A Dynamic Multi–Objective Approach for Dynamic Load Balancing in Heterogeneous Systems

Alberto Cabrera, Alejandro Acosta, Francisco Almeida, and Vicente Blanco

Abstract—Modern standards in High Performance Computing (HPC) have started to consider energy consumption and power draw as a limiting factor. New and more complex architectures have been introduced in HPC systems to afford these new restrictions, and include coprocessors such as GPGPUs for intensive computational tasks. As systems increase in heterogeneity, workload distribution becomes a more core problem to achieve the maximum efficiency in every computational component. We present a Multi-Objective Dynamic Load Balancing (DLB) approach where several objectives can be applied to tune an application. These objectives can be dynamically exchanged during the execution of an algorithm to better adapt to the resources available in a system. We have implemented the Multi–Objective DLB together with a generic heuristic engine, designed to perform multiple strategies for DLB in iterative problems. We also present Ull Multiobjective Framework (UllMF), an open–source tool that implements the Multi-Objective generic approach. UllMF separates metric gathering, objective functions to be optimized and load balancing algorithms, and improves code portability using a simple interface to reduce the costs of new implementations. We illustrate how performance and energy consumption are improved for the implemented techniques, and analyze their quality using different DLB techniques from the literature.

Index Terms—Dynamic load balancing, energy efficiency, iterative algorithms, heterogeneous computing

1 Introduction

The urge to reduce energy consumption in computational systems in the past decade has driven hardware architecture in high performance computing (HPC). New architectures in HPC incorporate specialized hardware to accelerate parallel codes, such as Field–Programmable Gate Arrays (FPGAs) and General Purpose Graphical Processing Units (GPGPUs). These architectures are more efficient both for execution time and energy consumption in numerous scientific applications, but introduce heterogeneity due to the specialization of the hardware. Systems that implement these technologies have appeared in the most powerful parallel computers listed in the Top500. The difference in computational capabilities for all the resources in a system node introduce a new difficulty layer to achieve the optimal use of computational resources. As the natural outcome for next years is to increase the computing capabilities of these systems, power consumption has also become a major issue. In order to be able to tune applications for this new metric, energy measurements have to be performed before, during and after parallel codes are executed. The numerous measurement devices and software available [1], [2], [3] to obtain metrics increase the heterogeneity of the scientific community applications, further increasing the number of difficulties to address. To improve the performance of applications in heterogeneous environments, load balancing algorithms are crucial. Load balancing makes use of metrics in order to improve application performance, a task that can be difficult to address if these metrics have to be gathered and utilized at runtime. We propose to follow a Multi-Objective approach for load balancing where the applications can be dynamically tuned according to several objectives (time, energy, communications, ..., or their combinations). The method adapts to the objective that provides a better use of the resources at any moment through a dynamic objective function, which can change over time. This is particularly useful when the metrics used on any objective are asynchronous and differ in accuracy.

As a first contribution of this paper we present a generic dynamic load balancing heuristic engine that allows this adaptive Multi–Objective approach. Our previous work targeted performance or energy using specialized techniques [4], [5], the first based on the computational capacity of each process and the latter using energy efficiency metrics. As a step towards generalization, we developed a new approach for single objective dynamic workload balancing in iterative problems [6]. In this work, we present a new engine that allows to switch from the performance analysis to energy analysis, and to use metrics combining both. At the same time, we have built a development Framework (*Ull Multiobjective Framework (UllMF)* [7]) that allows an efficient and friendly use of the method over heterogeneous architectures.

• *The authors are with the HPC Group of Universidad de La Laguna, Escuela Superior de Ingeniería y Tecnología, 38270 San Cristóbal de La Laguna, Tenerife, Spain.*
E-mail: {Alberto.Cabrera, aacostad, falmeida, vblanco}@ull.es.

Manuscript received 29 June 2019; revised 27 Jan. 2020; accepted 13 Apr. 2020. Date of publication 23 Apr. 2020; date of current version 19 May 2020. (Corresponding author: Alberto Cabrera.)
Recommended for acceptance by B. Di Martino.
Digital Object Identifier no. 10.1109/TPDS.2020.2989869

We enumerate the contributions in this paper:

- An in–depth analysis of various objective functions for the proposed generic heuristic engine. The Multi–Objective approach uses the energy delay product (EDP) to reduce the disadvantages of energy measurement. We have implemented the performance version of the heuristic to compare the EDP to both their time and energy Single–Objective counterparts.
- A new technique to use dynamic objective functions during the dynamic load balancing algorithm. We propose two different approaches, *time–then–energy (TTE)* and *time–energy–switch (TES)*, that lessens the delay caused in the load balancing phase when using only energy consumption metrics at runtime.
- An open source tool, *Ull Multiobjective Framework (UllMF)*, designed to perform dynamic load balancing over heterogeneous systems. *UllMF* implements the previous contribution with very low code intrusion. Its abstraction hides the computational intricacies to provide non–expert users the capacity to port their codes to different architectures. The portability is also achieved by offering independence to gather energy consumption metrics. Additionally, in *UllMF* we present the following contributions:
  - The implementation of our generic heuristic to perform dynamic load balancing as a programming skeleton. To apply a new restriction to perform dynamic load balancing, only the new objective function has to be added.
  - A mechanism to switch the load balancing algorithm, the objective function or the measurement tool at runtime, which could be used to develop dynamic load balancing techniques, such as some cases presented in this work.
  - An easily extensible design using modules. Algorithms, objective functions and measurement devices are abstracted to each other. A user can implement a new algorithm using the existing objective functions or measurement devices without considering specific details of the modules.

To validate all our work, we have performed a set of experiments using an heterogeneous multi GPU cluster over iterative problems. Iterative problems are a class of problems that appear in multiple scientific fields. Examples of these problems in the literature are the Jacobi method, the stencil codes, the longest common sub–sequence problem, sparse triangular solvers and in general any dynamic programming algorithm [8], [9]. Existing implementations for these algorithms can be executed in modern architectures. However, due to the high heterogeneity of the computational environments, both execution time and energy consumption would be heavily affected for them. Moreover, parallel implementations need to be tuned to specific systems to achieve the most efficient resource usage, a very difficult task. Load balancing techniques address this task redistributing the workload of an application in order to decrease execution time or to reduce energy consumption, among other possibilities. Four different types of dynamic programming iterative problems have been tested in our computational experience. Our results are compared against an homogeneous distribution of the work, a dynamic load balancing algorithm [4] that optimizes execution time, and our previous work. In every case, we improve the use of resources when compared to the homogeneous distribution and at least one of the provided implementations is equal or better than the previous dynamic load balancing techniques.

This paper is structured as follows: In Section 2 an overview of related work in the field is presented. Section 3 presents the heuristic algorithm that allows for the Multi–Objective and dynamic approach. *UllMF* is described in depth in Section 4. Section 5 describes our computational experience with various dynamic programming algorithms. Finally, our conclusions and future work are outlined in Section 6.

2 Related Work

The load balancing problem is present in the literature and has been discussed thoroughly to improve performance in high performance systems. Multiple linear algebra packages implement solvers and computational models based on Directed Acyclic Graphs. Such is the case of MAGMA [18], Flame [19] and PLASMA [20]. Load balancing is also applied outside of the HPC context, mainly to improve performance based on a different resource usage. Peer Virtual Machines (VMs) aggregation [21] is proposed to perform a communication–aware placement for parallel applications. VMs in this environment are rescheduled based on determined communication patterns. Load balance techniques are also combined with data–aware scheduling through a work stealing technique for data intensive applications [22].

Multiple performance and power/energy–aware algorithms have been developed following load balancing techniques. In Table 1, we have included a collection of relevant contributions available in the literature. First, even if they are not directly related to our proposal, it is worth mentioning energy–aware scheduling algorithms. CEEDMIP [10], is a Contention-aware, Energy Efficient, Duplication based Mixed Integer Programming formulation, which focuses on optimizing the use of energy and the duplication for communication intensive applications. The high amount of task scheduling techniques available in the literature for heterogeneous multiprocessors also motivated the development of QHA [11], a quantum–inspired energy–aware hyper–heuristic that tackles the power and performance trade–off optimization problem automatically managing low-level heuristics. On a higher scale, EPPADS [12], a high–performance light–weight scheduler for improving job processing in large scale clusters, combining performance optimization with power saving management schemes to avoid the limitations of MapReduce schedulers. At a GPU level, KSRE [13] is a kernel scheduling approach for saving energy consumption for concurrent GPU kernels. KSRE extracts the features of a kernel, classifies it and obtains and potential energy savings using a regression model in order to schedule any task. In integrated CPU–GPU architectures, all the different automatic power management implementations by hardware vendors hinders the resource usage optimization, and many are not exposed to the end–user. A black–box approach was introduced [14] to avoid these issues, partitioning work across the CPU and GPU cores.

