ComprehensiveBench: a Benchmark for the Extensive Evaluation of Global Scheduling Algorithms

Laércio Lima Pilla, Tiago Bozetti, Marcio Castro, Philippe Navaux, Jean-François Méhaut

ComprehensiveBench: a Benchmark for the Extensive Evaluation of Global Scheduling Algorithms

Laércio L Pilla[1], Tiago C Bozzetti[2], Márcio Castro[1], Philippe O A Navaux[2] and Jean-François Méhaut[3]

[1]Department of Informatics and Statistics, Technology Center, Federal University of Santa Catarina, Florianópolis, BR
[2]Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, BR
[3]University of Grenoble Alpes, LIG, CEA-INRIA, Grenoble, FR

E-mail: laercio.pilla@ufsc.br, tcbozzetti@inf.ufrgs.br, marcio.castro@ufsc.br, navaux@inf.ufrgs.br, jean-francois.mehaut@imag.fr

Abstract. Parallel applications that present tasks with imbalanced loads or complex communication behavior usually do not exploit the underlying resources of parallel platforms to their full potential. In order to mitigate this issue, global scheduling algorithms are employed. As finding the optimal task distribution is an NP-Hard problem, identifying the most suitable algorithm for a specific scenario and comparing algorithms are not trivial tasks. In this context, this paper presents ComprehensiveBench, a benchmark for global scheduling algorithms that enables the variation of a vast range of parameters that affect performance. ComprehensiveBench can be used to assist in the development and evaluation of new scheduling algorithms, to help choose a specific algorithm for an arbitrary application, to emulate other applications, and to enable statistical tests. We illustrate its use in this paper with an evaluation of Charm++ periodic load balancers that stresses their characteristics.

1. Introduction

Science has advanced in the last decades partially by virtue of numerical simulations. These scientific applications are developed using parallel programming languages and interfaces in order to benefit from the computing and memory resources available in High Performance Computing (HPC) platforms. These applications are decomposed into parallel tasks (such as threads or processes) that are distributed over the available resources. Due to the nature of the simulations, tasks can possess different computational loads, complex communication graphs, or both. These behaviors result in load imbalance and communication overhead that affect the applications' performance and scalability. In these scenarios, global scheduling algorithms such as load balancers are employed to distribute tasks in a manner to mitigate performance issues [1].

As the problem of finding an optimal task distribution is known to be NP-Hard [2], several load balancing, work stealing, and task mapping algorithms have been proposed for different scenarios [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. These algorithms show different complexities, consider different information, and optimize or improve different objectives. Due to all this diversity,

the comparison between different global scheduling algorithms is not a trivial task. This affects several questions related to the problem, including the following and their combination:

- *Is one scheduling algorithm better than another? Do we have statistically significant results to verify that?*
- *Should my new algorithm consider this information? Does it improve its task distribution?*
- *What is the best algorithm for my application? What is the best algorithm for my parallel platform?*
- *Should I migrate my application to an environment that supports task rescheduling?*

Current benchmarks used for the evaluation of scheduling algorithms lack a standard, usually limiting or ignoring some characteristics of scientific applications that influence algorithms' performance. Two examples of characteristics left out are the amount of data that has to be migrated with a task and the dynamic behavior of a task's load.

In this context, we propose a novel benchmark for the evaluation of global scheduling algorithms on shared and distributed memory platforms named ComprehensiveBench[1]. It considers the main characteristics of parallel applications that affect global scheduling algorithms' performance, and is more expressive in the way these characteristics are specified, enabling the simulation of dynamic behavior. To illustrate its capability of stressing the main characteristics of global scheduling algorithms, we perform experiments with ComprehensiveBench in different scenarios and load balancing algorithms. Our results confirm that ComprehensiveBench can be used to support the development and evaluation of scheduling algorithms.

The remaining sections of this paper are organized as follows: related work is discussed in Section 2. ComprehensiveBench is characterized in Section 3. Its implementation in Charm++ is presented in the same section. The experimental setup used to exemplify ComprehensiveBench's use and the global scheduling performance results are shown in Section 4. Finally, concluding remarks and future work are discussed in Section 5.

2. Related Work

Benchmarks have been extensively used to evaluate the performance of parallel platforms, parallelization tools, parallel libraries, and scheduling algorithms [3, 4, 5, 6, 7, 8, 9, 10, 11, 12]. Examples include the NAS Parallel Benchmarks (NPB) [13] and their multi-zone versions (NPB-MZ) [14], the Princeton Application Repository for Shared-Memory Computers (PARSEC) [15], the HPC Challenge benchmark suite (HPCC) [16], and the Rodinia benchmark suite [17]. Nevertheless, these benchmarks have limitations that impair the evaluation of global scheduling algorithms. For instance, NPB is limited to preset input sizes and is mostly useful for the evaluation of thread and process mapping algorithms that focus on improving communication performance, PARSEC and Rodinia can only be used in shared memory environments, and HPCC does not show load imbalance, focusing only in stressing parts of a parallel platform. The Mantevo Project [18] shows more flexibility with its miniapps, as MiniFE can be tuned to present load and communication imbalance, and Prolego can be defined using different code fragments. Nevertheless, dynamic features do not seem to be implemented.

