Performance Modeling of a Geophysics Application to Accelerate the Tuning of Over-decomposition Parameters through Simulation

Rafael Keller Tesser, Lucas Mello Schnorr, Arnaud Legrand, Christian Heinrich, Fabrice Dupros, Philippe Olivier Alexandre Navaux

Received TBD; | Revised TBD; | Accepted TBD

DOI: xxx/xxxx

RESEARCH ARTICLE

Performance Modeling of a Geophysics Application to Accelerate the Tuning of Over-decomposition Parameters through Simulation

Rafael Keller Tesser*[1,2] | Lucas Mello Schnorr[1,2] | Arnaud Legrand[2] | Franz Christian Heinrich[2] | Fabrice Dupros[3] | Philippe O. A. Navaux[1]

[1] Institute of Informatics, Federal University of Rio Grande do Sul – UFRGS, Porto Alegre, Brazil

[2] Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France

[3] BRGM, Orléans, France

Correspondence
Rafael Keller Tesser, Email: rktesser@inf.ufrgs.br

Summary

Finite-difference methods are commonplace in High Performance Computing applications. Despite their apparent regularity, they often exhibit load imbalance that damages their efficiency. We characterize the spatial and temporal load imbalance of Ondes3D, a typical finite-differences application dedicated to earthquake modeling. Our analysis reveals imbalance originating from the structure of the input data, and from low-level CPU optimizations. Ondes3D was successfully ported to AMPI/CHARM++ using over-decomposition and MPI process migration techniques to dynamically rebalance the load. However, this approach requires careful selection of the over-decomposition level, the load balancing algorithm, and its activation frequency. These choices are usually tied to application structure and platform characteristics. In this article, we propose a workflow that leverages the capabilities of SimGrid to conduct such study at low experimental cost. We rely on a combination of emulation, simulation, and application modeling that requires minimal code modification and manages to capture both spatial and temporal load imbalance to faithfully predict the performance of dynamic load balancing. We evaluate the quality of our simulation by comparing simulation results with the outcome of real executions and demonstrate how this approach can be used to quickly find the optimal load balancing configuration for a given application/hardware configuration.

KEYWORDS:
High-Performance Computing, Load balancing and over-decomposition, Performance prediction, Computer System Simulation, Geophysics FDM application

1 | INTRODUCTION

Simulations at scale have become essential for predicting earthquake ground motion. Ondes3D (1) is developed by computational science researchers at BRGM (French Geological Survey) as an implementation of the finite-differences method (FDM) to model the propagation of seismic waves in three-dimensional media. The simulations provide the relevant parameters for risk mitigation and assessment of damage in hypothetical earthquake scenarios. The design of this code is representative for a typical iterative application tailored for homogeneous HPC platforms with static domain decomposition. Despite the regularity of the FDM micro-kernels it employs, Ondes3D has limited scalability because of both spatial (2) and temporal load imbalance.

The performance of Ondes3D can be improved by updating its design to run on top of modern heterogeneous HPC platforms, composed of accelerators and traditional multi-core processors. Such endeavor is normally quite difficult and time consuming because it involves refactoring the way Ondes3D is internally organized. Although some partial efforts were already laid out towards this end (3), this usually leads to a complete new application design. The undesired effect is that the people who actually understand the physics behind the original code will be less likely to contribute anymore.

A viable alternative to improve load balancing with minor code modifications is to rely on domain over-decomposition and a runtime with support for dynamic process migration. Such a model has been implemented by Charm++ (4). Legacy MPI applications are supported in this runtime through the Adaptive MPI (AMPI) middleware (5, 6). AMPI is a full-fledged MPI implementation built on top of the Charm++ runtime and benefits from its load balancing infrastructure. It works by encapsulating each MPI process in a virtual process that can be dynamically migrated, when necessary, requiring global variables to be correctly encapsulated. Migrations occur when the non-standard `MPI_Migrate` operation is explicitly called by the application. The load balancer then decides the new process/resource mapping that will be used from that moment on, based on load measurements taken during the last observation period. The direct advantage is that any spatio-temporal load imbalance is dynamically mitigated, effectively improving performance, as previously observed in an Ondes3D upgrade (7) to AMPI.

The problem of the Ondes3D upgrade is that finding out the best configuration of the AMPI runtime involves real experiments at scale. Typically, one would want to identify the best (a) over-decomposition level, (b) load balancing heuristic, (c) load balancing frequency, and (d) number of resources to request. The best parameter choice depends directly on the characteristics of the platform. Tuning them is very resource and time-consuming, but is paramount to justify the upgrade to support dynamic process migration. As a consequence, quickly predicting beforehand how much performance gain is achievable with this kind of adaptive HPC runtime is fundamental.

In this paper, we propose a simulation-based methodology to quickly evaluate the potential performance benefits brought by adaptive MPI runtimes to legacy MPI codes. Our methodology is configured with a faithful model of the computation and communication signatures of Ondes3D for representative workloads on scale. All performance modeling is carried out with SimGrid's SMPI emulation (8, 9) and trace replay mechanisms (10), which have received extensive validation efforts (11, 12). We show that our methodology is faithful, in terms of total makespan as well as from the load balancing perspective observed in real AMPI runs. With our methodology, the application has to be executed only once to obtain a coarse-grain trace that can then be downscaled or upscaled to a different number of processes, using aggregation and extrapolation techniques. Once the traces have been adjusted to a different number of processes, they can be replayed multiple times to identify the best load balancing parameters for a given or hypothetical HPC platform. Since the replay is fast – usually a few minutes on a laptop – it enables a quick exploration of many of these parameters.

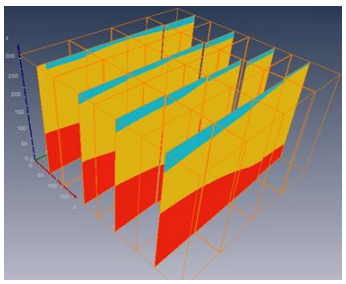
Section 2 details the performance modeling of Ondes3D. It characterizes the spatial and temporal load imbalances in the behavior of Ondes3D, as well as its key characteristics that enable us to aggregate and extrapolate behavior from a single representative trace of a real execution. Section 3 presents AMPI and difficulties of the load balancing evaluation process, serving as motivation for our main contributions on simulation. Section 4 details our simulation-based proposal, its methodological workflow, and its validation procedure. Section 5 details a comparison between our methodology and real executions, demonstrating that the observations from simulation are faithful with real measurements. We also confirm the usefulness of our fast methodology to identify a good load balancing parameter combination. Section 6 presents related work on simulation-based tools, justifying our design choices. Section 7 concludes the paper, listing our major contributions and future work.

This paper is an extended version of our paper published in the proceedings of the Euro-Par 2017 conference (13). In this extension, we detail the source of the temporal load imbalance of Ondes3D and build on its characteristics to understand how spatial trace aggregation (Section 4.1) can be used together with domain decomposition scaling techniques (Section 4.2) to further accelerate the simulations (Sections 4 and 5.5). These techniques are used together to artificially create trace files for a reduced number of MPI ranks, using as input the computational behavior of a single emulation of the original Ondes3D code. They are employed in our upgraded SAMPI workflow (Section 4). We employ a reproducible research methodology (14), using R, git and a laboratory notebook. All the data that has been collected in this work is publicly available[1]. Many parts of our SAMPI simulation framework have recently been integrated in the SimGrid source code [2], for the benefit of other simulators that require dynamic workload information.

[1] Dataset and analysis source code for this article is at https://zenodo.org/record/1289969
[2] See https://github.com/simgrid/simgrid/pull/214

2 | ONDES3D: A TYPICAL IMBALANCED MPI CODE

Ondes3D (1) is a geophysics simulator to conduct seismic hazard assessment at regional scale. It approximates the partial differential equations (PDE) governing the elastodynamics of rock medium using the finite-differences method (FDM) (15). The problem domain is statically partitioned in cuboids, as depicted in Figure 1a. Each iteration (see Figure 1b) corresponds to a given time step and consists in calling three macro kernels (`Intermediates`, `Stress`, and `Velocity`) that apply a series of FDM micro kernels (see the `CPML4` example in Figure 1c) to the whole domain. Message passing consists in asynchronous neighborhood communications intertwined with the three macro kernels. There is no global barrier, which enables a slightly asynchronous evolution of each process depending on the computational cost associated to a given part of the subdomain decomposition. Each process manages a cuboid with the same fixed geometry and runs the same regular code. Despite this apparent regularity, Ondes3D suffers from major load imbalance that limits its scalability. The main source of imbalance has been previously identified (2) as the extra-computation needed to deal with boundary conditions. The processes on the border of the domain have thus more work, which causes significant spatial imbalance when compared to processes that compute the inner part of the 3D region being simulated.



(a) 3D view of the heterogeneous rock medium, with a 4×4 domain decomposition; each process calculates a cuboid.

```
for (ts = 0; ts < N; ts++){
  Intermediates();

  Stress();
  //Intertwined Asynchronous
  //Neighborhood Communication

  Velocity();
  //Intertwined Asynchronous
  //Neighborhood Communication

}
```

(b) The main loop with three kernels: `Intermediates`, `Stress` and `Velocity`; no global synchronization.

```
static inline double CPML4 (double vp, double dump,
      double alpha, double kappa, double phidum, double dx,
      double dt, double x1, double x2, double x3, double x4) {
  double a, b;
  b = exp(-(vp * dump / kappa + alpha) * dt);
  a = 0.0;
  if (abs(vp * dump) > 0.000001)
    a = vp * dump * (b - 1.0) / (kappa * (vp * dump + kappa * alpha))
      ;
  return b * phidum + a *
    ((9. / 8.) * (x2 - x1) / dx - (1. / 24.) * (x4 - x3) /dx);
}
```

(c) The `CPML4` kernel, called nine times for each cell i, j, k in the `Intermediates` kernel when processing a sub-domain. Variables `x1`, `x2`, `x3`, and `x4` represent the rock medium states (e.g., speed) that evolve along the iterations.

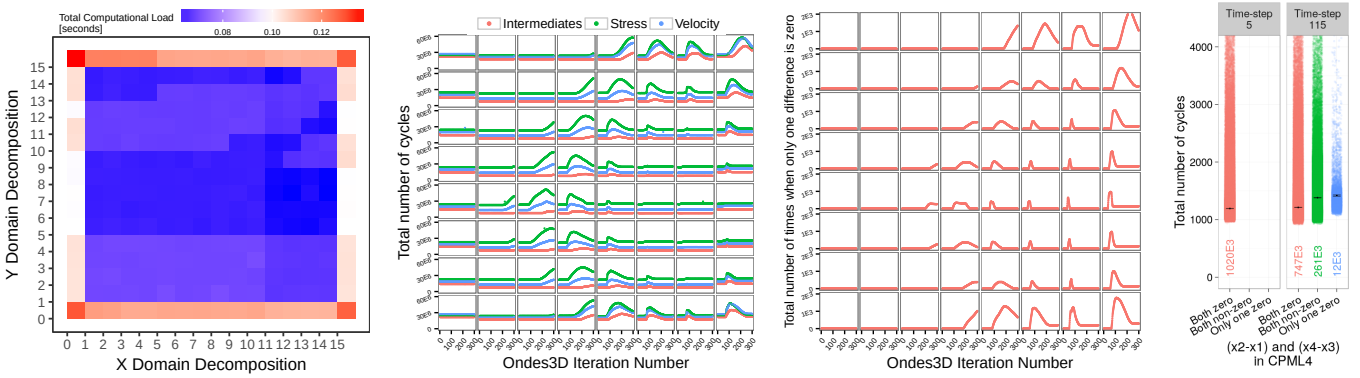
FIGURE 1 The Ondes3D application: (a) an example of domain decomposition for 16 processes; (b) the three large kernels of the main loop, with intertwined neighborhood communications; (c) a detailed view of the small `CPML4` kernel (out of 24).

In Section 2.1, we confirm the previously identified spatial imbalance, and report another, more subtle, source of spatial imbalance. This is caused by the heterogeneous simulation substrate. Until now, due to the highly regular structure of the code, studies had solely focused on spatial imbalance and had overlooked possible temporal imbalance. We also present that this imbalance is much stronger than the spatial one, and present evidences of its origin being related to low-level optimizations taking place inside the CPU. Both spatial and temporal load variations are highly related to the nature of the computation and thus seem quite hard to predict with accuracy.

Section 2.2 presents a discussion about how dynamic load balancing can be effectively employed to improve the performance of Ondes3D. Below, in an in-depth study, the spatial and temporal load imbalance of Ondes3D is illustrated with a historical Mw 6.3 earthquake workload (16) that arose in Liguria (north-western Italy). The code has been compiled with GCC 6.1.1 with `-O3` and instrumented with PAPI (17) and SMPI (9) and executed for various decomposition levels and numbers of processes. While we report results only for this workload and setup, we have systematically observed similar issues with other workloads, CPUs (Xeon X3440, X5650, E5-2630, and i7 4600M), and compilers.

