Next-generation shared-memory parallel computing

Albert-Jan Yzelman

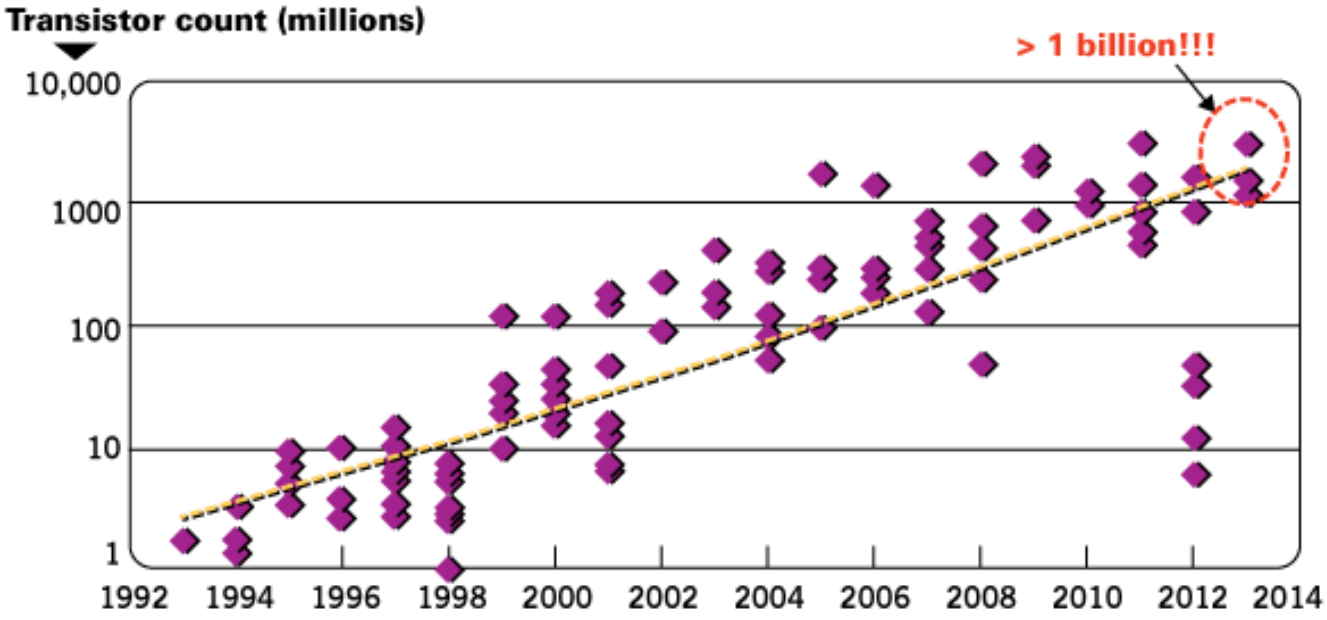
11th of November, 2015

France Research Centre on
Parallel Algorithms and Big Data

Outline

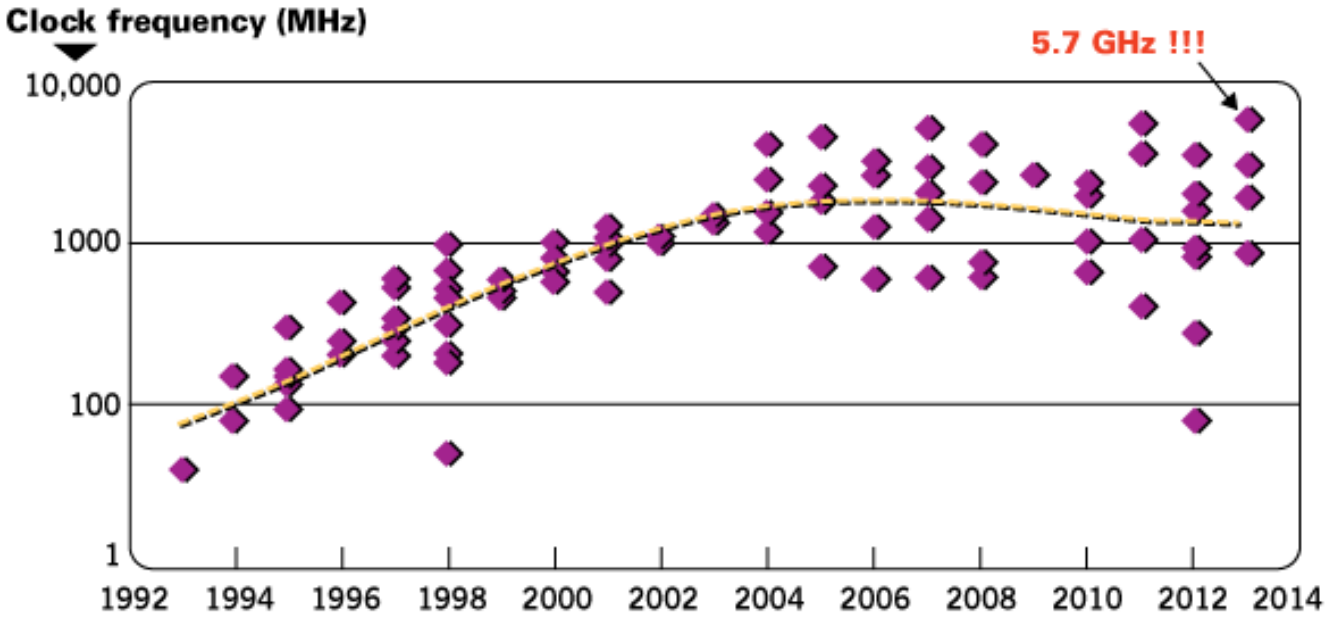
Shared-memory architectures

Hardware trends: Moore's Law



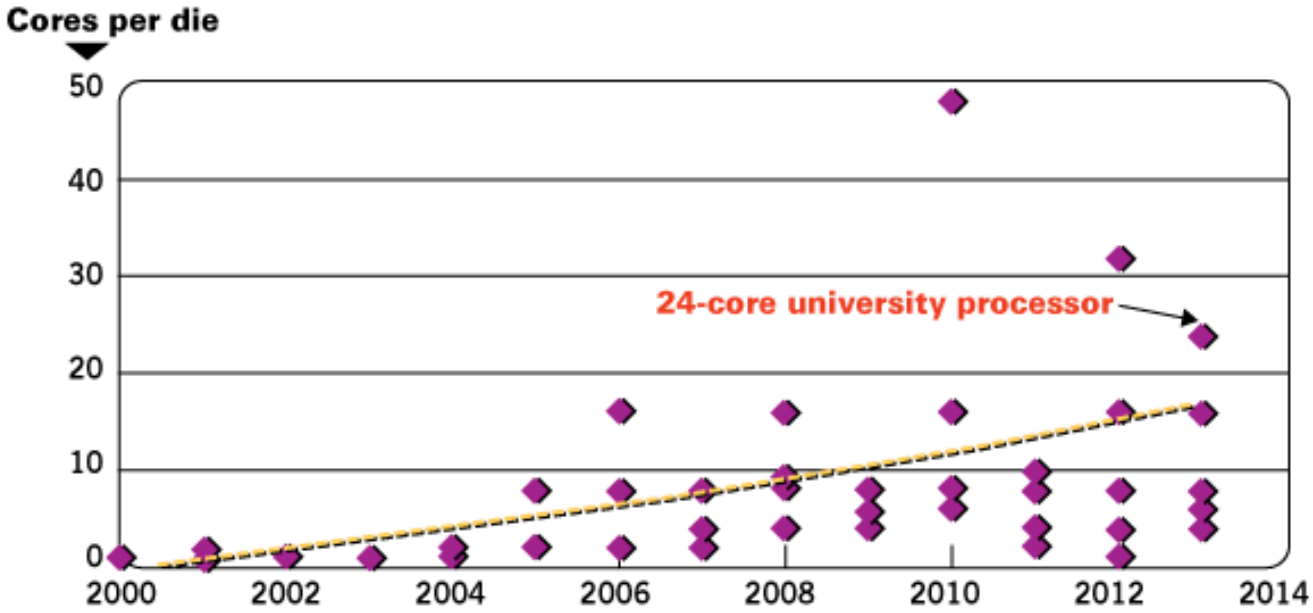
Graphics from Ron Maltiel, Maltiel Consulting
(`http://www.maltiel-consulting.com/ISSCC-2013-High-Performance-Digital-Trends.html`)

Hardware trends: core speeds



Graphics from Ron Maltiel, Maltiel Consulting
(`http://www.maltiel-consulting.com/ISSCC-2013-High-Performance-Digital-Trends.html`)

Hardware trends: core count



Graphics from Ron Maltiel, Maltiel Consulting
(`http://www.maltiel-consulting.com/ISSCC-2013-High-Performance-Digital-Trends.html`)

