# MBSPDiscover: An Automatic Benchmark for MultiBSP Performance Analysis

Marcelo Alaniz[1], Sergio Nesmachnow[2], Brice Goglin[3], Santiago Iturriaga[2], Verónica Gil Gosta[1], and Marcela Printista[1]

[1] Universidad Nacional de San Luis, Argentina
[2] Universidad de la República, Uruguay
[3] Inria Bordeaux–Sud-Ouest, University of Bordeaux, France

**Abstract.** Multi-Bulk Synchronous Parallel (MultiBSP) is a recently proposed parallel programming model for multicore machines that extends the classic BSP model. MultiBSP is very useful to design algorithms and estimate their running time, which are hard to do in High Performance Computing applications. For a correct estimation of the running time, the main parameters of the MultiBSP model for different multicore architectures need to be determined. This article presents a benchmark proposal for measuring the parameters that characterize the communication and synchronization cost for the model. Our approach discovers automatically the hierarchical structure of the multicore architecture by using a specific tool (hwloc) that allows obtaining runtime information about the machine. We describe the design, implementation and the results of benchmarking two multicore machines. Furthermore, we report the validation of the proposed method by using a real MultiBSP implementation of the vector inner product algorithm and comparing the predicted execution time against the real execution time.

## 1 Introduction

Performance prediction is an important tool for performance analysis of parallel applications [5]. This technique involves modeling program performance as a function of the hardware and software characteristics of a system. By changing these characteristics in the model, the execution time of standard programs can be accurately predicted for a variety of platforms and configurations.

The Bulk Synchronous Parallel (BSP) model [7], is one of the most popular among several analytical models proposed. The model assumes a BSP abstract machine with identical processors. Each processor has access to its own local memory and they communicate with each other through a all-to-all network, providing uniform point-to-point access time and bandwidth capacity.

The BSP model was introduced for distributed computers, but assuming only one core per computing node. 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 MultiBSP by Valiant [8]. MultiBSP 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, MultiBSP incorporates memory size as an additional parameter in the model, which was not included in the original BSP.

In this line of work, the research reported in this paper is focused on solving the problem of characterizing multicore computing architectures, which are described by a series of parameters such as size, latency, and memory levels. When a parallel algorithm based on the MultiBSP computational model is designed, the programmer needs to know the value of the parameters that describe the architecture, since the performance of the resulting algorithm depends on these parameters. Moreover, the MultiBSP programmer needs to conceive his application with multiple levels of abstraction that require the appropriate use of threads, cache memories and the cores that share these caches.

The proposed benchmark has the following features: a) it computes the MultiBSP parameters using a bottom-up technique for discovering the architecture and building the hierarchy levels using the MultiBSP approach and b) it is implemented using the same library that implements the abstraction levels of the application, so it measures the critical operations taking into account not only the theoretical aspects, but also the specific implementation.

In order to develop the proposed benchmark, we address the following topics: *i*) based on the detection of the hierarchy of levels in a multicore machine, we show how to translate the hierarchy into the components of an abstract MultiBSP machine. *ii*) we explain formally all parameters, specially focusing on communication and synchronization costs. *iii*) we introduce the concept of $h$-communication, which is an adaptation of the $h$-relation of BSP for the specific case of shared-memory relations within a single node.

Our benchmark is applied to characterize two High Performance Computing (HPC) multicore machines. We also report the validation of the proposed method by using a real MultiBSP implementation of the vector inner product algorithm and comparing the predicted execution time against the real execution time.

The research reported in this article is developed within the project "Scheduling evaluation in heterogeneous computing systems with hwloc" (SEHLOC[1]). The main goal consists in the development of 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.

The paper is organized as follows. Section 2 introduces the BSP and MultiBSP models, and relevant related work about BSP benchmarking. Section 3 describes the design and implementation of the MBSPDiscover benchmark. Section 4 reports the application of the proposed benchmark for two case studies and the validation using a real MultiBSP application. Finally, Section 5 presents the conclusions and formulates the main lines for future work.

