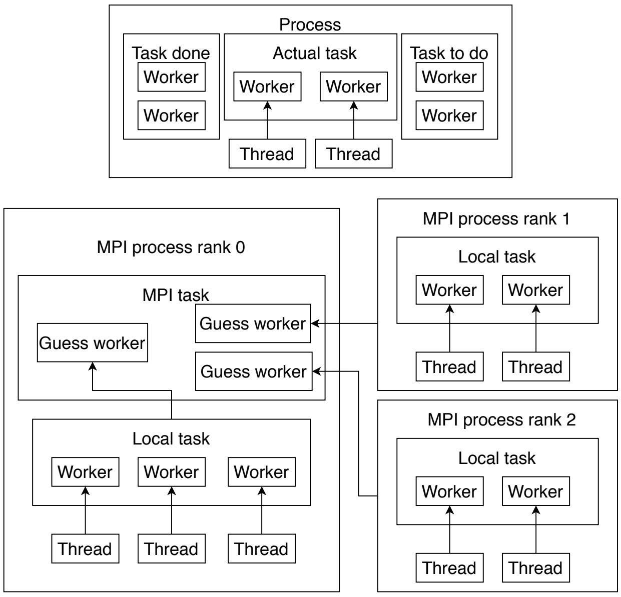

Figure 1 Top: Thread balance system schema for a process with 3 tasks. Bottom: MPI balancing schema for 3 MPI processes and 1 task.

code, and the following ones, the use of locks and the sanity checks on variable values have been omitted for simplicity. The *report* method takes as argument three values: a measure of the number of completed iterations, the measure timestamp and the worker index that performed these iterations. Regarding the execution, first, we use two auxiliary *worker*'s methods, *working* and *elapsed*. The first one returns *true* if the *worker* is still executing the task, otherwise returns *false*, and the second one returns the elapsed time since the last report. Following, the *worker* method *addMeasure* (Figure 2 right) is used to compute and store its speed measured since the last report (t_r). In addition, that method returns the quotient s/s_l, where s is the new speed to register and s_l is the registered speed in the previous report, that is, the speed deviation from the previous report. This information will be used to calculate, in the *report* method, the suggested time interval until next report (Δt).

Each thread will compute its own reports independently, i.e. the threads do not require to synchronise to perform the report at the same time. The same goes for the *checkpoint* method, whose pseudocode is shown in figure 3 left. This *task* method, redistributes the workload among its *workers* according to the information stored by reports. First of all, the algorithm calculates three values: the total simulation speed (s_t), the total reported iterations done (I_t) and the predicted iterations done (I_{pred}). To obtain I_{pred}, we use the auxiliary *worker* method *predDone*, which returns the predicted iterations done by the worker assuming no changes on its speed since last report. Notice that the calculation of task speed excludes the already finished workers. Then, we check if the required iterations have been done. If that happens, the assigned iterations of each worker will be set to its reported iterations done, i.e. force workers to finish the task. On the other hand, if there are still iterations to do, we evaluate a prediction of the remaining execution time (t_{res}) according to I_{pred} and s_t. Finally, if t_{res} is greater than the threshold (t_{min}), the iterations assigned to each active worker will be recalculated according to its speed factor.

At some point of the execution, the workers will consider that they have finished the task. At this point, workers will ask to finish to the *task* object, which will allow or refuse

```
$report(i, I_{done}, t)$
Input:
$i \rightarrow$ Worker index
$I_{done} \rightarrow$ Number of completed iterations
$t \rightarrow$ Report timestamp
Output:
$\Delta t \rightarrow$ Suggested time until next report

if $w_i.working()$ then
  $\Delta t \leftarrow w_i.elapsed(t)$
  $dev \leftarrow w_i.addMeasure(t, I_{done})$
  $dev \leftarrow ABS(dev - 1)$
  if $dev > ds_{max}$ then
    $\Delta t \leftarrow \Delta t \cdot max(1 - (dev - ds_{max}), 0.8)$
  else if $dev < 0.1 \cdot ds_{max}$ then
    $\Delta t \leftarrow \Delta t \cdot min(1 + (0.5 \cdot ds_{max} - dev), 1.2)$
  end if
  if $\Delta t > \Delta t_{pc}$ then
    $\Delta t \leftarrow \Delta t_{pc} \cdot 0.8$
  end if
else
  $\Delta t \leftarrow -1$
end if
```

```
$addMeasure(t, I_{done})$
Input:
$I_{done} \rightarrow$ Number of completed
        iterations
$t \rightarrow$ Measure timestamp
Output:
$dev \rightarrow$ Speed deviation

$\Delta t \leftarrow t - t_r$
$\Delta t_m \leftarrow t - t_i$
$\Delta I \leftarrow I_{done} - I_d$
$s_l \leftarrow speed()$
$s \leftarrow \Delta I / \Delta t$
$I_d \leftarrow I_{done}$
$t_r \leftarrow t$
$dev \leftarrow s/s_l$
$m \leftarrow (\Delta t_m, s)$
```

