The BSP Model

- **BSP**: Bulk-Synchronous Parallel
- BSP is designed to be architecture independent
  - Portable programs
- BSP considers at a global level (bulk) computation and communication
- Execution time of a BSP program is computed by the local execution time and from few parameters tied to the particular architecture that is used

**The BSP Programming Style**

- Vertical Structure
  - Sequential composition of “supersteps”
    - Local computation
    - Process Communication
    - Barrier Synchronization
- Horizontal Structure
  - Concurrency among a fixed number of virtual processors.
  - Processes do not have a particular order
  - Locality plays no role in the placement of processes on processors
  - \( p = \) number of processors

**The BSP Model**

- In BSP, each processor has local memory
- “One-sided” communication style is advocated
- There are globally-known “symbolic addresses”
- Data may be inconsistent until next barrier synchronization

**The BSP Model**

- Computational model of parallel computation
- The model consists of:
  - A set of processor-memory pairs
    - Number of processors in model can be greater than number of processors of machine
  - A communications network that delivers messages in a point-to-point manner
  - A mechanism for the efficient barrier synchronization for all or a subset of the processes
- There are no special combining, replicating, or broadcasting facilities

**The BSP Model**

- BSP programs are composed of supersteps
- In each superstep, processors execute computational steps using locally stored data, and also can send and receive messages
- Processors synchronize at end of superstep (at which time all messages have been received)
- Oxford BSP is a library of C routines for implementing BSP programs. It provides:
  - Direct Remote Memory Access
  - Bulk Synchronous Message Passing (sort of like nonblocking message passing in MPI)
The BSP Model

- The BSP computer is a MIMD system
- It is loosely synchronous at the superstep level
  - While the PRAM model was synchronous at instruction level
- Within a superstep, different processes execute asynchronously at their own paces

The BSP Model

- The BSP model is more realistic than the PRAM model because it accounts for all overheads except the parallelism overhead for process management
- The execution time of a superstep is determined by the local computation, the communication and the synchronization

The BSP Model – \( w \)

- To account for load imbalance, the computation time \( w \) is the maximum time spent on computation operations by any processor

The BSP Model

- MIMD Superstep:
  - Computation
  - Communication
  - Barrier
- Variable grain
- Loosely synchronous
- Nonzero overhead
- Message passing or shared variable

The BSP Model

- The BSP model does not require any specific memory interaction mechanism
- Within a superstep, each computation operation uses only data in its local memory
- These data are put into the local memory either at the program start—up time or by the communication operations of previous supersteps
### The BSP Model

- A communication is always realized in a point-to-point manner
  - Thus it is not allowed for multiple processes to read or write the same memory location in the same cycle
- All memory and communication operations in a superstep must completely completely finish before any operation of the next superstep begins

### The BSP Model – \( gh \)

- Parameter \( g \) measures the permeability of the network to continuous traffic addressed to uniformly random destinations
  - The parameter \( g \) is defined such that an \( h \)-relation will be delivered in time \( gh \)
  - The communication overhead is \( gh \) cycles, where \( g \) is the proportional coefficient for realizing an \( h \) relation
- The value of \( g \) is platform-dependent, but independent of the communication pattern
  - In other words, \( gh \) is the time to execute the most time-consuming \( h \) relation

### The BSP Model – \( h \)

- The BSP model abstracts the communication operations in a BSP superstep by the \( h \)-relation concept
- An \( h \)-relation is an abstraction of any communication operation, where each node sends at most \( h \) words to various nodes and each node receives at most \( h \) words

### Communication

- BSP considers communication *en masse*
  - Makes it possible to bound the time to deliver a whole set of data by considering all the communication actions of a superstep as a unit
  - If the maximum number of incoming or outgoing messages per processor is \( h \), then such a communication pattern is called an \( h \)-relation

### The BSP Model – \( l \)

- The synchronization overhead is \( l \), which has a lower bound of the communication network latency (i.e., the time for a word to propagate through the physical network) and is always greater than zero
Barrier

