Analisis and Tools for Performance Prediction

González J.A. [1], León C.[1], Piccoli, F. [2], Printista M.[2], Roda J.L.[1], Rodríguez C.[1]
Rodríguez J.M.[1] and Sande F.[1]

[1]Dpto. Estadística, Investigación Operativa y Computación,
Universidad de La Laguna, Tenerife, Spain

[2] Universidad Nacional de San Luis
Ejército de los Andes 950, San Luis, Argentina
{mpiccoli, mprinti}@unsl.edu.ar

Abstract. We present an analytical model that extends BSP to cover both oblivious synchronization and group partitioning. There are a few oversimplifications in BSP that make difficult to have accurate predictions. Even if the numbers of individual communication or computation operations in two stages are the same, the actual times for these two stages may differ. These differences are due to the separate nature of the operations or to the particular pattern followed by the messages. Even worse, the assumption that a constant number of machine instructions takes constant time is far from the truth. Current memory hierarchies imply that memory access vary from a few cycles to several thousands. A natural proposal is to associate a different proportionality constant with each basic block, and analogously, to associate different latencies and bandwidths with each "communication block". Unfortunately, to use this approach implies that the evaluation parameters not only depend on given architecture, but also reflect algorithm characteristics. Such parameter evaluation must be done for every algorithm. This is a heavy task, implying experiment design, timing, statistics, pattern recognition and multi-parameter fitting algorithms. Software support is required. We have developed a compiler that takes as source a *C* program annotated with complexity formulas and produces as output an instrumented code. The trace files obtained from the execution of the resulting code are analyzed with an interactive interpreter, giving us, among other information, the values of those parameters.

Keywords: Complexity model, Performance analysis, Performance prediction, Performance profiling, Oblivious synchronization.

1 Introduction

Most of the approaches to performance analysis and prediction fall into two categories: Analytical Modeling and Performance Profiling. Analytical methods use models of the architecture and the algorithm to predict the program runtime. The analysis can be independent of the target architecture. Among the analytical models, the Bulk Synchronous Parallel (BSP) model [13] is the most popular . Profiling may be conducted on an existing parallel system to recognize current performance bottlenecks. Performing measurements requires special purpose hardware and software and, since the target machine is used , the measurement method can be highly accurate [4, 5, 9, 11, 14]. Although much work has been developed in Analytical Modeling and in Parallel Profiling, sometimes seems to be a divorce between them. Analytical modeling is considered to be too theoretical to be accurate in practical cases and profiling analysis is criticized of lack of generality. This work attempts to find a hybrid approach, proposing and analytical model supported by a profiling tool. The class of parallel algorithms whose performance behavior can be predicted includes the Bulk Synchronous Parallel Algorithm class.

The asynchronous nature of some parallel paradigms like farms and pipelines hampers the efficient implementation in the scope of a flat-data-parallel global-barrier Bulk Synchronous Programming software like the BSPLib [10]. To overcome these limitations, the Paderborn University BSP library (PUB [1]) offers the use of collective operations, *processor-partition* operations and *oblivious synchronization*. In addition to the BSP most common features, PUB provides the capacity to partition the current BSP machine into several subsets, each of which acts as an autonomous BSP computer with their own processor numbering and synchronization points. The authors of the *BSP Worldwide Standard Library* report claim that an unwanted consequence of group partitioning is a loss of accuracy [7, page 18]. Other of the novel features of PUB is the oblivious synchronization. It is implemented through the *bsp_oblsync(bsp,n)* function, which does not return until *n* messages have been received. Although its use mitigates the synchronization overhead, it implies that different processors can be in different supersteps at the same time. The BSP semantic is preserved in PUB by numbering the supersteps and by ensuring that the receiver thread buffers messages that arrive out of order until the correct superstep is reached.

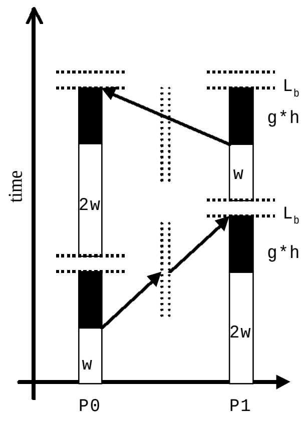

Fig. 1. Oblivious Supersteps

Fig. 1 on the left illustrates the impact of the second improvement, oblivious synchronizations, in BSP prediction accuracy. The diagram corresponds to an application running on a *2*-processor machine in *2* supersteps. White areas correspond to computation while black areas stand for communication. During the first superstep, processor *P1* performs a task heavier (*4 sec*) than the performed by processor *P0* (*2 sec*). After an exchange operation *(2 sec)* and an oblivious synchronization, the situation is inverted and processor *P0* does the lighter part compensating the former unbalance. Finally, there is another oblivious exchange between processors *P0* and *P1 (2 sec)*. While the actual time is *10 sec*, the BSP prediction corresponding to a global synchronous barrier is of *12 sec*.