Different approaches have been used to evaluate and compare global scheduling algorithms. When the main focus of the work is to improve one application's performance, only said application is used for comparison. This is the case for applications such as NAMD [4], BRAMS [5], Betweenness Centrality [7], and Ondes3D [19]. Another approach involves the generation of artificial application data for the offline comparison of algorithms, as done by

[1] Available online in the HieSchella project page: `http://forge.imag.fr/projects/hieschella/`

Catalyurek et al. [3]. A more toilsome way to compare algorithms requires the implementation of specific methods as benchmarks, as done by Lifflander, Krishnamoorthy and Kale [20]. Lastly, a more broad approach benefits from existing global scheduling benchmarks and mini-apps – such as *Adaptive Mesh Refinement* (AMR), *kNeighbor*, *lb_test*, *LeanMD*, and *stencil4D* – for experiments [6, 8, 9, 10]. Still, these benchmarks limit the expression of load imbalance, dynamic load behavior, communication behavior, initial mapping, size of tasks, or a combination of these. The use of benchmarks with limited ranges of parameters and behaviors jeopardizes the attainment of statistically significant results [21]. We explain how ComprehensiveBench circumvents these issues in the next section.

3. ComprehensiveBench

We describe in this section the main characteristics of our novel global scheduling benchmark. First, we discuss its main desired characteristics and features. This is followed by a description of all ComprehensiveBench's parameters with examples of use. Lastly, we detail how the benchmark is implemented.

3.1. Features

ComprehensiveBench displays a series of different characteristics to enable the comprehensive evaluation of global scheduling algorithms and emulation of applications. They include the expression of the following attributes:

- *Irregular loads*: tasks can have different loads.
- *Dynamic loads*: a task's load can change during execution.
- *Mixed loads*: tasks can execute integer or floating point operations.
- *Irregular communication*: the number of messages sent by a task and their sizes can vary among tasks.
- *Dynamic communication*: the number of messages sent by a task and their sizes can change during execution.
- *Diverse communication graphs*: tasks can be set to communicate with others following different patterns.
- *Irregular task sizes*: tasks can have different memory footprints.
- *Diverse initial mappings*: tasks can be initially mapped to cores following different patterns.

3.2. Parameters

ComprehensiveBench's parameters can be divided into two categories: simple and complex. *Simple parameters* are defined using natural numbers or keywords, while *complex parameters* are defined using expressions involving arithmetic operations, variables, numbers, and conditional (ternary) operators. We discuss each category in the following sections.

3.2.1. Simple Parameters ComprehensiveBench's static characteristics are all set using natural numbers or special keywords. Table 1 lists these benchmark characteristics. The number of tasks to be executed by ComprehensiveBench (`n`) directly affects the execution time of global scheduling algorithms, as a larger number of tasks leads to more time spent on mapping decisions. The number of cores to be considered by the benchmark (`p`) has a similar effect, as the larger the decision space is, the longer a scheduling algorithm may take to decide where to map a task.

The number of benchmark iterations (`r`) affects for how long it may execute. The longer ComprehensiveBench executes, the more time scheduling decisions have to affect

Table 1. ComprehensiveBench's simple parameters.

Parameters	Meaning
`n`	Number of tasks.
`p`	Number of processing units (cores).
`r`	Number of iterations.
`lbfreq`	Rescheduling (load balancing) frequency.
`int_op`	Data type to be processed.
`gcomm`	Communication graph.

performance. The rescheduling frequency (`lbfreq`) has an effect related to it. If the rescheduling periodicity is small (few iterations), then performance issues like load imbalance and a high communication overhead can be fixed quickly. Nevertheless, if the algorithm takes too long computing a new task distribution each time, then its overhead may overcome its benefits. Meanwhile, if the rescheduling periodicity is large (many iterations), then it may take too long to correct a poor task distribution.

The `int_op` parameter is used to set the data type to be processed by tasks. If set to true, tasks work with integer data, otherwise tasks execute single precision floating-point operations. This parameter can be used to stress processors and influence the decisions of temperature- or energy-aware global scheduling algorithms.

The last simple parameter is the communication graph (`gcomm`). While the previous parameters are set using natural numbers mostly, `gcomm` is based in three predefined graphs: a ring, a 2D mesh, and a 3D mesh. The communication graph defines which tasks communicate with each other, and is independent of the machine topology. While the communication graph defines which tasks communicate with each other, it does not define how many messages will be exchanged nor their sizes. These are left to complex parameters discussed next.