- “Often expensive and should be used as sparingly as possible”
  - Developers of BSP claim that barriers are not as expensive as they are believed to be in high performance computing folklore
- The cost of a barrier synchronization has two parts
  - The cost caused by the variation in the completion time of the computation steps that participate
  - The cost of reaching a globally-consistent state in all processors
- The parameter $l$ captures the latter of these costs
  - Lower bound on $l$ is the diameter of the network
  - However, it is also affected by many other factors, so that, in practice, an accurate value of $l$ for each parallel architecture is obtained empirically

The BSP Model

- The BSP model allows the overlapping of the computation, the communication, and the synchronization operations within a superstep
- If all three types of operations are fully overlapped, the time for a superstep becomes $\max(w, gh, l)$
  - However, the more conservative $w + gh + l$ is typically used

The BSP Model

- $h$: communication time
- $w$: computation time
- $l$: synchronization time
- $gh$: communication overhead
- The time for a superstep is estimated by the sum $w_i + gh_i + l$

Example

- Algorithm to compute the maximum of a n-elements array
  - On a BSP, since there is no shared memory, we have to say where the data are
    - $A[0..n-1]$ distributed block-wise across $p$ processors
    - For instance, each processor can have a portion of the array
      - $\pi/p$ elements
  - To describe an algorithm on a BSP machine, we have to define all supersteps
    - Local computing operations
    - Communication operations
    - Synchronization barrier

The BSP Model

- $h$: communication time
- $w$: computation time
- $l$: synchronization time
- $gh$: communication overhead
- The time for a superstep is estimated by the sum $\max w_i + \max gh_i + l$

Maximum

Try to design the algorithm on a $p$ processors BSP machine
Maximum

- Superstep1
  - Local computation phase
    - for all $A[i]$ in my local partition of $A$, $M = \max(m, A[i])$;
    - Communication phase:
      - if $myPID != 0$ send ($m$, 0);
      - else // on P0:
        - for each $i$ in $\{1..p-1\}$ recv ($m$, $i$);
    - Superstep2
      - if $myPID = 0$ for each $i$ in $\{1..p-1\}$ $m = \max(m, m_i)$ [TIME???]

Maximum

- Superstep1
  - Local computation phase [n/p]
    - for all $A[i]$ in my local partition of $A$, $M = \max(m, A[i])$;
  - Communication phase [gh, with $h=p-1$ (WHY???)]
    - if $myPID != 0$ send ($m$, 0);
    - else // on P0:
      - for each $i$ in $\{1..p-1\}$ recv ($m$, $i$);
  - Superstep2
    - if $myPID = 0$ for each $i$ in $\{1..p-1\}$ $m = \max(m, m_i)$ [TIME???]
Maximum

- Total
  \[ \Theta(n/p + g(p-1) + l + p) = \Theta(n/p + gp + l) \]

Example

Algorithm for inner-product using 8-processor BSP computer in 4 supersteps (“small” communication):

- **Superstep 1**
  - Computation: ????
  - Communication: ????
  - Barrier synchronization

Example

- Algorithm for inner-product with 8 processors
- Given two arrays \( x \) and \( y \), we want to compute \( \sum x_i y_i \)
- In a BSP program, it is crucial to define how data are split among processors
  - For instance, in this example, **how can we divide the vectors' elements??**

Example

Algorithm for inner-product using 8-processor BSP computer in 4 supersteps (“small” communication):

- **Superstep 1**
  - Computation: Each processor computes its local sum in \( w = 2N/8 \) time
  - Communication: Processors 0, 2, 4, 6 send their local sums to processors 1, 3, 5, 7
    - Apply 1-relation here
  - Barrier synchronization

Example

- Algorithm for inner-product with 8 processors
- Given two arrays \( x \) and \( y \), we want to compute \( \sum x_i y_i \)
- In a BSP program, it is crucial to define how data are split among processors
  - For instance, in this example, the vectors' elements can be divided cyclically or in blocks

<table>
<thead>
<tr>
<th>Cyclic:</th>
<th>P0</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block:</td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P0</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
</tr>
</tbody>
</table>

In any case, it is better having both \( x \) and \( y \) on the same processor!!

Example

- **Superstep 2**
  - Computation: ????
  - Communication: ????
  - Barrier synchronization
Example

- **Superstep 2**
  - Computation: Processors 1, 3, 5, 7 each perform one addition ($w = 1$)
  - Communication: Processors 1 and 5 send their intermediate results to processors 3 and 7
    - 1-relation is applied here
  - Barrier synchronization

Example

- **Superstep 3**
  - Computation: ????
  - Communication: ????
  - Barrier synchronization

Example

- **Superstep 4**
  - Computation: ????
  - Communication: ????