[1] `http://runtime.bordeaux.inria.fr/sehloc/`

## 2 BSP and MultiBSP Models

To set the scope of this paper, this section describes the BSP and MultiBSP models. We start with a brief description of the flat BSP model and how it evolved into the concept of multicore, which emphasizes hierarchies of components.

### 2.1 The Original BSP Model

The BSP model considers an abstract parallel computer, which is fully modeled by a set of parameters: $p$—number of processors, $s$—processor speed, $g$—communication cost, and $l$—synchronization cost. Using these parameters, the execution time of any BSP algorithm can be calculated.

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.

The practical model of programming is Single Program Multiple Data (SPMD), implemented as C/C++ program copies running on $p$ processors, wherein communication and synchronization among copies are performed using specific libraries such as `BSPlib` [4] or `PUB` [2]. 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 calculated as the accumulative sum of the cost of its supersteps, where the cost of each superstep is the sum of three quantities: *i*) $w$, the maximum number of calculations performed by each processor; *ii*) $h \times g$, where $h$ is the maximum of the 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 New MultiBSP Model

Modern supercomputers are made of highly parallel nodes with tens of cores. The efficiency of these nodes required improvements of the memory subsystem by adding multiple hierarchical levels of caches as well as a distributed memory interconnect causing Non-Uniform Memory Access (NUMA). In 2010, Valiant updated the BSP model to account for this situation, resulting in the MultiBSP model. It was defined with the same abstractions and bridge architecture as the original BSP, but adapted to multicore machines.

The MultiBSP Model 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 MultiBSP machine is specified by a list of tuples (levels) where each tuple has four parameters $(m_i, p_i, g_i, L_i)$ where:

- $p_i$ is the number of $i$-1th level components inside an $i$th component. For $i = 1$, these 1st level components consist of $p_1$ raw processors, which can be regarded as 0th level components. One computation step of such a processor on a word in level 1 memory is taken as one basic unit of time.
- $g_i$ is the communication cost parameter, it is 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* here is the amount of data on which a processor operation is performed. We assume that the level$_1$ memories can keep up with the processors, and hence that the data rate (corresponding to the notation $g_0$) has the value one.
- $L_i$ is the cost for the barrier synchronization for a level $i$ superstep. The 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 an $i$th level component that is not inside any $i-1$th level component.

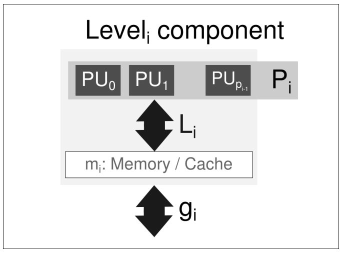


**Fig. 1.** Schematic view of the $i$th component level of MultiBSP model

Fig. 1 shows a component of level $i$. A level $i$ superstep is a construct running at a level $i$ component that allows each of its $p_i$ level $i-1$ components to execute independently (including supersteps of level $i-1$). Once all $p_i$ finish their computation, they can all exchange information with the $m_i$ memory of the level $i$ component with a communication cost determined by $g_{i-1}$. The cost charged will be $mg_{i-1}$, where $m$ is the maximum number of words communicated between the memory of the $i$th level component and any one of its level $i-1$ subcomponents. After a barrier between the $p_i$ components, the next superstep may begin.

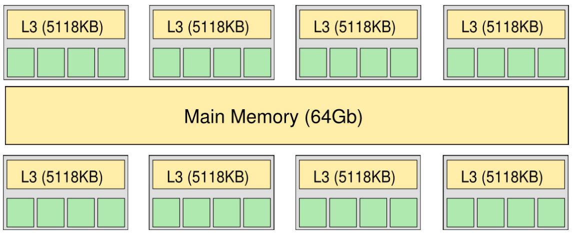


**Fig. 2.** MultiBSP model: (5118KB, 4, $g_1$, $L_1$),(64 GB, 8, $g_2$, $L_2$)

For instance, Fig. 2 shows a machine, whose architecture can be specified by three MultiBSP components (level$_0$, level$_1$ and level$_2$): (0, 1, 0, 0), (5118KB, 4, $g_1$, $L_1$) and (64 GB, 8, $g_2$, $L_2$). We can ignore the level$_0$ because it represents only one processing unit and thus does not involve internal synchronization or communication. Therefore we only have two components, which corresponds to the two level of hierarchy in the architecture.

