BSP Algorithm Design

Disclaimer: These notes DO NOT substitute the textbook for this class. The notes should be used in conjunction with the textbook and the material presented in class. If a statement in these notes seems to be incorrect, report it to the instructor so that it be fixed immediately. These notes are distributed to the students of CIS 668; distribution outside this group of students is prohibited.

As an example of how traditional PRAM algorithm design differs from architecture independent parallel algorithm design, example algorithm for broadcasting in a parallel machine is introduced.

Problem: In a parallel machine with p processors numbered $0, \ldots, p-1$, one of them, say processor 0, holds a one-word message The problem of *broadcasting* involves the dissemination of this message to the local memory of the remaining $p-1$ processors.

The performance of a well-known exclusive PRAM algorithm for broadcasting is analyzed below in two ways under the assumption that no concurrent operations are allowed. One follows the traditional (PRAM) analysis that minimizes parallel running time. The other takes into consideration the issues of communication and synchronization as viewed under the BSP model. This leads to a modification of the PRAM-based algorithm to derive an architecture independent algorithm for broadcasting whose performance is consistent with observations of broadcasting operations on real parallel machines.

Algorithm. Without loss of generality let us assume that p is a power of two. The message is broadcast in $\lg p$ rounds of communication by binary replication. In round $i = 1, \ldots, \lg p$, each processor j with index $j < 2^{i-1}$ sends the message it currently holds to processor $j + 2^{i-1}$ (on a shared memory system, this may mean copying information into a cell read by this processor). The number of processors with the message at the end of round i is thus 2^i.

Analysis of Algorithm. Under the PRAM model the algorithm requires $\lg p$ communication rounds and so many parallel steps to complete. This cost, however, ignores synchronization which is for free, as PRAM processors work synchronously. It also ignores communication, as in the PRAM the cost of accessing the shared memory is as small as the cost of accessing local registers of the PRAM.

3

Analysis of Algorithm. Under the BSP cost model each communication round is assigned a cost of $\max\{L, g \cdot 1\}$ as each processor in each round sends or receives at most one message containing the one-word message. The BSP cost of the algorithm is $\lg p \cdot \max\{L, g \cdot 1\}$, as there are $\lg p$ rounds of communication. As the communicated information by any processors is small in size, it is likely that latency issues prevail in the transmission time (ie bandwidth based cost $g \cdot 1$ is insignificant compared to the latency/synchronization reflecting term L).

In high latency machines the dominant term would be $L \lg p$ rather than $g \; \lg p$. Even though each communication round would last for at least L time units, only a small fraction g of it is used for actual communication. The remainder is wasted. It makes then sense to increase communication round utilization so that each processor sends the one-word message to as many processors as it can accommodate within a round.

Input: p processors numbered $0 \ldots p-1$. Processor 0 holds a message of length equal to one word.

Output: The problem of *broadcasting* involves the dissemination of this message to the remaining $p-1$ processors.

Algorithm 2. In one superstep, processor 0 sends the message to be broadcast to processors $1, \ldots, p-1$ in turn (a "sequential"-looking algorithm).

Analysis of Algorithm 2.

The communication time of Algorithm 2 is $1 \cdot \max\{L, (p-1) \cdot g\}$ (in a single superstep, the message is replicated $p-1$ times by processor 0).

Algorithm 3. Both Algorithm 1 and Algorithm 2 can be viewed as extreme cases of an Algorithm 3. The main observation is that up to L/g words can be sent in a superstep at a cost of L. Then, It makes sense for each processor to send L/g messages to other processors. Let $k-1$ be the number of messages a processor sends to other processors in a broadcasting step. The number of processors with the message at the end of a broadcasting superstep would be k times larger than that in the start. We call k the degree of replication of the broadcast operation.

Architecture independent Algorithm 3. In each round, every processor sends the message to $k-1$ other processors. In round $i = 0, 1, \ldots$, each processor j with index $j < k^i$ sends the message to $k-1$ distinct processors numbered $j + k^i \cdot l$, where $l = 1, \ldots, k-1$. At the end of round i (the $(i+1)$-st overall round), the message is broadcast to $k^i \cdot (k-1) + k^i = k^{i+1}$ processors. The number of rounds required is the minimum integer r such that $k^r \geq p$, The number of rounds necessary for full dissemination is thus decreased to $\lg_k p$, and the total cost becomes $\lg_k p \max \{L, (k-1)g\}$.

