

A Performance Modeling, Analysis and Prediction System for Parallel Programs

M. Printista, F. Piccoli

Universidad Nacional de San Luis
Ejército de los Andes 950,
(5700) San Luis, Argentina
mprinti@unsl.edu.ar

J.A. González, C. Rodríguez León, G. Rodríguez and F. de Sande

Dpto. Estadística, I.O. y Computación
Universidad de La Laguna
La Laguna, 38271, Spain
casiano@ull.es

Abstract

The performance of parallel applications is affected by many factors which need not be considered with sequential programs. Consequently, performance analysis for parallel applications takes a multi-dimensional approach.

OBSP* is a particular complexity analysis model based on the BSP model. The model makes still use of the BSP concept of super-step but it considers *oblivious* synchronization and groups partitioning. Furthermore, it associates a different proportionality constant with each code basic block, and analogously, it associates different latencies and bandwidths with each different communication block. Using this approach implies that the parameters evaluation not only depend on the given architecture, but also reflect algorithm characteristics. Such parameter evaluation must be done for every algorithm. This is a heavy task, implying design of experiments, timing, statistics and multi-parameter fitting algorithms. Software support is required. This research presents a *Performance System* for effecting these tasks. The study presented in this paper attempts to combine a simple analytical model with run-time informations to provide a reasonable support for algorithms analysis and prediction. The Performance System we describe use of statistical analysis to determine model coefficients. *Multivariate Techniques* are an essential part of the overall system.

1 Introduction

Performance prediction is an important tools for performance analysis of parallel applications. It involves modeling program performance as a function of the hardware and software characteristics of a system. By changing these characteristics in the model, program execution time can be predicted for a variety of platforms and configurations.

Among the analytical models, the Bulk Synchronous Parallel (BSP) model [10] is the most popular. The computational BSP model considers a parallel machine made of a set of P processor-memory pairs, a global communication network and a mechanism for synchronizing the processors. The BSP computations consists of a sequence of super-steps, separated by *barrier synchronizations*. The term barrier synchronization is traditionally used to denote a logical point in a program that all processors must reach before the computation may proceed. In the BSP model, however, it means something stronger. It also requires that

all communication operations have completed. Because of this, a simple binary tree algorithm can not be implemented efficiently.

Several authors recognized that a global barrier synchronization is expensive in current parallel architectures and propose to expand the BSP software with a zero-cost synchronization mechanism, known as *Oblivious Synchronization*. The Paderborn University BSP (PUB) library [1] is a parallel library based on BSP model. It provides, in addition to basic BSP functionality, a rich set of collective communication operations, provides the possibility to dynamically partition the processors into several subsets and introduces the concept of BSP objects.

In the PUB library, a barrier is implemented by a global reduction on the number of message each processor has outstanding. This method causes overhead. However, in many problems, the programmer knows how many messages each processor is due to receive. In that case, communication can be avoided altogether by using oblivious synchronization. In a previous work, the authors introduced the Oblivious BSP model (OBSP) [5] to deal with both oblivious synchronization and group partitioning.

There are other sources of inaccuracy intrinsic to the definition of BSP. One comes from characterizing the computing time through a single parameter s , considering that all the elementary local operations take the same quantity of time (called time step). Significant differences are observed in practice, partly due to the separate nature of the operations (number of floating point arithmetic operations, number of memory transfers, etc.) involved ([12], page 123). The OBSP model associates a different proportionality constant with each basic block (maximal segment of code without jumps). Analogously, we found that some operations communications have significantly different costs than others depending on the particular communication pattern involved. In [9] is showed the impact of such patterns in the h -relation time. In the following section, we present the OBSP* model, which associates different latencies and bandwidths with the same h -relation, depending on the pattern involved.

These parameters are not only architecture dependent, but also reflect algorithm characteristics. Such parameter evaluation is a heavy task, implying experiment design, timing and statistics.

