

A semi-Automatic Approach for Parallel Problem Solving using the Multi-BSP Model

M. Alaniz^{a,*} and S. Nesmachnow^{a,**}

^aUniversidad de la República Herrera y Reissig 565, Montevideo, Uruguay

*e-mail: marcelo.alaniz@fing.edu.uy

**e-mail: sergion@fing.edu.uy

Received September 20, 2019; revised October 6, 2019; accepted October 16, 2019

Abstract—The Multi-Bulk Synchronous Parallel (Multi-BSP) model is a recently proposed parallel programming model for multicore machines that extends the classic Bulk Synchronous Parallel model. Multi-BSP aims to be a useful model to design algorithms and estimate their running time. This model heavily relies on the right computation of parameters that characterize the hardware. Of course, the hardware utilization also depends on the specific features of the problems and the algorithms applied to solve them. This article introduces a semi-automatic approach for solving problems applying parallel algorithms using the Multi-BSP model. First, the specific multicore computer to use is characterized by applying an automatic procedure. After that, the hardware architecture discovered in the previous step is considered in order to design a portable parallel algorithm. Finally, a fine tuning of parameters is performed to improve the overall efficiency. We propose a specific benchmark for measuring the parameters that characterize the communication and synchronization costs in a particular hardware. Our approach discovers the hierarchical structure of the multicore architecture and compute both parameters for each level that can share data and make synchronizations between computing units. A second contribution of our research is a proposal for a Multi-BSP engine. It allows designing algorithms by applying a recursive methodology over the hierarchical tree already built by the benchmark, focusing on three atomic functions based in a divide-and-conquer strategy. The validation of the proposed method is reported, by studying an algorithm implemented in a prototype of the Multi-BSP engine, testing different parameter configurations that best fit to each problem and using three different high-performance multicore computers.

DOI: 10.1134/S0361768819080103

1. INTRODUCTION

The BSP model was introduced in the late 1980's as a programming model for distributed computers assuming only one processing unit (i.e., core) per computing node [9]. Although the model was very successfully used in the 1990s, it gradually became less used with the emergence of new multicore architectures in the last decade. As the evaluation of computers gained renewed importance, the BSP model was extended to Multi-BSP [10]. Multi-BSP extends BSP in two ways: i) it is a hierarchical model, with an arbitrary number of components, taking into account the physical structure of multiple memory and cache levels within single chips as well as in multi-chip architectures; and ii) at each level, Multi-BSP incorporates memory size as an additional parameter in the model, which was not included in the original BSP model.

By including specific features that model nowadays parallel architectures, Multi-BSP offers a more comprehensive model, which allows designing efficient parallel algorithms. In this line of work, the research

reported in this paper is focused on developing a full-stack approach for designing and implementing parallel applications over the Multi-BSP model using state-of-the-art tools, considering not only the performance but also the portability of the algorithms. The full stack development implies a characterized hardware architecture, a pattern for algorithm design, a cost prediction function for the algorithm, and a specific methodology for implementation over the parallel hardware.

The main contributions of the research reported in this article are: i) the proposal of a specific methodology for discovering the hierarchical structure of a multicore architecture and benchmarking the parameters that characterize the communication and synchronization costs in a particular parallel hardware; ii) a Multi-BSP engine allowing the design of algorithms by applying a recursive Divide-and-Conquer (D&C) pattern over the hierarchical tree already built by the benchmark is introduced; and iii) the validation of the proposed approach by using an algorithm implemented in the Multi-BSP engine, evaluating different parameter configurations that best fit to each problem

to solve and using three different high performance multicore computers, including a Xeon Phi coprocessor, which has not been focus of similar studies previously. Part of the research reported in this article was developed within the project “Scheduling evaluation in heterogeneous computing systems with hwloc” (SEHLOC). The main goal of the SEHLOC project consisted in developing runtime systems that allow combining characteristics of the software applications and topological information of the computational platforms, in order to get scheduling suggestions to profit from software and hardware affinities and provide a way for efficiently executing realistic applications. This article extends our previous conference paper “MBSP-Discover: an automatic benchmark for Multi-BSP performance analysis” [1], where the first ideas about implementing a Multi-BSP benchmark was presented.

This article proposes a more comprehensive approach and introduces an engine for Multi-BSP that considers the whole design process for parallel algorithms.

The article is organized as follows. Section 2 introduces the BSP and Multi-BSP models, and the automatic tool to discover the architecture features in order to assure portability. Section 3 describes relevant related work on BSP benchmarking and the design and implementation of the proposed MBSPDiscover benchmark. Section 4 describes the advantages of designing algorithms using Multi-BSP and the proposed engine, focusing on a recursive pattern design for parallel applications to assure portability. A cost function is developed using the Multi-BSP cost model. The engine is instantiated for a simple algorithm and evaluated over three parallel machines, characterized using the proposed benchmark, in Section 5. Finally, Section 6 presents the conclusions and formulates the main lines for future work.

2. THE BSP AND MULTI-BSP MODELS

To set the scope of this article, this section describes the BSP and Multi-BSP models. A brief description of the flat BSP model is presented, and how the model evolved into the concept of multicore, which emphasizes on hierarchies of components. The analytical methods for model prediction, which are needed to understand the foundations of both models, are described next. After that, the Multi-BSP cost function is described. At the end of this section, we argue about the need of an automatic process to discover the architecture features in order to assure portability, and the use of a specific software package is described.

2.1 The Original BSP Model

The BSP model considers an abstract parallel computer, which is fully modeled by a set of parameters:

the *number of available processors* (p), the *processor speed* (s), the *communication cost* (g), and the *synchronization cost* (l). Using these parameters, the execution time of any BSP algorithm can be exactly computed.

In the BSP model, the computations are organized in a sequence of global supersteps, which consist of three phases: i) every participating processor performs local computations, i.e., each process can only make use of values stored in the local memory of the processor; ii) the processes exchange data between themselves to facilitate remote data storage capabilities, and iii) every participating process must reach the next synchronization barrier, i.e., each process waits until all other processes have reached the same barrier. Then, the next superstep can begin.

In practice, the programming model is Single Program Multiple Data (SPMD), implemented as several C/C++ program copies running on p processors, wherein communication and synchronization among copies are performed using specific libraries such as BSPlib [5] or PUB [3]. In addition to defining an abstract machine and imposing a structure on parallel programs, the BSP model provides a cost function modeled by the architecture parameters.

The total running time of a BSP program can be computed as the cumulative sum of the cost of its supersteps, where the cost of each superstep is the sum of three values: i) w , the maximum number of calculations performed by each processor; ii) $h \times g$, where h is the maximum number of messages sent/received by each processor, with each word costing g units of time; and iii) l , the time cost of the barrier synchronizing the processors. The effect of the computer architecture is included by the parameters g and l . These values, along with the processor speed s , can be empirically determined for each parallel computer by executing benchmark programs at installation time.

2.2 The Multi-BSP Model

Modern supercomputers are made of highly parallel nodes with many cores. The efficiency of these nodes demanded specific improvements of the memory subsystem by adding multiple hierarchical levels of caches as well as a distributed memory interconnect, which lead to Non-Uniform Memory Access (NUMA).

In 2010, Valiant updated the BSP model to account for this situation, resulting in the Multi-BSP model [10]. The same abstractions and bridge architecture used in the original BSP were adapted to multicore machines in Multi-BSP, which describes a model instance as a tree structure of nested components, where the leaves are processors and each internal node is a BSP computer with local memory or some storage capacity.