A benchmarking algorithm for the MultiBSP model will need an automatic process for discovering the specific hardware architectures. Accordingly, in our work we use the *portable HardWare LOCality* (`hwloc`) tool [3][2] that allows obtaining runtime information about the architecture of the machine, such as processors, caches, memory nodes, etc. in an abstract way.

The use of the hwloc software package has been proposed in the SEHLOC project in order to have a tool for automatically detecting the architecture features of multicore systems, defining the interconnection topologies and the hierarchies for neighboring cores. We use the version 1.7.2 of hwloc, which provides a portable abstraction (across OS, versions, architectures, etc.) of the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and simultaneous multithreading. It also gathers various system attributes such as cache and memory information as well as the locality of I/O devices such as network interfaces, InfiniBand HCAs or GPUs. It primarily aims at helping applications with gathering information about modern computing hardware so as to exploit it accordingly and efficiently.

## 2.3 Related Work

The program `bspbench` from BSPEdupack[4] has been the main benchmarking program on BSP model. The proposed 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 this cases the resulting $g$ obtained by benchmarking program `bspbench` is called pessimistic. The Oxford BSP toolset

[2] Available under the BSD license at `http://www.open-mpi.org/projects/hwloc`

[4] has another benchmarking program, `bspprobe`, which measures optimistic $g$ values using larger packets insted of single words. BSP benchmarking also can be done by using mpibench from MPIedupack[4].

The benchmarking of the MultiBSP computational model has been recently addressed in the article by Savadi and Hossein [6], using a similar approach as the one we apply here. 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 characterics inside the output of will chosen 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 advantage of our proposal is to evaluate real MultiBSP operations implemented for the library `MulticoreBSP for C` [9]. In addition, our results are validated using a real MultiBSP program, comparing the real execution time of the inner product algorithm against the predicted running time using the theoretical MultiBSP cost function.

## 3 The MBSPDiscover Benchmark for MultiBSP

This section presents the design and implementation of the MBSPDiscover benchmark to estimate the $g$ and $L$ parameters that characterize a MultiBSP machine.

### 3.1 Motivation

Multicore architectures are widely used for HPC applications, and both the number of cores and the cache levels 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.

As mentioned previously, this work follows the MultiBSP model which specifies the parameters needed to characterize a multicore machine. In this model, the performance of a parallel algorithm depends on parameters such as communication and synchronization costs, number of cores, and the size of caches.

Because it is hard to build analytical equations involving those variables, performing computer benchmarking via a computational model is therefore a reasonable method to evaluate performance and characterize the architecture.

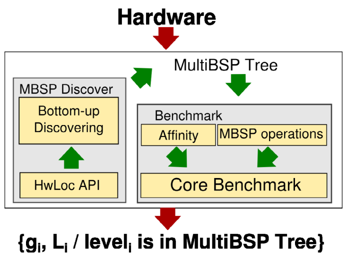
It is important to emphasize that the quality of a benchmarking tool should not depend on particular architecture. This extra requirement is solved by discovering the relations of the different cores within each level of cache.

## 3.2 MBSPDiscover Design

The existing benchmark `BSPbench` for the standard BSP model [1] was used as a reference baseline to design the MBSPDiscover tool. The obvious 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 MultiBSP case, the processing is made inside of multicore nodes instead of outside nodes through the network.