Hardware trends: bandwidth

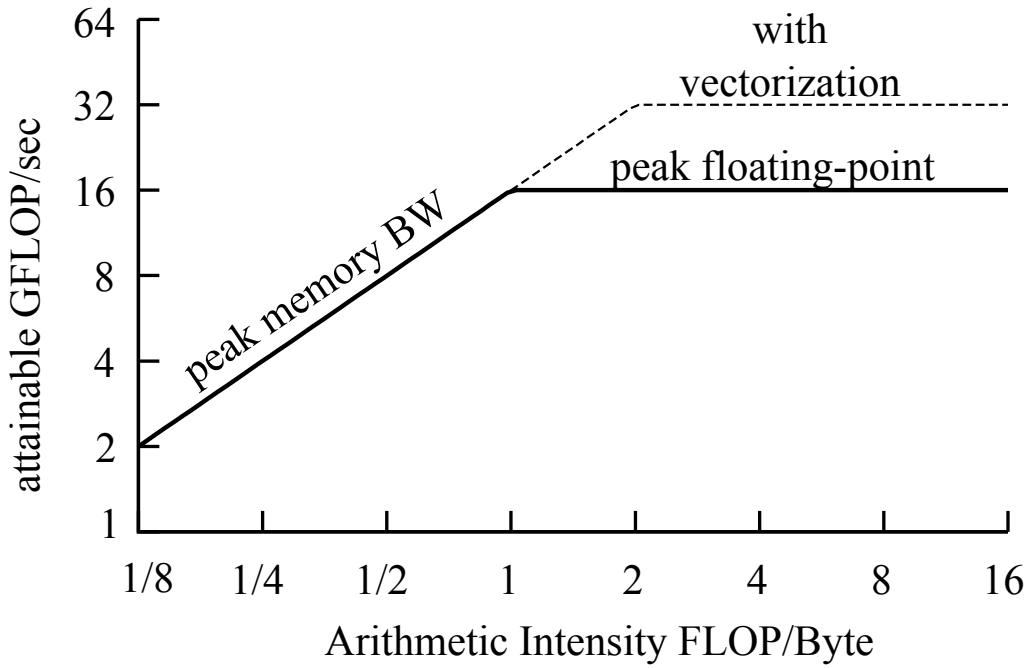
CPU speeds stall, but Moore's Law now translates to an increasing amount of cores per die, i.e., **the effective flop rate of processors still rises** as it always has.

- But what about **bandwidth**?

Technology	Year	Bandwidth	cores
EDO	1970s	27 Mbyte/s	1
SDRAM	early 1990s	53 Mbyte/s	1
RDRAM	mid 1990s	1.2 Gbyte/s	1
DDR	2000	1.6 Gbyte/s	1
DDR2	2003	3.2 Gbyte/s	2
DDR3	2007	6.4 Gbyte/s	4
DDR3	2013	11 Gbyte/s	8
DDR4	2015	25 Gbyte/s	14

Effective bandwidth per core, stalled at best...

Arithmetic intensity

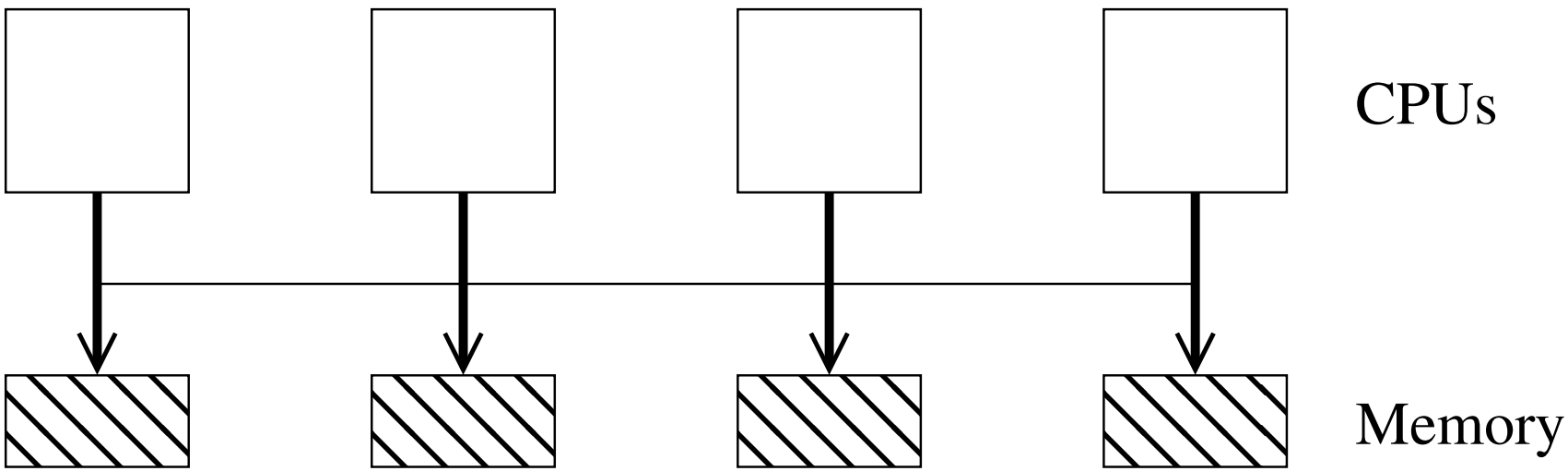


- If your computation has enough work per data element, it is **compute bound**; otherwise, it is **bandwidth bound**.
- If you are bandwidth bound, reducing your memory footprint, e.g., by **compression**, directly results in faster execution.

(Image courtesy of Prof. Wim Vanroose, UA)

Multi-socket architectures: NUMA

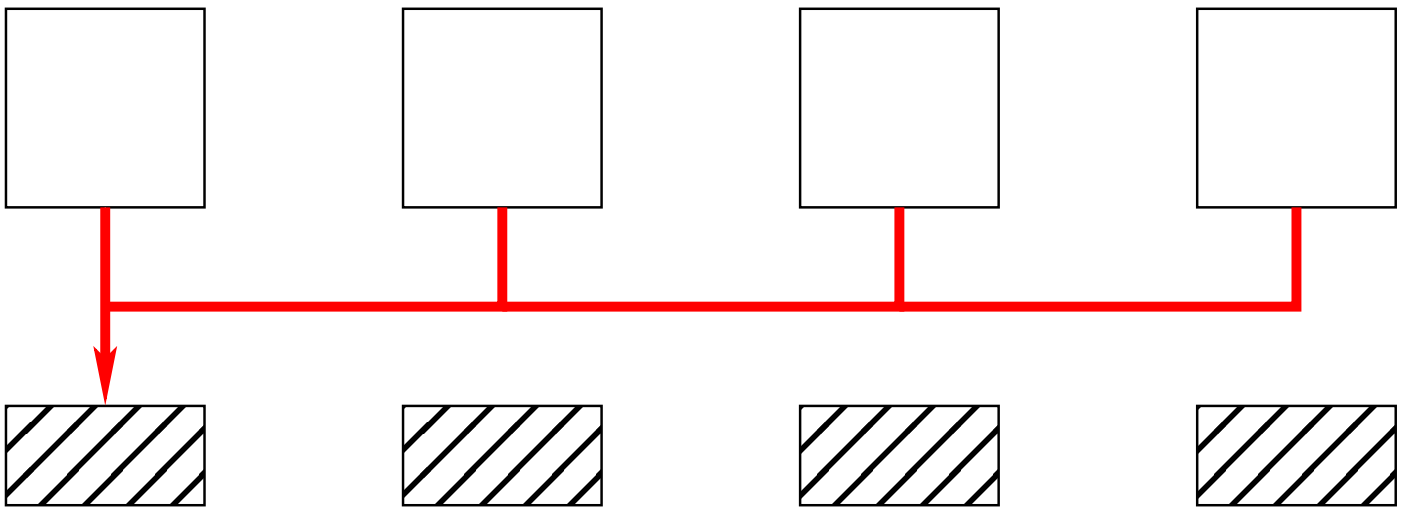
Each socket has **local** main memory where access is **fast**. Between sockets, access is slower.



Access times to shared-memory depends on the physical location, leading to *non-uniform memory access* (NUMA).

