D12.5: Workload collocation based on container technologies to improve isolation

Deliverable No.:	D12.5
Deliverable Name:	Workload collocation based on container technologies to improve isolation
Contractual Submission Date:	30/04/2019
Actual Submission Date:	30/04/2019
Version:	v2.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730897.

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	**HPC-Europa3**
Project Full Name:	Transnational Access Programme for a Pan-European Network of HPC Research Infrastructures and Laboratories for scientific computing
Deliverable No.:	D12.5
Document name:	Workload collocation based on container technologies to improve isolation
Nature (R, P, D, O):	R
Dissemination Level (PU, PP, RE, CO):	PU
Version:	v2.0
Actual Submission Date:	30/04/2019
Author, Institution: E-Mail:	Oleksandr Rudyy, Marta Garcia-Gasulla, Raül Sirvent: BSC oleksandr.rudyy@bsc.es, marta.garcia@bsc.es, raul.sirvent@bsc.es
Other contributors	Giuseppa Muscianisi: CINECA

ABSTRACT:
The aim of this report is to study containers resource isolation and explore Dynamic Resource Assignment techniques to maximize hardware utilization. Containers bring virtualization benefits to High-Performance Computing systems, thus allowing the safe execution of concurrent applications within the same computational node. In this situation it becomes possible to dynamically re-assign resources of multiples applications given their workload, maximizing the useful time the hardware is being used. Nevertheless, containers do not fully isolate their environment from the host, so full isolation is not guaranteed. This report designs, implements and functionally evaluates the workload collocation concept applying dynamic resource assignment solutions for containers.

KEYWORD LIST:
High Performance Computing, HPC, Containers, DLB, Workload collocation, Dynamic Resource Assignment

MODIFICATION CONTROL			
Version	**Date**	**Status**	**Author**
0.1	1/04/2019	Template	PMT
1.0	18/04/2019	Complete Draft	Oleksandr Rudyy
2.0	26/04/2019	Final Version	PMT

TABLE OF CONTENTS

Executive Summary

This deliverable describes the results of Task 12.4: Workload collocation based on container technologies to improve isolation, in the Joint Research Activity (JRA) Work Package 12 of the HPC-Europa3 project. The document starts with an Introduction, motivating the need of Dynamic Resource Assignment (DRA) on HPC environments, and how container solutions can help us to achieve that scenario.

We also include a description on the environment, which briefly summarises Singularity as the container technology used, parallel programming models (MPI, OpenMP and OmpSs), the Dynamic Load Balancing library (the tool that helps us to achieve DRA capabilities), and finally the supercomputers used as testbeds for our evaluations: MareNostrum4 (from BSC), and MARCONI (from CINECA).

The biggest part of the deliverable is dedicated to the functional evaluation of the workload collocation scenarios we propose. Starting with the description of the three scenarios designed: Resource Isolation (avoiding interference between applications running in the same node), Resource Sharing (using free resources for the same or other applications, running in the same node), and the new development TALP (the Tracking Application Low-level Performance runtime), which gets online performance metrics of the applications and acts accordingly to improve their performance. The development of TALP is also a contribution of Task 12.4. All the different scenarios are thoroughly tested with the BT-MZ benchmark and analysed using performance analysis tools (Extrae and Paraver) to understand their insights. Additionally, the real world applications Quantum Espresso and Amber are tested for DRA capabilities using containers for their deployment.

Our conclusions show that, the DRA use case is possible and recommended in supercomputers, and containers help us to achieve it. The main gain we see is a faster response time in the execution of workloads (i.e. the set of applications finishes earlier), and an increased utilisation of the system, which is ideal for any supercomputer owner willing to exploit its big investment to its maximum capacity. We therefore see a good future for DRA and containers in current and future supercomputers, particularly with the current trend of having more and more resources inside a single computing node.

1 Introduction

A supercomputer contains a big set of resources that many different users want to exploit at the same time. This is the most classical scenario of what we call *resource allocation* in Computer Science: the set of available resources must be assigned to a set of different users, trying to maximise their usage, while achieving a fair share or following a specific scheduling policy. Supercomputers are very complex machines, and the set of resources they provide is therefore complex: computing cores organised in nodes, cache memories, main memory per node, distributed file systems, local file systems, different network interfaces, and so on.The mentioned resource allocation problem becomes really difficult, not only because many resources are offered, but also because a big number of users want to use them.

The most common approach in supercomputers nowadays is to book a set of resources for a single user, exclusively (i.e. not allowing other users run anything on the reserved nodes). Since supercomputers run queuing systems that deal with nodes in their scheduling, in most cases the granularity considered uses the concept of node as the basic resource unit to be assigned. This way, when an application asks for resources, it gets a reservation to use a number of nodes (depending on its demands), and the application runs there in an isolated manner. The node approach has two main advantages: on one hand, a simple unit for resource allocation is selected (the node), which makes the scheduling problem much more simple; on the other hand, reserving a node for a single user avoids interferences between applications, which could cause misbehaviour in terms of performance for both applications trying to share some resources. Besides, while current batch schedulers, e.g. SLURM, are capable of allocating multiple jobs per node (i.e. share a node's resources), the allocation achieved is static, based on the cores needed, and it is very likely that performance of the applications sharing the node are negatively affected, due to the lack of specific control of the resources used by each application.

Current trends in computer architecture, in particular architectures dedicated to build supercomputers, show us that the number of resources available in a single node is increasing fast. We can see that the number of cores has increased significantly during recent years, and not only that, but also new storage systems are commonly seen in new machines (the so-called NVMs, Non-Volatile Memories), together with specific processors to deal with big amounts of data in a vectorial way (i.e. the Graphical Processing Units, or GPUs). The more resources a node has, the less probable is that a single application is going to be able to exploit all of them at the same time. For instance, we can see applications with a high computational demand, that may be interested in using the GPUs to speed up the processing (or not using them at all), but on the other hand, applications with a higher demand on the storage system may wish to exploit the NVM available (or, again, leaving NVMs free). Having these scenarios in mind, we are confident to foresee that, in a near future, supercomputers will have to share resources inside a node for different applications, while ensuring at the same time their performance remains unaffected (as mentioned in the previous paragraph). This is why in the HPC-Europa3 project we envisioned the need of Dynamic Resource Assignment (DRA) features to achieve both effective and efficient sharing of resources.

New emerging supercomputing paradigms also demand more dynamicity in the way resources are assigned, For instance, the Interactive Supercomputing paradigm (as proposed in the Human Brain Project). In this scenario, an application (e.g. a simulation) is supposed to start using all the resources that have been made available to it, and, at a certain moment, an interactive action is requested by the user (i.e. an analysis of partial results, or their visualization). This kind of scenario cannot be fulfilled in current supercomputers, since the set of resources is fixed, and the only solution is to overload the resources provided, with the corresponding affectance in performance for both simulation and analytics. Therefore, to efficiently solve this, the system is expected to be able to adapt the number of resources the simulation is using, to make room for the analytics or visualization job that must be served at that moment. Again, DRA features can solve this situation in the most effective manner.

Container technologies have emerged as a lightweight solution to achieve portability of applications between systems, as opposed to full-stack virtualisation. Containers remove the Operating System virtualisation layer to reduce the typical overhead found when using full-stack virtualisation, and this makes them a suitable solution to achieve portability between HPC systems, where performance is of paramount importance. When considering container solutions for supercomputers, we can find schedulers able to assign containers to a set of disjoint resources (e.g. Kubernetes). However, none of these schedulers includes the capacities we want with respect to DRA (i.e. share resources between containers in a single node).

The objective of Task 12.4 in HPC-Europa3 is to achieve a Dynamic Resource Assignment scenario with the help of containers. Our starting point will be hybrid applications (i.e. that exploit both intra- and inter-node parallelism levels) as the most common paradigm used in supercomputers to achieve their full potential. Besides, BSC contributes to this task with DLB [1], a Dynamic Load Balancing library able to act as a *node scheduler*, and thus enact the shifting of resources between applications inside a node. The current implementation of DLB is able to handle CPU cores as resources, although future versions could include more node's resources (i.e. memory, disk, …), therefore the *core* will be the resource we will target for dynamicity. This essentially means that, if a parallel application using a number of threads enters a sequential phase, or a synchronization point where some threads cannot progress with work, the unused resources will be lent to other threads of the same application (or to another application) so it can speed up its execution. Effectively, this means that the utilisation of the system is overall increased, as is the main interest of the supercomputer owner.

This deliverable D12.5 – Workload collocation based on container technologies to improve isolation (due M24), reports the results of Task 12.4 in the JRA (Joint Research Activity) of the project. It includes two years of work, since the initial installation and configuration of the environments used, the design of the different scenarios to be tested, and finally the presentation of their evaluation. As it will be presented in Section 3, three main scenarios are considered:

- Resource isolation: where DLB will ensure that the applications inside a node do not disturb each other, avoiding performance issues.
- Resource sharing: where DLB will be able to shift unused resources from an application to another.
- Online statistics: where instead of reacting to an action of releasing resources, a runtime system (TALP) will collect online statistics that will serve to take decisions on resource allocation in the node.

The three scenarios will cover the cases we foresee for a supercomputer. In the scenarios, the concept of scheduling, compared with current practise, gets a new degree of complexity with our approach, since the unit to be assigned to a user is no longer a node, but a set of resources inside it (i.e. cores). This is our contribution to achieve a more dynamic scheduling of resources in supercomputers.

2 Environment

2.1 Singularity

Singularity is a container implementation focused to be run on High-Performance Computing environments and the one we chose to carry out this study. In Deliverable D12.1-Container-as-a-service analysis report [2], it is concluded that Singularity best suits the HPC requirements due to its easier deployment, safer security paradigm and presumably better performance. Therefore, our workflow collocation tests will be performed using Singularity as a way to not only avoid installation problems for the HPC sys-admins regarding security and performance issues, but also to reach the maximum number of users interested in DRA techniques with containers.

In addition to its ease of use, Singularity needs minimal integration to leverage dynamic scheduling of its resources. By default, every Singularity container does not apply cgroups[1] to constraint hardware usage and only isolates its Mount[2] namespace while sharing the others with the host. As a result, applications running within containers have a complete view of the hardware resources of the host and are able to communicate with the processes outside the containerized environment.

2.2 Parallel Programming Models

Since the tendency in HPC centers is to deploy clusters with massive amounts of processing units (i.e., cores), it is mandatory for scientific applications to exploit either the inter- or intra-node parallelism of its code through parallel programming models. For the inter-node parallelism, the standard programming model is the Message Passing Interface (MPI), whereas for the intra-node is OpenMP. Nevertheless, the current OpenMP API (Application Programming Interface) is inadequate for dynamic resource reassignment because it does not offer a fine grained control over the parallel regions of an application. To solve OpenMP deficiencies, there exists the OmpSs [3] [4] programming model, which allows a better resource control.

OmpSs is a parallel programming model developed at Barcelona Supercomputing Center (BSC) with the aim to extend OpenMP so it can support asynchronous parallelism and heterogeneity. The OmpSs environment is built on top of BSC's Mercurium compiler [5], and Nanos++ runtime system [6]. Applications using OmpSs are more malleable by the Nanos++ runtime because of its execution model. While OpenMP framework uses a fork-join model, OmpSs works based on a thread-pool model represented in Figure 1, which is easily malleable by the runtime since it can directly manage the processes' threads and the workload of each one.

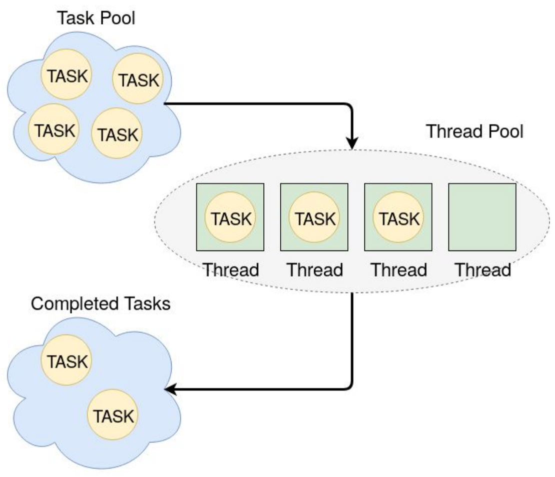


Figure 1: The thread pool model used by OmpSs

[1] More details about cgroups in: http://man7.org/linux/man-pages/man7/cgroups.7.html
[2] More details about Mount namespace in: http://man7.org/linux/man-pages/man7/mount_namespaces.7.html

For our tests, we will focus on MPI+OpenMP and MPI+OmpSs programming models to take advantage of both intra- and inter-node parallelism. On one hand, we will be able to measure the workload imbalance between different MPI ranks. On the other hand, OmpSs will allow us to easily reassign the number of available threads for each rank in order to reduce the detected imbalance. In favor of reproducibility, we will be using the Open MPI implementation [7] of the MPI interface and GNU compilers.

2.3 Dynamic Load Balancing (DLB) Library

The Dynamic Load Balancing (DLB) [1] [8] library is a dynamic library designed to speedup hybrid applications by enabling an efficient utilization of the computational resources.

The DLB library is transversal to the different layers of the software stack as shown in Figure 2. It coordinates with them using standard mechanisms or APIs.

The current stable version of DLB (2.1) offers support for OpenMP, OmpSs and MPI.

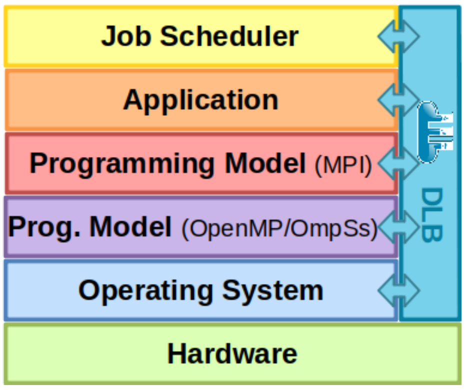


Figure 2: DLB interaction with the application layers

DLB offers an API that can be called from job schedulers, runtime systems or applications. It intercepts the MPI calls using the PMPI interposition mechanism. DLB also uses standard API and functionalities from the operating system and OpenMP.

