Scientific Programming 11 (2003) 191–198
IOS Press

A tool for performance modeling of parallel programs

J.A. González[a], C. Rodríguez[a,*], G. Rodríguez[a], F. de Sande[a] and M. Printista[b]
[a]*Dpto. Estadística, I.O. y Computación, Universidad de La Laguna, La Laguna, 38271, Spain*
Tel.: +34 922 318187; Fax: +34 922 318170; E-mail: casiano@ull.es
[b]*Universidad Nacional de San Luis, Ejército de los Andes 950, San Luis, Argentina*
E-mail: mprinti@unsl.edu.ar

Abstract. Current performance prediction analytical models try to characterize the performance behavior of actual machines through a small set of parameters. In practice, substantial deviations are observed. These differences are due to factors as memory hierarchies or network latency. A natural approach 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 of parameters must be done for each algorithm. This is a heavy task, implying experiment design, timing, statistics, pattern recognition and multi-parameter fitting algorithms. Software support is required. We present 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.

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 [14] is one of the most popular. Profiling may be conducted on a parallel system to recognize current performance bottlenecks. Performing measurements require special purpose hardware and software and, since the target machine is used, the measurement method can be highly accurate [4,6,8,11,12]. Although much work has been developed in Analytical Modeling and in Parallel Profiling, sometimes there seems to exist a divorce between them. Analytical modeling is considered to be too theoretical to be accurate in practical cases and profiling analysis is criticized for a lack of generality.

This work explores a hybrid approach, proposing an 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 efficient expression of some common parallel paradigms, like farms and pipelines, is difficult in the scope of a flat-data-parallel global-barrier Bulk Synchronous Programming software like the BSPlib [9]. To overcome these limitations, the Paderborn University BSP library (PUB [11]) offers the use of collective operations, processor-partition operations and oblivious synchronization. In addition to the most common features of BSP, 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, p. 18].

Another novel feature of PUB is the oblivious synchronization. It is implemented through the `bsp_oblsync(bsp,n)` function, which does not re-

*Corresponding author.

ISSN 1058-9244/03/$8.00

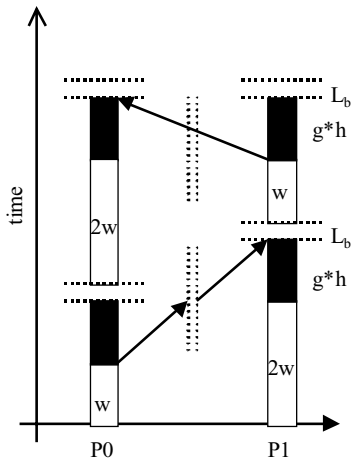

Fig. 1. BSP prediction accuracy.

turn 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.

Figure 1 illustrates the impact of the second improvement, oblivious synchronizations, in 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 ($2w$) than that performed by processor $P0$ (w). After an exchange operation (w) and an oblivious synchronization, the situation is inverted and processor $P0$ does the lighter part compensating the former imbalance. Finally, there is another oblivious exchange between processors $P0$ and $P1$ (w). While the actual time is $5w$, the BSP prediction corresponding to a global synchronous barrier is $6w$.

There are other sources of inaccuracy. 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, p. 123]. Another comes from characterizing the communication time through two single parameters g and L, considering that any h-relation takes the same amount of time, independently of the particular communication pattern involved [10]. In [13] we studied the impact of such patterns on the h-relation time.

A more realistic (but more difficult) 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. Still this approach does not suffice 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 of address translation low, the most recently used page addresses are cached in a translation lookaside buffer (TLB). While an $L1$ cache hit typically takes 2 or 3 cycles, a TLB miss requiring only reload of the TLB can take of the order of 2000 cycles [2, p. 3]. The assumption that a constant number of machine instructions takes constant time is an oversimplification. To suppress such simplification implies the introduction 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 out by hand such tasks for every program developed.

We 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 modeling, analysis and prediction of parallel and sequential programs. The tool consists of a "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.

2. The OBSP* Model

As in ordinary BSP, the execution of a PUB program on a BSP machine $X = 0, \ldots, P-1$ consists of super-

steps. 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.

