Noname manuscript No.
(will be inserted by the editor)

A Load Index and Load Balancing Algorithm for Heterogeneous Clusters

Jose Luis Bosque · Pablo Toharia · Oscar D. Robles · Luis Pastor

the date of receipt and acceptance should be inserted later

Abstract This paper presents a load balancing algorithm specifically designed for heterogeneous clusters, composed of nodes with different computational capabilities. The method is based on a new index which takes into consideration two levels of processors heterogeneity: the number of cores per node and the computational power of each core. The experimental results show that this index allows achieving balanced workload distributions even on those clusters where heterogeneity can not be neglected.

Keywords Heterogeneous computing, load balancing, load index.

1 Introduction

High performance computing (HPC) has significantly evolved during the last decades, making it possible to integrate up to hundreds of thousands cores into the currently available petaflop machines [2]. It seems clear that the roadmap to the next generation of exascale computers passes through the integration of large distributed and heterogeneous systems, in which their computational nodes are in turn composed of multi-core processors with a large number of cores and hardware accelerators.

The design and implementation of efficient parallel applications for heterogeneous systems is still a very important challenge. The heterogeneity of these systems is a relevant distortion factor that prevents load balancing models and algorithms designed for homogeneous systems to achieve accurate and consistent results [3]. In particular, heterogeneity has a deep impact on workload

Jose Luis Bosque
Dpto. de Electrónica y Computadores
Universidad de Cantabria. E-mail: joseluis.bosque@unican.es

Pablo Toharia · Oscar D. Robles · Luis Pastor
Dpto. de Arquitectura y Tecnología de Computadores y Ciencia de la Computación e Inteligencia Artificial
Universidad Rey Juan Carlos E-mail: {pablo.toharia,oscardavid.robles,luis.pastor}@urjc.es

distribution, where ignoring that different nodes might have different computing capabilities usually results in poorly balanced systems.

A key aspect for workload distribution is characterizing properly the computing capabilities of each of the system nodes. For this, it is necessary to consider both static factors, such as the number of cores per node and their computational power, and dynamic factors such as the number of tasks being executed on each node and their computational requirements.

This paper presents a new load balancing algorithm for heterogenous clusters based on a *load acceptance index* that takes into consideration both static and dynamic factors, and which can be used for determining the status of nodes in terms of their availability for accepting additional workload at every single instant. The proposed index considers the two levels of heterogeneity that can be found in a multi-core system: the number of cores per node and the computing power of each node; the first factor is discrete while the second one is continuous. In order to test this new parameter, a distibuted, global, emitter-initiated load balancing algorithm has been implemented; an interesting feature of this algorithm is that it can also turn itself off whenever the whole system gets overloaded.

There are other approaches for computing load indexes that can be found in the bibliography. [6] presents EMAS, an Evolutionary Mobile Agent System. It proposes a load index, *Server_Utilization_Status*, based on 4 different parameters that are merged, although these parameters have different nature, being measured in different units. Also, [7] uses the *Current real load* of each node. This index is based on parameters such as CPU occupancy rate, memory usage, system I/O usage and network bandwidth occupancy rate, whose influence is altered depending on the different services offered by the cluster. Another similar approach that merges parameters of different nature in a single load index can be found in [9]. A very different approach is proposed in [4]; this paper attempts to improve the accuracy of host load predictions by applying a neural network predictor. Regarding load balancing, a couple of specific algorithms for balancing the workload of iterative algorithms on heterogeneous multiprocessors have also been proposed [5,8]. Their main goal consists in designing a strategy that allows to dynamically analyze the computational power of the processors involved in the heterogeneous system and to determine the computational burden that must be located at each processor.

2 Load Balancing Algorithm Design

This paper presents a dynamic, distributed, global and non-preemptive load balancing approach. It is dynamic, because the assignment of a task to an specific node is performed in run time; the task is then completely executed in the assigned node, without any kind of task migration. Also, it is distributed, since every node in the cluster makes its own decisions based on local stored information: Once a node decides to perform a load balancing operation, it selects a partner among all of the available nodes in the cluster.