Formally, a Multi-BSP machine is specified by a list of levels. Each level is described by four parameters (p_i, g_i, L_i, m_i):

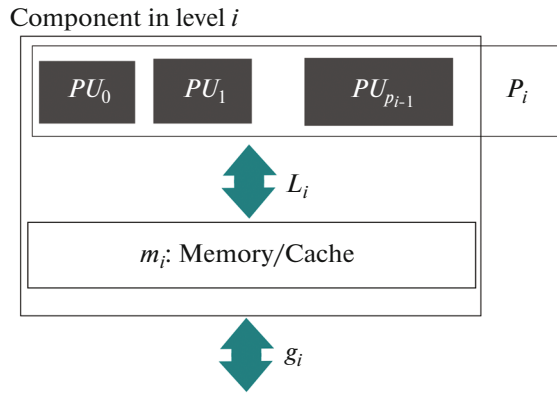


Fig. 1. Schematic view of a component on level i of the Multi-BSP model.

p_i is the number of components of the level $(i - 1)$ inside a component on level i . For $i = 1$, the components in the first level are p_1 raw processors, which can be regarded as the components of the level 0. One computation step of such a processor on a word in the memory in level 1 is taken as one basic unit of time.

g_i is the communication cost parameter, defined as the ratio of the number of operations that a processor can perform in a second and the number of words that can be transmitted in a second between the memories of a component at level i and its parent component at level $(i + 1)$. A *word* is defined as the amount of data on which a processor operation is performed. The model assumes that the memories in level 1 are located with the processors, and hence that the data rate (corresponding to the value of g_0) has the value one.

L_i is the cost for the barrier synchronization for a superstep in level i . This definition requires barrier synchronization of the subcomponents of a component, but no synchronization across above branches in the component hierarchy.

m_i is the number of words of memory inside a component in the level i that is not inside any component in level $(i - 1)$.

Figure 1 shows a component of level i in the Multi-BSP model. The superstep of level i is a set of instructions executing inside of a component located at level

i , which allows each of its p_i components at level $(i - 1)$ to execute independently, including all level $(i - 1)$ supersteps. Once all the p_i components finish their computation, they can exchange information with the memory m_i of the component in level i . This operation has a communication cost determined by $g_i - 1$. The cost charged will be $m \times g_i - 1$, where m is the maximum number of words communicated between the memory of the component in level i and anyone of its subcomponents in level $(i - 1)$. After executing the barrier that synchronize all the p_i components, the next superstep may begin.

As an example, Fig. 2 shows the diagram of a computer, whose architecture can be specified by three Multi-BSP components (level₀, level₁, and level₂): (1, 0, 0, 0), (4, g_1 , L_1 , 5118KB) and (8, g_2 , L_2 , 64 GB). We can ignore the level₀, because it represents only one processing unit and thus it does not involve internal synchronization or communication. Therefore, the computer has two components, which corresponds to the two level of hierarchy in the architecture.

2.3 Cost Prediction for the Multi-BSP Model

Like other abstract computational models, one of the main goals of Multi-BSP is to provide a precise notion of the execution time for a computer program. This subsection presents the mathematical formulation for the execution cost model, based on the full definition based on the operational semantics by Yzelman [11]. Later on this article, Section 4.3 introduces a simplification of this general formulation and provides a detailed definition for the Multi-BSP Engine we propose as the main contribution of our work.

The cost prediction on a specific computer is expressed in terms of computing, data movement, and latency, according to the expression in Equation 1, where L corresponds to the number of levels in the Multi-BSP tree, N_k is the number of supersteps on k^{th} level, $h_{k,i}$ is the maximum of all h -relations within the i^{th} superstep on level k , and $w_{k,i}$ the maximum of all work within the i^{th} superstep on level k .

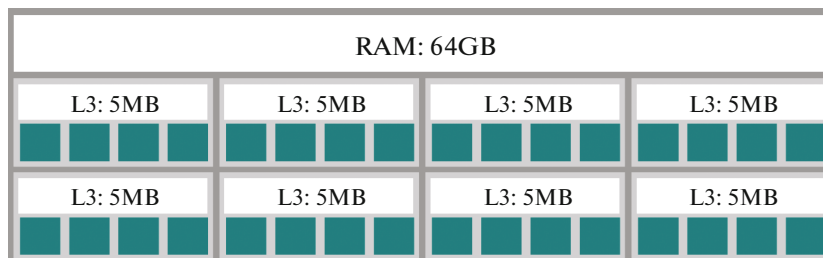


Fig. 2. Multi-BSP model: (4, g_1 , L_1 , 5118KB), (8, g_2 , L_2 , 64 GB).

$$T = \sum_{k=0}^{L-1} \left(\sum_{i=0}^{N_k-1} w_{k,i} + h_{k,i} \times g_k + l_k \right) \quad (1)$$

The formula in Equation 1 corresponds to the sum of the costs of the supersteps for each Multi-BSP component k . The cost of an individual superstep is split in two independent terms: the computation costs and the communication costs. The communication costs include the cost for synchronization (l_k , always one per superstep) and the term $h_{k,i} \times g_k$, which depends of the amount of put/get operations between threads, formally defined by the concept of h -relation. The superstep execution is sequential within each component, and inside a superstep each thread works in parallel. Thus, the values $h_{k,i}$ and $w_{k,i}$ correspond to the maximum of all h -relations within the i -th superstep on level and the maximum of all work within the i -th superstep on level k , respectively.

To guarantee portability, a full-stack development using Multi-BSP needs to use a procedure for discovering the properties of the underlying computer architecture. The Multi-BSP benchmark use the *portable HardWare LOCALity* (hwloc) tool [4] for discovering the underlying hardware features. hwloc allows obtaining runtime information about a computer. We use version 1.7.2 of hwloc, which provides a portable abstraction (across OS, versions, architectures, etc.) of the hierarchical topology of modern architectures, including processors, NUMA memory nodes, sockets, shared caches, cores and locality of I/O devices.

3. THE DISCOVERING AND BENCHMARKING TOOL FOR Multi-BSP

Multicore architectures are widely used for developing and executing HPC applications [6]. Both the number of cores and the cache levels in a multicore architecture have been steadily increasing in the last years. Therefore, there is a real need to identify and evaluate the different parameters that characterize the structure of cores and memories, not only to understand and compare different architectures, but also for using them wisely for a better design of HPC applications. This characterization is motivated by the fact that the performance improvements when using a multi-core processor strongly depend on software algorithms, their implementation, and the utilization of the hardware capabilities.

In the Multi-BSP model studied in this article, the performance of a parallel algorithm depends on parameters that characterize a multicore machine, such as communication and synchronization costs, number of cores, and the size of caches. Building analytical equations to compute those parameters is a hard task. Therefore, performing computer benchmarking

is a viable method to evaluate performance and characterize the architecture.

This section presents a review of related works about benchmarks for the BSP and Multi-BSP models, and the design and implementation of a specific discovering and benchmarking tool (Multi-BSP-Disc-Bench) to estimate the g and L parameters that characterize a Multi-BSP machine.

3.1. Related Work

The BSPbench program from the BSPedupack suite [3] has been the main benchmarking tool for the BSP model. This benchmark measures a full h -relation, where every processor sends and receives exactly h data words. The methodology tries to measure the slowest possible communication, putting single data words into other processors in a cyclic fashion. This reveals whether the system software indeed combines data for the same destination and whether it can handle all-to-all communication efficiently. In those cases, the resulting value of parameter g obtained by the BSPbench benchmarking program is called pessimistic.

The Oxford BSP toolset [5] includes another benchmarking program for BSP, bspprobe. This benchmark measures optimistic g values using larger packets instead of single words. Another option for BSP benchmarking is using the mpibench from MPIdupack [5].

The benchmarking of the Multi-BSP computational model has been recently addressed by Savadi and Hossein [7], using a closely-related approach as the one we apply in this article. The classic BSP benchmarking is used as a baseline, but the specification of a model instance is different. Unlike the benchmarking methodology followed in our work, the authors consider deep architecture details such as cache coherency, for instance for propagation of values in the memory hierarchy. In their approach, the analysis of results is made by comparing the real values obtained by the process of benchmarking against theoretical values of the g and L parameters, which are computed as optimistic lower bounds (i.e. the authors suppose that the memory utilization is always lower than the cache size, and that all cores work at maximum speed). Our approach differs since we do not make any assumption about the underlying hardware platform but rather hide its characteristic inside the output of the benchmarks. We believe this strategy is well suited to modern architectures that are too complex for precise models depending on their advanced, hidden and/or rarely well documented features.