Let us 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 $\Omega_{s,i}$ for a given processor i and superstep s as the set:

$$\Omega_{s,i} = \{j \in X \mid \text{processor } j \text{ sends a message to processor } i \text{ in superstep } s\} \cup \{i\} \quad (1)$$

while $\Omega_{s,i} = X$ when the superstep 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 $\xi = (\xi_0, \ldots, \xi_{P-1})$ 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 superstep s is recursively defined by the formulas:

$$\begin{aligned} &\Phi_{1,i}(T, X, \xi) \\ &= \max\{W_{1,j} + \xi_j \quad \mid j \in \Omega_{1,i}\} \\ &\quad +(g * h_{1,i} + L), \quad i = 0, \ldots, P-1, \\ &\quad \Phi_{s,i}(T, X, \xi) \\ &= \max\{\Phi_{s-1,j}(T, X, \xi) + W_{s,j} \quad \mid j \in \Omega_{s,i}\} \\ &\quad +(g * h_{s,i} + L), \\ &\quad s = 2, \ldots, R, i = 0, \ldots, P-1 \end{aligned} \quad (2)$$

where constant R denotes the total number of supersteps and $W_{s,j}$ 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:

$$\begin{aligned} &h_{s,i} = \max\{in_{s,j} \mathbin{@} out_{s,j} \quad \mid j \in \Omega_{s,i}\}, \\ &s = 1, \ldots, R, \ i = 0, \ldots, P-1 \end{aligned} \quad (3)$$

Table 1
Sets $\Upsilon_{A_i}, i = 0, 1, 2$

A_0	basically refers to the setup of the outer loop
A_1	corresponds to statements `j<N; j++` and `i=0`
A_2	stands for `i<N; i++` and `a[i][j] = 0`

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 $+$ depending on the input/output capabilities of the network interface.

2.1. Parameter evaluation

Notice that, in general, what the "algorithm designer" provides is a formula $f_{s,j}(N, M, \ldots)$ that gives the total number of operations performed by processor j in superstep s in terms of the input parameters $N, M, \ldots$.

As an example, the analysis of the sequential code in Fig. 3, give us $f_{1,0}(N) = A_0 + A_1 \times N + A_2 \times N^2$.

What is the meaning of the constants $A_0, A_1, \ldots A_n$?. Assuming that the processors have an instruction set $\{I_1, \ldots, I_t\}$ of size t, where the i-th instruction I_i takes time p_i, an approximation of the time $W_{s,j}$ is given by the formula:

$$W_{s,j} \simeq \sum_{i=1,t} w_{s,i,j} * p_i$$

where $w_{s,i,j}$ is the number of I_i instructions executed by processor j in step s.

The actual situation is more complex than this, since the time p_i is a random variable that takes a small number of different values. If, for instance, I_i is a memory access, we may have two or three different values of p_i according to the number of misses and hits. The same statement applies for communication constants g and L.

Each constant A_i is associated with the cost of a set Υ_{A_i} of statements (see Table 1). Of the three "constants" in the example of Fig. 3, only A_2 manifestly "varies" with the size of the input N. The other two do not change so noticeably, since their instructions are related with scalar accesses and only exploit temporal locality. Therefore, we will have that the formula $f_{1,0}(N)$ predicts the behavior with acceptable accuracy if we use two or three different values for A_2: one corresponding to small values of N (large percentage of cache hits) and the others to larger values of N.

