SABO: Dynamic MPI+OpenMP Resource Balancer

Cassandra Rocha Barbosa, Pierre Lemarinier, Guillaume Papauré, Marc Pérache, Michaël Krajecki

SABO: Dynamic MPI+OpenMP Resource Balancer

Cassandra Rocha Barbosa
Atos, Echirolles, France
LICIIS, LRC DIGIT,
University of Reims Champagne Ardennes, Reims, France
cassandra.rocha-barbosa@univ-reims.fr
cassandra.rochabarbosa@atos.net

Pierre Lemarinier
Atos
Echirolles, France
pierre.lemarinier@atos.net

Guillaume Papauré
Atos
Echirolles, France
guillaume.papaure@atos.net

Marc Pérache
CEA, DAM, DIF and CEA, DAM, DIF, LRC DIGIT F-91297 Arpajon, France
Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, Bruyères le Châtel, France
marc.perache@cea.fr

Michaël Krajecki
LICIIS, LRC DIGIT,
University of Reims Champagne Ardennes
Reims, France
michael.krajecki@univ-reims.fr

***Abstract*—Scientific parallel applications often use MPI for inter-node communications and OpenMP for intra-node orchestration. Parallel applications such as particle transport, seismic wave propagation simulator, or Finite-Element applications often exhibit workload imbalance due to their nature of ongoing data movement. These applications usually develop software balancing strategies triggered when some imbalance thresholds are detected to reduce this imbalance. These developments are complex to implement and impact the entire distributed applications' performance by synchronizing and exchanging the load over the network. This paper proposes a method to dynamically detect load imbalance and balance the computation by redistributing OpenMP threads between MPI processes local to the node. With minimal impact on the applications' codes, we demonstrate how this technique can improve the overall applications' performance up to 28% on MiniFE, 17% on Quicksilver, and 3% on Ondes3D. We also present its impact when executing multiple nodes and our proposed approach's limitations.**

***Index Terms*—HPC , MPI+OpenMP, Load Balancing.**

I. Introduction

In the pursuit of higher performance, processors nowadays can exceed one hundred cores. Utilizing all these resources in an HPC platform along with its complex memory hierarchy can be challenging for scientific applications. To harness the performance of such platforms, applications such as physics simulations rely on new programming models. It usually consists of a mix of MPI [1] for communication between nodes and processes, and a dedicated runtime for orchestrating the intra-node or intra-process computation. While different task based runtimes get traction for intra-node, the main environment remains OpenMP [2].

While using such a combination of MPI and OpenMP allows to leverage the computing capabilities of the different cores of the machine, possibly leading to relying on multiple multi-threaded MPI processes per nodes, balancing the load between the different processes can be challenging. In particular, some scientific applications exhibit workloads that dynamically evolve, leading to load imbalance during their execution.

Different types of such load imbalance can be distinguished. For instance, an imbalance can happen inside a process between the different OpenMP threads as they execute a different computation workload. Another type of imbalance may also happen between processes, even if their respective OpenMP regions are balanced. This is the case for particles transport where the simulation depicts moving objects in a mesh. The main computation will happen initially in part of the mesh, but will move dynamically during the execution depending on the ongoing computation of the simulation: when balancing the workload at the start of the execution for the initial conditions on the different processes, the computation will evolve and lead to a later imbalance between processes. As these simulations usually translate into alternate phases of computation and state exchange using synchronization and communication, this imbalance is characterized by having some processes with more work to do while others wait in MPI communications. This situation creates under-utilization of the resources allocated to the application. In this paper, we propose to focus on this type of imbalance.

The standard solution to address this imbalance consists in application specific redistribution of the data and its workload, which can be quite challenging to implement. In this paper, we propose to address this imbalance issue by dynamically redistributing the underlying computation resources while minimizing the impact on application source code. To achieve this goal, we introduce a technique to detect load imbalance and a method to redistribute the number of OpenMP threads per process. Our proposed methods rely on the OpenMP tools interface (OMPT) [3] to interact with the underlying runtime and measure the time spent by each thread on the computation in each OpenMP section.

For applications for which each processes exhibit the same number of OpenMP sections per computation phase, no code modification are required to benefit from our proposed solution. However for applications where two processes may have a different number of OpenMP sections per computation

phase, we introduce a single function with no parameter that the application developer shall use to mark the end of a computation phase. This enables us to estimate for this type of applications the workload of an entire computation phase for balancing. We then leverage this function to evaluate each process' workload and decide on the new number of OpenMP threads to dedicate dynamically to each process.

We implemented the different callbacks to OMPT and the proposed function inside our library called SABO. Using this library, we demonstrate the impact on performance on two mini-apps: Quicksilver and MiniFE with respectively up to 17% and 28% improvement on execution time. We also demonstrate the benefit of our approach on a seismic simulation called Ondes3D with an improvement of up to 3%. While our proposed technique does not address load imbalance between nodes, we think it can efficiently complement the classical application-level data balancing by reducing the frequency at which it is triggered.

