The Journal of Supercomputing
https://doi.org/10.1007/s11227-020-03175-4

Improving utilization of heterogeneous clusters

Esteban Stafford[1] · **José Luis Bosque**[1]

Abstract
Datacenters often agglutinate sets of nodes with different capabilities, leading to a sub-optimal resource utilization. One of the best ways of improving utilization is to balance the load by taking into account the heterogeneity of these clusters. This article presents a novel way of expressing computational capacity, more adequate for heterogeneous clusters, and also advocates for task migration in order to further improve the utilization. The experimental evaluation shows that both proposals are advantageous and allow improving the utilization of heterogeneous clusters and reducing the makespan to 16.7% and 17.1%, respectively.

Keywords Heterogeneous clusters · Utilization · Load index · Task migration

1 Introduction

The fast evolution of computer architecture together with the way datacenters acquire nodes, in time spaced renovation campaigns, is causing that at a given time, datacenters have several groups of nodes with different configurations and capabilities. The typical way in which administrators manage these heterogeneous clusters is to organize nodes with equal configurations in separate partitions, so that each partition is homogeneous. Then, the users are left to decide to which partition they will submit their jobs, a situation that can lead to inefficiencies. Since users tend to submit to the partitions with the newest nodes, these get overused while other queues with older or worse nodes are not exploited enough and found idle for significant periods of time [1]. Keeping nodes running regardless of their occupation causes a waste of energy. But even if the nodes are powered down when found idle, the cluster is not being utilized to its full potential. Moreover, it is known that power cycles affect the reliability of the nodes and increase maintenance costs [2, 3].

✉ Esteban Stafford
esteban.stafford@unican.es

José Luis Bosque
joseluis.bosque@unican.es

[1] Department of Computer Science and Electronics, University of Cantabria, Santander, Spain

Published online: 27 January 2020

One of the most important challenges in successfully leveraging the performance of these large systems is to consistently distribute the load among the available resources, proportionally to their computing capacity [4–6]. This has been traditionally attempted through dynamic algorithms that are able to adapt to the varying requirements of the workloads. However, since the execution times of the latter are not constrained to short bursts, achieving a perfect balance often requires relocating tasks which are already in execution. Task migration is a costly operation, and therefore, it must only be undertaken when its benefit compensates its cost [7, 8]. Achieving a performance gain through task migration is easier in a heterogeneous cluster, since it is possible to find faster nodes to send tasks to [9]. Furthermore, the fact that execution times of scientific and big-data applications currently executed in HPC datacenters are well in the range of hours, or even days, makes the overhead of migration less relevant [10].

This article presents two proposals that improve the utilization of heterogeneous clusters. The first is a new way to express the computing capacity available on the nodes that takes heterogeneity into account. This new *load index* considers the performance and the load of each node of the cluster, to assign tasks to the nodes that give the highest computing capacity at a given time. The second is to leverage task migration to take advantage of the nodes with the highest computing capacity whenever possible.

The experimental evaluation included in this article suggests two main conclusions. First, the new load index improves the cluster utilization. In these experiments, the reduction of the makespan and the wait time was 16.7% and 27.1%, respectively. Second, despite the cost of task migration, it has a positive impact on the utilization. The experimentation showed a timespan reduction of 17.1%.

The remainder of this article is structured as follows: Section 2 presents the proposals themselves, followed by Sect. 3 that gives some important details of how the proposals were implemented in a load balancing algorithm. Section 4 outlines the methodology employed throughout the experimentation. Section 5 continues with the experimental results and empirical evaluation of the proposals. Section 6 deals with related work found in the literature. Finally, Sect. 7 summarizes some concluding thoughts and future lines of work.

2 Proposals

This section explains in detail the main two proposals of this article: one concerns the load index of a heterogeneous cluster and the other analyzes when task migration can be profitable.

2.1 Capacity load index

A load balancing algorithm must base its decisions on up-to-date information of the computational capabilities and workload of the cluster nodes. Thus, nodes must periodically collect local information and calculate their *load index*. Since the index

must be calculated frequently, the process of doing so must be very efficient. Furthermore, the choice of load index has a huge impact on load balancing efficiency [11].