3.2.2. Complex Parameters ComprehensiveBench's complex parameters are expressed using simple parameters and two different variables with values set at run time. These variables are presented in Table 2. By using a task's identifier (`t`) to influence its load and communication, ComprehensiveBench enables irregularity among tasks. Meanwhile, the use of the current iteration (`i`) makes it possible to vary loads and communication throughout the benchmark execution, bringing dynamicity. In this sense, the complex parameters work as functions that have to be computed for each task when first creating them or by each task at each iteration.

Table 2. ComprehensiveBench's run time variables.

Variable	Meaning	Range
`t`	Task identifier.	$0 \leq \texttt{t} < \texttt{n}$
`i`	Current iteration.	$0 < \texttt{i} \leq \texttt{r}$

Table 3 lists the complex parameters used in the benchmark, including their unities and set of variables and simple parameters used in their expressions. All functions result in integer values. For instance, the initial mapping of tasks to cores (`initmap`) can be defined using the task identifier, the total number of tasks being executed, and the total number of cores. Equation 1 shows the definition of a Round-Robin task distribution, while Equation 2 shows a compact task distribution, where a range of tasks are mapped to the same core, and Equation 3 shows a mapping where all tasks start executing in the same core. In all cases, `initmap` must be in

the range $0 \leq \texttt{initmap} < \texttt{p}$. By changing the initial task mapping, ComprehensiveBench enables the creation of different scenarios with poor work and communication distributions to be improved by global schedulers.

$$\texttt{initmap} = \texttt{t} \mod \texttt{p} \tag{1}$$

$$\texttt{initmap} = (\texttt{t} * \texttt{p})/\texttt{n} \tag{2}$$

$$\texttt{initmap} = 5 \tag{3}$$

Table 3. ComprehensiveBench's complex parameters.

Parameters	Meaning	Variables and Parameters	Unit
`initmap`	Initial mapping.	`t`, `n`, `p`	Core id
`tasksize`	Task memory footprint.	`t`, `n`	Bytes
`load`	Task execution time.	`t`, `i`, `n`	ms
`msgnum`	Number of messages.	`t`, `i`, `n`	—
`msgsize`	Data volume per message.	`t`, `i`, `n`	Bytes

Each task occupies some space in memory associated with its code, internal variables and data structures. Whenever a task is assigned to a core different from its current one, its data has to be migrated. In this sense, a task's memory footprint has direct influence over its migration overhead and the scheduling algorithm's execution time. This data volume is expressed in bytes with the `tasksize` parameter. Its value is limited by the maximum integer possible in the platform (this is the same for `load`, `msgnum` and `msgsize`). All tasks can either occupy the same amount of memory, as shown in Equation 4, or have different memory footprints. The latter is exemplified in Equation 5, where tasks with higher identifiers have more data. A global scheduling algorithm that is aware of tasks' data volumes may opt to not migrate a task based in its estimated migration overhead.

$$\texttt{tasksize} = 1000 \tag{4}$$

$$\texttt{tasksize} = 100 * \texttt{t} * \texttt{t} \tag{5}$$

The execution time of a task is set to each of the benchmark's iterations by the `load` parameter. Its expression is evaluated at the beginning of each iteration in order to enable dynamicity. For instance, Equation 6 shows a situation where the load of all tasks increases by 10 ms at each iteration. Tasks can also be split into groups with different loads, as illustrated by the use of the conditional operator in Equation 7. A combination of dynamic and irregular loads is presented in Equation 8. Using this powerful expression, ComprehensiveBench is able to create varied load imbalance scenarios to stress global scheduling algorithms and to simulate the behavior of real applications.

$$\texttt{load} = 10 * \texttt{i} \tag{6}$$

$$\texttt{load} = \texttt{t} < \texttt{n}/2 \ ? \ 10 : 50 \tag{7}$$

$$\texttt{load} = 5 * \texttt{i} * (\texttt{t} + 1) \tag{8}$$

The last complex parameters are the number of messages to be sent to another task (`msgnum`) and the size of the messages in bytes (`msgsize`). By setting the communication behavior of tasks independently from one another, ComprehensiveBench provides the opportunity to test the

communication resources of parallel platforms (i.e., memory and network) and the effectiveness of global scheduling algorithms in reducing the overall communication time of applications. Equation 9 illustrates a scenario where the number of messages increases with the execution of the benchmark, while Equation 10 simulates a scenario where tasks exchange more data at every five iterations.

$$\texttt{msgnum} = 1 + \texttt{i}/5 \tag{9}$$

$$\texttt{msgsize} = \texttt{i} \mod 5 < 1 \ ? \ 10000 : 100 \tag{10}$$

3.3. Implementation Details

ComprehensiveBench is implemented using Charm++ [22, 23]. Charm++ is a parallel programming language and runtime system (RTS) based on C++ with the goal of improving programmer productivity. It abstracts architectural characteristics from the developer and provides portability over multiprocessor and multicomputer platforms. Charm++ applications are composed of specially designed C++ objects named *chares* that play the role of tasks, being responsible for their own work and data. Chares communicate using asynchronous method invocations following a message-driven execution.