In the rest of the paper, we expand on motivation for SABO in section II followed by related work in section III, before detailing our proposed method for balancing in section IV and its performance evaluation in section V.

II. Motivation

This section presents some context of parallel scientific applications on supercomputers motivating our solution.

Scientific computing applications often use one of the following two main types of parallelism or mix them:

MPI/domain decomposition parallelism: Domain decomposition [4]–[6] is well known and efficient. It can be used with one or multiple compute nodes. Nevertheless, an application using only MPI often suffers from a higher memory consumption due to ghost cells required by the domain decomposition approach. Ghost cells are a copy of cells owned by neighbor processes and which are synchronized regularly to keep data coherency. This overhead in memory consumption is directly linked to the size of the stencil and the number of dimensions.

OpenMP/Thread parallelism: This parallelism is used within a node. It is easier to set up compared to MPI decomposition parallelism. Nevertheless, it is often harder to get full performance due to NUMA effects or memory contention.

Most of the time the trade-off is a mix of these two programming models. Usually, to deal with NUMA effects and contention, applications use more than one MPI process per node. We did an evaluation of this trade-off in Table I. One can see that for the Quicksilver application for 1 or 2 nodes having each 48 cores, using 8 MPI processes with 6 OpenMP threads per node is better than using only one MPI process with 48 OpenMP threads. This is also the case for Ondes3D with 4 nodes.

Even if MPI + Threads is an efficient way to parallelize applications, with possibly multiple MPI ranks per node, the static decomposition implied by the domain decomposition parallelism may still lead to unbalanced executions. To dynamically balance such executions, it is first essential to have a tool that detect imbalance during the execution of the application.

Application	Number of nodes	Exec time for X processes per node			
		X=1	X=4	X=8	X=16
Quicksilver	1	948,81	296,77	278,95	291,51
Quicksilver	2	573,76	181,34	192,98	182,64
Ondes3D	4	2258	648.75	558,46	439.02

TABLE I
Execution time (in seconds) of Quicksilver and Ondes3D with different number of processes per node.

This tool must be less intrusive not to alter applications' behavior when measuring imbalance. Moreover it should introduce minimal to no impact to applications' codes in order to favor its utilization. It is also necessary to have a method for correcting the detected imbalance that is both accurate and quick to perform, not increasing the application's execution time. In this paper, we propose a library implementing methods for detecting imbalance and improve load balancing for this kind of application with minimal to no code modifications.

III. Related work

This section details a list of methods that reduce workload imbalance in an application in the execution context introduced in the previous section. We will also briefly introduce the key differences between existing methods and SABO.

Dynamic resource allocation of cores to processes is a well-known problem and long-studied issue. This problem is often delegated to the operating system (OS) where Several scheduling policies have been studied and implemented. Most OS scheduling policies do not rely on users' hints or interventions, nor applications' dynamic workload information. In the context of High-Performance Computing, dynamic changes of resource allocation to an application done by the OS scheduler at any given time may significantly impact its performance due to context switches and cache degradation [7]. This is why the classical approach in HPC is to assign resources to each application statically for its entire execution duration. While static resource allocation leads to better performance overall for most HPC applications, it may also lead to resource waste. For example, when computing threads allocated to computing resources are not progressing, these idle resources will not be switched with other applications' threads.

With the rise of many-core machines, the standard programming model for HPC applications has evolved from MPI to MPI+X. In this context, the number of cores assigned to each process is also statically allocated so that no two processes compute over the same resource, and it is up to the shared memory runtime to manage the load on each computing resources to maximize their utilization. Usually, the number of threads to use within this shared memory runtime is set depending on the underlying allocated resources.

As discussed in the previous section, it is common to use multiple MPI+X processes on each node. Nevertheless, some applications have an unbalanced workload by cell/MPI subdomain. This load imbalance can come from border domain conditions that can introduce more compute work to add reflection or absorption conditions. One other source of imbalance is present in particle transport simulation, where

the compute cost of a particle varies according to the type of material it interacts with.

For example, in a Monte Carlo particles transport, a solution to design the simulation is to split the computing domain into subdomains dispatched to multiple processes to get a better execution time [8]. These types of applications may also exhibit load imbalance during their execution. In this context, several approaches exist for dynamic load balancing. These methods can take place at different levels of the application stack. The first approach involves implementing dynamic load balancing directly in the application source code. This is currently the preferred method used in imbalanced HPC applications [8], [9]. However, this method puts the burden on the shoulder of the application developer, who knows her application and it increases the complexity of the application's code. Some solutions in particles transport applications use static data partitioning obtained from tools such as scotch [10] or metis [11]. Using these techniques requires defining a cost model to provide input to the partitioning tool and choosing between geometric and non-geometric partitioning. With geometric partitioning, each subdomain has a regular border and a simple communication scheme, but data decomposition is hard to perform due to geometric constraints. Using non-geometric partitioning allows reaching well-balanced decomposition but implies a complex communication scheme and irregular subdomain border. Partitioning is also coarse grain: even if we can have multiple calls to the partitioning tool during execution, we cannot perform fine-grain load balancing such as loop load balancing. While these methods have their merits, in this paper, we propose a different approach with minimal modifications to the applications for providing automatic load balancing within a node.