Dealing with NUMA: distribution types

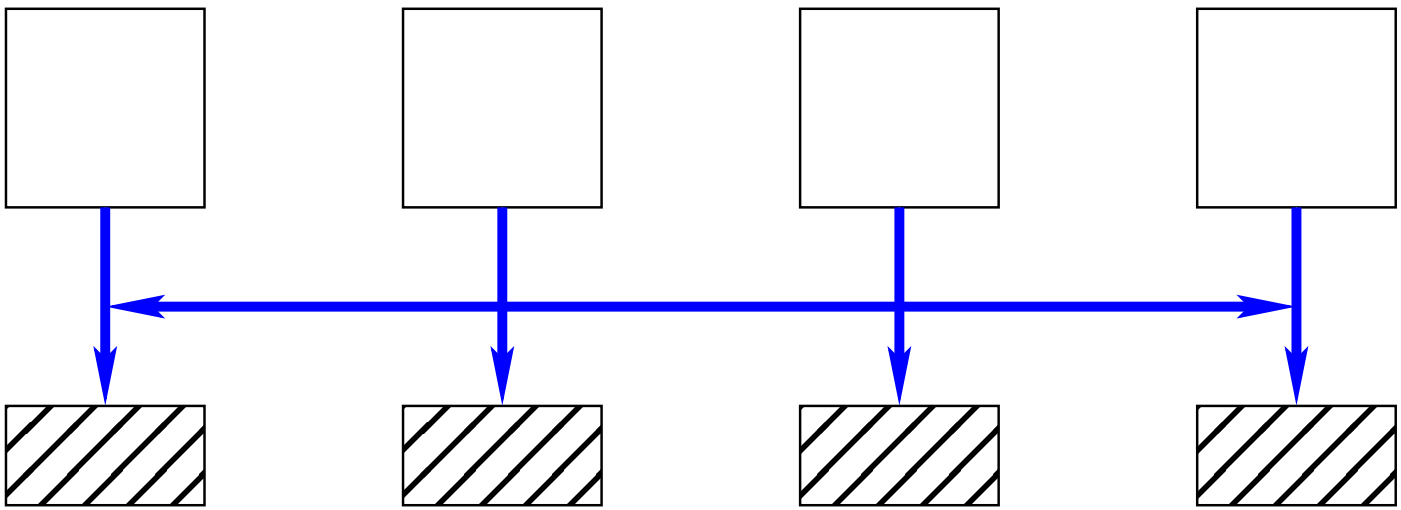
Implicit distribution, centralised local allocation:



If each processor moves data to the same single memory element, the **bandwidth is limited** by that of a single memory controller.

Dealing with NUMA: distribution types

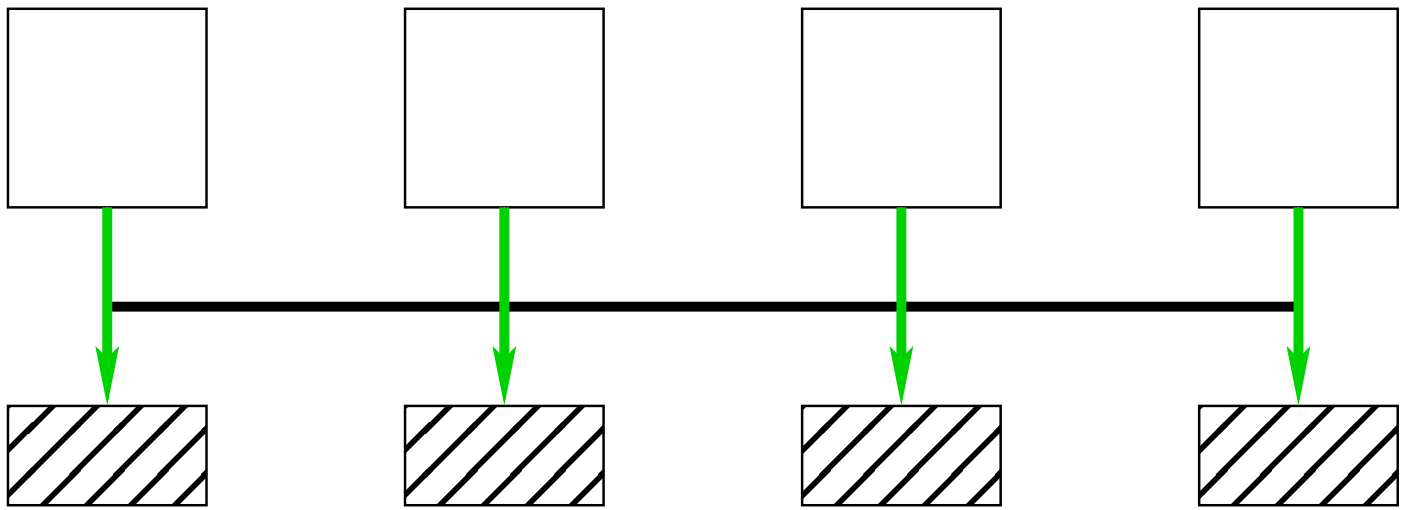
Implicit distribution, centralised interleaved allocation:



If each processor moves data from all memory elements, the bandwidth multiplies **if accesses are uniformly random**.

Dealing with NUMA: distribution types

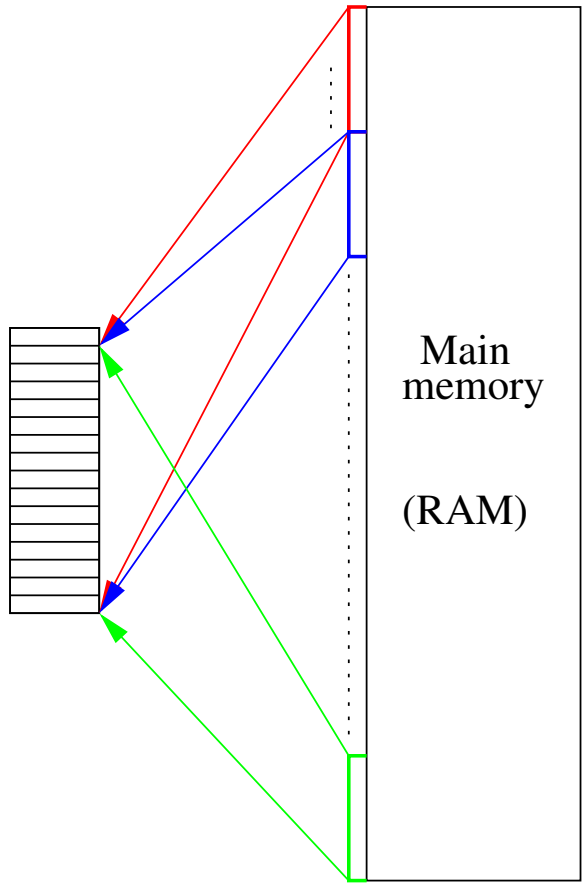
Explicit distribution, distributed local allocation:



If each processor moves data from and to its own unique memory element, the **bandwidth multiplies**.

Caches

Divide the main memory (RAM) in stripes of size L_S.



The ith line in RAM is mapped to the cache line i mod L, where L is the number of available cache lines. **Can we do better?**