**Software Architecture and Modules.** Fig. 3 shows the software architecture for the kernel of the MBSPDiscover proposal. The functionality for each of the processes displayed in the figure is explained below:

– *Discovering module*: the hardware architecture is collected by using `hwloc` and it is loaded in a tree of resources. This structure is inside the `hwloc` API box.
– *Interface*: Once the tree structure is generated, a set of functions walk across the tree using a bottom-up process for building a new tree named `MBSPTree` that contains all the information needed to support the MultiBSP model.
– *Benchmarking module*: It retrieves core indexes and memory size from the `MBSPTree` for each level. Then it measures communication and synchronization cost through a MultiBSP submodule, as well as an affinity submodule for pinning levels on the right cores. Finally it computes the resulting $g$ and $L$ parameters.



```
MBSPTree = multibsp_discover()

foreach (level in MBSPTree) {
 g,L = coreBenchmark (level)
}
```

**Algorithm 1.1.** MBSPDiscover pseudocode.

**Fig. 3.** Schematic view (left) and pseudocode (right) of the MBSPDiscover process

`MBSPTree` acts as the interface between both modules. Fig. 4 shows the structure corresponding to the hardware architecture presented in Fig. 5.

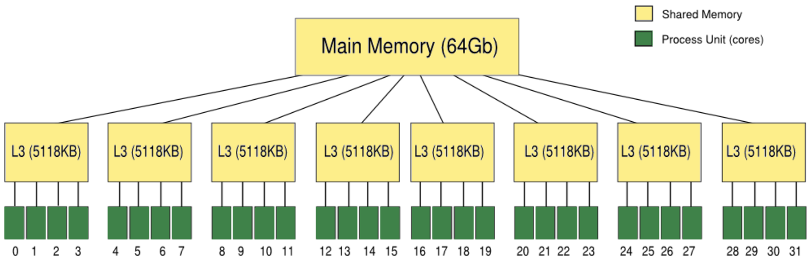


**Fig. 4.** MBSPTree structure generated by MBSPDiscover

**The Corebenchmark Module.** We explain in detail the implementation of the `coreBenchmark` module for computing the parameters $g_i$ and $L_i$.

The `coreBenchmark` function is shown in Algorithm 1.2. It receives as parameters the information of the corresponding level based in the MultiBSP Model, and data for affinity like the core indexes and the size of cache memory, which are stored in the `MBSPTree` structure. At the beginning (line 2), `coreBenchmark` uses the `setPinning` function from the `affinity` module. `setPinning` binds the threads spawned by the `begin` function (line 3) 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 MultiBSP component using `computingRate` function (line 4). Each level has a set of cores sharing one memory, then for benchmarking a level, only those cores are considered.

The `computingRate` function measures the time required to perform 2×n×`DAXPY` operations. The `DAXPY` routine performs the vector operation $\boldsymbol{y} = \alpha * \boldsymbol{x} + \boldsymbol{y}$, adding a multiple of a double precision vector to another double precision vector. `DAXPY` is a standard BLAS1 operation [3] for estimating the platform efficiency when performing memory-intensive floating point operations.

```
1   function coreBenchmark(level) {
2     setPinning(level.cores_indexes)
3
4     begin(level.cores)
5     rate = computingRate(level)
6     sync()
7
8     for (h=0; h<HMAX; h++) {
9
10      initCommunicationPattern(h)
11      sync()
12
13      t0 = time()
14
15      for (i=0; i<NITERS; i++) {
16        communication()
17        sync()
18      }
19
20      t = time() - t0
21
```

[3] BLAS operations are described at `http://www.netlib.org/blas/`

```
22       if (master) {
23         times.append(t*rate/NITERS);
24       }
25     }
26     level.g, level.L = leastSquares(times)
27     return (level.g, level.L)
28   }
```

**Algorithm 1.2.** `coreBenchmark` function.

Then a synchronization for the current level is performed (line 5) in order to assure that all threads have the computing rate value.

The `coreBenchmark` function measures a full $h$-communication, which we define as the extension of a $h$-relation for the shared-memory case within a single node. It 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 $h$ parameters used in the communications patterns for each level. It may need to be different for different levels of the hierarchy, we plan to find suitable values by trial and error.