DLB aims at being as transparent as possible to the application, using whenever it is possible standard mechanisms or APIs (i.e, PMPI interface or OpenMP standard API). But, at the same time, it offers the user the flexibility to use a simple API from the application code.

DLB is organized in modules that are independent but also compatible and complementary. The three modules currently implemented in DLB are: LeWI, DROM and TALP, and they are explained in next sub-sections.

2.3.1 LeWI: Lend When Idle

The LeWI module will improve the load balance of hybrid applications by changing the number of threads assigned to each process. To achieve a better load balance between processes, LeWI will lend the computational resources assigned to one process when it enters an MPI blocking call to another process running in the same node.

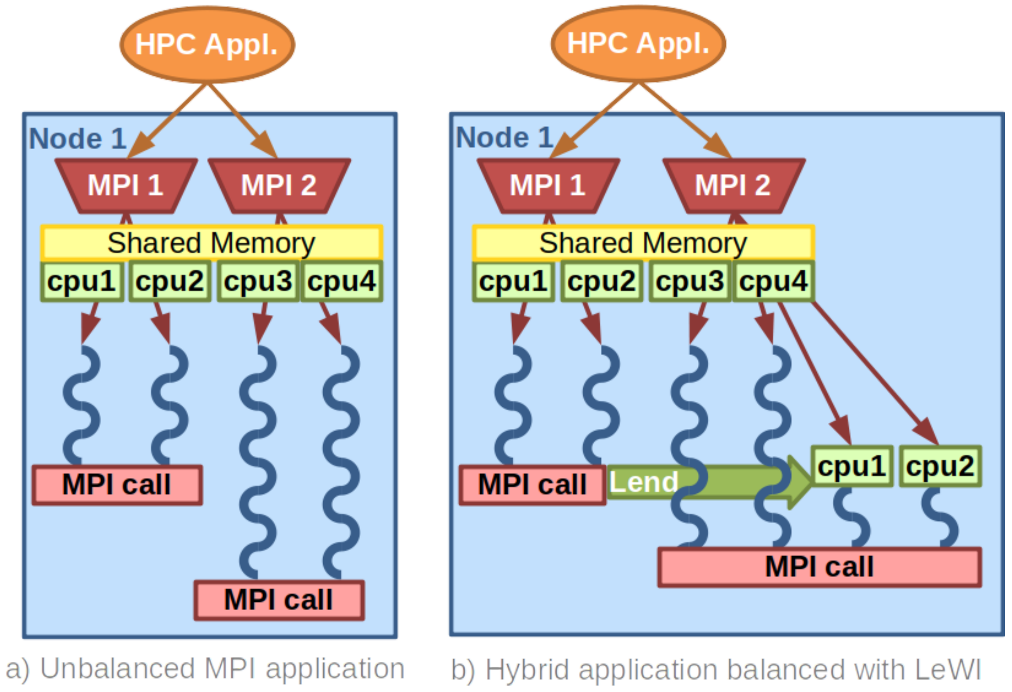


Figure 3: Example of LeWI functioning

In Figure 3, we can see an example, on the left hand side we can see a hybrid application, with two MPI processes and each one spawning two OpenMP threads. On the right hand side, the same application being balanced using LeWI, when the MPI process 1 enters the MPI blocking call it will lend its two CPUs (cpu1 and cpu2) to the MPI process 2. At this point, MPI process 2 can use 4 threads (instead of 2) and finish its computation faster. When the MPI process 1 exits the MPI blocking call it will recover its previous resources.

2.3.2 DROM: Dynamic Resource Ownership Management

The DROM module [9] is designed to be used by resource managers, for example job schedulers, but it offers an API that can also be called from the application.

DROM allows to change the number of computational resources assigned to a process by changing the number of threads of the second level of parallelism (i.e. OpenMP or OmpSs) during its execution.

This module can be used, for example, by a job scheduler to give priority to one job by reducing the number of CPUs assigned to running jobs and assigning the freed CPUs to a high priority job.

2.3.3 TALP: Tracking Application Low-level Performance

The TALP module obtains performance metrics of the MPI processes during their execution and provides an API to consult them by the application or other entities. For each MPI process, TALP collects the time spent doing MPI communication and the time spent doing useful computation. For the different CPUs in the system, it registers the amount of time that they are being used, idle or lent. Additionally, TALP provides two API calls: dlb_autosizer_start and dlb_autosizer_end, that will check the performance metrics of the calling process and based on them TALP can decide to change the number of threads that the process is using by calling the DROM module.

2.3.4 Porting of DLB to containers

In this project, we have ported and tested DLB in a containerized environment.

All the communication and data stored in DLB is done through a shared memory allocation that is visible to all the processes in the system. This allows DLB to behave within a container in the same way it would behave in bare metal.

One of the advantages is that DLB can be installed inside the container allowing a more transparent deployment for the user. As we can see in Figure 4 the communication with the different layers of the software stack is the same as when working in a bare metal environment.

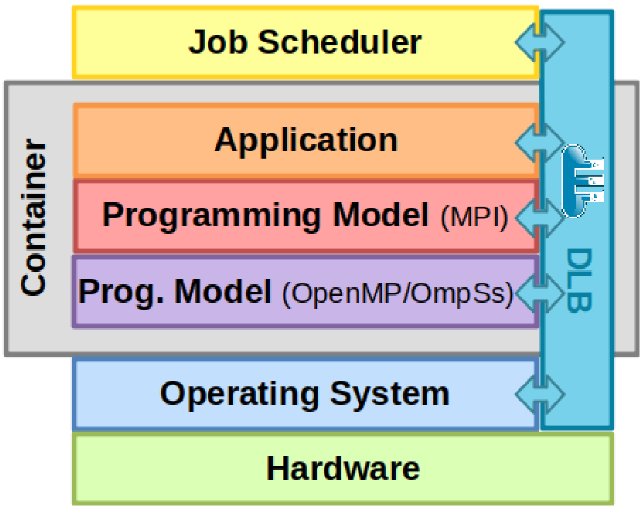


Figure 4: DLB interaction with the layers of the containerized app

2.4 Testbed Description

For this deliverable we have 2 high-end supercomputers available where to evaluate container technology: MareNostrum4 and MARCONI. With them, we will perform our tests regarding containers and DRA strategies. The details of both systems are described in next two sub-sections.

2.4.1 MareNostrum4

MareNostrum4 is a Tier-0 supercomputer in production at Barcelona Supercomputing Center (BSC) in Barcelona, Spain. It has available 3456 nodes in total where each node is based on Intel Xeon Platinum 8160 CPUs with 48 cores [10]. As depicted in Figure 5, one MareNostrum4 node contains 2 sockets with 24 cores per socket. Cores from different sockets share the main memory (96GB), but not cache memories. However, cores within a socket have a common level 3 cache. The fast MPI interconnection network is 100 Gbit/s Intel Omni-Path.

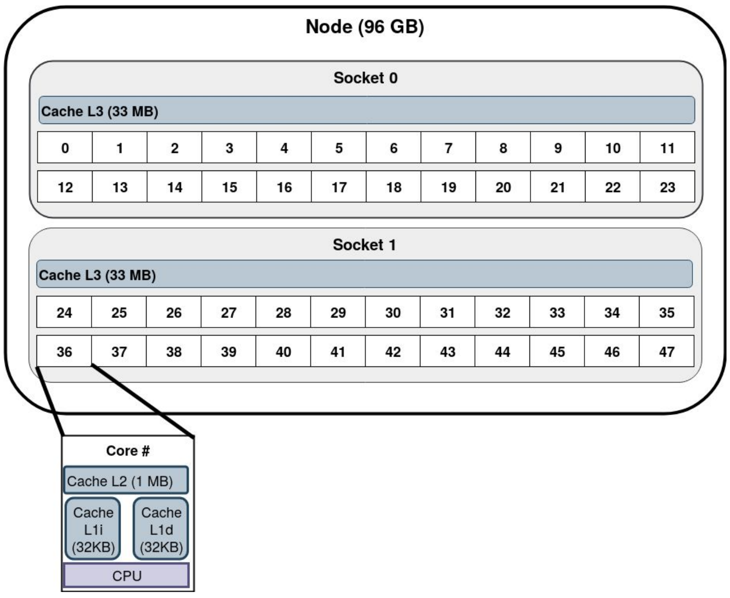


Figure 5: Depiction of 1 node from MareNostrum4 containing 2 sockets with 24 cores each

MareNostrum4 runs Linux 4.4.12 kernel and uses a modules environment, so it has multiple versions of Open MPI and GNU compilers available. It possesses a functional installation of Singularity 2.4.2 which we will use to run our tests with containers.

2.4.2 MARCONI

MARCONI is the CINECA Tier-0 system based on the Lenovo NeXtScale platform, located in Bologna, Italy. This Tier-0 system started its life in mid-2016, with the set-up of the first partition (A1) based on Broadwell chips and a computational power of 2 PFlops peak. After several upgrades, at the beginning of 2018 it was configured in three partitions: A1, made of 720 Intel Broadwell nodes; A3, with 2306 Intel SkyLake nodes; and A2, a scale-out partition made of 3600 many-core Intel Knights Landing nodes, with a total peak performance of about 20 Pflops. This configuration has been further enhanced in October 2018, with the upgrade of all A1 nodes to SKL (SkyLake).

The current configuration of MARCONI is comprised of:

- 3600 Intel Knights Landing nodes, each equipped with 1 Intel Xeon Phi 7250 @1.4 GHz, with 68 cores each and 96 GB of RAM, also named as MARCONI A2 - KNL
- 3216 Intel SkyLake nodes, each equipped with 2 Intel Xeon 8160 @ 2.1 GHz, with 24 cores each and 192 GB of RAM, also named as MARCONI A3 - SKL.

This supercomputer takes advantage of the Intel Omni-Path Architecture, which provides the high performance interconnectivity required to efficiently scale out the system's thousands of servers. A high-performance Lenovo GSS (GPFS Storage Server) storage subsystem, that integrates the IBM Spectrum Scale™ (GPFS) file system, is connected to the Intel Omni-Path Fabric and provides data storage capacity for about 10 PByte.

The MARCONI A3 (SkyLake) partition will be used for testing. On such nodes, the operative system is CentOS 7.3.1611. The software is available by a modules, and the Singularity version used for testing will be 3.0.1.

3 Functional Evaluation of Workload Collocations

3.1 Scenarios

In order to evaluate the workload collocations and resource isolation of containers we have used three different scenarios that we will explain in the following sub-sections.

3.1.1 Resource Isolation

The first scenario is the one ensuring resource isolation. In this scenario, different applications have allocated disjoint computational resources inside different containers, but both are executed in the same computational node.

Each application can use DLB and LeWI to improve its performance but the computational resources will not be shared across different containers.

In Figure 6, we can see an example of two applications running into containers inside a node. They will not share resources with each other but they will be able to improve its individual load balance with DLB and

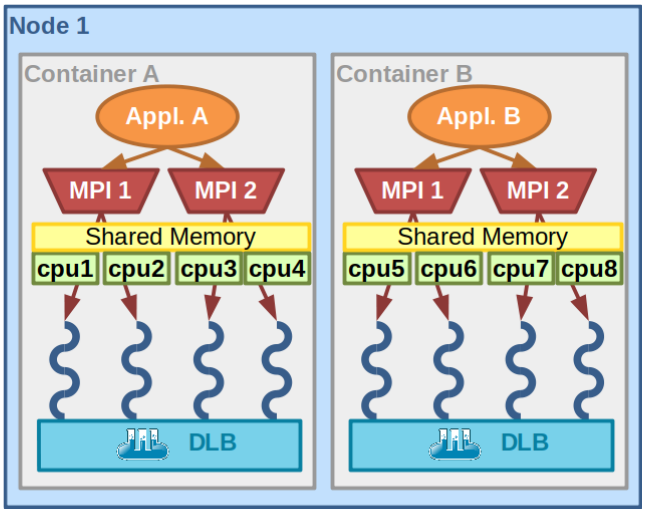


Figure 6: Example of two hybrid applications running within two different containers and sharing a node

LeWI. This way we achieve the Dynamic Resource Assignment (DRA) scenario targeted in this task, ensuring resource isolation between applications.

We will refer to this scenario as "Resource Isolation".

3.1.2 Resource Sharing

In this scenario, we will try to take advantage of the load balancing capabilities of DLB by allowing resource sharing between applications running in different containers.

In Figure 7, we can see an example of two applications running in a computational node inside two different containers and sharing resources with DLB and LeWI.

We will refer to this scenario as "Full Resource Sharing".

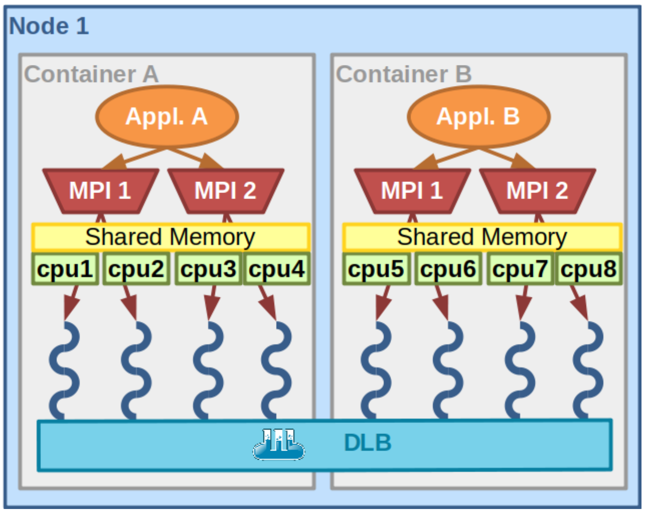


Figure 7: DLB Full Resource Sharing scenario

As the number of cores per node has increased in recent generations of supercomputers, also their architectures become more complex. It is usual nowadays to find computational nodes composed by two sockets (or processors) in a NUMA (Non Uniform Memory Access) configuration. With this kind of architectures, it is common to see that it is not optimal to use cores from the other socket if all the threads of your application are running in the other socket, because all the data of the application is already allocated in one of the sockets' corresponding memory.

