Multi-BSP computing: the next step?

3rd of December 2014
Albert-Jan Yzelman
Outline

1. The Multi-BSP model
2. Philosophy
3. Algorithm design
The Multi-BSP model

1. The Multi-BSP model
2. Philosophy
3. Algorithm design
Three concepts

The Bulk Synchronous Parallel

1 computer,
2 algorithm,
3 cost model.
BSP computer
CPUs, memory, network.

A BSP computer \((p, g, l)\).
BSP algorithm

- computations are grouped into **phases**,
- **no communication** during computation,
- communication is only allowed **between** computation phases.
BSP cost model

The **cost of computation** during the $i$th superstep is

$$T_{\text{comp},i} = \max_{s} w_{i}^{(s)}.$$  

The total **cost of communication** during the $i$th superstep is

$$T_{\text{comm},i} = h_{ig}.$$  

Adding up superstep costs, separated by the latency $l$, yields the **full BSP cost**:

$$T = \sum_{i=0}^{N-1} \left( T_{\text{comp},i} + T_{\text{comm},i} + l \right) = \sum_{i=0}^{N-1} \left( \max_{s} w_{i}^{(s)} + h_{ig} + l \right),$$
Multi-BSP

Multi-BSP is the recursive application of the BSP model.

- **BSP**: a computer consists out of \( p \) CPUs/processors/cores/...
- **Multi-BSP**: a computer consists out of
  1. \( p \) other Multi-BSP subcomputers (recursively), or
  2. \( p \) units of execution (leaves).
- Each Multi-BSP computer:
  - connects its subcomputers or leaves via a network, and
  - provides local memory.

Reference:

Multi-BSP computer model

CPUs, memory, network.

A BSP computer \((p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})\).
Multi-BSP computer model

CPUs, memory, network.

A BSP computer \((p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})\).
Multi-BSP computer model

CPUs, memory, network.

A BSP computer \((p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})\).
Multi-BSP computer model

CPUs, memory, network.

A BSP computer \((p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})\).
Multi-BSP computer model
Multi-BSP computer model
Multi-BSP algorithm model

This change of computer model changes the algorithmic model:

- only **local communication** allowed using the local \((l_k, g_k)\),
- local memory requirements do not exceed the **local memory** \(M_k\),
- ‘local’ is given by the **current tree level** \(k\).
Multi-BSP cost model

We require extra notation:

- \( L \): number of levels in the tree,
- \( N_i \): number of supersteps on the \( i \)th level,
- \( h_{k,i} \): the maximum of all \( h \)-relations within the \( i \)th superstep on level \( k \),
- \( w_{k,i} \): the maximum of all work within the \( i \)th superstep on level \( k \).

The decomposability of Multi-BSP algorithms, just as with the ‘flat’ BSP model, again results in a transparent cost model:

\[
T = \sum_{k=0}^{L-1} \left( \sum_{i=0}^{N_k-1} w_{k,i} + h_{k,i}g_k + l_k \right). 
\]
Half-time summary

- Multi-BSP is a **better model** for modern parallel architectures. It closely resembles
  - contemporary shared-memory multi-socket machines,
  - multi-level shared and private cache architectures, and
  - multi-level network topologies (e.g., fat trees).

- Hierarchical modeling also has **drawbacks**. It is more difficult to
  - **prove optimality** of hierarchical algorithms, and
  - **portably** implement hierarchical algorithms.

Would you like to:

- prove optimality of an algorithm in 16 parameters?
- develop algorithms for a four-level machine?
Half-time summary

– Multi-BSP is a better model for modern parallel architectures. It closely resembles
  • contemporary shared-memory multi-socket machines,
  • multi-level shared and private cache architectures, and
  • multi-level network topologies (e.g., fat trees).

– Hierarchical modeling also has drawbacks. It is more difficult to
  • prove optimality of hierarchical algorithms, and
  • portably implement hierarchical algorithms.

Would you like to:

– prove optimality of an algorithm in 16 parameters?
– develop algorithms for a four-level machine?

Multi-BSP can actually simplify these issues!
Philosophy

1. The Multi-BSP model
2. Philosophy
3. Algorithm design
Motivation

Reasons for parallel computing:
① to speed up a long computation, or
② to split up a huge computation.

That is,

we wish to scale in time and memory.
Definition (Speedup)

Let $T_{\text{seq}}$ be the sequential running time required for solving a problem. Let $T_p$ be the running time of a parallel algorithm using $p$ processes, solving the same problem. Then the speedup is given by

$$S = \frac{T_{\text{seq}}}{T_p}.$$ 

Scalable in time:

- Ideally, $S = p$;
- if we are lucky, $S > p$;
- realistically, $1 \ll S < p$;
- if we do very badly, $S < 1$. 