2.1 | Characterizing Spatial and Temporal Load Imbalances

Figure 2a depicts a 16×16 domain decomposition where each cell in the cartesian grid represents one of the 256 processes in charge of a cuboid subdomain. In this heatmap, the color indicates the total computational load per process during the first iteration of the Ondes3D program, before the triggering of the initial shock that originates in the subdomain at position 13×5. Processes on the borders demonstrate a much higher computational load (red color) than those located inside the physical domain. Another, much more subtle, source of spatial imbalance (blue shades), depends mostly on the rock multi-layer configuration of the input (six layers for this scenario). Although minor, such effect exists and solely depends on the substrate geometry.



(a) Spatial imbalance for the first iteration represented by a color gradient for each rank in a 16×16 grid (256 processes).

(b) Total number of CPU cycles for each kernel along iterations for each rank in a 8×8 grid (64 processes).

(c) How much the differences evaluate to contrasting values in the `CPML4` kernel along iterations, for each of the 64 ranks.

(d) Expected duration (cycles) and number of calls of `CPML4`.

FIGURE 2 Load imbalances for the Ligurian workload: (a) spatial load imbalance; (b) temporal load imbalance for three kernels; (c) `CPML4` substrate variables evaluating to distinct values; and (d) explaining temporal anomaly with `CPML4` argument analysis.

The Ondes3D code does not exhibit any branching structure (convergence loops, refinements, thresholds) that could contribute to an evolution of computational load along simulation iterations. There are conditional branches, but they are related to absorbing conditions and thus solely to the fixed problem geometry. Yet, as illustrated in Figure 2b, one can observe a variability in computational cost along iterations that is even higher than the spatial variability incurred by the absorbing boundary conditions. This figure details the behavior of 64 processes (each box in the 8×8 grid), showing (in the vertical axis of each box) the total number of CPU cycles (the `PAPI_TOT_CYC` hardware counter) per macro kernel as a function of the iteration (horizontal axis). This metric seems to follow the earthquake shock progression, standing out around the eightieth iteration.

To explain the origin of this variable computational cost, we use the `CPML4` kernel shown in Figure 1c. This kernel is only one out of the 24 small inlined kernels but is perfectly representative of the other ones. It is called by the `Intermediates` macro kernel that iterates over the cuboid sub-domain with three nested loops. For each cell of the subdomain, the `CPML4` kernel is called nine times with slightly different parameters, resulting in more than a million calls per process/iteration in a 64-process simulation, for this workload. In this code, the variables `dx` and `dt` are simulation constants, while variables `x1`, `x2`, `x3`, and `x4` represent the rock medium state, such as speed, pressure, and others, that unfolds along the iterations.

For completeness, we proposed and verified several hypotheses to explain the temporal load variation, among them: conditional branches in micro kernels (e.g., the `if` in `CPML4` code of Figure 1c), variable duration of math functions depending on their input (e.g., the `exp` call in the `CPML4` code), arithmetic exceptions at the architecture level, compiler level optimizations that could, for example, introduce conditional branches in the assembly code, data cache effects, higher cost of divisions when compared to multiplications, and so on. By either playing with compiler options, carefully looking at the assembly code, or finely instrumenting the different operations, we have ruled out all these assumptions. Nevertheless, we noticed a strong correlation between the evolution of the load and the instruction cache misses (`PAPI_L2_ICM`) and branch miss-predictions (`PAPI_BR_MSP`). As we will show next, this correlation is a consequence and not the cause of the evolution.

Let us consider the `x1`, `x2`, `x3`, and `x4` arguments of the `CPML4` kernel (still in Figure 1c). The `return` statement contains an arithmetic expression, evaluated in the FPU, composed by many operations including two differences (`x2-x1` and `x4-x3`). We have instrumented the `CPML4` kernel to count how many times per time step and per process each of these differences are equal to zero. For this purpose, let us calculate $z(x, y)$ where z results in 1 when $(x - y) = 0$ and results in 0 otherwise. Then let us count the number of occurrences of $z(x, y) = 1$. Looking at $z(x2, x1)$ and $z(x4, x3)$ separately indicates no correlation with the temporal load evolution. However, the expression $\sum_{i=1}^{i=n} |z_i(x2, x1) - z_i(x4, x3)|$, depicted in Figure 2c, where n is the number of executions of `CPML4`, is perfectly correlated with the computational load change shown in Figure 2b, and with the growth of the branch miss-prediction counters. Intuitively, this value measures how often only one of the two differences is zero.

To confirm this hypothesis, we instrumented the `CPML4` kernel to record its duration for each call (in cycles) along with the result of the two differences ($x2 - x1$ and $x4 - x3$). Since this kernel is called very often, we traced only one rank in the top-right corner of the domain decomposition, which demonstrates a strong variability in time, and only for two very different time steps: iteration 5 with low computational load and iteration 115 with maximal computational load. Figure 2d shows the number of cycles of a single call to the `CPML4` kernel depending whether both differences are zero, non-zero or whether only one of them

is zero. Just after simulation starts, in iteration 5, when the seism has not yet reached this region, both differences are always zero (red points in Figure 2d) and the number of cycles is relatively small. In iteration 115, both differences are non zero (green points in Figure 2d) for more than 25% of the `CPML4` calls and the average execution time is then 168 cycles slower. The FPU optimization to speed up multiplications by zero is thus worthwhile. Interestingly, the situation where only one difference is zero (blue points in Figure 2d) is sporadic (about 1% of calls) but both the average and the minimum execution time are significantly longer. It is interesting to note that the corresponding cells i, j, k for which the condition is true are spatially organized but they do not relate at all to cache alignment and do not generate additional data cache misses but clearly more branch miss-predictions and L2 instruction cache misses that participate in the slow down of the code. The observed increased duration originates from the combination of both a speed-up of multiplications by zero and of branch miss-predictions in the FPU incurred by the irregular sequence of zeros and non-zeros.

Finally, even if we focused on the `CPML4` kernel here, all other small inlined kernels share the same structure. It is thus the aggregated contribution of all these few additional cycles that generates the temporal load variation.

2.2 | The Necessity of Dynamic Load Balancing

On modern architectures, Ondes3D is, fundamentally, a spatially and temporally imbalanced code, despite the regularity of its computational kernels and data partitioning. Such imbalance is usually overlooked, as it requires a careful and fine analysis to be identified. Therefore, we believe that it may also be present in many other so-called regular applications. Another recent example is the MPDATA code, part of the EULAG (Eulerian/semi-Lagrangian fluid solver), which has been shown to suffer from heavy spatial imbalance when executed on a recent Xeon Phi architecture (18) despite its strong regularity.

Modeling and predicting the load imbalance of Ondes3D is difficult, as it strongly depends on the initial and evolving conditions of a given earthquake simulation. Load balancing at the application level would require previous knowledge of the evolution of the load distribution along the execution, which is dependent on the input configuration of earthquake simulation. Besides, we have to consider the high amount of effort that has to be undertaken by the application developer to put such scheme in place. That is why we propose to use a simpler approach (from the application developer mindset) by mixing load balancing at runtime with over-decomposition. A strategy in the scope of the AMPI framework (5, 6). The next section presents our AMPI-aware version of Ondes3D.

3 | ADAPTIVE MPI: LOAD BALANCING WITH OVER-DECOMPOSITION

AMPI stands for Adaptive MPI (5, 6), and is an MPI implementation built upon the Charm++ framework and runtime (4). Thus, MPI applications ported to AMPI benefit from the underlying dynamic load balancing infrastructure. Charm++ enables the decomposition of the problem domain in more tasks than the number of available physical cores (i.e., over-decomposition). Tasks are mapped to Virtual Processors (VP), which are implemented as user-level threads suitable for migration. At runtime, many VPs are mapped to the same system process which is pinned to a physical processor. The runtime can migrate these VPs between system processes. Together, over-decomposition and migration enable Charm++ to periodically redistribute tasks, according to a load balancing heuristic. The heuristics use load information from the near past to define a new VP mapping. Therefore, the load balancing in AMPI is dynamic since it responds to load fluctuations during runtime.

To exploit AMPI's load balancing functionality, the application has to be changed in three ways. (1) First, relying on user-threads requires the code to have no global or static variables (otherwise they would be shared among VPs). (2) Second, Pack-and-Unpack (PUP) functions must be implemented to serialize dynamically allocated data and allow migrations to take place. When the load balancer rules for a migration, thread data is packed in the system process of origin and moved to be unpacked in the destination process. Writing these functions can be tedious, since it may entail a profound understanding of the data structures of the application. (3) The last modification is the addition of a call to `MPI_Migrate`. When triggered, this call indicates that the application is ready for load balancing. The application developer must call this function only when there are no active communications or open files. Typically, this call is placed at the end of the outermost loop iteration (the time-step loop in the case of Ondes3D), being thus called periodically.

3.1 | Performance-impacting Parameters and Potential Performance Gains

A number of parameters influence the effectiveness of the load balancing. Some **load balancing heuristics** are naturally more scalable than others. In particular, there is a trade-off between achieving a perfect load balancing and the time spent migrating

data to achieve it. The best heuristic therefore highly depends on the application dynamicity and on platform characteristics. The **level of over-decomposition** influences how well the load balancer is capable to redistribute the load. In this aspect, the more sub-domains, the better. Conversely, increasing the amount of sub-domains generally increases the communication cost at the application level. At some point, this cost exceeds the benefit of load balancing. The number of tasks also increases the amount of work done by the load balancer, which can become significant at large scale. Likewise, **the number of computing resources** is a critical parameter in the overall performance of a parallel program and the more is not always the better. Finally, fine-tuning the **frequency of load balancing** is essential to obtain good performance. Indeed, the more frequent the load balancing, the more likely we are to detect load imbalance and tackle it. Yet, if the balancing is too frequent, it will incur too much overhead. Moreover, since calling `MPI_Migrate` incurs a global barrier, it may also destroy any natural compensation of load imbalance throughout iterations afforded by asynchronous neighborhood communications.

All these parameters impact directly the performance of Ondes3D, which has been already ported to AMPI. This resulted in significant gains, up to 28.35% on an 8-node cluster (64 cores) (7). Here, we present new performance measurements on a 12-node cluster (288 cores) at BRGM to illustrate the typical gains of the dynamic load balancing provided by AMPI and how expensive is the search for the best values for the load balancing frequency and over-decomposition level.

This BRGM platform with its 288 cores is typical of the scale at which Ondes3D is expected to run. It has two AMD Opteron 6344 processors per node, using an Infiniband 40G network (MT27500 Family). We simulated the first 500 time steps of the Mw 6.6 2007 Niigata Chuetsu-Oki earthquake (19), using a grid of dimensions $1152 \times 1152 \times 384$, adjusted to the increased computational power of this cluster. We evaluate four load balancing heuristics (RefineLB, HwTopoLB (20), and HierarchicalLB (21)) with two over-decomposition levels (8 and 16), and an `MPI_Migrate` call every 20 time steps. RefineLB, part of Charm++, has been chosen because it was the best performing one in our previous work. HierarchicalLB is a distributed hierarchical algorithm that employs load balancing at two levels: platform level and host level. The tested configuration employs HwTopoLB at both levels. Results are compared against the pure MPI-based implementation and the AMPI one with no load balancing (baseline). The results in Figure 3 show the average makespan of at least 10 executions for each case, with a 95% confidence interval. Our AMPI baseline is approximately 6% better than the default MPI library. In a standard situation, with only one VP per physical core, AMPI is not designed to outperform MPI libraries. Nevertheless, results described in (22) with a three-dimensional Jacobi stencil also demonstrate significant speedups over the native MPI library (a maximum of 12% on 512 processors) without any overloading strategy. The authors explain these results by the benefits coming from the internal mechanisms of the AMPI library. Additionally, as explained in (6), the AMPI library can benefit from a better strategy for communications at the shared-memory-level in comparison with costly copies of buffers generally implemented by the native MPI library. RefineLB is about 36% faster (using an 8-factor over-decomposition) than the baseline and the best load balancer overall, probably because topology awareness was not essential in this homogeneous cluster with a low-latency network. This clearly demonstrates the significant benefits of dynamic load balancing and over-decomposition to iterative MPI codes such as Ondes3D.

