PERFORMANCE AND PREDICTABILITY OF MPI AND BSP PROGRAMS ON THE CRAY T3E

González J.A., Rodríguez C., Roda J.L., Morales D.G.,

Sande F., Almeida F., León C.

Departamento de Estadística, I.O. y Computación,
Universidad de La Laguna
Tenerife – Spain
{jaglez, jlroda, casiano}@ull.es

Abstract. It has been argued that message passing systems based on pairwise, rather than barrier, synchronization suffer from having no simple analytic cost for model prediction. The BSP Without Barriers Model (BSPWB) has been proposed as an alternative to the Bulk Synchronous Parallel (BSP) model for the analysis, design and prediction of asynchronous MPI programs. This work compares the prediction accuracy of the BSP and BSPWB models and the performance of their respective software libraries: Oxford BSPlib and MPI. Three test cases, representing three general problem solving paradigms are considered. These cases cover a wide range of requirements in communication, synchronisation and computation. The results obtained on the CRAY-T3E show not only a better scalability of MPI but that the performance of MPI programs can be predicted with the same exactitude than Oxford BSPlib programs.

1 Introduction

The computational Bulk Synchronous Parallel (BSP) model [10] considers a parallel machine made of a set of p processor-memory pairs, a global communication network and a mechanism for synchronizing the processors. A BSP calculation consists of a sequence of supersteps. The cost of a BSP program can be calculated by summing the cost of each separated superstep executed by the program; in turn, for each superstep the cost can be decomposed into: (i) local computation; (ii) global exchange of data and (iii) barrier synchronization. The communication pattern performed in a given superstep is called an h-relation if the maximum number of packets a processor communicates in the superstep is h:

$$h = max\ \{in_i\ @\ out_i\ /\ i \in \{0,...,p-1\}\} \qquad (1)$$

where @ denotes a binary operator, usually the maximum or the sum. Values in_i and out_i respectively denote the number of packets entering and leaving processor i. The particular operation @ and the packet size depend on the particular architecture.

Operation @ is closer to the maximum for machines allowing a complete parallel processing of incoming/outgoing messages. Correct interpretation for each case depends on the number of input/output ports of the network interface. Our experiments prove that for most current machines @ is closer to the sum than to the maximum.

The two basic BSP parameters that model a parallel machine are: the gap g_p, which reflects per-processor network bandwidth, and the minimum duration of a superstep L_p, which reflects the time needed by all processors to synchronize as well as the time to send an unitary packet through the network. Let denote by W the maximum time spent in local computation by any processor during the superstep. The BSP model guess that the running time of a superstep is bounded by the formula:

$$Time\ Superstep = W + g_p\ h + L_p \qquad (\mathbf{2})$$

Although BSP programs can be expressed using existing communication libraries as MPI, the counterpart is not always true. It has been stated that message passing systems based on pairwise, rather than barrier, synchronization suffer from having no simple analytic cost for model prediction ([8] page 22). The **BSP Without Barriers** model (BSPWB) introduced in [7] is a proposal to fill this gap. The execution of a MPI program in any processor consists in phases of computation followed by the necessary communications to provide and obtain the data for the next phase. Communication in this context means a *continuous stream* of messages. BSPWB divides a MPI program in what is called 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 M-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 time $t_{s,i}$ when processor i finishes its *s-th* M-step is the value $\Phi_{s,i}$ given by the BSPWB model. We define the set $\Omega_{s,i}$ for a given processor i and M-step s as the set

$$\Omega_{s,i} = \{\ j\ /\ \text{Processor } j \text{ sends a message to processor } i \text{ in M-step } s\ \} \cup \{\ i\} \qquad (\mathbf{3})$$

Processors in the set $\Omega_{s,i}$ are called "**the incoming partners of processor *i* in step *s***". The BSPWB time $\Phi_{s,i}$ of a MPI program is recursively defined by the formulas:

$$\begin{aligned} \Phi_{1i} &= max\ \{\ w_{1,j}\ \ /\ j \in \Omega_{1,i}\} + (g\ h_{1,i} + L),\ \ i = 0,1,..., p\text{-}1 \\ \Phi_{s,i} &= max\{\Phi_{s-1,j}+w_{s,j}\ \ /\ j\in\Omega_{s,i}\} + (g\ h_{s,i} + L),\ s=2,..,R,\ \ i=0,1,..., p\text{-}1 \end{aligned} \qquad (\mathbf{4})$$

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:

$$h_{s,i} = max \{in_{s,j} @ out_{s,j} / j \in \Omega_{s,i}\}, \ s = 1,...,R, \ i = 0,1,..., p-1 \quad \mathbf{(5)}$$

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.