For this reason, we define the "Affinity Aware" scenario; this scenario can share resources between containers as the "Full Resource Sharing" scenario, but, DLB will take into account the affinity of resources. This means that each application can only use CPUs from the sockets where the application was originally launched.

In Figure 8, we can see an example where 3 applications running in three different containers run in an "Affinity Aware" scenario. Application A and B share Socket 1, while application C runs in Socket 2. In this case, the three applications can share resources to improve the load balance through DLB and LeWI, but the affinity aware scheduler will not allow application C to run in CPUs 1 to 4, neither application A nor B to run in CPUs 5 to 8. But, application A and B will be able to run in each other resources.

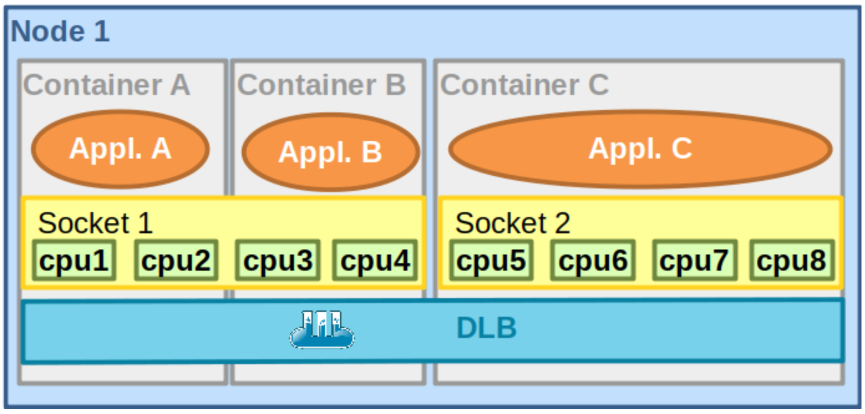


Figure 8: DLB Affinity Aware scenario

As a final consideration, DLB allows to use the computational resources of one application for another after the first one has finished its execution. In a classical HPC environment, when one application finishes its resources are returned to the resource manager or job scheduler. With DLB we consider the option of using the resources "Post-Mortem", meaning that when one application running within the DLB system finishes its execution, its resources can be used by another application running in the same DLB system.

3.1.3 Dynamic Resource Assignment

Finally, we consider a scenario where applications monitor themselves and take appropriate actions regarding resource utilization to improve the overall efficiency.

For this scenario, we use DROM and TALP modules from DLB and we instrument the application adding calls to the API. The API calls are added at the most external loop of the application to monitor its parallel efficiency at the MPI level. In the following piece of code we show an example of the instrumentation and use of the API:

```
for (iter=0; iter<N; iter++)
{
    dlb_autosizer_start()
    ...
    //Application code for each step
    ...
    dlb_autosizer_end()
}
```

The call to dlb_autosizer_start at line 3 initializes the counters to gather the metrics for this iteration and based on the metrics gathered in the previous iterations takes the appropriate actions. These actions include modifying the amount of computational resources assigned to itself, either by increasing or decreasing them. To apply these changes, it will call the DROM module which is in charge of changing the resources assigned to a process.

At line 7, the call to dlb_autosizer_end will reduce the metrics collected during the iteration and store them in the shared memory of DLB so they are available to be consulted later.

Note that in this scenario the assignment of resources is done based on measurements of previous iterations, therefore it can solve imbalances or inefficiencies produced algorithmically and iterative.

Although processes manage themselves the resources this is coordinated within DLB to avoid core oversubscription. I.e, a process can only get more resources when another processes has released or freed some CPUs because it was infra-loaded.

In the following evaluation we will refer to this scenario as TALP.

3.2 NPB BT-MZ

BT-MZ 3.3.1 [7], [8] belongs to the set of NAS Parallel Benchmarks, which are derived from computational fluid dynamics (CFD), focused on evaluating the parallel performance of supercomputers. From all the set of NPB, we have chosen BT-MZ (Block Tri-diagonal solver Multi-Zone) because it is designed to exploit multi-level parallelism (MPI+OpenMP) and presents uneven workload allocation. CPU, network and memory bandwidth/latency are what mostly affect BT-MZ's performance.

Nevertheless, we noticed that the latest version of BT-MZ performs a load balancing algorithm during its initialization. This algorithm distributes intelligently the data across the available MPI ranks. So, we decided to modify the source code to disable the default mapping algorithm so the workload imbalance is more evident. In addition, the original sources of BT-MZ use OpenMP directives incompatible with the OmpSs programming model, thus, we have been forced to adapt the parallel regions declarations of the code. As a result, our benchmark is a modified version of BT-MZ 3.3.1 which presents a notable workload imbalance and is able to run with both OpenMP and OmpSs programming models, depending on the compiler used.

As an example of BT-MZ workload imbalance, in Figure 9 we show the MPI calls pattern being performed by BT-MZ during the execution of its iterations. In the picture, we can appreciate four rows each row representing one rank MPI of BT-MZ. In each row there are green blocks representing the *MPI_waitall* call being executed by the respective process, and between the blocks an empty space which indicates computation time. Thanks to the picture, you can observe how the 4th rank possesses a lot more work than ranks 1, 2 and 3 because it does not remain blocked by the MPI runtime. Just the opposite of the first rank, which almost has no work compared to its neighbors and must remain blocked until the last rank has finished.

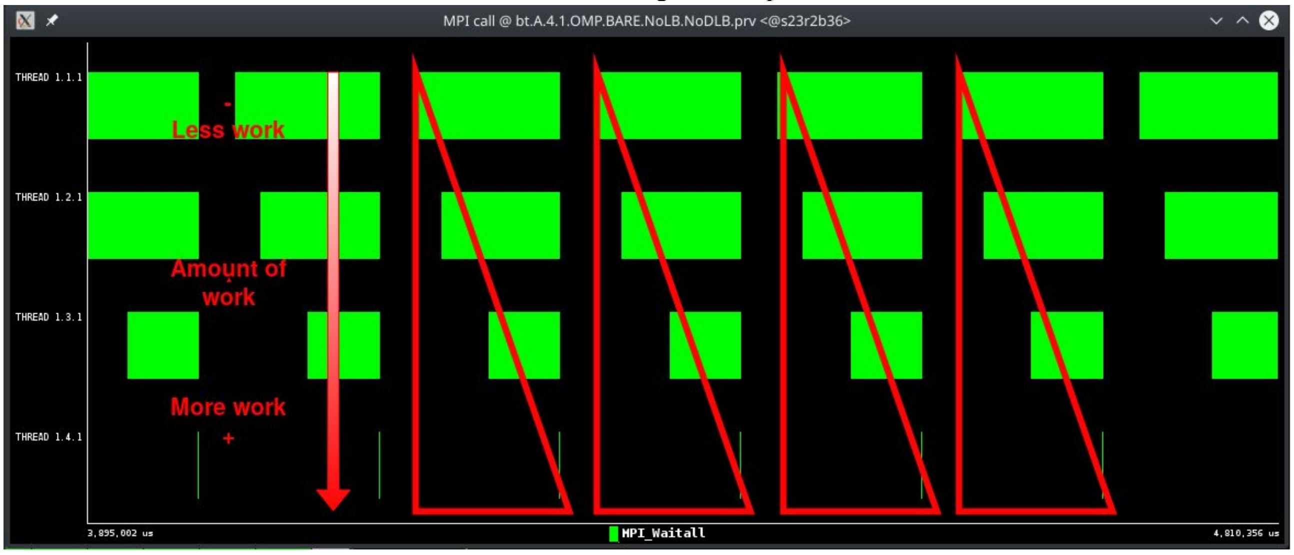


Figure 9: MPI call trace of 4 ranks BT-MZ

Because of the load imbalance of BT-MZ, we believe that this application fits the best for our evaluation. When testing various workload allocations and applying the Dynamic Load Balancing library we will be able to easily quantify the speedup and demonstrate the pros and cons of each strategy.

In the next sub-sections we present a functional evaluation and preliminary results of the previously described scenarios (Section 3.1). First of all, we will validate whether the performance of native and containerized executions of BT-MZ is the same. Next, we will apply each scenario one by one to BT-MZ within a container and evaluate its execution time. In the end, we will summarize all the tests and compare the performance obtained by each strategy.

3.2.1 Environment Validation

Before evaluating workload allocation and dynamic resource management strategies, we must check that the application performance remains unaffected by the container layer. For this purpose, we will execute our modified version of BT-MZ benchmark using all cores of one MareNostrum4 node with and without containers. In addition, we will explore different MPI+OpenMP allocations to see if it affects somehow the virtualization. However, since we are interested in exploiting the container isolation to run various applications within the same node, we will also verify that the simultaneous execution of 2 applications within the node does not affect their performance. We will first execute our tests with one single BT-MZ. Then, we will compare its execution times with a test where we run 2 different BT-MZ at the same time. In Table 1, we summarize the BT-MZ versions and allocation configurations we will explore in the first validation test.

Technology	Native, Singularity				
Simultaneous applications running	1				
BT-MZ version	MPI+OpenMP, MPI+OmpSs				
Ranks MPI x threads per rank	2x12	4x6	6x4	8x3	12x2

Table 1: Summary of the first validation test

For the first validation test we will compare the native and within a container execution times of one BT-MZ process. We also will compare its execution time when compiled with OpenMP and OmpSs to demonstrate that OmpSs achieves the same performance. Figure 10 shows an example of how we will allocate the resources for this test with 2x12 MPI+OpenMP allocation. Notice that for all the explored configurations we will only fill one socket of the node while the other remains empty. This will allow us to later fill the remaining socket with another BT-MZ process and compare the outcomes of this test with the following.

The Singularity container we will be using consists of a Debian 8 distribution where we have installed Open MPI 1.10.7 libraries (same as in the host) and the GNU compilers offered by Debian repositories. The container also contains the runtime environment required by OmpSs (Nanos++ and Mercurium compiler).

Figure 11 shows the average execution time of 5 runs with OpenMP and OmpSs exploiting different resource allocations. In the *x*-axis the number of MPI ranks and threads per rank deployed are represented, while the *y*-axis shows the average execution time in seconds. Each label from the *x*-axis contains 4 bars with light blue, strong blue, light red and strong red. The light colors belong to native executions with OpenMP and OmpSs, whereas the strong ones to container executions. It is visible that, despite a very slight variability, the performance difference among technologies and programming models in every resource allocation is minimum and negligible. That variability is due to using differents Nanos++ and compiler versions within the native and the containerized environment. One can also appreciate that BT-MZ performs better as more MPI ranks it has, but this is dependent on the benchmark.

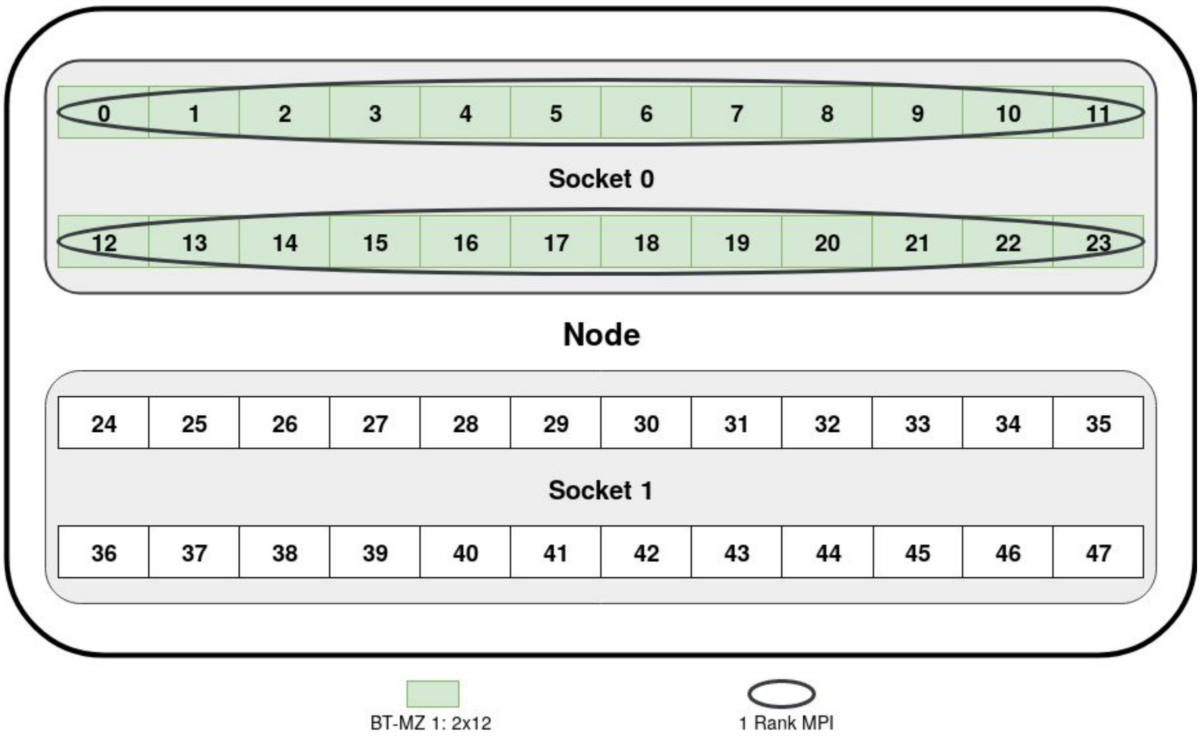


Figure 10: Depiction of 1 BT-MZ resource allocation with 2 ranks MPI and 12 threads per rank