Non Monte-Carlo simulations may also exhibit load imbalance. For example [12] presents a framework solution for Adaptive Mesh Refinement. This solution balances the load by migrating sections in a mesh between MPI ranks using MPI communications. This solution, based on domain decomposition, can be used for different types of simulations such as Tsunami Wave Propagation or Two-Phase Porous Media Flow. These data-balancing solutions necessitate heavy applications' codes modifications.

Another possibility for introducing load balancing is to orchestrate it directly in an application's runtime. In the case of particles transport simulation, it can consist of dynamic data partitioning in the multi-threads context. Different techniques have been proposed, depending on the considered runtime. For instance, the OpenMP runtime [13] enables the use of tasks and introduces the notion of dynamic task scheduling. Other runtimes [14] rely on a task-based approach to propose better load balancing techniques. With such runtimes, the task orchestration and data load balancing are quite transparent to the user. However, it can be challenging to modify/design an application's code to fit this task programming model. Moreover, this approach is often limited to performing load balancing separately within each MPI process.

Work-stealing techniques can be used in the context of MPI + OpenMP applications. In [15] and [16] authors propose two work-stealing solutions for MPI + OpenMP applications. The first proposes a push approach where works are pushed to other threads, and the second describes a pull approach. In both cases, work-stealing is performed between OpenMP threads, and a communication thread is required. Because of this communication thread, applications can only use n-1 threads for computation for each MPI process. Both of these solutions use data balancing. Compared to these efforts, we propose balancing the resources instead of the data.

All previous methods are some form of data load balancing. One method called LeWi [17], [18] uses a resource load balancing. This method is totally transparent to the user and works with a hybrid MPI + OpenMP model. With this method, when a process *A* is blocked in a MPI communication like MPI_Send, the LeWi algorithm reassigns the cores of this process *A* to another process *B* in the same node to improve computing capacity and reduce the waiting time of process *A*. When process *B* finishes its computation, all cores are given back to the process *A*. Compared to this approach, we propose evaluating the imbalance dynamically and assigning OpenMP threads to the different processes accordingly.

DROM [19] is an interface for enabling malleability. The proposed method connects a resource manager such as Slurm to an application. It allows this resource manager to request the dynamic change of the number of threads allocated to an application. Compared to this work, we propose to compute the load imbalance dynamically local to each node and to automatically compute new OpenMP threads distribution for each process without the coordination from the resource manager.

IV. Contribution

We will detail in this section the approach we propose to increase the performance of applications by balancing dynamically the underlying resources compared to each process load. We first introduce our proposed mechanism for detecting imbalance and then describe our balancing technique.

A. OpenMP Tools Interface (OMPT) and SABO library

To both instrument the detection of imbalance, and assign a new number of threads and their mapping to cores during the execution of applications, we propose to rely on the OpenMP Tools Interface (OMPT) which allows providing callbacks to some OpenMP internal functions.

With respect to the detection of imbalance using OMPT, we insert a callback at the beginning of an OpenMP section and at the synchronization barrier of each OpenMP threads to compute each thread's time spent inside an OpenMP section and accumulate these time measurements among multiple sections. It is then possible to sum these measurements on each process to get an estimation of its workload. When considering computation phases of MPI + OpenMP scientific applications, two possibilities arise. First possibility: every process enters the same number of OpenMP sections in each phase. In this case we can simply rely on estimating workload of every process at any given same number of

OpenMP section entered. Counting the number of section entered is performed within the OMPT callbacks and there is no need for modifying the targeted application code in order to estimate the workload. Second possibility: at least two processes can exhibit a different number of OpenMP sections per computation phase. In this case we cannot rely on counting OpenMP sections to perform a process' workload estimation relevant to computation phases. Instead we introduce a new function that we call omp_balanced(). This function does not take any parameter. It is then up to the application's developer as she is the one knowing the structure of her application to mark the end of a computation phase by calling our proposed function. Such call to omp_balanced() enables us to sum the workload relevant to a computation phase for each process locally to a node.

To simplify, for the first case where computation phases include the same number of OpenMP sections on every process, we simply call omp_balanced() directly within the OMPT callback triggered at the end of an OpenMP zone. This enables us to implement the exchange of information and possible resulting balancing steps inside omp_balanced() for both cases.