TABLE 1
Comparative Between Optimization Techniques in Parallel Environments

Author	Optimization Metric	Description	Remarks
Singh *et al.* [10]	Energy efficiency	Contention–aware duplication based mixed integer programming model for scheduling task graphs on heterogeneous multiprocessors.	Uses Mixed Integer Programming to optimize energy efficiency
Chen *et al.* [11]	Performance, Energy efficiency	Quantum–inspired hyper–heuristic for energy–aware scheduling on heterogeneous computing systems	Addresses energy-constrained performance optimization and performance-constrained energy optimization
Hamandawana *et al.* [12]	Performance and Energy efficiency	Enhanced Phase-based Performance Aware Dynamic Scheduler (EPPADS) proposed as an alternative to MapReduce schedulers	Coordinates scheduling with power saving management schemes to improve energy usage
Li *et al.* [13]	Performance, Energy efficiency	Kernel scheduling approach for reducing energy consumption (KSRE) of concurrent kernels in GPU environments	Extracts the features of the kernels, classifies them, and obtain potential energy savings through a regression model
Barik *et al.* [14]	Energy efficiency	Black-box scheduling technique to improve energy usage by partitioning work in System–on–chip CPU–GPU arquitectures across both the CPU and GPU cores	Combines a power model with runtime information of a specific workloads
Cabrera *et al.* [6]	Generic Single Objective	Generic Single-Objective Heuristic to tackle a target objective function using dynamic load balancing in iterative problems	Requires defining the target single objective function
Garzón *et al.* [15]	Energy consumption	Load balancing algorithm for System–on–chip CPU–GPU arquitectures	Two–step procedure to determine allocation of processes and workload distribution
Reddy *et al.* [16]	Performance, Energy consumption	Bi–objective optimization technique for multicore homogeneous clusters	Models the target system to estimate optimal workload distributions
Rodríguez-Gonzalo *et al.* [17]	Energy consumption, Performance–per–watt	Dynamic spawn of MPI processes to optimize the chosen metric in a parallel application	Models the target system using performance counters

Our contribution, built as the *UllMF* framework, is a technique to use multi–objective objective functions in a dynamic load balancing algorithm using metrics gathered at runtime. The objective of some previous works align with ours, however, the technique we propose targets completely different algorithms that are provided as a high level library instead of scheduling tasks directly. This high level library only deals with workloads reassignments for data already distributed among the processes. Task scheduling use to involve more complex strategies to manage processes or tasks. This line of work is closer to other techniques presented in the literature.

In E-ADITHE [15] a technique for redistributing workload between processors is applied. Prior to the load balance phase, it executes a heuristic to select the optimal number of processing units in system–on–chips (SoCs), applying a two step algorithm. Our algorithm tries to optimize using all the available processes in the parallel program and is integrated as a fully developed framework, usable by non–expert users. ALEPH [16] is another energy–aware optimization technique, where performance and energy consumption are addressed as a bi–objective optimization problem. This technique models the objective system to estimate an optimal workload distribution in a many core device.

Still, our proposal is a novel multi–objective dynamic load balancing approach in iterative problems, where the objective functions can be dynamically modified at runtime.

Programming Skeletons is a different technique that is based on the development of highly efficient generic structures designed to abstract the programmer from the underlying system architecture. SkelCL [23] and Marrow [24] are skeletons that generate OpenCL code. SkePU2 [25] is a more modern approach that uses C++11 constructs to jtarget multiple heterogeneous architectures, including OpenMP, CUDA and OpenCL. Our heuristic engine conceptually follows an approach similar to the generic frameworks Mallba [26], ParadisEO [27], jMetal [28] and Metco [29] that allow a flexible design of metaheuristics, or the DPSKEL [30] skeleton,that provides a generic solver for dynamic programming algorithms. By only defining the specifications of a problem, these frameworks are able to provide implementations for various parallel architectures. The proposed heuristic engine apply the same principle in an smaller scope, abstracting the user from the objective function to solve. All these solutions have as common defining characteristic that they isolate specific problem details from the algorithm resolution steps.

In *UllMF*, we implement the heuristic engine as a module to perform dynamic load balancing and, following the flexible design ideas from these works, isolates metric gathering, the algorithms and objective functions to maximize code reuse.

3 Heuristic Algorithm

We propose a new Multi–Objective approach to address the dynamic load balancing problem, using various optimization objectives. A generic heuristic engine has been developed as an adaptive technique that dynamically exchanges different objectives. Whether the optimization goal is to improve performance, reduce energy consumption or achieve multiple objectives, the objective function can be selected at runtime. This technique lessens the impact of asynchronous metric gathering as better and faster metrics could be used to find improved workload distributions in earlier stages of a parallel process.

Algorithm 1 illustrates the generic structure of the parallel iterative method where our heuristic is able to improve the workload distribution. This type of algorithm requires to determine the amount of work assigned to every process (the workload), and in the case of using a Message Passing implementation, an explicit gathering of the data after each computed iteration. Then, for each state of the problem, work has to be redistributed, solved, and gathered back to all processes to continue the following iteration. Workload has to be redistributed at each state due to the total workload variability that certain iterative problems show in between iterations. Gathering the results forces a synchronization phase where every process has to wait until all the current work is finalized. In what follows we will assume a Message Passing implementation based on MPI.

Algorithm 1. Iterative Algorithm Structure

```
1: procedure ITERATIVE PROBLEM
2:   for all process p do
3:     workloads[p] ← 1/num_procs
4:   for all state ∈ IterativeAlgorithm do
5:     distribute_work(workloads)
6:     solve(workloads[p], state)
7:     gather_results(workloads)
```

This scheme presents disadvantages for homogeneous workload distributions, which could be unbalanced if any of the following characteristics are present:

- The problem to solve has an irregular nature. Two of the problems presented in our computational experience have this behavior. An homogeneous workload could be problematic, causing long waiting times in the synchronization phase.
- Our target system has an heterogeneous architecture. The computational capabilities of each process can be different, leading to the same circumstance happening in regular problems.

Algorithms 2 and 3 illustrate how an iterative problem has to be modified to incorporate our heuristic. Applying a dynamic load balancing technique, we find better workload distributions to minimize the misuse of resources in these situations, which ensue the improvement of our desired objective. Based on the principles of the metaheuristic frameworks, the problem constraints are isolated from the procedures to search different workload distributions. This characteristic allows to apply the same algorithm structure to improve any of the objectives we intend to optimize, be it energy consumption, execution time, or any other resource required by the user.

Algorithm 2. Heuristic Applied to an Iterative Algorithm

```
1: procedure ITERATIVE PROBLEM
2:   for all process p do
3:     workloads[p] ← 1/num_procs
4:   search_distance ← 1/num_procs
5:   reset_probability ← smallvalue
6:   for all state ∈ IterativeAlgorithm do
7:     distribute_work(workloads)
8:     solve(workloads[p], state)
9:     gather_results(workloads)
10:    measurements ← gather_resource_usage()
11:    if search_distance < threshold then
12:      if random() < reset_probability then
13:        reset_probability ← smallvalue
14:        search_distance ← 1/num_procs
15:        workloads ← heuristic_search(
16:          workloads, measurements, search_distance
17:        )
18:      else
19:          Increase(reset_probability)
20:    else
21:      workloads ← heuristic_search(
22:          workloads, measurements, search_distance
23:      )
```

Algorithm 3. Heuristic Search Algorithm

```
1: function HEURISTIC_SEARCH                  ▯ Input: workloads
2:                                            ▯ Input: measurements
3:                                            ▯ Input: search_distance
4:                                            ▯ Output: new_workloads
5:   for all process p do
6:     resources[p] ← measurements[p]/workloads[p]
7:   candidates ←
8:     generate_distributions(num_procs, search_distance)
9:   new_workloads ← nil
10:  best_resource_eval ← ∞
11:  for all c ∈ candidates do
12:    c_resource_eval ←
13:      evalue_workload_distribution(resources, c)
14:    if c_resource_eval < best_resource_eval then
15:      new_workloads ← c
16:      best_resource_eval ← c_resource_eval
17:  return new_workloads
```

The following notation was used in Algorithms 2 and 3:

- num_procs, total number of parallel processes in the execution.
- p, a parallel process.
- $workloads$, vector of workload allocated to each processor.
- $search\_distance$, maximum workload movement for each processor.
- $threshold$, smallest workload movement allowed.
- $reset\_probability$, probability of restarting the dynamic load balancing technique.
- $resources$, vector of resource usages per workload unit assigned to each processor.
- c, candidate workload distribution.
- $c_{resource\_eval}$, estimated resource usage of a candidate workload distribution c.