This work attempts to find a hybrid approach, proposing an analytical model supported by a profiling tool. A profile will show the amount of time spent in different programs components obtained by sampling techniques. Performing measurements requires special purpose hardware and software and, since the target machine is used, the measurement method can be highly accurate [3, 4, 7, 8]. Performance analysis and prediction process consists of static analysis, dynamic analysis and prediction phases, which we explain in section 3. Multivariate data analysis techniques simplify our performance prediction process by deriving system parameters from experimental runs of the same application. In Section 4 we present some experimental results. The conclusion are showed in Section 5.

2 The OBSP* Model

As in ordinary BSP, the execution of a PUB program on a BSP machine $X = 0, \dots, P - 1$ consists of super-steps. However, as a consequence of the oblivious synchronization, processors may be in different super-steps at a given time. Still it is true that:

- Super-steps can be numbered starting at 1
- The total number of super-steps R , performed by all the P processors is the same
- Although messages sent by a processor in super-step s may arrive to another processor executing an earlier super-step $r < s$, communications are made effective only when the receiver processor reaches the end of super-step s

Let us assume in first instance that no processor partitioning is performed in the analyzed task T . If the super-step s ends in an oblivious synchronization, we define the set $\Omega_{s,i}$ for a given processor i and super-step s as the set:

$$\Omega_{s,i} = \{j \in X \mid \text{processor } j \text{ sends a message to processor } i \text{ in super-step } s\} \cup \{i\}$$

while $\Omega_{s,i} = X$ when the super-step ends in a global barrier synchronization. Processors in the set $\Omega_{s,i}$ are called "the incoming partners of processor i in step s ". Usually it is accepted that all the processors start the computation at the same time. The presence of partition functions forces us to consider the most general case in which each processor i joins the computation at a different initial time ξ_i . Denoting by $\xi = (\xi_0, \dots, \xi_{P-1})$ the vector for all processors, the OBSP* time $\Phi_{s,i}(T, X, \xi)$ taken by processor $i \in X$ executing task T to finish its super-step s is recursively defined by the formulas:

$$\Phi_{1,i}(T, X, \xi) = \max_{j=0, \dots, P-1} \{ \mathbf{W}_{1,j} + \xi_j \mid j \in \Omega_{1,i} \} + (\mathbf{g} * \mathbf{h}_{1,i} + \mathbf{L}) \quad (1)$$

$$\Phi_{s,i}(T, X, \xi) = \max_{s=2, \dots, R} \{ \Phi_{s-1,j}(T, X, \xi) + \mathbf{W}_{s,j} \mid j \in \Omega_{s,i} \} + (\mathbf{g} * \mathbf{h}_{s,i} + \mathbf{L}) \quad (2)$$

where constant R denotes the total number of super-steps and $\mathbf{W}_{s,j}$ denotes the time spent in computing by processor j in step s .

The value $h_{s,i}$ is defined as the number of bytes communicated by processor i in step s , that is:

$$h_{s,i} = \max \{ in_{s,j} @ out_{s,j} \mid j \in \Omega_{s,i} \}, \quad s = 1, \dots, R, i = 0, \dots, P-1$$

and $in_{s,j}$ and $out_{s,j}$ respectively denote the number of incoming/outgoing bytes to/from processor j in the super-step s . The @ operation is defined as \max or $+$ depending on the input/output capabilities of the network interface.

The next section exemplifies the use of the OBSP* model to predict the time spent by PUB algorithms.

2.1 Parallel Binary Search: OBSP* Analysis

The example used is *The Parallel Binary Search* (PBS) algorithm presented by the authors of PUB in their introductory paper [1]. The PBS problem consists on locating $SIZE = m * P$ queries (where P is the number of processors) in a butterfly data structure of $N = n * P$ keys. The root of the tree is replicated P times, and each other level is replicated half as many times as the level above. Let be $m = SIZE/P$ the number of queries that each processor has to locate. The algorithm will create a new BSP-machine at each new level of the tree. The code in Figure 1 shows the PUB implementation of the parallel binary search. Hence, the computation in each recursive level takes two super-steps; the first one involves the routing process and the second the recursive partition process.