Formula (5) 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. Formula (5) charges the communication time of processor i with the maximum communication time of any of its incoming partner processors. The model assumes that buffering capabilities are large enough or that processors have the capability 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 change in its slope. 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 program in the BSPWB model is given by

$$\Psi = max \{ \Phi_{R,j} / j \in \{0,...,p-1\}\} \quad \mathbf{(6)}$$

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.

This work compares the prediction accuracy of the two models and the performance of their respective software platforms, BSPlib [2] and MPI [1], on a CRAY T3E. The next paragraph discusses the experiments to evaluate the parameters for the two models. The test cases representing three general problem solving paradigms are considered: Greedy, Divide and Conquer and Dynamic Programming. These cases cover a wide range of requirements in communication, synchronization and computation. The results not only show the better scalability of MPI but also that the performance of MPI programs can be predicted with the same exactitude than Oxford BSPlib programs.

2 BSP and BSPWB Parameters

The accuracy of the two considered models depends on the estimation of the gap and latency parameters, g and L. Runtime systems oriented to the BSP model try to get closer actual machines to the BSP ideal machine by packing individual messages generated during a superstep, and optimizing communication time by rearranging the order in which messages are delivered at the end of the superstep. This policy reduces the influence of the communication pattern, since it gives place to a roughly *AllToAll* communication at the end of each superstep. From this perspective, it is justified the proposal [3], [8] of taking as value for g the associated with the *AllTo-All* pattern. This is, roughly speaking, the policy used to obtain the different estimations of the BSP parameters on the Oxford BSPlib. The values of g_p and L_p for the CRAY T3E appear in Table 1.

processors	**2**	**4**	**8**	**16**
***L* (sec)**	5.46E-06	7.24E-06	1.03E-05	1.52E-05
***g* (sec/word)**	5.30E-08	3.59E-08	3.33E-08	3.37E-08

Table 1. Latency and gap values of the BSP Model.

Juurlink [5] and Hill [3] 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 $\Pi = \{E, PP, OA, POA, AO, AA\}$ of six communication patterns among those that most commonly appear in parallel algorithms: *Exchange (E), PingPong (PP), OneToAll (OA), PersonalizedOneToAll (POA) AllToOne (AO) and AllToAll (AA)*. Under the @ = + assumption, all these six patterns give place to the same h-relation if the following sequence of equalities hold:

$$h = m_{PP} = 2 * m_E = (P-1) * m_{OA} = (P-1) * m_{POA} = (P-1) * m_{AO} = 2*(P-1)*m_{AA} \tag{7}$$

where m_X denotes the message size of the pattern X. According to the h-relation hypothesis, communication time of these patterns for these message sizes have to be the same. The values of g, L_0 and L for the native versions of MPI were obtained using least square fit on the average time values in the different communication patterns and processor numbers. These values appear in Table 2.

	L_0 (sec)	***L* (sec)**	***g* (sec/word)**
BSPWB	1.00E-5	-2.11E-5	2.92E-8

Table 2. BSPWB Model parameters for the CRAY T3E.

Instead of using a single universal computational constant ***s***, our analysis (both for BSP and BSPWB) of the computing times implies experimental evaluation. The maximal portions of code taking analytical constant time must be timed. The values of the computational constants appearing of the three examples in the next paragraph can be found in ftp://ftp.csi.ull.es/pub/parallel/*BSPWB/pvmmpi99.html*.