From a practical point of view, the main feature of the discovering and benchmarking tool we propose in this article is to evaluate real Multi-BSP operations implemented for the library MulticoreBSP for C [12]. In addition, our results are validated using a set of real Multi-BSP programs, comparing the real execution

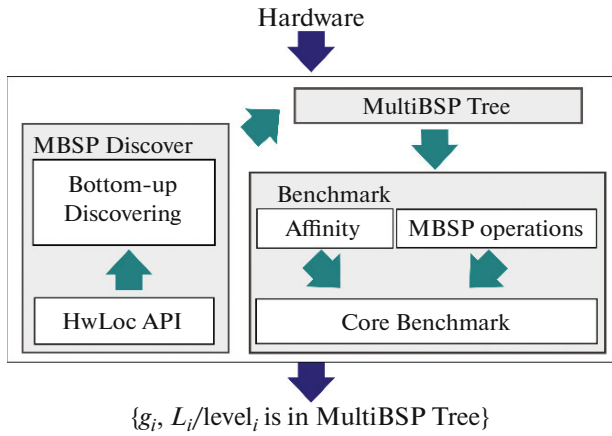


Fig. 3. Schematic view of the software architecture of Multi-BSP-Disc-Bench, and the discovering and benchmarking process.

time against the predicted running time using the theoretical Multi-BSP cost function, over several HPC platforms.

3.2. Design of the Multi-BSP-Disc-Bench Tool

We use general ideas from the existing benchmark for the standard BSP model, BSPbench [3], and extend the benchmark for Multi-BSP programs to design the proposed Multi-BSP-Disc-Bench tool. The main difference between the existing benchmark and the new one is the need of obtaining pairs of values for the g and L parameters for each level of components in the Multi-BSP model. In addition, in the Multi-BSP case, the processing is made inside of multicore nodes instead of outside nodes through the network.

It is important to emphasize that the quality of a benchmarking tool should not depend on a particular architecture. This extra requirement is solved by automatically discovering how the different cores are related within each level of cache. Another relevant goal for the Multi-BSP benchmark is to discover the

architecture features in run time. For this reason, we use the hwloc tool.

The components of the proposed benchmark are described in the following subsections.

3.2.1. Software architecture and modules. Figure 3 presents a schema of the software architecture and modules of the proposed Multi-BSP-Disc-Bench tool.

The functionality of the modules in Multi-BSP-Disc-Bench are as follows:

- Discovering module (*Multi-BSP Discover*). This module collects data about the hardware architecture and features using hwloc and loads the data into a tree of resources (a memory structure defined inside the hwloc API box).
- Interface (*Multi-BSP Tree*). Once the structure containing the information about resources is generated, a set of functions in the Interface walk across the tree, using a bottom-up process for building a new tree named Multi-BSP Tree. This new tree contains all the information needed to support the Multi-BSP model.
- Benchmarking module (*Multi-BSP Bench*). This module retrieves the core indexes and the memory size from the Multi-BSP Tree for each level. After that, it measures the communication cost and the synchronization cost through a Multi-BSP submodule and using an affinity submodule for pinning each level on the right core. Finally, this module computes the resulting g and L parameters.

The previously described steps of the benchmarking process are applied according to the pseudocode shown in Algorithm 1.

```

1: Multi-BSP-Tree ← Multi-BSP-Discover ()
2: for each level in Multi-BSP-Tree do
3:   [ $g$ ,  $L$ ] ← coreBenchmark(level)
4: end for
    
```

Algorithm 1 Multi-BSP Discover pseudocode.

Multi-BSP Tree acts as the interface between Multi-BSP Discover and the benchmarking module. As an example, Fig. 4 shows the structure corresponding to a specific hardware architecture having 32 cores, as generated by Multi-BSP Discover.

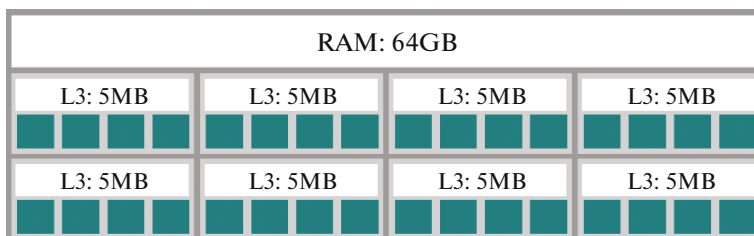


Fig. 4. Multi-BSP Tree structure generated by Multi-BSP-Disc-Bench.

3.2.2. The coreBenchmark module. The coreBenchmark module is conceived for computing the parameters g_i and L_i , according to the pseudocode shown in Algorithm 2.

```

01: setPinning(level.cores indexes)
02: begin(level.cores)
03: rate ← computingRate(level)
04: sync()
05: for h = 0 to HMAX do
06:   initCommunicationPattern(h)
07:   sync()
08:   t0 ← time()
09:   for i = 0 to NITERS do
10:     communication()
11:     sync()
12:   end for
13:   t ← time() − t0
14:   if master then
15:     times.append(t × rate/NITERS)
16:   end if
17: end for
18: level.g, level.L ← leastSquares(times)
19: return (level.g, level.L).

```

Algorithm 2: Corec Benchmark Function

coreBenchmark receives as parameters the information of the corresponding level based in the Multi-BSP Model, and the data used for thread affinity (i.e., the core indexes and the size of cache memory) which are stored in the Multi-BSP Tree structure. At the beginning (line 1 in Algorithm 2), the setPinning function from the affinity module is used to bind the threads spawned by the begin function (line 2) to the cores corresponding to the current level. The function spawns one thread per core in that level and calculates the computing rate of the Multi-BSP component using the computingRate function (line 3). Each level has a set of cores sharing one memory, then for benchmarking a level, only those cores are considered.

The computingRate function (line 3) measures the time required to perform $2 \times n \times \text{DAXPY}$ operations. The DAXPY routine performs the vector operation $y = \alpha \times x + y$, adding a multiple of a double precision vector to another double precision vector. DAXPY is a standard operation for estimating the efficiency of the computing platform when performing memory-intensive floating-point operations, from the Basic Linear Algebra Subprograms—Level 1 (BLAS1, as described at <http://www.netlib.org/blas>). After that, a synchronization for the current level is performed (line 4) in order to guarantee that all threads have the correct computing rate value.

We use the coreBenchmark function to measure a full h-communication. This is an abstraction that we define as the extension of a h-relation from the standard BSP model, but in this case the concept is applied to the shared memory case within a single node. An h-communication is implemented as a communication where every core writes/reads exactly h data words. We consider the worst case, measuring the slowest communication possible by cyclically reading single data words into other processors. In that way, the values of g_i and L_i computed using the benchmark are pessimistic values, and the real values will be always better. The variable h represents the largest number of words read or written in the shared memory of the level. HMAX is the maximum value for all the parameters (h) used in the communications patterns for each level. HMAX may need to be different for different levels of the hierarchy; we propose to find suitable values by empirical analysis.

The communication times using the h-communication pattern are initialized by the initCommPattern routine (line 6). This process is repeated NITERS times (lines 9–12), because each operation is too fast to be measured with proper precision. After that, the master thread in each level saves the flops used for each h-communication (line 15).

Finally, the parameters g and L are computed using a traditional least squares approximation method to fit the data to a linear model (line 18), according to the results and approximations found in the related works [1, 5].

This way, the method provides accurate approximations for g_i and L_i .

3.3. Empirical Evaluation of h-communications

The methodology applied to measure the h-communications and then estimate the parameters g and L is based on measuring the implementation of Multi-BSP operations. We refer to Multi-BSP operations as the functions/procedures needed to implement an algorithm designed with the Multi-BSP computational model. In our software design, the Multi-BSP operations module contains the implementation of these functions, including operations provided by the MulticoreBSP for C library [12]. This library establishes a methodology for programming according to the Multi-BSP computational model.