Using our heuristic in an iterative problem, as presented in Algorithm 2, does not require to modify the iterative algorithm. The execution starts from an homogeneous workload distribution, and the core behavior of the execution (distribute, solve and gather the partial results) remain unmodified. Once an iteration is executed, we gather the metrics related to one or many of the objectives we are optimizing. After the metrics are gathered, the main procedure of our method is applied and a new workload distribution is obtained. Initially, the solution space is explored broadly to avoid local optima. Subsequent iterations reduce the search distance for a new solution, until it is settled near to an optimal workload distribution. This behavior is controlled in the $generate\_distributions$ function. A threshold is set as a stop condition, which determines the optimality of our workload distribution. As this methodology is not deterministic, we require to establish a limitation to avoid very fine corrections of the solution. Without it, the heuristic algorithm would continue for negligible improvements, and the overhead of the heuristic search would hinder the overall performance of the iterative algorithm. This design is inspired by the Variable Neighborhood Search (VNS) [31] metaheuristic. Once the improved workload distribution is determined and the $search\_distance$ is smaller than the threshold, the heuristic stops.

Finally, as previously discussed, the irregularity of certain problems may cause an optimal workload distribution at a definite iteration, to be a poor quality solution in a subsequent state. Hence, we defined a mechanism to reinitialize the heuristic search with a certain probability. This reset is also inspired by a metaheuristic, the Simulated Annealing (SA) [32]. The SA metaheuristic accepts new solutions based on a probability, which decreases over time to reach a local optima. In this algorithm, we increase the reset probability as the execution progresses, to evaluate if the solution has deteriorated. The reset probability can be set to 0 in regular problems. However, in this work, we will address each problem as a completely unknown execution to illustrate the quality of our proposal.

In Algorithm 3, we present the steps to find a new workload distribution in each iteration of the iterative problem. First, we determine which are the candidate workload distributions based on the current search distance. Then, using the resource usage per unit of work assigned to each process, the candidate workloads are evaluated. The core of our contribution lies in how we have improved the evaluation of the different workload distributions. We use the resource usage to estimate the quality of each candidate, based on current measurements. As a result of the heuristic structure, we can apply multiple–objectives to evaluate workload distributions, which can also change dynamically to lessen the impact of difficult metric gathering, bad accuracy or changes in the restrictions determined by the user. $evalue\_workload\_distribution$ is a procedure that returns the resource consumption estimation of candidate workload distributions. This procedure, in fact, implements the objective to be optimized, so a dynamic exchange in this procedure would be enough to implement a dynamic objective function to optimize the desired resources, as long as appropriate measurements are gathered. In the next section, we will further explain how this is implemented to avoid gathering unnecessary metrics and reducing the overhead of our algorithm at the sametime.

In our experimentation, the resource estimation for the candidates can be improved, as using current measurements to estimate the future behavior of irregular problems could lead to worse solutions. However, we prove in this work our methodology improves the resource usage in a parallel system despite using simple estimations.

4 ULL MULTIOBJECTIVE FRAMEWORK

We present a tool to address the classic load balancing problem dynamically in iterative problems. Implemented in C, *UllMF* offers various mechanisms to cope with different objectives for workload redistribution between processors.

We considered the development of *UllMF* as we found multiple challenges to implement the heuristic algorithm from its pseudo–code. First, the end user codes have to be minimally modified. If we compare Algorithms 1 and 2, the amount of code intrusion in the iterative structure is very high, triplicating the amount of lines of code in the algorithm. Multiple metric gathering options also have to be implemented as well to use this heuristic. Energy metrics also entail additional difficulties to apply load balancing dynamically. Despite using existing state–of–art solutions from the scientific community, metrics are slow for a per–iteration usage, and measurement is often performed asynchronously. Finally, to put in practice the dynamic objective function, the heuristic implementation needs an interchangeable abstract method to evaluate workloads.

The main objective of *UllMF* is to provide abstraction over the underlying heterogeneous hardware and a portable tool to perform load balancing. *UllMF* offers a set of library calls that hide the specifics of measurement, providing portability to the experimental codes. The tool has several tasks to accomplish in order to deal with heterogeneity in the system. Two Single–Objective functions are implemented by default, minimizing energy consumption and the classic performance maximization, and one Multi–Objective approach using the energy delay product, EDP. The objective dynamic fuctions are obtained by combining the former.

Furthermore, it provides the implemented heuristic engine that can be modified with a custom objective function to fulfill specific needs for the user. To simplify its implementation, the framework is divided in multiple modules, illustrated in Fig. 1.

- The *user interface* provides the required functions and procedures to initialize the dynamic load balancing. It also provides the data structures required within its context. All the components of the library are named with the prefix `ullmf_`.
- The *strategy selector* allows the user to select the calibration modules to perform the load balancing at runtime. This module allows to tune parameters for the different calibration modules, or to swap load balancing algorithms during an execution.
- The *calibration* modules are different implementations of load balancing algorithms. Currently there

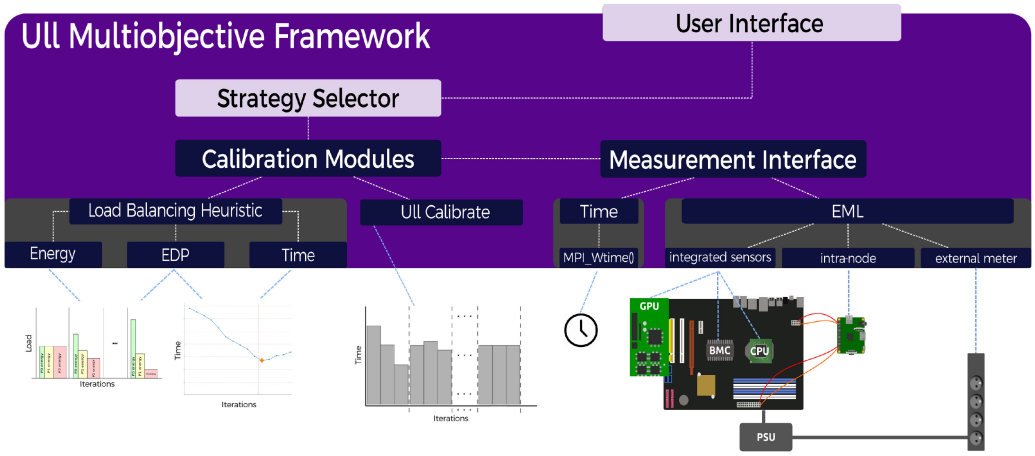


Fig. 1. *UllMF* component diagram.

are available four modules, two of which calibrate minimizing execution time, one for energy efficiency, and the Multi–Objective approach using EDP.

These modules are:

- An adaptation of the `Ull_Calibrate_Lib` load balancing technique, which redistributes workload proportionally to the processing capabilities of every processor in each iteration.
- An implementation of the heuristic algorithm for minimizing execution time. It shares the objective of the previous module, but searches the workload distribution using the algorithm from Section 3.
- An implementation of the heuristic algorithm for minimizing energy consumption.
- A multi–objective implementation of the heuristic, using the energy delay product (EDP).

The *measurement interface* module, which provides an abstraction to gather specific metrics. Currently, we only offer access to energy measurements trough the use of EML [33], although any other could be integrated. As our calibration modules work with energy and time, EML covers our needs with the aforementioned due to its capability to detect the available measurement devices at runtime, while the latter is covered through the use of `MPI_Wtime`. This interface is provided to allow easy implementations of personalized measurement devices or complex metrics. The energy heuristic implementation is portable due to this module, which provides generalization for measurements.

Fig. 1 also depicts the interconnection between all these modules through the white lines. The highlighted modules, *user interface* and *strategy selector*, are the ones used or configured by the final user to perform the load balance. The rest of the modules provide their functionalities without requiring to interact explicitly with them.