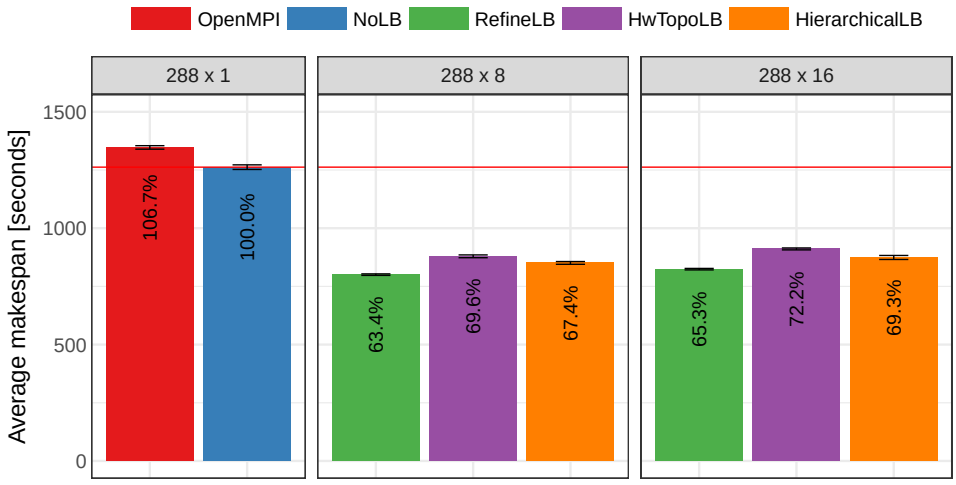


FIGURE 3 Mean execution time of Ondes3D on 288 cores with different task granularities (VP counts) and load balancers.

3.2 | Difficulties to Evaluate the Benefits of Load Balancing

While, as demonstrated above, evaluating the benefits of load balancing with real execution is feasible, it presents many difficulties. Several parameters (heuristic, level of decomposition, frequency, and so on) can strongly impact performance of the load balancing and the **optimal configuration** may be dependent on application dynamics and on platform characteristics. The results presented in the previous section required an empirical exploration of this configuration space. Running the same Earthquake simulation several dozens of times at scale on a production system solely to determine such parameters is **resource and time consuming**.

This is why we propose a simulation workflow that addresses these difficulties in a lightweight way. The application benefits can be evaluated with minimal code changes (even if it has not been ported to AMPI yet), and it needs to be executed only once to outline its execution as we have shown in Section 2. These features save development and evaluation time. Besides, the tracing process can be done using sequential emulation, which requires a single host.

4 | SAMPI: A SIMULATION WORKFLOW TO EVALUATE AMPI PERFORMANCE

Simulated AMPI (SAMPI) is our simulation workflow, which aims to provide an estimation of the benefits of over-decomposition and dynamic load balancing at low cost. Each simulation should be fast, use few resources and require minimal modification of the application. Besides that, we focus on the improvement of legacy iterative MPI-based applications such as Ondes3D. Therefore, we mimic the behavior of the dynamic load balancing algorithms implemented by Adaptive MPI. Our workflow relies on SimGrid's SMPI (9), which offers two key features on which we have built. (1) First, SMPI allows to study MPI applications either in *emulation mode* or through *trace-replay*. In emulation mode, unmodified MPI applications are sequentially executed, in a controlled way, on top of the simulator. Such approach can be costly in terms of simulation time (all the computations are executed) but faithfully captures the behavior of complex applications while using only one core. In trace replay mode, the events of an MPI application are replayed on top of the simulator, which is much faster than a normal execution at full scale. It typically takes one or two minutes on a single core compared to 15 minutes on a cluster for our Ondes3D simulations. Thus, we uphold our resource requirement goal. (2) Second, SMPI builds on the hybrid flow-level network models of SimGrid (23) that allow to faithfully model communications and contention, which is essential in the context of our study since load balancing in AMPI induces Virtual Process migration that can be costly in terms of communications.

Figure 4 shows the initial SAMPI simulator workflow, which exclusively uses the trace-replay mode of SMPI. The simulator takes as input a `TIT` trace (Time Independent Trace, i.e., events have no timestamps) with a certain number of VPs. SAMPI replays the trace to provide makespan predictions for a given load balancing configuration (heuristic and frequency) and platform (number of hosts and their computing power, network bandwidth/latency). The use of the trace replay mode allows to quickly and easily explore the AMPI parameter space to find the best combination of load balancer heuristic and frequency, for a given application workload (earthquake).

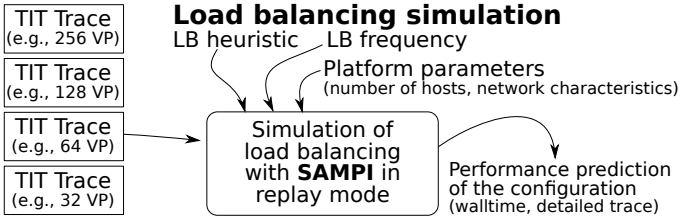


FIGURE 4 `TIT` traces with different numbers of VPs (256, 128, 64, and 32 in this example) are received as input; the `SAMPI` simulator replays these traces to provide makespan predictions for alternative load balancing heuristics, migration frequency, and platform characteristics (number of hosts, network capacity).

Three main modifications in SMPI have been necessary to make SAMPI possible. (1) First, we had to augment the SMPI interface with the non-standard `MPI_Migrate` function both in the emulation mode (to generate an event in the trace) and in the trace replay mechanism. In trace replay (the last step of the workflow), whenever the load-balancing is activated, this function

calls the `MPI_Barrier` function, the load balancing heuristic that defines the new process mapping, and simulates all resulting processes migrations. (2) To obtain a faithful behavior of AMPI load balancing heuristics, it is essential to stay as close as possible to their real implementation. We investigated the possibility to directly plug Charm++'s load balancers into our simulator, but they heavily rely on Charm++ specific elements. Fully porting the Charm++ runtime ecosystem on top of SimGrid would require a significant development effort and a deep knowledge of the Charm++ internals. Instead, we have manually extracted and slightly adapted by hand two centralized load balancers: **GreedyLB** and **RefineLB**. These were chosen due to being relatively simple, centralized, load balancers, that make their decision based on a single metric (the amount/duration of computations). This facilitates their integration into the simulator. Moreover, despite the apparent simplicity of their implementation, they can achieve surprising good results in some cases, as indicated both by our previous work (7) (where RefineLB was the best option) and by the results presented in this paper. Besides, we believe they are sufficiently different in their approaches to provide an interesting comparison. Their integration into SAMPI consisted mainly in removing internal references to Charm++, making sure that the heuristic implementation remains intact. A few trace replay routines also had to be modified to collect the load data that is fed to the load balancing heuristics. (3) Finally, the migration cost is accounted for in the simulated execution time. We rely on SimGrid's contention-aware network models when sending the data belonging to the migrated task from its original location to its destination. The migration payload is estimated by trapping `malloc` and `free` functions in emulation.

The main problem of our SAMPI simulator is that it takes as input a different `TIT` trace for every specific VP count. There are two alternatives to obtain them: (a) to allocate resources on a cluster to run Ondes3D, trace its computing/communication behavior during the execution, and then to transform those traces to the time-independent format of `TIT` files; and (b) to emulate Ondes3D using SMPI in a single node, and automatically collect `TIT` files since these can be generated as output by SMPI. Each of these alternatives have advantages/drawbacks: the first option requires a reservation of the whole cluster, but trace collection is much faster. Besides the resource-consumption, another penalty is that traces have to be converted to `TIT` files; the second option is much cheaper in resources, but it is extremely time-consuming, since SMPI emulation serializes the execution of the application.

Our goal here is to reduce the cost of obtaining multiple `TIT` traces by deriving all of them from a single SMPI emulation, i.e., a single `TIT` trace. To do so, we propose two performance modeling strategies that build on the computational characteristics of Ondes3D. Even in the presence of the multiple sources of load imbalances (see Section 2), these two techniques enable to faithfully capture the computational behavior of Ondes3D:

1. **MPI Downscaling through Spatial Aggregation** (Section 4.1): we show how a coarse grain execution trace (e.g., 16 MPI processes) can be derived from a fine grain execution trace (e.g., 256 MPI processes) by spatially aggregating computational costs considering the fixed domain decomposition used in Ondes3D. Such procedure is straightforward, brings no information loss, and enables one to artificially reduce the number of MPI processes.

2. **Application Upscaling through Temporal/Spatial Extrapolation** (Section 4.2): we detail how we can derive fine grain execution traces from coarse grain by adjusting the temporal/spatial resolution of the problem domain. Although some computational performance details might be lost, the bulk of its computational behavior is kept, up to a certain extrapolation factor. This procedure makes it much faster to obtain an initial trace of the behavior of the application.

4.1 | Spatial Aggregation to reduce the number of MPI Processes

The previous analysis shows that both spatial and temporal load variation are nearly smooth but it seems difficult to anticipate the performance by solely looking at the input data, without running the code at least once. Yet, since this code relies on domain-based decomposition, we show that it is possible to derive a coarse grain execution trace from a fine grain execution trace, effectively creating a new trace with a reduced number of MPI processes. Indeed, the amount of work induced by a given area of the domain at a given time step corresponds to the sum of the amount of work of the corresponding sub-areas.

To illustrate how this process works, let us consider a representative scenario with a 16×16 decomposition, as shown on Figure 2a. When using instead a coarse grain 4×4 grid (a 16-factor spatial aggregation), the computational load of each process is the sum of the 16 corresponding processes in the fine grain 16×16 decomposition. To do so, we divide the X and Y coordinates of the original domain decomposition and then sum up the computational load of the 16 processes belonging to each combination of the X and Y coordinates of the target grid. Figure 5 shows the computing time (vertical axis) of each main loop iteration as a function of the time step (horizontal) for each rank (colors). The left facet shows the measurements from an execution with a

coarse-grain (4×4) grid. The right one shows the resulting computing time for 16 processes, but now aggregated from a fine-grain (16×16) grid. The difference between the coarse-grain trace and the aggregation from the fine-grain trace is minimal and allows to fully capture the temporal evolution for a given earthquake scenario after the spatial aggregation.

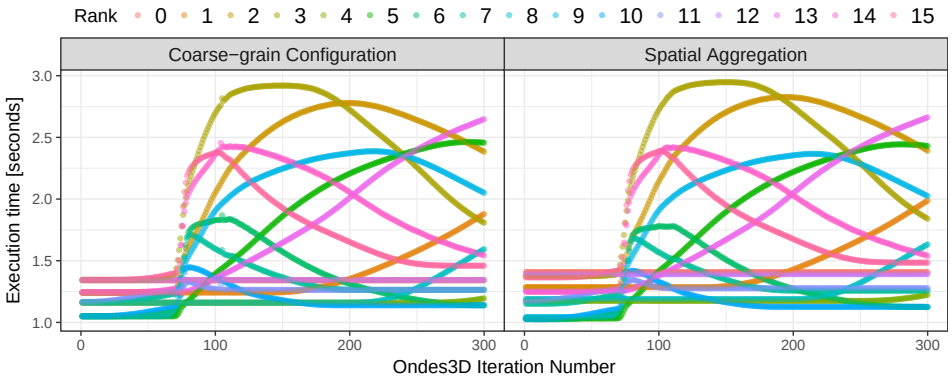


FIGURE 5 Execution time (vertical axis) along the time steps (horizontal) for each rank (color) considering all three kernels; the left facet is the behavior of a 16-process run, the right facet shows the same, but calculated from a 256-process run.

The procedure illustrated in Figure 5 is straightforward because it adopts division factors leading to rounded (in the integer space) downscaling, in both dimensions. However, very frequently we need to downscale the number of processes to a non-trivial configuration, i.e., from 256 to 21 processes. That would obviously lead to a non-integer repartition. Our approach to do spatial aggregation captures such non-trivial scenarios, enabling one to reconstruct any $P \times Q$ domain decomposition configuration. It works by creating a target grid and then calculating the computational intersection against the original grid, for every combination of new X and Y coordinates. Since the target grid is much more fine-grain (smaller cells), we need to sum up the computational cost of the original cells that is proportional to the area of the intersection. This methodology is illustrated in Figure 6, showing the original domain decomposition 16×16 (left) and four derived grids for which the computational behavior has been aggregated: 7×10 (center left), 4×4 (center right) and 3×5 (right). We have chosen to depict the `Velocity` kernel on iteration 115 of the Ligurian case since it is the moment with the largest load imbalance (see Figure 5). This mechanism allows us to generate coarse-grain traces at barely no cost using the `R` language and the `tidyverse` package.