Typically, modern resource managers use a load index that does not consider the different capabilities of the cores in the cluster. Therefore, they do not allocate more than one task to a single core, which makes perfect sense in a homogeneous cluster, assuming serial tasks. In a heterogeneous one, cores will have very different computing capabilities, and sometimes, these differences can be as high as an order of magnitude. Therefore, it is necessary to find a load index that takes into account the characteristics of the cores, fostering a better usage of the heterogeneous cluster resources.

This paper proposes a new load index called *capacity index*. Using this, each node quantifies the computing power it will provide to a new task. Taking as an example a single node with four cores, the capacity index it offers when it is idle is 1, meaning that it has at least one free core for a new task to run on. Additional concurrent tasks will also see index 1 as they find idle cores, but the fifth task will not be offered the same. Since the node only has four cores, a fifth task will cause a sharing of the cores. The capacity index in this case would be $\frac{4}{5} = 0.8$, meaning that each task will obtain 80% of a core. The capacity index of the node will continue to diminish as the number of running tasks increases.

In a heterogeneous cluster, it is necessary to adapt the capacity index to the computing performance offered by each node in order to have values that are consistent across the cluster. Considering two idle nodes where one has double the computing power of the other, the index of the slowest node must be half of that of the fastest. Therefore, the index considers the relative performance of each node as $R_i = \frac{T_{\min}}{T_i}$, where T_i is the execution time of a given benchmark on node i and $T_{\min}$ is the same for the fastest node in the cluster. With this, the capacity index I_i is formulated as follows:

$$I_i = \begin{cases} R_i & T_i \leq C_i - 1 \\ \frac{R_i C_i}{T_i + 1} & T_i > C_i - 1, \frac{R_i C_i}{T_i + 1} \geq R_{\min} \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

where C_i and T_i are the number of cores and number of tasks currently executing in node i. $R_{\min}$ is the relative performance of the slowest node in the cluster. The first term of the equation models the node when it is underutilized, and there are idle cores, and the index is simply the relative performance of the node. The second term models when there is at least one task per core, and then, capacity offered to a new task will be the relative performance of all the cores combined $R_i C_i$ shared among the running tasks T_i plus the new one. The third term ensures that a new task never receives less computing power than that of the slowest core. By giving an index value of zero, it signifies that the node is saturated and can not accommodate a new task.

To illustrate how the capacity index works, Fig. 1 shows a synthetic evaluation. This considers four quad-core nodes with different relative performances. The

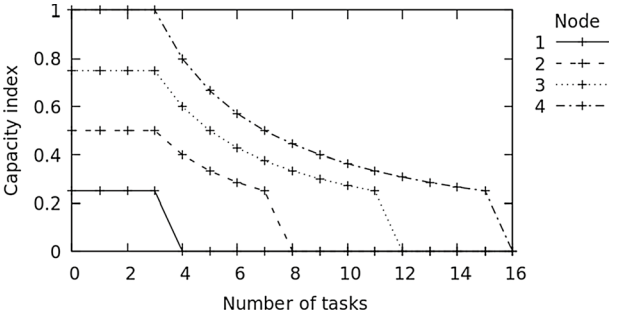

Fig. 1 Evolution of the index with increasing tasks on nodes with relative performances 0.25, 0.5, 0.75 and 1

figure presents the evolution of the index as the number of running tasks increases. It is noteworthy that the index allows more powerful nodes to have more tasks than cores, as they might still offer a computing power higher than slower nodes, even if these are completely idle. Note also that no node offers an index lower than 0.25 which is the relative performance of the slowest node, and that this node will not execute more than four tasks, meaning that the granularity of the resource allocation is equal to the performance of one core of the slowest node.

The multiple advantages of this index affect the cluster in two major situations. First, if the cluster is not fully loaded, then a new task is assigned to the node that offers the highest computing power, independently of the number of running tasks in this node. Consequently, despite the oversubscription, this new task will execute faster than if it had been sent to a slower node with less load. Second, if the cluster is full, it can cope with more simultaneous task executions, reducing the wait time. Using the example cluster from Fig. 1, a traditional resource manager would concurrently execute 16 tasks, while with the capacity index, it would execute 40 tasks. Furthermore, the index will give the same computing resources to all tasks running at a given time. From the users' perspective, this appears more fair, as they would get similar response time from nodes of different capabilities. And, since the cluster admits more tasks, the wait time is reduced.

2.2 Task migration

The capacity index is able to assign equal resources to the tasks that are submitted to the cluster. However, the dynamic nature of the workload of the cluster can lead to imbalance situations that can only be resolved once the tasks are running.