Listing 4.1 illustrates a basic example of *UllMF* usage. The code is structured to take advantage of iterative methods. Once initialized in line 6, a strategy has to be selected as shown in lines 7 and 8. The chosen strategy `ullmf_strategy_heuristic_energy` is the implementation of the heuristic algorithm for energy efficiency. When set, the memory and initial values are allocated using `ullmf_mpi_setup` in line 9, which needs as parameters the calibration structure `calib`, the amount of work `counts` for each process, the location of that work in the data array `displs` and the strategy. The dynamic load balance is then performed using the procedures `ullmf_mpi_start` and `ullmf_mpi_stop`, which modify the workload distribution stored within the variables `counts` and `displs`.

The `ullmf_mpi_start` and `ullmf_mpi_stop` procedures manage the flow of the calibration process, starting and stopping the appropriate measurements though the use of the calibration functions selected by the chosen strategy.

4.1 Strategy Selector

The *strategy selector* provided by *UllMF*, is connected through the *user interface* to provide the main functionalities to perform the dynamic load balancing. This interface hides the calibration modules from the user and provides access to them through the `ullmf_calibration_t` datatype. `ullmf_calibration_t` contains the *strategy selector* module options and decision making, composed by another data structure, `ullmf_strategy_t`. Specific parameters can be used to tune the modules within the strategy structure. `ullmf_calibration_t` uses `ullmf_strategy_t` to implement a strategy pattern in order to modify the calibration decision making at runtime.

Listing 4.2 illustrates with a dummy code this functionality. The strategies, once imported, can be changed within the code as desired, shown in lines 6 and 11. In this code, we make use of two strategies provided by *UllMF*. This kind of approach can be used to differentiate critical sections of an algorithm, that can be balanced for performance, and less important sections that can be rebalanced to improve energy efficiency. A user–defined strategy can also be provided in the same fashion, if the user implements its own development of `ullmf_strategy_t`. Notice how the calibration modules are hidden from the user, which are never used directly in the code. Complex decision making could be used inside the program to swap strategies. With this module, we can perform dynamic optimization by varying the objective function.

Listing 4.1. Calibration Code Example

```
// num_procs = Number of processors;
// id = ProcessID; N = ProblemSize
workload[id] = 1 / num_procs;
displs[id] = id * N * workload[id];
ullmf_calibration_t* calib;
ullmf_mpi_init();
ullmf_strategy_t* strategy =
 ullmf_strategy_heuristic_energy;
ullmf_mpi_setup(calib, workload, displs,
strategy);
for (i = 0; i < n; i++)
{
 ullmf_mpi_start(calib);
 counts[id] = N * workload[id];
 for (i = displs[id]; i < displs[id] + counts
 [id]; i++)
 {
  // ... Work ...
 }
 ullmf_mpi_stop(calib, workload, displs);
 MPI_Allgather(&problem[displs[id]], counts
 [id], ...);
 // ... Share results with other processes
}
ullmf_mpi_shutdown(); // Calibration module
finish
ullmf_mpi_free(calib);
```

Listing 4.2. Dynamic Objective Function Swap Example

```
#include ''ullmf.h''
#include ''ullmf/strategy_heuristic_time.h''
#include ''ullmf/strategy_heuristic_energy.h''
ullmf_calibration_t* calib;
// ... initialization ...
calib->strategy = ullmf_strategy_heuristic
_time;
ullmf_mpi_start(calib);
// ... Work ...
ullmf_mpi_stop(calib, counts, displs);
// Strategy change
calib->strategy = ullmf_strategy_heuristic_
energy;
ullmf_mpi_start(calib);
// ... More work ...
ullmf_mpi_stop(calib, counts, displs);
```

4.2 Calibration Modules

We provide four modules to be utilized by *UllMF* users. These load balancing strategies have simple logic and exist to select and tune some parameters of the chosen load calibration method.

4.2.1 Ull_calibrate_lib

The `Ull_Calibrate_Lib` module was designed based on the technique presented by their original authors in the literature [4]. This load balance strategy redistributes workload between the different processors depending on their performance solving a given problem. The speed of each process is calculated by measuring the time spent solving the assigned tasks, taking into account the amount of work given to every process and the size of the problem. This load balance technique is performed by proportionally redistributing the workload based on this previously calculated performance. The objective achieved through these operations is to minimize the time difference between the fastest and the slowest process at every iteration to reduce waiting times between iterations.

4.2.2 Heuristic Implementation: Energy, Time, and EDP

The provided heuristic modules are developed following the principles of skeleton programming. It differentiates from `Ull_Calibrate_Lib` in how new workload distributions are generated. The skeleton generates new candidates, the different workload distributions, and selects the best based on a user developed function, *evalue_workload _distribution*.

Listing 4.3. *UllMF* Heuristic Search Implementation

```
void heuristic_search(ullmf_calibration_t*
calib)
{
 ullmf_strategy_heuristic_t * heuristic =
  (ullmf_strategy_heuristic_t *) calib->strat
  egy;
 // Calculate resources per unit of work
 double resource_ratios[calib->num_procs];
 for (int i = 0; i < calib->num_procs; i++)
  resource_ratios[i] = get_resource_ratios(
    calib->measurements[i], calib->workload
    );
 // Generate heuristic population of candidates
 ullmf_workload_t** candidates;
 int num_candidates =
   generate_distributions(calib->workload, &
   candidates);
 // Evaluate heuristic population
 double best_resource_eval =
  heuristic->evalue_workload_distribution(
    calib, calib->workload, resource_ratios
    );
 double candidate_resource_eval;
 for (int i = 0; i < num_candidates; i++)
 {
  candidate_resource_eval =
   heuristic->evalue_workload_distribution(
     calib, candidates[i], resource_ratios
     );
  if (candidate_resource_eval < best_resource_
  eval)
  {
     candidate_resource_eval = best_resource_
     eval;
     calib->strategy->best_candidate = candida
     tes[i];
  }
 }
}
```

Part of the heuristic code is provided in Listings 4.3 and 4.4. These two functions, `heuristic_search` and

heuristic_calibrate are our implementation of Algorithms 2 and 3. The only user developed function, *evalue_workload _distribution*, is called in the heuristic_search procedure, in lines 17 and 24.

evalue_workload_distribution returns a numerical value, candidate_resource_eval, which quantifies the resource usage of the given workload, i. e., the amount of time spent or the energy consumed. This value is the variable $c_{resource\_eval}$ in the Algorithm pseudocode. In our implementaion, a smaller value of $c_{resource\_eval}$ represents a better candidate workload distribution.

Listing 4.4. *UllMF* Heuristic Dynamic Load Balancing Implementation

```
1  int  heuristic_calibrate(ullmf_calibration_
   t * calib)
2 {
3  // Energy measurements are time dependent
4  if (calib->strategy->mdevice->is_measuring)
5   return ULLMF_TAG_CALIBRATED;
6  ullmf_heuristic_heuristic_t* heuristic =
7    (ullmf_heuristic_heuristic_t*) calib->str
     ategy;
8   if (heuristic->search_distance < heuristic-
    >threshold)
9  {
10  // Search distance too small, trying to reset
11  double reset = random();
12  if (reset < heuristic->reset_probability)
13  {
14    heuristic->search_distance = heuristic-
      >reset_distance;
15    heuristic->reset_probability = heuristic-
      >initial_reset;
16  }
17  else
18  {
19    heuristic->reset_probability += heuristic-
      >increment
20    return ULLMF_TAG_CALIBRATED;
21  }
22 }
23 // Listing IV.3
24 heuristic_search(calib);
25 return ULLMF_TAG_RECALIBRATING;
26 }
```

The function described in Listing 4.3 is inside the heuristic_calibrate function that determines whether the heuristic gets executed or not, as shown in Listing 4.4. This function envelops the heuristic procedure, and is located within ullmf_mpi_stop in the experimental code. The general overview of the algorithm as a whole, is that it performs the following tasks, necessary for the dynamic load balancing:

- The generation of candidate workload distributions is inspired by the Variable Neighborhood Search. At the beginning of the problem, workload candidates are disperse i.e., with great variation respecting the current distribution. Less modifications are allowed between the candidates as the execution progresses.
- The stop condition, determined in the line 8 of Listing 4.4. The stop condition, related to the VNS, determines that the method should stop when the distance to the new workload distributions is very small.
- The Simulated Annealing (SA) inspired reset. The distance for the generated candidates used as stop condition resets with a probability, which restarts the heuristic dynamic load balance. Since we expect problem irregularity, a reset allows to search a new optimal workload distribution as it changes over time.