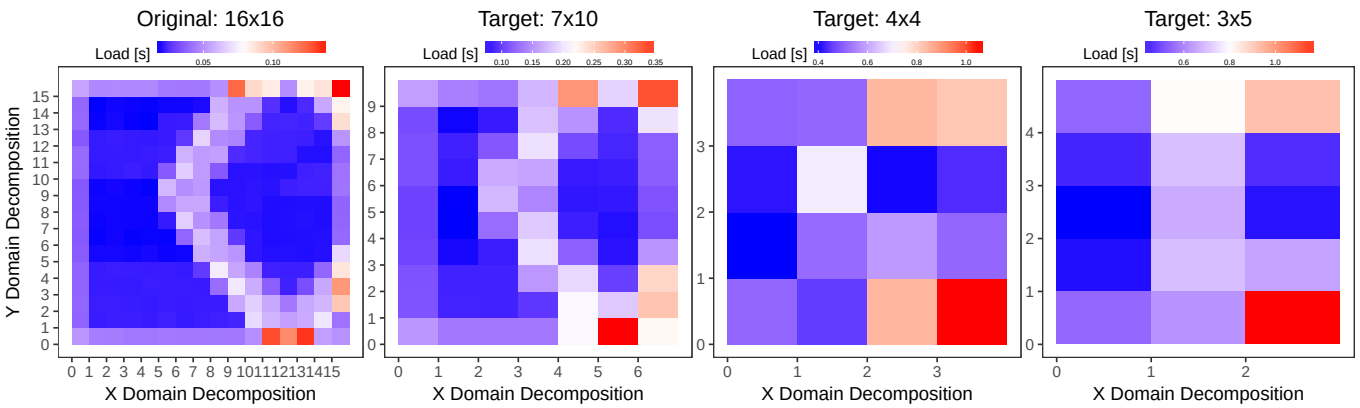


FIGURE 6 Four domain decompositions: the original 16x16 scenario and three spatial aggregation scenarios: 7x10 (center left), 4x4 (center right), and 3x5 (right) from the original 16x16 traces (left); the computational load scale (color palette) is different for each of the four cases.

4.2 | Application-Level Rescaling

The resolution of the domain that is simulated by Ondes3D is governed by two parameters: `ds`, which provides the spatial resolution, the distance between each element in the discretized domain; and `dt`, indicating the temporal resolution, the length

of the time step. From the geophysics perspective, both values can be slightly modified as long as their ratio remains in an acceptable range (according to the CFL numerical condition for stability criterion (15)). Increasing both values together by a certain factor, reduces the number of cells in the cuboids and the number of iterations and therefore decreases the total execution time at the price of a poor resolution of the results in term of physics. These values are set by geophysics investigators and are tailored for each workload. However, here we are interested, purely, in the computation time profile of each process in the domain decomposition. Running at a coarser-grain may therefore be sufficient to capture the evolution of the computational load. Figure 7 illustrates this scenario: each facet shows the duration of the computation (on the Y axis) of the `Stress` macro-kernel along the iterations (on the X axis). The left facet shows the computational behavior with the default `dt` and `ds` values, then the next four facets show the computational cost when we increase `dt` and `ds` together by the given percentage, up to 20%. We can see that as we lower the resolution (larger values of `dt` and `ds`), the computational cost becomes smaller and smoother (without the peak at the ≈100 iteration). The cost per iteration reduces, in average, from ≈0.0623, in the original version, to ≈0.0381 seconds (≈39% faster) in the case where `dt` and `ds` values get an increase of 20% (right facet). In this figure, we show only the `Stress` component, for only two ranks: rank 0 (red color, top-left corner in the fixed domain decomposition) and 63 (blue color, bottom-right corner in the domain) out of the 64-process run. The behavior of other ranks and of the other two macro-kernels are similar. This procedure only works up to a certain factor level, from which the computational behavior gets too smooth to be able to reconstruct the original shape. For the Chuetsu-Oki scenario, $\delta = 1.2$ (i.e., an increase of 20%) was the largest scaling factor allowing to faithfully observe the load imbalance evolution.

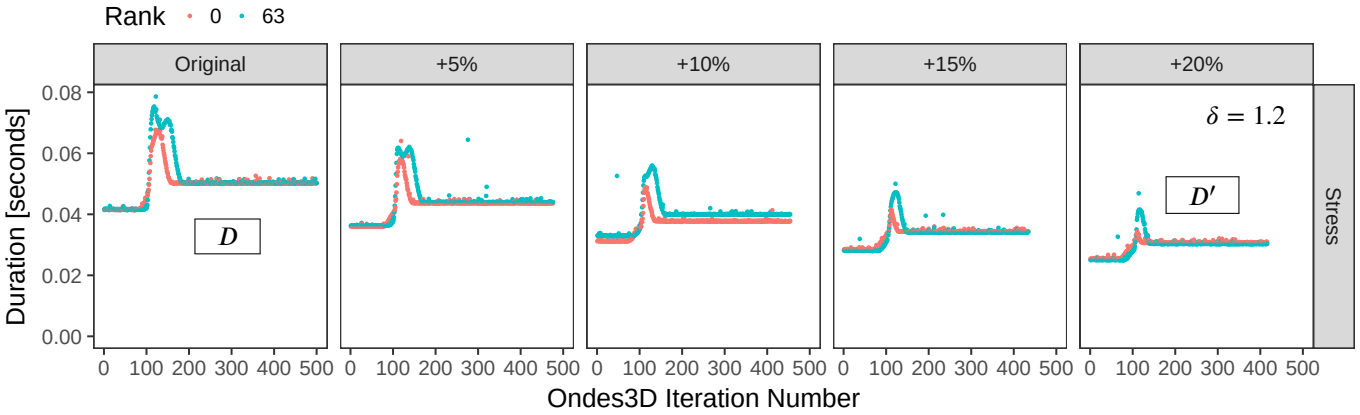


FIGURE 7 The duration (on the Y axis of each facet) for the `Stress` macro-kernel along the Ondes3D iteration numbers (on the X axis of each facet) for two processes (0, red, and 63, blue) of a 64-node run. The horizontal facetting indicates the scaling factor used towards a coarse-grain simulation: Original (left column), then 5% to 20% increase in `dt` and `ds` (i.e., a decrease in spatial and temporal resolution).

We now explain how the trace of a coarse-grain execution (larger `dt` and `ds` values) can be rescaled to obtain a faithful (in terms of computational behavior) approximation of a fine-grain execution (original `dt` and `ds` values). Let us denote by $D_{i,p}$ (resp. $D'_{i,p}$) the duration of computations at iteration i on rank p when using the original (resp. rescaled by a factor δ) `dt` and `ds`. Our goal is to reconstruct a trace $\widetilde{D}$ from D' such that $\widetilde{D}$ and D have the exact same number of iterations and $\widetilde{D} \approx D$. Since time is rescaled, the number of iterations needed to simulate T seconds is $N = T/\texttt{dt}$ (resp. $N' = T/\delta\texttt{dt}$). Therefore iteration i in D corresponds approximately to iteration $\lfloor i/\delta \rfloor$ in D'. From Figure 7, it is natural to define $\widetilde{D}$ as:

$$\forall i, p : \widetilde{D}_{i,p} = \alpha . D'_{\lfloor i/\delta \rfloor, p}, \text{ with a well chosen } \alpha.$$

Obviously, we could fit a perfect α to that $\widetilde{D}$ and D are as close as possible from each other but it would require fully knowing D, which is precisely what we try to avoid. We claim that a correct α can be found solely from the first iteration of D and fit α so that $\sum_p \left(D_{1,p} - \widetilde{D}_{1,p} \right)^2$ is minimized, which is obtained with a simple linear regression (with no intercept).

To summarize, approximating the high resolution traces from a coarse grain configuration involves three steps: (1) obtaining the coarse grain trace; (2) running the first iteration of the high resolution setup to obtain an estimation over all processes of how much additional work is required compared to the coarse grain setup; (3) computing the magnitude of the computational cost rescaling factor α and apply it to the fine-grain trace. Note that such rescaling should be performed kernel by kernel since the consequences of rescaling `dt` and `ds` by δ may be very different.

Figure 8 shows a comparison of the original behavior (left column) with two δ rescaling scenarios: when traces using an increase of `dt` and `ds` of 10% (center column) and a 20% (right column) are rescaled back to the original `dt` and `ds`. We show the duration of `Stress` and `Velocity` macro-kernels as a function of the iteration numbers for two ranks only: rank 0 in red, and rank 63 in blue. We can see that the rescaling is imperfect but is sufficient to capture the magnitude of the computational cost of the original run. Although we only show here the rescaling for two ranks (in the corners of the domain decomposition), similar results are obtained with other ranks, no matter their location in the domain decomposition.

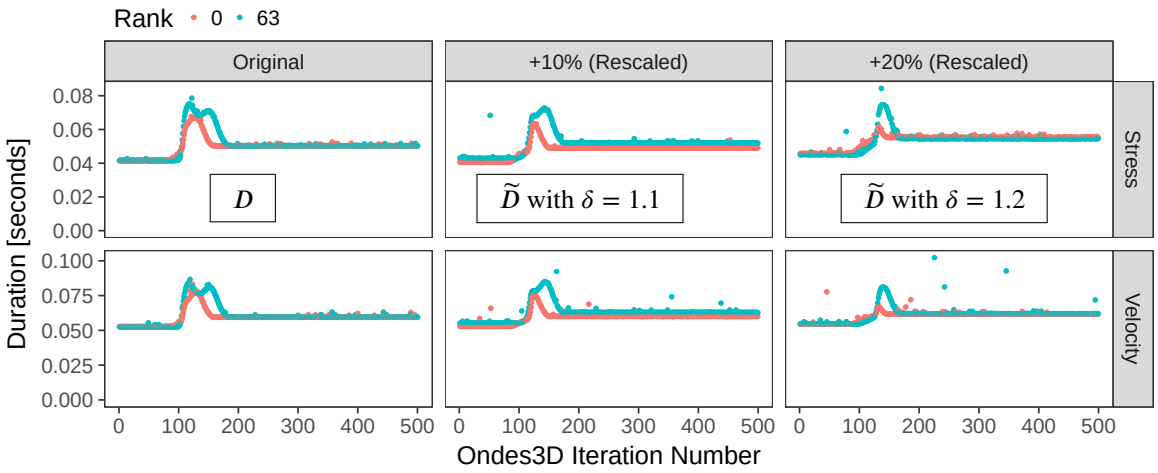


FIGURE 8 The duration (on the Y axis) of two macro-kernels (`Stress` and `Velocity`) in two ranks (colors): 0 in red, 63 in blue, as a function of the Ondes3D iteration number (on X). The data shown in the left column is collected with the original spatial/temporal resolution; while the points shown in the center and right columns are obtained by rescaling measurements taken with lower (by 10% and 20%) spatial/temporal resolution.

4.3 | Low-cost Workflow to Study Ondes3D-like Applications

The performance modeling strategies detailed in the previous section include spatial aggregation (Figure 5) and application scaling (Figure 8). Aggregation enables the downscaling of the number of MPI processes used to execute Ondes3D, while application scaling enables a faster execution/emulation of the original code. Together, they can be used to create (e.g., using a simple R script) a computational load *profile* for each rank, kernel and iteration of the Ondes3D application taking as input only one execution trace of the application. Our upgraded workflow stands on these performance modeling strategies and trace manipulation to enable a faster exploration of over-decomposition parameters. Figure 9 presents our three-step workflow to study Ondes3D, based on the SAMPI simulator previously described. Although we evaluate our workflow with Ondes3D, we believe that it is generic enough to apply to any imbalanced MPI code that is iterative and relies on static domain decomposition.

(Step 1: Application Characterization) The execution of the original MPI application is traced while configured to use a given workload, a given application scaling factor, and the largest number of processes to be studied (e.g., 256 in the Figure 9). The scaling factor is used to change the resolution of the temporal/spatial domain resolution of that workload, towards a coarser grain definition. As previously described in Section 4.2, for Ondes3D, this procedure makes the execution much faster. This execution can be done through SMPI emulation or in a real cluster, recording the number of cycles with PAPI instead of real timings as done in (24). The trace must also contain events to mark the beginning and end of each main loop iteration, and the computational cost (in micro-seconds or cycles) of the three Ondes3D macro kernels interleaved with the communication pattern (source, destination, payload) of the application. A very efficient computer program written in R is responsible to rescale back the trace to the original temporal/spatial resolution for the given workload, as if we had executed the program with the original resolution. Note that the resulting trace is representative of a given input scenario (earthquake) but not linked to a given platform anymore.