Figure 2 shows the effect of migration on the execution of 44 equal tasks, on the cluster described in Sect. 2.1. Each horizontal line corresponds to a task execution; tasks of the same color are executed by the same node. Gray lines show initial task submissions, and gray arrows stand for task migrations. In both experiments, the 40 first tasks are executed immediately, while 4 are queued. Unfortunately, the slowest node is the one that finishes its tasks first, thus receiving the 4 queued tasks, but the faster nodes finish soon after and remain idle. Thanks

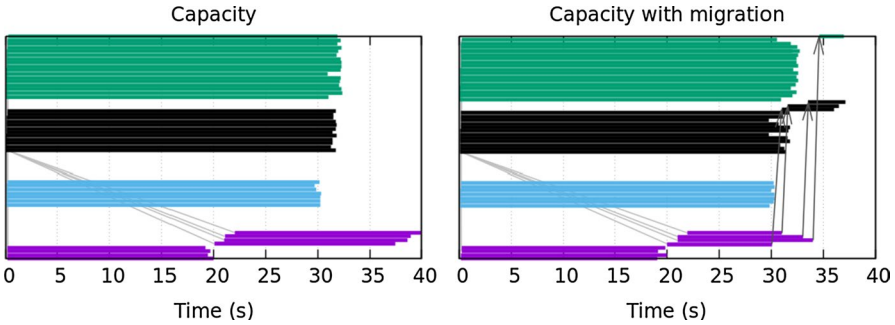

Fig. 2 Comparison of the capacity index with and without migration

to task migration, it is possible to mitigate this unfortunate situation by moving tasks from the slower node to the fastest ones and thus improving the global execution time.

Migration consists of suspending the execution of a task in one node and restarting it in the same state in another node. This is done in the following three steps:

- *Checkpoint* suspends the execution of the task and saves its state in a checkpoint file with all the necessary information to restart its execution later.
- *Transfer* moves the checkpoint file to the destination node.
- *Restart* reads the checkpoint file, restores the task state in memory and resumes its execution.

Naturally, these steps take time, so migration will introduce significant overhead in the execution of the tasks. Therefore, it is necessary that the gain brought by migration compensates the overhead caused by these steps. To this aim, this paper presents a model to establish when it is worth performing a task migration. It attempts to guess when and between which nodes will a migration will be advantageous.

First, it considers the remaining execution time of a task $T_{\text{remaining}}$, which is in principle unknown. But this model assumes that all tasks are in the middle of their execution [12] at the time of migration. Then, a new execution time can be forecasted, considering that the remaining time is equal to the current execution time. This will be proportionally reduced by the capacity index difference of the sender node I_{sender} and a potentially faster receiver node I_{receiver}. And finally, adding the overhead caused by the migration steps enumerated above $T_{\text{migration}}$, the model considers migration beneficial only when the following expression holds.

$$T_{\text{remaining}} > T_{\text{remaining}} \frac{I_{\text{receiver}}}{I_{\text{sender}}} + T_{\text{migration}} \tag{2}$$

Consequently, migrations can occur if the estimated execution time on the sender node is longer than the improved execution time due to the higher capacity of the receiving node plus the migration overhead.

3 Implementation

In order to validate the previous proposals, a dynamic load balancing algorithm has been created. This section provides some important details of this implementation.

3.1 Node state information

An important part of a load balancing algorithm is to periodically calculate the capacity index explained in Sect. 2.1. The period is a configurable parameter, and for this article, it has been set to one second.

Nodes need to keep up-to-date state information of all other nodes. Then, each change in the load index of a node triggers a set of messages informing the rest of the cluster. To avoid flooding the network with messages, only index changes above a configurable threshold trigger message sending. When these messages are received, each node updates an *index vector* that contains the capacity index of all the nodes in the cluster. This is useful to select nodes appropriate for load balancing operations. Since all nodes keep updated information about the global state of the cluster, this is considered a global algorithm.

3.2 Load balancing operations

When tasks are submitted by the users, the algorithm aims to select the best node for execution. For this, it considers all the nodes in the cluster, including itself. By considering the nodes with nonzero index from the index vector, a node with the highest index is selected as the receiver. It can happen that sender and receiver are the same node. If there is no eligible receiver node, the task is added to the task queue of that node. This happens when the load index of all the nodes is zero, and the cluster is considered saturated.