Speedup
Speedup

Definition (Speedup)

Let $T_{\text{seq}}$ be the sequential running time required for solving a problem. Let $T_p$ be the running time of a parallel algorithm using $p$ processes, solving the same problem. Then the speedup is given by

$$S = \frac{T_{\text{seq}}}{T_p}.$$ 

Scalable in time:

Ideally, $S = p$;
if we are lucky, $S > p$;
realistically, $1 \ll S < p$;
if we do very badly, $S < 1$.

But what is a good speedup?
Maximum attainable speedup

Consider a graph $G = (V, E)$ of a given algorithm, e.g.,

- Nodes correspond to data, edges indicate which data is combined to generate a certain output.

**Question:** If we had an infinite number of processors, how fast would we be able to run the algorithm shown on the right?
Maximum attainable speedup

Consider a graph $G = (V, E)$ of a given algorithm, e.g.,

- Nodes correspond to data, edges indicate which data is combined to generate a certain output.

**Question:** If we had an infinite number of processors, how fast would we be able to run the algorithm shown on the right?

**Answer:** $T_{\text{seq}} = |V| = 9$, while the critical path length $T_\infty$ equals 4.

The maximum speedup hence is:

$$T_{\text{seq}} / T_\infty = 9/4.$$
What is parallelism?

**Definition (Parallelism)**

The *parallelism* of a given algorithm is its maximum attainable speedup:

\[ T_{\text{seq}} / T_{\infty}. \]

\( T_{\infty} \) is known as the **critical path length** or the **algorithmic span**.

This leads to a theoretical **upper bound on speedup**:

\[ S = T_{\text{seq}} / T_p \leq T_{\text{seq}} / T_{\infty}. \]

This type of analysis forms the basis of **fine-grained parallel computation**.
Fine-grained parallel computing

Decompose a problem into many small tasks, that run **concurrently** (as much as possible). A **run-time scheduler** assigns tasks to processes.

- What is small? **Grain-size**.
- Performance model? **Parallelism**.

Algorithms can be implemented as graphs either explicitly or **implicitly**:
- Intel: Threading Building Blocks (TBB),
- **OpenMP**,
- Intel / MIT / Cilk Arts: **Cilk**,
- Google: **Pregel**,
- ...  

By contrast, BSP computing is **coarse-grained**.
Example via SpMV multiplication. Assuming a Hilbert nonzero ordering:

$$A = \begin{pmatrix}
4 & 1 & 3 & 0 \\
0 & 0 & 2 & 3 \\
1 & 0 & 0 & 2 \\
7 & 0 & 1 & 1
\end{pmatrix}$$

COO (triplet) storage:

$$A = \begin{cases}
V & [7 1 4 1 2 3 3 2 1 1] \\
J & [0 0 0 1 2 2 3 3 3 2] \\
I & [3 2 0 0 1 0 1 2 3 3]
\end{cases}$$

for $k = 0$ to $nz - 1$ do

add $V_k \cdot x_{J_k}$ to $y_{I_k}$
OpenMP

Example via SpMV multiplication. Assuming a Hilbert nonzero ordering:

\[
A = \begin{pmatrix}
4 & 1 & 3 & 0 \\
0 & 0 & 2 & 3 \\
1 & 0 & 0 & 2 \\
7 & 0 & 1 & 1
\end{pmatrix}
\]

COO (triplet) storage: \( T_{\text{seq}} / T_\infty = 2nz \ldots ? \)

\[
A = \begin{cases}
V & [7 \ 1 \ 4 \ 1 \ 2 \ 3 \ 3 \ 2 \ 1 \ 1] \\
J & [0 \ 0 \ 0 \ 1 \ 2 \ 2 \ 3 \ 3 \ 3 \ 2] \\
I & [3 \ 2 \ 0 \ 0 \ 1 \ 0 \ 1 \ 2 \ 3 \ 3]
\end{cases}
\]

\#omp parallel for private( k ) schedule( dynamic, 8 )

for \( k = 0 \) to \( nz - 1 \) do

add \( V_k \cdot x_{J_k} \) to \( y_{I_k} \)
OpenMP

Example via SpMV multiplication. Assuming a Hilbert nonzero ordering:

\[
A = \begin{pmatrix}
4 & 1 & 3 & 0 \\
0 & 0 & 2 & 3 \\
1 & 0 & 0 & 2 \\
7 & 0 & 1 & 1 \\
\end{pmatrix}
\]

