A NEW MODEL FOR THE ANALYSIS OF ASYNCHRONOUS PARALLEL ALGORITHMS

Roda J.L, Rodríguez C., Sande F., Morales D.G., Almeida F.

Dpto. Estadística, Investigación Operativa y Computación,
Universidad de La Laguna, Tenerife, Spain
jlroda@ull.es

Abstract. The asynchronous nature of some MPI/PVM programs does not easily fit inside the BSP model. Through the suppression of barriers and the generalization of the concept of superstep we propose a new model, the BSP Without Barriers (BSPWB) model. The BSPWB parameters and their quality are evaluated on four standard parallel platforms: the CRAY T3E, the IBM SP2, the Origin 2000 and the Digital Alpha Server 8400. This study proves that the time spent in an *h*-relation is more independent on the number of processors than on the communication pattern. We illustrate the BSPWB model using two problem-solving paradigms: the Nested Parallel Recursive Divide and Conquer Paradigm and the Virtual Pipeline Dynamic Programming Paradigm. The proposed paradigms explain how nested parallelism and processor virtualization may be introduced in MPI and PVM without having any negative impact in the performance and model accuracy. The prediction of the communication times is robust even for problems where communication is dominated by small messages.

1. Introduction

The **BSP Without Barriers** model is proposed in the next section. In section 3 we estimate the BSPWB gaps and latencies in 4 representative platforms: an IBM-SP2, a Silicon ORIGIN 2000, a Digital Alphaserver 8400 and a Cray T3E using MPI. The use of the BSPWB model will be illustrated later in section 4. Subsection 4.1 concentrates in a Nested Parallel Recursive Divide and Conquer Algorithm. The model is applied in subsection 4.2 to a Virtual Pipeline Multistage Dynamic Programming Algorithm. The two examples cover two opposite communication scenarios: while the first one is coarse grain, the second corresponds to a fine-grain intensive communication situation. The computational resources prove that nested parallelism and processor virtualization can be introduced without negative impact in the performance and model accuracy.

2. BREAKING BARRIERS: The BSPWB model.

The execution of a PVM or MPI program in any processor consists in phases of computation followed by the communications necessary to provide and obtain the data for the next phase. Communication in this context means a *continuous stream* of messages. We propose a generalization of the BSP concept of superstep [5] to MPI/PVM programming that we call Message steps or "M-steps". In any M-step each processor performs some local computation, sends the data needed by the other processors and receives the data it needs for the next M-step. Processors may be in different M-steps at a given time, since no global barrier synchronization is used. The absence of barriers is the main difference with the BSP programming style. However, we assume that:

- The total number of M-steps R, performed by all the p processors is the same.
- Communications always occur among processors in the same step s.

To achieve these two goals, the designer can arbitrarily divide the computation stages in “dummy M-steps” at any processor.

An approach to the actual MPI/PVM time $t_{s,i}$ when processor i finishes its s-th M-step is the value $\Phi_{s,i}$ given by the **BSP Without Barriers** (**BSPWB**) model we propose here. We define the set $\Omega_{s,i}$ for a given processor i and M-step s as the set

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

Processors in the set $\Omega_{s,i}$ are called "**the incoming partners of processor *i* in step *s***". The **BSP Without Barriers** (**BSPWB**) time $\Phi_{s,i}$ of a MPI/PVM program is recursively defined by the formulas:

$$\Phi_{1i} = max\,\{\, w_{1,j} \,/\, j \in \Omega_{1,i} \} + (g\, h_{1,i} + L),\;\; i = 0,1,..., p-1$$
$$\Phi_{s,i} = max\{\Phi_{s-1,j}+w_{s,j} \,/\, j \in \Omega_{s,i}\} + (g\, h_{s,i} + L),\; s=2,..,R,\;\; i=0,1,..., p-1 \qquad \textbf{(2)}$$

where $w_{s,i}$ denote the time spent in computing and $h_{s,i}$ denotes the number of packets communicated by processor i in step s:

$$h_{s,i} = max\,\{in_{s,j} \,@\, out_{s,j} \,/\, j \in \Omega_{s,i}\},\;\; s = 1,...,R,\;\; i = 0,1,..., p-1 \qquad \textbf{(3)}$$

and $in_{s,i}$ and $out_{s,i}$ respectively denote the number of packets incoming/outgoing to/from processor i in the M-step s. The @ operation is defined as *max* or the *sum* depending on the input/output capabilities of the architecture. Gap and Latency values g and L can be computed as proposed in the next paragraph.