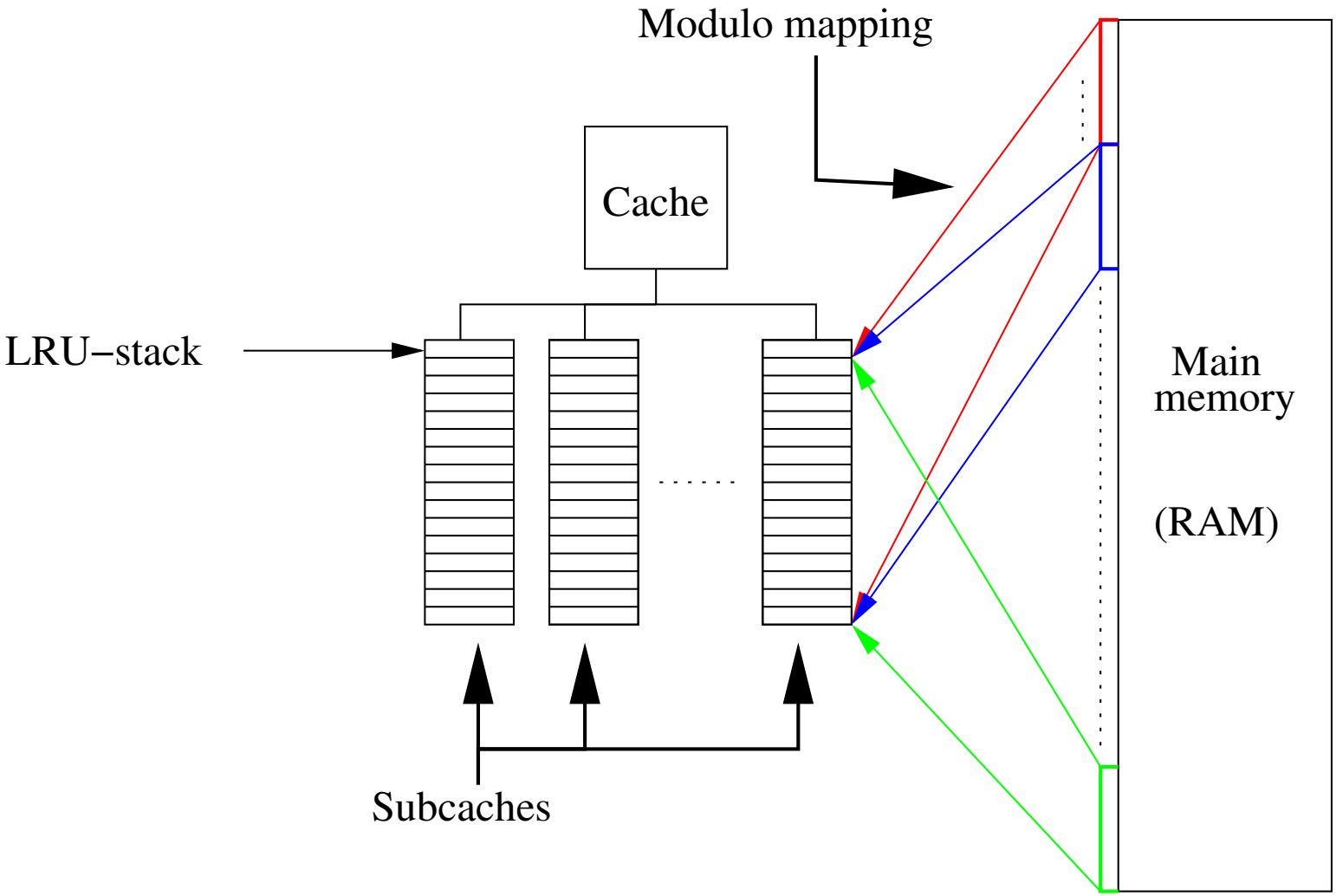
Caches

A smarter cache follows a pre-defined policy instead; for instance, the 'Least Recently Used (LRU)' policy:

		Req. $x_1, \ldots, x_4$		Req. x_2		Req. x_5
		x_4		x_2		x_5
	$\Rightarrow$	x_3	$\Rightarrow$	x_4	$\Rightarrow$	x_2
		x_2		x_3		x_4
		x_1		x_1		x_3

Caches

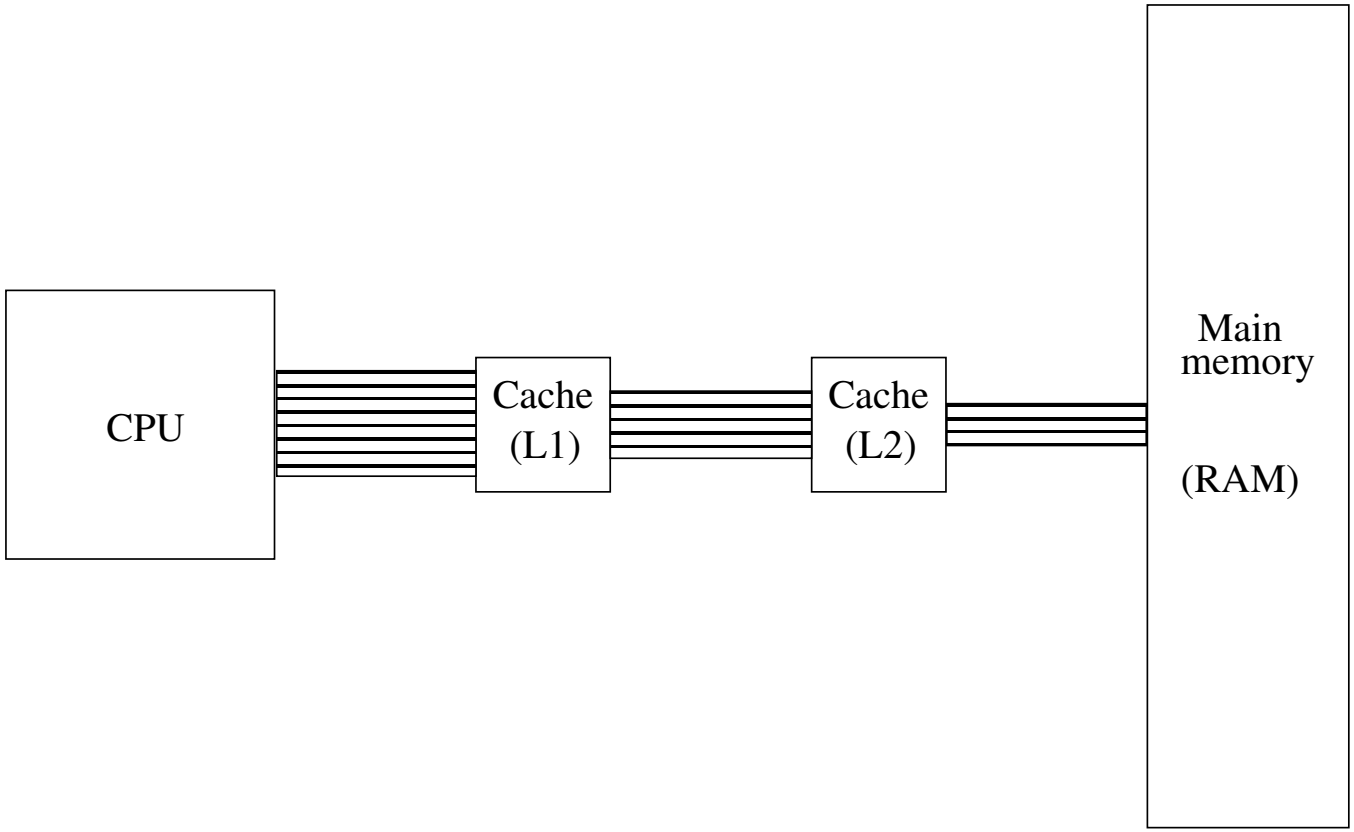
Realistic caches combine modulo-mapping and the LRU policy:



k is the number of subcaches; there are L/k LRU stacks.

Caches

Realistic caches are used within multi-level memory hierarchies:



	Intel Core2 (Q6600)		AMD Phenom II (945e)		Intel Westmere (E7-2830)	
L1:	32kB	$k = 8$	$S = 64$kB	$k = 2$	$S = 256$kB	$k = 8$
L2:	4MB	$k = 16$	$S = 512$kB	$k = 8$	$S = 2$MB	$k = 8$
L3:	-	-	$S = 6$MB	$k = 48$	$S = 24$MB	$k = 24$

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching and $S = 4$:

x_0

$\Longrightarrow$

———

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching and $S = 4$:

x_0 a_{00}

x_0

$\Longrightarrow$ $\Longrightarrow$

____ _____

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching and $S = 4$:

$$\begin{array}{ccccc} x_0 & & a_{00} & & y_0 \\ & & x_0 & & a_{00} \\ & \Longrightarrow & & \Longrightarrow & x_0 \quad \Longrightarrow \\ \hline & & \hline & & \hline \end{array}$$

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching and $S = 4$:

$$\begin{array}{ccccccccc} x_0 & & a_{00} & & y_0 & & x_1 & & \\ & & x_0 & & a_{00} & & y_0 & & \\ & \Longrightarrow & & \Longrightarrow & x_0 & \Longrightarrow & a_{00} & \Longrightarrow & \\ \hline & & \hline & & \hline & & x_0 & & \\ & & & & & & \hline & & \end{array}$$

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching and $S = 4$:

$$\begin{array}{ccccccccc} x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} \\ & & x_0 & & a_{00} & & y_0 & & x_1 \\ & \Longrightarrow & & \Longrightarrow & x_0 & \Longrightarrow & a_{00} & \Longrightarrow & y_0 & \Longrightarrow \\ \hline & & \hline & & \hline & & x_0 & & a_{00} \\ & & & & & & \hline & & \mathbf{x_0} \end{array}$$

Caches and multiplication

Dense matrix–vector multiplication

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

Example with LRU caching and $S = 4$:

$$\begin{array}{ccccccccccc} x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} & & y_0 \\ & & x_0 & & a_{00} & & y_0 & & x_1 & & a_{01} \\ & \Longrightarrow & & \Longrightarrow & x_0 & \Longrightarrow & a_{00} & \Longrightarrow & y_0 & \Longrightarrow & x_1 \\ \hline & & \hline & & \hline & & x_0 & & a_{00} & & a_{00} \\ & & & & & & \hline & & \mathbf{x_0} & & \mathbf{x_0} \end{array}$$

Caches and multiplication: NUMA again

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row. This causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Caches and multiplication: NUMA again

When k, L are larger, we can predict:

- lower elements from x are evicted while processing the first row. This causes $\mathcal{O}(n)$ cache misses on $m - 1$ rows.