COO (triplet) storage: data race! (concurrent writes to the same \(y_i\))

\[
A = \begin{cases}
V & [7 \ 1 \ 4 \ 1 \ 2 \ 3 \ 3 \ 2 \ 1 \ 1] \\
J & [0 \ 0 \ 0 \ 1 \ 2 \ 2 \ 3 \ 3 \ 3 \ 2] \\
I & [3 \ 2 \ 0 \ 0 \ 1 \ 0 \ 1 \ 2 \ 3 \ 3] \\
\end{cases}
\]

#pragma omp parallel for private(k) schedule(dynamic, 8)

for \(k = 0\) to \(nz - 1\) do

add \(V_k \cdot x_{J_k}\) to \(y_{I_k}\)
OpenMP

Example via SpMV multiplication. Assuming row-major nonzero ordering:

\[
A = \begin{pmatrix}
4 & 1 & 3 & 0 \\
0 & 0 & 2 & 3 \\
1 & 0 & 0 & 2 \\
7 & 0 & 1 & 1 \\
\end{pmatrix}
\]

CRS storage:

\[
A = \begin{cases}
V & [4 1 3 2 3 1 2 7 1 1] \\
J & [0 1 2 2 3 0 3 0 2 3] \\
\hat{I} & [0 3 5 7 10]
\end{cases}
\]

\[
\text{for } i = 0 \text{ to } m - 1 \text{ do} \\
\text{for } k = \hat{I}_i \text{ to } \hat{I}_{i+1} - 1 \text{ do} \\
\text{add } V_k \cdot x_{J_k} \text{ to } y_i
\]
OpenMP

Example via SpMV multiplication. Assuming row-major nonzero ordering:

\[
A = \begin{pmatrix}
4 & 1 & 3 & 0 \\
0 & 0 & 2 & 3 \\
1 & 0 & 0 & 2 \\
7 & 0 & 1 & 1
\end{pmatrix}
\]

CRS storage: \( T_{\text{seq}}/T_\infty = nz/\max_i |a_i| \) (with \( a_i \): the \( i \)th row of \( A \))

\[
A = \begin{cases}
V & [4 \ 1 \ 3 \ 2 \ 3 \ 1 \ 2 \ 7 \ 1 \ 1] \\
J & [0 \ 1 \ 2 \ 2 \ 3 \ 0 \ 3 \ 0 \ 2 \ 3] \\
\hat{I} & [0 \ 3 \ 5 \ 7 \ 10]
\end{cases}
\]

```c
#pragma omp parallel for private(i, k) schedule(dynamic, 8)
for i = 0 to m - 1 do
    for k = \hat{I}_i to \hat{I}_{i+1} - 1 do
        add \( V_k \cdot x_{J_k} \) to \( y_i \)
```
Cilk

Only two parallel programming primitives:

1. (binary) **fork**, and
2. (binary) **join**.
Only two parallel programming primitives:

1. (binary) **fork**, and
2. (binary) **join**.

Example: calculate \(x_4\) from \(x_n = x_{n-2} + x_{n-1}\) given \(x_0 = x_1 = 1\):

```
int f(int n) {
    if (n == 0 || n == 1) return 1;
    int x1 = cilk_spawn f(n-1); //fork
    int x2 = cilk_spawn f(n-2); //fork
    cilk_sync; //join
    return x1 + x2;
}

int main() {
    int x4 = f(4);
    printf("x_4 = %d\n", &x4);
    return 0;
}
```
Cilk

Only two parallel programming primitives:
1. (binary) **fork**, and
2. (binary) **join**.

**Definition (Overhead)**

The **overhead** of parallel computation is any extra effort expended over the original amount of work $T_{seq}$

$$T_o = pT_p - T_{seq}.$$ 

The parallel computation time can be expressed in $T_o$:

$$T_p = \frac{T_{seq} + T_o}{p}.$$
Cilk

Only two parallel programming primitives:

1. (binary) **fork**, and
2. (binary) **join**.

Spawned function calls are assigned to one of the available processes by the Cilk **run time scheduler**.

The Cilk scheduler **guarantees**, under some assumptions on the determinism of the algorithm, that

\[
T_0 = \mathcal{O}(pT_\infty),
\]

resulting in a parallel run-time \(T_p\) bounded by

\[
\mathcal{O}(T_1/p + T_\infty).
\]
MapReduce

Google Pregel is a successor of **MapReduce**, a parallel framework that operates on **large data sets** of key-value pairs

\[ S \subseteq K \times V, \]

with \( K \) a set of possible **keys** and \( V \) a set of **values**.