Once we obtain a workload estimation for each process within omp_balanced(), we exchange this information between every process located on the same node through MPI appropriate communicator and MPI_Allgather collective operation. This operation happens on each node separately. Information exchange is local on each node of the application independently of each other and is not performed between the nodes.

B. Determining a new resource distribution

Once load information is shared between the processes local to the same node, they can independently compute the respective load of each process compared to the sum of all processes load on this node. From this information, they can deduce their respective share of the total of available cores on this node that they will request and thus the corresponding new number of threads of each local process. Due to NUMA effect, we take the policy to not let a process span over multiple sockets, which would hinder its performance. Thus the last step consists in finding a proper mapping of these threads to the node, potentially adjusting the resulting number of threads per process to reflect this NUMA policy, and reallocating resources accordingly.

By deciding a new number of threads per process based on their workload, it is possible not to find any combination of processes to map on each socket. We propose here a method for mapping processes per socket and adjusting the number of threads per process by using a research graph. This step is important because a process may change of the socket from the previous state.

This method is based on the well known branch and bound algorithm [20]. It relies on an exploration tree with an associated decision function to stop the exploration. We map processes in decreasing order of workload and explore the tree with a depth-first search approach. A state of a node of the tree corresponds to the list of processes associated with each socket along with the number of available free cores remaining from this current mapping. Initially, we map one process per socket, resulting in the state of the root of the exploration tree. Let us consider a node n of this tree. We first apply the decision function which counts the number of cores c that are missing to fulfill the proposed mapping. This function keeps the lowest value of such a number min_c obtained so far once every process has been mapped, thus at a previously seen leaf of the tree, and is set initially to infinity. If $c >= min_c$, we do not continue to explore this branch of the tree as a better mapping has been found elsewhere. If $c < min_c$, we generate a child n' that corresponds to the mapping of the next process to the first socket with a positive number of available cores. Once all processes have been mapped, we update the value of min_c if necessary. This algorithm creates a tree with a depth of *number of processes per node - the number of sockets per node.* An example of this exploration tree is provided in Fig.1 with 4 processes started with 12 threads each. The processes 0,1,2 and 3 request respectively 14, 9, 13, 12 threads. In this example, the node has 2 sockets with 24 cores each. In this example we explore 4 nodes and the root. In the end, the solution with processes 0 and 1 on socket 1 and processes 2 and 3 on socket 2 is the best solution with $min_c = 1$ (arrow 4).

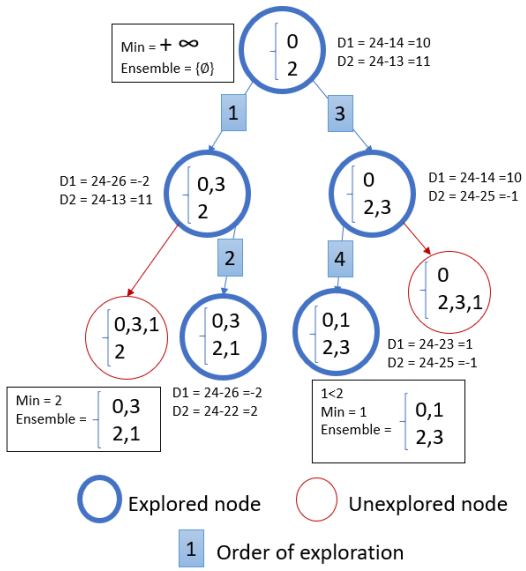

Fig. 1. Example of Exploration Tree.
Dx represents the number of free cores available on socket x.

Once obtained a number of threads and their mapping, the resource allocation is changed by resetting OMP_NUM_THREADS and KMP_AFFINITY. Note that as this environment variables changes happen between OpenMP sections and not within, it is consistent with the OpenMP standard.

C. Using SABO library on balanced applications

We want our proposed library to have no impact on applications that are balanced or which exhibit limited imbalance. In

the case of limited imbalance, the resulting number of threads for each process computed by our method can be equal to their previous number of threads. In this case, we simply ignore the mapping resulting from the computation of the research graph and do not change the mapping of these processes, in order not to introduce cache misses and context switches induced by a different thread placement.

It is also possible that the limited load imbalance leads our method to compute a different number of threads per process by adding or removing one thread on some processes. If the load imbalance is limited, it is possible that removing a thread from one process to increase another one degrades the overall application's performance. To address this issue, we introduce a verification step after computing the tree decision, in which we forecast the possible impact of the proposed change in the number of threads per process. This forecast is computed using the following method: for each process we know its last number of thread $lastNb_{th}$, its last execution time $time$ and how many threads $newNb_{th}$ they would get with the proposed balancing decision. We can predict the future process execution time as $time_P = \frac{time * lastNb_{th}}{newNb_{th}}$. Let max_{time} be the highest execution time of all processes in the last step. If $\exists P \in Ens_P, time_P >= max_{time}$, with Ens_P is the ensemble of all processes of a node, then we decide to not balance the application's resources. Addressing these issues also enables using our proposed SABO library even with limited knowledge of applications' load imbalance as it prevents affecting already balanced applications.