Charm++ includes a load balancing framework [6] that provides an interface for developing load balancing plugins to Charm++ applications. A load balancer is provided with information regarding the last iterations of the application and its current mapping, and the runtime system expects in return a new task distribution to migrate tasks. Several global scheduling algorithms are distributed with this framework, which helps experimenting with our benchmark.

In order to enable the use of complex expressions to define the tasks' loads and other characteristics, all of ComprehensiveBench's parameters are set in an input file that is preprocessed right before the benchmark is compiled.

At the start of ComprehensiveBench's execution, all tasks are created and distributed among cores following the initial mapping expression. Their communication graph is also set during initialization. Since several tasks execute in the same core during an iteration, their loads cannot be simulated by just putting them to sleep. To overcome this issue, a task in the first core of the platform is responsible for estimating how many internal iterations must be done for a task to compute for a millisecond. This value is then used by all tasks to guide their loads at each iteration. This standardization of the workload also enables using ComprehensiveBench in environments with different processors or processors running at different clock frequencies.

4. Experimental Evaluation

In this section, we present experiments performed with ComprehensiveBench that illustrate its capability of stressing the characteristics of global scheduling algorithms. First, we detail the experimental setup with the parallel platform and Charm++ load balancing algorithms used. This is followed by the description and explanation of the results of four different experimental configurations.

4.1. Experimental Environment

Table 4 describes the parallel platform and system software used in the experiments. In total, 128 cores were used in the experiments. It is important to emphasize that this machine has a nonuniform memory access (NUMA) architecture with NUMA factors (ratio between remote latency and local latency) between 6.4 and 9.6. This results in slow communication between different NUMA nodes.

In the interest of reducing variability in the results, threads were pinned to cores in all experiments using `numactl`.

Table 4. Experimental platform.

Processors	16× Intel Xeon E5-4640
Cores	16 × 8 @ 2.4 GHz
Memory	16 × 32 GB DDR3 @ 1600 MHz
Linux kernel version	3.0.1010.46
G++ version	4.3.4
CHARM++ version	6.6.0
CHARM++ build	Multicore-linux64

4.2. Global Scheduling Algorithms

Five different load balancing algorithms implemented in CHARM++ were used in the experiments. Their main characteristics are discussed below.

- ***GreedyLB***: A centralized greedy algorithm that only uses task loads for its decisions. It is employed to quickly mitigate load imbalance. It sorts tasks in decreasing load order and iteratively maps the unassigned task with the highest load to the least loaded core. The initial task distribution is not considered in this process, which leads to a large number of task migrations.
- ***GreedyCommLB***: Similar to *GreedyLB* with the addition of communication information in its decisions. The algorithm computes a task's communication load based on the data volume that it sends to tasks mapped in other cores. After sorting tasks in decreasing load order, *GreedyCommLB* selects the unassigned task with the highest load and maps it to the core with the smallest total load, which includes both processing and communication loads. The algorithm has the tendency to map tasks with intense communication to the same core.
- ***RefineLB***: A less aggressive algorithm than the greedy strategies. The algorithm considers the current task mapping in its decisions, iteratively refining it to a state where no core is overloaded or no task migration improves it. *RefineLB* uses a threshold to define if a core is overloaded. At each iteration, it verifies all possible task migrations from the most overloaded core to all underloaded cores, migrating the task that brings its new core the closest to the threshold. As *GreedyLB*, it does not consider communication in its decisions.
- ***RefineCommLB***: Similar to *RefineLB*, but considers the communication behavior of tasks in its decision of the most suitable destination core for a task.
- ***RandCentLB***: Random algorithm that does not take into consideration any information about the application or the current state of the platform. It maps tasks to cores randomly using a uniform distribution.

Besides these five algorithms, all experiments were also executed without any load balancer to provide a baseline for the results.

4.3. COMPREHENSIVEBENCH *Parameters*

Four different synthetic setups were used with COMPREHENSIVEBENCH: the first one illustrates the impact of tasks' sizes on performance; the second one presents scenarios with static load imbalance; the third one shows how different algorithms handle dynamic load imbalance; and the forth one stresses how load balancers handle communication-intense applications on a NUMA platform. The parameters used in all three setups are listed in Table 5. All results presented in the next section represent the average times measured in 32 runs and present a statistical confidence of 95% by Student's t-distribution and a 2% relative error.

Table 5. ComprehensiveBench parameters for the different scenarios tested.