In this saturated state, the algorithm waits for a node to send a nonzero index value. It then sends tasks from the queue to this node until it becomes full again or the queue is empty. The execution operation is cheaper in terms of time than the migration, since the latter requires checkpointing, transmitting and restarting a task. Therefore, migrations can only take place when the queues of all the nodes are empty.

Nodes with empty queues send messages to all other nodes, voting for migration. Migration is allowed only when the number of votes is equal to the number of nodes. This process is canceled when a node receives a new submission and, as a consequence, sends a negative vote.

When migrations are allowed and a node shares a nonzero index, other nodes with less index consider their running tasks and evaluate the possibility of migrating one, using the model presented in Sect. 2.2. Each node can only be involved in one task migration at a given time, regardless of its role being sender or receiver.

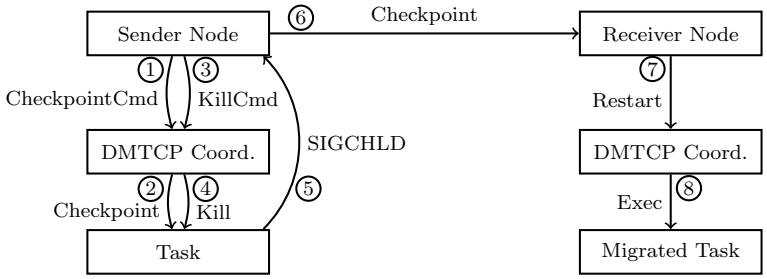

Fig. 3 Task migration with DMTCP

3.3 Checkpointing tool

In this article, the Distributed MultiThreaded CheckPointing (DMTCP)[1] [13] has been selected as checkpointing tool. It executes exclusively in userspace, and it does not need to add any module or configure the kernel in any way.

In a checkpoint file, DMTCP stores the following data: the relationships to parent and child processes, the memory segments of the process, the registers of all threads, the state of the file descriptors such as open files, pipes and signal handlers or even sockets. The tool allows the generated files to be compressed, to reduce their size. However, the time required for compression is also significant.

The DMTCP tool operates around the concept of a coordinator process. This is started when the task is first executed. When performing a checkpoint is required, a separate command-line tool communicates with the coordinator, that is listening for commands on a TCP port.

In the load balancing algorithm presented in this article, the sequence of events that occur during a migration is shown in Fig. 3. When a sender node agrees with a receiver node that a migration can take place, it sends a checkpoint command (1), followed by a kill command (3) to the coordinator of the task. This, in turn, triggers the actual checkpointing of the task (2), followed by the end of the process (4). Since the task is a child of the sender process, it receives the SIGCHLD signal (5) and therefore knows that the checkpoint is ready. Next, it sends the checkpoint file to the receiver (6), who has been engaged waiting since both nodes agreed to carry out the migration. Once the receiver has a copy of the checkpoint file, it can execute the DMTCP tool to restart the task (7, 8).

4 Methodology

The empirical evaluation of the ideas proposed in this article has been carried out in a cluster with 20 computational nodes. Each has one Intel Core i5-7500 CPU with 4 cores, 8GB main memory. The heterogeneity of the cluster is due to the different

[1] The code is available at https://github.com/dmtcp.

frequencies of the nodes. There are five nodes with each of the following frequencies: 3.4GHz, 2.5GHz, 1.5GHz and 800MHz. Each node has a SATA hard disk, on which the checkpoint files are written. The latter are transferred to the receiving nodes through a GigaBit Ethernet network.

The tasks used for the experiments are executions of a serial matrix multiplication program, which are compute intensive, although there is a fair amount of memory access due to the size of the matrices.

The figures of merit used to compare the different experiments are the makespan and the wait time. Makespan is the time difference between the starting of the first task and the end of the last. The wait time is the sum of the time each task waits in the queue.

To assess the advantages of the proposals, their performance will be compared to a version of the load balancing algorithm that assigns one task per core and does not consider task migration. In the experiment results, this algorithm is called *Max-Tasks*. It is assumed that this is the typical behavior of current cluster resource managers.

5 Evaluation

This section presents an empirical evaluation of the different proposals of this article. It first quantifies the benefits of using the capacity index in contrast to traditional task-per-core schedulers. Second, it addresses the improvement on the algorithm given by task migration.