MapReduce defines two operations on \( S \):

\[
\begin{align*}
\text{map} &: K \times V \rightarrow K \times V; \\
\text{reduce}(k) &: \mathcal{P}(\{k\} \times V) \rightarrow K \times V, \quad k \in K.
\end{align*}
\]

- The map operation is **embarrassingly parallel**: every key-value pair is mapped to a new key-value pair, an entirely local operation.
Google Pregel is a successor of **MapReduce**, a parallel framework that operates on **large data sets** of key-value pairs

$$S \subseteq K \times V,$$

with $K$ a set of possible **keys** and $V$ a set of **values**.

MapReduce defines two operations on $S$:

- **map:** $K \times V \rightarrow K \times V$;
- **reduce**($k$): $\mathcal{P}(\{k\} \times V) \rightarrow K \times V, \ k \in K$.

- The map operation is **embarrassingly parallel**: every key-value pair is mapped to a new key-value pair, an entirely local operation.
- The reduction reduces **all** pairs in $S$ that have the same key $k$, into **one** single key-value pair: **global communication**.
Calculating $\alpha = x^T y$ using MapReduce:

Let $S = \{(0, \{x_0, y_0\}), (1, \{x_1, y_1\}), \ldots\}$.

1. **Map:** for each pair $(i, \{a, b\})$ write $(\text{partial}, a \cdot b)$. Applying this map adds $\{(\text{partial}, x_0y_0), (\text{partial}, x_1y_1), \ldots\}$ to $S$.

2. **Reduce:** for all pairs with key ‘partial’, combine their values by addition and store the result using key $\alpha$. Applying this reduction adds $(\alpha, \sum_{i=0}^{n-1} x_iy_i)$ to $S$.

3. **Done:** $S$ contains a single entry with key $\alpha$ and value $x^T y$.

The set $S$ is safely stored on a **resilient file system** to cope with hardware failures.
Pregel

Consider a graph $G = (V, E)$. **Graph algorithms** may be phrased in an SPMD fashion as follows:

- For each vertex $v \in V$, a thread executes a user-defined SPMD algorithm;
- each algorithm consists of successive local compute phases and global communication phases;
- during a communication phase, a vertex $v$ can only send messages to $N(v)$, where $N(v)$ is the set of neighbouring vertices of $v$; i.e., $N(v) = \{w \in V \mid \{v, w\} \in E\}$.

MapReduce and Pregel are variants of the BSP algorithm model!

- a type of **fine-grained BSP**.
- parallelism in Pregel is slightly odd to think about; e.g., what does its compute graph look like?
Fine-grained summary

Optimisation targets and performance metrics:

- Optimise algorithms to maximise **parallelism** and (thus) minimise the **algorithmic span**;
- run-time scheduler with **bounded overhead** (e.g., $pT_\infty$ for Cilk).

Questions:

1. does this account for all realistic overheads, in your experience?
2. does more parallelism always mean better performance?
3. we wrote Cilk bounded the parallel run-time by $\mathcal{O}(T_1/p + T_\infty)$. Is there a difference between $T_{\text{seq}}$ and $T_1$?
Fine-grained summary

Answers:

– Q: does this account for all realistic overheads, in your experience?
– A: no. Not accounted for are: memory overhead, and, most importantly, the costs of data movement!

Definition (Memory overhead)

Let \( M_{\text{seq}} \) be the memory requirement of a sequential algorithm, and let \( M_p \) be the memory requirement of a parallel algorithm that solves the same problem. Then the parallel overhead in memory is

\[
M_o = pM_p - M_{\text{seq}}, \quad \text{or, rewritten:}
\]

\[
M_p = \frac{M_{\text{seq}} + M_o}{p}.
\]

(Cilk bounds \( M_o \), whereas other fine-grained schemes may not.)
Answers:

- **Q**: *does more parallelism always mean better performance?*
- **A**: **No**; for starters, we typically have less than $\infty$ processors. Also: maximum speedup is relative to the chosen algorithm; other algorithms that solve the same problem may be preferable!

Example:

Consider the naive $\Theta(n^2)$ Fourier transformation; its span is $\Theta(\log n)$, so its parallelism is $\Theta(n^2/\log n)$. **Lots of parallelism**!

The FFT formulation has work $\Theta(n \log n)$, also with span $\Theta(\log n)$ resulting in $\Theta(\frac{n \log n}{\log n}) = \Theta(n)$ parallelism. **Less parallelism**...

Question: how many processors do you need to achieve a theoretical parallel processing time equal to the span, for both the naive and the fast algorithm?
Fine-grained summary

Answers:

- **Q: does more parallelism always mean better performance?**
- **A: No**; for starters, we typically have less than $\infty$ processors. Also: maximum speedup is relative to the chosen algorithm; other algorithms that solve the same problem may be preferable!