We also provide implementations for *evaluate_workload_distribution* to define how to interpret the metrics. For the *Energy Heuristic*, *evaluate_workload_distribution* returns the sum of all the energies consumed by each process. This consumption is estimated for the candidate workload with help of the current resource usage per unit of work. For the *Time Heuristic*, the implementation is the maximum execution time from all the processes.

4.2.3 Dynamic Objective Functions

As previously stated, all the algorithmic implementations can be changed during the execution of an algorithm. This functionality, illustrated in Listing 4.2, has been used as basis for implementing two complex dynamic load balancing techniques.

They have the objective of reducing the resources wasted during the first iterations of the iterative problems when the desired metric is energy consumption. These complex strategies are not included in *UllMF*, but have been implemented using the mechanisms described in this section.

The first and most simple implementation developed to diminish the waste of resources during energy measurement is *time–then–energy* (*TTE*). As its name indicate, the dynamic load balancing starts using time metrics to balance the workload using the heuristic algorithm. Once the search is finished and the restarting phase starts, the strategy is changed to perform the heuristic using energy metrics. With this procedure, we achieve an improvement of the overall energy consumption of an iterative problem, and reduce the startup impact caused by energetic measurement. Once the beginning phase is avoided, the energy heuristic search can be applied normally.

The second approach imitates the behavior of the EDP without using a Multi–Objective approach. To do so, we alternate between the time and energy heuristic implementations. Starting with the time algorithm to avoid the issues solved by the *TTE*, both methods are used alternatively once a reset phase of the algorithm is reached. A positive aspect of this technique, denominated *time–energy–switch* (*TES*), is that by optimizing both objectives alternatively we avoid local optima caused by each function individually.

5 Experimentation

UllMF and the proposals have been tested in an heterogeneous cluster composed by 4 GPU nodes. Table 2 has a summary of the characteristics of the hardware used in the experimentation. The kernel installed in each node is 4.9.0-6-amd64 #1 SMP Debian 4.9.82-1+deb9u3

TABLE 2
Experimentation Cluster

Nodes	CPUs (Xeon)	Memory	GPU	
Verode16	2x E5-2660	64 GB	M2090	
Verode17	2x E5-2660	64 GB	K20c	
Verode18	2x E5-2660	64 GB	K40m	
Verode20	2x E5-2698 v3	128 GB	M2090	
GPU	# Cores	RAM	Mem BW	Power
K20c	2496	5GB	208 GB/s	225 W
K40m	2880	12GB	288 GB/s	235 W
M2090	512	6GB	177.6 GB/s	225 W

(2018-03-02). The build and execution environments use the same software. Every library was compiled using GCC Version 4.8.5, OpenMPI 3.0.0 and the CUDA sdk version 7.5. Energy metrics were gathered using the NVidia Management Library driver (NVML) EML module.

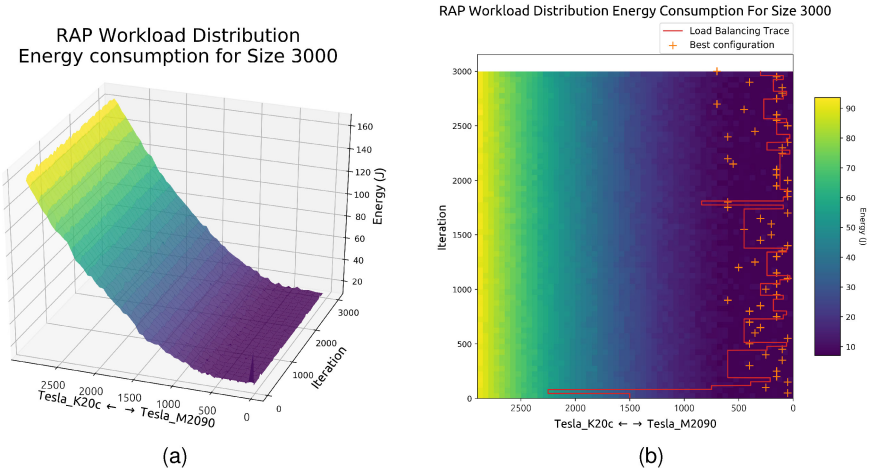
Our computational experience was gathered with multiple executions of four different iterative problems, implemented using dynamic programming: the Knapsack Problem (KP), the Resource Allocation Problem (RAP), the Triangulation of Convex Polygons (TCP) and the Cutting Stock Problem (CSP) [34]. The KP provides an example of a regular fine grained problem, where most of the performance is lost during the communication phases. The RAP exemplifies the regular counterpart, a compute bound, coarse grained problem. Additionally, its workload is irregular within each iteration, as every partial solution depends on the previous calculated ones. The TCP and CSP are also coarse grained, similar to the RAP. However, they differ in the workload distribution between each iteration. The dynamic programming table is filled diagonally, increasing the total amount of operations per iteration as the problem progresses. At the beggining of the problem, both CSP and TCP have very low computational requirements. As the problem progresses, the workload is increased until a maximum size is reached. Finally, the total workload diminishes until the problem is completely solved.

The implementations of these four dynamic programming algorithms were developed using DPSkel [35]. DPSkel already provides the solving mechanisms for various dynamic programming algorithms, and we only required to provide the problem specific GPU implementation and the *UllMF* code instrumentation. Since every process requires all the calculations performed in the previous iterations, data communication is solved through an `MPI_Allgather` call after the data has been copied from the GPU to the main memory. This communication phase forces a barrier, which we use to perform the load balancing phase to minimize the impact of the workload redistribution.

5.1 *UllMF* Implementation Analysis

In order to better understand how each different heuristic behaves, we have performed an extense study of the solution space for our objective algorithms. Figs. 2 and 3 illustrate a representative part of these studies in two different GPUs: a Tesla K20c and a Tesla M2090. In every surface, the axis labeled as *Tesla K20c* ← → *Tesla M2090* represent the workload assigned to one of the two processes, while the remaining workload to reach the problem size is assigned to the other one. This translates as, the left side of the chart gives all workload to process 0, the right side gives all workload to process 1, and the middle of the axis represents the homogeneous distribution. The axis labeled as *Iterations* depicts how the problem evolves during its execution. In Figs. 2b and 3b, the beginning of the execution is at the bottom while the last iteration is at the top of the chart. The energy consumed for each combination of workload distribution and current iteration is represented through the colormap, where darker colors represent less energy consumption. The elevation in the 3d surfaces, Figs. 2a and 3a, further illustrates the change in the workload through the space of the problem. Finally, the optimal workload distribution for each iteration is marked with the symbol + and an example of a load balancing algorithm is represented through a solid trace line.

Fig. 2a illustrates an execution of the RAP, size 3,000. In this Figure, we can observe the irregularity mentioned earlier within each iteration. It is not caused solely by the disparity in the computational capabilities of each GPU, but it is part of the problem definition. If the hardware was homogeneous, distributing the workload equally would not achieve an optimal usage of resources. The optimal distribution and the load balancing trace is illustrated in Fig. 2b, at the right region of the chart. Our heuristics reach the



(a) (b)

Fig. 2. Resource Allocation Problem (RAP) analysis. (a) RAP 3d solution space. (b) RAP solution space dynamic load balancing execution trace.