D. Other considerations on omp_balanced() calls

omp_balanced() is the function where both workload information is exchanged and balancing is performed. We implemented the possibility of executing omp_balanced() only every X time this function is called, or only once at the Xth call. This leads to reducing the number of synchronization injected in the application and conversely reduces the number of opportunities for balancing the applications' resources. Selecting the value of X is currently the responsibility of the user using an environment variable. In principle such a selection could be automated within our proposed solution. It is not trivial and has impact on the considered application performance, and is thus out of the scope of this paper.

Finally, note that when omp_balanced() is inserted at the end of a computation phase by a developer, the introduction of synchronization has limited impact as usually the end of a computation phase already includes synchronization and exchange step on its own. The supplementary synchronization introduced with this MPI_Allgather should have a minimal impact on application performance in that case.

V. Experiments

We detail in this section the performance evaluation of the SABO library. We first describe our experimental conditions and two benchmarks we used, Quicksilver and MiniFE, before detailing our results using the SABO library explained in the previous section. For simplification, we present in this section the results of the executions done with omp_balanced function call placed in the application.

A. Experimental setup

All the experiments were performed on our platform named Pise located in Echirolles, France. It consists of 32 nodes connected by InfiniBand Mellanox ConnectX-6 HDR @200Gb/s. Each node is composed of a dual-socket, 24 cores each, AMD EPYC 7402 CPU @2.8GHz, and 128GB of memory.

The installed environment is Red Hat Enterprise Linux Server release 8.2, with a Linux kernel version 4.18.0. We used Open MPI 4.0.4 with the PML ucx. The Intel environment we used is composed of the compiler 2020 Update 4.

B. Experimental benchmarks

1) Quicksilver: Quicksilver [21] is a proxy application, from the CORAL benchmark suites, which implements a simplified dynamic Monte Carlo particles transport problem of the Mercury workload. Quicksilver includes an optional module for balancing data on demand during the computation.

2) MiniFE: MiniFE [22] Finite Element mini-application, also from the CORAL benchmark suites, which implements some kernels representative of implicit finite element applications. MiniFE, compared to Quicksilver, has a parameter to generate load imbalance between MPI processes during the execution. This parameter simulates the imbalance in real application with multiple materials and/or physics.

C. Methodology

For each benchmark, four sets of experiments were done. The first set consists in analyzing the impact on the performance of the library on a single node for different number of processes, different values of imbalance for MiniFE, and balancing triggered in different applications' steps. Note that changing the number of processes also affects the initial number of threads per process. For these first experiments, the problems' sizes do not change. These experiments exhibit the impact of the library on the execution time of the applications. The second set of experiments consists in showing the impact on performance when the number of nodes increases and the problems' sizes do not change. This is done for different values of imbalance for MiniFE.The value used in [23] is the largest value use in our expermientations. The third set of experiments consists in showing the impact on performance when the number of nodes increases and the problems' sizes change for MiniFE. The last set of experiments consists in showing the impact of a threads redistribution happening in the 2nd step, the best case for our solution as exhibited in the next section, on performance compared to the Quicksilver data balancing module.

When not specified otherwise, The threads redistribution happens in the 20th step for MiniFE as it corresponds to the best step following the study presented next section. In all our experiments, we dynamically balance the benchmark's execution only once, as these benchmarks exhibit a relatively small execution time. This is legitimate in the context of these

benchmarks but in a real execution, the execution time will be longer and several load-balancing will be necessary.

D. Single node analysis

The objective of these experiments is to show the impact on the performance of the library in one node. In these experiments, the size of each problem does not change. Different imbalance values for MiniFE are tested, and calling our balancing technique at different application's step values are evaluated for Quicksilver.

The step value chosen to call SABO is important. If the call of SABO is too late, the balancing will happen toward the end of the execution and the gain will be less important compared to doing the call of SABO after a little number of steps. Nevertheless using SABO after a little number of steps allows to have a better execution time in all cases with Quicksilver. This variation is visible in Fig. 2, which shows the benefit of using SABO on Quicksilver execution time. The time obtained without using SABO corresponds to the time reported in the first line of Table I. If the cell is yellow the gain is greater while if the cell is blue the gain is smaller or negative in one case. For example, in step 9 the gain is 1% whereas if the call is in step 2 the gain can be up to 12%. This variation is also visible in Fig. 3, which shows the gain for MiniFE with 1 node and 8 processes. In this example, the total number of steps is 200 and the best gain is obtained with balancing at the 20^{th} step of this application for different values of imbalance. Nevertheless using SABO with 8 processes allows to have a better execution time in all values of imbalance.