3 Benchmarks: Prediction and Performance

3.1 Reduction Parallelism Test: Computing the Minimum Spanning Tree

Let *G = (V,E)* be a graph with a set of *N* vertices *V* and a set of edges *E*. A graph is called a tree if it is a connected acyclic graph. A *spanning tree* of an undirected graph *G* is a subgraph *T = (V, E')* of *G* that is a tree containing all the vertices of *G*. In a weighted graph each edge *(i,j)* is labeled with a weight *w[i][j]*. The weight of a subgraph is the sum of the weights *w[i][j]* of the edges *(i,j)* in the subgraph. A minimum spanning tree *MST* for a weighted undirected graph is a spanning tree with minimum weight. The parallel version of the Prim algorithm [9] to compute the *MST* presented in Figure 1 is a canonic example of the greedy technique.

```
1. void Prim(VertexSetType  V_name , T, VertexType  r)
2. {
3.   T ={r};
4.   d[r] = 0;
5.   for v in (V_name-T)
6.     d[v]=w[r][v];
7.   for i=2 to |V| {
8.     Let be u_name such that d[u_name]=min{d[v]/v in V_name-T};
9.     REDUCE(CompareDist,z,u_name,sizeof(VertexType));
10.    T := T ∪ {z};
11.    for v in (V_name-T)
12.      d[v] = min{d[v], w[z][v]};
13.  };
14.} /* Prim */
```

Fig. 1. Parallel pseudo-code for the MST.

The set of vertices *V* is partitioned among the *p* processors in disjoint subsets V_{name} of size *N/p*. The matrix *w* is distributed in such way that processor *name* contains the columns *w[i][j]* for *i* in V_{name} and *j* in *V*. At the end of the algorithm each processor have a replica of the minimum spanning tree *T*. At any time, the vector *d[v]* indexed in the vertices *v* of *V-T* contains the minimum distance from *v* to the current tree *T*. The tree and the *d* vector are initialized in time *A+B*N/p* in lines 3-6. On each stage *i* of the loop in lines 7- 13 a new vertex is introduced in the minimum spanning tree *T*. Each processor *name* finds the nearest vertex u_{name} to the tree *T* (line 8) in time *C+D*N/p*. An all to all generic reduction on the commutative operation *CompareDist*(u_i, u_j) is used to obtain the global nearest vertex *z* to the tree *T*. Both libraries MPI and Oxford BSPlib provide generic reductions routines: *MPI_Allreduce* in MPI and *bsp_fold* in the *Oxford BSPlib*. Thus, the differences in performance between the two implementations of the algorithm are basically due to the differences in efficiency of these two routines. Since the *h*-relations involved in the reduction correspond to one word, the time spent is dominated by the latency *L*. Although the insides of these routines are hidden to the programmer, we can expect the time of any smart implementation to behave logarithmically in the number of processors $R*log(P)+S = (R'+L+g)*log(P)+S$, where L and g are the Latency and gap of the respective models (L_0 for BSPWB). Constants R' and S reflect the computing overhead of the reduction subroutine. That overhead depends on the operation, and can not be neglected. Since the task of estimating the communication time inside the reduction can not be traced, we have instead estimated R and S by linear square fit in the logarithm of the processor number.

Once the nearest vertex z has been introduced in the tree (line 10), the distances $d[v]$ of the vertices v to the tree are computed in time $H*N/P+J$ in lines 11 and 12. Therefore, the total cost, both in the BSP and BSPWB models, is given by

$$A+B*N/p + C +D*N/p+ (R'+L+g)*log(P)+S +H*N/P+J \qquad \textbf{(8)}$$

Table 3 corresponds to the average times of five executions of the former algorithm for a 4096 vertices graph on the CRAY-T3E at CIEMAT (www.ciemat.es).

processors	**2**	**4**	**8**	**16**
Predicted BSP	4.87	3.34	2.72	2.54
Real BSPlib	5.45	4.00	3.37	2.94
Predicted BSPWB	4.55	3.07	2.22	1.84
Real MPI	5.41	3.78	2.96	2.14

Table 3. Predicted and real times (Prim algorithm).

The generic reduction MPI routine is always faster than the corresponding Oxford BSP routine. Although it does not appear in the tables, the differences percentage *100*(MPIAlltoall-BSPfold)/MPIAlltoall* between the time of the two reduce subroutines increases with the number of processors, going from 35% for two processors to 107% for sixteen processors. The error percentage (Table 4) incurred by the predictions (*100*(Real-Model) /Real*) are similar for both the MPI and BSPlib times (round 13% for 16 processors).

processors	**2**	**4**	**8**	**16**
BSP	10.60	16.48	19.36	13.44
BSPWB	15.83	18.75	24.94	13.93

Table 4. BSP and BSPWB prediction percentage errors (Prim algorithm).