After the first super-step, two new BSP-machines are created. For these new BSP-machines, the computing time executed by a processor j before its call to the *bsp_partition* function is:

$$w_{s,j}^* = (Cmem_0 + Cmem_1 * m/2) + Partt$$

The $m/2$ factor is due to the assumption that queries are uniformly distributed. Therefore, the initial distributions of times $(\xi_{i,d})$ when processor i starts the *bin_search* function at depth level d is given by:

```

1. void bin_search(pbsp bsp, int d, int m) {
2.   msg = bsp_createmsg(bsp, m*OVERSAMPLE * sizeof(int));
3.   temp = bspmsg_data(msg);
4.   pid=bsp_pid(bsp); nprocs=bsp_nprocs(bsp);
5.   for(i=new_m=other=0; i < m; i++)
6.     if(((query[i]<=bkey[d]) && (pid>=nprocs/2)) ||
7.        ((query[i]>bkey[d]) && (pid<nprocs/2)))
8.       temp[other++] = query[i];
9.   else
10.    query[new_m++] = query[i];
11. bsp_sendmsg(bsp, (pid+nprocs/2) % nprocs, msg, other*sizeof(int));
12. bsp_oblsync(bsp, 1);
13. msg = bsp_getmsg(bsp, 0);
14. memcpy(&query[new_m], bspmsg_data(msg), bspmsg_size(msg));
15. new_m += bspmsg_size(msg)/sizeof(int);
16. if (d==0)
17.   local_search(new_m);
18. else {
19.   part[0]= nprocs/2; part[1]= nprocs;
20.   bsp_partition(bsp, &subbsp, 2, part);
21.   bin_search(subbsp, d-1, new_m);
22.   bsp_done(&subbsp);
23. }
24. }
25.}

```

Figure 1: Parallel Binary Search Algorithm

$$\xi_{i,d} = \begin{cases} 0 & \text{for } d = 0 \\ \Phi_{1,i}(PBS, S_{d-1}, \xi_{i,d-1}) + (Cmem_0 + Cmem_1 * m/2) + Partt & \text{for } d = 1, \dots, \log(P) - 1 \end{cases}$$

Where $\Phi_{1,i}(PBS, S_{d-1}, \xi_{i,d-1})$ is the time when processor i finishes its first super-step (that is, reaches line 12 in Figure 1. Constants $Cmem_0$ and $Cmem_1$ are associated with lines 14 and 15 and constant $Partt$ corresponds to the partition process.

Let us compute $\Phi_{1,i}(PBS, S_{d-1}, \xi_{i,d-1})$. As the queries are equally distributed, the value m can be considered constant. The time spent by processor i in lines 2 – 10 is:

$$w_{1,i} = (Route_0 + Route_1 * m)$$

The h -relations in all super-steps are the same:

$$h_{1,i} = (m * sizeof(int))/2$$

and the incoming partners of processor i in super-step 1 are:

$$\Omega_{1,i} = \{i, i \text{ xor } 2^{\log(\|S_d\|)-1}\} \quad \text{for all } (PBS, S_d, \xi_{i,S_d}) \text{ and } d = 0, \dots, \log(P) - 1$$

The time taken by processors in the first super-step is calculated by formula:

$$\begin{aligned}\Phi_{1,i}(PBS, S_d, \xi_{i,d}) &= ((Route_0 + Route_1 * m) + \xi_{i,d}) + \\ &Comm_0 + Comm_1 * m/2 * sizeof(int) \\ &(\text{for } d = 0, \dots, \log(P) - 1)\end{aligned}$$

The time when processor i finishes step 2 at depth d is given by:

$$\Phi_{2,i}(PBS, S_d, \xi_{i,d}) = \begin{cases} \Phi_{2,i}(PBS, S_{d+1}, \xi_{i,d+1}) + D & \text{for } d = 0, \dots, \log(P) - 2 \\ \Phi_{1,i}(PBS, S_d, \xi_{i,d}) + (Cmem_0 + Cmem_1 * m/2) + SeqT & \text{for } d = \log(P) - 1 \end{cases}$$

The time $\Phi_{2,i}(PBS, S_{d+1}, \xi_{i,d+1})$ stands for the instant when the recursive call at line 17 (recursion level $d + 1$) finishes. The constant D corresponds to the *bsp_done()* call at line 23.