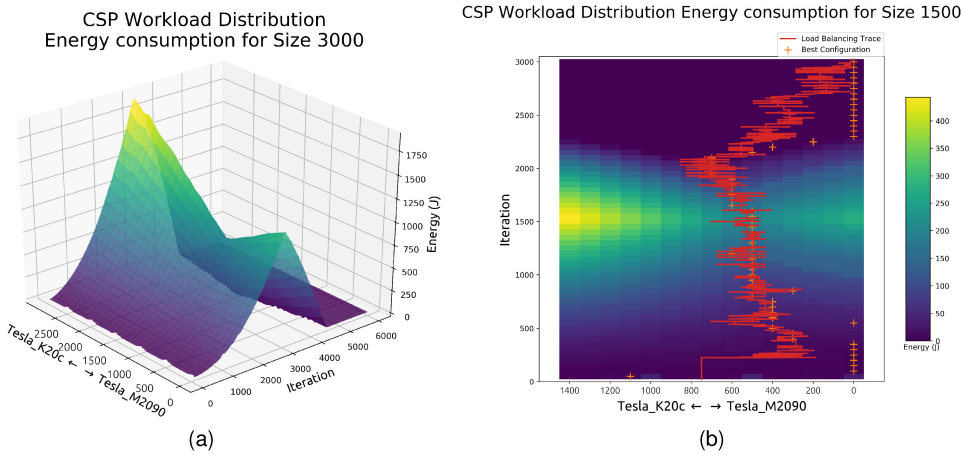


Fig. 3. Cutting Stock Problem (CSP) analysis. (a) CSP 3d solution space. (b) CSP solution space dynamic load balancing execution trace.

optimal workload distributions relatively fast. We could argue that a static load balancing technique would be sufficient for this problem and our methodology is not justified. However, a static workload balancing technique would require previous knowledge for the target problem, which is not the case for our methodology.

Fig. 3a represents a different case which fits better a dynamic load balancing technique. In the CSP executions, the optimal workload distribution is a trajectory in the solution space, instead of the presented RAP static region. The first iterations, the total workload is very small and using a single process yields the optimal energy consumption. As the execution progresses, the increasing total workload starts to impact the energy consumption and the optimal distribution changes. Once half of the problem is solved, the total workload starts to decrease until the single process is optimal again. Fig. 3b depicts the optimal distribution and the load balancing trace for the performance implementation of our heuristic. Using *UllMF* we are able to find workload distributions with very low overhead that approximates to the optimal trajectory.

We analyzed the overhead introduced by *UllMF* and its implementation. The results were gathered by performing executions of our heuristic algorithm and saving the workload distribution at each step of the problem, as a trajectory. Afterwards, these workload trajectories were used to execute the problems, without our library code. The obtained results, presented in Table 3, show that the overhead is caused by a constant cost of initialization, that is reduced as the problem size increases.

The smallest execution, of size 1,000, is around 7.5 percent, but as the size increases, the overhead settles around 1.4 percent for execution time. Energy consumption is affected similarly, ranging from 11.0 to 0.8 percent for the biggest problem size. We concluded that the overhead is negligible, even more when we consider the potential improvement of the resource usage shown in the following section.

TABLE 3
UllMF Overhead

CSP	Time (s)			Energy (J)		
	Manual	*UllMF*	Diff. (%)	Manual	*UllMF*	Diff. (%)
1000	7.15	7.74	7.5	1082	1216	11.0
1500	22.89	23.62	3.1	3631	3775	3.8
2000	52.63	53.38	1.4	8473	8620	1.7
2500	100.40	101.72	1.3	16351	16634	1.7
3000	170.52	172.95	1.4	28619	28850	0.8

5.2 Computational Results

Tables 4, 5, 6 and 7 contain the time and energy experimental measurements. The reference time, labeled as *Ref*, is gathered from the iterative problems executed without the *UllMF* instrumentation using an homogeneous distribution. The columns labeled as *Calib* show the experimentation performed with the adapted `Ull_Calibrate_Lib` module. Labels *EnerH* and *TimeH* represent the data gathered from the heuristic algorithm implementations for energy and time respectively. The Multi–Objective implementation is labeled as *EDP*, and finally, the dynamic objective functions, *TTE* and *TES* for the time–then–energy and time–energy–

TABLE 4
KP Experimental Data

KP	Time (s)						
	Ref	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
2000	**0.93**	1.03	1.03	1.04	1.03	1.01	1.08
4000	3.39	3.25	3.40	3.23	3.40	**3.05**	3.43
6000	7.12	6.74	7.12	6.78	7.13	**6.10**	6.87
8000	12.53	11.40	11.85	11.39	11.88	**10.11**	10.63
10000	19.05	18.67	19.13	18.38	19.17	**16.20**	20.15
12000	27.08	25.95	26.79	25.14	26.85	**22.62**	24.11
14000	35.48	34.76	35.49	33.78	35.57	**29.99**	31.58
16000	46.21	44.16	45.97	42.38	46.08	**37.57**	39.99

KP	Energy (J)						
	Ref	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
2000	**136**	146	148	150	148	144	155
4000	507	476	497	473	498	**447**	499
6000	1067	995	1049	999	1051	**899**	1010
8000	1882	1689	1756	1683	1759	**1496**	1566
10000	2873	2762	2831	2714	2838	**2394**	2971
12000	4064	3843	3971	3720	3979	**3340**	3558
14000	5364	5172	5300	5027	5312	**4441**	4681
16000	7037	6557	6849	6271	6865	**5589**	5947

Best results in bold.

TABLE 5
RAP Experimental Data

	Time (s)						
RAP	*Ref*	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
1000	1.36	0.77	1.10	0.66	0.67	0.69	**0.64**
2000	6.81	2.98	5.39	2.23	**2.16**	2.21	2.20
3000	20.13	7.21	5.04	4.76	4.75	**4.67**	5.18
4000	46.03	12.79	8.36	8.43	8.02	**7.79**	8.45
5000	87.78	17.44	12.25	12.37	13.38	**11.45**	11.99
6000	149.07	27.17	17.81	18.14	17.75	**16.68**	17.96

	Energy (J)						
RAP	*Ref*	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
1000	182	105	145	87	89	93	**84**
2000	967	424	750	309	**300**	312	302
3000	2850	1030	699	676	673	**666**	733
4000	6623	1853	1189	1226	1155	**1126**	1222
5000	12790	2607	1757	1830	1979	**1681**	1765
6000	21888	4011	2635	2716	2610	**2486**	2686

Best results in bold.

TABLE 6
TCP Experimental Data

	Time (s)						
TCP	*Ref*	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
500	0.60	**0.59***	0.65	**0.59***	0.60	0.61	0.62
1000	3.93	3.17	3.39	**3.12**	3.22	3.28	3.41
1500	12.51	8.45	9.36	**8.38**	8.74	9.07	8.97
2000	29.48	19.20	20.88	**19.02**	19.85	20.45	21.46
2500	55.20	38.37	41.48	**38.33**	39.99	41.13	40.73
3000	95.41	67.31	72.83	**66.99**	70.92	73.63	75.24
3500	149.73	109.77	118.23	**109.25**	115.53	118.70	119.85
4000	224.64	**166.87**	179.03	167.04	177.78	179.30	187.71

	Energy (J)						
TCP	*Ref*	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
500	**86***	87	96	87	87	89	92
1000	589	491	522	**483**	498	506	522
1500	1913	1325	1450	**1319**	1363	1415	1396
2000	4590	3064	3285	**3034**	3136	3230	3363
2500	8707	6183	6602	**6177**	6379	6556	6500
3000	15286	10956	11737	**10905**	11440	11856	12098
3500	24314	18066	19220	**17985**	18840	19325	19478
4000	37122	**27891**	29576	27902	29319	29686	30743

Best results in bold. '' indicates when a value is different for Time and Energy.*

switch implementations. The best performing algorithm uses a bold font to help understand these computational results. In what follows, resource usage will be addressed as $(time, energy)$ tuples.

The KP is the least compute intensive problem from the set. The experimentation clearly shows that for the smallest size, applying load balancing is slightly detrimental, as the workload redistribution and the library overhead demand more resources than the improvement achieved by applying any dynamic load balancing technique. As the problem size increases, resource usage improves, starting from size 4,000. Without considering the smallest case, as the absolute error is in a much smaller magnitude, resources are improved using the `Ull_Calibrate_Lib` method by (4.44, 6.11 percent), the *Energy Heuristic* by (0.91, 2.57 percent), the *Time Heuristic* by (6.04, 7.84 percent), the *EDP* by (0.69, 2.36 percent), the *TTE* dynamic objective function by (15.58, 17.19 percent) and, finally, the TES by (6.74, 8.70 percent). Table 4 presents the numerical data from these experiments. When the load balancing techniques are applicable, the best option is to apply the dynamic objective functions, as they improve the workload distribution much better than the mono–objective and Multi–Objective counterparts.