The idea proposed here is to have a tool that, having the formula $f_{1,0}(N) = A_0 + A_1 \times N + A_2 \times N^2$ as input, automatically finds the sets $\aleph_{A_i}$ of different

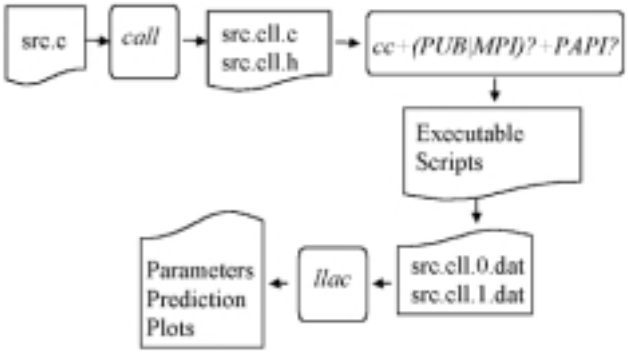

Fig. 2. The CALL environment. The ? stand for optionality.

```
1.  for(j=0;j<N;j++)
2.    for(i=0;i<N;i++)
3.      a[i][j] = 0;
```

Fig. 3. Matrix initialization.

Table 2
Sets $\aleph_{A_i}$

$\aleph_{A_0} = \{(A_0, N \in [2, \infty)\}$
$\aleph_{A_1} = \{(A_1, N \in [2, \infty)\}$
$\aleph_{A_2} = \{(A_2^0, N \in [2, C_0]), (A_2^1, N \in (C_0, \infty])\}$

values and intervals of the input variable N where each of these values apply (see Table 2).

Once these sets $\aleph_{A_i}$ have been evaluated, they can be used for prediction.

The tool will use a "machine database" that extends the well-known BSP table with entries (s, g, L). This database contains a vectorial detailed description of the architecture, including the costs of different memory access times, the costs of different floating point operations, etc. This database can be generated by an "architecture analyzer" program working much in the same way as the `bsp_probe` program included with the BSPlib library.

From this database and the knowledge of the sets Υ_{A_i} the tool, through the analysis of the statements, can guess the values of the sets $\aleph_{A_i}$ in the architectures included in the database.

Observe that the formula $f_{1,0}(N) = A_0 + A_1 \times N + A_2 \times N^2$ will still be valid if we exchange the loops in lines 1 and 2 in Fig. 3. But, since the current access to `a[i][j]` at line 3 will change its "stride" from N to 1, the gain in locality will produce much lower values for A_2 and a higher value of C_0. The number of operations in both algorithms are the same, the instructions involved are the same, but the two constants A_2 are significantly different.

This example illustrates that, to predict the performance in other architectures it is not enough to have the knowledge of Υ_{A_i} and the database: the instrumentation also has to collect run time information about the percentage of cache misses.

Locality and the order of memory accesses affects the values of the sets $\aleph_{A_i}$. Roughly speaking, and as a previous step, the programmer has to be aware of choosing among the several semantically equivalent orders, one that minimizes the A_i parameters with strongest impact on the "complexity" formula.

2.2. Processor sets in PUB

At any time, processors are organized in a hierarchy of processor sets. A processor set in PUB (also called a *BSP* object) is implemented through a data structure named `t_bsp`.

Let $Q \subseteq X$ 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,

$$Q = \cup_{0 \leqslant i \leqslant r-1}\ S_i,$$
$$S_0 = \{0, \ldots, Y[0] - 1\},$$
$$S_i = \{Y[i-1], \ldots, Y[i] - 1\},$$
$$1 \leqslant i \leqslant r - 1$$

After the partition step, each subgroup S_i acts as an autonomous BSP computer with its own processor numbering, message 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

$$\Phi_{R_i,j}(T_i, S_i, \Phi_{s-1,j} + w^*_{s,j})$$
$$\text{such that } j \in S_i,$$
$$i = 0, \ldots, r-1,$$

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. Functions `bsp_partition` and `bsp_done` produce no communication. This implies that different processors in a given subset can arrive at the partition process (and leave it) at a 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, 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]. Figure 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 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

```
1. #pragma cll parallel PUB gbsp\
    procs = 1:32:2
   ...
2. #pragma cll for(N=1024; N<262144; N*=2)
3. initialize(N, a);
4. Roots(N/2, W);
5. #pragma cll sync f f[0]+f[1]*log(P)+\
    f[2]*(N/P)*log(N/P)+f[3]*N*(P-1)/P
6. parDandCFFT(A, a, W, N, 1, D, gbsp);
7. #pragma cll end f
8. #pragma cll end for
   ...
9. #pragma cll report all
```

Fig. 4. The fft experiment.