Example:

Consider the naive $\Theta(n^2)$ Fourier transformation; its span is $\Theta(\log n)$, so its parallelism is $\Theta(n^2 / \log n)$. **Lots of parallelism!**

The FFT formulation has work $\Theta(n \log n)$, also with span $\Theta(\log n)$ resulting in $\Theta\left(\frac{n \log n}{\log n}\right) = \Theta(n)$ parallelism. **Less parallelism**...

Question: how many processors do you need to achieve a theoretical parallel processing time equal to the span, for both the naive and the fast algorithm? Answer: naive $\Theta(n^2)$ processors, FFT $\Theta(n)$ processors.
Answers:

- **Q**: is there a difference between considering $T_{seq}$ or $T_1$?
- **A**: **definitely**; there may be multiple sequential algorithms to solve the same problem. When comparing, always **compare to the best**. For parallel sorting:

\[
S^{odd-even \ sort} = \frac{T_{seq}^{qsort}}{T_p^{odd-even \ sort}}.
\]

In some cases, however, using $T_1$ does make sense; for instance, when measuring $T_o$ to see if the implementation behaves as expected from a theoretical analysis:

\[
T_o^{odd-even \ sort} = pT_p^{odd-even \ sort} - T_{seq}^{odd-even \ sort}.
\]
Questions

Regarding the Cilk parallelisation of $x_n = x_{n-1} + x_{n-2}$, $x_0 = x_1 = 1$:

```c
int f( int n ) {
    if( n == 0 || n == 1 ) return 1;
    int x1 = cilk_spawn f( n-1 ); //fork
    int x2 = cilk_spawn f( n-2 ); //fork
    cilk_sync; //join
    return x1 + x2;
}
```

Questions:

- what is $T_1$ (asymptotically)?
- what is $T_\infty$ (asymptotically)?
- what is $T_{seq}$?
- is this trivial parallelisation a good idea?
Questions

Regarding the Cilk parallelisation of \( x_n = x_{n-1} + x_{n-2}, x_0 = x_1 = 1 \):

```c
int f( int n ) {
    if( n == 0 || n == 1 ) return 1;
    int x1 = cilk_spawn f( n-1 ); //fork
    int x2 = cilk_spawn f( n-2 ); //fork
    cilk_sync; //join
    return x1 + x2;
}
```

- \( T_1 = \Theta(2^n) \), \( T_\infty = \Theta(n) \), but
- \( T_{\text{seq}} = n! \) This parallelisation makes no sense.
In the bulk synchronous parallel setting, how do the concepts such as span and overhead translate?

Question:
- how do the computation graph, $T_\infty$, and $T_o$ look for BSP?
BSP and scalability

Q: what is the BSP computation graph, span, and overhead?

Graph:
- Sequential part, SPMD (supersteps), sequential part.
- The SPMD part is coarse-grained over $p$ processors.
Q: what is the BSP computation graph, span, and overhead?

Span:

- Critical path follows compute phases with maximum work $w_i^{(s)}$, and
- communication phases with cost proportional to $h_i$;
- $T_\infty = T_p = \sum_{i=0}^{N-1} (\max_s w_i^{(s)} + h_i g + l)$.
- The span is the BSP cost!
Q: what is the BSP computation graph, span, and overhead?

Overhead (as in $T_p = \frac{T_{\text{seq}}+T_o}{p}$):

$$T_o = p \left( \sum_{i=0}^{N-1} \max_s w_i^{(s)} + h_{ig} + l \right) - T_{\text{seq}}.$$  

Data movement, latency costs, and extra computations on top of the bare minimum required, all add up to the overhead.
BSP and scalability

Definition (Efficiency)

Given $T_{seq}$, $p$, and $T_p$, the efficiency $E$ of the parallel algorithm is

$$E = S/p = \frac{T_{seq}}{pT_p}.$$ 

An algorithm is scalable (in time) if $E$ is constant as $p, T_{seq} \to \infty$.