Figure 2 *Task report* method (left) and *Worker addMeasure* method (right).

the request to finish according to the *task* stored information. There are two reasons to deny this request. The first reason is that the *task* object has registered less iterations done by the worker than the ones assigned. In this case, a new report will be required. The second reason is that the estimated remaining execution time to complete the task is greater than t_{min}. This last case requires a new checkpoint to reassign the number of iterations for each worker. If neither of both conditions are accomplished, the worker can finish the task. Thus, the *worker* method *working* will return *false* hereinafter. Once all workers have finished, the task is considered as finished.

2.2 MPI balance

If MPI load balancing is enabled, this is handled at two levels, as shown in figure 1 bottom. First, locally to each MPI process, where the threads are balanced using the method described in the previous section. Then, the number of iterations to do is split between MPI processes. The rank 0 will handle the assignment of iterations for each process *task*, thus the I_n value is not constant on MPI. For that purpose, both objects *worker* and *task* are extended as follows. First, since the local thread reports are performed asynchronous, the iterations done and speed registered at local tasks are, in general, outdated. To counteract that, the MPI balance procedure registers the predicted iterations done, and not the reported ones. This procedure requires a new type of worker, which has been created as a derived object of the *worker* saw at section 2.1. That new worker object used for MPI balance has been named *guess worker*, which shares the same state as the base *worker* class (table 1). However, notice that *guess workers* do not represent a single thread, as the workers of section 2.1. Instead, a

```
$checkPoint()$
Input:
Output:
$t_{pc} \leftarrow actualTime()$
$s_t \leftarrow 0$
$I_t \leftarrow 0$
$I_{pred} \leftarrow 0$
for each $worker$ in $w$ do
  $I_t \leftarrow I_t + worker.I_d$
  if $worker.working()$ then
    $s_t \leftarrow s_t + worker.speed()$
    $I_{pred} \leftarrow I_{pred} + worker.predDone(t)$
  else
    $I_{pred} \leftarrow I_{pred} + worker.I_d$
  end if
end for
if $I_n <= I_t$ then
  for each $worker$ in $w$ do
    if $worker.working()$ then
      $worker.I_n \leftarrow worker.I_d$
    end if
  end for
else
  $I_{res} \leftarrow I_n - I_{pred}$
  $t_{res} \leftarrow I_{res}/s_t$
  if $t_{res} > t_{min}$ then
    for each $worker$ in $w$ do
      if $worker.working()$ then
        $s_{fact} \leftarrow worker.speed()/s_t$
        $worker.I_n \leftarrow worker.I_d +$
                  $s_{fact} \cdot (I_n - I_t)$
      end if
    end for
  end if
end if
```

```
$addMeasure(t, I_{done})$
Input:
$I_{done} \rightarrow$ Iterations completed
        prediction
$t \rightarrow$ Measure timestamp
Output:
$dev \rightarrow$ Speed deviation
if $speed() = 0$ then
  $dev \leftarrow worker :: addMeasure(t, I_n)$
else
  $\Delta t \leftarrow t - t_r$
  $\Delta t_m \leftarrow t - t_i$
  if $I_d > I_{done}$ then
    $\bar{s_1} \leftarrow I_d/(t_r - t_i)$
    $\bar{s_2} \leftarrow I_{done}/(t - t_i)$
    $dev \leftarrow \bar{s_2}/\bar{s_1}$
  else
    $\Delta I_e \leftarrow speed() \cdot \Delta t$
    $\Delta I_r \leftarrow I_{done} - I_d$
    $dev \leftarrow \Delta I_r/\Delta I_e$
  end if
  $s \leftarrow dev \cdot speed()$
  $t_r \leftarrow t$
  $m \leftarrow (\Delta t_m, s)$
end if
```