The $(PBS, S_{\log(P)-1}, \xi_{i,\log(P)-1})$ -machines are the last BSP-machines created, and their second super-step includes the calling to the sequential binary search (*SeqT*). Since our study concentrates in the prediction accuracy of the communication stages, in the experiments, we have substituted the sequential binary search by an empty body.

The oblivious OBSP* time $\Phi_{2,i}(PBS, S_0, \xi_{i,0})$ of the *PBS* algorithms in a BSP-machine with P processors at the outmost level can be obtained by successive substitutions:

$$\begin{aligned}\Phi_{2,i}(PBS, P, 0) &= \log(P) * [Route_0 + Route_1 * m + Comm_0 + \\ &Comm_1 * m/2 * sizeof(int) + (Cmem_0 + Cmem_1 * m/2)] \\ &+ (\log(P) - 1) * [Partt + D] + SeqT\end{aligned}\tag{3}$$

We will continue to use this example to illustrate others aspects in future sections.

3 Performance Analysis and Prediction System

Performance Prediction System is a process to measure the time spent in the execution of the code portions corresponding to the algorithm we want to study. The objective is to adjust the formula coefficients with a statistical approximation and to obtain the exact complexity formula.

The logical structure consists of 3 phases: *static analysis*, *dynamic analysis* and *prediction phase*. In the static phase, the time complexity of each basic block is specified manually by inserting compiler **pragma** directives around the blocks. Then, a source-to-source compiler takes as source this program with complexity formula and produce as output an instrumented code. This output has extra code in order to carry out the time measurement task and to determine the relative contribution of each block in application code.

During the dynamic analysis, the instrumented program produced by the source-to-source compiler is compiled and executed in the usual way. The output data generated with run time information are gathered in the appropriated trace-files. The Performance Prediction System must to profile the time spent for every instance on every processor on the parallel machine. This is a complex situation. The same blocks in different processors may take longer in one machine than in another, can start or stop in non deterministic way, some instances can interact with others or run completely different portions of code depending on processor *Identifier*.

On the Prediction phase, the trace files are inputs of a profiler/analyzer interpreter. The trace files consist of a statistically significant number of experimental runs with different problem sizes and number of processors. Multivariate techniques are used to obtain the values of the communications and computations constants, to establish the segment where the values of the constants are valid and to facilitate the prediction of the performance of the algorithm for any input values.

3.1 Multivariate Data Analysis

Multivariate statistics refers to a group of inferential techniques that have been developed to handle situations where sets of variables are involved as predictors of performance. The performance system that we describe here is designed to make use of statistical analysis in determining model coefficients.

The system centers in each "basic block", called an *experiments*. Each experiments, Γ , has an associated instructions set Υ_Γ . In term of the number of times that each different operation is performed, Υ_Γ is partitioned in k disjoint subset Υ_{Γ_i} :

$$\bigcup_{i=1}^k \Upsilon_{\Gamma_i} = \Upsilon_\Gamma \text{ and } \bigcap_{i=1}^k \Upsilon_{\Gamma_i} = \emptyset$$

A model relating the experiment execution time to a set of independent variables Υ_{Γ_i} is of the form:

$$f_\Gamma = \sum_{i=1}^k \beta_i * f_{\Gamma_i} \quad (4)$$

where f_Γ specifies the execution time model for Γ experiment, f_{Γ_i} are the basis functions of the model and $\beta_i (i = 0, \dots, k)$ will be estimated by the parameter estimation algorithm. The basis function can be performance determinants (problem size, loops variables, number of processors) or function of performance determinants.

Note that every these models, f_Γ is a linear function of unknown parameters values β_i and not necessarily a linear function of known constant f_{Γ_i} . This type of model is called a *linear regression statistical model* [6].