Question:

- Express $E$ in terms of $T_{seq}$ and $T_o$
BSP and scalability

Express $E$ in terms of $T_{\text{seq}}$ and $T_o$, answer:

$$E^{-1} = \frac{p T_p - T_{\text{seq}}}{T_{\text{seq}}} + 1 = \frac{T_o}{T_{\text{seq}}} + 1.$$

The efficiency only depends on $T_{\text{seq}}$ and $T_o$:

$$E(T_{\text{seq}}, p) = \frac{1}{1 + \frac{T_o(T_{\text{seq}}, T_p)}{T_{\text{seq}}}}.$$  

An algorithm is scalable if $T_o / T_{\text{seq}}$ is kept constant. This principle is known as **iso-efficiency** (as advocated by Grama et al.).
BSP and scalability

Express $E$ in terms of $T_{\text{seq}}$ and $T_o$, answer:

$$E^{-1} = \frac{pT_p - T_{\text{seq}}}{T_{\text{seq}}} + 1 = \frac{T_o}{T_{\text{seq}}} + 1.$$

The efficiency only depends on $T_{\text{seq}}$ and $T_o$:

$$E(T_{\text{seq}}, p) = \frac{1}{1 + \frac{T_o(T_{\text{seq}}, T_p)}{T_{\text{seq}}}}.$$

An algorithm is scalable if $T_o/T_{\text{seq}}$ is kept constant. This principle is known as **iso-efficiency** (as advocated by Grama et al.).

Questions: how is iso-efficiency affected when considering

1. strong scalability, i.e., $S = \frac{T_{\text{seq}}}{T_p} = \Omega(p)$ as $p \to \infty$ while $T_{\text{seq}}$ is assumed constant.

2. weak scalability, i.e., $S = O(1)$ as $T_{\text{seq}} \to \infty$ while $p$ is constant.
BSP and scalability

Questions: how is iso-efficiency affected when considering

1 strong scalability, i.e., $S = \frac{T_{seq}}{T_p} = \Omega(p)$ as $p \to \infty$ while $T_{seq}$ is assumed constant.

2 weak scalability, i.e., $S = O(1)$ as $T_{seq} \to \infty$ while $p$ is constant.

Answers:

1 strong scalability:

$$E \sim S/p = \frac{\Omega(p)}{p} = \Omega(1).$$

2 weak scalability,

$$E \sim \frac{O(1)}{p} = O(1).$$

That is, both strong and weak scalability induce iso-efficiency.
BSP and scalability

**Question:** Is a BSP algorithm with $T_p = O(T_{seq}/p + p)$ strongly scalable? How about weakly? What is its iso-efficiency?

**Answer:** No, yes, and $p^2/T_{seq}$. ($T_{seq}$ grows quadratically w.r.t. $p$).

-I.e., when $T_p = O(T_{seq}/p + p)$, there is no strong scalability.

**Question:** for a small constant $c$, do we have strong scalability if

1. $T_p = T_{seq}/p + cT_{seq}$,
2. $T_p = T_{seq}/p + c$,
3. $T_p = T_{seq}/p + c/p$.

**Answers:**

1. No: $T_o = cpT_{seq}$,
2. No: $T_o = cp$,
3. Yes: $T_o = c$; strong scalability means constant overhead.
BSP and scalability

**Question:** Is a BSP algorithm with $T_p = O(T_{seq}/p + p)$ strongly scalable? How about weakly? What is its iso-efficiency?

**Answer:** no, yes, and $p^2/T_{seq}$. ($T_{seq}$ grows quadratically w.r.t. $p$.)
**BSP and scalability**

**Question:** Is a BSP algorithm with $T_p = O(T_{seq}/p + p)$ strongly scalable? How about weakly? What is its iso-efficiency?

**Answer:** no, yes, and $p^2/T_{seq}$. ($T_{seq}$ grows quadratically w.r.t. $p$.)

I.e., when $T_p = O(T_{seq}/p + p)$, there is no strong scalability.

**Question:** for a small constant $c$, do we have strong scalability if

1. $T_p = T_{seq}/p + cT_{seq}$,
2. $T_p = T_{seq}/p + c$,
3. $T_p = T_{seq}/p + c/p$. 
BSP and scalability

**Question:** Is a BSP algorithm with $T_p = \mathcal{O}(T_{\text{seq}}/p + p)$ strongly scalable? How about weakly? What is its iso-efficiency?

**Answer:** no, yes, and $p^2/T_{\text{seq}}$. ($T_{\text{seq}}$ grows quadratically w.r.t. $p$.)

I.e., when $T_p = \mathcal{O}(T_{\text{seq}}/p + p)$, there is no strong scalability.

**Question:** for a small constant $c$, do we have strong scalability if

1. $T_p = T_{\text{seq}}/p + cT_{\text{seq}},$
2. $T_p = T_{\text{seq}}/p + c,$
3. $T_p = T_{\text{seq}}/p + c/p.$

**Answers:**

1. No: $T_o = cpT_{\text{seq}},$
2. No: $T_o = cp,$
3. Yes: $T_o = c$; strong scalability means constant overhead.
BSP and scalability

For non-embarrassingly parallel applications, **strong scalability does not exist**. But is that a bad thing?