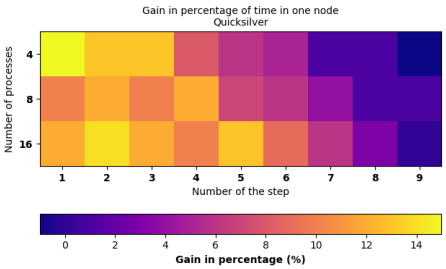

Fig. 2. Percentage of execution time reduction for three different numbers of processes when using SABO's balancing method at different application's step for Quicksilver

When the number of processes increases in the same node, the number of threads per process decreases, as the total amount of cores does not change and we do not want to oversubscribe cores. The Table II and Figure 2 show that the gain depends on the initial number of threads. The impact on performance is better when the number of threads is not too small for some application. When the initial number of threads per process is too small, like 3 threads per process such as presented in Table II with 16 processes, our proposed balancing technique can have a negative impact on performance. This is due to the architecture of the machine we are conducting the experiments on. It is composed of dual AMD 7402 processors of 24 cores each, with 8 x L3 shared caches, each L3 cache being shared by 3 cores. The initial execution condition of the

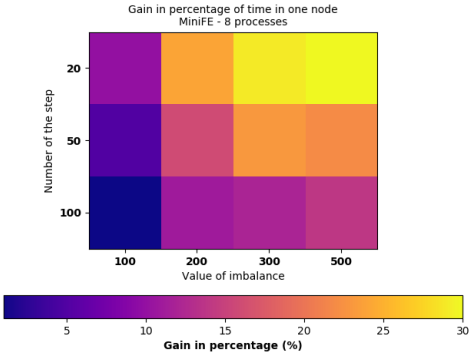

Fig. 3. Percentage of execution time reduction for different values of imbalance in MiniFE. Three MiniFE step values of the total 200 are evaluated for triggering our balancing technique

Number of processes	Number of threads	Value of imbalance		
		100	***200***	***500***
8	6	10%	24%	30%
16	3	-18%	-1%	2%

TABLE II
PERCENTAGE OF EXECUTION TIME REDUCTION ON ONE NODE WHEN USING SABO COMPARED TO EXECUTION WITHOUT FOR MINIFE

application makes it benefit fully from this cache architecture which is not the case anymore once we change the number of threads of some processes. We also measured the overhead of our library by comparing runs of MiniFE with 8 processes, 6 threads each, with a imbalance value of 0 with and without our library. We obtained an overhead of 0,26 seconds over the total execution time of 66,96 seconds, which is quite negligible.

E. Strong scaling

For these experiments, the number of processes per node is set to 8, 6 threads each, and the problem size is set to $400 \times 400 \times 400$ for MiniFE and 100 000 000 particles for Quicksilver. Selecting 8 processes per node results from having the best execution time in one node with 8 processes. Even if the gain for 16 processes with SABO is better in the percentage of execution time, it is not better compared to 8 processes with SABO. Fig. 4 shows that when the imbalance is small, the impact on application performance of our local-to-the-node balancing can be negative. Yet, when the imbalance is large enough, a gain is achieved. Our proposed technique does not address inter-node load balancing but achieves intra-node balancing. For distributed execution on multiple nodes, balancing the slowest nodes and decreasing their execution time will result in an overall performance improvement. The limit of our approach is that it will not improve the performance further once the slowest node exhibit a balanced workload between its processes, in which case a balancing strategies between nodes is required. We believe our method can still complement and reduce the number of such data balancing to perform at the application level between nodes.

F. Weak scaling

For these experiments, the number of processes per node is set to 8, 6 threads each. The problem size is defined to ensure that each process has the same initial data size for MiniFE,

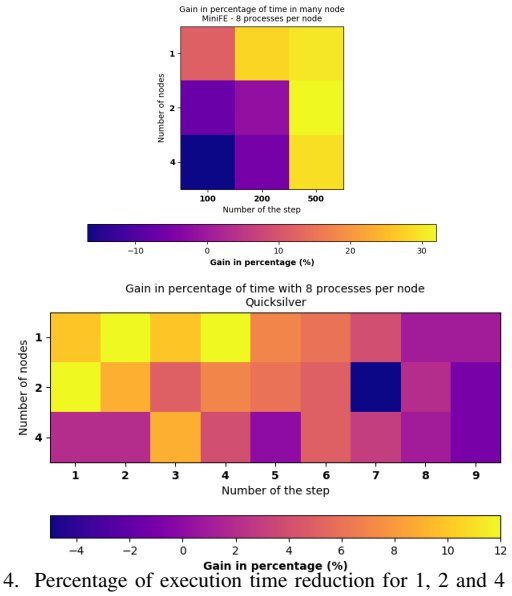

Fig. 4. Percentage of execution time reduction for 1, 2 and 4 nodes

with an initial size of 400^3 for 1 node. Table. III shows that SABO library has still an impact on distributed computation spanning multiple nodes.