Fix:

- stop processing a row before an element from x would be evicted; first continue with the next rows.

This results in column-wise 'stripes' of the dense A:

$$A = \left(\begin{array}{c|c|c|c} & & \cdots & \end{array} \right).$$

Caches and multiplication: NUMA again

$$A = \begin{pmatrix} | & | & \cdots & | \end{pmatrix}.$$

But now:

- elements from the vector y can be prematurely evicted; $\mathcal{O}(m)$ cache misses on each *block of columns*. (Already much better!)

Caches and multiplication: NUMA again

$$A = \begin{pmatrix} | & | & \cdots & | \end{pmatrix}.$$

But now:

- elements from the vector y can be prematurely evicted; $\mathcal{O}(m)$ cache misses on each *block of columns*. (Already much better!)

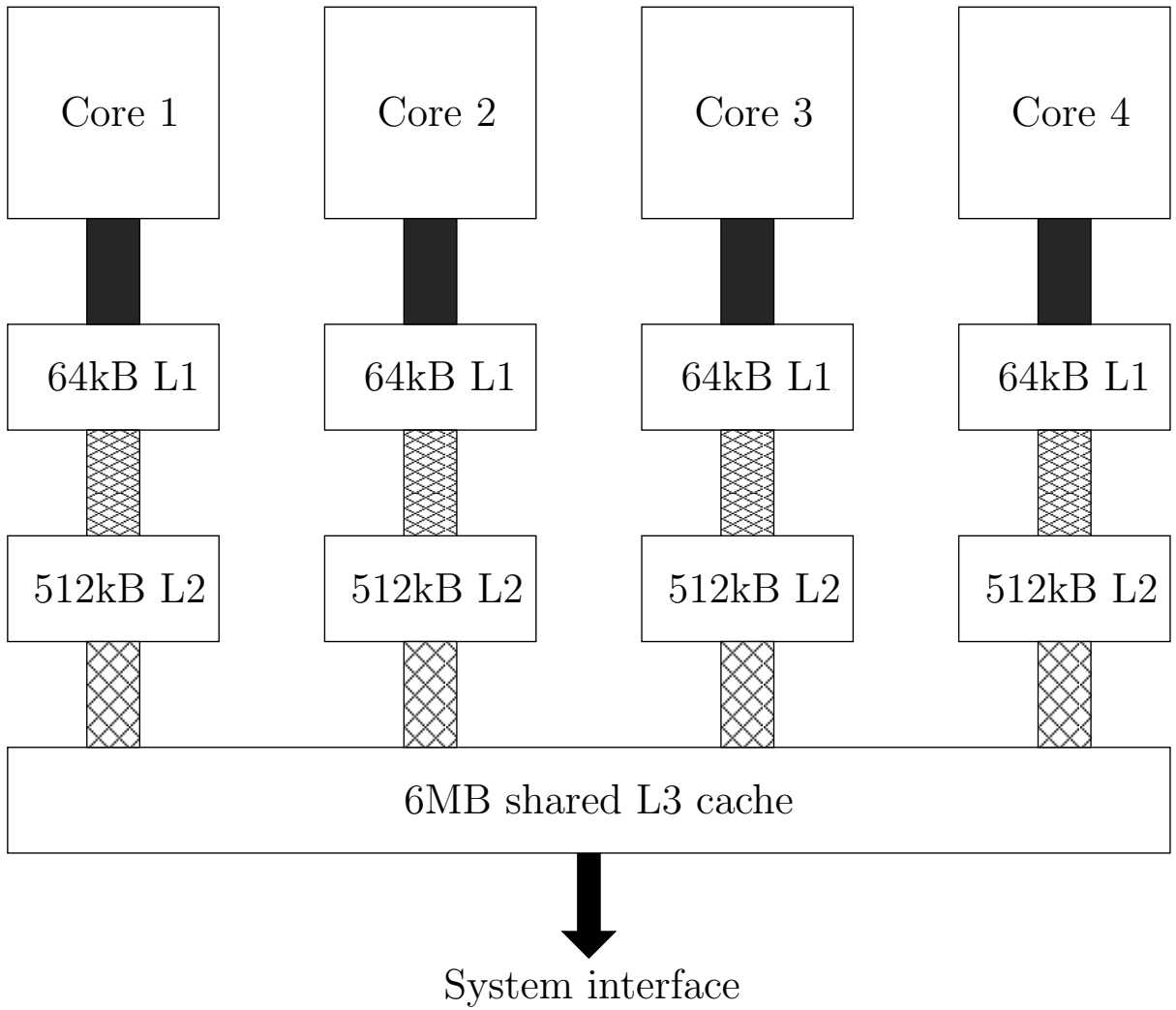
Fix:

- stop processing before an element from y is evicted; first do the remaining column blocks.

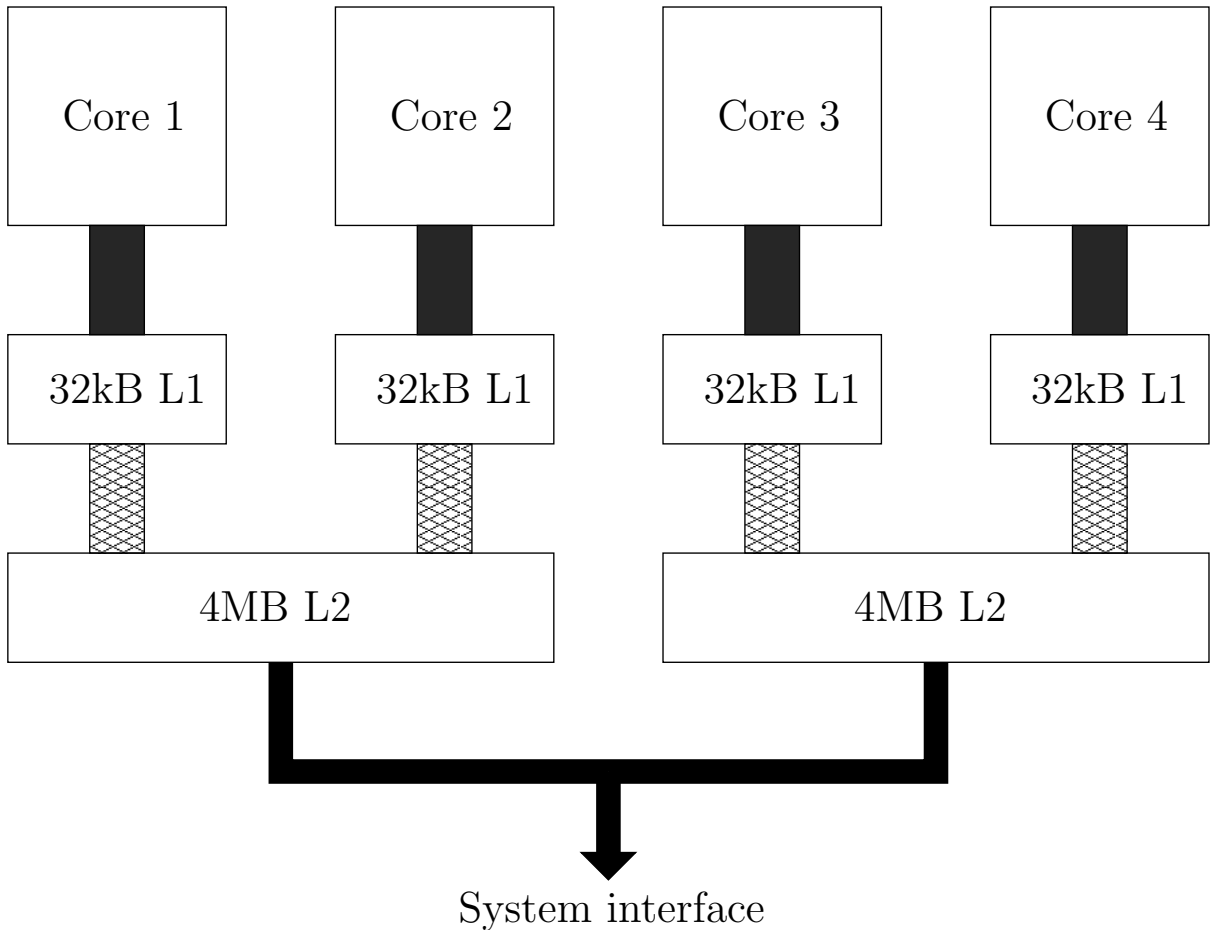
This is *cache-aware blocking*.

Caches and multicore

Most architectures employ shared caches.