There are other sources of inaccuracy intrinsic to the definition of BSP. One comes from characterizing the computing time *W* 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 [15, page 123]. The other comes from characterizing the communication time through two single parameters *g* and *L*, considering that any *h*-relation takes the same quantity of time, independently of the particular communication pattern involved. In [12] we studied the impact of such patterns in the *h*-relation time.

A natural (and more realistic) alternative is to associate a different proportionality constant with each basic block (maximal segment of code without jumps), and analogously, to associate different latencies and bandwidths with the same *h*-relation, depending on the pattern. However, although these new parameters means an improvement, this approach does not suffices to have accurate predictions. Most modern microprocessors have at least two levels of cache. Furthermore, operating systems use main memory as a cache for a larger virtual address space for each process and translate between virtual addresses used by a program and the physical addresses required by the hardware. Memory is divided into blocks called pages. To keep the overhead address translation low, the most recently used page addresses are cached in a translation lookaside buffer (TLB). While a L1 cache hit typically takes 2 or 3 cycles a TLB miss requiring only reload of the TLB can take the order of 2000 cycles [2 page 3]. The assumption that a constant number of machine instructions takes constant time is far from the truth. Any model attempting to be accurate has to have into account this paradoxical "variation of the constants". On the other side of the balance, the model has to be simply enough to be practicable. To have into account these considerations implies the evaluation of a finite (but perhaps large) number of parameters. These parameters are not only architecture dependent, but also reflect algorithm characteristics. Such parameter evaluation is a heavy task, implying experiment design, timing, statistics and multi-parameter fitting algorithms. It does not seem reasonable to ask the algorithm designer to carry on by hand such tasks for every developed program.

Our proposal attempts to give a solution to all the aforementioned problems. In a previous work, the authors introduced the *Oblivious BSP model* (OBSP) to deal with both oblivious synchronization and group partitioning [6]. Starting from OBSP, we now address the problem of how to relax the number of parameters without introducing an unbearable complexity. The resulting model, called OBSP* is introduced in the following section. The third section presents *CALL*, a prototype of a software tool for the analysis and prediction of PUB BSP (and most MPI) programs. The tool consists of "*pragma*" language extending *C*, its associated compiler and a profiler/analyzer interpreter of the trace files generated by the instrumented target. The analyzer provides the values of the communication and computation constants, establishes the segments where the *values of the constants are valid* and facilitates the prediction of the performance of the algorithm for any input values. Both the theory and the computational experiences allow us to conclude, in the fourth section, that an OBSP* analysis means an improvement in prediction accuracy when compared with using traditional BSP (if in scope) or OBSP.

2 The OBSP* model

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

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

Lets assume in first instance that no processor partitioning is performed in the analyzed task *T*. If the superstep *s* ends in an oblivious synchronization, we define the set $W_{s,i}$ for a given processor *i* and superstep *s* as the set

$$W_{s,i} = \{j \in X \ / \text{ Processor } j \text{ sends a message to processor } i \text{ in superstep } s\} \cup \{ i\} \qquad \textbf{(1)}$$

while $W_{s,i} = X$ when the superstep ends in a global barrier synchronization. In fact, this last expression can be considered a particular case of formula (1) if it is accepted that barrier synchronization carries (directly or indirectly) an *AllToAll* communication pattern. Processors in the set $W_{s,i}$ 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 x_i. Denoting by $x = (x_0 , ..., x_{p-1})$ the vector for all processors, the *OBSP** time $F_{s,i}(T, X, x)$ taken by processor $i \in X$ executing task *T* to finish its superstep *s* is recursively defined by the formulas:

$$F_{1,i}(T, X, x) = max\ \{W_{1,j} + x_j \ \ / j \in W_{1,i} \} + (g * h_{1,i} + L),\ i = 0,..., P-1, \qquad \textbf{(2)}$$

$$F_{s,i}(T, X, x) = max\ \{F_{s-1,j}(T, X, x) + W_{s,j} \ \ / j \in W_{s,i}\} + (g * h_{s,i} + L),$$
$$s = 2,..,R,\ \ i = 0,..., P-1$$

where $W_{s,j}$ denotes the time spent in computing by processor *j* in step *s*. Assuming the processors have an instruction set $\{I_1 ,...,I_t\}$ of size *t*, where the *i*-th instruction I_i takes time p_i, the time $W_{s,j}$ is given by the formula:

$$W_{s,j} = \sum_{i=1,t} w_{s,i,j} * p_i \text{ where } w_{s,i,j} = \text{number of instructions of the class } Ii \qquad \textbf{(3)}$$
$$\text{executed by processor } j \text{ in step } s.$$