Figure 3 Method *checkPoint* for *task* object (left) and *addMeasure* for *guess worker* object (right).

guess worker registers the information of the whole task running on one of the MPI processes (figure 1). In addition, a *guess worker* object uses a different *addMeasure* method, whose pseudocode is shown in figure 3 (right). This *addMeasure* method corrects the last measured speed using the deviation between the reported and the expected prediction of iterations done at the time t. Notice that this method based on speed correction could fail if 0 iterations per second is reported. To handle this situation, the *addMeasure* method of the base *worker* object (figure 2) will be called.

On the other hand, to adapt *task* objects to handle MPI balance, we add the variables listed in table 2 to its state. As indicated in the following descriptions, the usage of the new variables depends on the MPI process rank. For example, as shown in figure 1, only the rank 0 uses the vector w^{MPI} to save the local task reports.

Table 2 MPI *task* state extension.

Variable	Description
w^{MPI}	Vector of *guess workers*. Stores one for each MPI process.
$finished^{MPI}$	Flags MPI balancing finish
I_n^{MPI}	Iterations to do between all MPI processes
$finish_{req}^{MPI}$	Flags MPI finish request
$finish_{sent}^{MPI}$	Flags MPI finish request sent

With these modifications, the report and balance steps are handled by a single thread in each MPI process via the *monitor* method. This one has a different behaviour regarding its rank number, as shown in figure 4. Both are explained below.

For rank 0 (figure 4 left), Δt_i^{report} and Δt_i^{next} save, respectively, the elapsed time between reports and the time until next report for the *guess worker* number i. Then, *receiveAny* waits until some request is received, regardless the origin rank, or until the elapsed time reaches the *timeout*. In both cases, the elapsed time will be stored at Δt. If a request is received, it is stored at *req*. After the *receiveAny* call, the time until the next report request for each MPI process will be updated according to Δt. Also, if $\Delta t >= \Delta t_i^{next}$, a report will be requested to the process with rank i. Already sent report requests are flagged with $\Delta t_i^{next} = 0$. Finally, the timeout is set to the minimum value in the Δt^{next} array.

Regarding the procedure to handle the requests, there exists three possible requests. The first one, with identifier 0, handles the workers start petitions. As response to this request, the rank 0 sends a preliminary iteration assignation that will be updated when the first report is received. This part of the code uses the auxiliary method $done^{MPI}()$, which returns the number of the predicted iterations done by all the MPI processes.

The second instruction, with identifier 1, handles the reception of the reports. For that purpose, the method *receiveReport* is used to handle the petition. The functionality of *receiveReport* is very similar to the already shown methods *report* and *checkpoint*, except that it works with predictions of the computed iterations via the *guess worker addMeasure* method. So, it stores the new measure, updates the iteration assignment for MPI workers, and sends to the rank i its new assignation together with a flag to indicate if the MPI balance continues or finishes. As local balance (section 2.1), this will finish when the predicted remaining time is below the threshold. When the MPI balance finishes, the number of assigned iterations for each MPI process will remain unaltered hereinafter. To save space, the pseudocode of this function is not included at this document. However, the details can be found at the provided source code repository. Finally, once the response has been sent, the corresponding time until the next report and the timeout are updated.

The last instruction, with identifier 2, handles the finish requests. Like the method used at section 2.1, MPI workers can request to finish the task, attaching a report to their request. The reasons to send a finish request will be explained at the *monitor* description for non zero ranks. For instance, these requests are handled by *receiveReport* too. Finally, we check if all workers have been notified that the MPI balance has finished. In this case, the monitor execution ends.

For the other ranks, which constitute the MPI workers, the monitor pseudocode is shown in figure 4 right. First of all, the monitor sends a start petition to the rank 0 and receives the initial assignation of iterations to do. Once inside the loop, the function *waitAny* waits to receive a petition or a response from the rank 0 or until the value of the variable $finish_{req}^{MPI}$ changes to *true*.