3.2 Data Parallelism Test. The Fast Fourier Transform

The pseudocode in Figure 2 shows a divide and conquer (D&C) parallel algorithm computing the Discrete Fourier Transform. The code is essentially the same both for MPI and BSP. The differences between the two implementations have been hidden in the macro PAR (Figure 3).

The algorithm has as input a replicated vector of complex A, and the number n of elements. The result vector B is also required to be in all the processors. The variable *NUMPROCESSORS* holds the number p of processors available at any instant.

If there is only one processor, a call to the sequential algorithm *seqFFT* solves the problem in time $C*n/p*log(n/p)+D$. Otherwise, the algorithm tests if the problem is trivial. If not, function *odd_and_even()* decompose the input signal *A* in its odd and even components in time proportional to the size of the vector. After that, the *PAR* construct in line 15 makes two parallel recursive calls to the function *parFFT*. The results of these computations are returned in *A1* and *A2* respectively. Therefore, the time spent in the successive divisions is:

$$(A*n/2+B)+(A*n/4+B)+...+(A*n/p+B) = A*n*(p-1)/p + B*log(p) \quad \textbf{(9)}$$

```
1. void parFFT(Complex *A, Complex *B, int n) {
2.   Complex *a2, *A2;       /* Even terms */
3.   Complex *a1, *A1;       /* Odd terms */
4.   int m, size;
5.
6.   if(NUMPROCESSORS > 1) {
7.     if (n == 1) {            /* Trivial problem */
8.       b[0].re = a[0].re;
9.       b[0].im = a[0].im;
10.     }
11.     else {
12.       m = n / 2;
13.       size = m * sizeof(Complex);
14.       Odd_and_even(A, a2, a1, m);
15.       PAR(parFFT(a2,A2,m), A2,size,
16.           parFFT(a1,A1, m), A1,size);
17.       Combine(B, A2, A1, m);
18.     }
19.   }
20.   else
21.     seqFFT(A, B, n);
22. }
```

Fig. 2. Code for the FFT.

```
#define PAR(f1, r1, s1, f2, r2, s2) {
  int partner;
  NUMPROCESSORS /= 2; BIT--;
  partner = NAME ^ (1 << BIT);
  if ((NAME & (1 << BIT)) == 0) {
    f1;
    SWAP(r1,s1,partner,r2,s2);
  }
  else {
    f2;
    SWAP(r2,s2,partner,r1,s1);
  }
  NUMPROCESSORS *= 2; BIT++;
}
```

Fig. 3. Code of the macro PAR.

Figure 3 shows the pseudocode produced by the expansion of the *PAR* macro. The macro divides the set of processors in two groups, *G* and *G'*. Each processor *NAME* in a group *G* chooses a processor *PARTNER* in the other group *G'*, and the results are interchanged between partners. As it is proved in [6], this policy is more effective that the use of *MPI_Comm_split*. The first exchange of the two vectors of size n/p marks the end of the first superstep in BSP and the first M-step in BSPWB (lines 7 and 11 of figure 3). Consequently, the time of this first step is given by:

$$A*n*(p-1)/p + B*log(p) + (C*n*log(n/p)/p+D) + h_M*g_M+L_M \qquad \textbf{(10)}$$

Where g_M and L_M denote the gap and latency in the parallel computing model M and h_M is the h-relation size: $h_{BSP} = n/p$ for BSP and $h_{BSPWB} = 2*n/p$ for BSPWB.