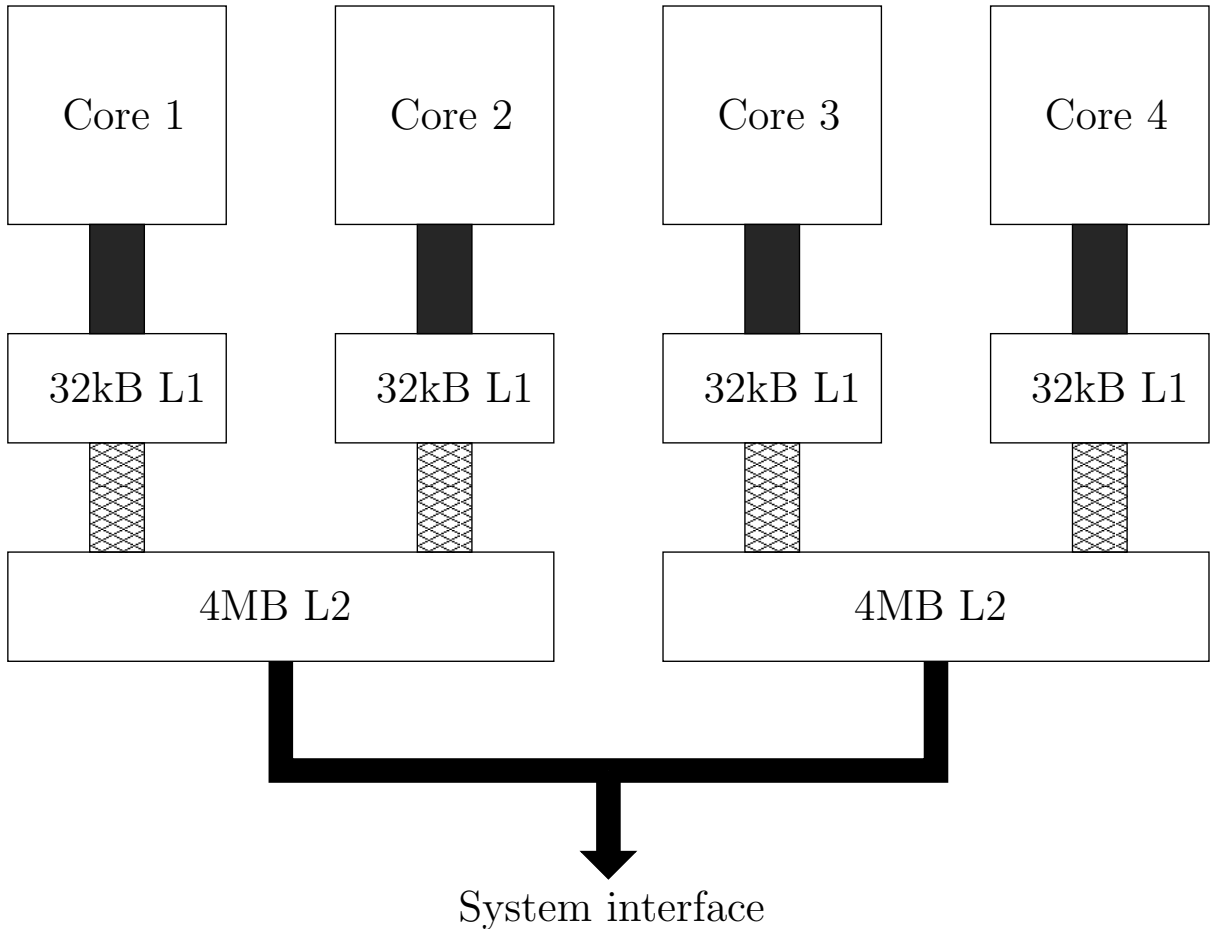


Caches and multicore: NUMA



In BSP: $(4, 2.4\ \text{GHz}, l, g)$. Is this the correct view?

Caches and multicore: NUMA



In BSP: $(4, 2.4\ \text{GHz}, l, g)$. But **Non-Uniform Memory Access**!

Shared-memory parallel programming

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double α;' **globally allocated**:

- $\alpha = 0.0$
- for $i = s$ to n step p
- $\quad\quad \alpha \mathrel{+}= x_i y_i$
- return α

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double α;' **globally allocated**:

- $\alpha = 0.0$
- for $i = s$ to n step p
- $\quad\quad \alpha \mathrel{+}= x_i y_i$
- return α

Data race! (for $n = p = 2$, output can be $x_0 y_0$, $x_1 y_1$, **or** $x_0 y_0 + x_1 y_1$)

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

s	the current processor ID,
p	the total number of processors (threads),
n	the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double α[**p**];' globally allocated:

- for $i = s$ to n step p
- $\alpha_s \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p-1} \alpha_i$

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with '$\alpha[\mathbf{p}]$;' globally allocated:

- for $i = s$ to n step p
- $\alpha_s \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p-1} \alpha_i$

False sharing! (processors access and update the same cache lines)

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double $\alpha[\mathbf{8}p]$;' globally allocated:

- for $i = s$ to n step p
- $\quad\quad \alpha_{8s} \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

- s the current processor ID,
- p the total number of processors (threads),
- n the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with '$\text{double } \alpha[\mathbf{8}p];$' globally allocated:

- for $i = s$ to n step p
- $\quad\quad \alpha_{8s} \mathrel{+}= x_i y_i$
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

Inefficient cache use!

(All threads access virtually all cache lines; $\Theta(pn)$ data movement)

Shared-memory programming intro

Suppose x and y are in a shared memory. We calculate an inner-product in parallel, using the cyclic distribution.

Input:

s	the current processor ID,
p	the total number of processors (threads),
n	the size of the input vectors.

Output: $x^T y$

Shared-memory SPMD program with 'double $\alpha[8p]$;' globally allocated:

- for $\mathbf{i} = \mathbf{s} \cdot \lceil \mathbf{n}/\mathbf{p} \rceil$ **to** $(\mathbf{s}+\mathbf{1}) \cdot \lceil \mathbf{n}/\mathbf{p} \rceil$
- α_{8s} += $x_i y_i$
- return $\sum_{i=0}^{p-1} \alpha_{8i}$

(Now inefficiency only at boundaries; $\mathcal{O}(n+p-1)$ data movement)

Central obstacles for SpMV multiplication

Given a **sparse** $m \times n$ matrix A, an input vector x, and an output vector y. How to calculate

$$y = Ax$$

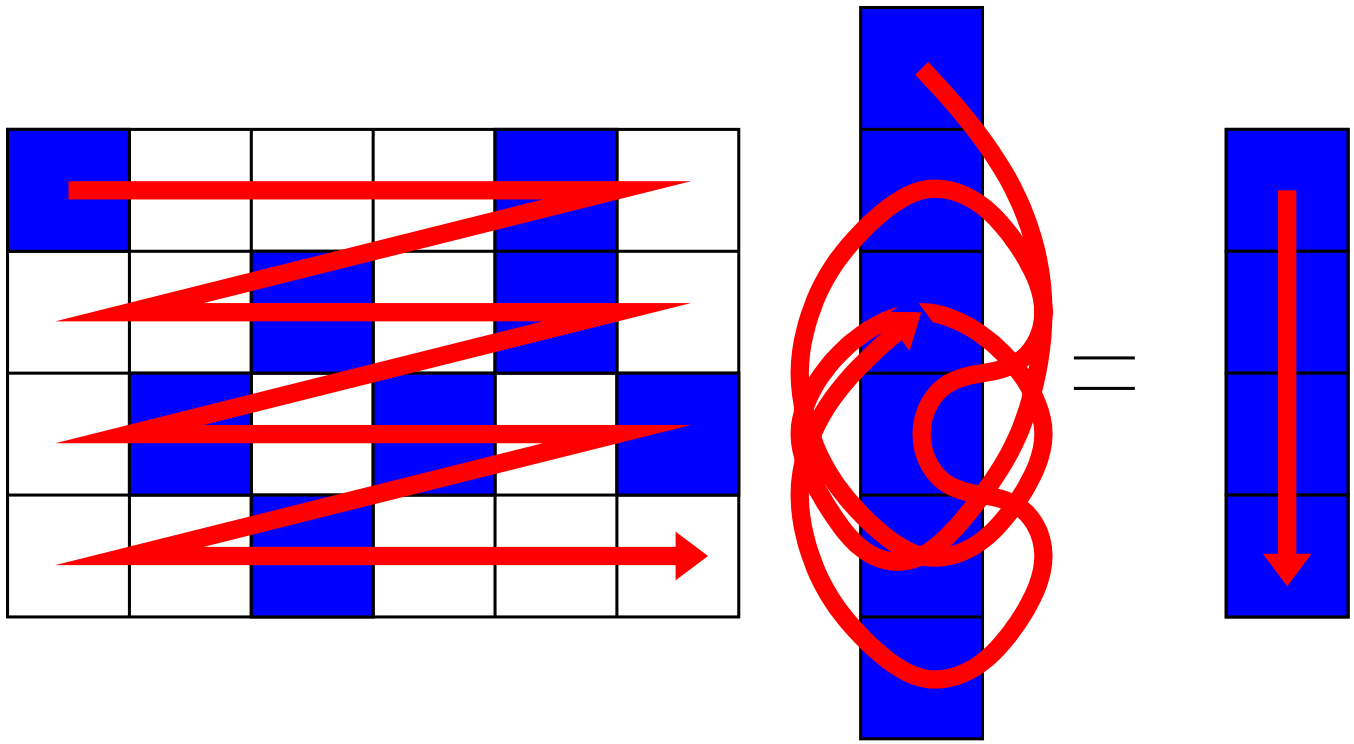
on a **shared-memory** parallel computer, as fast as possible?