On the first case, whether the received instruction identifier is 1 or 2, the monitor sends the predicted computed iterations (I_d^{pred}) at time instant t. Then, it waits to receive the response of the rank 0 with the new iteration assignation and the flag to finish the MPI balance ($finished^{MPI}$). If the MPI balancing has finished, the monitor process ends. Finally, if this request is a response of a finish petition (instruction 2), the $finish_{sent}^{MPI}$ is set to $false$ to allow triggering new finish petitions.

Instead, if $finish_{req}^{MPI}$ has changed its value to *true*, the monitor sends an instruction petition 2 to ask to finish the MPI balance. Also, the values of the flags $finish_{req}^{MPI}$ and $finish_{sent}^{MPI}$ are changed to $false$ and *true*, respectively. The value of $finish_{req}^{MPI}$ can be changed to *true* by local threads when they try to finish the task. This happens when a worker satisfies the criteria to finish the local task shown in section 2.1. However, if the MPI balance is still active, the number of iterations to carry out could change. For instance, the local task cannot allow its workers to exit the task. Instead, the local task sends a finish petition to rank 0. In addition, the flag value could also change when a local *checkpoint* call reaches a remaining time lower than the threshold.

3 Results

To test the efficiency of the proposed algorithm, we have simulated the variable overhead caused by neighbour VMs on an on-premises cloud managed by OpenStack. Its underlying infrastructure is composed by nodes with two Skylake Gold 6130 at 2.1 GHz with 16 cores each and 768 GB RAM DDR4@2666.

The deployed infrastructure for our experimentation consists of two physical nodes, as shown in figure 5. On the first, a single VM was deployed with 64 vCPUs to ensure that the physical node is not shared with any other VM. The second one is filled with smaller VMs with 8 vCPUs each one. On the second node, only one of the small VMs will execute the PenRed simulations. Also, four of the other small VMs, will execute a dummy process whose CPU usage depends on the time of day. These overhead tasks are bash scripts which run the command *yes* followed by a *sleep*. The sleep time depends, as we said, on the time of day. With this approach, we simulate a variation of the CPU usage of the neighbours VMs. The other VMs remain idle, and their only purpose is to fill the physical node.

Regarding the application to balance, we have selected PenRed [6] code system, which implements the PENELOPE [13] physics in an extensible parallel engine for radiation transport in matter simulations. Some of its usages are performing simulations of clinical radiation treatments, radiological protection, or industrial applications. To test RUPER-LB we will use the PenRed simulation example *2-plane*, provided as part of the software distribution.

```
$monitor()$
Input:
Output:
for $i = 0$ until $w^{MPI}.size() - 1$ do
  $\Delta t_i^{report} \leftarrow \Delta t_{pc}$
  $\Delta t_i^{next} \leftarrow 0$
end for
$timeout \leftarrow \Delta t_{pc}$
while $true$ do
  $req \leftarrow receiveAny(timeout, \Delta t)$
  $timeout \leftarrow 10^9$
  for $i = 0$ until $w^{MPI}.size() - 1$ do
    if $\Delta t_i^{next} > 0$ then
      if $\Delta t_i^{next} <= \Delta t$ then
        $requireReport(i)$
        $\Delta t_i^{next} \leftarrow 0$
      else
        $\Delta t_i^{next} \leftarrow \Delta t_i^{next} - \Delta t$
        if $timeout > \Delta t_i^{next}$ then
          $timeout \leftarrow \Delta t_i^{next}$
        end if
      end if
    end if
  end for
  if $req$ then
    if $req.instruction = 0$ then
      $I_{rem} = I_n^{MPI} - done^{MPI}()$
      $req.send(I_{rem}/w^{MPI}.size())$
      $\Delta t^{next}_{req.node} \leftarrow \Delta t^{report}_{req.node}$
    else if $req.instruction = 1$ then
      $\Delta t^{report}_{req.node} \leftarrow receiveReport(req)$
      $\Delta t^{next}_{req.node} \leftarrow \Delta t^{report}_{req.node}$
      if $timeout > \Delta t^{next}_{req.node}$ then
        $timeout \leftarrow \Delta t^{next}_{req.node}$
      end if
    else if $req.instruction = 2$ then
      $receiveReport(req)$
    end if
    if $allFinished()$ then
      exit
    end if
  end if
end while
```

```
$monitor()$
Input:
Output:
$I_n \leftarrow send(0)$
while $true$ do
  $req \leftarrow waitAny(finish^{MPI}_{req})$
  if $req$ then
    if $req.instruction = 1$ or $2$ then
      $t \leftarrow actualTime()$
      $I_d^{pred} \leftarrow predDone(t)$
      $req.send(t, I_d^{pred})$
      $(I_n, finished^{MPI}) \leftarrow req.receive()$
      if $finished^{MPI}$ then
        exit
      end if
      if $req.instruction = 2$ then
        $finish^{MPI}_{sent} \leftarrow false$
      end if
    end if
  else
    $send(2)$
    $finish^{MPI}_{req} \leftarrow false$
    $finish^{MPI}_{sent} \leftarrow true$
  end if
end while
```