It is important to take into account the software design for the Multi BSP-Disc-Bench tool in Fig. 3, because when Multi-BSP algorithms are programmed using other libraries, it is possible to reconFig. the tool, changing the Multi-BSP operation module and re-characterizing the architecture by running the dis-

covering and benchmarking procedure with the new configuration. Further details on the methodology for the empirical evaluation of h-communications are reported in our conference article [1].

4. AN ENGINE PROPOSAL FOR Multi-BSP ALGORITHMS

This section introduces a proposal of an engine for designing and executing Multi-BSP algorithms. The proposed engine will perform data partitioning, threads management, and execution, encapsulating all the underlying logic, which will be hidden to the programmer. Behind the scenes, the proposed engine applies a recursive Divide-and-Conquer strategy.

4.1. Main Concepts

The proposed engine is conceived as a base layer, hiding the implementation details needed to work with Multi-BSP algorithms. The main goal of the proposed engine is to provide an easy and meaning way to design and implement Multi-BSP code, just paying attention to the problem-solving strategy instead of focusing on the specific details of the Multi-BSP model, such as thread management, data partitioning, and distributed execution.

The proposed engine uses a discovering process to determine the underlying architecture of a multicore computer and formulates a Divide-and-Conquer strategy to solve the problem. The Divide-and-Conquer strategy is applied recursively over the different hardware architecture levels, which is represented as a tree. The strategy focuses on solving three main issues: data partition, thread expansion, and thread reduction to compute the final results.

The proposed engine is conceived to provide the programmer several benefits, the three most relevant are: i) the programmer will have a better specification for its algorithms; ii) a single software layer will manage common issues needed for every Multi-BSP implementation; and iii) the approach will provide portable designs of Multi-BSP algorithms.

A useful feature of any Multi-BSP implementation is that the programmer needs to guarantee that the applied data partition will use the right cache memory in an effective way, i.e., trying to produce the major number of hits and minimize the misses. This is naturally produced when the algorithm applies a data partition strategy based in the available hardware and the threads or processes are executed in the nearest processing units to that memory. As a general rule, the size of a data partition should never be bigger than the size of the corresponding cache size.

Another challenge of Multi-BSP algorithms is the need of designing them closely tied to the hardware architecture. In this situation, the portability of a specifically designed algorithm cannot be assured, and maybe it will only execute properly in the same type of machines (i.e., depending on the size of each cache level, the distribution of processing units, and other features). To address this problem, the proposed engine implements a generic method to handle all specific hardware details and the programmer will only need to provide general functions that can be used in different architectures without suffering from major issues, thus assuring portability. Next section describes the features of the proposed Multi-BSP engine.

4.2. Multi-BSP Engine Design

As explained, the engine is a generic recursive procedure that traverses the tree that representing a specific hardware architecture. The path followed by the engine defines the strategy to perform the data partition, the thread expansion, and thread reduction. In the proposed implementation, a preorder traversal algorithm is applied, but this is just a decision to take advantage of a simple strategy. The traversal algorithm can be customized to follow different paths in the tree.

The engine uses the same hardware autodiscovering process applied in Multi-BSP-Disc-Bench to generate a Multi-BSPTree. Once having that information, the engine processes the tree in a recursive way, where each recursion level is mapped with a level of the Multi-BSP computation model.

A pseudocode of the Multi-BSP engine definition is presented in Algorithm 3. The engine has two input parameters: i) the current tree node representing a Multi-BSP component and ii) the data to work with. All the information needed for thread affinity is already available in the current Multi-BSP component. This data structure has pointers to its subcomponents (t.sons) and each son has the right indexes work with.

Each thread has a unique identification $p \in [0 \dots n - 1]$, where n is the number of threads obtained using the function `bsp_nprocs`. The thread identification p is determined by calling `bsp_pid` (lines 3 and 4 in Algorithm 3).

After that, the partition function is applied on the current data, using as parameters the values of p and n (line 5). Before processing the tree, a barrier synchronization (using the `bsp_synch()` function) is needed to guarantee that the private piece of data is available in each one of the destination threads (line 6). Then, the recursion starts processing nodes in the tree.

```

01: bsp_set_pin(t.sons){ Affinity using sons components }
02: bsp_begin() { Spawn threads }
03: n ← bsp_nprocs() { Amount of threads in the current level }
04: p ← bsp_pid() { thread id / component number at level n }
05: dpi ← partition(|d|, p, n) { Data for thread p and level i }
06: bsp_synch() { Sync to guarantee all threads have their partitions }
07: if n > 1 then { is not a leaf component / it has sons }
08: foreach tson, i in t.sons do
09:   vr[i] ← mbspEngine(dpi, tson) { recursion down over sons}
10: end for
11: bsp_synch() { Waiting to receive the result of every son }
12: if master then
13:   r ← reduce(vr) { The master of the level executes reduce}
14: end if
15: else
16:   r ← work(dij) { The leaf thread executes work function }
17: end if
18: bsp_synch() { Wait for master to have r or sons running work}
19: return r

```

Algorithm 3 Multi-BSP Engine

Data processing in those components that are not leaf nodes is described by the code within lines 7–16. A recursive call of the engine is performed (lines 8–10), using as parameters the current data partition and the tree node for each subcomponent. Each call returns a result, which is stored in the vector vr . A synchronization is needed to guarantee that all results are stored in vr (line 11). Then, the invoking thread (i.e., the master thread) reduces the values, applying the reduce function over vector vr (lines 12–14). Finally, the actions to execute when the recursion reaches the leaf nodes of Multi-BSPTree is shown in lines 15–17, where the function work is executed to compute the partial results.

Fig. 5 shows a typical execution of the engine to solve a separable problem, describing the application of the functions defined by the user: partition, work, and reduce. The execution is performed in a computer whose architecture has two levels, each one having two subcomponents. Each gray square represents a Multi-BSP component instance and each arrow represents communications between them.

4.3. The Cost Function for the Proposed Engine

The proposed engine has three sequential phases in each execution step T performed over the architectural tree: the recursion from top to bottom in the tree (CD), the work function in the bottom (CW) and the recursion return from bottom to top (CU). Each step T represents the work needed to process a piece of data D .

The cost of performing an execution of the proposed engine is the sum of the cost functions for each one of the three sequential steps: $CT = CD + CW + CU$. The cost function for each step and component is computed based on Equation 1, as is presented next.

Recursion from top to bottom. The data decomposition performed in D is made applying the partition function only one time (i.e., one superstep) per level. Then, the corresponding value of Nk in Equation 1 is 1, and because the partition function is sequential and thread safe (i.e., it does not involve any parallel computation), the standard big O notation can be used for it, resulting in the expression in Equation 2.

$$C_D = \sum_{k=0}^{L-1} O(partition_k) + h_k \times g_k + l_k \quad (2)$$

The component working in the partition phase has p subcomponents and consequently it needs to communicate to each one on its sub-partition. For this reason, the value of h_k is the maximum for each partition. Therefore, Equation 3 holds.

$$C_D = \sum_{k=0}^{L-1} O(partition_k) + \frac{D_k}{p_k} \times g_k + l_k \quad (3)$$

Work. The work function it is even simpler, because it only executes once in one specific level: the leafs of the tree, corresponding to the processing units of each component. Then, Nk and L are 1, and the cost is given by Equation 4.