In the physical interpretation of the linear model, f_Γ is equal to an expected value plus a random error, ϵ , $E(f_\Gamma) = \sum_{i=1}^k \beta_i * f_{\Gamma_i} + \epsilon$. From a practical point of view, ϵ acknowledges our inability to provide an exact model. In repeated experimentation f_Γ bobs about $E(f_\Gamma)$ in a random manner because we have failed to include in our model all the many variables that may affect f_Γ .

Linear *least-squares models (LSQ)* estimate the coefficients $\beta_0, \beta_1, \dots, \beta_k$ to minimize the squared sum of errors between predicted and experimental values. If the response and predictors corresponding to the i th of $numtests$ observations are $f_{\Gamma,i}, f_{\Gamma_{i,1}}, \dots, f_{\Gamma_{i,k}}$, then the fitting criterion chooses the β_i to minimize:

$$\sum_{test=1}^{numtests} (f_{\Gamma,test} - (\sum_{i=1}^k \beta_i * f_{\Gamma_{i,test}}))^2 \quad (5)$$

One side effect of using the *LSQ* criterion is that outliers (experimental values with a large error) tend to have a big effect on the derivation of β_i . This is reasonable because the seriousness of an error in prediction goes up faster than linearly with the magnitude of the error. In order to minimize the effects of erroneous measurements, the system accepts to remove manually outliers which were suspect. Moreover, we have observed that a simple model is not accurate for all range of basis functions. This suggests to decomposes the space of performance determinants into regions. The estimation algorithm automatically calculates the set of parameters for each new region. The algorithm to find the intervals of the parameters implies the use of heuristic statistical techniques. An ordinary multidimensional linear fit is performed over the preprocessed sample. If the errors are larger than a fixed "error threshold", the variable space is divided in two. The point

that maximizes the variation of the error is chosen as splitting point. This process is repeated until the errors obtained are under the error threshold or the number of intervals exceeds a “number of intervals threshold”.

At this point the system can make predictions. The set of algorithms and techniques that implement the prediction phase can be used to predict program execution time for different problem sizes and number of processors and to examine other aspects of program performance. The algorithm evaluates the linear function $\sum_{i=1}^k \beta_i * f_{\Gamma_i}$. It already has estimative coefficients β_i and the required bases functions have to be evaluated using the prediction requirement inputs.

3.2 CALL: An environment for Parallel Performance System

The CALL system [11] consists of a translator (called `call`), a run time library (`call.h`) and an analyzer interpreter (`llac`). It can not only be used for the analysis of sequential programs, but also for OxfordBSP, PUB, OpenMP and MPI parallel programs. More than a prediction tool, it is a performance measurement and modeling tool. It can be used to confirm or reject the predictive accuracy of a given performance model, not just OBSP*. The OBSP* model needs the CALL tool to be feasible, but the tool itself is independent of the performance model.

The run time library makes use, if installed, of the PAPI library [2]. From a sequential or parallel C program annotated with `call` **pragmas**, the `call` compiler produces two files containing the necessary code (`*.call.c`) and structures (`*.call.h`) to save variable values, to time the corresponding code and to produce the reports required by the `llac` analyzer. Once the program has been compiled and executed, the `llac` interpreter allows the programmer to play with the resulting data, considering subsets, transformations of them or merging them with other data coming from other experiments. The `llac` analyzer deduces the values of the parameters involved, the segments where they are valid, the variation of these parameters with the input values, predicts the behavior of the different experiments under study and allows their graphic visualization. It also warns the user when the lack of accuracy is due to possible errors in the proposed model (errors in the proposed formula).

To exemplify the combined use of the OBSP* model and the CALL tool to predict the time spent by PUB programs we revisit the **PBS** algorithm.

3.2.1 Parallel Binary Search: Performance prediction using CALL

Line 1 in code of Figure 2 warns the `call` compiler to notice that this is a BSP parallel program using PUB. The optional argument `gbps` points to the global BSP object. This information will be used by the `report` clause in line 15. When executed, the code generated from this line, will collect all the statistics sampled by the different processors, routing them to processor 0, where they will be dumped on the corresponding output file `pbs.call.#n.dat`.