The formula says that processor i in its step s can not start the reception of its messages after it has finished its computing time $\Phi_{s-1,i} + w_{si}$ and so have done the other processors j sending messages to processor i. The formula charges the communication time of processor i with the maximum communication time of any of its incoming partner processors. The model assumes either that buffering capacities are large enough or that processors have the capacity of doing send and receives in parallel (@ = *max*).

Another issue to discuss is what is the appropriate unit size of a packet. This size depends on the target architecture. Our experiments prove that, more than a linear behavior, the communication time conducts according to a linear by pieces function on the h-relation size. There is always a slope for small size messages different from the slope for large messages. We define the **BSPWB *packet size*** as the size in which the time/size curve of the architecture has the first inflection point. Special gap g_0 and latency L_0 values have to be used for messages of sizes smaller than the BSPWB packet size. The total time of a MPI/PVM program in the BSPWB model is given by

$$\Psi = max \{ \Phi_{R,j} / j \in \{0,...,p\text{-}1\}\} \tag{4}$$

where R is the total number of M-steps. Instead of bounding the time as if there were a hypothetical global barrier synchronization at the end of the M-step, the BSPWB model implies a synchronization of the processor with its incoming partners.

3. Estimation of the BSPWB parameters.

Some authors [1] estimate the BSP values of g and L by generating random h-relations. However, in every day algorithm practice, not all the h-relations have the same probability. There are certain communication patterns that occur frequently. Instead of random h-relations we decided to select for our study a set $\Pi = \{E, PP, OA, AO, AA\}$ of 5 communication patterns among those that most commonly appear in parallel algorithms: *Exchange (E), PingPong (PP), OneToAll (OA), AllToOne (AO) and AllToAll (AA).* In an Exchange pattern, E, a certain number p of processors pairs simultaneously send messages of length m_E and immediately proceeds to receive the messages. In the more asymmetric PingPong pattern, PP, one of the processor in the pair sends a message of size m_{PP} and the other receives the message. In the OneToAll, OA, a processor sends a message of size m_{OA} to each of the other processors. Reciprocally, in the AllToOne pattern, AO, an incoming processor receives $p\text{-}1$ messages of size m_{AO} sent by the other processors. Under the AllToAll pattern, AA, each one of the p processors sends their message of size m_{AA} to all the other $p\text{-}1$ processors. Under the @ = + assumption, all these 5 patterns give place to the same h-relation if the following sequence of equalities hold:

$$h = m_{PP} = 2 * m_E = (p\text{-}1) * m_{OA} = (p\text{-}1) * m_{AO} = 2*(p\text{-}1)*m_{AA} \tag{5}$$

According to the h-relation hypothesis the communication time of these patterns for these messages sizes have to be the same. Our experiments prove that for any architecture, any pattern $\rho \in \Pi = \{E, PP, OA, AO, AA\}$ and any number i of processors, the time $Time_{\rho,i}(h)$ spent in an h-relation, for values of h larger enough than the **BSPWB *packet size***, can be approached by a linear function:

$$Time_{\rho,i}(h) = g_\rho(i)*h+L_\rho \tag{6}$$

Figure 1 shows the variation of $g_\rho(i)$ expressed in sec/word. Observe that, except for the Digital 8400, the h-relation time of the other three machines is remarkably independent of the number of processors. However, the pattern influence in the h-relation time goes from a factor of 2 for the IBM SP2 to a factor of 4 in the less scalable machine: the Digital 8400. The slowest pattern in all the machines is the PingPong, since all the h bytes are sequentially sent through the output port. The other patterns benefit from the existence of different ports for input and output. It is remarkable the improvement of the OneToAll pattern with the number of processors. The authors in previous work have explained this phenomenon [3].