(Step 2: Application behavior extrapolation) The second step of the workflow aims at extrapolating the previous fine grain trace to any coarser decomposition level (i.e., using fewer processes). As shown in Section 4.1, we can spatially aggregate computational costs without information loss. As consequence, based on a fine-grain decomposition, one can define (e.g., using a simple R script) the computational load for each rank, kernel and iteration of the Ondes3D application for any other coarser

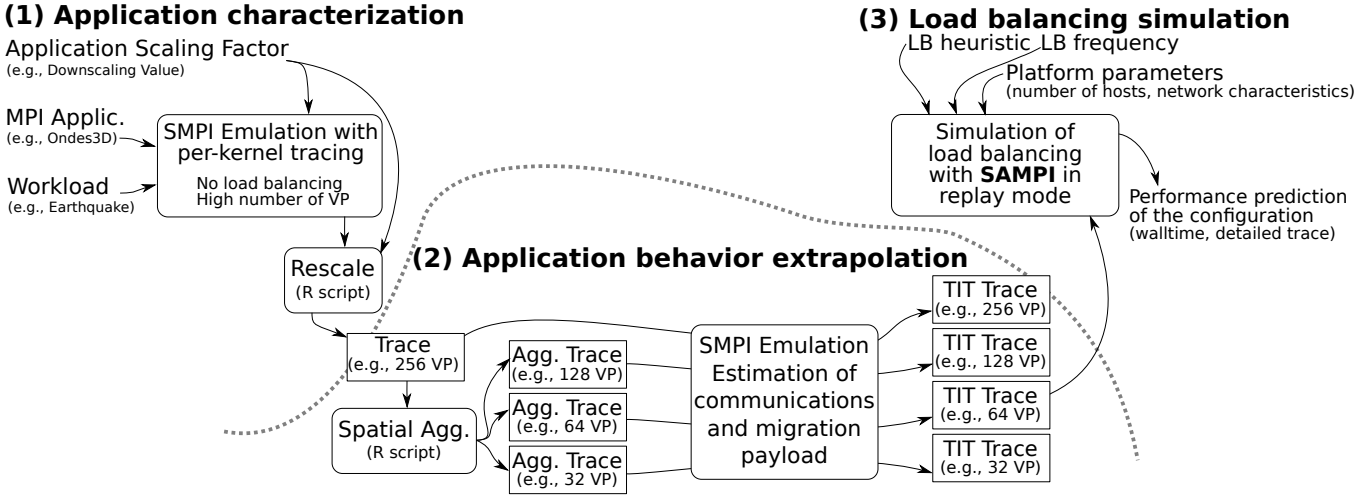


FIGURE 9 Performance evaluation workflow. The irregular structure of the load of the application over space and time needs to be carefully and finely captured in step 1. It can then be spatially aggregated in step 2 to study different levels of over decomposition. The final step 3 allows to quickly study the influence of load balancing parameters.

decomposition level. In the example on Figure 9, the computational behavior of the 256-process trace is spatially aggregated to 128, 64, and 32 processes. As result of this procedure, we obtain traces that contain only the aggregation of the computational cost of each macro-kernel, for each iteration and rank. SMPI handles all the MPI process management, and its scheduling algorithm does not perform context switches while the active process is performing computation. Thus, by aggregating only the macro-kernel computations, we aim to avoid capturing the extra context-switch and process management costs from the fine-grain execution. Aggregating communication patterns can be more complicated, although techniques such as the ones in (25) could be used. A simpler approach, involving a minor modification of Ondes3D, consists in using SMPI in hybrid emulation/trace replay mode: the code is emulated again, as in the first step, but with fewer VPs and every computationally intensive part (i.e., macro-kernels) is skipped and replaced by the corresponding spatially aggregated computational profile read from the coarse grain profile. To do so, we have manually modified the Ondes3D code to inject the corresponding load in the simulator instead of actually running the CPU-bound code. As the computation cost outside the macro-kernels tends to be more dependent on the number of MPI ranks, it needs to be measured again at the coarser grain. Therefore, this second emulation also serves to re-execute all the code that is located outside the aggregated parts. As a result, we obtain a complete *time independent trace* (TIT) with both computations and communications for a reduced number of VPs. If building on the spatial aggregation property is not possible, then a trace for every envisioned level of decomposition should be obtained directly as in the first step. We have shown in Section 2 that this is unnecessary for Ondes3D, since spatial aggregation faithfully captures its real behavior.

(Step 3: Load balancing simulation) Finally, the last step consists in running SAMPI simulation as previously described, but now using `TIT` files that have been spatially aggregated and rescaled from one single emulation or real execution.

5 | RESULTS AND EVALUATION

Several issues should be solved to correctly validate the accuracy of predictions obtained in simulation. Solely comparing the (predicted) makespan of simulations with the one of real-life executions on a few examples may be insufficient to be fully trusted. Another approach, much richer in temporal and spatial dimensions, is to compare detailed execution traces of an application as Ondes3D. For example, at first we tried to perform our analysis based on a space/time visualization, such as the one shown in Figure 10, which depicts two executions of Ondes3D with AMPI using 16 hosts with an over-decomposition level of 4 (for a total of 64 VPs). On each row, the behavior of each of the hosts (the vertical axis) is depicted along time (the horizontal axis) using different states (colors). In the top row, the view depicts the case when no load-balancing activities are carried out, while in the bottom row it depicts the computational behavior when GreedyLB is being used. Unfortunately, we are only able to collect AMPI traces at the runtime system level, instead of at application level. Additionally they generate a large number of states (up to 36 in our tests). So, to facilitate visualization, we decided to aggregate them into five types: *CentralLB* states are related to

the load balancing heuristic, *idle* represents when the process is not performing computation, *Ck* and *TCharm* are related to Charm++, *ampi* to internal AMPI states, and *MPI* represents MPI calls done by the runtime system. Another consideration is that the representation of the duration of states in Figure 10 is inaccurate, since gaps between occurrences of the same state tend to disappear at this zoom level. Even if some details can be empirically derived from such views through user interaction i.e., some colors (representing states) appear more often in one view than in the other, the analyst has no idea of the load imbalance evolution along time. In the end, we had to abandon this approach, since load balancing metrics are particularly hard or even simply impossible to observe using such space/time views.

Other ad hoc intermediate and aggregated representations are thus needed. In our context, application iterations and load imbalance among resources are of primary importance. Therefore, we decided to track the resource usage per processor and per iteration and to study its evolution both temporally and spatially. We calculate such metric by slicing the time by significative periods, such as the application iterations, and then calculating, for each slice of time and process, the amount of time spent in computation and in idle/communication states. Using exclusively the time in computation, and extracting the duration of each iteration, we can calculate how long each process spent computing in each iteration. If a global per-iteration metric is needed, we calculate the ratio between the sum of the actual computation of all resources and the total amount of resources available. In this section, we use this performance metric to compare reality and simulation both qualitatively and quantitatively.

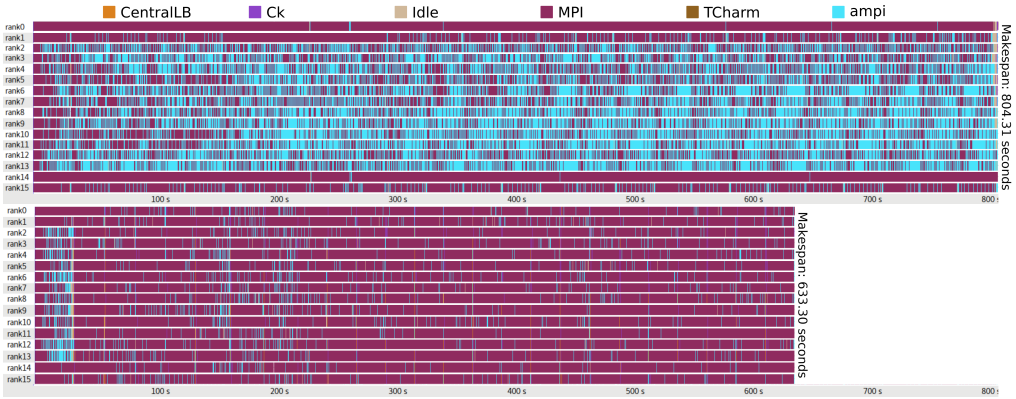


FIGURE 10 Illustration of the traditional space-time visualization, depicting the behavior of the Charm++/AMPI middleware when Ondes3D is executed using 16 hosts and 64 VPs (4-level over-decomposition). Colors represent different states of each MPI rank. The top row depicts an execution with no load balancer, while the bottom row represents the behavior with GreedyLB.

Section 5.1 presents details about the testbed and our validation methodology. The two Ondes3D workloads are also briefly detailed. Section 5.2 presents our validation effort to verify that simulations results from SAMPI are realistic when compared with emulations with SMPI. Section 5.3 presents a case-study where SAMPI is used to tune the load-balancing parameters for a given platform. Section 5.4 presents a capacity planning study with SAMPI/Ondes3D to define the best number of hosts necessary for a given workload. Finally, in Section 5.5 we bring a discussion about our workflow performance upgrades.

5.1 | Testbed, Validation Methodology and Ondes3D Workload

All experiments have been conducted in a controlled environment on 16 nodes of the Parapluie cluster, which is part of the Grid'5000 platform (26). Each node has two 12-core 1.7GHz AMD Opteron 6164 HE processors and is interconnected through a 20G Infiniband 4x QDR network. Ondes3D was compiled with GCC 4.8.4 with optimization flags `-O2 -march=native`, and the AMPI tests use Charm++ 6.6.1.

The behavior of the AMPI version of Ondes3D, representing the reality baseline we are comparing to, is registered using Tau (27), integrated in Charm++'s Tau tracemode. We configured AMPI to use the MPI layer for all communication operations and manually instrumented Ondes3D to track the timestamps of each iteration of the main loop, using the Tau API. Time-independent traces for our trace replay, according to our SAMPI workflow, have been obtained through sequential emulation using SMPI. All emulations have been conducted in one node of the cluster to obtain faithful computational load profiles.

We tested two very different earthquake scenarios in Ondes3D. The first one is the Mw 6.6 Niigata *Chuetsu-Oki* (2007) from Japan (19). Running the full simulation (6000 time steps) takes an unreasonable amount of time, especially because several

executions are needed to obtain statistically significant results. So, in order to keep a reasonable duration for the experiments, we limited this simulation to the first 500 time steps. For the same reason, we adopt a $300 \times 300 \times 150$ cell decomposition instead of the original one. The second simulated scenario is the Mw 6.3 *Ligurian* (1887) from north-west Italy (16). It was limited to 300 time steps and the number of cells to $500 \times 350 \times 130$. In both cases, the duration of the simulations match with the Earthquake strong-motion records.

5.2 | Validation: Comparing SAMPI (simulation) against AMPI

In our validation experiments, we fix the domain decomposition to 64 VPs (always mapped to 16 processes), and call `MPI_Migrate` every 20 time steps. From our experience, this configuration is relatively good and allows to focus our evaluation on sound scenarios. Figure 11 depicts the comparison of the computational load of simulation outputs with real AMPI traces for situations without load balancer, with GreedyLB and with RefineLB, for the two workloads: Chuetsu-Oki (left) and Ligurian (right).

5.2.1 | Per-process Computational Load Analysis (Heatmaps)

Heatmaps in Figures 11a (Chuetsu-Oki) and 11b (Ligurian) show the computational load (as a color gradient) for each core (in the vertical axis) along the Ondes3D iterations. A reddish color represents higher computational load, while blue represents idleness. Each heatmap corresponds to an execution, either real (AMPI in the top row) or simulated (SAMPI in the bottom row), with a given load balancer (no load balancing on the left column, GreedyLB in the center, and RefineLB on the right column). In all configurations, the real and simulated load distribution are very similar, which shows the ability of our workflow to capture the complex behavior of AMPI.