Example

- **Superstep 3**
  - Computation: Processors 3 and 7 each perform one addition ($w = 1$)
  - Communication: Processor 3 sends its intermediate result to processor 7
    - Apply 1-relation here
  - Barrier synchronization

Example

- **Superstep 4**
  - Computation: Processor 7 performs one addition ($w = 1$) to generate the final sum
  - No more communication or synchronization is needed

Example

- The total execution time (8 processors) is ????
**Example**

- The total execution time is $2N/8 + 3g + 3l + 3$ cycles.
- In general, the execution time is $2N/p + (g+l+1)\log p$ supersteps on a $p$-processor BSP.

**Alternative Solution**

- With a constant number of supersteps.

  **How do we proceed?**

**Example**

- The total execution time (8 processors) is $2N/8 + 3g + 3l + 3$ cycles.
- In general, the execution time is $2N/p + (g+l+1)\log p$ cycles on a $p$-processor BSP.

  - **How much is the parallel time on PRAM computer with $p$ processors?**

**Alternative Solution**

- With a constant number of supersteps.
- How do we proceed?
  - Each processor computes locally its values.
  - Each processor broadcasts to all the others its computed value.
  - All values can now compute locally the final value.

  - All processors in this solution has the final value.

  **Cost????**

**Example**

- The total execution time is $2N/8 + 3g + 3l + 3$ cycles.
- In general, the execution time is $2N/p + (g+l+1)\log p$ cycles on an $n$-processor BSP.
  - This is in contrast to the time $2N/p + \log p$ on a PRAM computer.
- The two extra terms, $\log p$ and $l \log p$ correspond to communication and synchronization overheads, respectively.

**Alternative Solution**

- With a constant number of supersteps.
- How do we proceed?
  - Each processor computes locally its values.
  - Each processor broadcasts to all the others its computed value.
  - $gh$, with $h = ?????$.
  - All processors can now compute locally the final value.

  - All processors in this solution has the final value.

  **O(????)**
Alternative Solution

- With a constant number of supersteps
- How do we proceed?
  - Each processor computes locally its values
    - $O(N/p)$
  - Each processor broadcasts to all the others its computed value
    - $gh$, with $h = p-1$
  - All processors can now compute locally the final value
    - All processors in this solution has the final value
    - $O(p)$

Matrix Multiplication – PRAM

- Each element of $C$ can be computed in parallel using $n$ processors on a CREW PRAM
  - $O(\log n)$ time
    - Basically, it's a SUM in parallel
- All $c_{ij}$ can be computed in parallel using $n^3$ processors in $O(\log n)$ time

Matrix Multiplication

- We want to multiply two matrices, $A$ and $B$
  - $A_{max} \times B_{max} = C_{max}$
- The standard algorithm uses $p \leq n^2$ processors
  - If $p = n^2$, then each processor can compute the value of a single element in $C$
    - Already seen in the PRAM solution

Matrix Multiplication

- In the BSP model we need to find a way of dividing the input among processors, and to optimize the communication

Ideas????
Matrix Multiplication

- In the BSP mode we need to find a way of dividing the input among processors, and to optimize the communication.
- To each processor we assign the sub-problem of computing a sub-matrix of C, of size \( \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} \).
- Thus, each processor receives in input \( \frac{n}{\sqrt{p}} \) rows of A and \( \frac{n}{\sqrt{p}} \) columns of B.

Matrix Multiplication

Let \( n = 4 \)

\[
A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix}
\]

\[
C = A \times B = \begin{bmatrix}
c_{11} & c_{12} & c_{13} & c_{14} \\
c_{21} & c_{22} & c_{23} & c_{24} \\
c_{31} & c_{32} & c_{33} & c_{34} \\
c_{41} & c_{42} & c_{43} & c_{44}
\end{bmatrix}
\]

Matrix Multiplication

Let \( p = 4 \) (\( p_1, p_2, p_3, p_4, p < n^2 \))

- Each processor computes \( \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} = \frac{n^2}{p} \) elements of C.
- Thus, each processor receives in input \( \frac{n}{\sqrt{p}} \) rows of A and \( \frac{n}{\sqrt{p}} \) columns of B.