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 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 $O(N \times log(N))$. 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 two. Line 1 in Fig. 4 warns the `call` compiler that this is a BSP parallel program using PUB. The optional argument `gbsp` points to the `t_bsp` object describing the BSP machine. This information will be used by the report clause in line 9. 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.0.dat`. The second clause `procs = 1:32:2` makes the compiler generate a batch script to run the program for different numbers of processors.

Lines 2 and 8 produce a loop to sample the algorithm behavior for different values of N. Since CALL pragmas are identified by a defined prefix, they are ignored by a C compiler. One of the goals of CALL is to allow developers to use the same source code base for building their application and the instrumented code.

Lines 5 to 7 in Fig. 4 define a "`call` experiment". The optional clause `sync` at the beginning of the experiment definition (line 5) indicates the need to synchronize the processors before starting the experiment.

```
1.void parDandCFFT(Complex *A, Complex *a, Complex *W,int N,int stride,t_bsp *gbsp) {
2. .... /* variable declarations */
3. if (bsp_nprocs(gbsp) > 1) {
4.  if (N==1) {A[0].re = a[0].re; A[0].im = a[0].im;}
5.  else {
6.  #pragma cll A A[0]
7.    n = N/2; size = n * sizeof(Complex); B = A; C = A + n;
8.  #pragma cll end A
9.  #pragma cll B B[0]
10.   subgroup[1] = bsp_nprocs(gbsp);
11.   subgroup[0] = (bsp_nprocs(gbsp)/2);
12.   bsp_partition(gbsp, &bsp_new, 2, subgroup);
13. #pragma cll end B
14.   if(bsp_pid(gbsp)<subgroup[0]) {
15.     parDandCFFT(B, a, W, n, stride * 2, &bsp_new);
16. #pragma cll C C[0]
17.     partner = bsp_pid(&bsp_new) + subgroup[0];
18.     bsp_done(&bsp_new);
19. #pragma cll end C
20. #pragma cll D D[0] + D[1] * size
21.     bsp_hpsend(gbsp, partner, B, size);
22.     bsp_oblsync(gbsp, 1);
23. #pragma cll end D
24. #pragma cll E E[0] + E[1] * n
25.     C = (Complex *)bspmsg_data(bsp_getmsg(gbsp, 0));
26. #pragma cll end E
27.   } else {
28.    parDandCFFT(C, a + stride, W, n, stride * 2, &bsp_new);
29. #pragma cll C C[0]
30.    partner = bsp_pid(&bsp_new);
31.    bsp_done(&bsp_new);
32. #pragma cll end C
33. #pragma cll D D[0] + D[1] * size
34.    bsp_hpsend(gbsp, partner, C, size);
35.    bsp_oblsync(gbsp, 1);
36. #pragma  ll end D
37. #pragma cll E E[0] + E[1] * n
38.    B = (Complex *)bspmsg_data(bsp_getmsg(gbsp, 0));
39. #pragma cll end E
40.   }
41. #pragma cll F F[0] + F[1] * n
42.   combine(A, B, C, W, n);
43. #pragma cll end F
44.  }
45. } else
46. #pragma cll G G[0] + G[1] * N * log(N)
47.   seqDandCFFT(A, a, W, N, stride);
48. #pragma cll end G
49. }
```

Fig. 5. Parallel Fast Fourier Transform.

The CALL compiler will insert a `barrier` in the generated code. The following identifier is the name of the experiment. Thus, the name of the experiment defined in line 5 is `f`. Then follows the complexity formula for $\Phi_{2,i}(FFT, X, 0)$. 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 experimental constant must be the only multiplicative constant in each term. This constraint is due to singularities appearing in the multidimensional fit algorithm [5] used by the interpreter.