As suggested in [10], most workload balancing methods can be usually decomposed in several stages. The algorithm presented here can be decomposed in the following four stages:

- *State measurement phase*. Every node evaluates its current state in order to find busy and idle resources. In this work, it is controlled by the **Load process**.
- *Information rule*. During this phase, some state information is exchanged among the cluster nodes. In this work this is done both by the **Load** and **Global processes**.
- *Initiation rule*: This stage determines whenever a load balancing process should be initiated. It is part of the **Balance process**.
- *Load balancing operation*: In this phase, the load balancing operation is effectively carried out. It is also assumed by the **Balance process**. It includes:
  a. *Localization rule*, that aims at selecting a suitable node for performing the load balancing operation.
  b. *Distribution rule*, that decides how the load is shared among the involved nodes.

Finally, the algorithm does not present any overhead if the nodes are naturally balanced, turning itself automatically off also whenever the system is globally underloaded or overloaded.

Figure 1 shows how the load balancing algorithm has been implemented with three different processes. Design and implementation details can be found in the following sections.

2.1 Measuring the state of a node

Determining the node's load is essential, because decisions such as starting the local execution of a new task or launching a new load balancing operation are taken based on the nodes' workload. The first stage is devoted to this purpose, gathering the local information needed for estimating each node's workload state. For this purpose, it is necessary to know the number of cores of each node and their computational power, as well as the number of tasks being executed in that node.

Regarding the state of a node, there are two different possibilities:

- The number of tasks is lower than the number of cores in the node. Therefore, there are some free cores and this node can accept more tasks, so it can be considered as a *recipient* node.
- The number of tasks is larger than the number of cores. In this case, the load acceptance index (*load index*, from now onwards) can be computed for that node using the following expression:

$$Load_{Index} = \frac{P_i}{P_{max}} \cdot \frac{\#Cores}{\#Tasks + 1} \tag{1}$$

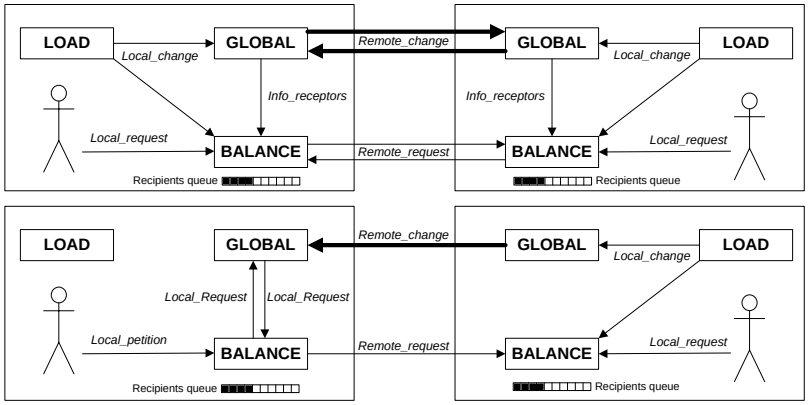


Fig. 1 Structure of Processes and communication. Thick lines represent broadcast operations while thin lines show point to point communications.

where P_i is the computational power of the node i, defined as the amount of work finished during a unit time span in that node, measured in basic operations per time unit. This value is normalized by using P_{max}, the power of the node with greatest value. $\#Cores$ and $\#Tasks$ take the corresponding values for that specific node and time instant.

This *load index* measures how available a node is, with respect to the most powerful node, for receiving a new task. In this way, while a node has free cores will be able to receive new tasks, and the load index will depend only on the relation between P_i and P_{max}. It must be noticed that higher figures indicate higher availability.

It can be seen that the load index is dynamic, and thus it has to be computed periodically at predefined time instants by the **Load process**. This interval should be long enough to minimize the overhead introduced in the system by performing this operation, but it should also be short enough to keep the load index value updated.