Number of nodes	size	gain
1	400^3	30%
2	504^3	28%
4	635^3	21%
8	800^3	25%
16	1008^3	24%
20	1086^3	18%

TABLE III
COMPARISON OF EXECUTION TIME GAIN ON MINIFE WITH 8 PROCESSES PER NODE AND IMBALANCE=500

G. Comparison with load balancing algorithm integrated in Quicksilver

Some applications, such as Quicksilver, implement their own balancing algorithms. These algorithms are effective as they are tuned specifically for one application. Thus these methods provide a performance target for a generic method like SABO. Here we compare our method with Quicksilver's load balance algorithm, we performed runs with 8 processes per node with different numbers of nodes. To have a longer execution time and get closer to a real application, the number of steps launched was increased to 20 and the call of the SABO library was made at steps n=2 for our method. Results are summarized in Table IV. Although the executions made with the SABO library do not allow us to reach the level of performance of these algorithms, we can obtain decent performance results with up to 17% of performance gain where a dedicated algorithm reaches 29%. Yet, for applications that do not have such algorithms, our generic method can be used quite easily with at most its single function to be inserted in the application code.

VI. ONDES3D EXPERIMENTS

We detail in this section the performance evaluation of the SABO library in a real simulation application. We first describe the Ondes3D [24] application, before detailing our results using the SABO library explained in the previous section. All performance evaluation were conducted on the same experimental setup as described section V-A.

Method	Number of nodes		
	1	**2**	**4**
SABO step 2	17%	9%	12%
Balance module	29%	36%	42%

TABLE IV
COMPARISON OF EXECUTION TIME GAIN ON QUICKSILVER WITH 8 PROCESSES PER NODE

Application exec time (s)	Max time spent in OpenMP (s)	Mean time spent in OpenMP (s)	Max gain
558,46	511,56	399,51	20%

TABLE V
POSSIBLE GAIN WITH ONDES3D WITH 1000 STEPS

A. Ondes3D

Ondes3D [24], [25] is an application dedicated to earthquake modeling. It is a numerical finite-element simulation of seismic wave propagation.

B. Methodology

Ondes3D's code repository provides two test examples with their data. We chose to use the SISHUAN example. This test case is executed on 4 nodes with 8 processes per node. Each process has 6 threads at the beginning of the execution. For these experiments, we run 11 times each type of execution. The first run is discarded as a warm-up execution and we keep the last 10 executions. For all these experiments, Ondes3D performs 1 000 computation steps. We evaluate the performance impact of balancing resources once, at either step 100, 200 or 300 and report our finding in the following result section.

C. Results

Table V represents the maximum theoretical gain one can achieve by perfectly balancing the load of the application. If every process spends the same time in the OpenMP zone in average, this gain would be around 20%. Our library does not balance resources between different nodes and thus can not reach this target.

Figure 5 shows the execution time with and without using the SABO library for different values of steps. We exhibit the average time of 10 executions, the minimum execution time achieved and the maximum execution time of these 10 runs. Whichever is the selected step for balancing resources with SABO, the average execution time is better than not balancing at all.

We can have a gain with SABO library of 3.19% in the best case. Almost all of the executions have a small gain but we notice that using the SABO library does not increase the average execution time. Without the SABO library the performance variation is between -0.70% and 0.71% and with the SABO library it is between -0.62% and 3.19%. Based on these experiments, we can deduce that the use of the SABO library when the application is quite balanced does not degrade the average performance.

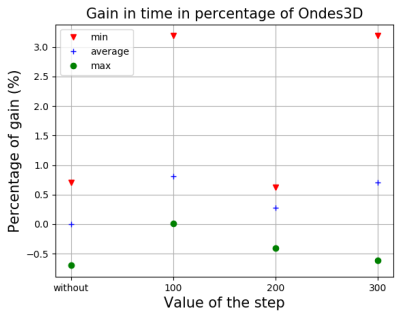

Fig. 5. Percentage of execution time of Ondes3D with 4 nodes and 8 processes per node

VII. Conclusion

We presented our balancing technique and its implementation into the SABO library. It relies on computing the load of every process' OpenMP thread on the same node and redistributes dynamically the number of threads accordingly. The proposed balancing happens on each node separately. The SABO library is easy to use and requires little knowledge of an application to understand if it needs, and where to insert, its single function compared to implementing a dynamic data redistribution in a scientific application. We evaluated on a single node and multiple nodes the impact on the performance of our proposed method using two different mini-applications representative of actual simulations and obtained performance improvement in both cases. Currently, our proposed technique does not address load balancing between nodes but could complement a data balancing implemented in the application. Future work aims at overcoming this limitation, for instance by making use of distributed shared memory environments. With the distributed shared memory, the cores of one node could be used for doing the work of a process in another node. This solution would enable load balancing strategies between processes located on different nodes.