Parameter	1^{st} scenario	2^{nd} scenario	3^{rd} scenario	4^{th} scenario
`n`	2000	2000	2000	2000
`r`	50	20	20	20
`lbfreq`	5	5	5	5
`int_op`	`false`	`false`	`false`	`false`
`gcomm`	`Ring`	`Ring`	`Ring`	`Ring`
`initmap`	$(\texttt{t} * \texttt{p})/\texttt{n}$	$\texttt{t} \mod \texttt{p}$	$\texttt{t} \mod \texttt{p}$	$(\texttt{t} * \texttt{p})/\texttt{n}$
`tasksize`	{10 KB, 100 KB, 1 MB}	100	100	100
`load`	10	$10 + \{0, 1, 2, 4\} * (\texttt{t} \mod \texttt{p})$	$10 + ((\texttt{i}/10) * \{1, 2, 4\} * (\texttt{t} \mod \texttt{p})$	10
`msgnum`	1	1	1	$\{1, 5, 10\}$
`msgsize`	100	100	100	10000

4.4. Results

The results of the experiments with each of the four scenarios are presented in the following sections.

4.4.1. Impact of Tasks' Sizes This first scenario simulates a balanced application whose tasks have nontrivial memory footprints, resulting in tasks with high migration costs. Tasks are set with small loads and little communication, and no benefit is expected to be achieved with load balancing. In this scenario, ComprehensiveBench helps evaluate the capacity of the different load balancing algorithms in estimating and avoiding migration costs.

The total execution times achieved by the different load balancing algorithms for tasks with sizes of 10 KB, 100 KB, and 1 MB are presented in Figure 1. The baseline represents the execution of ComprehensiveBench without any load balancer and its total execution time stays the same for all task sizes as no tasks are migrated. The increase in migration overhead with the increase of task size can be seen for load balancers *GreedyLB*, *GreedyCommLB*, and *RandCentLB*. These load balancers do not take into consideration the current task mapping and result in several task migrations at each load balancing call. The total migration overhead for these load balancers is approximately 4.6, 7, and 18 seconds for the increasing task sizes. This is a result of migrating 99% of the tasks at each load balancing call. *RandCentLB* shows a bigger increase in total execution time because its random migrations also result in load imbalance.

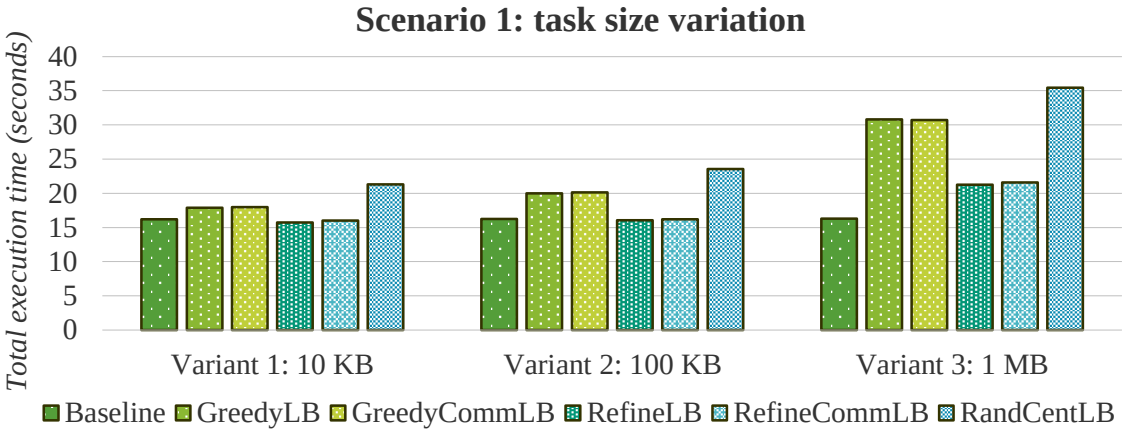

Figure 1. Load balancers' performance with varying task sizes.

Meanwhile, *RefineLB* and *RefineCommLB* achieved the same total execution time as the

baseline for the two variants with smaller tasks, increasing execution time only when migrating 1 MB tasks. As these algorithms try to improve the current task mapping, they are able to notice that the load is mostly balanced, suffering only with some communication overhead and OS jitter, and thus, resulting in almost no migrations. This indicates that refinement-based algorithms can be much more suitable for application with large memory footprints or parallel platforms with high migration costs.

4.4.2. Influence of Static Load Imbalance The second scenario simulates an imbalanced application whose tasks have static loads. The load imbalance comes from having tasks in 128 load groups, where each group has tasks with the same load mapped to the same core. Tasks' sizes and communication are kept at a minimum to avoid influencing the final performance. The use of ComprehensiveBench helps to identify the possible performance gains with different algorithms for applications with irregular but static behavior, and how quickly these algorithms are able to mitigate load imbalance.

Figure 2 presents the total execution time with global scheduling algorithms and different levels of load imbalance. In Variant 1, all tasks and cores have the same load. The total execution time without load balancing and with all algorithms except *RandCentLB* is approximately 6.8 seconds, including iterations, communication, runtime overhead and initialization. Even though greedy algorithms migrate many tasks at each load balancing call, this migration overhead only increases total execution time by 0.6 seconds.