The $log(p)$ successive calls to the function *Combine()* (line 17) compute the Fourier transform of the input vector in time $E*m+F$ proportional to the sizes m of the auxiliary vectors $A1$ and $A2$. Unless for the last combination, the first $log(p)-1$ combinations are followed by the swapping of the results. Thus, we have $log(p)$ steps with cost:

$$(E*(p-1)*n/p+F*log(p))+g_M*\&_M*n*(p-2)/p+(log(p)-1)*L_M \qquad \textbf{(11)}$$

where the factor $\&_M = 1$ for BSP and $\&_M = 2$ for the BSPWB is a consequence of the different interpretations of @ in formula (1).

processors	**2**	**4**	**8**	**16**
Predicted BSP	5.29	2.97	1.88	1.36
Real BSPlib	5.11	2.95	1.92	1.45
Predicted BSPWB	5.29	2.98	1.89	1.38
Real MPI	5.29	2.99	1.90	1.38

Table 5. Predicted and real times (FFT algorithm).

processors	**2**	**4**	**8**	**16**
BSP	-3.61	-0.87	2.31	5.73
BSPWB	-0.06	0.24	0.76	0.06

Table 6. BSP and BSPWB predicted percentage errors (FFT algorithm).

An overlapping of the predicted and actual times on both software platforms is observed. As for the former experiment, the better performance of the MPI version grows with the processor number. The percentage error of the times predicted by the BSPWB model for MPI is less than 1%, which is smaller than the obtained using the BSP model for the Oxford BSPlib. This is a remarkable result when considering that the values of g and L used in BSPWB do not depend on the number of processors.

3.3 Pipeline Parallelism Test. The 0-1 Knapsack Problem

The 0-1 Knapsack Problem (KNP) is defined by a set of N objects of given weights w_i an profits p_i and a knapsack with a limited capacity M. The goal is to find a subset of objects fitting in the knapsack and maximizing the total profit. Let us denote by $Q[k][m]$ the optimal income for the KNP subproblem constituted by the first k objects and a knapsack of capacity m. Applying the Dynamic Programming Principle to this problem leads to the following state equation:

$$Q[k][m]=max\ \{Q[k-1][m],\ Q[k-1][m-wk]+p_k\},\ N \geq k \geq 1,\ m \geq w_k \qquad \textbf{(12)}$$

$$Q[k][m]=Q[k-1][m],\ N \geq\ k \geq 1,\ m < w_k$$

$$Q[0][m] = 0,\ \ m = 0,\ 1, ...,\ M$$

A parallel pipeline algorithm using *N* processors can be easily designed taking advantage of the fact that dependencies in formula (12) occur among adjacent values of *k*. Each processor *name* takes charge of computing the values of a different row *Q[name].* Figure 4 presents the parallel pseudocode.

```
1.  int KNP(int M, int Last) {
2.     int *Q, q, c, i;
3.     Q=(int*)calloc(M+1, sizeof(int));
4.     for(c=0;c<=M;c++) {
5.       receive(PREVIOUS,&q);
6.       Q[c]=max(Q[c], q);
7.       i = c+wname;
8.       if (i ≤ M) Q[i]=max(Q[i], q+pname);
9.       if (!Last) send(NEXT, &Q[c]);
10.      /* barrier if BSP */
11.    }
12.    q=Q[M];
13.    free(Q);
14.    return q;
15. }
```

Fig. 4. Pipeline algorithm for the KNP.

Through the call to the *calloc* function in line 3, each processor *name* initializes to zero its vector of optimal values *Q* in time A_1*M+B_1. The computation inside the loop from lines 5 to 10 takes constant time A_2. To optimize communications, the MPI implementation combines the send at line 9 and the receive at line 5 in a single *MPI_Sendrecv* operation. Thus, each iteration of the loop gives place to a new step (M-step for the BSPWB model, superstep for the BSP model). On each step a single word is received and another is sent. The time taken by the *h*-relation is dominated by the synchronization/latency time *L* (constant L_0 in BSPWB). Neglecting the contribution of the message size, the loop time can be approached by $(M+1)*(A_2+L)$. Denoting by A_3 the constant time consumed in lines 12-14, we can write the time of the code in figure 4 as:

$$A_1*M+B_1+\ (M+1)*(A_2+L)\ +\ A_3 \qquad \textbf{(13)}$$

The *N* virtual processors of the former algorithm have been mapped into the *p* physical processors using a cyclic policy. The computation is divided in *N/p* stages. On stage *b* processor *k* virtualizes processor *name* = *b*N/p+k*. In the MPI version the BSPWB time complexity is increased by the virtualization factor *N/p* plus the time $L+(p-2)*(A_2+L)$ necessary to load the pipe:

$$L+(p-2)*(A_2+L)+ \{A_1*M+B_1+ (M+1)*(A_2+L) + A_3\}*N/p \qquad \textbf{(14)}$$

This virtualization is also carried out in the Oxford BSPlib version. The synchronous style of BSP forces each processor to synchronize in every superstep before any data is available for that processor. Since, in our implementation, the pipe is loaded at the beginning of each band, the BSP time of the algorithm is given by:

$$\{A_1*M+B_1+ (M+1+ (p-1))*(A_2+L_p) + A_3\}*N/p \qquad \textbf{(15)}$$

	2	**4**	**8**	**16**
Predicted BSP	27.73	15.64	9.34	5.92
Real BSPlib	30.95	16.83	9.46	6.62
Predicted BSPWB	36.75	18.37	9.19	4.59
Real MPI	34.61	17.67	9.03	4.86

Table 7. Actual and Predicted times for BSP and BSPWB for the KNP.

Errors	**2**	**4**	**8**	**16**
BSP	10.41	7.07	1.25	10.57
BSPWB	-6.17	-3.99	-1.73	5.51

Table 8. BSP and BSPWB percentage errors.

As it was observed in the FFT case, the Oxford BSPlib version is faster up to 4 processors. However, for 8 and 16 processors the MPI version presents a better time. Lets again emphasize that although the values of g_p and L_p in the BSP prediction formulas have been parameterized in the number p of processors, the errors observed in BSP are larger than the observed for the BSPWB model.

4 Conclusions

The results obtained for the three considered parallel applications not only show a better scalability of MPI but that the performance of MPI programs can be predicted with the same exactitude than the performance of Oxford BSPlib programs. However, for two cases, the Oxford BSPlib is slightly faster when the number of processors is small and for the Prim example the times are similar. The performance of MPI is better for the three cases when the number of processors increases. The differences both in the performance and in the precision can not be considered substantial.

Acknowledgements

We wish to thank to the Centro de Investigaciones Energéticas Medioambientales y Tecnológicas and to the Centre de Computació i Comunicacions de Catalunya for allowing us the access to their resources.

References

1 Gropp W. and Lusk E. and Doss N. and Skjellum A. *A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard*. Journal of Parallel Computing, vol.22, number 6, pp.789-828, Sep 1996.

2 Hill J., McColl B., Stefanescu D., Goudreau M., Lang K., Rao S.B., Suel T., Tsantilas T., Bisseling R. *BSPlib: The BSP Programming Library.* University of Oxford Parallel Applications Centre. May. 1997.

3 Hill, J., Donaldson, S. *Stability of Communication Performance in Practice: From the CRAY-T3E to Networks of Workstations*. Oxford University Computing Laboratory. PRG-TR-33-97. 1997.

4 Ibaraki T. *Enumerative Approaches to Combinatorial Optimization. Part I and II.* J.C. Baltzer Scientific Pub. Co. 1987.

5 Juurlink B.H.H., Wijshoff H.A.G. *A Quantitative Comparison of Parallel Computation Models.* Proc. SPAA'96. 1996.

6 Roda J., Sande F., León C., González J., Rodríguez C. *The Collective Computing Model*. Proc. 7th EuroMicro WorkShop on Parallel and Distributed Processing. pp.19-26. IEEE. 1999.

7 Roda J., Rodríguez C., Morales D.G., Almeida F., Pulido P., Dorta D. *Breaking the Barriers: Two Models for MPI Programming*. Proc. of Int. Conference on Parallel Architectures and Compilation Techniques. pp. 248-255, 1998.

8 Skillcorn, D.B., Hill, J., McColl, W.F. *Questions and Answers about BSP*. Oxford University Computing Laboratory. Report PRG-TR-15-96. 1996.

9 Kumar V., Grama A., Gupta A., Karypis G. *Introduction to Parallel Computing. Design and Analysis of Algorithms*. Benjamin Cumming Pub. California. 1994.

10 Valiant L.G. *A Bridging Model for Parallel Computation*. Communications of the ACM, 33(8). pp. 103-111, 1990.