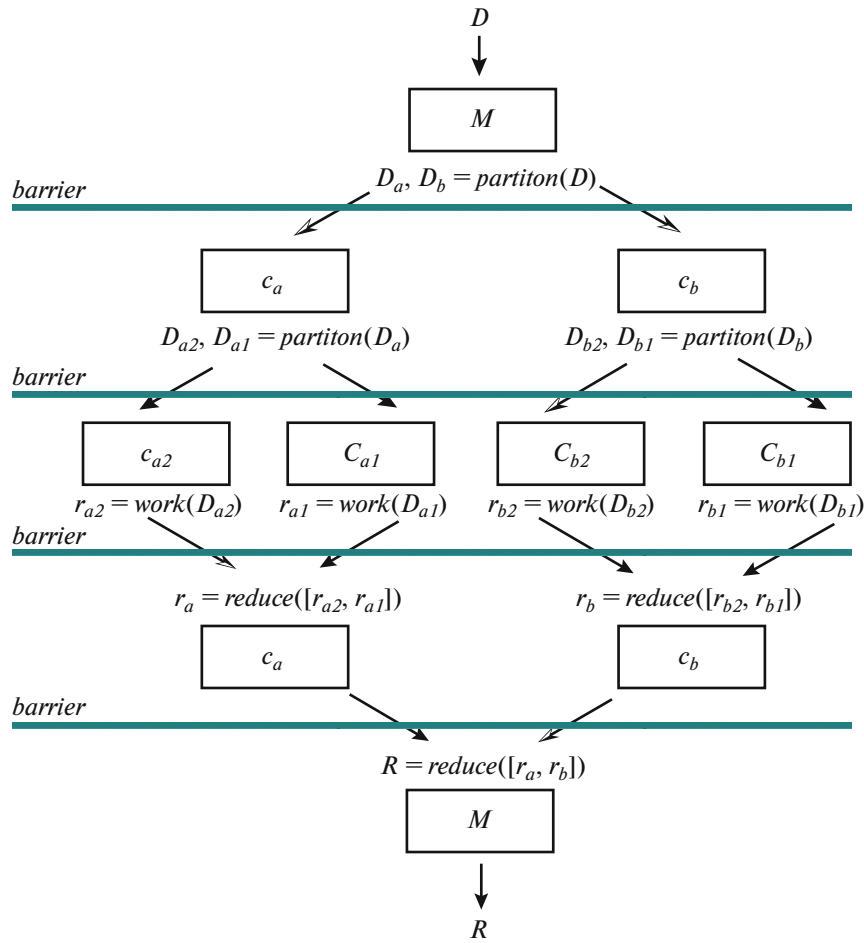


Fig. 5. Schematic view of an MBSP Engine execution.

$$C_W = O(work_{leaf}) + l_0 \quad (4)$$

Recursion from bottom to top. The cost of the bottom-up recursion is composed by the cost of the reduction function per level. One important fact is that every execution of the reduction function depends on the number of sons for that level. After called by the reduction function, every son will return one value, which implies one bottom-top communication per subcomponent. Given a component at level k with p_k subcomponents that is running a reduction, the amount of communications (h_k) will be p_k , one per subcomponent. Then the cost is given by Equation 5.

$$C_U = \sum_{k=0}^{L-1} O(reduction_k) + p_k \times g_k + l_k \quad (5)$$

The resulting equation for the execution time for a given algorithm designed for the Multi-BSP engine is given by Equation 6.

$$C_T = O(work_{leaf}) + l_0 + \sum_{k=0}^{L-1} (O(partition_k) + O(reduction_k)) + p_k \times g_k + \frac{D_k}{p_k} \times g_k + 2l_k \quad (6)$$

Next subsection presents an example of an algorithm designed using the proposed engine, applying specific specification for the partition, work, and reduce functions.

4.4. A sample Instance of the Proposed Engine

This subsection describes a simple example of the proposed engine applied to solve a simple decomposable problem. This sample provides a baseline to design and build more complex instances for solving problems that only need to perform the three basic functions included in the engine: partition, work and reduction.

The Partition function is executed splitting the original data, one slice per subcomponent. The processing is performed recursively over the MBSP-Tree and the data slice of a component is the source for the next partition call inside of its subcomponents. When the recursion reaches the stop phase, in component at level 0, processors run the Work function. Finally, the Work function will return the result and a master process will merge these results with the Reduce function. The result of the Reduce function is sent to the father component. The process of executing partition inductively, working in processors and finally executing the reduction recursively is represented in Fig. 5.

Said that, it is important to note that not every parallel algorithm will fit with the general scheme in the proposed engine. The recursive divide and conquer strategy is not usually the best fit for problems that cannot be partitioned in an arbitrary way.

For designing portable MultiBSP algorithms applying the engine, data should be able to be partitioned recursively over the hardware architecture discovered.

Furthermore, the programmer does not know a priori the architecture of the computers for which he is designing a specific program. A programmer may design a specific algorithm tightly coupled to a given computer, but in that case, it is highly probable that the algorithm will not be able to exploit efficiently the features of other hardware platforms (e.g., when a larger number of computing resources/cores are available). In some cases, the algorithm will not be able to run on a different architecture. Using the proposed engine allows the programmer to design and implement portable algorithms. Such algorithms will discover the architecture of the computer they are executing on, and will take advantage of the available resources and topology. According to the discovered information, the partition tasks will execute in the induction stage, then the work function will execute in the available resources, and the results will be reduced in the recursion stage. This way, for any algorithm that fits in the general scheme proposed by the engine (i.e., its data can be partitioned arbitrarily), the engine provides a useful method for taking advantage of the underlying hardware architecture in a transparent and effectively manner.

To better illustrate the benefits of the proposed engine, this section presents as an example a “dot product” algorithm. It obviously fit for a recursive divide/conquer strategy (i.e., following the data parallel approach) and therefore for the proposed engine. The algorithm is simple, but it is very instructive for the purpose of showing how to work with the proposed engine.

Let’s start specifying the three functions needed by the engine. The Partition function (Algorithm 4) sets up a specification for the engine based on the original data, a component number, and the number of com-

ponents in that level. For the sample instance presented, the function implements a partition usable for the dot product algorithm: a slice of the total data divided by the number of components. The specific number of each component number is used to calculate the slice to be used by each component

```
01 Partition(interval, componentNumber, n) {
02   sliceSize = interval.length / n
02   return interval.slice [
03     sliceSize * componentNumber,
04     sliceSize * (componentNumber+1)
05 ]
06 }
```

*Algorithm 4: Partition Function
for Dot Product Instance*

The Work function (Algorithm 5) receives a slice or interval of the original data. This function is executed, in this case, only for the leaf components. These are the most elemental components, i.e., processors with their nearest memory. As it was shown in the Fig. 5, every thread working at this level represents a direct mapping between the number of processors and the number of threads (i.e., one thread per processor). The returned value will be used by the Reduce algorithm as it is shown next.

```
01 Work (slice) {
02   for value in slice {
03     result += value*value
04   }
05   return result
06 }
```

Algorithm 5: Work Function for Dot Product Instance

Finally, the Reduce function (Algorithm 6) receives an array of values. The input values are obtained either as a result of the Work function or as a result of another execution of the Reduce function in son components.

```
01 Reduce( arrayValues ) {
02   for v in arrayValues {
03     result += arrayValues[i]
04   }
05 }
```

Algorithm 6: Reduce for Dot Product Instance

5. EXPERIMENTAL ANALYSIS

This section reports the values for g and L parameters obtained for different architectures using the proposed benchmark. These values will be used later in result validation of the algorithm designed using the proposed engine, contrasting the real time vs. the estimated time as reported by the model.

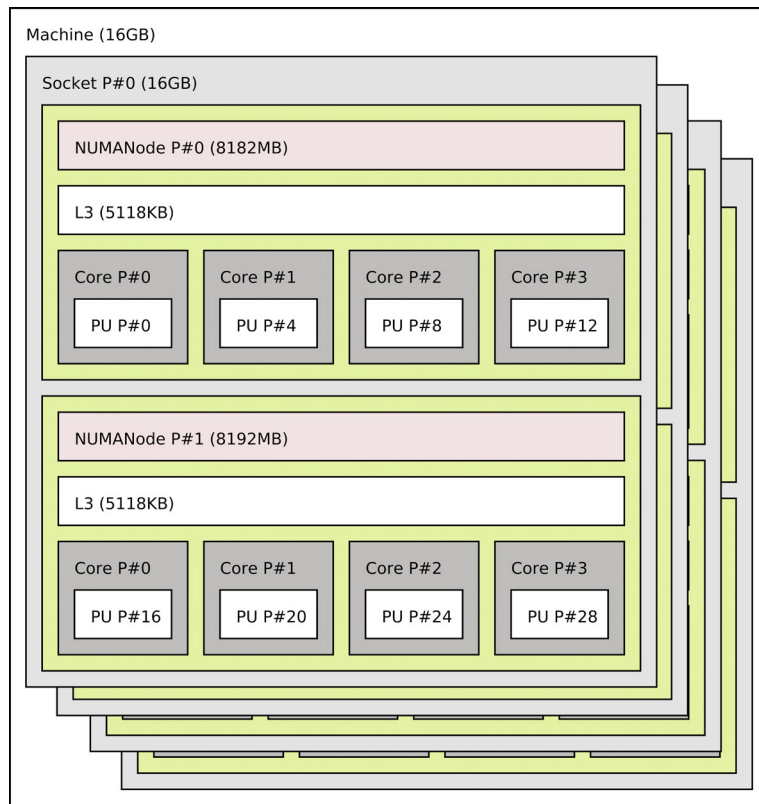


Fig. 6. hwloc output describing the topology of the dell32 multicore computer.