5.1 Capacity load index

In the first experiment, several batches of 1000 identical tasks are submitted to the cluster. Due to the heterogeneity of the cluster, the tasks will not necessarily have the same execution time in all the nodes.

Figure 4 shows two representative batch submissions: one with Max-Tasks and the other with capacity. As in Fig. 2, task executions are represented by horizontal lines, and the thickness of the lines gives the idea of how many tasks execute simultaneously. The x-axis represents the time and the y-axis the nodes of the cluster.

As can be seen in the figure, the makespan is reduced when the capacity index is used. This is due to the fact that since the beginning of the batch, all but the slowest nodes are executing more tasks than in Max-Tasks. Paying closer attention to the indices, it can be seen that the Max-Tasks will never execute more than 80 tasks simultaneously, one per core. In contrast, the capacity index allows executing up to 200 simultaneous tasks. Note that the number of simultaneous tasks on each node is proportional to its computing power.

The combined analysis of 10 batches for each index shows an average reduction of the makespan of 16.7%. Additionally, the variability of the makespan is also reduced. Due to the increased number of simultaneous executions, the tasks are less time held in the queue and consequently, the wait time is reduced to 27.1%.

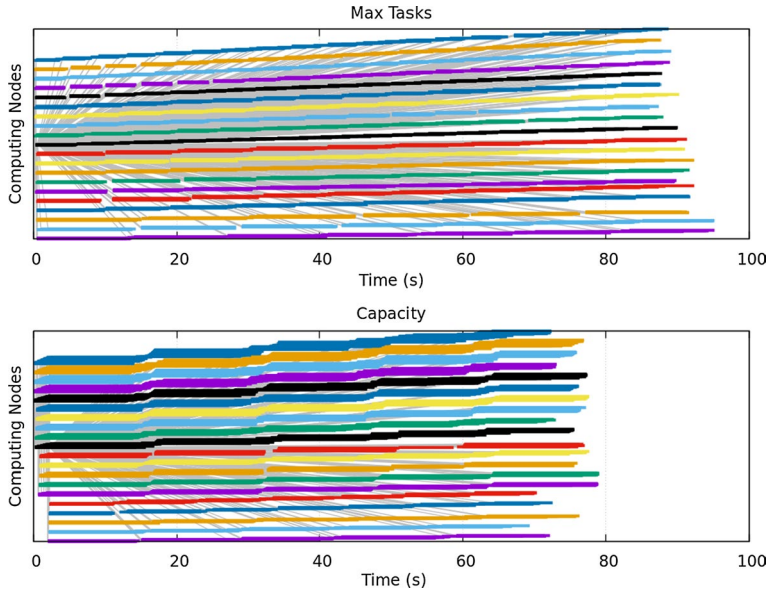

Fig. 4 Comparison of the task time distribution with Max-Tasks and capacity indices

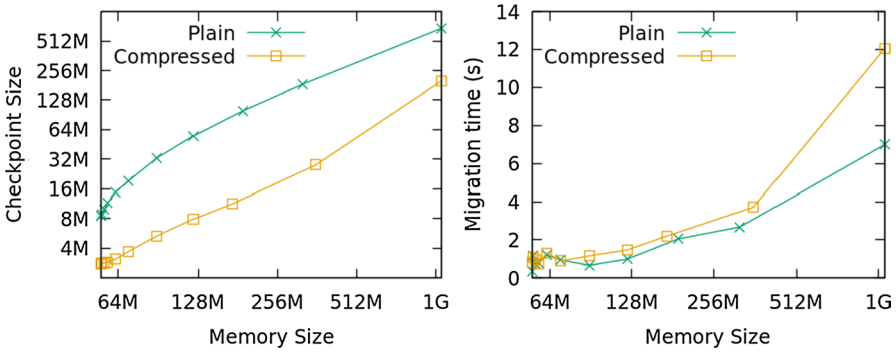

Fig. 5 Migration overhead with respect to the task memory footprint

5.2 Task migration

It has been said that task migration has a cost in terms of time. The migration model presented in Sect. 2.2 needs to be able to quantify this overhead. This depends on the size of the checkpoint file, which in turn depends on the amount of memory used by the task. To this aim, the following experiment makes a sweep of executions with different memory requirements in order to extract a relation between that and the migration time. Figure 5 shows, on the left, the size of the checkpoint file compared to the memory footprint of the task and, on the right of the figure, the total migration time, including the checkpoint, transfer and restart