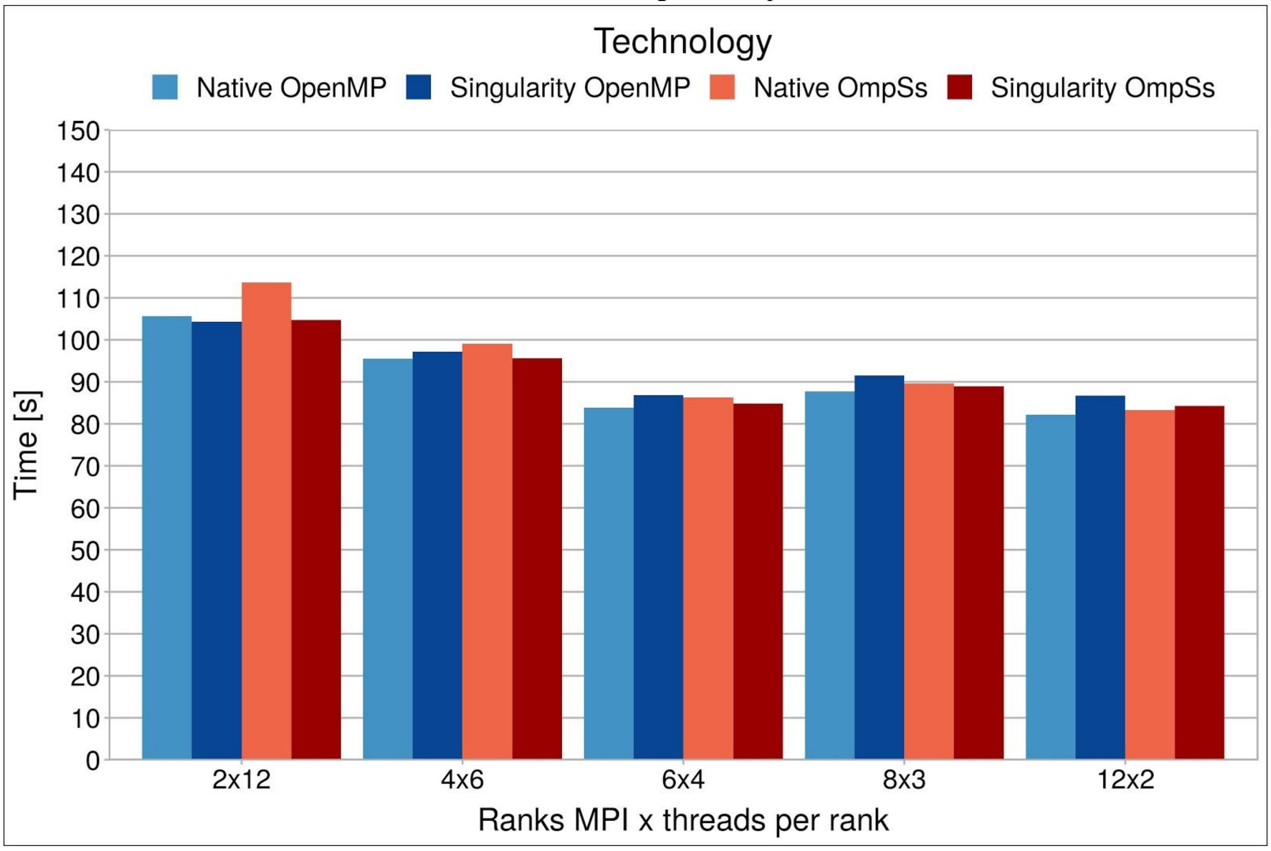


Figure 11: Execution time comparative of BT-MZ with native and containers

After verifying the performance running one application process within containers equals native executions, we will check that the execution times when running 2 applications or containers simultaneously do not vary. In the test above, we were only exploiting one socket of the two available on purpose. Now, we can launch in the second socket another BT-MZ process running with the same resource allocations. Figure 12 shows an example of this test, where we are exploiting the whole node with 2 different BT-MZ processes.

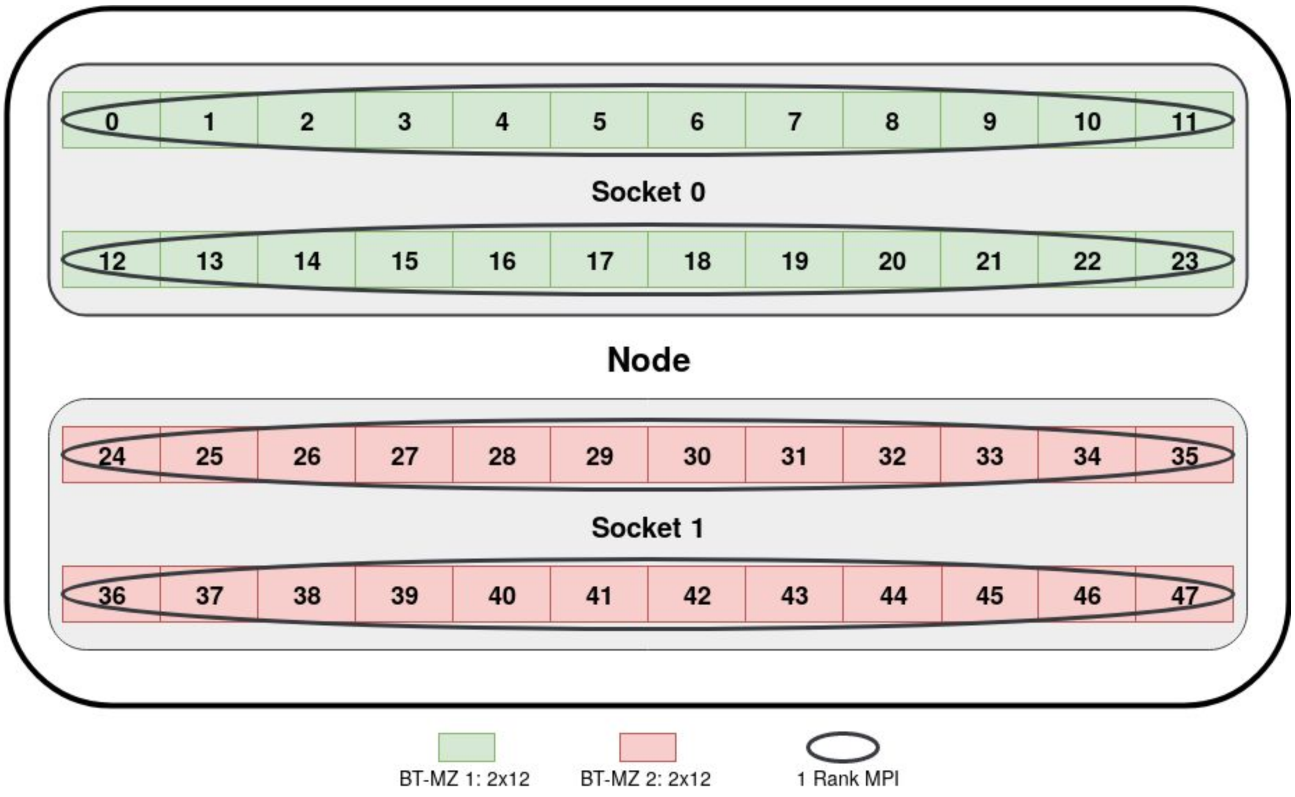


Figure 12: Depiction of 2 BT-MZ running simultaneously with 2 ranks MPI and 12 threads per rank

To avoid repeating the same experiment, instead of comparing OpenMP and OmpSs versions of BT-MZ, we will execute BT-MZ with OmpSs plus DLB using resource isolation and full resource sharing strategies. Again, we also will repeat the runs using native and virtualized environments. Table 2 summarises the test's details.

Technology	Native, Singularity				
Simultaneous applications running	2				
BT-MZ version	MPI+OmpSs with DLB				
Resource collocations strategies	Resource Isolation, Full Resource Sharing (FRS)				
Ranks MPI x threads per rank	2x12	4x6	6x4	8x3	12x2

Table 2: Summary of the second validation test

We run both BT-MZ processes 5 times with each technology, resource allocation, and strategy to get the average execution time. We observed that, in this case, both processes take the same execution time to finish, which shows us that no process interferes with its neighbour. Because of that, in Figure 13 we only represent the maximum average execution time from each configuration. In the *x*-axis it is represented the used resource allocation and in the *y*-axis the time in seconds. The light green and purple bars belong to native executions whereas the strong green and purple to executions within containers. Together, the green color belongs to Resource Isolation strategy and the purple to Full Resource Isolation.

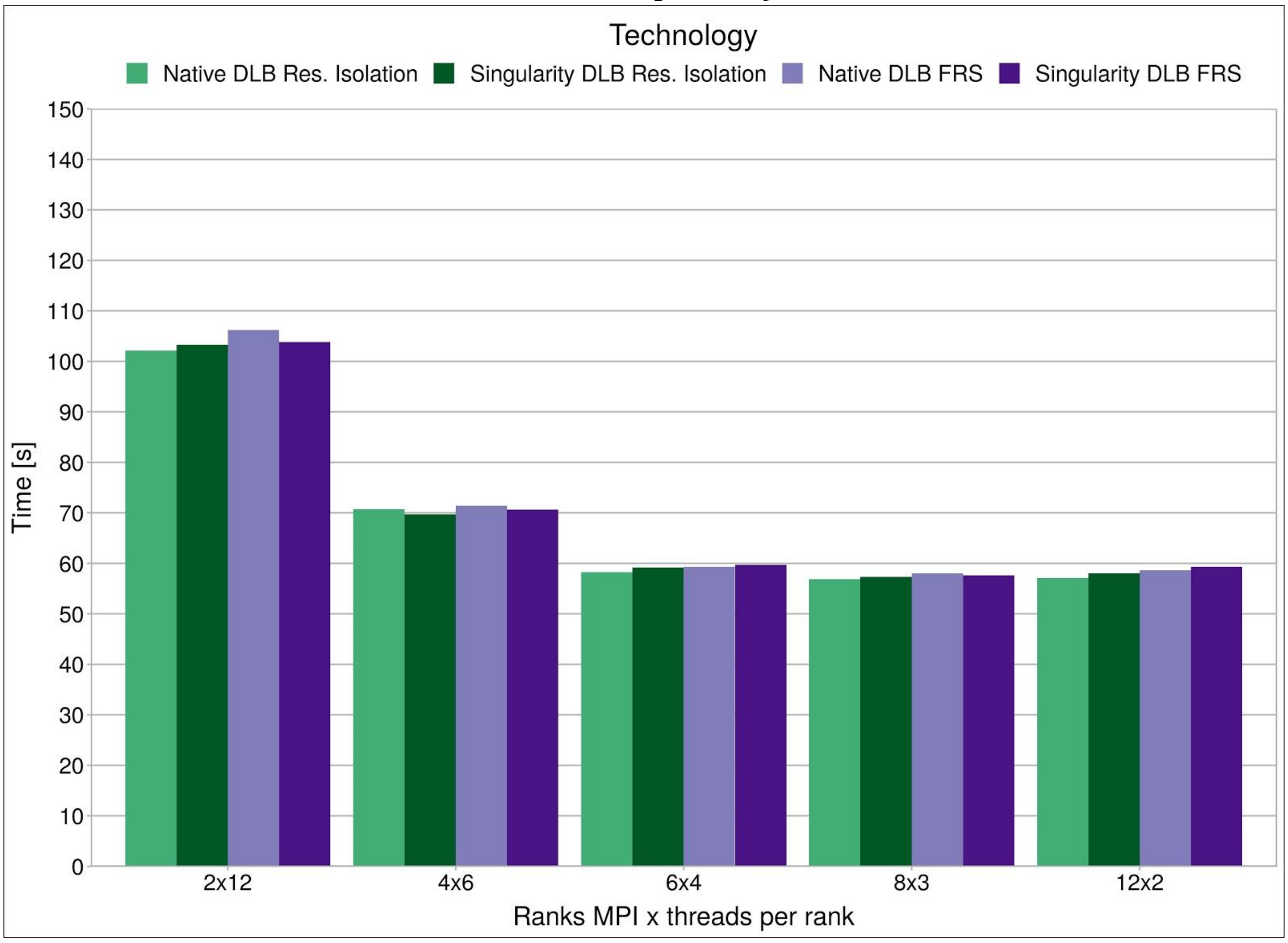


Figure 13: Maximum execution times comparative of BT-MZ with native and containers, this time when running 2 applications concurrently

As in the first validation test, native and within containers executions take almost the same time in every MPI+OmpSs configuration.. The minimal difference observable in each resource allocation is due, again, to the different versions of the runtime and compiler used within the container.

If we compare the two DLB strategies tested it might seem strange that the FRS strategy does not present any improvement, since each application has available a lot more cores than the initially allocated. Considering that the two BT-MZ being executed possess the same resource allocation and data, their computational patterns match perfectly meaning that none of them can benefit from the resources of the other process. Besides, the thread migration between sockets causes misses in L3 cache, penalizing this kind of scenarios.

Finally, to end this section, we determined the speedup obtained by each technology and BT-MZ version with respect to the BT-MZ with OmpSs execution in native from the first validation test. The speedup has been obtained as:

$$Speedup^{configuration}_{MPI\ x\ threads} = \frac{\bar{t}^{configuration}_{MPI\ x\ threads}}{\bar{t}^{Native\ OmpSs}_{MPI\ x\ threads}}$$

Where *configuration* can be: Native/Singularity OmpSs, Native/Singularity DLB Resource Isolation or Native/Singularity DLB FRS; and *MPI x Threads*: 2x12, 4x6, 6x4, 8x3 or 12x2.

The *x*-axis and *y*-axis of Figure 14 represent the resource allocations and speedup respectively. Different bar colors belong to the various configurations of technologies (native or Singularity) and strategies (without DLB, DLB Resource Isolation and DLB Full Resource Sharing) experimented.