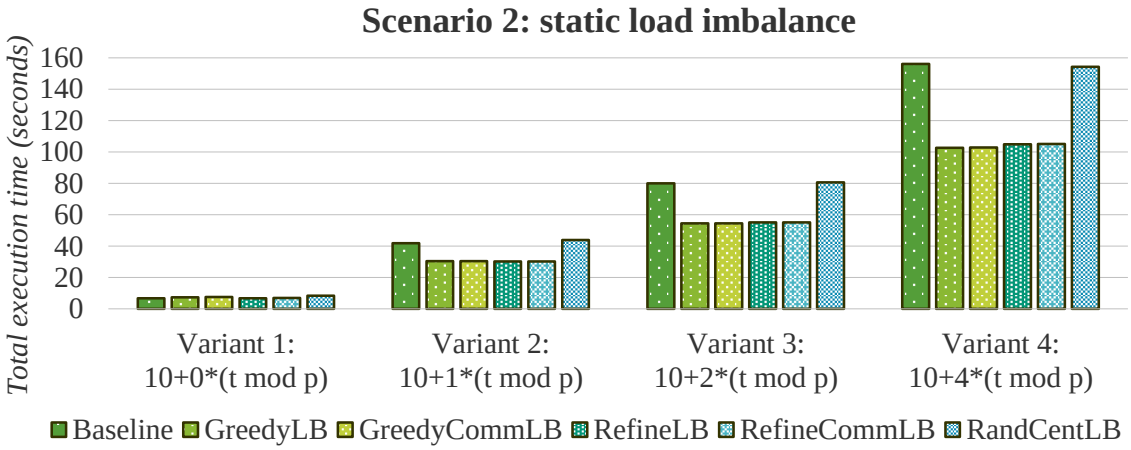

Figure 2. Load balancers' performance with static load imbalance.

As the load imbalance increases with the variant numbers, the total execution time of ComprehensiveBench increases accordingly. *GreedyLB*, *GreedyCommLB*, *RefineLB*, and *RefineCommLB* are able to improve performance in all variants, achieving almost optimal work distributions in all cases. The most visible difference in performance between greedy and refinement-based algorithms happens in Variant 4. Nevertheless, this difference is only 3% and is inside the error margin of the results.

Finally, it is interesting to see that even *RandCentLB* is able to improve performance (but still inside the error margin) as the initial load imbalance increases. This happens because the chances of having a mapping better than the original one increase as the difference between the most and least loaded cores increases.

4.4.3. Influence of Dynamic Load Imbalance The third scenario simulates an application that starts balanced but becomes imbalanced by the middle of its execution. This load imbalance is

a result of dynamic changes in the tasks' loads, which is a common behavior in some scientific applications [5, 19]. In addition, load imbalance may happen even after several load balancing calls. In these situations, online global scheduling algorithms become necessary to achieve performance and scalability. The possibility of expressing dynamic behaviors is one of the main benefits of using ComprehensiveBench.

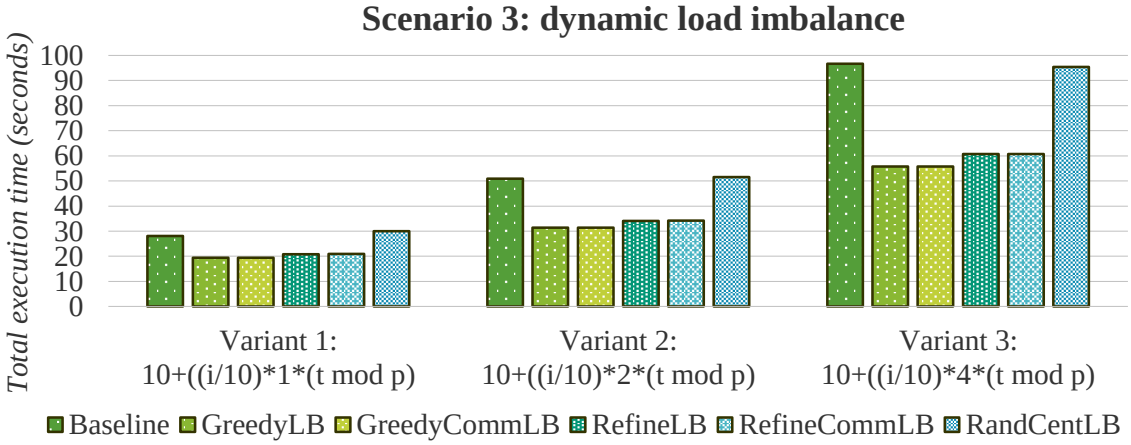

Figure 3. Load balancers' performance with dynamic load imbalance.