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 $\aleph_{f[0]}$, $\aleph_{f[1]}$, $\aleph_{f[2]}$ and $\aleph_{f[3]}$. For this example, the input variables are N and P. Generally speaking, usually there will be values of $f[0]$, $f[1]$, $f[2]$ and $f[3]$ for small values of N and P, different values for medium sizes and may be a third for larger values. When predicting the time for a concrete value (say $N = 1024$, $P = 32$) the programmer does not need

Table 3
Real and predicted times for the FFT on the CRAY T3E (2 Mega-Complex)

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

to be concerned with the exact parameter values. The `llac` system will choose the appropriate parameter values of $f[0]$, $f[1]$, $f[2]$ and $f[3]$ (the one for the small range of N and $P = 32$ for the example) to obtain a more precise prediction. However, a large number of intervals (more than 3) on a given variable (N, P, ...) may possibly imply an error in the complexity formula. In such case, `llac` issues a warning message.

The recognition of the intervals of $\aleph_{f[i]}$ of the parameters $f[i]$ 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 exceed a "number of intervals threshold".

The user can participate, influencing any of the phases, including the preprocessing. Although most of the information can be reported through CALL pragmas, the `llac` user can complement and include any additional information. For example, to specify through a graphical interface the sets of statements Υ_{A_i} associated with the experiment parameters A_i.

The code in Fig. 5 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. 5. 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 $i \in X^{d-1}$ to perform the first superstep, $\Phi_{1,i}(FFT, X^{d-1}, \xi^{d-1})$, consists of four computational blocks and one communication:

1. Input signal division into its even and odd components (line 7). Since 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 constant time $A[0]$.
2. The BSP machine X^{d-1} is partitioned in two submachines X_j^d 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 constant time $B[0]$.
3. 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 28 respectively. The times required by each of these submachines to perform their computations are given by $\Phi_{2,i}(FFT, X_j^d, \xi_i^{d-1} + A[0]+B[0])$. Here d is the recursion depth, X_j^d is the set of processors in the current BSP machine, ξ_i^{d-1} is the time when the calling FFT started and $w_{1,i}^* = A[0] + B[0]$ denotes the computation performed by the machine X_j^d in the current superstep before the submachine begins its computation.
4. When a submachine finishes its task, each processor determines its communication partner and then rejoins to the father group (lines 17–18 and 30–31 respectively). This operation is performed in constant time $C[0]$.
5. A communication bounds the superstep. Partial results are exchanged between partner processors (lines 21–22 and 34–35). 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.

$$\begin{aligned} &h_{1,i} = \text{size} = N * \text{sizeof(Complex)}, \\ &\Omega_{s,i} = \{i, partner_i\} \end{aligned} \tag{4}$$

Therefore, the time for the first superstep is:

$$\begin{aligned} &\Phi_{1,i}(FFT, X^{d-1}, \xi^{d-1}) \\ &= \max\{\Phi_{2,k}(FFT, X^d, \xi_k^{d-1} \\ &\quad + A[0] + B[0]) + C[0] \\ &\quad \mid k \in \Omega_{1,i}\} + D[1] * size + D[0]\} \end{aligned} \tag{5}$$

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

– In the first block (lines 25 and 38), the message received from the partner is retrieved from the communication library buffer to the process memory. This requires time $E[0] + E[1] \times n$.
– The combination itself is performed by the call to routine `combine` in line 42. This computation takes time proportional to the signal size, that is $F[0] + F[1] \times n$.

Thus, the formulas for the second superstep are:

$$\begin{aligned} &\Omega_{s,i} = \{i\} \\ &\Phi_{2,i}(FFT, X^{d-1}, \xi_i^{d-1}) \\ &= \Phi_{1,i}(FFT, X^{d-1}, \xi_i^{d-1}) \\ &\quad + E[0] + F[0] + (E[1] + F[1])n \end{aligned} \tag{6}$$

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 of the call to $seqDandCFFT$ in line 47, which transforms a signal with size N/P using a sequential version of the same algorithm. The computational complexity is $O(\frac{N}{P} \log \frac{N}{P})$, and is approximated by the linear expression:

$$\begin{aligned} &\Phi_{1,i}(FFT, S^0, \xi^0) \\ &= G[0] + G[1]\frac{N}{P} \log\left(\frac{N}{P}\right) \end{aligned} \tag{7}$$