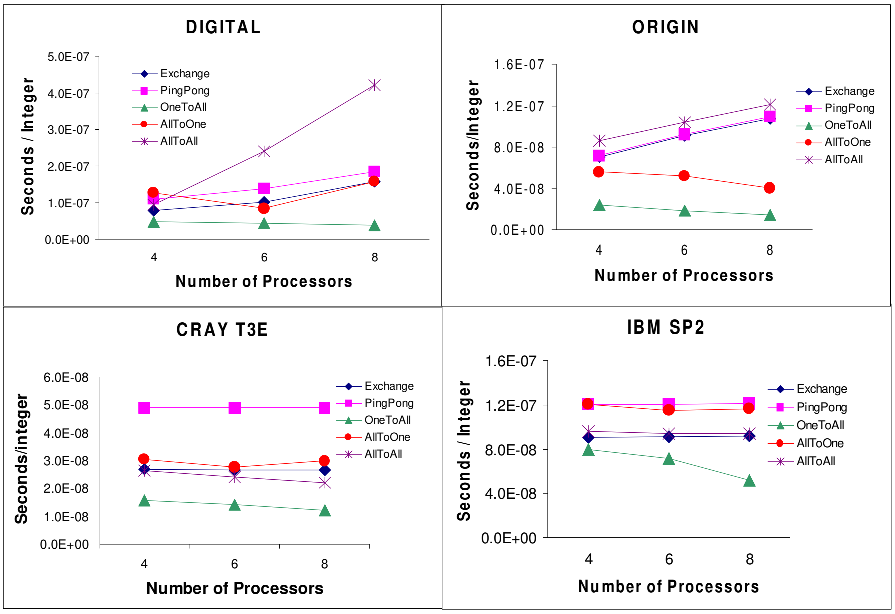

Fig. 1. Variation of the gap $g_{\rho,i}$ with the patterns and processors.

The **BSP Without Barriers** values of g and L at table 1 have been computed using linear fit on the average times $T_{average}$ (h) of the times $Time_{\rho,\,i}(h)$ obtained for $\rho \in \Pi = \{E, PP, OA, AO, AA\}$ and for the different number of processors $i \in H_\rho$:

$$T_{average}(h) = \sum_{\rho \in \Pi}\left(\sum_{i \in H_\rho} Time_{\rho,\,i}(h) \,/\, |H_\rho|\right) / \,|\Pi| \qquad (7)$$

where $H_\rho = \{2, 4, 6, 8\}$ for the Exchange and PingPong patterns, and $H_\rho = \{4, 6, 8\}$ for the OneToAll, AllToOne and AllToAll patterns. To produce the same h-relation size, message sizes m_ρ were chosen according to formula *(5)*. In order to compare the 4 architectures we have preferred to express h-relation sizes in words. The negative values of L are due to the unsuitability of taking one word as unit packet size. Row labeled L_0 shows the latency value for *1* word message.

Table 1. Values of L_0, L and g for all the architectures.

	CRAY T3E	DIGITAL	IBM SP2	ORIGIN
L_0	0.00001	0.0000025	0.000025	0.0000085
L	3.59E-05	4.24E-03	4.51E-04	-3.20E-03
g	2.75E-08	1.28E-07	9.79E-08	6.65E-08

No doubt, it is interesting to have some data to compare current MPI releases with current BSP libraries performance. The BSP value of g for the IBM SP2 using the

Oxford BSP library in the same installation at C^4 is g = *35E-8 sec/word* [2]. Since the Oxford definition of the *h*-relation is based in the operator @ = *max* it has to be divided by two, g = *17.5E-8 sec/word*. Compare this value with the MPI value of g = *9.79E-8 sec/word* in the table. The MPI is *1.8* times faster than the BSP Oxford Library.

4. Predicting the time of MPI/PVM programs with the BSPWB model.

To simplify the expression of the BSPWB M-step partition associated with a MPI program, we superimpose commented code to the example codes in sections 4.1 and 4.2. When uncommented, variable *M_step* contains at any time the value of the current BSPWB M-step. This M-step variable may be used as an additional tag when communicating messages, to check the validity of the BSPWB program. The use of this tag solves the well-known library-user tag conflict appearing in PVM that gave rise to the concept of communicator in MPI.