Three variants with increasing dynamic load differences were simulated. The total execution time achieved by the different algorithms is shown in Figure 3. In this situation, *GreedyLB* and *GreedyCommLB* improved performance the most. They have shown speedups of 1.7 over the baseline and 1.09 over *RefineLB* and *RefineCommLB* for Variant 3. As these greedy algorithms do no take the current mapping into account, they happen to redistribute tasks before their dynamic behavior takes place. This results in a less extreme load imbalance after the first ten iterations, and a smaller total execution time in the end. Meanwhile, the refinement-based algorithms move almost no tasks during the initial iterations and are faced with more load imbalance due to load dynamicity.

4.4.4. Influence of Communication The fourth and last scenario simulates communication-bound balanced applications. As tasks communicate in a ring and a compact task distribution was applied, locality is maximized with the initial mapping. In this situation, task migrations are unlikely to improve iteration time.

The total execution time achieved by the different scheduling algorithms in this scenario is presented in Figure 4. As the number of messages exchanged between tasks at each iteration increases, so does the total execution time of ComprehensiveBench. No load balancing algorithms was able to improve performance, just as happened in the first tested scenario. However, the increase in execution time in this scenario is a result of slower communication, not migration overhead. Finally, it can be noticed that the communication-aware scheduling algorithms, *GreedyCommLB* and *RefineCommLB*, show no improvement over their communication-oblivious counterparts. These algorithms underestimate the communication costs in this platform because they have no knowledge of the machine topology and memory hierarchy, which results in migrations that affect performance.

4.5. Summary

One of the most important results of all scenarios simulated with ComprehensiveBench is that no single scheduling algorithm was the best in all cases. For instance, *RefineLB* achieved the

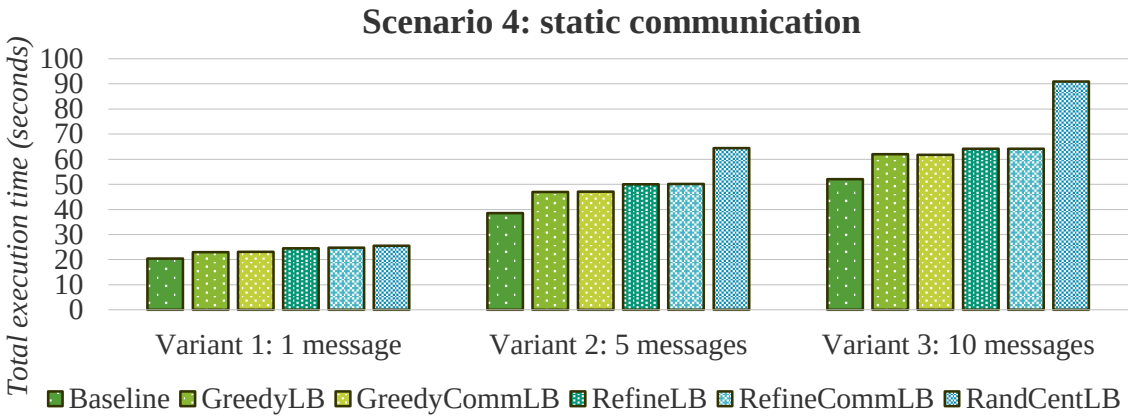

Figure 4. Load balancers' performance with varying numbers of messages.

best results on balanced applications whose tasks have nontrivial memory foot prints (Scenario 1) whereas *GreedyLB* achieved the best results on applications influenced by dynamic load imbalance (Scenario 3). Moreover, both scheduling algorithms presented similar results for imbalanced applications whose tasks have static loads (Scenario 2). Theses results emphasize the most indicated uses for some scheduling algorithms and the scenarios where they should be avoided. They also highlight the importance of a comprehensive benchmark to evaluate the pros and cons of different global scheduling algorithms.

5. Conclusion and Future Work

The difficulty of evaluating and comparing global scheduling algorithms led us to the development of a flexible and expressive benchmark. ComprehensiveBench considers the main characteristics of parallel applications that affect the performance of these algorithms, and enables the simulation of dynamic behavior through parameters set at compile time. ComprehensiveBench helps in the comparison of different global scheduling algorithms and the simulation of parallel applications.

We implemented ComprehensiveBench using the Charm++ parallel programming environment and used it to evaluate load balancing algorithms with four different scenarios. The first scenario highlighted the migration overhead of tasks with large memory footprints by algorithms that do not take the current task mapping into account for their decisions. The second scenario showed how greedy and refinement-based algorithms are able to achieve almost optimal work distributions when handling static load imbalance, while the third scenario illustrated how the greedy algorithms were better suited for dynamic load imbalance. Finally, the fourth scenario highlighted the weakness of *GreedyCommLB* and *RefineCommLB* in estimating the communication costs on NUMA platforms. In all tested cases, ComprehensiveBench allowed us to distinguish between the global scheduling algorithms and showed that no algorithm was the best for all scenarios.