At the end of each superstep the number of processors possessing the message is k times more than that of the previous superstep. During each superstep each processor sends the message to exactly $k-1$ other processors. Algorithm 3 consists of a number of rounds between 1 (and it becomes Algorithm 2) and $\lg p$ (and it becomes Algorithm 1).

BROADCAST $(0, p, k)$

```
1.    my_pid = bsp_pid();    mask_pid = 1;
2.    while (mask_pid < p)  {
3.        if  (my_pid < mask_pid)
4.            for (i = 1, j = mask_pid; i < k; i + +, j+ = mask_pid) {
5.                target_pid = my_pid + j;
6.                if  (target_pid < p)
7.                    bsp_put(target_pid, &M, &M, 0, sizeof(M));
8.            }
9.        bsp_sync();
10.       mask_pid = mask_pid * k;
```


Now suppose that the message to be broadcast consists of not a single word but is of size $n > p$. Algorithm 4 may be a better choice than the previous algorithms as one of the processors sends or receives substantially more than n words of information. There is a broadcasting algorithm, call it Algorithm 4, that requires only two communication rounds and is optimal (for the communication model abstracted by L and g) in terms of the amount of information (up to a constant) each processor sends or receives.

Algorithm 4.

Two-phase broadcasting

The idea is to split the message into p pieces, have processor 0 send piece i to processor i in the first round and in the second round processor i replicates the i-th piece $p - 1$ times by sending each copy to each of the remaining $p - 1$ processors (see attached figure).

Stage 1   max{L,gn}

Stage 2   max{L,gn}

0   1   2   3
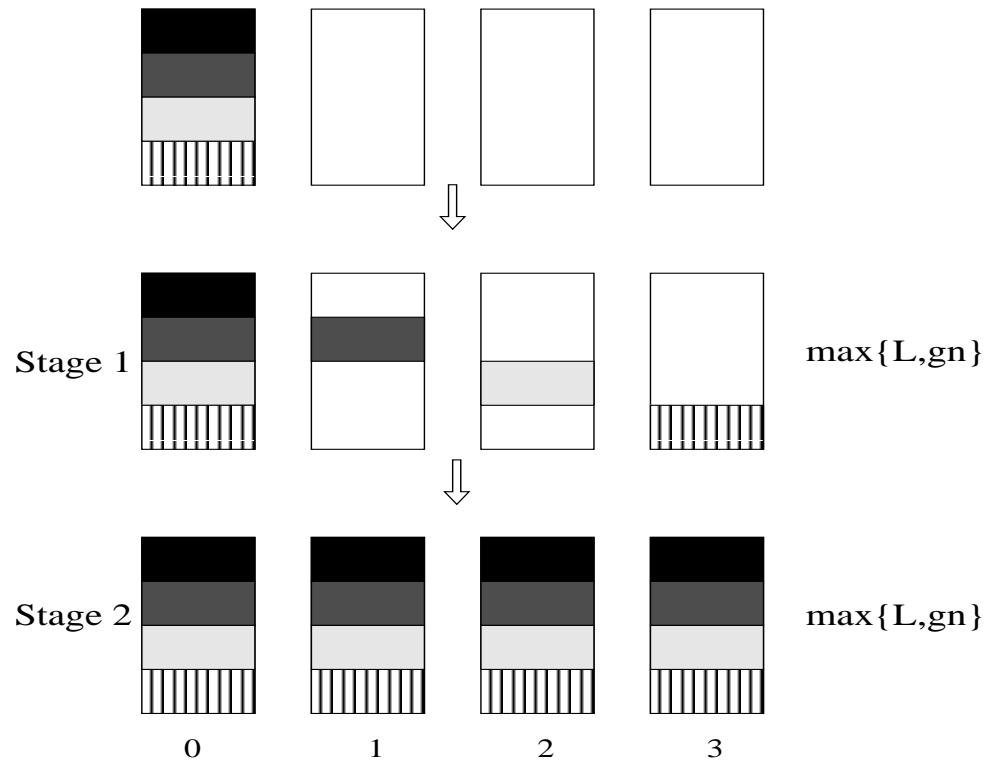
Exercise. What can you say about parallel prefix? Analyze the BSP performance of the PRAM algorithm for parallel prefix. Can you halve its number of supersteps yet maintain the same BSP cost?