The proposed method uses a discrete set of node states that reflect how loaded nodes are. These states involve a discretization of the load index which is very useful in order to minimizing the exchange of global information, and simplifying load balancing decisions. Three states determine the behaviour of a node:

- *Recipient State*: A node is in *recipient* state when the number of its running tasks is lower than the number of its cores, or when its load index value is larger than the threshold to change from *neutral* to *recipient*, meaning that it is available for sharing some of the other nodes' workload. In this case, the node can assume at least one more task, either local or remote.
- *Neutral State*: In this case all the cores in the node are executing at least one task. In this state the node can assume new local tasks but it will reject all remote requests. This state corresponds to intermediate values of the load index.

– *Emitter State*: A node is in *emitter* state when its load index is below the threshold for changing from *neutral* to *emitter*. This means that the node has many more tasks than cores, and therefore it can not accept any more tasks.

A node will change its state whenever a new measurement of the load index crosses a status threshold. As mentioned before, the **Load process** checks periodically each node's workload and computes its *load index*. When the newly computed value implies a change of state, it is propagated to the **Global** and **Balance** processes, sending to the latter the maximum number of tasks the node can accept, just in case of being involved in a balancing operation.

2.2 Global Information

The **Global process** keeps updated information from all of the nodes, together with a list of all of the possible candidates that could be involved in balancing operations. This means that the **Global process** must be able to receive messages indicating state changes from the local instance of the **Load process**. These changes are then broadcasted by the local instance of the **Global process** to every single node in the system (see Figure 1). It can be seen that the method can be labeled as global, having all nodes keeping updated information about the system's global state.

Load balancing operations can only take place from an *emitter* to a *recipient* node. Hence, only changes to or from the *recipient* state are significant enough to be communicated. In consequence, the number of messages is significantly reduced.

Each node maintains a *recipient-queue* with the information received from the other nodes. When a node becomes *recipient*, it broadcasts a message to all the nodes in the cluster so that each of them will place it at the end of its queue. On the other hand, when a *recipient* node changes its state to *neutral* or *emitter*, it broadcasts a message too and all the cluster nodes will discard it from their recipient-queues.

2.3 Load Balancing Operations

The **Balance process** determines when a new load balancing operation has to begin, and also performs the operations needed for carrying it out. Since the approach used here does not interrupt any task execution, the decision about in which node a task will be executed can only be taken in the actual moment of beginning the task, (the method follows an *emitter-initiated* rule to make this decision). Therefore, the decision about initiating a load balancing operation is *completely local* to the *emitter*. Also, it has to be noticed that messages are exchanged only whenever a load balancing operation is started; no load balancing operations means no messages, which helps keeping the communication overhead low.

Figure 1 shows the algorithm architecture, with the different processes and communication protocols. All of them help keeping the global information

updated for performing load balancing operations. Also, it can be seen how the arrival of requests increases the waiting process queue until there is a change of state; please notice how point to point and broadcast messages are represented.

The initiation rule must be evaluated every time a new task has to be launched. At this moment, the state of the node is checked, in order to find out if it can accept the local execution of the new task or if the search of a better candidate for the execution of the task should be initiated.

A node can only accept the local execution of a new task if it is in *recipient* or *neutral* state. Therefore, only if the node is an *emitter*, a load balancing operation is initiated. Once the decision of starting a load balancing operation is made, the following two steps must be performed:

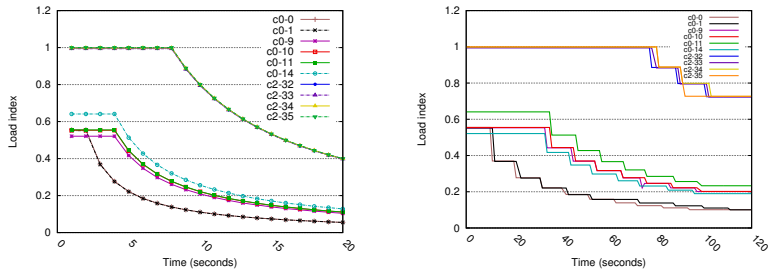
Partner localization. This is a completely local operation, since each *emitter* node looks for partners just in its own *recipients* queue. The selection of a specific partner is performed as follows: first a few nodes are randomly selected, their current load index is requested (see *Remote_request* messages in Figure 1) and a sorted list based on their load index is created. Then, the less loaded node is selected first. If the request is rejected, the next node in the list will be requested and so on. This strategy reduces the communications overhead since the load indexes do not have to be constantly updated. It is actually a trade-off between the goals of keeping message traffic low and selecting the least loaded nodes at each specific time instant. This method also minimizes the possibility of a node being selected by several nodes at the same moment because of being in a prominent position in the queue.