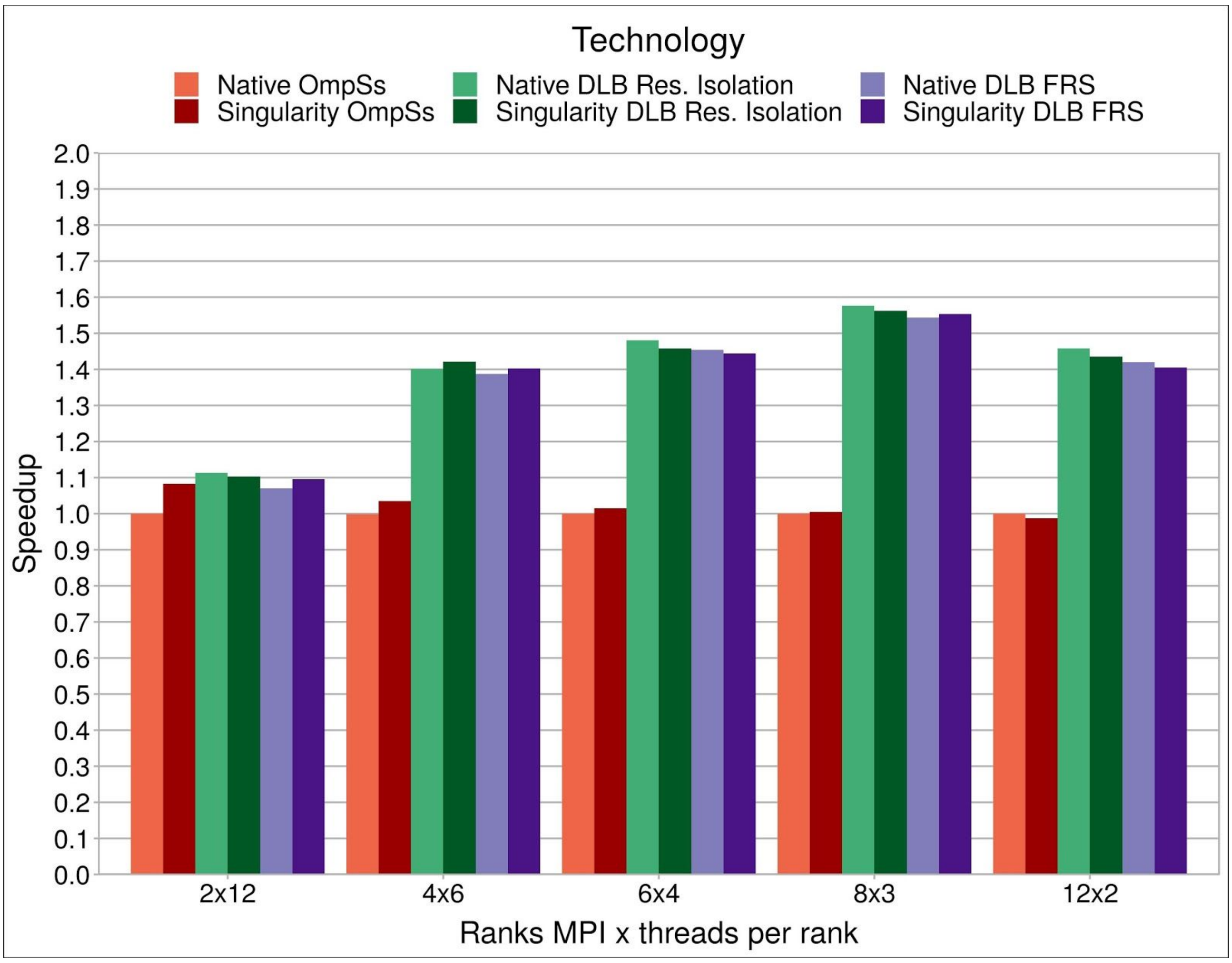


Figure 14: Execution time speedups with respect of native OmpSs

In the speedup plot it is clearly visible the gain provided by DLB and that the performance between native and Singularity remains akin. DLB speeds up each application on its own, achieving the Dynamic Resource Allocation scenario we intended, while at the same time not disturbing the other BT-MZ application running in the same node. With DLB we obtained the maximum speedup when applying the Resource Isolation strategy in the 8x3 allocation (around 1,58 of speedup). In the 2x12 configuration the speedup obtained by DLB is minimum because there are not enough parallel tasks in BT-MZ to fill all the potential threads supplied, as displayed in Figure 15. Each color block in the figure represents a task. Up to 14 threads there are enough tasks to fill each CPU, but from 15 threads onwards the runtime is not able to satisfy the work demand. Therefore, one way to optimize DLB library effect is to increase the amount of MPI workers so the most loaded ranks can borrow the threads of the less loaded.

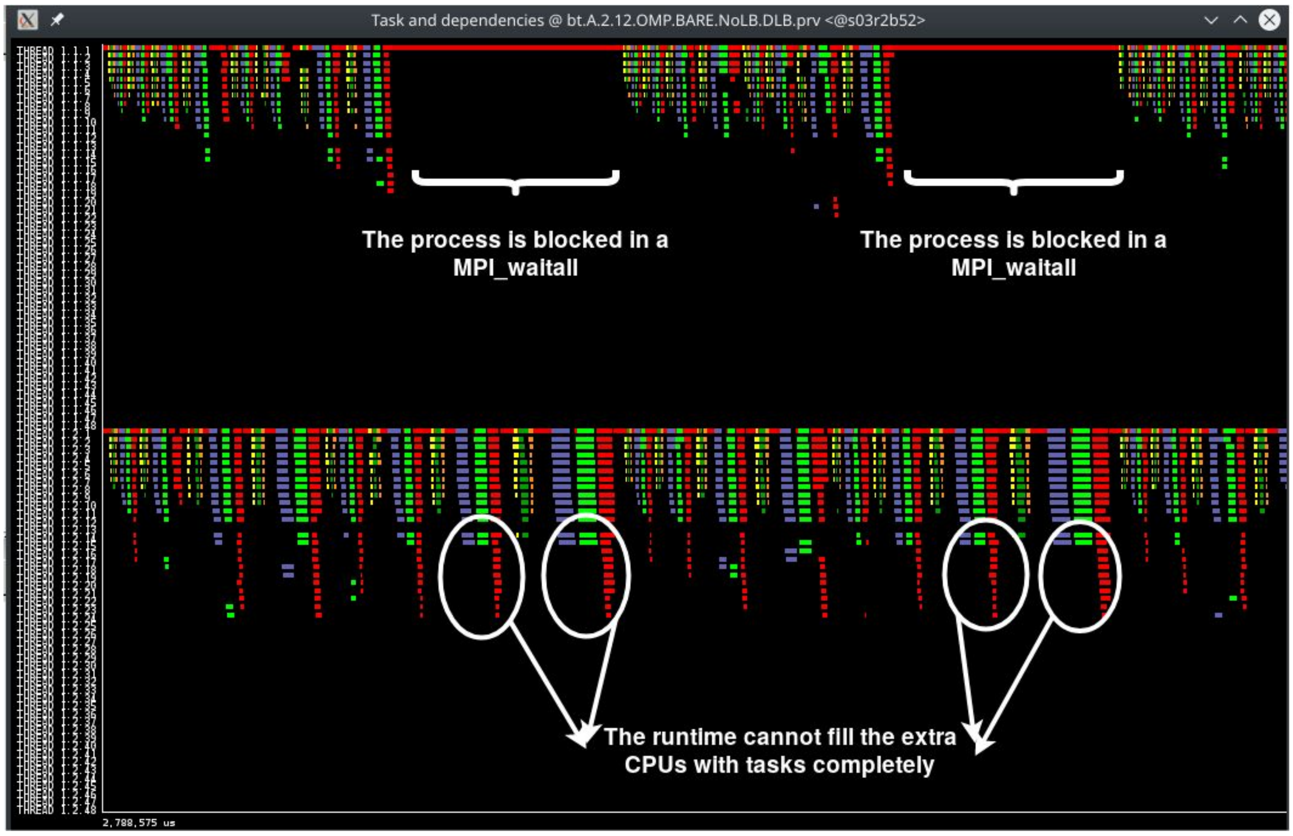

Task and dependencies @ bt.A.2.12.OMP.BARE.NoLB.DLB.prv <@s03r2b52>
The process is blocked in a MPI_waitall
The process is blocked in a MPI_waitall
The runtime cannot fill the extra CPUs with tasks completely
2,788,575 us

Figure 15: BT-MZ trace showing the execution of its tasks with 2 ranks MPI and 12 threads per rank. The rank in the bottom part of the figure uses the resources available from the top rank, while blocked

3.2.2 Scenario 1: Resource Isolation

In order to test our workload collocation and dynamic resource management strategies, we designed a possible use case where we are executing concurrently 5 different BT-MZ processes within the same computational node, each process with different resource allocation. Figure 16 depicts the use case where we launch 5 BT-MZ each one with 3x2, 6x3, 2x2, 2x5 and 5x2 ranks MPI and threads per rank, filling the whole node. Each BT-MZ process possesses a particular color to highlight that it is independent of the others.

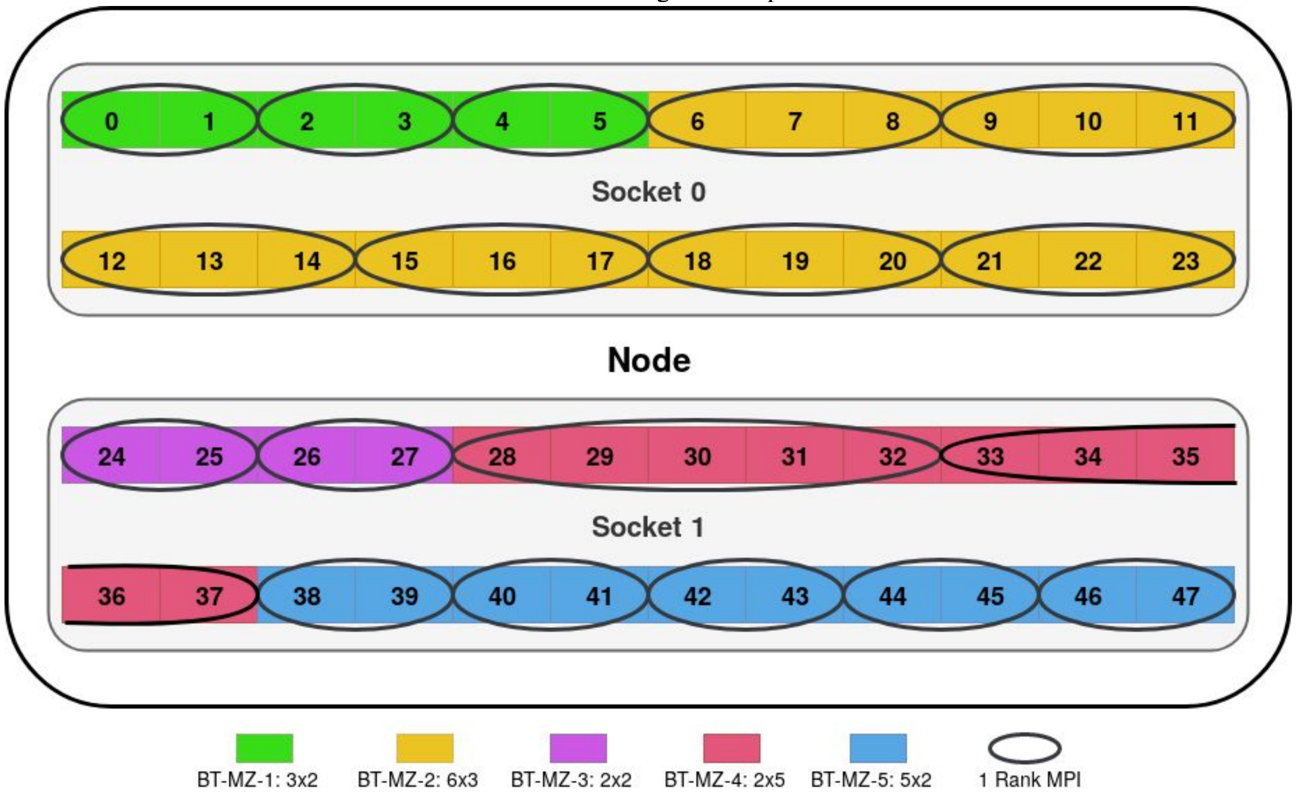


Figure 16: Designed use case for running our experiments. One node contains 5 processes with different resource allocations

In our first scenario "Resource Isolation" none of the applications running within the node share resources. This can be achieved very easily using containers, since each container builds its own software environment and assigns the resource management task to the host system. For example, one can deploy 2 applications with very different software stacks using the same shell script:

```
mpirun -np $X singularity exec $IMAGE_APP1 ./app1
mpirun -np $Y singularity exec $IMAGE_APP2 ./app2
```

This is possible thanks to containers capacity to store in a file (image) the entire environment. This is also achievable without using containers, however, the complexity of the process increases since the user must manage each application requirements using the environment of the host.

The Dynamic Load Balancing library uses the shared memory of the host to handle the cores of each process. Therefore, to ensure the isolation between applications running within containers, it is only necessary to define for each application a separated shared memory file.

In Figure 17, we show the average execution time of 5 runs using the use case depicted in Figure 16. Each BT-MZ process runs inside a Singularity container summoned with *mpirun*. The left side of the plot shows the execution time of our 5 BT-MZ processes without using DLB, that by default ensures resource isolation if the user has bound container cores through the host scheduler or MPI runtime. The right side of the plot shows the same information but applying DLB with resource isolation, that is, each application has a separated shared memory for DLB. In the *x*-axis, the 5 BT-MZ applications appear and in the *y*-axis, their execution time in seconds. It is clear how the execution time of the processes using DLB decreases thanks to the Dynamic Resource Assignment scenario achieved.