The structure of the four algorithms described for broadcasting can also be used to derive algorithms for parallel prefix that require similar number of supersteps (at most twice as many).

Algorithm 1 gives rise to a "sequential"-like parallel prefix algorithm. Algorithm 2 gives rise to the binary tree based algorithm that requires $2 \lg n$ supersteps. The corresponding PRAM algorithm, however, (that also runs on the butterfly) requires half as many supersteps and is thus more efficient on the BSP model. Algorithm 3 gives rise to the equivalents of 2 when the number of supersteps needs to be decreased.

We can generalize the prefix problem so that each processor instead of holding a single scalar value, holds a sequence/vector of scalar values n. In the case $n > p$, implementations following the counterparts of Algorithm 1,2 and 3 for broadcasting fail to provide optimal algorithms.

Algorithm 4 gives rise to a two-phase parallel prefix algorithm that is more efficient in architectures with large L for large independent prefix problems n.

SPMD program design stipulates that processors executes a single program on different pieces of data. For matrix related computations it makes sense to distribute a matrix evenly among the p processors of a parallel computer. Such a distribution should also take into consideration the storage of the matrix by say the compiler so that locality issues are also taken into consideration (filling cache lines efficiently to speedup computation). The are various ways to divide a matrix. Some of the most common one are described below.

- **block** distribution. Split an array into square blocks each one dimension $n/\sqrt{p} \times n/\sqrt{p}$ and store block i to processor i. The blocks need not be square in the general case.

- **column-block** distribution. Split matrix into p column blocks so that n/p consecutive columns form a block i that will be stored to processor i.

- **row-block** distribution. Similar to column-block distribution.

- **wrapped-around column** distribution. Assign individual columns to processors, say, by assigning column i to processor $i \pmod p$. This distribution is preferable in cases where elimination methods are used.

- other combination of these methods.

The BSP algorithm for matrix multiplication presented below was presented in the seminal work of Valiant. It works for $p \leq n^2$. Each processor is assigned the task of computing an $n/\sqrt{p} \times n/\sqrt{p}$ submatrix of the product $A \times B$. The input matrices A and B are divided into p block-submatrices, each one of dimension $m \times m$, where $m = n/\sqrt{p}$. We call this distribution of the input among the processors *block* distribution. This way, element $A(i,j)$, $0 \leq i < n, 0 \leq j < n$, belongs to the $(j/m)*\sqrt{p}+(i/m)$-th block that is subsequently assigned to the memory of the same-numbered processor. Let A_i (respectively, B_i) denote the i-th block of A (respectively, B) stored in processor i. With these conventions the algorithm can be described in Figure 1. The following Proposition describes the performance of the aforementioned algorithm.

---

begin MULT_A (C,A,B,n,p)
1.   Let $m = n/\sqrt{p}$;
       Each processor is also assigned a unique processor number q;
2.   Let $p_i = q \bmod \sqrt{p}$; $p_j = q/\sqrt{p}$; $C_q = 0$;
3. $a_l \leftarrow A_{p_i+l*\sqrt{p}}, 0 \leq l < \sqrt{p}$;
4. $b_l \leftarrow B_{p_j*\sqrt{p}+l}, 0 \leq l < \sqrt{p}$;
5.   **for** $0 \leq l < \sqrt{p}$ **do**
         $C_q = C_q + a_l \times b_l$;
end MULT_A

Figure 1: Procedure MULT_A.

Proposition 1 *Algorithm* MULT_A *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \max\{L, \frac{(2n-1)n^2}{p}\},$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \max\{L, g\frac{2n^2}{\sqrt{p}}\}.$$

One immediately realizes that algorithm MULT_A is not memory efficient since it requires more local memory per processor – by a factor of $\sqrt{p}$ – than the required one. Algorithm MULT_B shown in Figure 2 is the memory efficient variant of MULT_A. It is not synchronization efficient though since its number of supersteps is not constant any more; it has been increased by a factor of $\sqrt{p}$. The performance of algorithm MULT_B is summarized in Proposition 2.