Matrix Multiplication

Let \( p = 4 \) (\( p_1, p_2, p_3, p_4, p < n^2 \))

- Each processor computes \( \frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}} = \frac{n^2}{p} \) elements of C.
- Thus, each processor receives in input \( \frac{n}{\sqrt{p}} \) rows of A and \( \frac{n}{\sqrt{p}} \) columns of B.
Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say $p_i$.
  - How many row-by-column inner products $p_i$ does perform locally?

```latex
\begin{align*}
\text{Let us compute with input} & \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix} \\
\text{with input} & \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix} \\
\text{computed} & \begin{pmatrix} e_{11} & e_{12} & \cdots & e_{1n} \\ e_{21} & e_{22} & \cdots & e_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \cdots & e_{mn} \end{pmatrix} \\
\text{by a processor, say } p_i & \end{align*}
```

Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say $p_i$.
  - Given a local row and a local column of $p_i$.
    - How many sums does it perform?
    - How many multiplications does it perform?

```latex
\begin{align*}
\text{Let us compute with input} & \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix} \\
\text{with input} & \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix} \\
\text{computed} & \begin{pmatrix} e_{11} & e_{12} & \cdots & e_{1n} \\ e_{21} & e_{22} & \cdots & e_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \cdots & e_{mn} \end{pmatrix} \\
\text{by a processor, say } p_i & \end{align*}
```

Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say $p_i$.
  - Given a local row and a local column of $p_i$.
    - How many sums does it perform?
    - How many multiplications does it perform?

```latex
\begin{align*}
\text{Let us compute with input} & \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix} \\
\text{with input} & \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix} \\
\text{computed} & \begin{pmatrix} e_{11} & e_{12} & \cdots & e_{1n} \\ e_{21} & e_{22} & \cdots & e_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \cdots & e_{mn} \end{pmatrix} \\
\text{by a processor, say } p_i & \end{align*}
```

Matrix Multiplication

- Let us compute the number of local operations performed by a processor, say $p_i$.
  - Summing over all inner products performed by $p_i$.
    - How many sums does it perform?
    - How many multiplications does it perform?

```latex
\begin{align*}
\text{Let us compute with input} & \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \times \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix} \\
\text{with input} & \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1n} \\ c_{21} & c_{22} & \cdots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mn} \end{pmatrix} \\
\text{computed} & \begin{pmatrix} e_{11} & e_{12} & \cdots & e_{1n} \\ e_{21} & e_{22} & \cdots & e_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ e_{m1} & e_{m2} & \cdots & e_{mn} \end{pmatrix} \\
\text{by a processor, say } p_i & \end{align*}
```
Let us compute the number of local operations performed by a processor, say $p_4$.

- Summing over all inner products performed by $p_4$:
  - How many sums does it perform?
    - $n \times n/\sqrt{p} \times n/\sqrt{p}$
  - How many multiplications does it perform?
    - $n \times n/\sqrt{p} \times n/\sqrt{p}$

$p_4$ computes $c_{33} c_{34} c_{43} c_{44}$ with input $a_{31} a_{32} a_{33} a_{34}$ $a_{41} a_{42} a_{43} a_{44}$ $b_{13} b_{14} b_{23} b_{24}$ $b_{33} b_{34} b_{43} b_{44}$

Thus, each processor executes locally $n^3/p$ sums + $n^3/p$ multiplications.
- That is, $2n^3/p$ operations
  - $2 \times n \times n/\sqrt{p} \times n/\sqrt{p} = 2n^3/p$

Now, let us analyze the complexity of the communication phase.
- In order to execute its local operations, how many messages does each processor needs to receive?

$p_4$ computes $c_{33} c_{34} c_{43} c_{44}$ with input $a_{31} a_{32} a_{33} a_{34}$ $a_{41} a_{42} a_{43} a_{44}$ $b_{13} b_{14} b_{23} b_{24}$ $b_{33} b_{34} b_{43} b_{44}$

In general, each processor has to send each one of its local values to how many processors?
Matrix Multiplication

In general, each processor has to send each one of its local values to how many processors?