Three obstacles for an efficient shared-memory parallel sparse matrix–vector (SpMV) multiplication kernel:

- inefficient cache use,
- limited memory bandwidth, and
- non-uniform memory access (NUMA).

Inefficient cache use

Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in row-major order:



Accesses on the input vector are completely unpredictable.

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$

$\Longrightarrow$

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$x_?$ $a_{0?}$

$x_?$

$\Longrightarrow$ $\Longrightarrow$

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$$x_? \qquad a_{0?} \qquad y_0$$
$$\qquad\quad x_? \qquad a_{0?}$$
$$\qquad\qquad\quad x_?$$
$$\Longrightarrow \qquad \Longrightarrow \qquad \Longrightarrow$$

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

$$x_? \quad a_{0?} \quad y_0 \quad x_? \quad \Longrightarrow \quad x_? \quad \Longrightarrow \quad a_{0?} \quad \Longrightarrow \quad y_0 \quad \Longrightarrow$$
$$x_? \qquad\qquad a_{0?} \qquad\qquad x_?$$

Inefficient cache use

SpMV multiplication using CRS, LRU cache perspective:

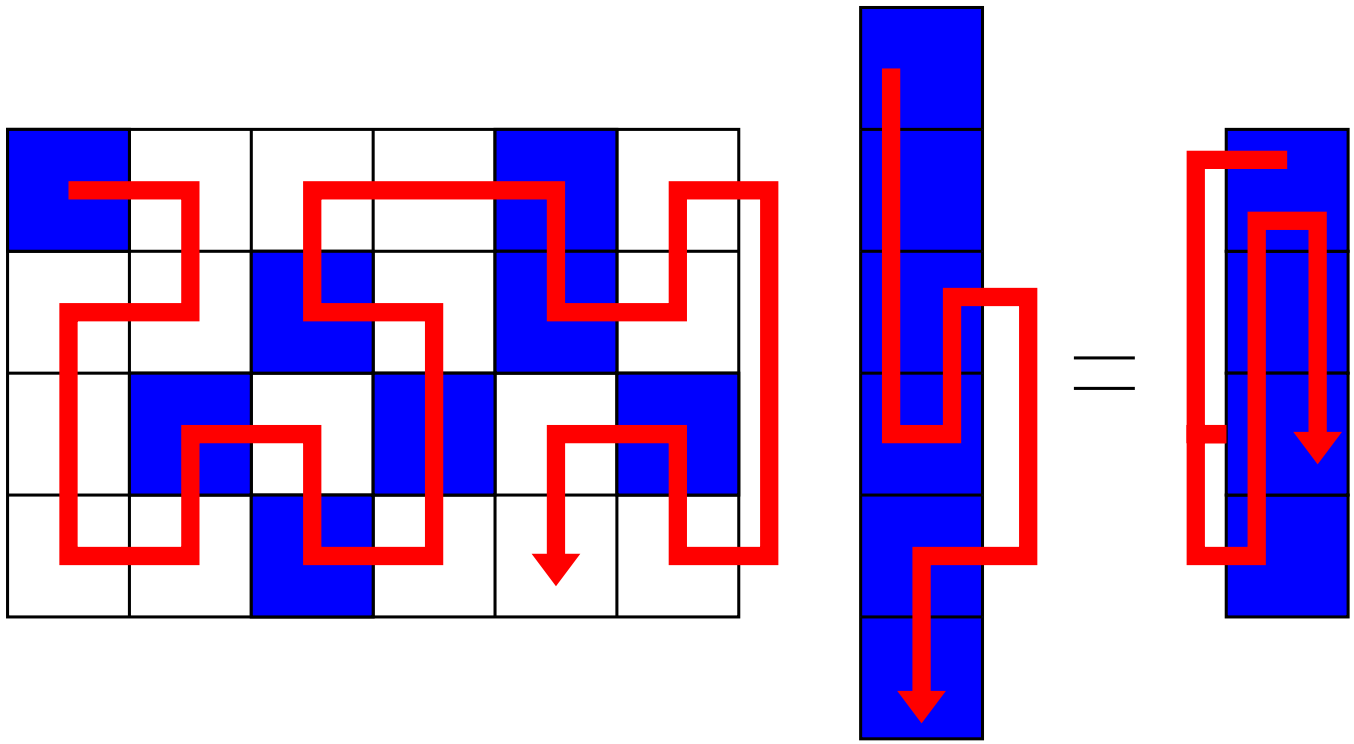
$$x_? \Longrightarrow \begin{matrix} a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} y_0 \\ a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} x_? \\ y_0 \\ a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} a_{??} \\ x_? \\ y_0 \\ a_{0?} \\ x_? \end{matrix} \Longrightarrow \begin{matrix} y_? \\ a_{??} \\ x_? \\ y_? \\ a_{0?} \\ x_? \end{matrix}$$

We cannot predict memory accesses in the sparse case:

- simple blocking is not possible.

Inefficient cache use

Visualisation of the SpMV multiplication $Ax = y$ with nonzeroes processed in an order defined by the **Hilbert curve**:



Accesses on both vectors have more **temporal locality**.

Bandwidth issues

The arithmetic intensity of an SpMV multiply lies between

$$\frac{2}{4} \text{ and } \frac{2}{5} \text{ flop per byte.}$$

On an 8-core 2.13 GHz (with AVX), and 10.67 GB/s DDR3:

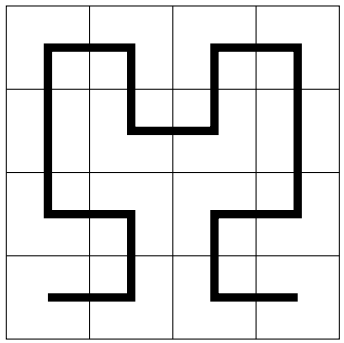
	CPU speed	Memory speed
1 core	$8.5 \cdot 10^9$ nz/s	$4.3 \cdot 10^9$ nz/s
8 cores	$68 \cdot 10^9$ nz/s	$4.3 \cdot 10^9$ nz/s

SpMV multiplication: clearly bandwidth-bound on modern CPUs.

Sparse matrix storage

The coordinate format stores nonzeroes in arbitrary order:

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



COO:

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

Storage requirements:

$$\Theta(3nz),$$

where nz is the number of nonzeroes in A.

Sparse matrix storage

Assuming a row-major order of nonzeroes enables **compression**:

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

CRS:

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

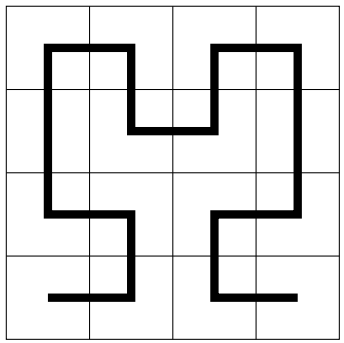
Storage requirements:

$$\Theta(2nz + m + 1).$$

SpMV multiplication

Multiplication using COO:

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



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

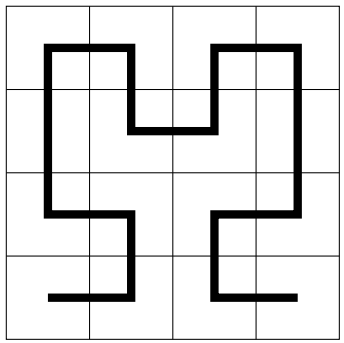
for $k = 0$ **to** $nz - 1$ **do**

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

SpMV multiplication

Multiplication using COO:

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



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

#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}

Is this OK?