Acknowledgment

The DEEP Projects have received funding from the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements n° 287530, 610476, 754304, and 955606. In the latter (DEEP-SEA), national contributions from the involved state members match the EuroHPC funding.

References

[1] "MPI: A message passing interface standard," http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf, June 2015.

[2] "OpenMP version 5.1," https://www.openmp.org/spec-html/5.1/openmp.html, november 2020.

[3] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT: An OpenMP tools application programming interface for performance analysis," in *International Workshop on OpenMP*. Springer, 2013, pp. 171–185.

[4] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio, "New challenges in dynamic load balancing," *Applied Numerical Mathematics*, vol. 52, no. 2-3, pp. 133–152, 2005.

[5] C. Rettinger and U. Rüde, "Dynamic load balancing techniques for particulate flow simulations," *Computation*, vol. 7, no. 1, p. 9, 2019.

[6] S. Eibl and U. Rüde, "A systematic comparison of runtime load balancing algorithms for massively parallel rigid particle dynamics," *Computer Physics Communications*, vol. 244, pp. 76–85, 2019.

[7] A. Morari, R. Gioiosa, R. W. Wisniewski, F. J. Cazorla, and M. Valero, "A quantitative analysis of os noise," in *2011 IEEE International Parallel Distributed Processing Symposium*, 2011, pp. 852–863.

[8] R. Procassini, M. O'Brien, and J. Taylor, "Load balancing of parallel monte carlo transport calculations," *Mathematics and Computation, Supercomputing, Reactor Physics and Nuclear and Biological Applications, Palais des Papes, Avignon, Fra*, 2005.

[9] E. R. Rodrigues, P. O. Navaux, J. Panetta, A. Fazenda, C. L. Mendes, and L. V. Kale, "A comparative analysis of load balancing algorithms applied to a weather forecast model," in *2010 22nd International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2010, pp. 71–78.

[10] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *International Conference on High-Performance Computing and Networking*. Springer, 1996, pp. 493–498.

[11] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[12] O. Meister, K. Rahnema, and M. Bader, "Parallel memory-efficient adaptive mesh refinement on structured triangular meshes with billions of grid cells," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 3, pp. 1–27, 2016.

[13] T. R. Scogland, B. Rountree, W.-c. Feng, and B. R. De Supinski, "Heterogeneous task scheduling for accelerated OpenMP," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 144–155.

[14] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, pp. 1422–1434, January 2018.

[15] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller, "Chameleon: reactive load balancing for hybrid mpi+ openmp task-parallel applications," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 55–64, 2020.

[16] P. Samfass, J. Klinkenberg, and M. Bader, "Hybrid mpi+ openmp reactive work stealing in distributed memory in the pde framework sam (oa)ˆ 2," in *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2018, pp. 337–347.

[17] M. Garcia, J. Corbalan, and J. Labarta, "Lewi: A runtime balancing algorithm for nested parallelism," in *Parallel Processing, 2009. ICPP '09. International Conference on*, Sept 2009, pp. 526–533.

[18] M. Garcia, J. Labarta, and J. Corbalan, "Hints to improve automatic load balancing with lewi for hybrid applications," *Journal of Parallel and Distributed Computing*, vol. 74, no. 9, pp. 2781–2794, 2014.

[19] M. D'Amico, M. Garcia-Gasulla, V. López, A. Jokanovic, R. Sirvent, and J. Corbalan, "DROM: Enabling efficient and effortless malleability for resource managers," in *Proceedings of the 47th International Conference on Parallel Processing Companion*, 2018, pp. 1–10.

[20] J. Clausen, "Branch and bound algorithms-principles and examples," *Department of Computer Science, University of Copenhagen*, pp. 1–30, 1999.

[21] D. F. Richards, R. C. Bleile, P. S. Brantley, S. A. Dawson, M. S. McKinley, and M. J. O'Brien, "Quicksilver: a proxy app for the monte carlo transport code mercury," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2017, pp. 866–873.

[22] "MiniFE performance benchmark and profiling," in HPC Advisory Council, http://www.hpcadvisorycouncil.com/pdf/miniFE_Analysis_and_Profiling.pdf, December 2015.

[23] T. Dionisi, S. Bouhrour, J. Jaeger, P. Carribault, and M. Pérache, "Enhancing load-balancing of mpi applications with workshare," in *European Conference on Parallel Processing*. Springer, 2021, pp. 466–481.

[24] F. Dupros, F. De Martin, E. Foerster, D. Komatitsch, and J. Roman, "High-performance finite-element simulations of seismic wave propagation in three-dimensional nonlinear inelastic geological media," *Parallel Computing*, vol. 36, no. 5-6, pp. 308–325, 2010.

[25] "Ondes3d," https://bitbucket.org/fdupros/ondes3d/, October 2017.