5.1. Architectures Used in the Experimental Analysis

For the reported experiments, the hierarchical levels of the considered architectures are especially relevant. The main goals of the experimental analysis are to verify the values reported for the Multi-BSP parameters computed correspond with the theoretical values.

Three real multicore infrastructures were selected for the experimental analysis, featuring a reasonably large number of cores and interesting cache levels:

λ Instance #1 is dell32, whose architecture is shown in Fig. 6 (the diagram obtained by applying hwloc). dell32 has four AMD Opteron 6128 Magny-Cours processors with a total of 32 cores, 64 GB RAM, and two hierarchy levels.

λ Instance #2 is jolly, whose architecture is shown in Fig. 7 (the diagram obtained by applying hwloc). jolly has four AMD Opteron 6272 Interlagos processors with a total of 64 cores, 128 GB RAM, and three hierarchy levels.

λ Instance #3 is XeonPhi node in the Cluster FING high performance computing platform from Universidad de la República, Uruguay [6]. Xeon Phi has 60 cores, 8G of RAM, L2 cache of 512Kb and L1 Cache of 32Kb. Each core has four process units for hyperthreading, making a total of 240 physical threads and its architecture is presented in Fig. 8.

For each of the targeted architectures, the first stage is to specify the corresponding instances in the Multi-BSP model. The following description proceeds step-by-step in the process of building the specification, for a better understanding of the Multi-BSP formulation.

In the case of instance #1, the dell32 computer, the procedure to build the specification starts from bottom (cores) to upper levels and builds the components in tuples that share a memory space. The first tuple is composed of a single core at level₀. This core does not share any memory with any other component, so its shared memory is 0 and both parameters g and L are zero by definition: $\text{tuple}_0 = \langle p_0 = 1, g_0 = 0, L_0 = 0, m_0 = 0 \rangle$. Regarding the following level, the four basic components in level₀ share the L3 cache memory with a size of 5 MB, building a new Multi-BSP component level₁. This new component is formally described by the tuple, $\text{tuple}_1 = \langle p_1 = 4, g_1, L_1, m_1 = 5 \text{ MB} \rangle$. Finally, all eight components in level₁ share the RAM memory, with size of 64 GB, building the next and last level, level₂, in a Multi-BSP specification. This one is formally described by $\text{tuple}_2 = \langle p_2 = 8, g_2, L_2, m_2 = 64 \text{ GB} \rangle$.

The final step is to join all tuples using a sequence for a complete Multi-BSP machine specification and discard the level₀ for our benchmark proposal, because the values of g_0 and L_0 are known by defini-

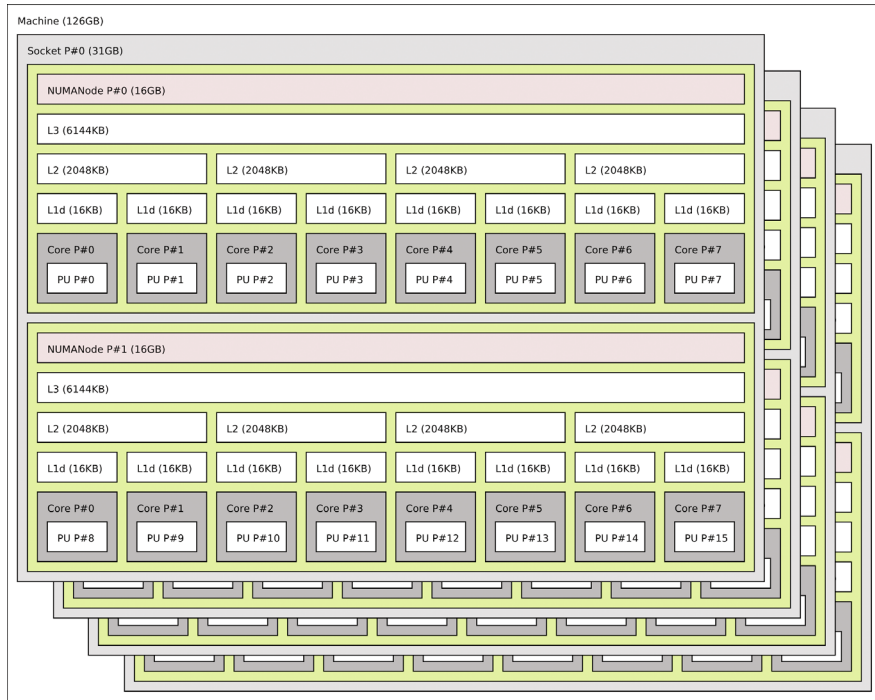


Fig. 7. hwloc output describing the topology of the jolly.

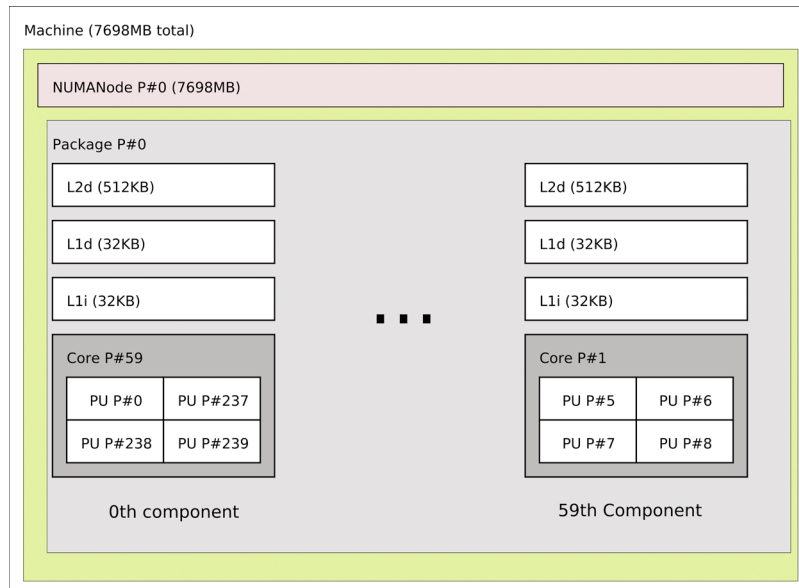


Fig. 8. hwloc output describing the topology of the XeonPhi co-processor.

tion. The architecture of instance #1 is then described by Eq. 16.

$$M_1 = [\langle p_1 = 4, g_1, L_1, m_1 = 5 MB \rangle, \langle p_2 = 8, g_2, L_2, m_2 = 64 GB \rangle] \quad (16)$$

Using the same procedure, the Multi-BSP specification for instance #2, jolly is built. Again, level₀ is

described by tuple₀ = ⟨p₀ = 1, g₀ = 0, L₀ = 0, m₀ = 0⟩. Level₀ has the same specification in all machines, except for cores that use the hyper threading technology (in that case, an extra level is need to specify physical threads).