> **begin** MULT_B (C,A,B,n,p)
> 1.   Let $m = n/\sqrt{p}$;
>   Each processor is also assigned a unique processor number q;
> 2.   Let $p_i = q \bmod \sqrt{p}$; $p_j = q/\sqrt{p}$; $C_q = 0$;
> 3.   **for** $0 \le l < \sqrt{p}$ **do**
>   **begin**
> 4. $a \leftarrow A_{((p_i+p_j+l) \bmod \sqrt{p})*\sqrt{p}+p_i}$;
> 5. $b \leftarrow B_{((p_i+p_j+l) \bmod \sqrt{p})+p_j*\sqrt{p}}$;
> 6. $C_q = C_q + a \times b$;
>   **end**
> **end** MULT_B

Figure 2: Procedure MULT_B.

Proposition 2 *Algorithm* MULT_B *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \le n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) = \sqrt{p} \, \max\{L, \frac{(2n-1)n^2}{p^{3/2}}\}$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \sqrt{p} \, \max\{L, g\frac{2n^2}{p}\}$$

Experimental Results

In order to show the efficiency of algorithm design on the BSP model we present some experimental results for matrix multiplication on Cray T3D; additional results can be found in the author's Web page. Algorithm MULTT_B is a variation of MULT_B where in order to multiply A with B, matrix A is first transposed and the loop for matrix multiplication is changed accordingly. This way the access patterns for both A and B are the same (column - column as opposed to row - column) thus improving locality (cache usage), and subsequently program performance.

	Algorithm MULT_B							
	$p = 1$		$p = 4$		$p = 16$		$p = 64$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	4.1	7.9	1.1	7.8	0.28	7.4	0.03	13.9
512	34.0	7.8	8.4	7.9	2.1	7.7	0.56	7.4
1024	289.8	7.4	68.4	7.8	16.9	7.9	4.3	7.7
2048	-	-	-	-	136.8	7.8	33.8	7.9

Table 1: Execution time for MULT_B on the Cray T3D

	Algorithm MULTT_B							
	$p = 1$		$p = 4$		$p = 16$		$p = 64$	
n	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate	Time (sec)	Mfl rate
256	2.3	14.3	0.58	14.4	0.15	13.7	0.03	15.1
512	20.7	12.9	4.7	14.1	1.16	14.4	0.30	13.5
1024	202.7	10.5	41.7	12.8	9.4	14.1	2.3	14.3
2048	-	-	-	-	83.5	12.8	19.0	14.1

Table 2: Execution time for MULTT_B on the Cray T3D

Finally, we outline a matrix multiplication algorithm that is computation, communication and synchronization efficient. It fails, however, to be memory efficient, as its memory requirements are a multiplicative factor $p^{1/3}$ from the optimal. Algorithm MultT_C is outlined in the remainder of this section.

In MultT_C matrices A and B (and the result C) are split into two ways into submatrices. Each matrix (A, B and the result C) is split into p "physical" block-submatrices, as in the previous algorithms, each of size $n/p^{1/2} \times n/p^{1/2}$. A "physical" block-submatrix indicates the part of the matrix stored in a single physical (processor) location (i.e. block-submatrix A_i is stored in processor i). At the same time, each of the three matrices is split into $p^{2/3}$ "virtual" block-submatrices each of size $n/p^{1/3} \times n/p^{1/3}$. A "virtual" block-submatrix indicates the block geometry that will be used in the matrix multiplication algorithm to be outlined below. The elements of a "virtual" block-submatrix may be stored in more than one physical processors.

Whereas in the first two algorithms "physical" and "virtual" block-submatrices coincided in number and dimension, in this communication efficient algorithm are clearly distinguished.

Let the "virtual" block-submatrices be identified as A_{ij}, B_{ij} and C_{ij}. Matrix multiplication will thus require the computation of all $C_{ij} = \sum_{k=1}^{p^{1/3}} C_{ijk} = \sum_{k=1}^{p^{1/3}} A_{ik}B_{kj}$, where $C_{ijk} = A_{ik}B_{kj}$.

The algorithm consists of the following steps. We name the processors (i, j, k) the way we did in the matrix multiplication algorithm on the hypercubic networks.