Constant R denotes the total number of supersteps, and constants g and L vary depending on the algorithm**.** The value $h_{s,i}$ is defined as the number of bytes communicated by processor i in step s, that is:

$$h_{s,i} = max\ \{in_{s,j}\, @\, out_{s,j} / j\, Î\, W_{s,i}\},\ \ s = 1,...,R,\ i = 0,...,P-1 \qquad \textbf{(4)}$$

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

At any time, processors are organized in a *hierarchy of processor sets*. A processor set in *PUB* is represented by a structure called a *BSP object*. Let $Q\, Í\, X$ be a set of processors (i.e. a *BSP* object) executing task T. When processors in Q execute function *bsp_partition(t_bsp *Q, t_bsp *S, int r, int *Y)*, the set Q is divided in r disjoint subsets S_i such that,

$$\begin{gathered} Q = È_{0\ £\ i £\ r-1}\ S_i\ , \\ S_0 = \{0,...,\ Y[0]-1\}, \\ S_i = \{Y[i-1],...,\ Y[i]-1\},\ 1\ £\ \ i\ £\ r-1 \end{gathered} \qquad \textbf{(5)}$$

After the partition step, each subgroup S_i acts as an autonomous BSP computer with its own processor numbering, messages queue and synchronization mechanism. The time that processor $j \in S_i$ takes to finish its work in task T_i executed by the BSP object S_i is given by

$$F_{Ri,\,j}(T_i,\ S_i,\ F_{s-1,j}+w^*_{s,j}) \text{ such that } j\, Î\ \ S_i\ ,\ i = 0,...,r-1, \qquad \textbf{(6)}$$

where R_i is the number of supersteps performed in task T_i and $w^*_{s,j}$ is the computing time executed by processor j before its call to the *bsp_partition* function in the s-th superstep of original set Q. Observe that subgroups are created in a stack-like order, so function *bsp_partition* and *bsp_done* incur no communication. This implies that different processors in a given subset can arrive at the partition process (and leave it) at different time. From the point of view of the parent machine, the code executed between the call to *bsp_partition* and *bsp_done* behaves as computation (i.e. like a call to a subroutine).

3 An OBSP* Environment for Performance Prediction

The *CALL* system consists of a translator (called *call)*, a run time library (*cll.h*) and an analyzer interpreter (*llac*). Although it can be used for the analysis of sequential programs, it gives also support for the prediction of *PUB* and *MPI* parallel programs. The run time library makes use, if installed, of the *PAPI* library [2]. Fig. 2 shows the execution system of *CALL*. From a sequential or parallel *C* program annotated with *call* pragmas, the *call* compiler produces two files containing the necessary code *(*.cll.c)* and structures (**.cll.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 experiences. The 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.

To exemplify the combined use of the *OBSP** model and the *CALL* tool to predict the time spent by *PUB* programs we have chosen the *Fast Fourier Transform* (FFT) algorithm. The Fourier Transform (FT) decomposes a function into its different-frequency sinusoidal components. In 1965, Tukey and Cooley [3] proposed a Discrete Fourier Transform algorithm with a number of computations of order *O(N log(N))*. It is a divide and conquer algorithm based on the fact that the transformation of a digital signal can be obtained by combining the transforms of its even and odd components. Although it is not a requirement, the expression of the algorithm is simplified using a signal size, *N* that is a power of 2.