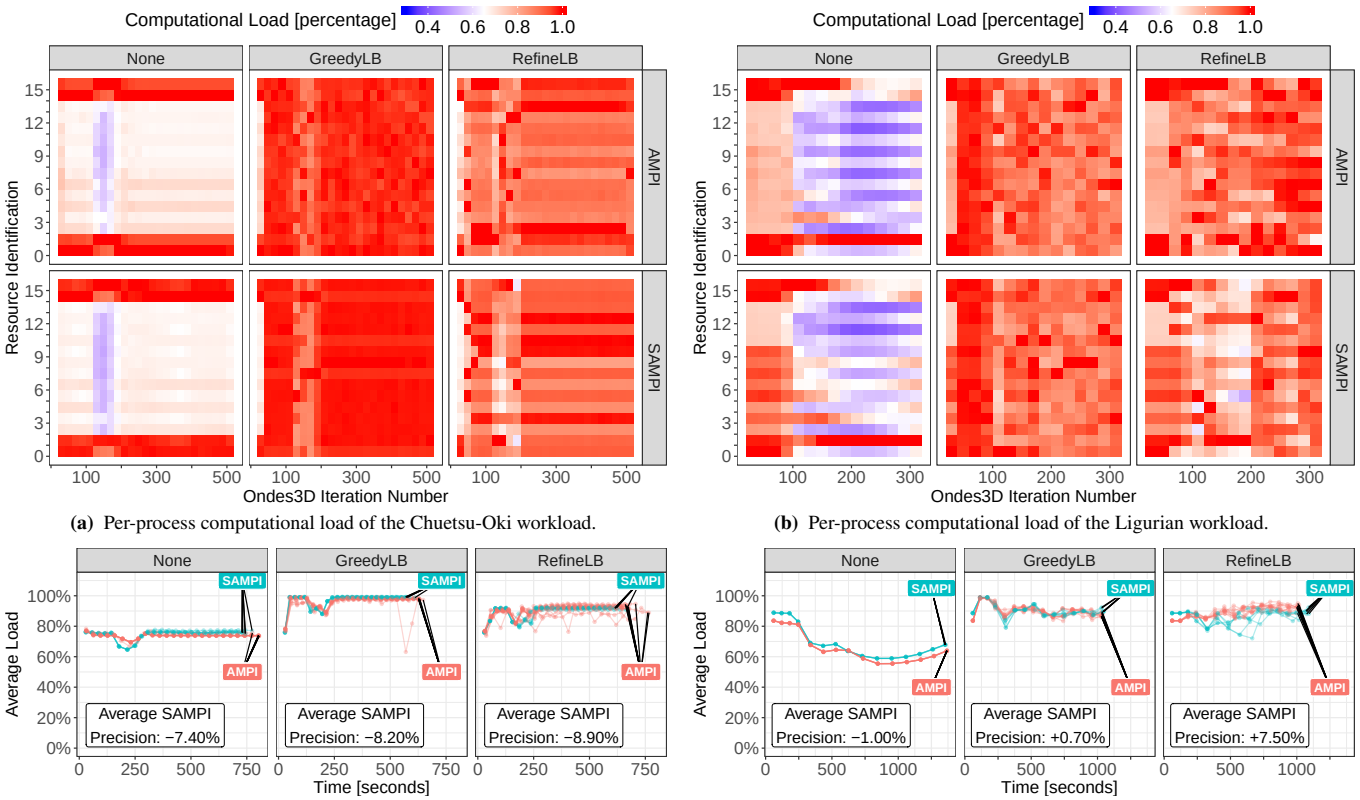
Figure 11a shows that for Chuetsu-Oki, the case without load balancing leads to many underutilized resources (white and bluish regions). Both LB seem to significantly improve this situation by making processes 2 to 13 receive more load. It is interesting to note that GreedyLB achieves a much better load balancing than RefineLB (being more conservative) and that this is visible in simulation as in real execution traces. The load structure for the Ligurian workload is quite different (see Figure 11b). There seems to exist an alternating load irregularity in processes whose ranks belong to the center of the domain decomposition (those with white and bluish colors without load balancing). GreedyLB and RefineLB are again effective to redistribute the load, since we observe a much more even computational load across processes although not as good as the one obtained for the Chuetsu-Oki workload.

The heatmap views are based on one run for each case (hence 12 runs considering in total both workloads). It should be noted that any new execution (either real or in simulation from a new trace) will lead to slightly different outcome and that focusing on the load of a given core at a given time step is thus not really meaningful. From such view, it seems that GreedyLB is the best choice from the load balancing point of view, but communication (both from the application and from load balancer) should also be taken into account. In the following, we provide makespan results using the average load as a function of the execution time.

5.2.2 | Average Load and Makespan Comparison Analysis

The plots in Figures 11c (Chuetsu-Oki) and 11d (Ligurian) depict the evolution of the average load across the cores participating in the execution. The average load (in vertical axis) is drawn as a function of time (horizontal) for both SAMPI (blue) and AMPI (red). The points along the lines indicate the moments where the metric is calculated (in the beginning of the `MPI_Migrate` operation, at the end of the load balancing interval); lines connect measurements of a given run and show the trend. As before, the horizontal facets in each figure indicate the computed metric without load balancing, with the GreedyLB and RefineLB load balancers. The *Average SAMPI Precision*, i.e., the difference between the average makespans for SAMPI and AMPI, is displayed in a box in the bottom of each chart. These charts enable richer analysis, depicting many runs on a single chart (at least five).

For the Chuetsu-Oki workload (see Figure 11c), we can confirm that GreedyLB performs better than RefineLB, both in simulation as in real life. One could expect GreedyLB to be worse than RefineLB, due to the large amount of migrations it performs. In this case, however, it seems that the default overload tolerance of 1.05 used by RefineLB is too high. Regarding the comparison of SAMPI against the real AMPI, one can note that across several runs, SAMPI is slightly too optimistic. That being said, such inaccuracy would not affect our choice of load balancer. It is interesting to see quite significant variability in the real executions (perfect isolation is tough to achieve on a cluster; although we reserved the whole parapluie cluster to collect the real measurements, some of the network switches are shared by several clusters), being generally larger than in the simulations.



(a) Per-process computational load of the Chuetsu-Oki workload.

(b) Per-process computational load of the Ligurian workload.

(c) Average Load and makespan using the Chuetsu-Oki workload.

(d) Average Load and makespan using the Ligurian workload.

FIGURE 11 Comparison of SAMPI executions (obtained in simulation) against AMPI runs (reality) for two representative workloads: Chuetsu-Oki (left) and Ligurian (right); for each of them, the top row shows six heatmaps (without load balancing, GreedyLB, and RefineLB) illustrating the computation load (color gradient) as a function of the Ondes3D iteration for all 16 processes; the bottom row shows the average aggregated load along time, highlighting the makespan of multiple runs.

Our simulations are trace-based, hence the variability of simulations necessarily comes from the tracing in the first step of the workflow. Although we used a dedicated node, operating system noise cannot be avoided and variability at tracing time translates into variability when simulating load balancing. For the Ligurian workload (see Figure 11d), as on the previous scenario, both simulation and real life have similar load unfolding. Again, there is a slight makespan disparity with SAMPI timings being pessimistic. Yet, the trends remain correct, and RefineLB achieves the best performance in both simulation and real life.

5.2.3 | Summary of the Validation Procedure

Our simulation is able to mimic in a realistic way the evolution of the load distribution of real life executions, which is one of the main aspects we are trying to analyze. There is still some inaccuracy (a few percent) in terms of absolute time prediction and we are currently investigating the source of this discrepancy. Yet, since the trends remain correct, this does not not affect the identification of the optimal load balancer in the two investigated scenarios. In the next section, we demonstrate how the SAMPI simulator can be used to explore different load balancing parameters.

5.3 | Tuning Load-balancing Parameters with Simulation

In this subsection, we present experiments to demonstrate the usefulness of our simulation approach, through the investigation of the parameter space of AMPI. The three parameters previously discussed are evaluated: load balancing frequency, heuristic and over-decomposition level. For these experiments, we decided to focus on the Ligurian scenario, since it is larger and more complex than the Chuetsu-Oki. Therefore, tuning in simulation is more likely to be useful for it. We use the previous workflow to study the makespan of Ondes3D with six load balancing intervals (every 1, 5, 10, 20, 30, and 40 iterations), two heuristics (GreedyLB and RefineLB), and six scenarios of over-decomposition (without it, and then with factors equal to 2, 3, 4, 8 and 16).

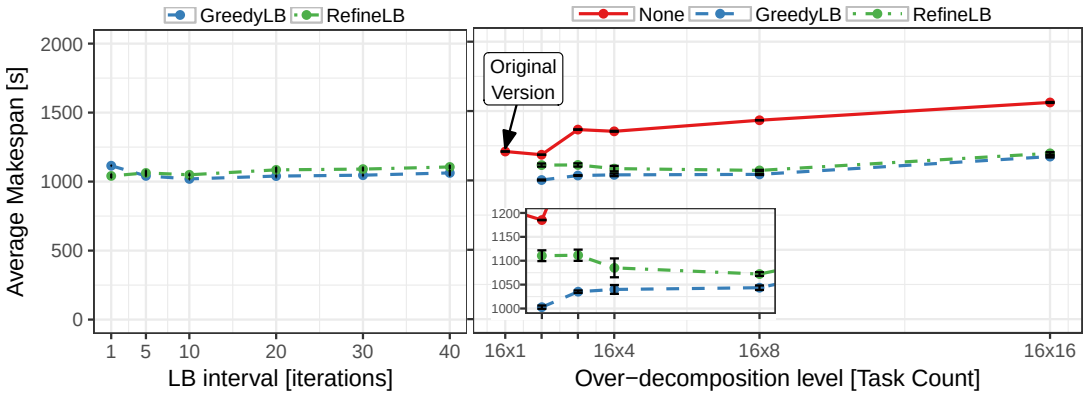


FIGURE 12 Simulated makespan predictions for the Ligurian earthquake simulation with (left) six load balancing intervals (in number of iterations) and (right) with five over-decomposition levels (2, 3, 4, 8, and 16) on 16 cores.

Investigating the influence of the load balancing frequency: a call to `MPI_Migrate` is present for each VP at the end of every time step in the traces obtained at the end of Step 2. During the simulation with SAMPI, we control and enforce a different load balancing frequency by actually calling the barrier and the load balancing at the configured load balancing interval for that particular experiment. Intuitively, the more frequent the calls, the better the load balancing but also the more important the barrier and data migration overhead. The left plot of Figure 12 shows the influence of the load balancing frequency (horizontal axis) on the makespan (vertical axis) of a 16×4 over-decomposition configuration. In this setting, it turns out that LB frequency has no or little influence in the performance attained when using GreedyLB or RefineLB. Even though GreedyLB balances the load carelessly whereas RefineLB is much more conservative, the communication performance of the system is sufficiently good to hide the migration costs.

Investigating the influence of the decomposition level: Another important performance affecting parameter is the over-decomposition level. The results on how over-decomposition affects the makespan of Ondes3D, when calling `MPI_Migrate` every 20 time steps, are in the right plot of Figure 12. The average makespan (vertical axis) is shown as a function of five over-decomposition configurations (horizontal) plus the absence of over-decomposition (e.g., 16×1). In the absence of load balancing (None), over-decomposing is, as expected, deterring since this creates extra-communication between VPs. Yet having more and smaller VPs allows for a better redistribution of the load. This trade-off was explained in Section 3. The RefineLB sweet spot is reached with a 16×8 decomposition ($\approx$13% gain over the original version). However, for GreedyLB the decomposition level should be as small as possible (which leads to $\approx$19% gain over the original version), which is explained by the fact that its careless migrations scale very badly. In the end, the 16×2 GreedyLB configuration is slightly better than the 16×4 RefineLB configuration but exhibits quite different load balancing behaviors.

Investigating usage of the execution time with different load balancers: To illustrate the reason behind the different behaviors of GreedyLB and RefineLB, we computed the fractions of time each resource spends on computation, MPI communication, load balancing heuristic, and migrations. At a given period of time, different MPI ranks on the same resource may be in different states (e.g., one rank may be computing while another is waiting for a message). So, there is an overlap of different states in the same resource. In this case, we define the state the resource following a priority order: (1) *Computing*, (2) *LB Heuristic*, (3) *Migration*, and (4) *MPI*. The highest priority is given to *Computing* to show how well it overlaps communications. As consequence, we may loose some detail on the communication behavior. This analysis was done for the 16×2 and 16×8 over-decomposition levels, which respectively resulted in the fastest executions for GreedyLB and RefineLB. The results are displayed on Figure 13, with the case without load balancing (left) to be used as reference. With a 16×2 over-decomposition, GreedyLB presents a shorter communication time than RefineLB, as well as a slightly better distribution of computation time. Besides, the GreedyLB heuristic is faster than RefineLB. All of this contributes to the better performance achieved by GreedyLB with this configuration. The migration costs are negligible in both cases. With a 16×8 over-decomposition, the results for both heuristics indicate an increase in the overall computation time, a reduction in non-overlapped communication time, and better distribution of computation time among resources. For RefineLB, the aforementioned reduction in communication time was enough to reduce its total makespan, but not to surpass GreedyLB. This indicates that RefineLB needs a higher over-decomposition level than GreedyLB to be able to satisfactorily rebalance the load. We can also see in Figure 13 that with the 16×8 over-decomposition the migration times for GreedyLB start to become significant. This could become a problem if we continued to increase the number of MPI ranks.

As an aside, from this figure one can infer, based on the area occupied in the charts, that the total amount of computation increases when we increase the over-decomposition level. That is to be expected, since here we are including computation that happens outside the three main computation kernels (Velocity, Stress, and Intermediates), which indeed becomes more complex as we need to manage more processes. This is exactly why, as explained on Section 4.3, our spatial aggregation process only aggregates the computation that happens inside the aforementioned kernels.