The communication times using the $h$-communication pattern are initialized by the `initCommunicationPattern` routine (line 7). This process is repeated `NITERS` times (lines 10–13), 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 16).

Finally, the parameters $g$ and $L$ are computed using a traditional least squares approximation method (line 19), to fit the data to a linear model, according to the related works [1,6], providing an accurate approximation for $g_i$ and $L_i$.

### 3.3 Methodology for the 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 *MultiBSP operations*. We refer to MultiBSP operations as the functions/procedures need to implement an algorithm designed with the MultiBSP computational model. In our software design, the `MBSP operations` module contains the implementation of these functions, including operations provided by the `MulticoreBSP for C` library [9]. This library establishes a methodology for programming according to the MultiBSP computational model.

The software design shown in Fig. 3 is important here because when MultiBSP algorithms are programmed using other libraries, it is possible to reconfigure the tool, changing the `MBSP operation` module and re-characterizing the architecture by running the benchmark with this new configuration.

## 4 Experimental Analysis

This section reports the experimental analysis of the proposed MultiBSP benchmark. First, we introduce the problem instances by describing the main features of the architectures used to test the benchmark. After that, the numerical results and the values for the $g$ and $L$ parameters are reported. Finally, the validation of our results using a real MultiBSP program is presented.

### 4.1 MultiBSP Architectures Used in the Experimental Analysis

For our experiments, the hierarchical levels of the considered architectures are specially relevant. The main goals of the experimental analysis are to verify the proper functionality of the proposed benchmark and also to compute the corresponding values for the parameters of the MultiBSP model.

We selected two real infrastructures for the experimental analysis, which feature a reasonably large number of cores and interesting cache levels:

– Instance #1 is *dell32*, whose architecture is shown in Fig. 5. *dell32* has four AMD Opteron 6128 *Magny-Cours* processors with a total of 32 cores, 64 GB RAM, and two hierarchy levels.
– Instance #2 is *jolly*, whose architecture is shown in Fig. 6. *jolly* has four AMD Opteron 6272 *Interlagos* processors with a total of 64 cores, 128 GB RAM, and three hierarchy levels.

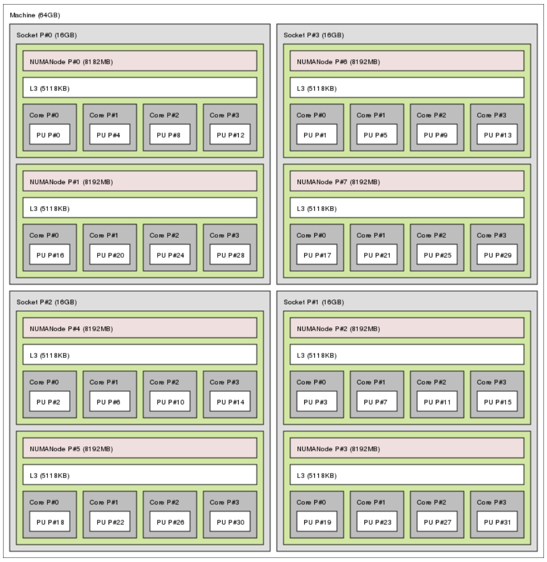


**Fig. 5.** `hwloc` output describing the topology of the *dell32* multicore machine

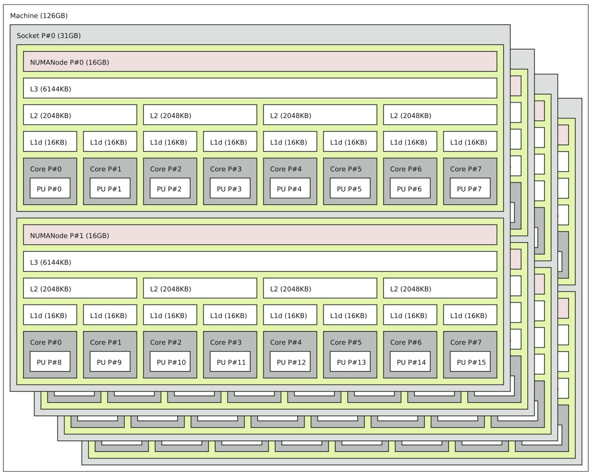


**Fig. 6.** `hwloc` output describing the topology of the *jolly* multicore machine

For each of those architectures, we need to specify the instances in MultiBSP. We proceed step by step for a better understanding of the MultiBSP formulation.