Line 1 in Figure 3 warns the *call* compiler to notice that this is a *BSP* parallel program using *PUB*. The optional argument *gbsp* points to the global *BSP* object. This information will be used by the *report* clause in line 10. When executed, the code generated from this line will collect all the statistics sampled in the different processors, routing them to processor *0,* where they will be dumped on the corresponding output file *fft.cll.#n.dat.*

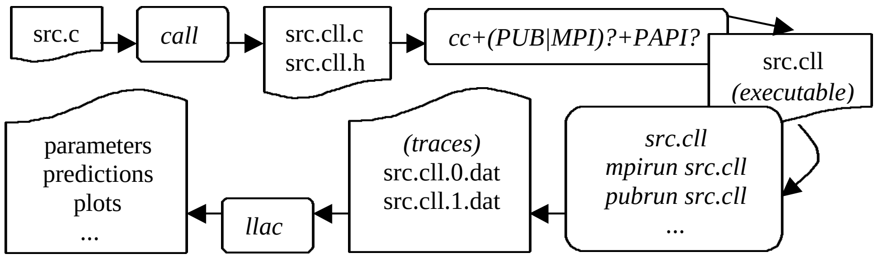

Fig. 2. Diagram of the *CALL* system

Lines 5 to 7 in Fig. 3 define a *"call experiment"*. The experiment is named after the identifier following the pragma, *fft* 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 4 constants, *fft[0], fft[1], fft[2]* and *fft[3].* 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 [8] used by the interpreter.

The complexity formula (11) for $F_{2,i}$ *(FFT,X,0)* that will be obtained for the algorithm in figure 4 is written in terms of the corresponding program variables:

*fft[0] + fft[1] * log(P)+ fft[2] *(N/P)* log(N/P)+ fft[3] *N*(P-1)/P* where the relations with the constants introduced in the next subsection are:

$$fft[0] = G[0];$$
$$fft[1] =A[0]+ B[0]+ C[0]+ E[0]+ F[0]+D[0]$$
$$fft[2] = G[1];$$
$$fft[3]=F[1]+D[1]$$

For each experiment the front end *call* compiler generates the code to time it and to save its state for later report and treatment. Starting from the trace files generated during the execution, the back end *llac* analyzer determines the values of *fft[0], fft[1], fft[2]* and *fft[3]*. Actually, what *llac* determines using linear multifit is a vector of values for each of these constants and the intervals where those values apply. For this example, the constants vary with *N* and *P*. There are usually values of *fft[0], fft[1], fft[2]* and *fft[3]* for small values of *N* and *P*, and another different values for medium sizes and so forth. When predicting the time for a concrete value –say *N = 1024, P = 32* the programmer does not need to concern with the exact parameter values. The *llac* system will choose the appropriate constant values of *fft[0], fft[1], fft[2]* and *fft[3]* (the one for the small range of *N* and *P = 32* for the example) to obtain a more precise prediction. The recognition of the intervals of validity of the constant parameters imply the use of heuristic statistical techniques.

```
#pragma parallel PUB gbsp
...
initizalize(N, a);
Roots(N/2, W);
#pragma cll fft fft[0]+fft[1]*log(P)+fft[2]*(N/P)*log(N/P)+\
                                           fft[3] *N*(P-1)/P
parDandCFFT(A, a, W, N, 1, D, gbsp);
#pragma cll end fft
bsp_sync (gbsp);
...
#pragma cll report all
...
```

Fig. 3. The *fft*_experiment

The code in Fig. 4 is a PUB implementation of the FFT algorithm annotated with *CALL* pragmas. It has as input a vector of complex numbers *a*, the vector *W* containing the *N*-th pre-computed roots of unity, the number *N* of elements, the *stride* determining a subproblem of the original problem and the pointer to the data structure, *gbsp* defining the current BSP machine. We assume that both the input data and the result vector *A* are replicated in each processor.

Let's denote by *FFT* the code presented in Fig. 4. At each level *d-1* of the recursion, there is a *PUB* machine X^{d-1} that executes two OBSP* supersteps. The time spent by a processor *i* Î X^{d-1} to perform the first superstep, $F_{1,i}(FFT, X^{d-1}, x)$, consists of four computational blocks and one communication:

a) Input signal division into its even and odd components (line 7). Because the input data is replicated on each processor, this operation can be implemented over the same vector *a*. Variable *stride* indicates the separation between logical consecutive elements in the input vector. This computation takes a constant time *A[0]*.

b) The BSP machine X^{d-1} is partitioned in two submachines X^d_j with $j = 0,1$ (lines 10-12). Under the assumption that the number of processors in X^{d-1} is a power of *2*, each submachine has the same number of processors. A *PUB* machine partition operation takes a constant time *B[0]*.