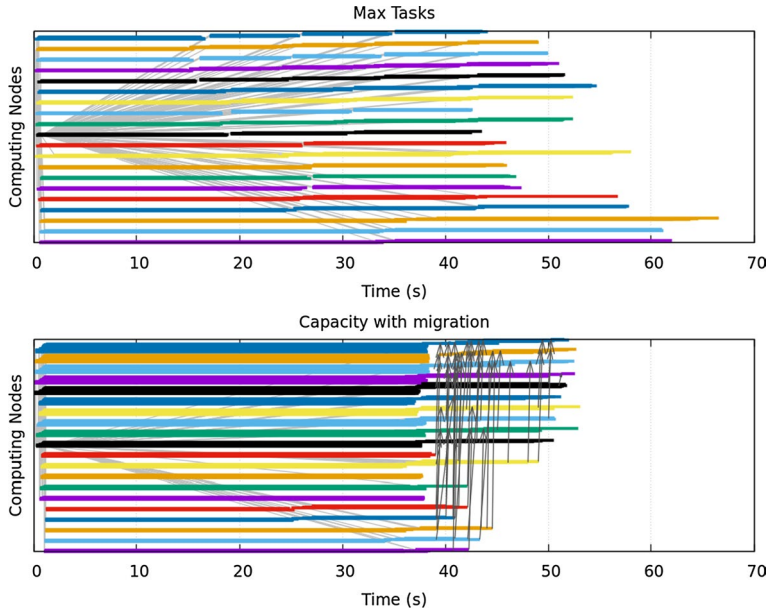

Fig. 6 Evaluation of the capacity index with migration

steps. Since the DMTCP tool allows compressing the checkpoint file, the figure compares both options.

The first observation that can be made is that the migration overhead is significant, even with tasks with small memory footprints, with a minimum time of 1s. Also, although the size of the checkpoint file is greatly reduced with compression, it causes a notable increase in the overhead. As a consequence, compression is not used in this article. A linear regression of these data was made and introduced in the migration model.

The last experiment consists of runs of batches of 250 identical tasks. The memory footprint of the tasks is around 64MB; thus, the checkpoint file will be slightly over 8MB and migration overhead around 1.5s, as can be determined in Fig. 5.

Representative instances of this experiment are shown in Fig. 6. In both cases, the experiments start by executing tasks until the cluster is saturated, the remaining tasks are held in a queue until some nodes become available again, and they can be executed. Thanks to the migration, once the queue is empty, tasks are migrated to nodes offering higher capacity, whereas without migration, these tasks remain in the chosen nodes until the end.

A statistical analysis of 20 experiments shows that the makespan is reduced to 17.1% in spite of the overhead caused by the migrations.

As an overall conclusion, both proposals provide benefits in the management of heterogeneous clusters in different situations. The capacity index favors the increased utilization of the computational resources, bringing fairness to the user experience, while the migration allows using more powerful nodes as they become available when the queue is empty and tasks are executing in slower nodes.

6 Related work

Scheduling and load balancing are two of the most important aspects allowing to squeeze performance and optimizing the energy consumption of current parallel and distributed systems. The importance of this topic is reflected in a large number of publications and several surveys as [14–18].

Heterogeneous clusters are composed of a set of computational nodes with very different computing capabilities. Therefore, to extract all the performance of these systems and minimize their energy consumption it is very important to take into account this heterogeneity, and distribute the workload proportionally to the computing capacity of each node. Thus, in the previous work [19], a dynamic, distributed, global and non-preemptive load balancing algorithm for heterogeneous clusters is proposed. This paper extends the previous one with a new load index and the ability to migrate tasks.

A profile-based load balancing algorithm for heterogeneous accelerator clusters (PLB-HAC) is proposed in [20]. It constructs a performance curve model for each resource at runtime and continuously adapts it to changing conditions. It dispatches execution blocks asynchronously, preventing synchronization overheads and other idle periods due to imbalances.

An heuristic dynamic algorithm to dynamically balance the workload between different parallel processes in iterative algorithms is presented in [15]. It is based on an arbitrary objective function that can be changed, and a specific implementation to minimize energy consumption is presented.