4.1 A Divide and Conquer using Nested Parallelism: The Sorting Example.

The *qs* function in code 1 is a natural parallel version of the classical quicksort algorithm solving the *AllToAll Sorting* of an array *A* of size *n*. We said that a problem is *AllToAll* if the initial input data is replicated in all the processors and the solution is also required in all the processors. The code is written using La Laguna C, a set of libraries and macros extending PVM and MPI with nested parallelism capacities.

```
void qs(int first, int last) {
  int s1, s2, i, j;
  if (NUMPROCESSORS > 1)
    if (first < last) {
      partition(&i, &j, &s1, &s2, first, last); /*A[k] ≤ A[q]  k≤ j< i ≤ q */
      PAR(qs(first, j), A + first, s1, qs(i, last), A + i, s2); /* M_step++ */
    }
    else quicksortseq(first, last);
}
```

Code 1. Parallel quicksort using the macro PAR.

Procedure partition spends *B n* time in dividing the array *A* in two intervals. When no processors are available the array is sorted in *C (n/p log(n/p))* time using the call to *quicksortseq(first, last)*. The macro *PAR* divides in constant time the set of processors in two subsets of equal size and the variable *NUMPROCESSORS* is updated. At the end of the two parallel calls, the two split subsets swap the sorted arrays giving place to an *h*-relation with $h = s1 + s2 = (last\text{-}first+1)$ integers. After the exchange, they join together to form the original set. Thus, the exchange communication inside the

PAR macro creates a new M-step. There are *log(p)* of such M-steps. The time of the first step by any processor *NAME* is divided in the *log(p)* calls to *partition* plus the time spent sorting its chunk plus the time of the first exchange.

$$T_1 = \Phi_{1,NAME} = \Sigma_{i=0,\ log(p)-1} B\ n/2^i + C(n/p)\ log(n/p) + g\ 2\ n/p + L \quad \mathbf{(8)}$$

Only exchange operations take place after this first M-step:

$$T_s = \Phi_{s,NAME} = T_{s-1} + g(2^{s-1}\ n/p) + L \quad \text{for } s = 2,\ldots,log(p) \quad \mathbf{(9)}$$

Substituting recursively, the total time is:

$$T_{log(p)} = \Phi_{log(p),NAME} = \Sigma_{i=0,\ log(p)-1} B\ n/2^i + C(n/p)log(n/p) + \Sigma_{s=1,\ log(p)-1}\ g(2^s n/p) + L \quad \mathbf{(10)}$$

The values of the computational constants *B* and *C* can be found in [4]. The model and actual time curves in figure 2 are so overlapped than can not be differentiated. The exception is the Digital 8400, which is the only architecture that does not scale.

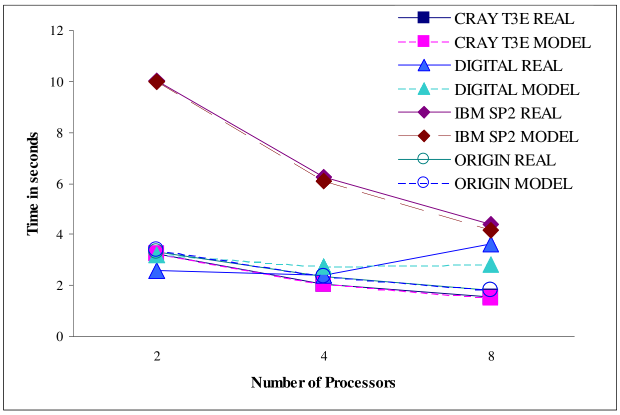

Fig. 2. Quicksort (7M integers) actual and model times.