c) While one of the submachines computes the transformation of the even components, the other does the same with the odd terms. These computations correspond to the recursive calls in lines 15 and 29 respectively. The times required by each of these submachines to perform their computations are given by $F_{2,i}(FFT, X^d_j, x_{d-1,i} + A[0] + B[0])$. Where d is the recursion depth, X^d_j is the set of processors in the current BSP machine, $x_{d-1,i}$ is the time when the calling *FFT* started and $w*_{1,i} = A[0] + B[0]$ denotes the computation performed by the machine X^d_j in the current superstep before the submachine begins its computation.

d) When a submachine finishes its task, each processor determines its communication partner and then rejoins to the father group (lines 17-18 and 31-32 respectively). This operation is performed in constant time *C[0]*.

e) A communication bounds the superstep. Partial results are exchanged between partner processors (lines 21-22 and 35-36). Each processor has to wait only for a message from its partner. Under the assumption that the input signal size is a power of 2, the *h*-relation is the same for all the processors. We work with the *h*-relation definition as the sum of incoming and outgoing message sizes.

$$h_{1,i} = size = N * sizeof(Complex), \qquad W_{s,i} = \{i, partner_i\} \quad (\mathbf{7})$$

Therefore, the time for the first superstep is:

$$F_{1,i}(FFT, X^{d-1}, x) = max\{F_{2,k}(FFT, X^d, x_k + A[0]+B[0]) + C[0] \,/\, k \in W_{1,i}\} + D[1]* size + D[0] \quad (\mathbf{8})$$

The second superstep deals with the combination phase. It consists of two computational blocks and no communication is required.

a) In the first computation block (lines 25 and 39), the message received from the partner is retrieved from the communication library buffer to the process memory. This requires time *E[0]*.

b) The combination itself is performed by the call to routine *combine* in line 43. This computation takes time proportional to the signal size, that is *F[0]+F[1] * n*.

Thus, the formulas for the second superstep are:

$$W_{s,i} = \{i\} \quad (\mathbf{9})$$

$$F_{2,i}(FFT, X^{d-1}, x_{d-1,i}) = F_{1,i}(FFT, X^{d-1}, x_{d-1,i}) + E[0] + F[0] + F[1] * n$$

This recursive process follows until only one processor remains in the *BSP* submachine. These single-processor machines only perform one superstep. No communication is needed and the computations consist on calling to routine *seqDandCFFT* in line 49, which transforms a signal with size *N/P* using a sequential version of the same algorithm. The computational complexity is *O((N/P)*log(N/P))*, and is approximated by the linear expression*:*

$$F_{1,i}(FFT, S_{log(P)}, x_{log(P)}) = G[0] + G[1] * N/P * log(N/P) \qquad \textbf{(10)}$$

Since all processors start the computation at the same instant $x_{0,i} = 0$. Using successively formulas 7, 8 and 9, leads to the expression:

```
void parDandCFFT(Complex *A, Complex *a, Complex *W,
                 int N, int stride,t_bsp *gbsp) {
  .... /* variable declarations */
  if (bsp_nprocs(gbsp) > 1) {
    if(N == 1) { A[0].re = a[0].re;  A[0].im = a[0].im; }
    else {
#pragma cll A A[0]
      n = N / 2; size = n*sizeof(Complex); B = A;   C = A + n;
#pragma cll end A
#pragma cll B B[0]
      subgroup[1] = bsp_nprocs(gbsp);
      subgroup[0] = (bsp_nprocs(gbsp) / 2);
      bsp_partition(gbsp, &bsp_new, 2, subgroup);
#pragma cll end B
      if(bsp_pid(gbsp) < subgroup[0]) {
        parDandCFFT(B, a, W, n, stride*2, &bsp_new);
#pragma cll C C[0]
        partner = bsp_pid(&bsp_new) + subgroup[0];
        bsp_done(&bsp_new);
#pragma cll end C
#pragma cll D D[0]+D[1]*size
        bsp_hpsend(gbsp, partner, B, size);
        bsp_oblsync(gbsp, 1);
#pragma cll end D
#pragma cll E E[0]+E[1]*size
        C = (Complex*)bspmsg_data(bsp_getmsg(gbsp, 0));
#pragma cll end E
      }
      else {
        parDandCFFT(C, a+stride, W, n, stride*2, &bsp_new);
#pragma cll C C[0]
        partner = bsp_pid(&bsp_new);
        bsp_done(&bsp_new);
#pragma cll end C
#pragma cll D D[0]+D[1]*size
        bsp_hpsend(gbsp, partner, C, size);
        bsp_oblsync(gbsp, 1);
#pragma cll end D
#pragma cll E E[0]+E[1]*size
        B = (Complex*)bspmsg_data(bsp_getmsg(gbsp, 0));
#pragma cll end E
      }
#pragma cll F F[0]+F[1]*n
      combine(A,B,C,W,n);
#pragma cll end F
    }
  }
  else
#pragma cll G G[0]+G[1]*N*log(N)
    seqDandCFFT(A, a, W, N, stride);
#pragma cll end G
}
```