The RAP is our first compute intensive problem from the selected test cases. It introduces irregularity in the workload as every operation computes all the previous values, thus the cost of every operation increments with time. Thus, an homogeneous distribution will always be suboptimal. This behavior is reflected in the high improvements in resource consumption despite the technique used to perform the load balancing. Computational results, gathered in Table 5, show an average improvement of (75, 75 percent) when using any of the load balancing techniques proposed in this work. The *Energy Heuristic* improves only by an average of (61.76, 62.43 percent) due to its behavior in smaller problem sizes. However, when the size increases, its improvements are as notable as the other heuristic techniques. In this case, `Ull_Calibrate_Lib` only achieves an average of (66.31, 65.98 percent), and takes longer than the *Energy Heuristic* to reach the efficiency of the other algorithms. This experimentation also illustrates that the Multi–Objective approach and the dynamic objective functions hides the slow energy measurement, with *TTE* achieving the best solutions. These techniques also improve the solution provided by the *Energy Heuristic* by (29.07, 29.07 percent) However, compared against the *Time Heuristic*, only the *TTE* presents an improvement of the solutions, by an average of (3.54, 3.22 percent).

The next case, the TCP, introduces irregularity between iterations, as the workload changes in different phases of its execution. Table 6 illustrates that both `Ull_Calibrate_Lib` and *Time Heuristic* obtain similar results, with the *Time Heuristic* being slightly better than the `Ull_Calibrate_Lib`. `Ull_Calibrate_Lib` improves the homogeneous distribution by (28.43, 26.94 percent), while the *Time Heuristic* improves them by (28.87, 27.36 percent) on average. On the other hand, gathering energy consumption metrics require more time than measuring the execution time, the dynamic workload balance techniques that involve energy consumption will be a disadvantage in these kind of irregular problems. The *Energy Heuristic* improves the original distribution by (22.56, 21.82 percent). Using the *EDP* Multi–Objective approach, the impact of using slow metrics is reduced. Compared to the *Energy Heuristic*, it performs slightly better due to supporting the energy metrics with time. The *EDP* improves the solutions by (25.40, 24.48 percent). In this case, there is no clear advantage for using a dynamic objective function. Still, the original workload distribution is improved by (23.40, 22.51 percent) using *TTE* and by (21.77, 21.21 percent) using *TES*.

Finally, the CSP implementation, in Table 7, is a most compute heavy problem of the set, and introduces the biggest irregularities in the workload between iterations. The best solutions are again achieved by the *Time Heuristic*,

TABLE 7
CSP Experimental Data

	Time (s)						
CSP	*Ref*	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
500	1.46	1.58	1.75	**1.34**	1.45	1.50	1.76
1000	9.03	9.25	9.90	**7.74**	8.54	11.33	10.58
1500	29.59	28.76	32.68	**23.62**	26.59	29.52	29.76
2000	69.37	65.58	64.71	**53.38**	62.24	79.34	64.19
2500	136.93	127.64	116.37	**101.72**	107.31	137.38	111.77
3000	235.09	218.63	205.70	**172.95**	196.57	228.48	204.06
3500	374.26	341.82	323.97	**271.95**	287.83	356.24	305.98
4000	558.00	510.68	505.15	**405.19**	432.57	517.91	460.83

	Energy (J)						
CSP	*Ref*	*Calib*	*EnerH*	*TimeH*	*EDP*	*TTE*	*TES*
500	218	238	263	**203**	220	227	265
1000	1386	1434	1515	**1216**	1324	1757	1640
1500	4582	4518	5067	**3775**	4183	4666	4692
2000	10845	10434	10330	**8620**	10457	12659	10284
2500	21515	20546	18893	**16634**	18030	22388	18203
3000	37291	35871	34206	**28850**	33027	38213	33981
3500	59638	56682	54428	**45863**	48361	59855	51411
4000	89507	85515	86786	**69134**	72680	88979	78407

Best results in bold.

with an average improvement of (23.48, 20.23 percent). In the CSP, the extreme workload irregularity between iterations causes all the techniques based on energy measurements to react poorly to the change in total workload. *TTE* is the algorithm that is affected the most, causing a performance degradation of (-3.6, 7.38 percent). By applying the Multi–objective function, *EDP*, the effect of using energy metrics is softened and solutions are improved by (15.63, 11.72 percent). The rest of the algorithms improve the workload distribution only after a certain size of the problem, 1,000 for `Ull_Calibrate_Lib` and 1,500 for the *Energy Heuristic* and the *TES*. On average, `Ull_Calibrate_Lib` improves the workload distribution by (5.25, 2.78 percent), the *Energy Heuristic* by (5.29, 2.45 percent), and the *TES* by (8.13, 4.99 percent).

These experiments indicate that despite there are some differences, our architecture shows high correlation between the energy and performance metrics, and faster solutions yield the better energy efficiency. The algorithm that achieves the lower execution time also consumes less energy to execute the iterative problems. More over, improvements in % respecting the reference for each case differ by an average of 1 percent.

After reviewing the whole set of experiments, we can conclude that the size of the problem or the iteration irregularity does not affect the optimal metric to apply to a given problem. From the results gathered executing the KP and the RAP, we can observe that energy measurements are not the best option for dynamic load balancing. However, if the energy metrics are supported by time measurements to perform dynamic load balancing, workload distributions are greatly improved in both, energy consumption and performance. We show how the dynamic objective function *TTE* is strictly better.

The irregularity in between iterations heavily disrupts the dynamic load balancing algorithm for energy metrics as we can see with the TSP and CSP problems. By observing the heuristic trajectories from the previous section and the experimental results, we can conclude energy metrics are not suitable if this feature is present. The polling rate of energy measurements and its asynchronous nature makes using them appropriately in real time very difficult with this kind of constraints. Moreover, only the Multi–Objective approach is able to mitigate its downsides, and the dynamic objective functions algorithms are not able to solve this issue.

Finally, we can conclude that in the case of low variation among iteration workload Multi–Objective dynamic objective function *TTE* is recommended, while using the performance approach has presented the best results in the case of high irregularity.

6 Conclusion

We presented a Multi–Objective dynamic load balancing approach for heterogeneous architectures using multiple GPUs. Several objectives can be applied to tune an application and we proved they can be dynamically exchanged to improve the resource usage in iterative algorithms. All contributions are implemented in an open source tool, the *Ull Multiobjective Framework* (*UllMF*), using a generic heuristic engine designed to easily perform the presented strategies. In *UllMF*, metric gathering, algorithms and objective functions are isolated to maximize code reuse and provides a simple interface to reduce costs associated to custom user implementations. The whole experimentation set illustrates the strong and weak points of the presented techniques, which improve the execution time and the energy consumption when compared to executions using an homogeneous distribution. In problems where the workload is regular in between iterations, energy metrics are proven to be very useful, specially if they are supported with execution time metrics using the dynamic Multi–Objective function *TTE*. For problems where the workload is irregular in between iterations, energy metrics have a negative effect and, while the Multi–Objective approach mitigates it, the best results are obtained using our *Time Heuristic*.

In future work, we intend to study multiple opportunities that appeared with the development and study of *UllMF*. Our first goal is to analyze how energy metrics affect our different dynamic load balancing implementations on architectures where the best performance do not imply the best energy efficiency. For longer–term research possibilities, we will focus in the effects of different powercap technologies in dynamic load balancing algorithms for parallel applications.

Acknowledgments

This work was supported by the Spanish Ministry of Science, Innovation and Universities through the TIN2016-78919-R project, the Government of the Canary Islands, with the project ProID2017010130 and the Grant TESIS2017010134, which is cofinanced by the Ministry of Economy, Industry, Commerce and Knowledge of Canary Islands and the European Social Funds (ESF), operative program integrated of Canary Islands 2014-2020 Strategy Aim 3, Priority Topic 74(85 percent); the Spanish network CAPAP-H, and the European COST Action CHIPSET.

REFERENCES

[1] F. Almeida, J. Arteaga, V. Blanco, and A. Cabrera, "Energy measurement tools for ultrascale computing: A survey," *Supercomput. Front. Innov.*, vol. 2, no. 2, 2015. [Online]. Available: http://superfri.org/superfri/article/view/45

[2] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. W. Cameron, "PowerPack: Energy profiling and analysis of high-performance systems and applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 5, pp. 658–671, May 2010.

[3] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, Aug. 2000.

[4] A. Acosta, R. Corujo, V. Blanco, and F. Almeida, "Dynamic load balancing on heterogeneous multicore/multiGPU systems," in *Proc. Int. Conf. High Perform. Comput. Simul.*, 2010, pp. 467–476, isbn: 978–1-4244-6827-0.