Load Distribution. The last step of a load balancing operation is to decide how much workload should be sent to the *recipient* node. A good distribution rule will try to give each node an amount of workload proportional to its current computational power. After a load balancing operation, the more similar the load indexes are, the better the load distribution is. As explained, the *emitter* node sends to the first node in the sorted list as much workload as needed to force a change in its load index from *recipient* to *neutral* state. The operation is repeated with the second *recipient* from the list. This process finishes when there is no more workload to send or there are no more *recipients*.

3 Experimental Results

A number of experiments have been run on a heterogeneous cluster composed of 10 nodes. As elsewhere [11], the NASA Parallel Benchmark [1] has been adopted to measure the computational power P_i of each node. From this 10 nodes included in the system used in the experiments, 4 nodes have 8 cores; 4 of them have 4 cores, and two of them have 2 cores. Table 1 summarizes their most relevant features, whereas Figure 2(a) shows their load index values.

In this last figure, it can be seen that there are three different groups of machines, according to their nodes' features. With respect to the load index figures, they are not difficult to explain. For example, for node C0-1, the load



(a) Theoretical values of the load index for all the nodes available in the system

(b) Homogeneous distribution: evolution of the nodes' load index with time without any load balancing algorithm. Higher index values mean less loaded nodes

Fig. 2 Theoretical values and results without any load balancing algorithm

index goes down from 0.555 to 0.368 whenever it gets an additional task, which means that 0.368 is the load index value that the node offers to a third task when it is already running two tasks, corresponding to a load index of 0.555.

Table 1 Nodes selected for the experiments.

Node	#Cores	MFlops	P_i/P_{max}
C0-0	2	669.02	0.555
C0-1	2	664.64	0.551
C0-9	4	628.57	0.521
C0-10	4	666.86	0.553
C0-11	4	669.08	0.555
C0-14	4	772.84	0.641
C2-32	8	1201.67	0.997
C2-33	8	1198.34	0.994
C2-34	8	1203.87	0.998
C2-35	8	1205.68	1.000

During the experiments, the system is always loaded with 100 tasks that perform different NAS benchmarks, each one with different requirements and computing times. Additionally, depending on the specific test, the tasks were loaded in just one node, several on all of the nodes.

In the first experiment the tasks are evenly distributed among all the nodes in the cluster without any load balancing algorithm. The results of this experiment will be the baseline to compare the results obtained with the others, using the load balancing algorithm.

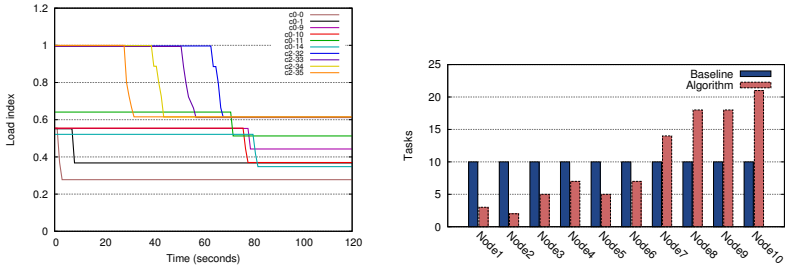
Figure 2(b) shows the evolution of the load indexes at each node along time. The larger the value of the index, the more unloaded a node is, and therefore, the more ready it is for accepting new tasks. Figure 2(b) also confirms there are three kinds of nodes: the most powerful ones, with load indexes above 0.75, the medium ones, with indexes around 0.2, and the least powerful ones, with indexes around 0.1. It took the system 1090 seconds to complete all the tasks.