After that, there are two components sharing the L2 cache, with a size of 2 MB. The level₁ is described

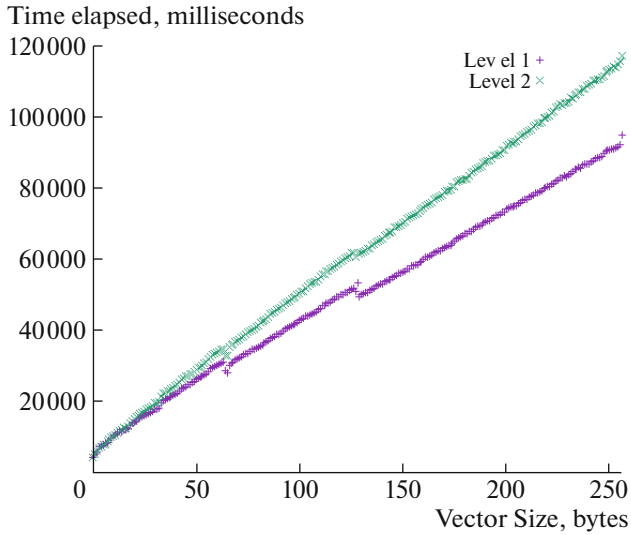


Fig. 9. Time for h-communications in dell32.

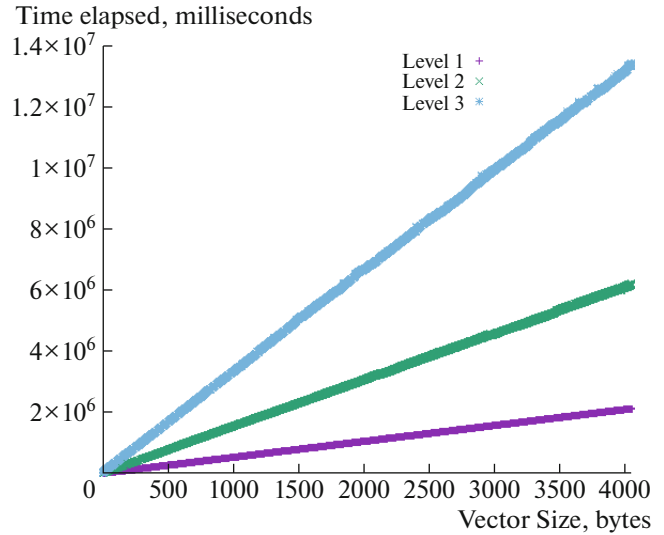


Fig. 10. Time for h-communications in jolly.

by tuple₁ = ⟨p₁ = 2, g₁, L₁, m₁ = 2 MB⟩. The components at level₁ are grouped by sharing four L3 cache memories, with a size of 6 MB, building the level₂, as defined by tuple₂ = ⟨p₂ = 4, g₂, L₂, m₂ = 6 MB⟩. In the last level, eight components from level₂ are grouped. They share the RAM memory, with a size of 128 GB, as specified by tuple₃ = ⟨p₃ = 8, g₃, L₃, m₃ = 128 GB⟩.

Finally, using the same procedure previously applied to the dell32 architecture (i.e. joining all tuples and discarding level₀), the Multi-BSP specification is expressed in Eq. 17.

$$M_2 = [\langle p_1 = 2, g_1, L_1, m_1 = 2 \text{ MB} \rangle, \langle p_2 = 4, g_2, L_2, m_2 = 6 \text{ MB} \rangle, \langle p_3 = 8, g_3, L_3, m_3 = 128 \text{ GB} \rangle] \quad (17)$$

The third architecture is the XeonPhi co-processor. Applying the same procedure, at the bottom level the processing units are identified, and they are included in the level₀. Like in the other architectures studied, the processing units do not share memory at all and the definition for this level is: tuple₀ = ⟨p₀ = 1, g₀ = 0, L₀ = 0, m₀ = 0⟩. After that, there are four components at level₀, sharing the L2 cache, which has a size of 512Kb. Therefore, level₁ is described by tuple₁ = ⟨p₁ = 4, g₁, L₁, m₁ = 512 Kb⟩. Finally, as described in Fig. 8, the last level is tuple₂ = ⟨p₂ = 60, g₂, L₂, m₂ = 7698 Mb⟩.

The aforementioned instances of the Multi-BSP model are applied in this article to predict the running time of a Multi-BSP algorithm executed in each computer. The g_i and L_i parameters in each tuple must be previously calculated using the benchmarking procedure explained in the previous section. Next section reports the values of g and L obtained for both architectures at each level.

5.2. Performance Results for the Studied Architectures

The performance experiments are oriented to determine the execution time of a simple algorithm designed and implemented using the proposed engine. Thus, this subsection reports the time to perform h-communications in each level, increasing the number h, as proposed in the coreBenchmark function.

The obtained results are reported in Figs. 9–11. Figure 9 reports the h-communications in each level for dell32 (level₁ and level₂). Figure 10 reports the same results for jolly (level₁, level₂, and level₃). Finally, Fig. 11 reports the h-communications in each level for Xeon Phi.

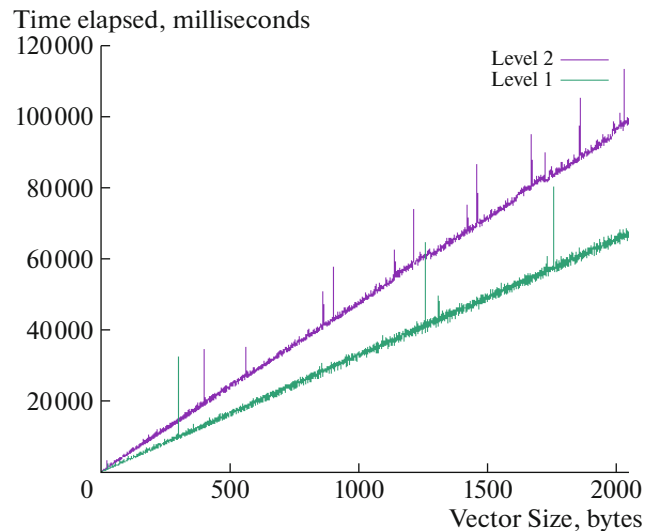


Fig. 11. Time for h-communications in Xeon Phi.

Table 1. Results of g and L parameters per level for dell32, jolly and Xeon Phi

Level	dell 32		jolly			Xeon phi	
	2	1	3	2	1	2	1
g	977.5	334.9	1315.9	549.9	105.3	2470.1	1947.9
L	15550.2	7792.9	16184.4	7157.9	498.2	1380955.3	1322578.2

In dell32, all level₁ communications are within the shared memory (L3 cache), so they are twice faster than in level₂, which use the RAM memory. For jolly, the communications in level₁ are within the L2 cache, thus they are three times faster than in level₂, where communications are performed through the L3 cache. In turn, they are 1.5× faster than those in level₃ of the hierarchy, which are performed by accessing the RAM memory.

Finally, the least squares method is applied to estimate the values of g_i and L_i over the h -communications for each level. The final values for dell32, jolly and Xeon Phi are reported in Table 1.

5.2. Dot Product Engine Instance Analysis

For validating the results computed in the previous subsection, an experiment was conducted using the dot product implemented over the MultiBSP engine. The validation process involves the following steps (applied for different vector sizes):

1. Estimate the number of communications and synchronizations at each level, by using hardware counters.
2. Compute the values of g_i and L_i parameters using the proposed benchmark.
3. Compute the runtime of the algorithm using the theoretical cost model for Multi-BSP, as presented in [10].
4. Execute the dot product algorithm.