Figure 4 Methods *monitor* of the object *task* for MPI rank 0 (left) and greater than zero (right).

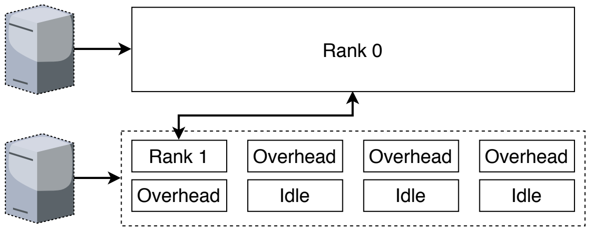

Figure 5 Test infrastructure schema.

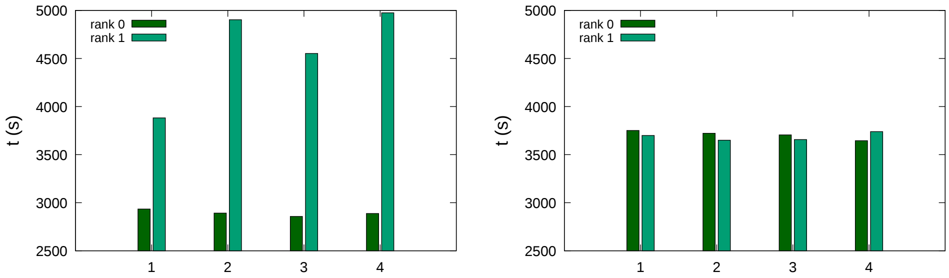

Figure 6 Execution time using 2 MPI processes with 8 threads each one. Left: without load balance. Right: with load balance.

With that experimental setup, we have executed the very same simulation with and without load balancing. We have configured the minimum time between checkpoints (Δt_{pc}) to 300 s, which has been selected according to process execution time order. Thus, we expect to see executing times delay between ranks and threads lower than 300 s. In the following experiments, two MPI processes have been used. The process with rank 0 runs on the large VM, i.e. with no neighbour influence. Thus, the process with rank 1 is executed at the node with multiple tenants. In addition, both processes use 8 threads each.

The same simulation was repeated 4 times both with and without load balancing. Figure 6 shows the execution time of every process by rank number, for each simulation run. As we can see, on the load balanced results, the delay between ranks is smaller than the selected Δt_{pc}. At the following test, we have increased the computational cost increasing the number of iterations (Figure 7). As expected, maintaining the same value of Δt_{pc}, the relative differences on execution time are reduced.

For the same simulation, with load balancing enabled, figure 7 right shows the execution

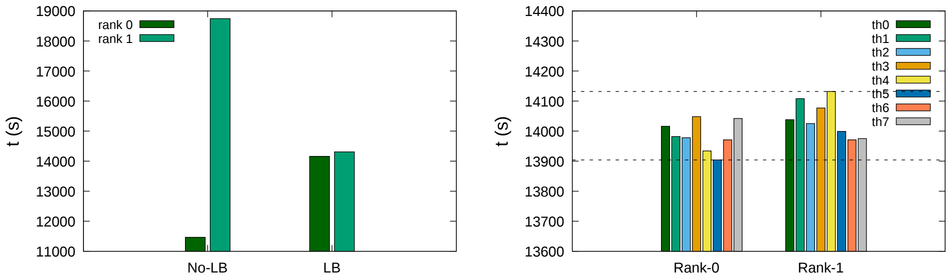

Figure 7 Execution times for simulations with a higher number of iterations, by rank (left) and by thread with load balance (right).