For *dell32* we start from bottom (cores) to upper levels and build the components in tuples that share a memory space. The first tuple is made of a single core at $level_0$. It does not shared any memory with any other component, so its shared memory is 0 and both parameters $g$ and $L$ are zero by definition: $tuple_0 = \langle p_0 = 1, m_0 = 0, g_0 = 0, L_0 = 0\rangle$. Then, the basic 4 components in level$_0$ share the L3 cache memory with a size of 5 MB, building a new MultiBSP component level$_1$. This new component is formally described by the tuple: $tuple_1 = \langle p_1 = 4, m_1 = 5\,\text{MB}, g_1, L_1\rangle$. Finally, all eight components in level$_1$ share the RAM memory, with size of 64 GB, building the next and last level, level$_2$, in a MultiBSP specification. This one is formally described by the tuple: $tuple_2 = \langle p_2 = 8, m_2 = 64\,\text{GB}, g_2, L_2\rangle$.

We join all tuples using a sequence for a complete MultiBSP machine specification and discard the level$_0$ for our benchmark proposal, because the values of $g_0$ and $L_0$ are known by definition. The architecture of instance #1 is then described by Eq. 1.

$$M_1 = [\langle p_1 = 4, m_1 = 5\,\text{MB}, g_1, L_1\rangle, \langle p_2 = 8, m_2 = 64\,\text{GB}, g_2, L_2\rangle] \qquad (1)$$

Using the same procedure, we build the MultiBSP specification for instance #2, *jolly*. Again, level$_0$ is described by $tuple_0 = \langle p_0 = 1, m_0 = 0, g_0 = 0, L_0 = 0\rangle$. It is the same in all machines, except for cores that use the hyperthreading technology (in that case, an extra level is need to specify physical threads). Then, there are two components sharing the L2 cache, with a size of 2 MB. The level$_1$ is described by $tuple_1 = \langle p_1 = 2, m_1 = 2\,\text{MB}, g_1, L_1\rangle$ The components at level$_1$ are grouped by sharing four L3 cache memories, with a size of 6 MB, building the level$_2$, as defined by $tuple_2 = \langle p_2 = 4, m_2 = 6\,\text{MB}, g_2, L_2\rangle$. In the last level

(#3), eight components from level$_2$ are grouped. They share the RAM memory, with a size of 128 GB, as specified by $tuple_3 = \langle p_3 = 8, m_3 = 128\,\text{GB}, g_3, L_3 \rangle$.

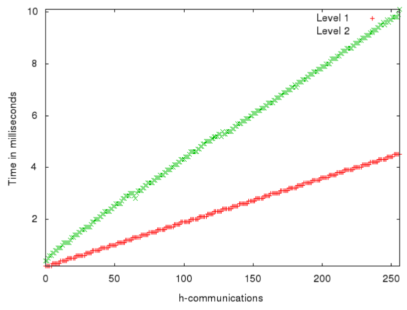
Finally, using the same procedure we previously applied to the dell32 architecture (i.e. joining all tuples and discarding level$_0$), we get the MultiBSP specification in Eq. 2.

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

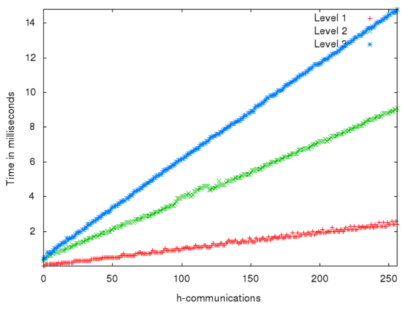
Using these instances of the MultiBSP model, we can predict the running time of a MultiBSP algorithm executed in each machine. The $g_i$ and $L_i$ parameters in each tuple must be previously calculated using the benchmarking procedure explained in the previous section. Next section reports the values of $g$ and $L$ obtained for both architectures at each level.