**Definition (Amdahl’s Paradox)**

Given an algorithm with a serial part of size $c$ and a parallel part of size $(1 - c)$, then the maximum speedup of that algorithm is given by

$$S = \frac{T_{\text{seq}}}{T_{\text{seq}}(c + (1 - c)/p)} = \frac{1}{1/p - c/p + c}.$$  

Note that for $p \to \infty$, $S \to \frac{1}{c}$, ‘and therefore parallel computing is of extremely limited use’ (paraphrased from Amdahl).

If $c = 10^{-2}$ (one percent), the maximum speedup is 100. For $c = 10^{-6}$, $S \leq 10^6$, etc.
BSP and scalability

For non-embarrassingly parallel applications, strong scalability does not exist. But is that a bad thing?

Definition (Amdahl’s Paradox)

Given an algorithm with a serial part of size $c$ and a parallel part of size $(1 - c)$, then the maximum speedup of that algorithm is given by

$$S = \frac{T_{\text{seq}}}{T_{\text{seq}}(c + (1 - c)/p)} = \frac{1}{1/p - c/p + c}.$$

Note that for $p \to \infty$, $S \to \frac{1}{c}$, ‘and therefore parallel computing is of extremely limited use’ (paraphrased from Amdahl).

If $c = 10^{-2}$ (one percent), the maximum speedup is 100. For $c = 10^{-6}$, $S \leq 10^6$, etc. So to scale, we need a lot more parallelisable work than non-parallelisable work. ‘This naturally happens when $T_{\text{seq}} \to \infty$’ (paraphrased from Gustavson); this is what led to weak scalability.
Recall that for BSP algorithms:

\[ T_o = p \left( \sum_{i=0}^{N-1} \max_s w_i^{(s)} + h_i g + l \right) - T_{seq}. \]

Iso-efficiency then requires that the following expression be kept constant:

\[ E^{-1} = 1 + \frac{T_o}{T_{seq}} = p \left( \sum_{i=0}^{N-1} \frac{\max_s w_i^{(s)} + h_i g + l}{T_{seq}} \right). \]

For BSP, usually \( T_p > T_{seq}/p + g + l \); we never expect strong scalability; but like always, making sure that \( T_p \sim T_{seq} \) gets us weak scalability.
Algorithm design

1. The Multi-BSP model
2. Philosophy
3. Algorithm design
Multi-BSP: broadcasting
Multi-BSP: broadcasting
Multi-BSP: broadcasting
Multi-BSP: broadcasting
Multi-BSP: broadcasting
Multi-BSP: broadcasting
Multi-BSP: broadcasting

Question: can we do better?

1. Communicate value upwards only (until the root node has it);
2. Broadcast values downwards.

Due to the imposed Multi-BSP computer model, each node must be visited, even in this minimal example.

For any non-trivial Multi-BSP algorithm, \( T \geq 2^p \left( l_k + g_k \right) \).
Multi-BSP: broadcasting

**Question:** can we do better?

Perform more balanced communication, two phases:

1. communicate value upwards only (until the root node has it);
2. broadcast values downwards.
Multi-BSP: broadcasting

**Question**: can we do better?

Perform more balanced communication, two phases:

1. communicate value upwards only (until the root node has it);
2. broadcast values downwards.

Due to the imposed Multi-BSP computer model, each node **must** be visited, even in this minimal example.

For any non-trivial Multi-BSP algorithm,

\[
T_o \geq 2p \sum_{k=0}^{L-1} (g_k + l_k).
\]
**Multi-BSP: broadcasting**

**Question:** can we do better?

Perform more balanced communication, two phases:

1. communicate value upwards only (until the root node has it);
2. broadcast values downwards.

Due to the imposed Multi-BSP computer model, each node **must** be visited, even in this minimal example.

For any non-trivial Multi-BSP algorithm,

\[ T_o \geq 2p \sum_{k=0}^{L-1} (g_k + l_k). \]

**Question:** is storing the broadcast value \(2p - 1\) times mandatory?
Multi-BSP: broadcasting

Memory embedding (shared address space):

Memory requirements are now bounded from below as $M_p \geq M_{seq}/p + p$. 
Multi-BSP: broadcasting

Optimal algorithm with embedding (shared address space):

Start SPMD section, entry at leaf level, leaf 6 is source.
Multi-BSP: broadcasting

Optimal algorithm with embedding (shared address space):

On this local BSP computer, communicate *val* to PID 0 and move up.
Multi-BSP: broadcasting

Optimal algorithm with embedding (shared address space):

On this upper level, send \textit{val} to PID 0, \textbf{broadcast}, and \textbf{move down}. 
Multi-BSP: broadcasting