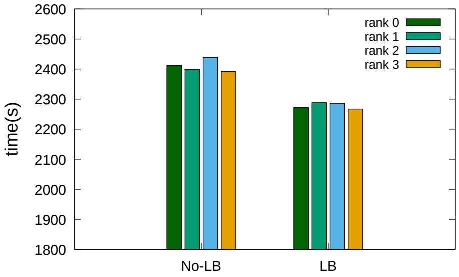

Figure 8 Simulations executed with 4 MPI processes and 8 threads each one on the single-tenant node.

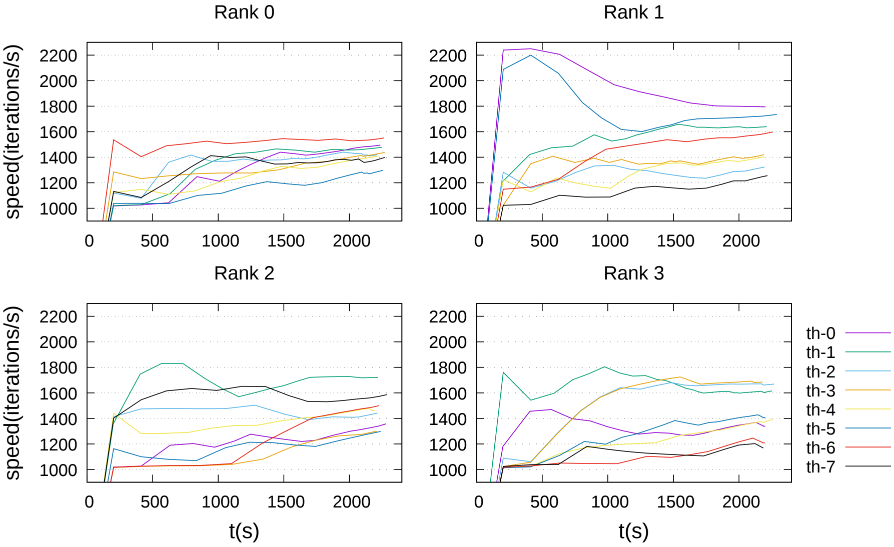

Figure 9 Evolution of the mean speed for each thread in each MPI process.

time for each thread of each MPI process. There, the dashed lines limits the fastest and the slowest thread for both ranks, and we can check that the corresponding delay is below Δt_{pc}.

To test how RUPER-LB can save execution time inside a single node, we have executed the same simulation using 4 MPI processes with 8 threads for each one, but all of them running on the single-tenant node. This simulation has been executed with and without load balancing. The corresponding execution times for each rank are shown in figure 8. The same simulation with load balancing enabled is about a $6-7\%$ faster. To understand the results shown in figure 8, we have represented the mean speed evolution of the threads of each MPI process in figure 9. As we can see, at the end of the execution the mean speeds present non negligible differences between the threads of the same MPI process. This fact explains why RUPER-LB achieves shorter execution times on this test. On the other hand, to explain why figure 8 seems to show no unbalance between ranks, notice that the execution time of each rank is determined by the slowest thread. Even if there exists unbalance between the threads, if the slowest thread of each rank requires approximately the same execution time in all of them, that gives the false appearance that the whole process is well balanced.

4 Conclusions

This work presents RUPER-LB, a load balancing system for applications with mixed MPI/multithreading parallelism support with loosely coupling. RUPER-LB focuses on iterative processes running on platforms with variable computational capabilities, such as cloud computing environments. We have shown the capabilities of RUPER-LB using a real world simulation software with MPI and multithreading capabilities. Due to its asynchronous approach, RUPER-LB introduces a negligible overhead on the processing time, making it suitable for applications with few communications. In addition, as RUPER-LB only require periodic reports of thread speeds, it is easily integrable on most applications.

Future work involves testing RUPER-LB running different kind of applications on both, public and on-premises cloud providers. Also, improving the finish request step to minimize threads waiting time. Finally, extending RUPER-LB to handle the iteration distribution for applications where the iteration migration requires some state transfer.