The same experiment was also performed on the same system, but using the load balancing algorithm. In this case, all the tasks were launched on just one node (c0-0), so that it had to dispatch them to the remainder nodes as a result of the load balancing algorithm. Table 2 shows the values selected

for the different parameters of the load balancing algorithm. Figure 3(a)

Table 2 Parameters used in the second experiment

Parameter	Value
Time interval to measure state	1 second
Location of Recipients	Sorted list of 3 candidates
Workload	100 tasks, variable length
Thresholds for changing state	*Recipient* to *Neutral* at 0.7 *Neutral* to *Emitter* at 0.4



(a) Evolution of the nodes' indexes along the time.

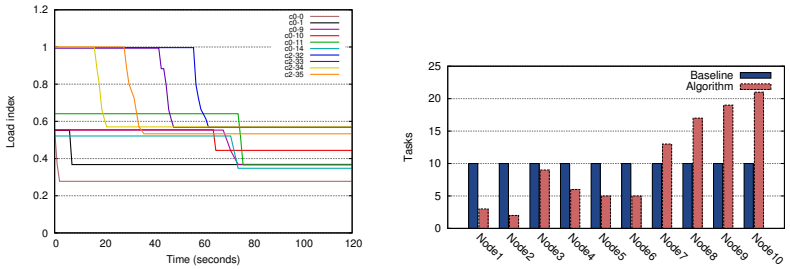
(b) Number of tasks executed in each node.

Fig. 3 Results obtained using the load balancing algorithm.

shows the evolution of the system nodes' load indexes during the experiment, showing that all of the indexes take more grouped values after a short initial balancing stage, which takes less than a minute and a half. This means that the workload has been distributed proportionally to the nodes' computational power, cutting the global execution time to 458 seconds, that is, achieving a speedup of 2.38.

Figure 3(b) shows the number of tasks executed on each node during these two experiments. Instead of having each node assigned the same number of tasks, the most powerful nodes get now larger workload shares.

A third experiment was carried out in order to check the effects of adjusting the thresholds on the overall system response. The initial idea came from the

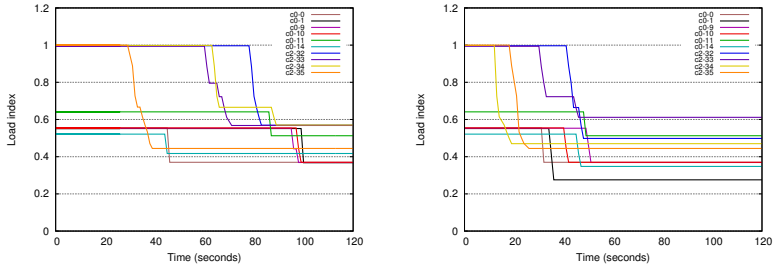


(a) Evolution of the nodes' indexes along time with the new thresholds.

(b) Number of tasks executed in each node.

Fig. 4 Results obtained with the new thresholds.

fact that Figure 3 shows several nodes with large load indexes values, meaning that they could probably accept additional tasks, but that could not change to the *Recipient* state because of the parameter settings. For that purpose, a new experiment was done using the following thresholds: *Recipient to Neutral* at 0.65 and *Neutral to Emitter* at 0.45. Figure 4 shows the results obtained in this case; it can be seen that he load index of the node that distributes the tasks (c0-0) keeps a similar value, due to the ratio between the refresh interval of the load index and the arrival rate of new tasks. Also, the index of the node c0-10 increases, while the indexes of the most powerful nodes decrease. It can be noticed also that the load indexes are more grouped than in the previous experiments, meaning they could effectively accept more load, and the total execution time is reduced to 427 seconds, achieving a speedup of 2.55.



(a) Task launched on the most powerful node: c2-35.

(b) Cluster with 2 users.

Fig. 5 Additional experiments: evolution of the nodes' load index along time.