## 4.2 Results

We report the time to perform $h$-communications in each level, increasing the number $h$ as in the `coreBenchmark` function. Reporting the flops for each $h$-communications is important because we compute the $g_i$ and $L_i$ using least squares to estimate the parameters at each level.



(a) Instance #1: dell32



(b) Instance #2: jolly

**Fig. 7.** Time to perform from $h$-communications per level in a MultiBSP tree, with $h$ between 0 and 256

Figure 7 show the $h_i$ communications in each level for *dell32* (level$_1$ and level$_2$) and *jolly* (levels 1, 2, and 3). In level$_1$ of *dell32*, the communications are within the shared memory (L3 cache), so they are twice faster than in level$_2$, which use the RAM memory. For *jolly*, the communications in level$_1$ are within the L2 cache, thus they are three times faster than in level$_2$, where communications are performed through the L3 cache. In turn, they are 1.5× faster than those in level$_3$ of the hierarchy, which are performed by accessing the RAM memory.

**Table 1.** Computed values for $g$ and $L$ parameters for the studied architectures

| *dell 32* | | | *jolly* | | |
|---|---|---|---|---|---|
| level | $g$ (flops/word) | $L$ (flops) | level | $g$ (flops/word) | $L$ (flops) |
| 2 | 977.5 | 15550.2 | 3 | 1315.9 | 16184.4 |
| 1 | 334.9 | 7792.9 | 2 | 549.9 | 7157.9 |
| | | | 1 | 105.3 | 498.2 |

Finally, using the least squares method we estimate the values of $g_i$ and $L_i$ over the $h$-communications for each level. The final values for *dell32* and *jolly* are reported in Table 1.

## 4.3 Validation of Results

For validating the results computed in the previous subsection, we conducted an experiment using a real application, the *vector inner product* from BSPedupack (actually the computation of the norm of a vector), described in Algorithm 1.3 in the MultiBSP programming model. We plan to extend the validation by considering a set of benchmark applications as future work.

```
innerProduct(level, vector) {
  if (level.next == NULL ) {
    return sequentialInnerProduct(vector);
  } else {
    begin_parallel_multibsp ( level.sons.length )
       ownslice = split_vector(vector, multibsp_pid );
       level = level.sons[ multibsp_pid ];
       sync()
       results = innerProduct(level, ownslice)
       sync()
       if (multbsp_id == master) {
         return sequentialInnerProduct(results);
       }
    end_parallel_multibsp
  }
}
MBSPTree = MBSPDiscover()
innerProduct(MBSPTree, data_vector)
```

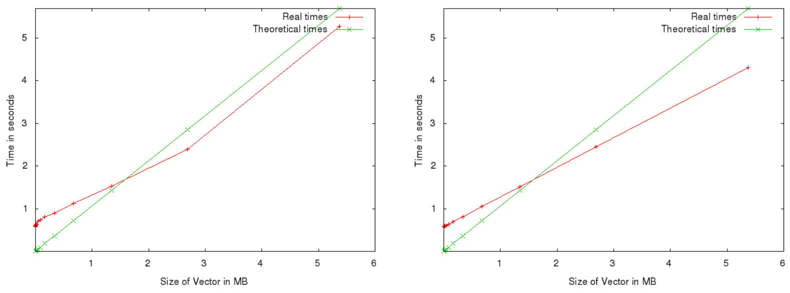
**Algorithm 1.3.** Vector Inner Product.

Algorithm 1.3 applies the MultiBSP programming model recursively, crossing the MCBSPTree obtained with MBSPDiscover in the proposed benchmark. Using the tree structure, the data vector is split in slices for each thread at level $i$. For $i > 0$, the data splitting is applied recursively. In level 0, a sequential inner product algorithm is used to compute a partial result. Then, after synchronizing all threads in each level, the result is the inner product for the whole data vector. The master thread applies a reduction phase, combining all results using the sequential inner product and then returns the result to the upper level.