As future work, we plan to employ ComprehensiveBench to evaluate different algorithms on a wide range of parallel platforms. We also plan to develop a mechanism for choosing global scheduling algorithms automatically given the characteristics of an application and a platform. ComprehensiveBench will help by providing varied test cases and enabling the use of machine learning. Finally, we project the use of ComprehensiveBench to emulate other parallel applications and evaluate their migration to environments that support task migration, such as Charm++.

References

[1] Casavant T L and Kuhl J G 1988 *IEEE Trans. Softw. Eng.* **14** 141–154 URL http://dx.doi.org/10.1109/32.4634

[2] Leung J Y T 2004 *Handbook of scheduling: algorithms, models, and performance analysis* Chapman & Hall/CRC computer and information science series (Chapman & Hall/CRC)

[3] Catalyurek U V, Boman E G, Devine K D, Bozdag D, Heaphy R and Riesen L A 2007 *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International* pp 1–11 URL http://dx.doi.org/10.1109/IPDPS.2007.370258

[4] Bhatele A, Kale L V and Kumar S 2009 *Proceedings of the 23rd international Conference on Supercomputing (ICS 2009)* (New York, NY, USA: ACM) pp 110–116 URL http://doi.acm.org/10.1145/1542275.1542295

[5] Rodrigues E R, Navaux P O A, Panetta J, Fazenda A, Mendes C L and Kale L V 2010 *Computer Architecture and High Performance Computing, Symposium on* **0** 71–78

[6] Zheng G, Bhatele A, Meneses E and Kale L V 2011 *International Journal of High Performance Computing Applications (IJHPCA)*

[7] Frasca M, Madduri K and Raghavan P 2012 *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* SC '12 (Los Alamitos, CA, USA: IEEE Computer Society Press) URL http://dl.acm.org/citation.cfm?id=2388996.2389125

[8] Pilla L L, Ribeiro C P, Cordeiro D, Mei C, Bhatele A, Navaux, Broquedis F, Mehaut J and Kale L V 2012 *Parallel Processing (ICPP), 2012 41st International Conference on* pp 118–127 URL http://dx.doi.org/10.1109/ICPP.2012.9

[9] Menon H and Kalé L 2013 *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis* SC '13 (New York, NY, USA: ACM) URL http://doi.acm.org/10.1145/2503210.2503284

[10] Pilla L L, Ribeiro C P, Coucheney P, Broquedis F, Gaujal B, Navaux P O A and Méaut J F 2014 *Future Generation Computer Systems* **30** 191–201 ISSN 0167-739X URL http://dx.doi.org/10.1016/j.future.2013.06.023

[11] Tessier F, Mercier G and Jeannot E 2014 *IEEE Transactions on Parallel and Distributed Systems* **25** 993–1002 URL http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.104

[12] Cruz E H M, Diener M, Pilla L L and Navaux P O A 2015 *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* pp 207–214

[13] Bailey D, Harris T, Saphir W, Wijngaart R V, Woo A and Yarrow M 1995 The NAS Parallel Benchmarks 2.0

[14] Van der Wijngaart R F and Jin H 2003 NAS Parallel Benchmarks, Multi-Zone Versions

[15] Bienia C, Kumar S, Singh J P and Li K 2008 *Parallel Architectures and Compilation Techniques (PACT)*

[16] Luszczek P R, Bailey D H, Dongarra J J, Kepner J, Lucas R F, Rabenseifner R and Takahashi D 2006 *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* SC '06 (New York, NY, USA: ACM) URL http://doi.acm.org/10.1145/1188455.1188677

[17] Che S, Sheaffer J W, Boyer M, Szafaryn L G, Wang L and Skadron K 2010 *Workload Characterization (IISWC), 2010 IEEE International Symposium on* pp 1–11

[18] Heroux M A, Doerfler D W, Crozier P S, Willenbring J M, Edwards H C, Williams A, Rajan M, Keiter E R, Thornquist H K and Numrich R W 2009 *Sandia National Laboratories, Tech. Rep. SAND2009-5574*

[19] Tesser R, Pilla L L, Navaux P O A, Dupros F, Mehaut J F and Mendes C 2014 *To be published on Parallel, Distributed and Network-Based Processing (PDP), 2014 22st Euromicro International Conference on* pp 1–8

[20] Lifflander J, Krishnamoorthy S and Kale L V 2012 *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing* HPDC '12 (New York, NY, USA: ACM) URL http://doi.acm.org/10.1145/2287076.2287103

[21] Touati S A A, Worms J and Briais S 2013 *Concurrency and Computation: Practice and Experience* **25** 1410–1426 URL http://dx.doi.org/10.1002/cpe.2939

[22] Kale L V and Krishnan S 1993 *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 1993)* (ACM) pp 91–108

[23] Acun B, Gupta A, Jain N, Langer A, Menon H, Mikida E, Ni X, Robson M, Sun Y, Totoni E, Wesolowski L and Kale L 2014 *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for* pp 647–658 URL http://dx.doi.org/10.1109/SC.2014.58