[5] A. Cabrera, A. Acosta, F. Almeida, and V. Blanco, "Energy efficient dynamic load balancing over multiGPU heterogeneous systems," in *Proc. Int. Conf. Parallel Process. Appl. Math.*, 2017, pp. 123–132.

[6] A. Cabrera, A. Acosta, F. Almeida, and V. Blanco, "A heuristic technique to improve energy efficiency with dynamic load balancing," *J. Supercomput.*, vol. 75, no. 3, pp. 1610–1624, Mar. 2019. [Online]. Available: https://doi.org/10.1007/s11227–018-2718-6

[7] ULL high performance computing group, Ull multiobjective framework repository, 2019. Accessed: Jan. 16, 2020. [Online]. Available: http://github.com/HPC-ULL/UllMF

[8] J.-F. Cardoso and A. Souloumiac, "Jacobi angles for simultaneous diagonalization," *SIAM J. Matrix Anal. Appl.*, vol. 17, no. 1, pp. 161–164, 1996.

[9] H. Anzt, E. Chow, and J. Dongarra, "Iterative sparse triangular solves for preconditioning," in *Proc. Eur. Conf. Parallel Process.*, 2015, pp. 650–661.

[10] J. Singh, S. Betha, B. Mangipudi, and N. Auluck, "Contention aware energy efficient scheduling on heterogeneous multiprocessors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 5, pp. 1251–1264, May 2015. [Online]. Available: https://doi.org/10.1109/TPDS.2014.2322354

[11] S. Chen, Z. Li, B. Yang, and G. Rudolph, "Quantum-inspired hyper-heuristics for energy-aware scheduling on heterogeneous computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1796–1810, Jun. 2016. [Online]. Available: https://doi.org/10.1109/TPDS.2015.2462835

[12] P. Hamandawana, R. Mativenga, S. J. Kwon, and T. Chung, "Towards an energy efficient computing with coordinated performance-aware scheduling in large scale data clusters," *IEEE Access*, vol. 7, pp. 140 261–140 277, 2019. [Online]. Available: https://doi.org/10.1109/ACCESS.2019.2943632

[13] J. Li, B. Guo, Y. Shen, D. Li, and Y. Huang, "Kernel scheduling approach for reducing GPU energy consumption," *J. Comput. Sci.*, vol. 28, pp. 360–368, 2018. [Online]. Available: https://doi.org/10.1016/j.jocs.2017.11.013

[14] R. Barik, N. Farooqui, B. T. Lewis, C. Hu, and T. Shpeisman, "A black-box approach to energy-aware scheduling on integrated CPU-GPU systems," in *Proc. Int. Symp. Code Gener. Optim.*, 2016, pp. 70–81. [Online]. Available: https://doi.org/10.1145/2854038.2854052

[15] E. M. Garzón, J. J. Moreno, and J. A. Martínez, "An approach to optimise the energy efficiency of iterative computation on integrated GPU–CPU systems," *J. Supercomput.*, vol. 73, no. 1, pp. 114–125, Jan. 2017.

[16] R. Reddy and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Trans. Comput.*, vol. 67, no. 2, pp. 160–177, Feb. 2018.

[17] M. Rodr íguez-Gonzalo, D. E. Singh, J. G. Blas, and J. Carretero, "Improving the energy efficiency of MPI applications by means of malleability," in *Proc. 24th Euromicro Int. Conf. Parallel Distrib. Netw.-Based Process.*, 2016, pp. 627–634.

[18] E. Agullo *et al.*, "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects," *J. Phys., Conf. Ser.*, vol. 180, no. 1, 2009, Art. no. 012037. [Online]. Available: http://stacks.iop.org/1742–6596/180/i=1/a=012037

[19] The FLAME Project, "FLAME: Formal linear algebra methods environment," 2011. [Online]. Available: http://z.cs.utexas.edu/wiki/flame.wiki/FrontPage

[20] P. Richmond and D. Romano, "FLAME: Flexible large-scale agent modelling environment on the GPU," 2010. [Online]. Available: http://www.flamegpu.com/

[21] I. Takouna, R. Rojas-Cessa, K. Sachs, and C. Meinel, "Communication-aware and energy-efficient scheduling for parallel applications in virtualized data centers," in *Proc. IEEE/ACM 6th Int. Conf. Utility Cloud Comput.*, 2013, pp. 251–255.

[22] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 119–128.

[23] M. Steuwer and S. Gorlatch, "SkelCL: A high-level extension of OpenCL for multi-GPU systems," *J. Supercomput.*, vol. 69, no. 1, pp. 25–33, 2014.

[24] R. Marqués, H. Paulino, F. Alexandre, and P. D. Medeiros, "Algorithmic skeleton framework for the orchestration of GPU computations," in *Proc. 19th Int. Conf. Parallel Process.*, 2013, pp. 874–885.

[25] A. Ernstsson, L. Li, and C. Kessler, "SkePU 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems," *Int. J. Parallel Program.*, vol. 46, no. 1, pp. 62–80, Feb. 2018.

[26] E. Alba *et al.*, "MALLBA: A library of skeletons for combinatorial optimisation," in *Proc. Eur. Conf. Parallel Process.*, 2002, pp. 927–932.

[27] S. Cahon, N. Melab, and E.-G. Talbi, "ParadisEO: A framework for the reusable design of parallel and distributed metaheuristics," *J. Heuristics*, vol. 10, no. 3, pp. 357–380, 2004.

[28] J. J. Durillo and A. J. Nebro, "jMetal: A java framework for multi-objective optimization," *Advances Eng. Softw.*, vol. 42, pp. 760–771, 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0965997811001219

[29] C. León, G. Miranda, and C. Segura, "METCO: A parallel plugin-based framework for multi-objective optimization," *Int. J. Artif. Intell. Tools*, vol. 18, no. 04, pp. 569–588, 2009.

[30] I. Peláez, F. Almeida, and F. Suárez, "DPSKEL: A skeleton based tool for parallel dynamic programming," in *Proc. 7th Int. Conf. Parallel Process. Appl. Math.*, 2007, pp. 1104–1113.

[31] P. Hansen, N. Mladenović, and J. A. Moreno Pérez, "Variable neighbourhood search: Methods and applications," *Ann. Operations Res.*, vol. 175, no. 1, pp. 367–407, Mar. 2010.

[32] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[33] A. Cabrera, F. Almeida, J. Arteaga, and V. Blanco, "Measuring energy consumption using EML (energy measurement library)," *Comput. Sci. Res. Develop.*, vol. 30, no. 2, pp. 135–143, 2014.

[34] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA, USA: MIT Press, 2009.

[35] I. Galindo, F. Almeida, and J. M. Bad ía-Contelles, "Dynamic load balancing on dedicated heterogeneous systems," in *Proc. 15th Eur. PVM/MPI Users' Group Meet. Recent Advances Parallel Virtual Mach. Message Passing Interface*, 2008, pp. 64–74.

Alberto Cabrera received the BA and MS degrees in computer sciences, in 2010 and 2013, respectively. He is currently working toward the PhD degree at Universidad de La Laguna, Santa Cruz de Tenerife, Spain. His research interests include primarily in the areas of parallel system analysis and prediction, parallel computing, heterogeneous computing, and energy efficiency in high performance systems.

Alejandro Acosta received the BA and MS degrees in computer engineering from the University of La Laguna, Santa Cruz de Tenerife, Spain, in 2010 and 2011, respectively, and the PhD degree in computer science, in 2015. His research focus include heterogeneous systems, parallelization and optimization techniques, performance evaluation, and energy efficient algorithms.

Francisco Almeida received the BSc and MSc degrees in mathematics, and the PhD degree in computer science from the University of La Laguna, La Laguna, Spain, in 1989, 1992, and 1996, respectively. He is currently a professor with the Department of Statistics and Computer Science, University of La Laguna. His research interests include primarily in the areas of parallel computing, parallel algorithms for optimization problems, parallel system performance analysis and prediction, skeleton tools for parallel programming, and web services for high performance computing and Grid technology.

Vicente Blanco received the BA and MS degrees in physics from the University of Santiago de Compostela, Santiago, Spain, in 1992 and 1993, respectively, and the PhD degree in physics, in 2002. In October 2000, he became an assistant professor with the Department of Statistics and Computer Science, University of La Laguna. Currently, he is an associate professor with the same department. His research interests include performance evaluation, prediction and visualization of parallel codes, parallel algorithms for dense and sparse algebra, Grid and GPGPU tecnology, and Energy aware algorithms/systems.