A different approach to the scheduling problem is to use a knowledge-based system (KBS) comprised of an individual set of if-then rules that depend on certain parameters [21–23]. In this way [21] presents a hybrid genetic fuzzy system (HGFS) that combines both a fuzzy and a non-fuzzy sets of rules. The fuzzy part is learned by means of a genetic-based machine leaning multi-objective evolutionary algorithm. The purpose of using a HGFS is to achieve better results in both rule readability and efficiency compared to the static KBS. On the other hand, in [22], through a forecast of the future workload and according to a utility function, an optimization problem is solved. Finally, in [23], a two-stage holistic optimization mechanism is proposed, composed of a stage that logically optimizes the resources and another that optimizes hardware allocation by leveraging a genetic fuzzy system. The model finds optimal trade-offs among different objectives. Instead of using a multi-objective learning algorithm that produces a set of rules, our solution is a distributed algorithm that operates first on a node allocation based on the capacity index of the nodes, and second on the possibility of task migration, should this improve the performance of the cluster as a whole.

Load balancing is a topic also addressed from the context of cloud computing. For instance, [2] proposes a server consolidation strategy that attempts to fully utilize nodes of a data warehouse, avoiding a sparse allocation of virtual machines to servers. Also [3] presents an automated server provisioning system that aims to meet workload demand while minimizing energy consumption in data warehouses by deciding what nodes can be turned off or must be powered on

and when. In our case, the big-data and HPC loads we consider differ in behavior to virtual machines, while scientific applications strive to minimize time-to-solution; the latter have prolonged execution times, responding to user requests and subject to SLAs.

Checkpoint/restart is a very useful technique to migrate running tasks between computational nodes in distributed systems avoiding loss of already developed work. This has been used both to balance the workload of computational nodes and for fault tolerance algorithms in HPC clusters [24]. [25] presents a multi-objective load balancing algorithm based on extremal optimization. It uses three objectives relevant to load balancing: computational load balance of processors, the volume of inter-processor communication and task migration metrics. Extremal optimization is used to find task migrations which dynamically improve processor load balance in a distributed system. [26] provides an extensive analysis of the performance, energy and I/O costs associated with a wide array of checkpointing policies. The results show ample room for achieving high-quality energy/performance trade-offs when using methods that exploit characteristics of real-world failures.

7 Conclusions

In summary, this article proposes two ideas that can be combined in a distributed load balancing algorithm that improves the utilization of heterogeneous clusters. The first proposal is a new load index, called capacity, that expresses the amount of computing power each node can offer a new task. This index takes into account all the different types of nodes in the heterogeneous cluster, as well as the load in each node. Thus, it is aware of the fact that executing a task in a fast node which is already executing more tasks than cores, can be more advantageous than running it in a slower node that is idle.

The second proposal studies the possibility of using task migration. It presents a model of when it is worth performing a migration, by taking into account the capacity index of the receiver node, the time penalty of performing the migration and the foreseen remaining execution time. This model only triggers a migration if the cost of migration is compensated by the higher capacity of the receiving node.

The experimental evaluation of these two ideas shows that the capacity load index can reduce the makespan to 16.7% compared to traditional task-to-core allocation schemes. The capacity index can also execute more tasks simultaneously, meaning that the queue time of the tasks is also reduced. The experiments with task migration show that there is a positive impact in using this technique, showing makespan reductions of 17.1%.

In the future, a wider range of experiments will be performed, with tasks of different time and memory requirements, or even parallel tasks, leading to a study of the effect of contention in memory access due to finite memory bandwidth.

Acknowledgements This work has been supported by the Spanish Science and Technology Commission under contracts TIN2016-76635-C2-2-R and TIN2016-81840-REDT (CAPAP-H6 network) and the European HiPEAC Network of Excellence.