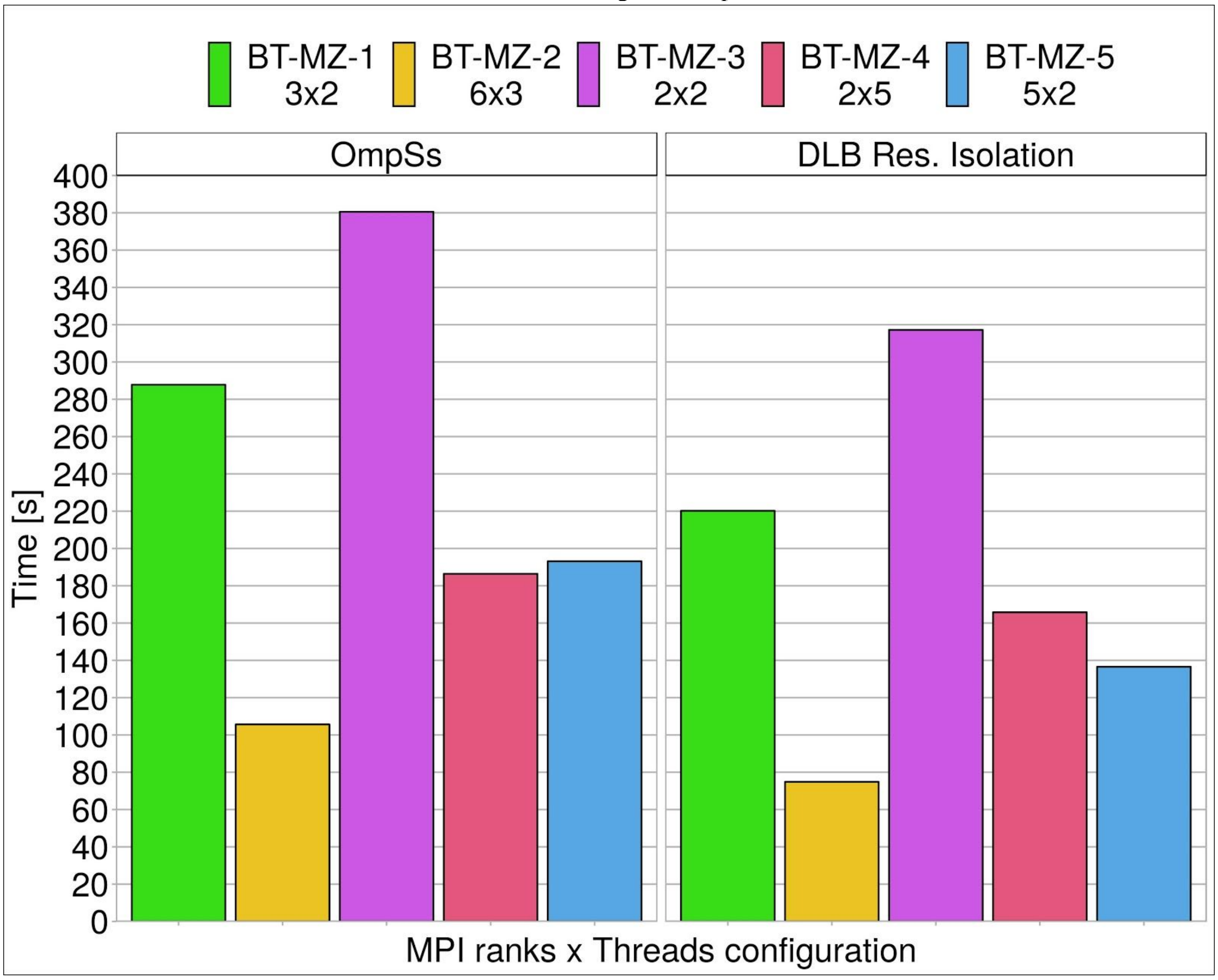


Figure 17: Average execution time of the use case in the Resource Isolation scenario

3.2.3 Scenario 2: Resource Sharing

To ensure resource isolation, we had to define different shared memory files for each application's DLB. Thus, for resource sharing we must configure each application's DLB to operate using the same shared memory. Once this is done, we have the possibility to adapt the resource management according to the application's features of our interests by following the Affinity Aware or Post-Mortem approaches (see Section 3.1.2).

Figure 18 shows the outcomes a user might expect when applying Resource Sharing to its containerized applications. The plot displays the same information as in Figure 17 with the difference that now the scenarios we are testing are: DLB Affinity Aware, DLB Affinity Aware + Post-Mortem, DLB Full Resource Sharing and DLB Full Resource Sharing + Post-Mortem.

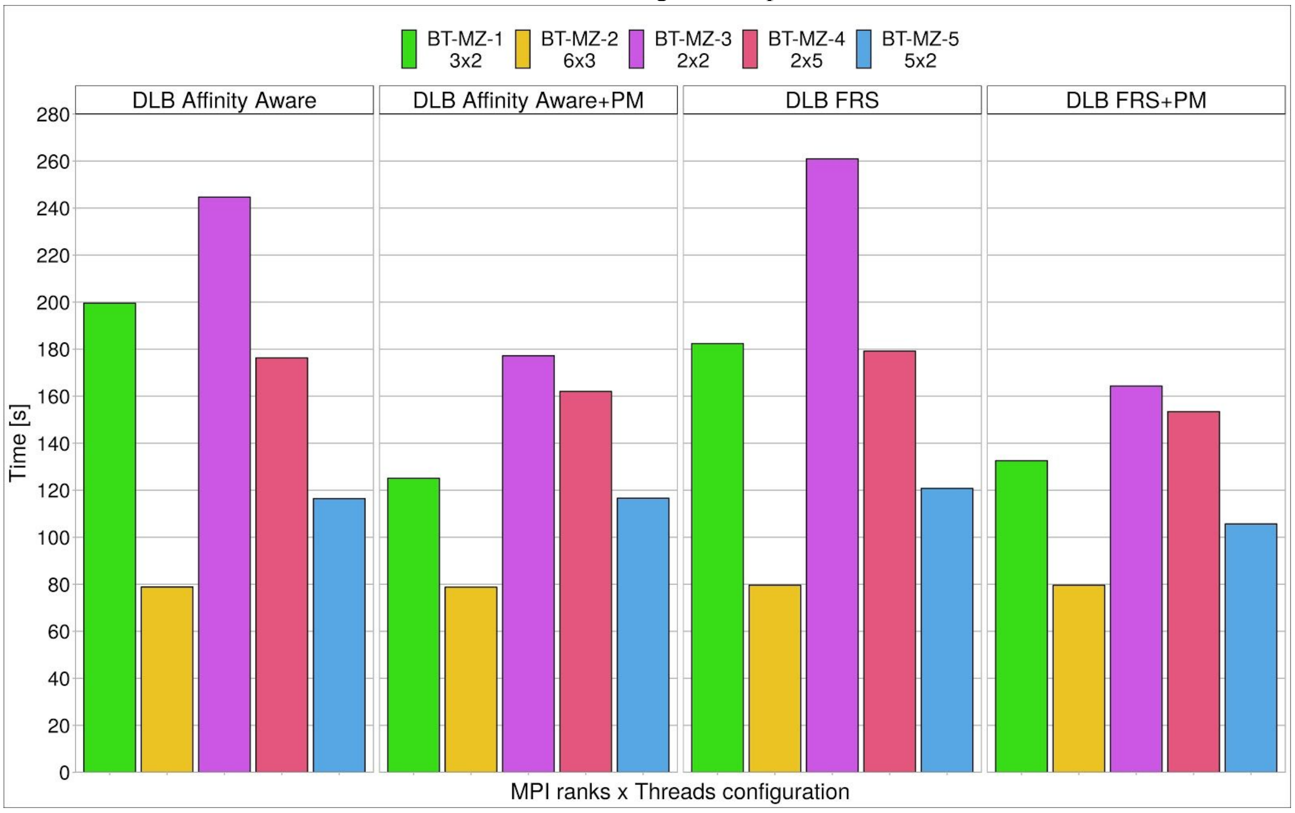


Figure 18: Average execution time of the use case in the Resource Sharing scenario

It is remarkable how the maximum execution time of the use case decreases by adding the Post-Mortem technique, which allows the slowest processes to borrow threads from the fastest when they have finished (in this case, BT-MZ-1 and BT-MZ-3 are the slowest; BT-MZ-2 and BT-MZ-5 the fastest). On the other hand, if we compare Affinity Aware with FRS, we can notice how Affinity Aware approach is a bit better because it does not cause cache misses during thread migrations between sockets. Nevertheless, FRS with Post-Mortem improves Affinity Aware + PM performance. Since in FRS the 3 BT-MZ running in the second socket have access to the resources of the first, and because we apply Post-Mortem, BT-MZ-3, 4 and 5 can leverage the cores from the first socket without the need of returning them to BT-MZ-1 or 2 (because they have already finished), thus minimizing the amount of thread migrations. With all these techniques we are achieving a higher utilisation of the node, therefore reducing the response time of the workload defined with these 5 executions of BT-MZ (they globally end faster).

3.2.4 Scenario 3: TALP

Our last scenario is with TALP, an alternative to the DLB's Lend When Idle algorithm (LeWI, see Section 2.3.1). TALP is interesting to be tested because it has the ability to reassign resources based on computational time metrics, rather than the MPI blocking calls. Again, applying TALP to containerized applications is trivial because, as with Resource Sharing, it only requires from the DLB of all applications to use the same shared memory. Figure 19 presents the average execution time from 5 runs of our use case using TALP, which will act as our baseline for comparison with the rest of scenarios.

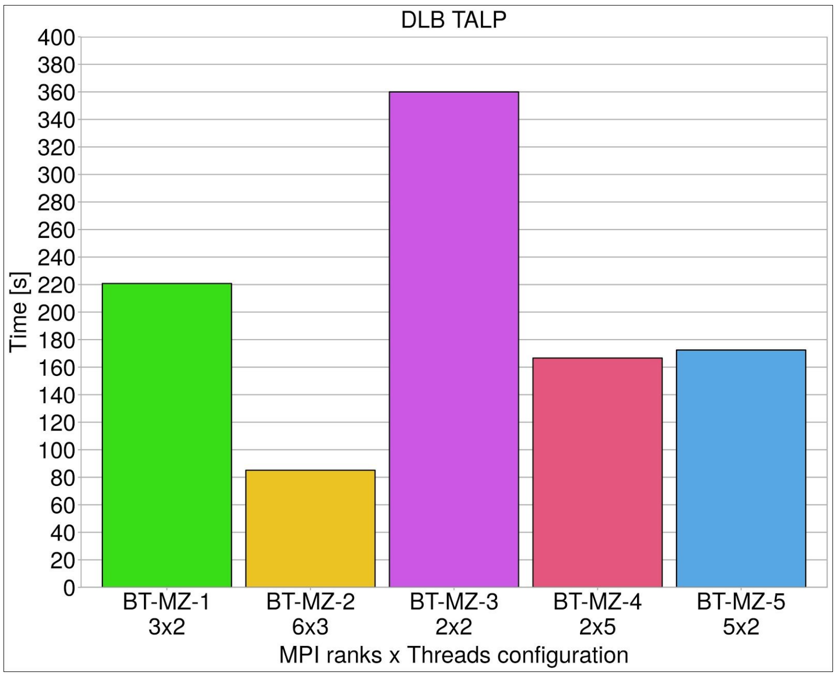


Figure 19: Average execution time of the use case in the TALP scenario

3.2.5 Summary

In order to compare the 3 presented scenarios, we attach Figures 20 and 21. Figure 20 shows the speedup of each scenario with respect to the execution of the ue case from Figure 16 on the host (Native) and using the pure OmpSs version of BT-MZ. The speedup has been obtained as:

$$Speedup_{APP\#}^{scenario} = \frac{\bar{t}_{APP\#}^{scenario}}{\bar{t}_{APP\#}^{Native\ OmpSs}}$$

Where *scenario* can be: DLB Res. Isolation, DLB Affinity Aware, DLB Affinity Aware+PM, DLB FRS, DLB FRS+PM or TALP; and *APP#* can be: BT-MZ-1 (3x2), BT-MZ-2 (6x3), BT-MZ-3 (2x2), BT-MZ-4 (2x5) or BT-MZ-5 (5x2).