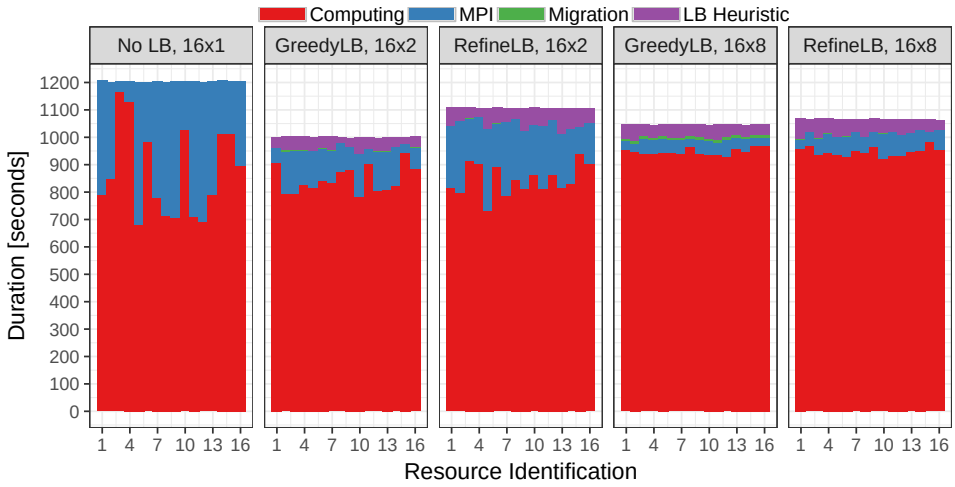


FIGURE 13 Fractions of the makespan spent on computation, communication, migration, and load balancing heuristic. This illustrates the performance behavior of GreedyLB and RefineLB, compared to the case without load balancing (No LB).

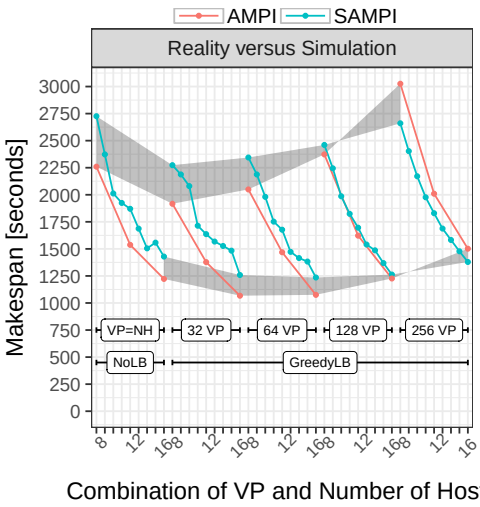
5.4 | Capacity Planning with SAMPI

The use of SAMPI in *trace-replay* mode brings several advantages. SAMPI can be used to determine the number of hosts that respect a certain efficiency requirement from a single representative `TIT` file with the behavior of 256 VPs, obtained by emulation in a single core of the same machine we intend to study. In this section, we present simulations for 128, 64, and 32 VPs, as well as for the *NoLB* simulations (no over-decomposition), that were done with traces obtained using the spatial aggregation technique explained in Section 4.1. As shown in Figure 11d, the difference between real executions and simulations (without spatial aggregation) for *NoLB* or for GreedyLB on the Ligurian case with 16 hosts and 64 VPs was within less than 1%. Yet, as can be noted in Figure 5, although the load evolution obtained through a spatial aggregation is quite faithful, it is slightly overestimated. Using spatial aggregation is therefore expected to be harmless when comparing the relative performance of load balancing heuristics but it may be problematic when trying to obtain absolute makespan predictions.

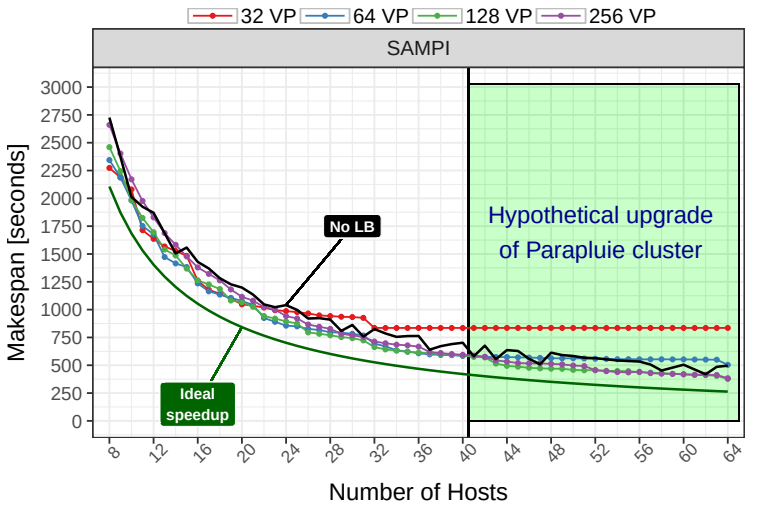
To evaluate the potential bias, we use the Parapluie cluster of Grid'5000 and compare reality with simulation on the Ligurian scenario, since it is larger and more complex than the Chuetsu-Oki one. In Figure 14a, we evaluate how spatial aggregation influences our makespan predictions. Duration (on the Y axis, in seconds) is represented as a function of the number of available hosts (on the X axis) and for five decomposition levels. When there is no over decomposition (*NoLB*, left curves), the simulation (SAMPI, top blue line) is very pessimistic compared to real executions (AMPI, bottom pink lines), especially for a small number of hosts. This can be explained by the fact that the loads obtained through spatial aggregation are all slightly overestimated and that all these differences add up. This bias (top and bottom shaded areas) decreases when the decomposition level becomes closer to the one used for doing the aggregation (256 VPs). Surprisingly, the bias is inverted for 256 VPs but this can be imputed to the process management overhead of AMPI when managing large overdecomposition levels (256 VPs in total on 8 hosts means that hosts are over-subscribed by a factor 32), which we do not model in SAMPI. For larger host counts (bottom shaded area, 16 hosts), the bias of spatial aggregation is significantly reduced as we get closer to a reasonable oversubscription.

One hypothesis that could explain this bias in aggregation is that we may be capturing the increased process management and context switches overhead that comes from executing the application with a larger number of processes. We believe this is an interesting hypothesis, which we intend to further investigate in our ongoing effort to improve our simulation workflow.

With such potential bias in mind, we can now study capacity planning and try to identify how the number of hosts impacts the overall execution time. In Figure 14b, we show the makespan predicted by SAMPI when it replays the computational behavior of the Ligurian case with 256, 128, 64 and 32 VPs, employing GreedyLB every 20 iterations. These makespans (on the Y



(a) We compare SAMPI against reality (AMPI), evaluating the bias caused by spatial aggregation. The bias is limited when there is little spatial aggregation.



(b) Capacity planning using SAMPI *trace-replay* and with 256, 128, 64, and 32 VPs; showing the resulting makespan as a function of the number of hosts (8 to 64). The ideal makespan (with a perfect speedup) is derived from the makespan with only one VP (not shown).

FIGURE 14 Evaluating with the Ligurian case and GreedyLB (a) the bias of the simulation, and (b) the scalability of Ondes3D. SAMPI predictions are based on the same 256 VPs `TIT` trace. Traces for other VP counts are generated using spatial aggregation. *NoLB* runs are not over-decomposed.

axis, in seconds) are represented as a function of the number of hosts (on X) available from a calibrated SimGrid platform file that represents the Parapluie cluster of Grid'5000. First, the general shape of these curves is interesting since their irregularity cannot be the result of platform variability but is something inherent to the load profile. Indeed, our simulations are perfectly deterministic and the exact same load distribution is used throughout all simulations. It is visible that the performance of the GreedyLB scenarios when increasing the number of hosts evolve much more smoothly than the performance of "No LB", which is a good property. Second, the results indicate that performance gains are good when moving from 8 to 24 hosts, where the makespan is reduced from 2,273 seconds to 856 seconds (≈62% less execution time). The addition of 16 extra hosts to get the total capacity of Parapluie (40 hosts) reduces the makespan to ≈26% of the original case with 8 hosts. This is only possible because of the load imbalance mitigation brought by GreedyLB and VP migrations.

We have artificially increased the number of hosts to 64 (24 more hosts than the real cluster) to check whether this upgrade of Parapluie would be interesting from the Ondes3D Ligurian workload point of view. Figure 14b shows that going from 40 hosts (the real capacity of Parapluie) to 64 hosts (the new hypothetical capacity) would reduce the execution time another 207 seconds. Although the gain is of much smaller extent than when going from 8 hosts to 16 hosts, these few minutes may be precious and worth the investment when conducting hazard assessment after a earthquake. Our simulations show that the use of load-balancing with AMPI allows to obtain a nearly perfect speed-up (dark-green curve). With such simulation we can forecast that at least 37 hosts should be allocated to obtain the earthquake simulation results in no more than 10 minutes. Obviously the uncertainty of the simulation should be taken into account but as we have seen, for large node counts, the oversubscription is reasonable and the impact of spatial aggregation is limited.

5.5 | Workflow Performance

As previously stated, the workflow must be fast to rapidly find the best load balancing parameters. Table 1 presents performance estimations of the workflow when it is run in a modern desktop, using exclusively emulation for the first step and two workloads: Ligurian and Chuetsu-Oki. There are two types of workflow: the `Upgraded` version contains the changes brought by acceleration strategies presented in Section 2 and has all the three steps; the `Raw` version has only the first and last steps, since the spatial aggregation is optional. For the estimations presented in such table, we assume that the goal is to define the best load balancing parameters considering one platform, four over-decomposition levels (256, 128, 64, and 32 processes), two heuristics (GreedyLB and RefineLB), and three load balancing frequencies (every 10, 20, and 30 iterations), for a total of 24 configurations. The estimations for the third step (SAMPI) are for a single simulation through trace replay.

TABLE 1 Estimated per-step workflow performance, in seconds.

Workload	Workflow	Step #1	Step #2	Step #3	Total
Ligurian	`Raw`	$4 \times \approx$21,000	`NA`	$24 \times \approx$200	≈88,800
	`Upgraded`	$1 \times \approx$10,000	≈3,650	$24 \times \approx$200	≈18,450
Chuetsu-Oki	`Raw`	$4 \times \approx$9,500	`NA`	$24 \times \approx$200	≈42,800
	`Upgraded`	$1 \times \approx$4,000	≈1,400	$24 \times \approx$200	≈10,200

The results on Table 1 indicate that the `Upgraded` version of the workflow is faster than the `Raw`. The first step in `Raw` is costlier because it involves the trace collection in all four decomposition levels in SMPI emulation mode. This could be made much faster if traces are gathered in a real cluster, but with the cost of resource reservation and access. Moreover, the `Upgraded` version has application scaling with a factor of 20% (see Section 4.2), only requiring the collection of a single very fine-grain trace (with the maximum number of ranks we want to study, in this example, 256 processes) plus the minor rescaling overhead. We estimate that with a 20% application scaling factor, we are capable to reduce the emulation time by ≈50% of the original emulation without scaling. Despite the necessity of Step #2, its cost is minor because of the hybrid emulation/trace-replay approach we adopted to downscale the communication pattern after spatially aggregating the computational load (see Section 4.1). At this step, the Ligurian case is much slower than Chuetsu-Oki (not shown) because Ligurian has a geological model in its input files. So, the time for its hybrid emulation comprises also the very expensive IO operations during model instantiation. The computational load injection cost is minor in this step, for both cases. Finally, the SAMPI simulations of Step #3 consist in replaying the TIT traces (from Step #1 in `Raw`, or from Step #2 in `Upgraded`) for all 24 parameter combinations. The cost of these simulations is stable for both approaches, with minor changes between the two workloads. For this evaluation, they are considered to be executed sequentially in a single computer.

To summarize, Table 1 shows that our Ondes3D modeling strategies are capable to improve the workflow performance from ≈25 to ≈5 hours, an improvement of 80% using the `Upgraded` version considering the Ligurian Workload. For Chuetsu-Oki, the gains are of 75%, from ≈12 to ≈3 hours. Additionally, since the simulations on Step #3 are all independent, the cost of this step could be further reduced by running all of the simulations in parallel. As a comparison, the time needed to conduct a similar study using solely real experiments and investigate all 24 configurations would require the whole cluster (32 nodes) for ≈7 hours for Ligurian and ≈5 hours for Chuetsu-Oki.