\*\( \sqrt{p} \)

Thus, the number of transmissions, for each processor, is this number of messages: \((2n^2/p) \times \sqrt{p} = 2n^2/\sqrt{p} \)

The cost of this BSP algorithm is

\[ - n^3/p + (n^2/p)^{3/2}g + l \]

The optimal cost \( O(n^3/p) \), with \( n^2/p \) memory for each processor, is achieved when

\[ - g = O(n/p^{1/3}) \]
\[ - l = O(n^3/p) \]

There exists a more sophisticated algorithm, by McColl and Valiant, that solves the problem with less messages

\[ - n^3/p + (n^2/p^{3/2})g + l \]

That is optimal when

\[ - g = O(n/p^{1/3}) \]
\[ - l = O(n^3/(p \log n)) \]
The BSP Model

• Properties:
  – Simple to write programs.
  – Independent of target architecture.
  – Performance of the model is predictable.

  • Considers computation and communication at the level of the entire program and executing computer instead of considering individual processes and individual communications.

The BSP Model

• Strategies used in writing efficient BSP programs:
  – Balance the computation in each superstep between processes
    • “w” is a maximum of all computation times and the barrier synchronization must wait for the slowest process
  – Balance the communication between processes
    • “h” is a maximum of the fan-in and/or fan-out of data
  – Minimize the number of supersteps
    • Determines the number of times the parallel slackness appears in the final cost.

The BSP Model

• Supports a SPMD style of programming.
• Library is available in C and FORTRAN
• Implementations available (several years ago) for
  – Cray T3E
  – IBM SP2
  – SGI PowerChallenge
  – Convex Exemplar
  – Hitachi SR2001
  – Various Workstation Clusters

  • Allows for direct remote memory access or message passing
  • Includes support for unbuffered messages for high performance computing

Summary

• BSP is a computational model of parallel computing based on the concept of supersteps
• BSP does not use locality of reference for the assignment of processes to processors
• Predictability is defined in terms of three parameters
• BSPlib has a much smaller API as compared to MPI/PVM

Summary

• BSP can be regarded as a generalization of the PRAM model
  – For which values of which parameters of the model we have that the BSP is really close to the PRAM????

Summary

• BSP can be regarded as a generalization of the PRAM model.
  – If the BSP architecture has a small value of $g (g=1)$, then it can be regarded as PRAM
• Use hashing to automatically achieve efficient memory management
• The value of $l$ determines the degree of parallel slackness required to achieve optimal efficiency
  – If $l = g = 1$, corresponds to idealized PRAM where no slackness is required
**LogP**

- Developed by Culler et al from Berkeley
- BSP differs from LogP in three ways
  - LogP uses a form of message passing based on pairwise synchronization
  - LogP adds an extra parameter representing the overhead involved in sending a message
    - Applies to every communication!
  - LogP defines $g$ in local terms
    - It regards the network as having a finite capacity and treats $g$ as the minimal permissible gap between message sends from a single process
    - The parameter $g$ in both cases is the reciprocal of the available per-processor network bandwidth: **BSP takes a global view of $g$, LogP takes a local view of $g$**

**L:** latency – time for message to go from $P_{\text{sender}}$ to $P_{\text{receiver}}$

**o:** overhead - time either processor is occupied sending or receiving message
- Processor can’t do anything else for $o$ cycles

**g:** gap - minimum time between messages
- Processor can have at most $L/g$ messages in transit at a time
- Gap includes overhead time (so overhead ≤ gap)

$P$: number of processors

$L$, $o$, and $g$ are measured in cycles

**Broadcasting**

Supponiamo che un processore voglia inviare la copia di un messaggio ad $n$ istanze di memoria, uniformemente distribuite su $p$ componenti.

Spedire una copia a ciascuna delle $p$ componenti può essere effettuato attraverso un albero d-ario logico, in $\log_d p$ passi.

- Ad ogni superpasso, ogni processore coinvolto trasmette $d$ copie a componenti distinte. Il tempo richiesto è quindi $d \log_p p$.

**LogP**

- When analyzing the performance of LogP model, it is often necessary (or convenient) to use barriers
- Message overhead is present but decreasing
  - Only overhead is from transferring the message from user space to a system buffer
- LogP + barriers - overhead = BSP
- Both models can efficiently simulate the other