Step 1. Processor (i, j, k) gets A_{ik} and B_{kj}. Note that each of these two "virtual" block-submatrices may originate from more than one processors. Each processor sends at most $2n^2/p$ elements (but each one replicated $p^{1/3}$ times) and receives at most $2n^2/p^{2/3}$ elements. The communication cost of Step 1 is $\max\{L, 2gn^2/p^{2/3}\}$. Subsequently, the two submatrices are multiplied as in the sequential case a step requiring at most $\max\{L, 2n^3/p\}$ time. Partial-submatrix C_{ijk} is thus computed on processor (i, j, k). Each element of such a submatrix is a partial sum of an element c_{lm} of the result matrix C.

Step 2. Each element of C_{ijk} is transmitted from (i, j, k) to that physical processor that stores the "physical" block-submatrix of C whose elements will be formed as sums of the receiving elements (partial sums) of C_{ijk}. Note that each (i, j, k) processor may send its elements to more than one physical processors. At the completion of this step, each of the p processors storing a block-submatrix of C of dimension $n/p^{1/2} \times n/p^{1/2}$ receives at most $p^{1/3} \cdot n^2/p$ such elements (partial sums). The complex communication performed in this step requires time $\max\{L, gn^2/p^{2/3}\}$.

Step 3. The received partial sums are added. $p^{1/3}$ partial sums are summed to give an element of C stored at a physical processor, for a total of n^2/p such elements (of a "physical" block-submatrix). The total computation time performed is $\max\{L, n^2/p^{2/3}\}$.

Proposition 3 *Algorithm* MULT_C *for multiplying two $n \times n$ matrices A and B stored according to the block distribution requires, for any $p \leq n^2$, computation time $C_{mul}(n)$ that is given by*

$$C_{mul}(n) \leq \max\{L, 2n^3/p\} + \max\{L, n^2/p^{2/3}\},$$

and communication time $M_{mul}(n)$ that is given by the expression

$$M_{mul}(n) = \max\left\{L, 2g\frac{n^2}{p^{2/3}}\right\} + \max\left\{L, g\frac{n^2}{p^{2/3}}\right\}.$$

The optimality in communication of the algorithm is established by the following result.

Theorem 1 *On a model of computation that allows the operations $\{+, *\}$ only, if any processor reads s elements of A and B and computes at most s partial sums of C, it can compute at most $O(s^{3/2})$ multiplicative terms for these sums.*

This way, if a processor reads at most s elements of A and B it can compute at most $O(s^{3/2})$ multiplicative terms of C. Combined, all p processors can compute $p\ O(s^{3/2})$ such terms which must be $\Omega(n^3)$. Therefore $s = \Omega(n^2/p^{2/3})$ and thus algorithm MULT_C is communication optimal.

- A problem with a long history

- Sequential Sorting

  - QuickSort [C.A.R Hoare, 1962]
  - SampleSort[Frazer and McKellar 1970]

- Parallel Randomized Sorting

  - Sample-based [Reischuk 1985]
  - Oversampling [Reif and Valiant 1987]

- Parallel Deterministic Sorting with regular sampling [Shi and Schaeffer 1992])

- A. SELECTSAMPLE

- B. SORTSAMPLE

- C. SELECTSPLITTERS

- D. DISSEMINATESPLITTERS

- E. DIVIDEKEYSBASEDONSPLITTERS

- F. REARRANGEDIVIDEDKEYS

- G. ARRANGERECEIVINGKEYS

- H. SORTLOCALKEYS

Randomized Sorting	Deterministic Sorting
A. SELECTRANDOMSAMPLE	H. SORTINPUTKEYS
B. SORTSAMPLE	A. SELECTDETSAMPLE
C. SELECTSPLITTERS	B. SORT(MERGE)SAMPLE
	C. SELECTSPLITTERS
D. BROADCASTSPLITTERS	D. BROADCASTSPLITTERS
E. BINARYSEARCHKEYSONSPLITTERS	E. MERGEKEYSANDSPLITTERS
F. ROUTEKEYSTOPROCESSORS	F. ROUTEKEYSTOPROCESSORS
H. SORTRECEIVINGKEYS	G. MERGERECEIVINGKEYS

- **Input:** n keys evenly but arbitrarily distributed.