6 | RELATED WORK

The simulation workflow we propose mostly depends on two factors. First, a faithful model of modern HPC networks and MPI implementations is essential since communications play a crucial role in the load balancing trade-offs. Second, the ability to run simulations both in trace-replay and emulation modes is more than helpful to select the approach most suited to the resources at hand. There is a plethora of simulation frameworks and tools to study MPI applications (8) and at least four of them support both emulation and trace-replay and could thus have been modified: BigSim (28), SST-Macro (29), Xsim (30), and SimGrid (8) (through SMPI (9)). BigSim is part of Charm++, thus supporting the simulation of AMPI applications, such as our Ondes3D implementation. Although it is linked to Charm++, it does not allow to change the load balancing parameters during trace replay and this feature would require major modifications to the BigSim simulator. SST-Macro (31) allows both trace replay through the DUMPI module and on-line simulation (emulation) through skeletonization. Although SST-macro is quite flexible, and has many network models, including flow-based ones, its emulation support requires to modify Ondes3D, which we wanted to avoid. Finally, Xsim mostly focuses on extreme-scale executions. Its sources are not currently available and it is unclear whether it would be able to emulate unmodified full-fledged MPI applications.

For this work, we therefore chose to rely on the free software SimGrid, whose SMPI (9) interface allows both emulation and trace replay of MPI applications. SMPI leverages SimGrid's thoroughly validated flow communication models (11), while also accounting for specific characteristics of MPI implementations (8). Hence, SMPI allows us to collect accurate execution traces from emulation, and its replay mechanism allows us to quickly simulate this execution as many times as needed. Finally, traces are in pure text, enabling an easy transformation through spatial aggregation.

7 | CONCLUSION

We propose a simulation based approach for the performance evaluation and tuning of dynamic load balancing applied to iterative MPI applications. Our approach allows the estimation of performance gains from load balancing at low cost, both in terms of time and of resource requirements. Although we apply it to a geophysics application (Ondes3D), its structure is very typical among legacy MPI applications. Therefore, we believe the usefulness of our approach is not limited to Ondes3D. Our contributions are four-fold: (a) An in-depth analysis of the spatial and temporal load balancing issues found in Ondes3D. The latter demonstrates how dynamic load imbalance can arise even when there is no visible source of temporal variability in the code. (b) Performance modeling strategies to reduce the time needed to characterize the application. We capture traces using coarse-grain decomposition and rescale the computational behavior back to the original fine-grain decomposition level. (c) Implementation and validation of a simulator, called *SAMPI*, that uses SimGrid to simulate over-decomposition and load balancing, mimicking the behavior of AMPI. This simulator extends the MPI trace replay functionality included in SimGrid, to enable the fast exploration of different load balancing scenarios from a single execution trace. (d) A fast performance evaluation workflow that uses a hybrid approach, which combines sequential emulation with time-independent trace replay to evaluate load balancing scenarios at a small fraction of the cost of real executions.

As future work, we plan to investigate other Ondes3D workloads to understand how the computational load evolves, possibly enabling us to extrapolate the behavior for higher process counts and to better guide and trigger the load balancing. Using SAMPI, we also plan to study how load-balancing strategies can be combined with history-based dynamic voltage scaling (DVS) strategies à la ADAGIO (32) to further reduce energy consumption, in particular when running on heterogeneous platforms. Finally, as a side effect of our work, although out of the scope of our initial goals, the knowledge gathered in this work is employed to improve the performance of the Ondes3D macro-kernels.

ACKNOWLEDGMENTS

We thank CAPES/Cofecub 764-13, FAPERGS/Inria ExaSE, FAPERGS 16/2551-0000 354-8 (PPP 2014), FAPERGS 16/2551-0000 488-9 (Green Cloud), CNPq 447311/2014-0, CNRS/LICIA Intl. Lab, the EU H2020 Programme and the MCTI/RNP-Brazil under the HPC4E Project, grant 689772. Some experiments were carried out at the Grid'5000 platform (https://www.grid5000.fr), with support from Inria, CNRS, RENATER and several other organizations.

References

[1] Dupros Fabrice, Martin Florent De, Foerster Evelyne, Komatitsch Dimitri, Roman Jean. High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media. *Parallel Comput.* 2010;36(5):308 - 325.

[2] Dupros Fabrice, Do Hiep-Thuan, Aochi Hideo. On Scalability Issues of the Elastodynamics Equations on Multicore Platforms. *Procedia Comput Sci.* 2013;18:1226 - 1234. 2013 International Conference on Computational Science.

[3] Martinez Victor, Michea David, Dupros Fabrice, et al. Towards Seismic Wave Modeling on Heterogeneous Many-Core Architectures Using Task-Based Runtime System. In: Borin Edson, Prasanna Viktor K., eds. *Proceedings of the 2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, ; 2015; Washington, DC, USA.

[4] Kale Laxmikant V., Krishnan Sanjeev. CHARM++: A Portable Concurrent Object Oriented System Based on C++. *SIGPLAN Not.* 1993;28(10):91–108.

[5] Huang Chao, Lawlor Orion, Kalé L. V.. Adaptive MPI. In: Rauchwerger Lawrence, ed. *Languages and Compilers for Parallel Computing*, :306–322Springer Berlin Heidelberg; 2004; Berlin, Heidelberg.

[6] Huang Chao, Zheng Gengbin, Kalé Laxmikant, Kumar Sameer. Performance Evaluation of Adaptive MPI. In: ACM , ed. *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, :12–21; 2006; New York, NY, USA.

[7] Tesser Rafael Keller, Pilla Laércio Lima, Dupros Fabrice, Navaux Philippe Olivier Alexandre, Méhaut Jean-François, Mendes Celso. Improving the Performance of Seismic Wave Simulations with Dynamic Load Balancing. In: Aldinucci Marco, D'Agostino Daniele, Kilpatrick Peter, eds. *Proc. of the 2014 22nd Euromicro Intl. Conf. on Parallel, Distributed, and Network-Based Processing*, PDP '14:196–203; 2014; Washington, DC, USA.

[8] Casanova Henri, Giersch Arnaud, Legrand Arnaud, Quinson Martin, Suter Frédéric. Versatile, scalable, and accurate simulation of distributed applications and platforms. *J Parallel Distrib Comput.* 2014;74(10):2899 - 2917.

[9] Clauss Pierre-Nicolas, Stillwell Mark, Genaud Stéphane, Suter Frédéric, Casanova Henri, Quinson Martin. Single node on-line simulation of MPI applications with SMPI. In: IEEE Computer Society , ed. *Parallel & Distributed Processing Symposium (IPDPS), IEEE Intl.*, :664–675; 2011.

[10] Desprez Frederic, Markomanolis George S., Suter Frederic. Improving the Accuracy and Efficiency of Time-Independent Trace Replay. In: IEEE Computer Society , ed. *Proc. of the SC Companion: High Perf. Comp., Networking Storage and Analysis*, SCC '12:446–455; 2012; Washington, DC, USA.

[11] Velho Pedro, Schnorr Lucas Mello, Casanova Henri, Legrand Arnaud. On the Validity of Flow-level TCP Network Models for Grid and Cloud Simulations. *ACM Trans Model Comput Simul.* 2013;23(4):23:1–23:26.

[12] Stanisic Luka, Thibault Samuel, Legrand Arnaud, Videau Brice, Méhaut Jean-François. Faithful performance prediction of a dynamic task-based runtime system for heterogeneous multi-core architectures. *Concurr Comput.* 2015;27(16):4075–4090.

[13] Tesser Rafael Keller, Schnorr Lucas Mello, Legrand Arnaud, Dupros Fabrice, Navaux Philippe Olivier Alexandre. Using Simulation to Evaluate and Tune the Performance of Dynamic Load Balancing of an Over-decomposed Geophysics Application. In: Rivera Fancisco F., Pena Tomás F., Cabaleiro José C., eds. *Internatioal European Conference on Parallel and Distributed Computing (Euro-Par)*, :192–205; 2017.

[14] Stanisic Luka, Legrand Arnaud, Danjean Vincent. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *SIGOPS Oper Syst Rev.* 2015;49(1):61–70.

[15] Moczo Peter, Robertsson Johan O.A., Eisner Leo. The Finite-Difference Time-Domain Method for Modeling of Seismic Wave Propagation. In: Wu Ru-Shan, Maupin Valerie, Dmowska Renata, eds. *Advances in Wave Prop. in Heterog. Earth*, Adv. in Geophysics, vol. 48: Elsev. 2007 (pp. 421 - 516).

[16] Aochi Hideo, Ducellier Ariane, Dupros Fabrice, Terrier Monique, Lambert Jérôme. Investigation of historical earthquake by seismic wave propagation simulation: Source parameters of the 1887 M6. 3 Ligurian, north-western Italy, earthquake. In: L'Association Française du Génie Parasismique (AFPS) , ed. *8ème colloque AFPS, Vers une maitrise durable du risque sismique*, ; 2011.

[17] Mucci Philip J., Browne Shirley, Deane Christine, Ho George. PAPI: A Portable Interface to Hardware Performance Counters. In: Post Douglass E., ed. *In Proceedings of the Department of Defense HPCMP Users Group Conference*, :7–10; 1999.

[18] Lastovetsky A., Szustak L., Wyrzykowski R.. Model-Based Optimization of EULAG Kernel on Intel Xeon Phi Through Load Imbalancing. *IEEE Trans Parallel Distrib Syst.* 2017;28(3):787-797.

[19] Aochi Hideo, Ducellier Ariane, Dupros Fabrice, et al. Finite Difference Simulations of Seismic Wave Propagation for the 2007 Mw 6.6 Niigata-ken Chuetsu-Oki Earthquake: Validity of Models and Reliable Input Ground Motion in the Near-Field. *Pure Appl Geophys.* 2013;170(1-2):43-64.

[20] Pilla Laércio L, Ribeiro Christiane P, Coucheney Pierre, et al. A topology-aware load balancing algorithm for clustered hierarchical multi-core machines. *Future Gener Comput Syst.* 2014;30:191–201.

[21] Pilla Laércio L.. Topology-Aware Load Balancing for Performance Portability over Parallel High Performance Systems. PhD thesis UFRGS; Université de Grenoble 2014.

[22] Pérache Marc, Carribault Patrick, Jourdren Hervé. MPC-MPI: An MPI Implementation Reducing the Overall Memory Consumption. In: Ropo Matti, Westerholm Jan, Dongarra Jack, eds. *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 16th European PVM/MPI Users' Group Meeting Proceedings*, :94–103; 2009.

[23] Bédaride Paul, Degomme Augustin, Genaud Stéphane, et al. Toward better simulation of MPI applications on Ethernet/TCP networks. In: Jarvis Stephen A., Wright Steven A., Hammond Simon D., eds. *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, :158–181; 2013.

[24] Casanova Henri, Desprez Frédéric, Markomanolis George S, Suter Frédéric. Simulation of MPI applications with time-independent traces. *Concurr Comput.* 2015;27(5).

[25] Noeth Michael, Ratn Prasun, Mueller Frank, Schulz Martin, Supinski Bronis R.. ScalaTrace: Scalable compression and replay of communication traces for high-performance computing. *J Parallel Distrib Comput.* 2009;69(8):696 - 710.

[26] Balouek Daniel, Carpen Amarie Alexandra, Charrier Ghislain, et al. Adding Virtualization Capabilities to the Grid'5000 Testbed. In: Ivanov Ivan I., Sinderen Marten, Leymann Frank, Shan Tony, eds. *International Conference on Cloud Computing and Services Science (CLOSER)*, Communications in Computer and Information Science, vol. 367: :3-20; 2013.

[27] Shende Sameer S., Malony Allen D.. The Tau Parallel Performance System. *Int J High Perform Comput Appl.* 2006;20(2):287–311.

[28] Zheng G., Kakulapati Gunavardhan, Kale L. V.. BigSim: a parallel simulator for performance prediction of extremely large parallel machines. In: IEEE Computer Society , ed. *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, ; 2004.

[29] Adalsteinsson Helgi, Cranford Scott, Evensky David A., et al. A Simulator for Large-Scale Parallel Computer Architectures. *Int J Distrib Syst Technol.* 2010;1(2):57–73.

[30] Engelmann Christian. Scaling to a million cores and beyond: Using light-weight simulation to understand the challenges ahead on the road to exascale. *Future Gener Comput Syst.* 2014;30:59 - 65.

[31] Rodrigues Arun F, Hemmert K Scott, Barrett Brian W, et al. The structural simulation toolkit. *SIGMETRICS Perform Eval Rev.* 2011;38(4).

[32] Rountree Barry, Lownenthal David K., Supinski Bronis R., Schulz Martin, Freeh Vincent W., Bletsch Tyler. Adagio: Making DVS Practical for Complex HPC Applications. In: ICS '09:460–469ACM; 2009; New York, NY, USA.

How cite this article: R. Keller Tesser, L. Mello Schnorr, A. Legrand, F. C. Heinrich, F. Dupros, and P. O. A. Navaux (2017), Performance Modeling of a Geophysics Application to Accelerate the Tuning of Over-decomposition Parameters through Simulation, *Concurrency and Computation: Practice and Experience*, *TBD*.