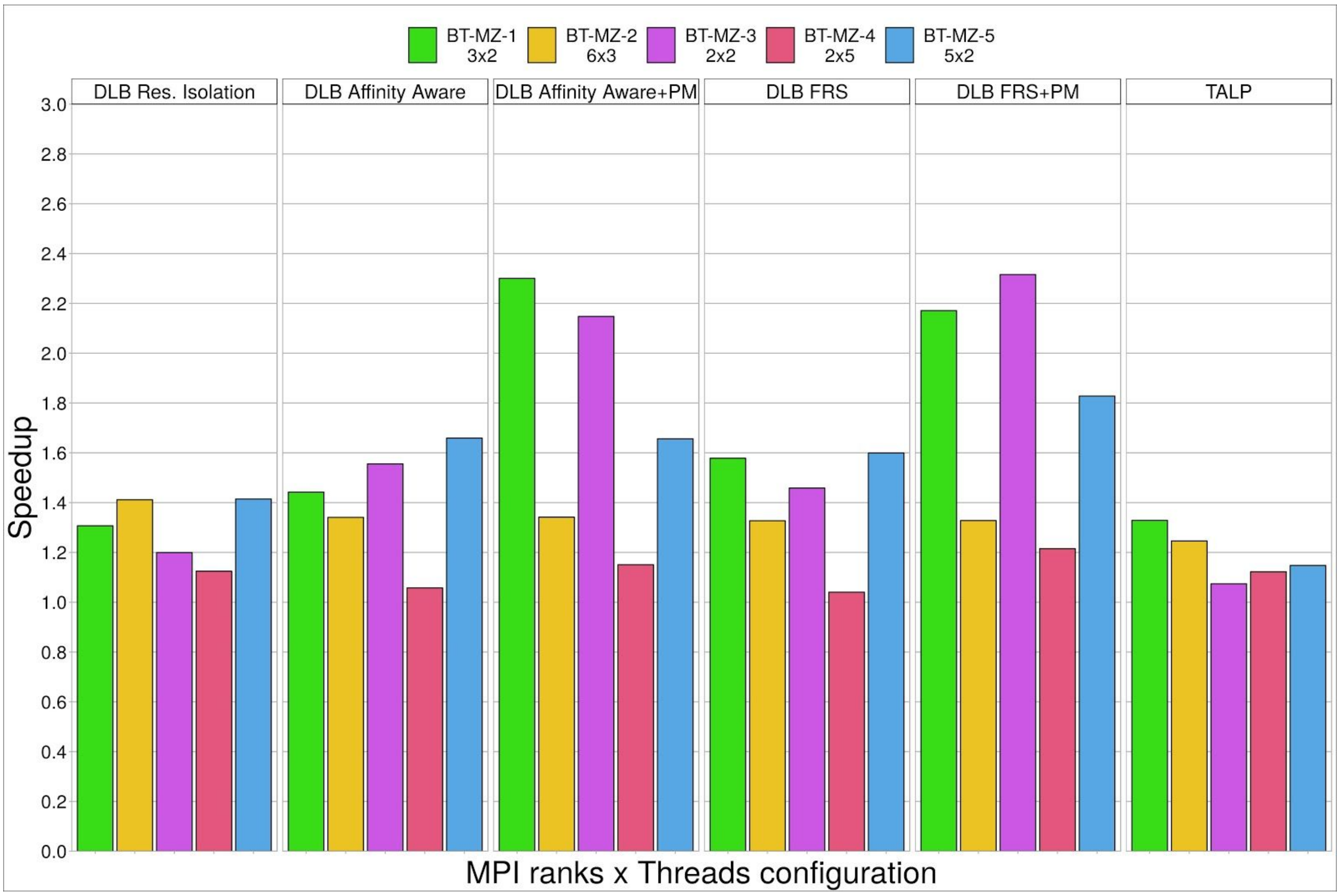


Figure 20: Execution time speedups of each scenario with respect of Native OmpSs

With DLB, our containerized application was able to obtain a maximum speedup of 2,3 (DLB FRS+PM, BT-MZ-3 case) and a minimum of 1,04 (DLB FRS, BT-MZ-4 case). DLB Affinity Aware and DLB FRS are very similar in performance, so the potential gain induced by these 2 scenarios will be dependent on the application's features. The Post-Mortem technique has given such good results because in this use case some processes finish much earlier (BT-MZ-2, BT-MZ-4 and BT-MZ-5) releasing their resources for the slower applications (BT-MZ-1 and BT-MZ-3).

Figure 21 shows the maximum execution time of the defined workload on each scenario, from where we can conclude that DLB Affinity Aware+PM and DLB FRS+PM get the fastest time. Contrary to this, DLB Resource Isolation and TALP achieve only a slightly better performance than the version without any Dynamic Resource Assignment capacities (OmpSs only, first column in the Figure). The DLB Resource Isolation scenario can only manage the allocated cores for its own application. TALP, however, is able to see and manage all the cores available in the node, but the current version of TALP is just a prototype with a very basic resource reassignment algorithm, which explains why the gain is so small. In the future, we plan to implement more advanced techniques to improve this gain.

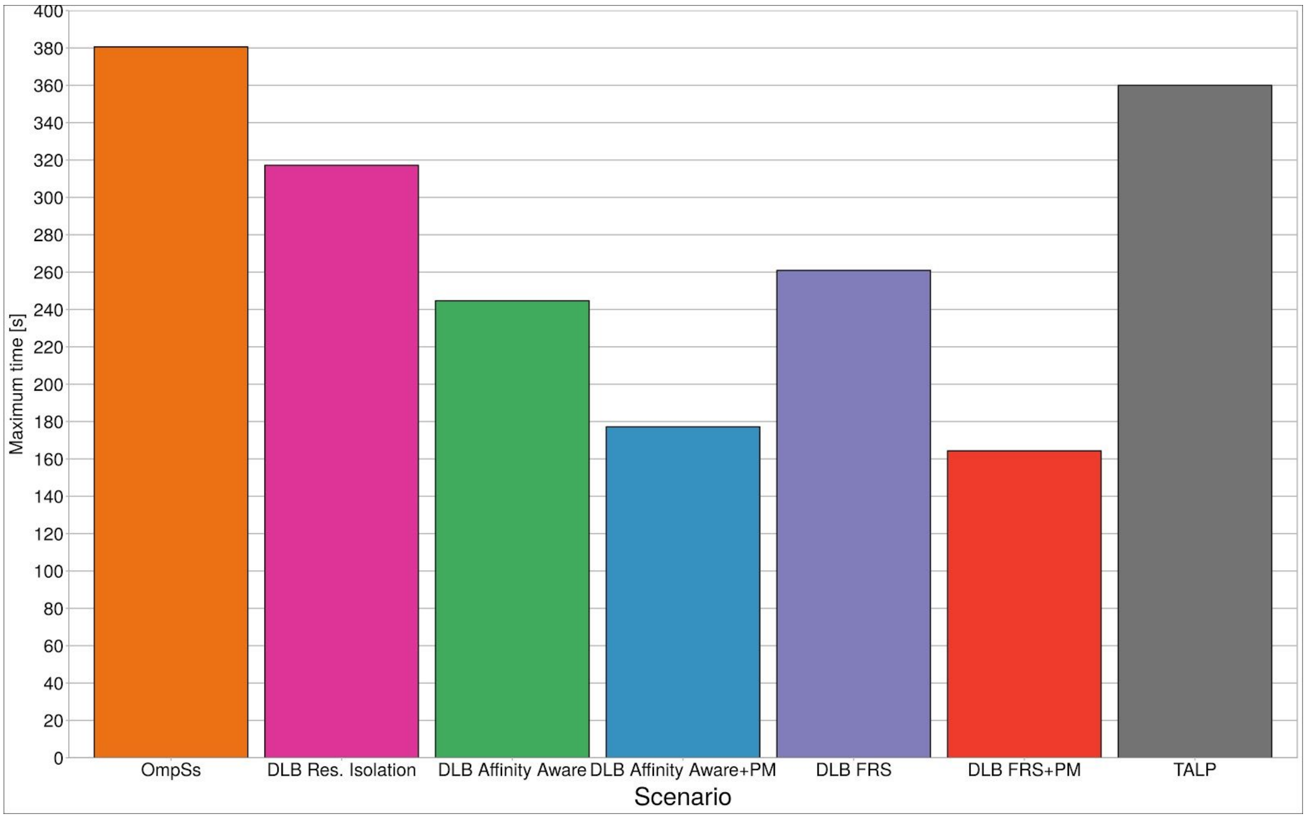


Figure 21: Maximum execution time of each scenario workload

3.3 DLB Integration with Containerized Scientific Applications

The potential of the DLB library to improve the performance of some widely used codes in material science and computational chemistry has been investigated, installing both the application and the DLB library in Singularity containers to be provided to users.

The chosen codes were Quantum Espresso 6.3 [11] and Amber 16 [12], due to the fact that both can be executed as pure MPI and as hybrid MPI+OpenMP also.

Quantum Espresso (QE) is an integrated suite of Open-Source computer codes for electronic-structure calculations and materials modeling at the nanoscale. It is based on density-functional theory, plane waves, and pseudopotentials.

Amber is a suite of biomolecular simulation programs. It began in the late 1970's, and is maintained by an active development community. The term "Amber" refers to two things. First, it is a set of molecular mechanical force fields for the simulation of biomolecules (these force fields are in the public domain, and are used in a variety of simulation programs). Second, it is a package of molecular simulation programs which includes source code and demos.

As it is now very well known that the containerization techniques do not introduce overhead in the code execution, it was decided to start first with tests on bare metal and then with positive results to move to the containers preparation. The results of the tests are reported here below.

3.3.1 Quantum Espresso

QE has been compiled with OpenMPI 2.1.1 on MARCONI (see Section 2.4.2), and the DLB library tool has been installed in the system.

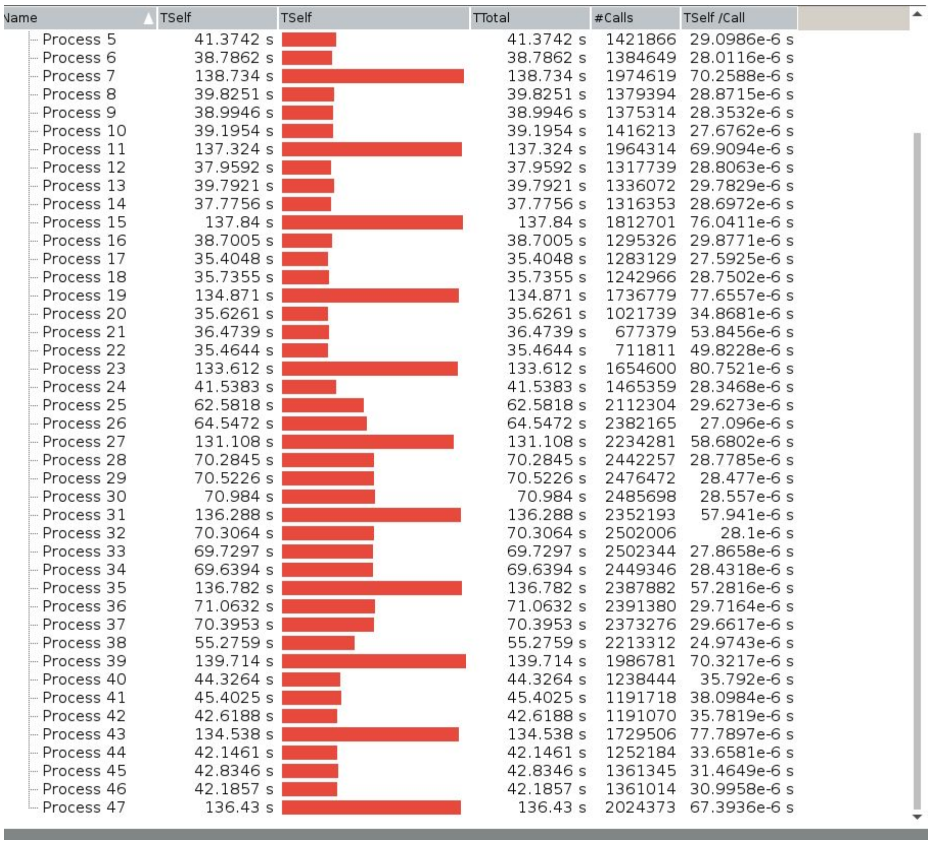
In the test case considered, a Silane (SiH4) molecule has been simulated. Several MPI+OpenMP combinations have been tested, using both a single node and multiple-node executions, varying the number of MPI tasks and OpenMP threads per task, and the total execution time was compared in the runs with and without DLB usage.

The tests show that, unfortunately, DLB was unable to increase the performance of such QE runs, due to the well balanced load among the MPI tasks in the code.

3.3.2 Amber

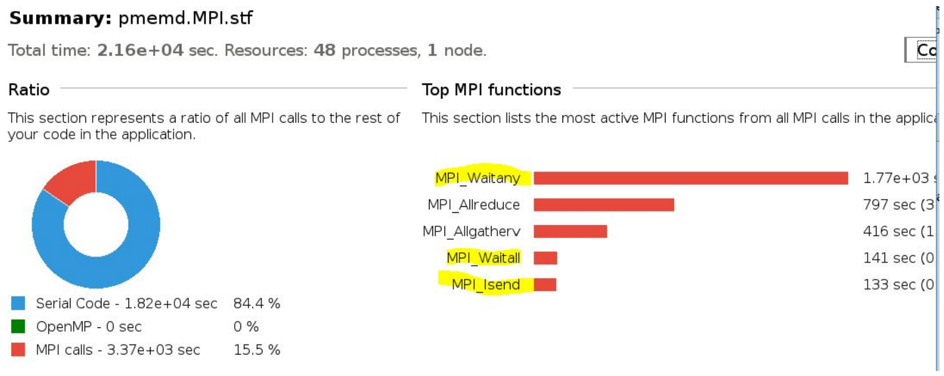
Considering the results obtained with QE, it was decided to replicate the test with another code. Other tools like GROMACS and OpenFOAM appeared to be already balanced as for QE and thus not useful for this specific test.

Based on the results shown in Figure 22 and Figure 23 for an execution with 48 MPI tasks on a SkyLake MARCONI node using Intel Trace Analyzer and Collector (ITAC) tool [13], Amber (compiled with Intel 2017) seemed to be a good candidate to have a boost of performance using DLB, due to its load imbalance between MPI ranks. The test case considered is named as PMEMD, that supports Particle Mesh Ewald simulations, Generalized Born simulations, Isotropic Periodic Sum, ALPB (Analytical Linearized Poisson-Boltzmann) and, as of AMBER v16, gas phase simulations using both the AMBER and CHARMM Force fields.