Fig. 4. Parallel Fast Fourier Transform

$$F_{2,i}(FFT, X, x) = log(P) * (A[0] + B[0] + C[0] + E[0]) + G[0] + G[1] * (N/P) * log(N/P) + log(P)* F[0] + F[1] * ((P-1)/P) * N + D[1]* ((P-1)/P) * size + log(P) * D[0] \qquad (11)$$

4 Results and Conclusions

Table 1 presents the results. The *OBSP** approach implies an improving in prediction accuracy for both computation and communication parts. The benefits are remarkable if they are compared with the errors obtained using raw BSP, where they can reach large values [15].

Table 1. Real and predicted times (sec.) for the FFT in the CRAY T3E (2 Megacomplex).

PROCS	TIME	OBSP*	ERROR %
1	11.7748	11.8096	-0.30
2	6.0036	5.8943	1.82
4	3.2120	3.0908	3.77
8	1.8939	1.7735	6.36
16	1.2750	1.1644	8.68
32	0.9664	0.8919	7.71

The *CALL* environment is currently under development. The values of the parameters and intervals of validity where manually computed . A first deliverable is expected to be publicly available by the end of 2001.

Acknowledgements

We would like to thank to Centro de Investigaciones Energéticas, Medioambientales y Tecnológicas (CIEMAT), like thus also to the National University of San Luis, CONICET, and ANPCYT, Argentina, from which continuous support is received. This research has been partially supported by Comisión Interministerial de Ciencia y Tecnología under project TIC1999-0754-C03, Spain.

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 Applications 14:3, Fall 2000, pp. 189-204.
3. Cooley, J. W. and Tukey, J. W.: *An algorithm for the machine calculation of complex Fourier series*. Mathematics of Computation, 19:90, 1965 pp. 297-301.
4. Espinosa A., Margalef, T., Luque E. *Automatic Performance Evaluation of Parallel Programs.* Proc. Of the 6th Euromicro Workshopon PDP. IEEE CS. 1998. 43-49.
5. Fahringer T, Zima H. *Static Parameter Based Performance Prediction Tool for Parallel Programs.* ICS. ACM Press. 1993. 207-219.
6. González, J.A., León, C., Piccoli, F., Pristinta, M., Roda, J.L., Rodríguez, C., Sande, F. *Performance Prediction of Oblivious BSP Programs.* EuroPar 2001. Springer-Verlag. 2001.
7. Goudreau, M. Hill, J., Lang, K. McColl, B., Rao, S., Stephanescu, D., Suel, T., Tsantilas, T. *A Proposal for the BSP Worldwide Standard Library.* http://www.bsp-worldwide.org/standard/stand2.htm. 1996.
8. Groom, D. E. et al. *Statistics.* The European Physical Journal C15 (2000). http://pdg.lbl.gov/2000/statrppbook.pdf
9. Heath M. Etheridge J. *Visualizing the Performance of Parallel Programs.* IEEE Software. 8 (5) September 1991. 29-39.
10. Hill J. McColl W. Stefanescu D. Goudreau M.. Lang K. Rao S. Suel T. Tsantilas T. Bisseling R. *BSPLib: The BSP Programming Library*. Parallel Computing. 24(14) pp. 1947-1980. 1988.
11. Labarta J., Girona S., Pillet V., Cortes T., Gregoris L. *Dip: A Parallel Program Development Environment.* Europar 96. Lyon. August. 1996.
12. Rodríguez C., Roda J.L., Morales D.G., Almeida F. *h-relation models for Current Standard Parallel Platforms*. EuroPar'98. Springer-Verlag. pp 234-243.
13. Valiant L.G. *A Bridging Model for Parallel Computation*. Communications of the ACM. 33(8). pp. 103-111. 1990.
14. Pallas. *Vampir 2.0. Visualization and Analysis of MPI Programs.* http://www.pallas.com.
15. Zavanella, A. Milazzo, A. *Predictability of Bulk Synchronous Programs Using MPI.* 8th Euromicro PDP pp. 118-123. 2000.