Lines 10 to 12 in code of Figure 2 define a basic block named “`call experiment`”. The experiment is named after the identifier following the pragma, `pbs` in the example. The complexity formula ruling the time taken by the segment of code delimited by the experiment appears after its name. The constants in the complexity formula are referenced indexing its name. In this case there are three constants, `pbs[0]`, `pbs[1]` and `pbs[2]`. The previous complexity formula (3) obtained for $\Phi_{2,i}(PBS, P, 0)$ has to be rewritten in terms of the corresponding program variables:

$$pbs[0] + pbs[1] * \log(gnprocs) + pbs[2] * M * \log(gnprocs) \quad (6)$$

where

$$\begin{aligned} pbs[0] &= SeqT - Partt - D \\ pbs[1] &= Route_0 + Comm_0 + Cmem_0 + Partt + D \\ pbs[2] &= Route_1 + Comm_1 * sizeof(int)/2 + Cmem_1/2 \end{aligned}$$

Any *call complexity formula* must be in canonical form, i.e. has to be a sum of terms made of complexity constants multiplied by expressions. More general, the experiment constant must be the only multiplicative constant in each term. This constraint is due to singularities appearing in the multidimensional fit algorithm [6] used by the call interpreter, *llac*.

For each experiment, the compiler generates the code to time it and to save its state for later report and treatment.

```

1. #pragma parallel PUB gbsp
2. ...
3. M = size/gnprocs;
4. gkey_init (size); /* generate keys */
5. temp = (int*) malloc ((M + OVERSAMPLE) * sizeof(int));
6. query = (int*) malloc ((M + OVERSAMPLE) * sizeof(int));
7. init_query (M); /* set the queries */
8. bsp_sync (gbsp);
9. d = log2(gnprocs)-1;
10. #pragma cll pbs pbs[0]+pbs[1]*log(gnprocs)+pbs[2]*M*log(gnprocs)
11. bin_search(gbsp, d, M);
12. #pragma cll end pbs
13. bsp_sync (gbsp);
14. ...
15. #pragma cll report all
16. ...

```

Figure 2: The *pbs* experiment in CALL

As you can see, experiments can also be *nested*. In the PBS example we can profile other blocks. In code of Figure 3 we insert pragma directive to measure the time spent in the execution of a code part, named "*Comm*".

```

1. void bin_search(pbsp bsp, int d, int m) {
2.     ...
13. #pragma cll Comm[0]+Comm[1]]* m/2
14.     bsp_sendmsg(bsp,(pid+nprocs/2) % nprocs,msg,other*sizeof(int));
15.     bsp_oblsync(bsp,1);
16.     msg = bsp_getmsg(bsp, 0);
17. #pragma cll end Comm
18.     ...
30. }

```

Figure 3: The *Comm* experiment in CALL

4 Results

The PBS algorithm was run in the Cray-T3E Supercomputer for different sizes. CALL registered the time insumed for each *cll experiment* in the PBS algorithm. Multivariate statistical analysis often requires a larger number of samples than other fitting methods. PBS and other similar algorithms are good candidates for this environment because multiple samples can be obtained from a single program using different problem sizes.

The previous complexity formula (6) involves 3 predictor variables which are shown in Table 1.

f_{Γ_0}	f_{Γ_1}	f_{Γ_2}
1	$\log(gnprocs)$	$\log(gnprocs)*M$

Table 1: Bases Functions on the PBS algorithm

The analyzer interpreter, *llac*, is then invoked to compute the best-fit parameters of each *cll experiment*. The row 3 in Table 2 shows these values for the *pbs* experiment and the prediction equation result then: $\beta_0 + \beta_1 * \log(gnprocs) + \beta_2 * \log(gnprocs) * M$. Additionally, *llac* return the variance-covariance matrix of the model parameters and the sum of squared of the residuals from the best-fit. These will allow us to develop test and confidence interval. The standard error row is an estimate of how much the regression coefficient β_i will vary from sample to sample.