References

1. Beltrán M, Guzmán A, Bosque JL (2006) Dealing with heterogeneity in load balancing algorithms. In: 5th International Symposium on Parallel and Distributed Computing (ISPDC 2006), 6–9 July 2006, Timisoara, Romania, pp 123–132
2. Deng W, Liu F, Jin H, Liao X, Liu H, Chen L (2012) Lifetime or energy: consolidating servers with reliability control in virtualized cloud datacenters. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings, pp 18–25
3. Guenter B, Jain N, Williams C (2011) Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In: 2011 Proceedings IEEE INFOCOM, pp 1332–1340
4. Alam T, Raza Z (2016) An adaptive threshold based hybrid load balancing scheme with sender and receiver initiated approach using random information exchange. Concurr Comput: Pract Exp 28(9):2729–2746
5. Bosque JL, Robles OD, Pastor L, Rodríguez A (2006) Parallel CBIR implementations with load balancing algorithms. J Parallel Distrib Comput 66(8):1062–1075
6. Martínez J, Almeida F, Garzón E, Acosta A, Blanco V (2011) Adaptive load balancing of iterative computation on heterogeneous nondedicated systems. J Supercomput 58(3):385–393
7. Belgaum MR, Soomro S, Alansari Z, Musa S, Alam M, Su'ud MM (2019) Load balancing with preemptive and non-preemptive task scheduling in cloud computing. In: CoRR, arXiv :abs/1905.03094
8. Ungureanu V, Melamed B, Katehakis M (2008) Effective load balancing for cluster-based servers employing job preemption. Perform Eval 65(8):606–622
9. Gerofi B, Ishikawa Y (2011) Workload adaptive checkpoint scheduling of virtual machine replication. In: 2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing, pp 204–213
10. Bartuschat Dominik, Rüde Ulrich (2014) Parallel multiphysics simulations of charged particles in microfluidic flows. J Comput Sci 8:1–19
11. Bosque JL, Toharia P, Robles OD, Pastor L (2013) A load index and load balancing algorithm for heterogeneous clusters. J Supercomput 65(3):1104–1113
12. Harchol-Balter M, Downey AB (1997) Exploiting process lifetime distributions for dynamic load balancing. ACM Trans Comput Syst 15(3):253–285
13. Ansel J, Arya K, Cooperman G (2009) DMTCP: transparent checkpointing for cluster computations and the desktop. In: IEEE International Symposium on Parallel and Distributed Processing, Rome, pp 1–12
14. Jiang Y (2016) A survey of task allocation and load balancing in distributed systems. IEEE Trans Parallel Distrib Syst 27(2):585–599
15. Cabrera Pérez A, Acosta A, Almeida F, Blanco Pérez V (2019) A heuristic technique to improve energy efficiency with dynamic load balancing. J Supercomput 75(3):1610–1624
16. Laredo JLJ, Guinand F, Olivier D, Bouvry P (2017) Load balancing at the edge of chaos: how self-organized criticality can lead to energy-efficient computing. IEEE Trans Parallel Distrib Syst 28(2):517–529
17. Sheetlani J, Khanna MS (2016) Classification of task partitioning and load balancing strategies in distributed parallel computing systems. Int J Comput Syst 3(5):371–375
18. Mishra P, Singh S, Mishra M, Agarwal S (2013) Comparative analysis of various evolutionary techniques of load balancing: a review. Int J Comput Appl 63(15):8–13
19. Bosque JL, Toharia P, Robles OD, Pastor L (2013) A load balancing algorithm for heterogeneous clusters. J Supercomput 65(3):1104–1113
20. Sant'Ana L, Cordeiro D, de Camargo RY (2019) PLB-HAC: dynamic load-balancing for heterogeneous accelerator clusters. In: Euro-Par 2019: 25th International Conference on Parallel and Distributed Computing, Proceedings, pp 197–209
21. Cocaña Fernández A, Ranilla J, Sánchez L (2015) Energy-efficient allocation of computing node slots in HPC clusters through parameter learning and hybrid genetic fuzzy system modeling. J Supercomput 71(3):1163–1174
22. Cocaña-Fernández A, Sánchez L, Ranilla J (2016) Leveraging a predictive model of the workload for intelligent slot allocation schemes in energy-efficient HPC clusters. Eng Appl Artif Intell 48:95–105

23. Cocaña-Fernández A, San José Guiote E, Sánchez L, Ranilla J (2019) Eco-efficient resource management in hpc clusters through computer intelligence techniques. Energies 12:2129
24. Kohl N, Hötzer J, Schornbaum F, Bauer M, Godenschwager C, Köstler H, Nestler B, Rüde U (2019) A scalable and extensible checkpointing scheme for massively parallel simulations. Int J High Perform Comput Appl 33(4):571–589
25. De Falco I, Laskowski E, Olejnik R, Scafuri U, Tarantino E, Tudruj M (2018) Effective processor load balancing using multi-objective parallel extremal optimization. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO '18, pp 1292–1299
26. El-Sayed N, Schroeder B (2018) Understanding practical tradeoffs in HPC checkpoint-scheduling policies. IEEE Trans Dependable Secure Comput 15(2):336–350