Finally, Figure 5 shows two additional experiments run to test the system behavior. In the first one all the tasks are launched on just one node, but this time it was selected the most powerful node (c2-35) instead of the least one (c0-0). It can be seen in Figure 5(a) that the nodes' load indexes are more grouped than in the previous experiments, since the most powerful node is now the one that has to change to *emitter* state but accepts more local tasks. It means there are less load balancing operations and therefore the performance is improved. The second experiment tests the system with concurrent users. For this purpose, the 100 tasks were launched in 2 different nodes (50 tasks each); Figure 5(b) shows that apart from the node c2-33, the rest of the nodes' load indexes are more similar, since the more even initial distribution helps the load balancing algorithm.

4 Conclusions and Future Work

This paper presents a workload balancing algorithm that takes into account the possible existence of node heterogeneity in present HPC machines. The method proposed here considers two levels of heterogeneity: the number of cores per node and the individual computing power of each core, being able to model accurately the behavior of present day machines. This method is a dynamic, distributed, global, emitter-initiated and non-preemptive algorithm, which is able to turn itself down when all the nodes are overloaded or underloaded, and

in consequence, performing load balancing operations would not provide any gain, introducing only additional overheads.

Nowadays, heterogeneity in HPC systems is originated mainly by differences regarding the nodes computing capabilities. For achieving equilibrated load distributions, it is essential that load balancing algorithms are able to evaluate dynamically those capabilities as accurately as possible; the load index proposed here can be used for this purpose. The experiments carried here show clearly how the load index presented in this paper can be used for achieving balanced load distributions, where the workload is divided among the nodes proportionally to their computing power. When this is correctly done, the method results in large improvements, with execution times divided by two. Finer parameter tuning can yield even better results.

Future work includes providing the algorithm with self-learning mechanisms to take into consideration the system's global workload, giving the algorithm the possibility to adapt itself by modifying its parameters dynamically.

Acknowledgements

This work was supported by the Spanish Ministry of Education, Science and Innovation (grants TIN2010-21289, TIN2010-21291-C02-02, Consolider CSD2007-00050 , and the Cajal Blue Brain Project, Spanish partner of the Blue Brain Project initiative from EPFL).

References

1. Nas parallel benchmarks. http://www.nas.nasa.gov/Software/NPB.
2. The top500 project. November 2010. http://www.top500.org.
3. J. Dongarra and A.L. Lastovetsky. High Performance Heterogeneous Computing. *IEEE International Symposium on Parallel and Distributed Computing*. John Wiley & Sons, 2009.
4. Truong Vinh Truong Duy, Y. Sato, and Y. Inoguchi. Improving accuracy of host load predictions on computational grids by artificial neural networks. In *IEEE International Symposium on Parallel Distributed Processing. IPDPS 2009*, pages 1 –8, may 2009.
5. Ismael Galindo, Francisco Almeida, and José Manuel Badía-Contelles. Dynamic load balancing on dedicated heterogeneous systems. In *PVM/MPI*, pages 64–74, 2008.
6. N.K. Gondhi and D. Pant. An evolutionary approach for scalable load balancing in cluster computing. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 1259 –1264, march 2009.
7. Wenzheng Li and Hongyan Shi. Dynamic load balancing algorithm based on fcfs. In *Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on*, pages 1528 –1531, dec 2009.
8. J. Martínez, F. Almeida, E. Garzón, A. Acosta, and V. Blanco. Adaptive load balancing of iterative computation on heterogeneous nondedicated systems. *The Journal of Supercomputing*, 58:385–393, 2011. 10.1007/s11227-011-0595-3.
9. Xiaonian Tong and Wanneng Shu. An efficient dynamic load balancing scheme for heterogenous processing system. In *International Conference on Computational Intelligence and Natural Computing, 2009. CINC '09*, volume 2, pages 319 –322, 2008.
10. Chengzhong Xu and Francis Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Boston, 1997.
11. Y., X.-H. Sun, and M. Wu. Algorithm-system scalability of heterogeneous computing. *Journal of Parallel and Distributed Computing*, 68(11):1403–1412, 2008.