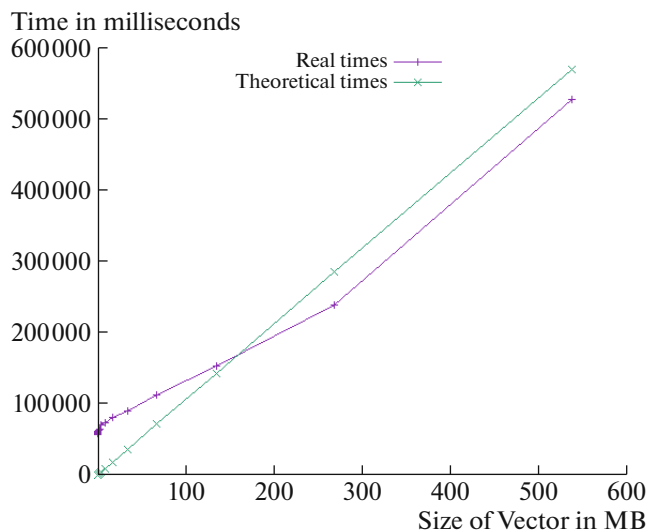
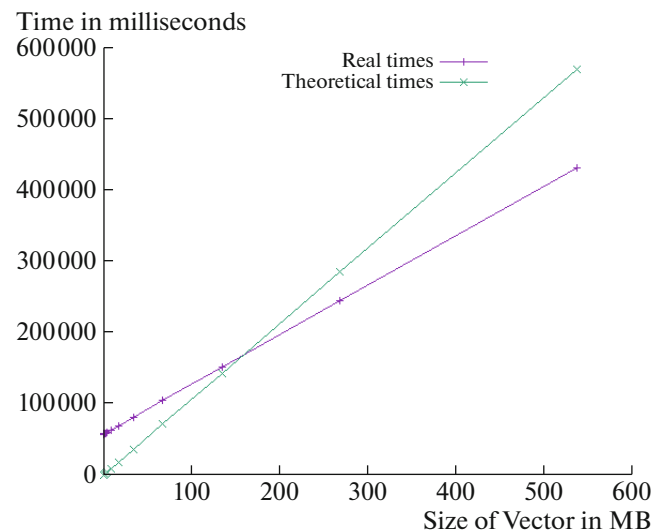
5. Compare the execution time of the dot product with the theoretical prediction of its time.

Figures 12–14 reports the comparison of the theoretical costs against the real costs for communications and synchronizations when executing the dot product algorithm implemented in the proposed engine. Figures show the execution time of dot product algorithm using the proposed MBSP engine vs. the estimated time using the theoretical model, considering an incremental size of the input data.

Results in Figs. 12–14 show the accuracy of the estimated time when compared to the real time. Relative errors are between 0 and 7%. In dell32, the mean error is 6% and the maximum error is 9%. In jolly, the mean error is 7% and the maximum error for the predicted time is 17%. The best results were obtained for Xeon Phi, for which the mean error is just 2% and the maximum error is 5%. These results suggest that the estimation provided for the dot product algorithm is accurate enough with respect to the real time for each studied architecture.

6. CONCLUSIONS AND FUTURE WORK

This article presented a proposal for a simplified approach to design and implement algorithms using the MultiBSP model.

**Fig. 12.** Theoretical vs real cost for the dell32 computer.**Fig. 13.** Theoretical vs real cost for the jolly computer.

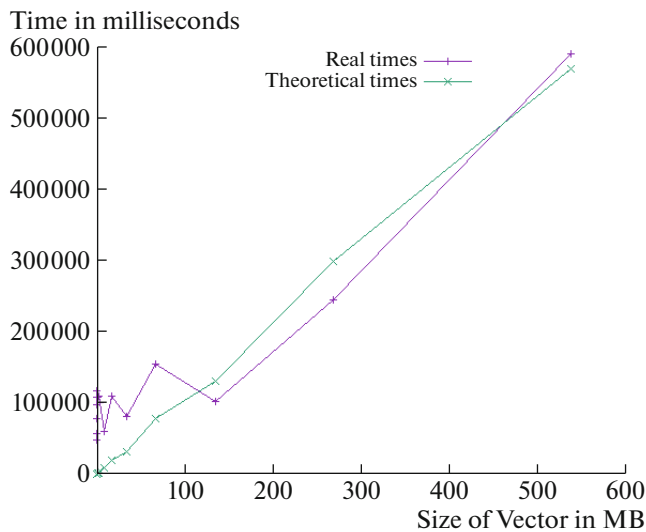


Fig. 14. Theoretical vs real cost for the Xeon Phi co-processor.

The proposed approach includes a methodology for automatically discovering the hardware features of a given computer and an engine to design and implement parallel algorithms following a general specification procedure. By following this approach, the programming does not need to focus on specific details of the MultiBSP implementation and benchmarking, which are encapsulated by the proposed engine. The programmer is encouraged to design MultiBSP algorithms using the general specification, based on a recursive divide and conquer strategy deployed over the architecture components (i.e., cores, cache, and RAM memory).

An implementation of a MultiBSP benchmark to characterize the underlying architecture is also presented. The benchmark is applied by the proposed engine, which also uses a discovering process to execute MultiBSP algorithms, hiding all the details about data-binding and pinning threads to the programmer.

The validation of the proposed implementations was performed over three modern high-performance architectures and a sample of the proposed engine was built and studied for solving a decomposable problem, the dot product algorithm.

The studied sample algorithm was used to analyze the theoretical execution time estimated using the cost model of MultiBSP against the real time of the dot product implementation in the proposed engine. Accurate results were obtained, accounting for mean relative errors between 2 and 7%. The best results were obtained for Xeon Phi, for which the mean error was just 2% and the maximum error is 5%.

The proposed methodology provides a foundation for developing a practical approach for a framework that includes a set of tools for designing, implemented,

and evaluating MultiBSP algorithms and accurately predict their execution times.

The main lines for future work are related to extend the analysis of the proposed methodology, e.g., by studying the capabilities of the engine with new, more complex algorithms. The engine can be extended to solve non-decomposable problems, taking advantage of its modular design and including specific problem knowledge defined by the user (i.e., the engine automatically handles the thread affinity, the parallel execution, and data locality for the partition function defined by the user). The hardware discovering process can be extended with extra levels including a network discovering process using specific software libraries and characterizing affinity and data-locality according to network speed and bandwidth.

REFERENCES

1. Alaniz, M., Nesmachnow, S., Goglin, B., Iturriaga, S., Gil Costa, V., and Printista, M., MBSPDiscover: an automatic benchmark for MultiBSP performance analysis, *Commun. Comput. Inf. Sci.*, 2014, vol. **485**, pp. 158–172.
2. Bisseling, R., *Parallel Scientific Computation: a Structured Approach Using BSP and MPI*, Oxford: Oxford Univ. Press, 2004.
3. Bonorden, O., Juurlink, B., von Otte, I., and Rieping, I., The Paderborn University BSP (PUB) Library, *Parallel Comput.*, 2003, vol. **29**, no. 2, pp. 187–207.
4. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R., Hwloc: a generic framework for managing hardware affinities in HPC applications, *Proc. 18th Euro-micro Conf. on Parallel, Distributed and Network-Based Processing*, Pisa, 2010, pp. 180–186.
5. Hill, J., McColl, B., Stefanescu, D., Goudreau, M., Lang, K., Rao, S., Suel, T., Tsantilas, T., and Bisseling, R., BSPlib: The BSP programming library, *Parallel Comput.*, 1998, vol. **24**, no. 14, pp. 1947–1980.
6. Nesmachnow, S., Computación científica de alto desempeño en la Facultad de Ingeniería, Universidad de la República, *Rev. Asoc. Ingenieros Uruguay*, 2010, vol. **61**, no. 1, pp. 12–15.
7. Savadi, A. and Deldari, H., Measurement latency parameters of the MultiBSP model: a multicore benchmarking approach, *J. Supercomput.*, 2014, vol. **67**, no. 2, pp. 565–584.
8. *Scalable Multi-Core Architectures*, Soudris D. and Jantsch A., Eds., New York: Springer-Verlag, 2012.
9. Valiant, L., A bridging model for parallel computation, *Commun. ACM*, 1990, vol. **33**, no. 8, pp. 103–111.
10. Valiant, L., A bridging model for multi-core computing, *J. Comput. Syst. Sci.*, 2011, vol. **77**, no. 1, pp. 154–166.
11. Yzelman, A., Fast sparse matrix-vector multiplication by partitioning and reordering, *PhD Thesis*, Utrecht: Utrecht Univ., 2011.
12. Yzelman, A., Bisseling, R., Roose, D., and Meerbergen, K., MulticoreBSP for C: a high-performance library for shared-memory parallel programming, *Int. J. Parallel Program.*, 2014, vol. **42**, no. 4, pp. 619–642.