Parameter	β_0	β_1	β_2
	<i>cll_pbs</i> [0]	<i>cll_pbs</i> [1]	<i>cll_pbs</i> [2]
Value	1.47707e-03	9.17329e-04	5.67159e-07
Std. Err.	1.21373e-06	3.49486e-07	3.39023e-12

Table 2: pbs cll experiment Parameters

The prediction algorithm of *llac* was invoked to predict performance for some instances. Table 3 presents the predicted results obtained for PBS algorithm for a $SIZE = 1048576$. The column *M* shows the number of queries that each processors had to locate.

Prediction Requirement Inputs		Predicted by CALL	Real Time	Error %
gnprocs	M			
2	524288	0.2997492	0.2981720	0.52%
4	262144	0.3006665	0.2930070	2.61%
8	131072	0.2272451	0.2233757	1.73%
16	65536	0.1538238	0.1541987	0.24%

Table 3: Real and OBS* predicted times for the PBS in the CRAY T3E

Errors are basically due to the lack of accuracy for the communication component. Nevertheless, these errors under 3% imply a considerable improving in accuracy, when compared with previous results. The benefit is remarkable if they are compared with the obtained using raw BSP, where the errors can increase up to much larger values [12].

5 Conclusions

The Performance System we described here is designed to make use of statistical analysis to determine the coefficients analytical model. The multivariate techniques described are particularly applicable because of the large number of samples the profiling system allows to obtain. In particular, these techniques improve the accuracy of our performance prediction model. The utility of prediction data is also increased by evaluating the confidence interval for estimated execution time.

Acknowledgments

We would like to thank to LIDIC, Universidad Nacional de San Luis, Argentina, Edinburgh Parallel Computing Centre, and to Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT). This research has been partially supported by Comisión Interministerial de Ciencia y Tecnología under project TIC1999-0754-C03 and by the European TRAC program.

References

- [1] Bonorden O., Juurlink B., von Otte I., Rieping, I.: *The Paderborn University BSP (PUB) Library-Desing, Implementation and Performance*. 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing (IPPs/SPDP), 1999.
- [2] Browne S., Dongarra J., Garner N. Ho G., Mucci P.: *A Portable Programming Interface for Performance Evaluation on Modern Processors*. The International Journal of High Performance Computing Application , 2000.
- [3] Espinosa A., Margalef T., Luque E.: Automatic Performance Evaluation of Parallel Programs. Proc. Of the 6th Euromicro Workshop on PDP. IEEE CS. 43-49. 1998.
- [4] Fahringer T., Zima H.: Static Parameter Based Performance Prediction Tool for Parallel Programs. ICS. ACM Press. 207-219. 1993.
- [5] González J.A., León C., Piccoli F., Pristinta M., Roda J.L., Rodríguez, Sande F.: Performance Prediction of Oblivious BSP Programmas 7th International Euro-Par Conference, Springer-Verlag, 2001.
- [6] Groom D.E. et al.: *Statistics*. The European Physical Journal **C15**, <http://pdg.lbl.gov/2000/statppbook.pdf>, 2000.
- [7] Heath M., Etheridge J.: Visualizing the Performance of Parallel Programs. IEEE Software. 8 (5), 29-39. 1991.
- [8] Labarta J., Girona S., Pillet V., Cortes T., Gregoris L.: Dip: A Parallel Program Development Environment. International Euro-Par'96 Conference, Springer-Verlag. 1996.
- [9] Rodríguez C., Roda J.L., Morales D.G., Almeida F.: *h-relation models for Current Standard Parallel Platforms*. 4th International Euro-Par Conference, Springer-Verlag, 234-243, 1998.
- [10] Valiant L.G.: *A Bridging Model for Parallel Computation*. Communications of the ACM, 33(8) (1990) 103-111
- [11] Rodriguez Herrera G.: *CALL: A Complexity Analysis Tool* Master degree thesis. (ps and html version available at [http://nereida.deioc.ull.es/\\$call/](http://nereida.deioc.ull.es/$call/))
- [12] Zavanella A., Milazzo A.: *Predictability of Bulk Synchronous Programs Using MPI*. 8th Euromicro PDP (2000) 118-123