4.2 A Pipeline Example: The Single Resource Allocation Problem.

On the opposite side to the former example, this section presents a fine-grain intensive-communication application. The function in code 2 uses *La Laguna Pipeline*, a set of macros and functions extending PVM and MPI. *La Laguna Pipeline* provides the programmer with a virtual pipeline whose size can be dynamically fixed by the user at any time. *La Laguna Pipeline* applies a cyclic mapping of the virtual processors in the actual machine. The code solves the Single Resource Allocation Problem (SRAP). The SRAP asks to map limited resources to a set of tasks to maximize their effectiveness. Assume that we have *M* units of an indivisible resource and *N* tasks. For each task *j*, function $f_j(x)$ gives the income obtained when a quantity *x* of resource is allocated to the task *j*. Let us denote by *Q[k][m]* the optimal income for the resource allocation subproblem constituted by the first *k* tasks and *m* units of resource. Applying the Dynamic Programming Principle to this problem leads to the following state equation:

$$Q[k][m]=max\ \{\ Q[k-1][m-x]+ f_k(x)\ /\ 0 \leq x\ \leq m\}\ \};\ Q[1][m]=f_1(m),\ m=0,...,M \qquad \textbf{(11)}$$

This formula leads to a very simple sequential algorithm with time $O(N\,M^2)$.

The pipeline algorithm in code 2 uses $P = N+1$ processors. It assigns to virtual processor *NAME* the computation of the optimal values *Q[NAME][m]* for all the resources *m* between *0* and *M*. The BSPWB requirement that *communications always occur among processors in the same step s* implies that processor *NAME* has to execute *NAME-1* dummy empty steps before the first reception. Thus, the last processor *N* performs *N-1* dummy steps plus the steps inside the loop in lines 4-10. Each iteration of the loop is divided in two M-steps by the two communications at lines 5 and 7. There are $R = N\text{-}1+ 2\ (M+1) = N + 2\ M+1$ M-steps in this algorithm. For any step *s* and each processor $NAME\ > 0$ it holds that

$$\begin{aligned} &\Omega_{s,NAME} = \{NAME\},\ h_{s,NAME}\ = 0, \text{ for } s < NAME\ , \\ &\Omega_{s,NAME}=\{NAME\},\ h_{s,NAME}=1, \text{ for } s = NAME+2m+1,\ m=0..M \\ &\Omega_{s,NAME}=\{NAME\text{-}1,NAME\},\ h_{s,NAME}=1, \text{ for } s = NAME+2m,\ m= 0..M \end{aligned} \qquad \textbf{(12)}$$

Since the BSPWB packet size of any of the considered machines is larger than the size of an integer, instead of using *L* and *g* we have to use the corresponding values L_0 and g_0 for small messages.

We have $\Phi_{s,NAME} =\ A\ (M+1)$ for all $s < NAME$, where *A* is the constant associated with the loop initializing *Q* in line 3. The first not dummy step takes place at M-step *NAME* when the first reception from the left neighbor occurs. According to (2):

$$\begin{aligned} \Phi_{NAME,\ NAME} &= max\ \{\ \Phi_{NAME\text{-}1,\ NAME\text{-}1} + w_{NAME,NAME\text{-}1},\ \Phi_{NAME\text{-}1,\ NAME} + w_{NAME,NAME}\ \} +\ L_0 \\ &= \Phi_{NAME\text{-}1,\ NAME\text{-}1} + w_{NAME,NAME\text{-}1} + L_0 = \Phi_{NAME\text{-}1,\ NAME\text{-}1} + B + L_0 \end{aligned} \qquad \textbf{(13)}$$

where *B* is the constant time spent in line 6. Substituting recursively, processor *NAME* starts its first non-dummy M-step *N*AME at time

$$\Phi_{NAME\ NAME}=(NAME\text{-}1)(B+L_0)+\Phi_{1,1} = (NAME\text{-}1)\ B+ NAME\ L_0+ A\ (M+1). \qquad \textbf{(14)}$$

From this M-step on:

```
 1 void transition ()  {
 2    int m, j, QkMinus1_m;
 3    for( m = 0; m <= M; m++) Q[m] = 0;
      /* for (M_step = 2; M_step < NAME; M_step++); */
 4    for (m = 0; m <= M; m++) {        /* Ω_{M_step,NAME} = { NAME-1, NAME} */
 5      IN_S(&QkMinus1_m);             /* M_step ++ ;  Ω_{M_step,NAME} = { NAME} */
 6      Q[m] = max(Q[m], QkMinus1_m + f(NAME, 0));
 7      OUT(&Q[m], 1, sizeof(int));     /* M_step++  */
 8      for (j = m + 1; j <= M; j++)
 9        Q[j] = max(Q[j],  QkMinus1_m + f(NAME , j - m));
10    }    /* for (M_step = 1; M_step<M+N; M_step++);*/
11  } /* transition */
```