Since all processors start the computation at the same instant $\xi_i^{\log(P)} = 0$. Using successively Eqs (5), (6) and (7), leads to expression:

$$\begin{aligned} &\Phi_{2,i}(FFT, X, \xi) \\ &= \log(P)(A[0] + B[0] + C[0] + E[0]) \\ &\quad + G[0] + G[1](N/P)\log(N/P) \\ &\quad + \log(P)F[0] + F[1]((P-1)/P) * N \\ &\quad + D[1]((P-1)/P)\,\text{size} + \log(P)D[0] \end{aligned} \tag{8}$$

4. Results

Table 3 presents the results for a 2 mega complex FFT. The sizes used to obtain the $\aleph_{f[i]}$ sets are the ones shown in line 2 of Fig. 4. The curious decreasing observed when going from 16 to 32 processors, is likely due to the addition of two compensating errors, that is, an over-estimation of one term and the under-estimation of another.

Acknowledgments

We would like to thank 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.

References

[1] O. Bonorden, B. Juurlink, I. von Otte and I. Rieping, The paderborn university BSP (PUB) library-design, implementation and performance, in *International Parallel Processing Symposium & Symposium on Parallel and Distributed Processing* (*IPPS/SPDP*), 1999.
[2] S. Browne, J. Dongarra, N. Garner, G. Ho and P. Mucci, A portable programming interface for performance evaluation on modern processors, *The International Journal of High Performance Computing Applications* (2000), 189–204.
[3] J.W. Cooley and J.W. Tukey, An algorithm for the machine calculation of complex fourier series, *Mathematics of Computation* **19** (1965), 297–301.
[4] A. Espinosa, T. Margalef and E. Luque, Automatic performance evaluation of parallel programs, in *Proc. Of the 6th Euromicro Workshop on PDP,* IEEE CS, 1998, pp. 43–49.
[5] D.E. Groom et al., Statistics, *The European Physical Journal* **C15** (2000), http://pdg.lbl.gov/2000/statrppbook.pdf.
[6] T. Fahringer and H. Zima, Static parameter based performance prediction tool for parallel programs, in: *International Conference on Supercomputing*, ACM Press, 1993, pp. 207–219.
[7] M. Goudreau, J. Hill, K. Lang, B. McColl, S. Rao, D. Stephanescu, T. Suel and T.A. Tsantilas, *Proposal for the BSP worldwide standard library,* http://www.bsp-worldwide.org/standard/stand2.htm, 1996.
[8] M. Heath and J. Etheridge, Visualizing the performance of parallel programs, *IEEE Software* **8**(5) (1991), 29–39.
[9] J. Hill, W. McColl, D. Stefanescu, M. Goudreau, K. Lang, S. Rao, T. Suel, T.A. Tsantilas and R.H. Bisseling, BSPlib: The BSP programming library, *Parallel Computing* **24**(14) (1998), 1947–1980.
[10] B.H.H. Juurlink and A.G. Wijshoff, The E-BSP model: Incorporating general locality and unbalanced communication into the BSP model, in *International Euro-Par'96 Conference,* (Vol. II), Springer-Verlag, 1996, pp. 339–347.
[11] J. Labarta, S. Girona, V. Pillet, T. Cortes and L. Gregoris, Dip: A parallel program development environment, in: *Euro-Par,* (Vol. II), 1996, pp. 665–674.
[12] W.E. Nagel, A. Arnold, M. Weber, H.C. Hoppe and K. Solchenbach, VAMPIR: Visualization and analysis of MPI resources, *Supercomputer* **12**(1) (1996), 69–80.
[13] C. Rodriguez, J.L. Roda, D.G. Morales and F. Almeida, h-relation models for current standard parallel platforms, in *4th International Euro-Par Conference,* Springer-Verlag, 1998, pp. 234–243.
[14] L.G. Valiant, A bridging model for parallel computation, *Communications of the ACM* **33**(8) (1990), 103–111.
[15] A. Zavanella and A. Milazzo, Predictability of bulk synchronous programs using mpi, in *8th Euromicro PDP,* 2000.