Optimal algorithm with embedding (shared address space):

On this level, only PID 0 has \textit{val}; \textbf{broadcast}, and done.
Multi-BSP: broadcasting

Optimal algorithm with embedding (shared address space):

On this level, only PID 0 has *val*; *broadcast*, and done.
Multi-BSP: broadcasting

Input: \( val \) (a value or \( \emptyset \)),
Output: the value that was broadcast.

\[
\text{if } val \neq \emptyset \text{ then} \\
\quad \text{set } source = true \\
\text{else} \\
\quad \text{set } source = false \\
\text{while not on the Multi-BSP root do} \\
\quad \text{if } source \text{ and } bsp\_pid() \neq 0 \text{ then} \\
\qquad \text{send } val \text{ to PID 0} \\
\qquad \text{move upwards in the Multi-BSP tree} \\
\text{while not on a leaf node do} \\
\quad \text{if } bsp\_pid() = 0 \text{ then} \\
\qquad \text{for } k = 1 \text{ to } bsp\_nprocs() \text{ do} \\
\qquad \qquad \text{send } val \text{ to PID } k \\
\qquad \text{move downwards in the Multi-BSP tree} \\
\text{return } val
\]
New Multi-BSP primitives

To control the flow up and down the Multi-BSP tree:
- `bsp_up()`, and
- `bsp_down()`.
These are **synchronising** primitives, like `bsp_sync()`.

To inspect Multi-BSP tree information:
- `bsp_lid()`, the leaf ID of this SPMD program;
- `bsp_leaf()`, whether this SPMD program runs on a leaf;
- `bsp_sleaves()`, the number of leaf nodes in this subtree;
- `bsp_nleaves()`, the total number of leaf nodes in the full tree.

That’s it!
Multi-BSP summary

- Non-trivial Multi-BSP algorithms require that

\[ T_p = \Omega\left(\frac{T_{\text{seq}}}{p} + \sum_{k=0}^{L-1}(g_k + l_k)\right); \]

If each node in the tree runs an SPMD program and is non-trivial, then
- leaf nodes distribute the sequential work,
- internal nodes communicate at least one word and synchronise once while going up the Multi-BSP computer tree, and
- synchronise again while going down the tree.

The minimal non-trivial parallel cost thus is as given above.
Multi-BSP summary

- non-trivial Multi-BSP algorithms require that

\[ pM_p = \Omega(M_{seq} + p), \text{ i.e., } M_o = \Omega(p); \]

Each Multi-BSP node is represented in memory, together with at least the \( M_{seq} \) data required for the entire problem, from which the above follows.
Multi-BSP summary

– communication can still be done using ‘horizontal’ primitives.

Algorithm data does not need to be replicated by using memory embedding. This also enables horizontal communication, just like what we are used to from ‘flat’ BSP.
Multi-BSP summary

- in general, no compute on non-leaf nodes.

In principle, the leaf node with PID 0 could handle the computation of its parent Multi-BSP node. Higher-level nodes can choose a representative in a recursive fashion, similar to memory embedding.

Distributing part of the computational cost over non-leaf nodes in this fashion, necessarily uses less than $p$ processors. Doing so leaves the remaining processes idle, thus contributing unnecessary overhead.
Multi-BSP summary

– we have demonstrated a Multi-BSP broadcasting algorithm with cost

\[ T_{p}^{\text{multibsp-bcast}} = \sum_{k=0}^{L-1} \left( g_k + l_k + T_k^{\text{bsp-bcast}} \right), \text{ with} \]

\[ T_k^{\text{bsp-bcast}} = \min_{b \in \{2, \ldots, p\}} ((b - 1)g_k + l_k) \log_b p_k. \]

The Multi-BSP broadcast traverses the Multi-BSP tree a minimal number of times; only once up, and once down.

– we can still use known BSP algorithms within Multi-BSP algorithms;
– only a few additional primitives enable writing portable codes.
Multi-BSP summary

- Non-trivial Multi-BSP algorithms require that

\[ T_p = \Omega \left( \frac{T_{\text{seq}}}{p} + \sum_{k=0}^{L-1} (g_k + l_k) \right); \]

- non-trivial Multi-BSP algorithms require that

\[ pM_p = \Omega (M_{\text{seq}} + p), \text{ i.e., } M_o = \Omega (p); \]

- communication can still be done using ‘horizontal’ primitives;
- in general, no compute on non-leaf nodes;
- we have demonstrated a Multi-BSP broadcasting algorithm and determined its Multi-BSP cost;
- we can still use known BSP algorithms within Multi-BSP algorithms;
- only a few additional primitives enable writing portable codes.