A. A sample of size $ks - 1 < n/2$ is selected at random.

B. The sample of size $ks - 1$ is sorted.

C. Every s-th key of the sample constitutes a splitter.

D and E. The position of each input key with respect to the $k - 1$ splitters is determined by binary search.

F. Route keys to processors.

H. Sort the k bucket.

Remark 1. If $ps < n/p$ choose $k = p$; otherwise choose $k = p^{1/r}$, algorithm consists of r iterations, each one repeating steps A-F for a subset of the processors.

Remark 2. Depending on how various steps are performed, various algorithms can be presented; sorting in stage B, for example, can use a sequential or parallel (Batcher sort) method.

- Only important terms are included.

- Assume $p << n$ so that $k = p$.

- S_D, S_R are the constants of the corresponding sequential algorithms. B is constant for binary search, M for merging, R for preprocessing the keys to be routed to achieve coarse-grained communication in F.

- $Rand\_Alg = B\frac{n}{p} \lg p + S_R\frac{n}{p} \lg \frac{n}{p} + (R + g)\frac{n}{p} + O(L) + o(n \lg n/p)$

  - It requires fast fine-grained communication.

- $Det\_Alg = M\frac{n}{p} \lg p + S_D\frac{n}{p} \lg \frac{n}{p} + (2 + g)\frac{n}{p} + O(L) + o(n \lg n/p)$

  - if $p << n/p$ term $2\frac{n}{p}$ becomes $p \lg \frac{n}{p}$

- One-optimal randomized BSP sorting [*Valiant and Gerbessiotis 94*]

1. If $\frac{n}{p} = n^t$, for any constant $0 < t < 1$, there is a randomized BSP sorting algorithm with

   Computation: $(1 + o(1))\frac{T_{seq}}{p}$

   Communication: $O(g\frac{n}{p})$

2. If $\frac{n}{p} = \log^{1+a} n$, for any constant $a > 0$, there is a randomized BSP sorting algorithm with

   Computation: $(1 + o(1))\frac{T_{seq}}{p}$

   Communication: $O(g\frac{n \lg n}{p \lg\lg n})$

T_{seq} is the time of an $O(n \lg n)$ sequential algorithm on a comparison-exchange model.

A (New??) Randomized BSP Algorithm

- H. SORTINPUTKEYS

- A. SELECTRANDOMSAMPLE

- B. SORT(MERGE)SAMPLE

- C. SELECTSPLITTERS

- D. BROADCASTSPLITTERS

- E. MERGEKEYSANDSPLITTERS

- F. ROUTEKEYSTOPROCESSORS

- G. MERGERECEIVINGKEYS

Consequences for Sequential Sorting

- Serialize parallel randomized algorithm (ignore communication, multiply by p parallel time)

- $Rand\_Algorithm = p(B\frac{n}{p}\lg p + \mathbf{S_R}\frac{n}{p}\lg\frac{n}{p})$

- $Sequential\_Algorithm = \mathbf{S}n\lg n$

- Let $p = n/t$ be the number of splitters for a sequential randomized sorting algorithm ($\lg t = o(\lg n), t = \Omega(\lg n)$). Assume that in general $1 < B \ll S$.

$$
\begin{aligned}
Rand\_Sequential &= Bn\lg p + Sn\lg\frac{n}{p} + S\frac{n}{t}\lg n/t \\
&= \mathbf{B}n\lg n + (S - B)n\lg t + o(n\lg n)
\end{aligned}
$$

- The constant of the Randomized algorithm is $\mathbf{B}$ instead of $\mathbf{S}$.

- It makes merge-sort and heap-sort (the corresponding andomized variants) as fast as quick sort.

- It offers a better abstraction of a parallel computer that allows the study of the computation, communication and synchronization requirements of parallel algorithms

- As it is not just an abstract computational model, it can also be used as a programming paradigm for the programming of parallel computers; experimental uses of BSP oriented libraries show its practicality as well.

- A BSP algorithm is a collection of algorithms; depending on n, the problem size, and the BSP parameters (p, L, g) of a particular machine, that member of the collection with the best performance is selected.

- Many of the abstractions included in BSP are reflected in the Virtual Interface Architecture Specification for System Area Networks by Compaq, Intel and Microsoft.