Code 2. Simple Pipeline Algorithm for the SRAP. Code for processor NAME.

$$\Phi_{s,NAME} = \Phi_{s\text{-}1,\,NAME} + B + L_0 \text{ for } s = NAME{+}2\,m{+}1,\; m = 0..M \qquad \textbf{(15)}$$

and for $s = NAME+2\,m,\; m = 1..\,M$ we have

$$\Phi_{s,NAME} = max\{\Phi_{s\text{-}1,\,NAME\text{-}1} + B,\; \Phi_{s\text{-}1,\,NAME} + C(M-m+1)\} + L_0 = \Phi_{s\text{-}1,\,NAME} + C(M-m+1) + L_0 \qquad \textbf{(16)}$$

where C is the constant factor associated with the loop in lines 8-9. The algorithm time $\Phi_{R,N}$ is obtained iterating the equations:

$$\Phi_{R,N} = \Phi_{N,\;N} + \Sigma_{i=0,M\text{-}1}\,\{(B + L_0) + (C\,(M\text{-}i) + L_0)\} + B \qquad \textbf{(17)}$$

When only P processors are available, the last processor $P\text{-}1$ starts at time $\Phi_{P\text{-}1,\,P\text{-}1}$

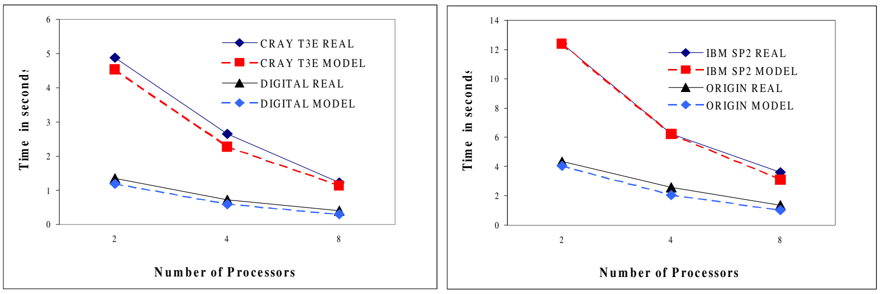

Fig. 3. SRAP (500 resources, 500 tasks) actual and model times.

Thus, the BSPWB complexity $\Phi_{R,N}$ on a P processor machine is obtained by multiplying the computing summand of formula *(17)* by N/P.

$$\begin{aligned} \Phi_{R,N} &= \Phi_{P\text{-}1\;\;P\text{-}1} + (\,\Sigma_{i=0,M\text{-}1}\,\{(B + L_0) + (C\,(M\text{-}i) + L_0)\} + B\,)\,N/P = \\ &= (P\text{-}1)\,B + P\,L_0 + A\,(M+1) + ((B + L_0)\,M + (C\,(M+1)/2 + L_0)\,M + B\,)N/P \end{aligned} \qquad \textbf{(18)}$$

Consult [4] for the values of the computational constants A, B and C. Figure 5 proves the prediction capacity of the model.

References

1. Juurlink B.H.H., Wijshoff H.A.G. *A Quantitative Comparison of Parallel Computation Models.* Proc. SPAA'96. 1996.
2. Marín, J. Martínez, A. *Testing PVM versus BSP Programming*, VIII Jornadas de Paralelismo, pp 153-160. Sept.1997
3. Roda J., Rodríguez C., Almeida F., Morales D.. *Prediction of Parallel Algorithms Performance on Bus Based Networks using PVM*. 6th Euromicro workshop on Parallel and Distributed Processing Madrid, Spain. IEEE Computer Society. Jan-1998.
4. Roda J.L, Rodríguez C., Sande F., Morales D.G., Almeida F. ftp://ftp.csi.ull.es/pub/parallel/BSPWB/constants.html
5. Valiant L.G.. *A Bridging Model for Parallel Computation*. Communications of the ACM, 33(33(8): 103-111, August 1990.