Name	TSelf	TSelf	TTotal	#Calls	TSelf /Call
Process 5	41.3742 s		41.3742 s	1421866	29.0986e-6 s
Process 6	38.7862 s		38.7862 s	1384649	28.0116e-6 s
Process 7	138.734 s		138.734 s	1974619	70.2588e-6 s
Process 8	39.8251 s		39.8251 s	1379394	28.8715e-6 s
Process 9	38.9946 s		38.9946 s	1375314	28.3532e-6 s
Process 10	39.1954 s		39.1954 s	1416213	27.6762e-6 s
Process 11	137.324 s		137.324 s	1964314	69.9094e-6 s
Process 12	37.9592 s		37.9592 s	1317739	28.8063e-6 s
Process 13	39.7921 s		39.7921 s	1336072	29.7829e-6 s
Process 14	37.7756 s		37.7756 s	1316353	28.6972e-6 s
Process 15	137.84 s		137.84 s	1812701	76.0411e-6 s
Process 16	38.7005 s		38.7005 s	1295326	29.8771e-6 s
Process 17	35.4048 s		35.4048 s	1283129	27.5925e-6 s
Process 18	35.7355 s		35.7355 s	1242966	28.7502e-6 s
Process 19	134.871 s		134.871 s	1736779	77.6557e-6 s
Process 20	35.6261 s		35.6261 s	1021739	34.8681e-6 s
Process 21	36.4739 s		36.4739 s	677379	53.8456e-6 s
Process 22	35.4644 s		35.4644 s	711811	49.8228e-6 s
Process 23	133.612 s		133.612 s	1654600	80.7521e-6 s
Process 24	41.5383 s		41.5383 s	1465359	28.3468e-6 s
Process 25	62.5818 s		62.5818 s	2112304	29.6273e-6 s
Process 26	64.5472 s		64.5472 s	2382165	27.096e-6 s
Process 27	131.108 s		131.108 s	2234281	58.6802e-6 s
Process 28	70.2845 s		70.2845 s	2442257	28.7785e-6 s
Process 29	70.5226 s		70.5226 s	2476472	28.477e-6 s
Process 30	70.984 s		70.984 s	2485698	28.557e-6 s
Process 31	136.288 s		136.288 s	2352193	57.941e-6 s
Process 32	70.3064 s		70.3064 s	2502006	28.1e-6 s
Process 33	69.7297 s		69.7297 s	2502344	27.8658e-6 s
Process 34	69.6394 s		69.6394 s	2449346	28.4318e-6 s
Process 35	136.782 s		136.782 s	2387882	57.2816e-6 s
Process 36	71.0632 s		71.0632 s	2391380	29.7164e-6 s
Process 37	70.3953 s		70.3953 s	2373276	29.6617e-6 s
Process 38	55.2759 s		55.2759 s	2213312	24.9743e-6 s
Process 39	139.714 s		139.714 s	1986781	70.3217e-6 s
Process 40	44.3264 s		44.3264 s	1238444	35.792e-6 s
Process 41	45.4025 s		45.4025 s	1191718	38.0984e-6 s
Process 42	42.6188 s		42.6188 s	1191070	35.7819e-6 s
Process 43	134.538 s		134.538 s	1729506	77.7897e-6 s
Process 44	42.1461 s		42.1461 s	1252184	33.6581e-6 s
Process 45	42.8346 s		42.8346 s	1361345	31.4649e-6 s
Process 46	42.1857 s		42.1857 s	1361014	30.9958e-6 s
Process 47	136.43 s		136.43 s	2024373	67.3936e-6 s

Figure 22: The execution time for each MPI task in a PMEMD simulation in Amber.



Summary: pmemd.MPI.stf

Total time: **2.16e+04** sec. Resources: **48** processes, **1** node.

Ratio

This section represents a ratio of all MPI calls to the rest of your code in the application.

🟦 Serial Code - 1.82e+04 sec	84.4 %
🟩 OpenMP - 0 sec	0 %
🟥 MPI calls - 3.37e+03 sec	15.5 %

Top MPI functions

This section lists the most active MPI functions from all MPI calls in the applica

MPI_Waitany	1.77e+03 s
MPI_Allreduce	797 sec (3
MPI_Allgatherv	416 sec (1
MPI_Waitall	141 sec (0
MPI_Isend	133 sec (0

Figure 23: The ITAC summary graph of the PMEMD Amber simulation.

Amber was compiled with Intel 2017 and the DLB library tool has been installed on the same system.

On a single MARCONI SkyLake node a run with 8 MPI tasks and 6 OpenMP threads was done, with and without the DLB library. The total time of the run was considered as performance indicator, but no performance improvements were noted.

At this point, further investigations are needed to understand better the features of the load balancing of MPI processes in the code (possibly with the use of Extrae and Paraver performance analysis tools). These additional analyses will be performed, even if the task is formally completed, in parallel with the benchmarking activities and reported at the next progress reporting (M18).

4. Conclusions

The conclusions of the first deliverable produced by the JRA (D12.1 - Container-as-a-service analysis report) have been very valuable for the execution of this work. We have been able to choose Singularity as the best container application solution and arrange our testbeds accordingly for experimentation. In addition to this, we have also designed, prepared and evaluated containers resource isolation and sharing capabilities. For this, we ported the NAS Parallel Benchmark BT-MZ to the OmpSs programming model and implemented a prototype of TALP, the new module of BSC's Dynamic Load Balancing Library (DLB). Last but not least, we attempted to integrate DLB with some scientific applications (Quantum Espresso and Amber) for subsequent containerization and testing with Singularity in the benchmarking activities (Deliverable 12.6 Benchmarking results, due M30).

Singularity appeared to be a highly malleable container implementation to control resource allocation and workload management. Because Singularity does not exercise *cgroups* capabilities, it delegates the resource allocation and scheduling of tasks to the host's resource manager, for example SLURM, the MPI runtime or others. Both resource isolation and resource sharing among containers are easily achievable depending on how the user or host has spawned the containers as they would be normal processes of the system. However, be aware that, by default, Singularity containers mount some directories from the host's filesystem hierarchy: */tmp* or */dev/shm*, which might be two host directories whose contents are visible and accessible for all containers. Thus, if users require the isolation of their application's temporary files, they must manually modify where these kinds of directories are mounted within the container. This can be easily done with the *--bind*|*-B* option.

We have demonstrated that DLB library is a powerful tool to speedup hybrid applications and to administer hardware resources between containers, which has helped us to demonstrate the Dynamic Resource Assignment capabilities promised in this task of HPC-Europa3. Two or more containerized applications leveraging DLB are able to be run within the same computational node concurrently while ensuring their software stack isolation and experimenting a performance boost thanks to DLB management. In addition, DLB provides several resource management strategies to better adapt to the application's features and user needs: Resource Isolation, if the user does not want to share resources; Full Resource Sharing or Affinity Aware, to share cores but minimizing thread migration overheads due to the node's hardware organization; Post-Mortem, to make available one container CPUs when it has finished its execution; and TALP, to distribute the resources based on computational metrics rather than MPI blocking calls.

A more extended demonstration of DLB usage has been introduced with its integration with Quantum Espresso and Amber in MARCONI supercomputer. Although the preliminary results do not show any performance improvements, we have demonstrated that DLB integration with scientific applications is easily achievable and Amber potentially could take advantage of it given the load imbalance Amber possesses between MPI processes.

5. References

[1] M. Garcia, J. Labarta, J. Corbalan, ‘Hints to improve automatic load balancing with LeWI for hybrid applications’, *Elsevier Journal of Parallel and Distributed Computing*, vol. 74, no. 9, pp. 2781–2794, Sep. 2014 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731514000926. [Accessed: 26-Mar-2019]

[2] N.Wilson, R. Sirvent, M. Renato, O. Rudyy, G. Muscianisi, Y. Cardenas, R. Laurikainen, A-M. Saren, D. Dellis, ‘Container-as-a-service analysis report’, D12.1, 2019 [Online]. Available: http://doi.org/10.23728/b2share.c93bf40018f74f04ab8db4636f55f143. [Accessed: 18-Mar-2019]

[3] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, and J. Planas, ‘Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures’, *Parallel Process. Lett.*, vol. 21, no. 02, pp. 173–193, Jun. 2011 [Online]. Available: https://www.researchgate.net/publication/220439810_Ompss_a_Proposal_for_Programming_Heterogeneous_Multi-Core_Architectures. [Accessed: 05-Apr-2019]

[4] ‘The OmpSs Programming Model | Programming Models @ BSC’. [Online]. Available: https://pm.bsc.es/ompss. [Accessed: 05-Apr-2019]

[5] ‘Mercurium | Programming Models @ BSC’. [Online]. Available: https://pm.bsc.es/mcxx. [Accessed: 19-Mar-2019]

[6] ‘Nanos++ | Programming Models @ BSC’. [Online]. Available: https://pm.bsc.es/nanox. [Accessed: 19-Mar-2019]

[7] E. Gabriel *et al.*, ‘Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation’, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2004, pp. 97–104 [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-30218-6_19. [Accessed: 26-Mar-2019]

[8] ‘Dynamic Load Balancing | Programming Models @ BSC’. [Online]. Available: https://pm.bsc.es/dlb. [Accessed: 05-Apr-2019]

[9] 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, p. 41 [Online]. Available: http://dl.acm.org/citation.cfm?id=3229710.3229752. [Accessed: 04-Apr-2019]

[10] ‘Support Knowledge Center @ BSC-CNS’. [Online]. Available: https://www.bsc.es/user-support/mn4.php. [Accessed: 19-Mar-2019]

[11] ‘QUANTUMESPRESSO - QUANTUMESPRESSO’. [Online]. Available: https://www.quantum-espresso.org/. [Accessed: 29-Mar-2019]

[12] ‘The Amber Molecular Dynamics Package’. [Online]. Available: http://ambermd.org/. [Accessed: 29-Mar-2019]

[13] ‘Intel® Trace Analyzer and Collector’, *Intel*, 12-Feb-2019. [Online]. Available: https://software.intel.com/en-us/trace-analyzer. [Accessed: 05-Apr-2019]

Appendix

Resource Isolation deployment with DLB

The *mpirun* commands used to run from the host the Resource Isolation test have the following format:

```
export NX_ARGS=”--enable-dlb --enable-block”
#BT-MZ 3x2
mpirun   -np   3   --rankfile   rank3x2.txt   -x   OMP_NUM_THREADS=2   -x
DLB_ARGS=”--lewi  --shm-key=bt-mz1”  -x  NX_ARGS  singularity  exec  $IMAGE
bt-mz.C.3
#BT-MZ 6x3
mpirun   -np   6   --rankfile   rank6x3.txt   -x   OMP_NUM_THREADS=3   -x
DLB_ARGS=”--lewi  --shm-key=bt-mz2”  -x  NX_ARGS  singularity  exec  $IMAGE
bt-mz.C.6
#BT-MZ 2x2
mpirun   -np   2   --rankfile   rank2x2.txt   -x   OMP_NUM_THREADS=2   -x
DLB_ARGS=”--lewi  --shm-key=bt-mz3”  -x  NX_ARGS  singularity  exec  $IMAGE
bt-mz.C.2
#BT-MZ 2x5
mpirun   -np   2   --rankfile   rank2x5.txt   -x   OMP_NUM_THREADS=5   -x
DLB_ARGS=”--lewi   --shm-key=bt-mz4”   -x   NX_ARGSsingularity   exec   $IMAGE
bt-mz.C.2
#BT-MZ 5x2
mpirun   -np   5   --rankfile   rank5x2.txt   -x   OMP_NUM_THREADS=2   -x
DLB_ARGS=”--lewi  --shm-key=bt-mz5”  -x  NX_ARGS  singularity  exec  $IMAGE
bt-mz.C.5
```

Resource Sharing deployment with DLB

The *mpirun* commands used to run from the host the Resource Sharing test have the following format.

- Affinity Aware

```
export DLB_ARGS=”--lewi --lewi-affinity=nearby-only”
export NX_ARGS=”--enable-dlb --enable-block”
#BT-MZ 3x2
mpirun -np 3 --rankfile rank3x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.3
#BT-MZ 6x3
mpirun -np 6 --rankfile rank6x3.txt -x OMP_NUM_THREADS=3 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.6
#BT-MZ 2x2
mpirun -np 2 --rankfile rank2x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 2x5
```

```
mpirun -np 2 --rankfile rank2x5.txt -x OMP_NUM_THREADS=5 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 5x2
mpirun -np 5 --rankfile rank5x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.5
```

- Affinity Aware + Post-Mortem

```
export DLB_ARGS="--lewi --lewi-affinity=nearby-only
--debug-opts=lend-post-mortem"
NX_ARGS="--enable-dlb --enable-block"
#BT-MZ 3x2
mpirun -np 3 --rankfile rank3x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.3
#BT-MZ 6x3
mpirun -np 6 --rankfile rank6x3.txt -x OMP_NUM_THREADS=3 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.6
#BT-MZ 2x2
mpirun -np 2 --rankfile rank2x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 2x5
mpirun -np 2 --rankfile rank2x5.txt -x OMP_NUM_THREADS=5 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 5x2
mpirun -np 5 --rankfile rank5x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.5
```

- FRS (Full Resource Sharing)

```
export DLB_ARGS="--lewi"
export NX_ARGS="--enable-dlb --enable-block"
#BT-MZ 3x2
mpirun -np 3 --rankfile rank3x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.3
#BT-MZ 6x3
mpirun -np 6 --rankfile rank6x3.txt -x OMP_NUM_THREADS=3 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.6
#BT-MZ 2x2
mpirun -np 2 --rankfile rank2x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 2x5
mpirun -np 2 --rankfile rank2x5.txt -x OMP_NUM_THREADS=5 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 5x2
mpirun -np 5 --rankfile rank5x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.5
```

- FRS+PM (Full Resource Sharing + Post-Mortem)

```
export DLB_ARGS=”--lewi --debug-opts=lend-post-mortem”
export NX_ARGS=”--enable-dlb --enable-block
#BT-MZ 3x2
mpirun -np 3 --rankfile rank3x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.3
#BT-MZ 6x3
mpirun -np 6 --rankfile rank6x3.txt -x OMP_NUM_THREADS=3 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.6
#BT-MZ 2x2
mpirun -np 2 --rankfile rank2x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 2x5
mpirun -np 2 --rankfile rank2x5.txt -x OMP_NUM_THREADS=5 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 5x2
mpirun -np 5 --rankfile rank5x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.5
```

TALP deployment

The *mpirun* commands used to run from the host the TALP test have the following format:

```
export DLB_ARGS=”--talp --drom”
export NX_ARGS=”--enable-dlb”
export SINGULARITYENV_LD_PRELOAD=$DLB_HOME/lib/libdlb_mpif.so
#BT-MZ 3x2
mpirun -np 3 --rankfile rank3x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.3
#BT-MZ 6x3
mpirun -np 6 --rankfile rank6x3.txt -x OMP_NUM_THREADS=3 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.6
#BT-MZ 2x2
mpirun -np 2 --rankfile rank2x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 2x5
mpirun -np 2 --rankfile rank2x5.txt -x OMP_NUM_THREADS=5 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.2
#BT-MZ 5x2
mpirun -np 5 --rankfile rank5x2.txt -x OMP_NUM_THREADS=2 -x DLB_ARGS -x
NX_ARGS singularity exec $IMAGE bt-mz.C.5
```