The validation involves the following steps (applied for different vector sizes):

1. Estimate the amount 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 of tje MultiBSP [8].
4. Run the vector inner product algorithm.
5. Compare the results with the theoretical prediction.



(a) Instance #1: *dell32* (b) Instance #2: *jolly*

**Fig. 8.** Comparison between the real execution time against the theoretical execution time

Fig. 8 graphically presents the comparison between the real execution time against the theoretical execution time for both studied architectures.

The results show that when using a vector with less than $2^8$ elements, the real execution time is larger than the theoretical time. This happens mainly because with few data, the time for spawning threads adds a significant overhead compared with the time to calculate a vector slice at level$_i$. For *dell32*, when computing vectors with more than $2^8$ elements, both curves have the same slope, then we can say that both times are relative and the measure is stabilized. For *jolly*, the predicted and execution times have a different behavior. There is an ideal point where both measures are the same, but when the vector is larger than $2^8$ elements, the execution time increases slower than the predicted time. The good results in Fig. 8(a) validates the proposed approach, as the values $g_i$ and $L_i$ used in the predicted time are very close to the real time. On other hand, in Fig. 8(b) the predicted time is not as close to the real time as we expect. However, the theoretical time is always greater than the real time, so it is useful as an accurate lower bound for predictions.

## 5 Conclusions and Future Work

This work presented MBSPDiscover[4], an automatic tool for characterizing multicore architectures based in the MultiBSP computational model. The proposed

[4] Available from `http://runtime.bordeaux.inria.fr/sehloc/`

benchmark computes the parameters $g$ and $L$ (communication and synchronization cost) for the MultiBSP model. It is adaptable to any hierarchical architecture and its output is a structure with the information of each level, useful for programming applications following the MultiBSP model.

We applied the benchmark to characterize and evaluate two actual HPC multicore systems. In order to validate the results, we designed and implemented a particular problem in the MultiBSP model, and predicted its execution costs. The results demonstrated that the execution time can be satisfyingly predicted when using the information from the benchmark, especially for the *dell32* machine.

The main lines for future work are related to verify the results of the MBSPDiscover benchmark using a suite of algorithms, and extend the library for heterogeneous multicore clusters by including a network level.

**Acknowledgements.** This research is partly funded by the STIC-AmSud program partners MINCyT (Argentina), Inria (France), and ANII (Uruguay), through the SEHLOC project.

## References

1. Bisseling, R.: Parallel scientific computation: a structured approach using BSP and MPI. Oxford University Press, Oxford (2004)
2. Bonorden, O., Juurlink, B., von Otte, I., Rieping, I.: The Paderborn University BSP (PUB) Library. Parallel Comput. 29(2), 187–207 (2003)
3. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: Hwloc: A generic framework for managing hardware affinities in HPC applications. In: 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, pp. 180–186 (2010)
4. Hill, J., McColl, B., Stefanescu, D., Goudreau, M., Lang, K., Rao, S., Suel, T., Tsantilas, T., Bisseling, R.: BSPlib: The BSP programming library. Parallel Computing 24(14), 1947–1980 (1998)
5. Lobachev, O., Guthe, M., Loogen, R.: Estimating parallel performance. J. Parallel Distrib. Comput. 73(6), 876–887 (2013)
6. Savadi, A., Deldari, H.: Measurement latency parameters of the MultiBSP model: A multicore benchmarking approach. J. Supercomput. 67(2), 565–584 (2014)
7. Valiant, L.: A bridging model for parallel computation. Commun. ACM 33(8), 103–111 (1990)
8. Valiant, L.: A bridging model for multi-core computing. J. Comput. Syst. Sci. 77(1), 154–166 (2011)
9. Yzelman, A.N.: Fast sparse matrix-vector multiplication by partitioning and reordering. Ph.D. thesis, Utrecht University, Utrecht, the Netherlands (October 2011)