References

1 Aws single-tenant. `https://aws.amazon.com/ec2/pricing/dedicated-instances/`. Accessed: 2020-02-07.

2 Ali Abedi and Tim Brecht. Conducting repeatable experiments in highly variable cloud computing environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 287–292, New York, NY, USA, 2017. Association for Computing Machinery. URL: `https://doi.org/10.1145/3030207.3030229`, `doi:10.1145/3030207.3030229`.

3 M. Cierniak, M. J. Zaki, and W. Li. Compile-Time Scheduling Algorithms for a Heterogeneous Network of Workstations. *The Computer Journal*, 40(6):356–372, 01 1997. URL: `https://doi.org/10.1093/comjnl/40.6.356`, `arXiv:https://academic.oup.com/comjnl/article-pdf/40/6/356/1227981/400356.pdf`, `doi:10.1093/comjnl/40.6.356`.

4 J. Ericson, M. Mohammadian, and F. Santana. Analysis of performance variability in public cloud computing. In *2017 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 308–314, 2017.

5 P. Fan, Z. Chen, J. Wang, Z. Zheng, and M. R. Lyu. Topology-aware deployment of scientific applications in cloud computing. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 319–326, June 2012. `doi:10.1109/CLOUD.2012.70`.

6 V. Giménez-Alventosa, V. Giménez Gómez, and S. Oliver Gil. Penred: An extensible and parallel monte-carlo framework for radiation transport based on penelope, 2020. `arXiv:2003.00796`.

7 A. Gupta, L. V. Kale, F. Gioachin, V. March, C. H. Suen, B. Lee, P. Faraboschi, R. Kaufmann, and D. Milojicic. The who, what, why, and how of high performance computing in the cloud. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, volume 1, pages 306–314, Dec 2013. `doi:10.1109/CloudCom.2013.47`.

8 A. Gupta and D. Milojicic. Evaluation of hpc applications on cloud. In *2011 Sixth Open Cirrus Summit*, pages 22–26, Oct 2011. `doi:10.1109/OCS.2011.10`.

9 A. Gupta, O. Sarood, L. V. Kale, and D. Milojicic. Improving hpc application performance in cloud through dynamic load balancing. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 402–409, May 2013. `doi:10.1109/CCGrid.2013.65`.

10 A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 104–113, May 2011. `doi:10.1109/CCGrid.2011.22`.

11 K. R. Jackson, L. Ramakrishnan, K. Muriki, S. Canon, S. Cholia, J. Shalf, H. J. Wasserman, and N. J. Wright. Performance analysis of high performance computing applications on the

amazon web services cloud. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 159–168, Nov 2010. doi:10.1109/CloudCom.2010.69.

12 Philipp Leitner and Jürgen Cito. Patterns in the chaos—a study of performance variation and predictability in public iaas clouds. *ACM Trans. Internet Technol.*, 16(3), April 2016. URL: https://doi.org/10.1145/2885497, doi:10.1145/2885497.

13 Salvat F. Penelope. a code system for monte carlo simulation of electron and photon transport. *Issy-Les-Moulineaux: OECD Nuclear Energy Agengy*, 2014.

14 O. Sarood, A. Gupta, and L. V. Kalé. Cloud friendly load balancing for hpc applications: Preliminary work. In *2012 41st International Conference on Parallel Processing Workshops*, pages 200–205, Sep. 2012. doi:10.1109/ICPPW.2012.30.

15 Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1–2):460–471, September 2010. URL: https://doi.org/10.14778/1920841.1920902, doi:10.14778/1920841.1920902.

16 Shiv Shankar, John M. Acken, and Naresh K. Sehgal. Measuring performance variability in the clouds. *IETE Technical Review*, 35(6):656–660, 2018. URL: https://doi.org/10.1080/02564602.2017.1393353, arXiv:https://doi.org/10.1080/02564602.2017.1393353, doi:10.1080/02564602.2017.1393353.

17 F. Xu, F. Liu, and H. Jin. Heterogeneity and interference-aware virtual machine provisioning for predictable performance in the cloud. *IEEE Transactions on Computers*, 65(8):2470–2483, Aug 2016. doi:10.1109/TC.2015.2481403.