SpMV multiplication

Multiplication using CRS:

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

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

Sequential kernel:

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

SpMV multiplication

Multiplication using CRS:

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

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

```
#omp parallel for private( i, k ) schedule( dynamic, 8 )
for i = 0 to m − 1 do
   for k = Î_i to Î_{i+1} − 1 do
      add V_k · x_{J_k} to y_i
```

SpMV multiplication

Parallel multiplication using Compressed Sparse Blocks:

$$A = \begin{pmatrix} A_{0,0} & \cdots & A_{0,n/\beta-1} \\ \vdots & \ddots & \\ A_{m/\beta-1,0} & \cdots & A_{m/\beta-1,n/\beta-1} \end{pmatrix},$$

cilk_for $i = 0$ **to** $m/\beta - 1$
  for $k = \hat{l}_i$ **to** $\hat{l}_{i+1} - 1$
    do block-local SpMV using A_{i,J_k},
      the J_kth block of x, and
      the ith block of y

Ref.: Buluç, Fineman, Frigo, Gilbert, and Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks", Proc. 21st annual symposium on Parallelism in algorithms and architectures, pp. 233-244 (2009)

Fine-grained parallelisation

The two previous SpMV multiplication algorithms are **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

Fine-grained parallelisation

The two previous SpMV multiplication algorithms are **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

The idea is that load-balancing, and scalability, are automatically attained by **run-time scheduling**.

- scalability is limited only by the amount of parallelism (i.e., the algorithmic span, or the critical path length).

Fine-grained parallelisation

The two previous SpMV multiplication algorithms are **fine-grained**.

- typically there are more rows than processes $m \gg p$, thus
- there are more tasks than processes.

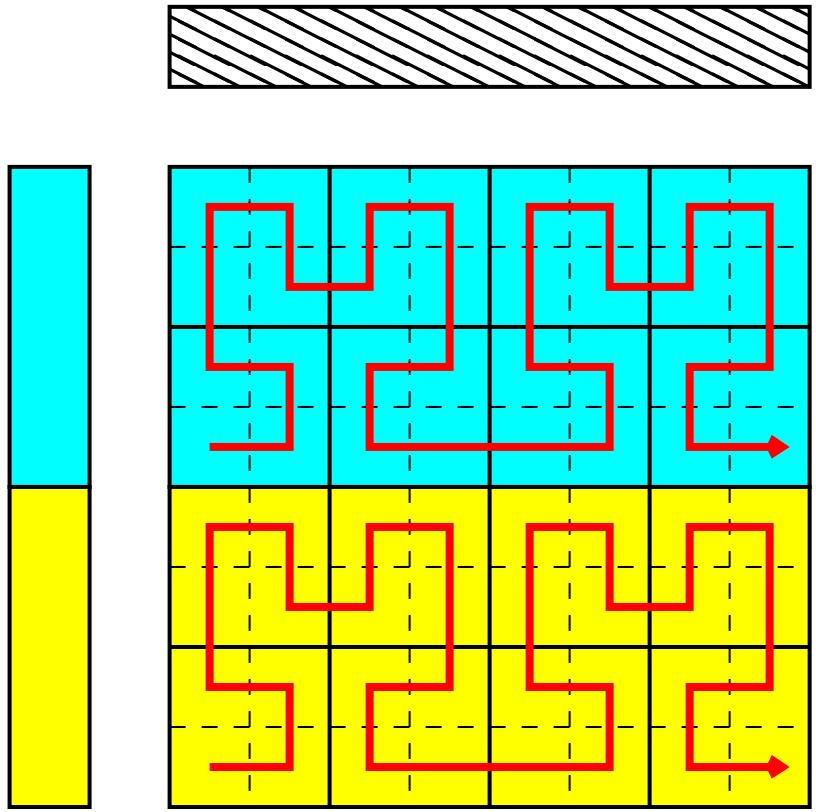
The idea is that load-balancing, and scalability, are automatically attained by **run-time scheduling**.

- scalability is limited only by the amount of parallelism (i.e., the algorithmic span, or the critical path length).

Requires implicit (interleaved) allocation of all data.

NUMA: Parallel 1D SpMV

Distribute rows to processes, do local blocking and Hilbert ordering:



Allows for explicit (local) allocation of the sparse matrix A and the output vector y; x is implicitly distributed and interleaved.

Ref.: Yzelman and Roose, "High-Level Strategies for Parallel Shared-Memory Sparse Matrix–Vector Multiplication", IEEE Trans. Parallel and Distributed Systems, doi: 10.1109/TPDS.2013.31 (2013).

NUMA: Parallel 1D SpMV

The SPMD code is still very simple. Initialisation:

- find which rows $I \subset \{0, \ldots, m-1\}$ are ours;
- order nonzeroes blockwise;
- impose a Hilbert-curve ordering on these blocks;
- allocate and store the local matrix $A^{(s)}$ (in the above order) using a compressed data structure;
- allocate a local $y^{(s)}$ (intialise to 0).

The input vector x is kept in global memory.

NUMA: Parallel 1D SpMV

The SPMD code is still very simple. Initialisation:

- find which rows $I \subset \{0, \ldots, m-1\}$ are ours;
- order nonzeroes blockwise;
- impose a Hilbert-curve ordering on these blocks;
- allocate and store the local matrix $A^{(s)}$ (in the above order) using a compressed data structure;
- allocate a local $y^{(s)}$ (intialise to 0).

The input vector x is kept in global memory. Multiplication:

- Execute $y^{(s)} = A^{(s)}x$.

Implemented in POSIX Threads.

NUMA: Parallel 2D SpMV

Sparse matrix partitioning **maps nonzeroes to processes**:

$$\pi_A : \{0, \ldots, m-1\} \times \{0, \ldots, n-1\} \to \{0, \ldots, p-1\}.$$

It also distributes the vectors x and y (by defining π_x and π_y).

If

- $a_{ij} \in A$ and $\pi_A(i,j) = s$, then we say a_{ij} **is local to process** s.
- a_{ij} is owned by process s but x_j is not, communication occurs.
- a_{ij} is owned by process s but y_i is not, communication occurs.

NUMA: Parallel 2D SpMV

Input vector communication:

- retrieving values from x is called **fan-out**, and
- is implemented by using **bsp_get**.
- Elements from x are communicated in a one-to-many fashion.

Output vector communication:

- sending contributions to non-local y is **fan-in**.
- Implementation happens through **Bulk Synchronous Message Passing** (BSMP).
- Elements from y are communicated in a many-to-one fashion.

Parallel 2D SpMV: BSMP

A BSMP message consists of two parts:

- an arbitrarily-sized **payload**, and
- a fixed-size identifier **tag**.

BSPlib is "buffered on source, buffered on receive":

- When sending a BSMP message, source data is **copied** in the **outgoing communications queue**.
- When receiving a BSMP message, the message is put in an **incoming queue** (during the communication phase).

Parallel 2D SpMV: BSMP

A BSMP message consists of two parts:

- an arbitrarily-sized **payload**, and
- a fixed-size identifier **tag**.

BSPlib is "buffered on source, buffered on receive":

- When sending a BSMP message, source data is **copied** in the **outgoing communications queue**.
- When receiving a BSMP message, the message is put in an **incoming queue** (during the communication phase).

(Dual-buffering also occurs for the `bsp_put` and `bsp_get`.)

Parallel 2D SpMV: BSMP

A BSMP **message** is a pair

$$(tag, payload).$$

The **bsp_send**$(s, tag, payload)$ primitive sends $(tag, payload)$ to process s.

- In BSPlib, the message is sent only during the next call to **bsp_sync()**.
- In BSPlib, messages arrive at their destination **in an unspecified order**.

Parallel 2D SpMV: BSMP

After a **bsp_sync**, messages may have arrived at process s.

- **bsp_qsize** checks how many messages are in queue.
- **bsp_get_tag** retrieves the tag of the first message in queue.
- **bsp_move** returns the payload of the first message, and removes this message from the queue.

The **bsp_set_tagsize** controls the size of the BSMP tag.

- when called, each SPMD program must call it;
- the new tagsize should be the same across all SPMD programs;
- the new tagsize is in effect only after the next **bsp_sync**.

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then:

1: **for each** j s.t. $\exists a_{ij}$ local to s while x_j is not local **do**
2: $\quad$ *bsp_get* x_j from remote process
3: *bsp_sync()*

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then:

1: **for each** j s.t. $\exists a_{ij}$ local to s while x_j is not local **do**
2: $\quad$ *bsp_get* x_j from remote process
3: *bsp_sync()*
4: **for each** row i for which $\exists a_{ij}$ local to s **do**
5: $\quad$ **for each** a_{ij} that is local to s **do**
6: $\quad\quad$ add $a_{ij} \cdot x_j$ to y_i

NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then:

1: **for each** j s.t. $\exists a_{ij}$ local to s while x_j is not local **do**
2: $\quad$ *bsp_get* x_j from remote process
3: *bsp_sync()*
4: **for each** row i for which $\exists a_{ij}$ local to s **do**
5: $\quad$ **for each** a_{ij} that is local to s **do**
6: $\quad\quad$ add $a_{ij} \cdot x_j$ to y_i
7: $\quad$ **if** y_i is not local **then**
8: $\quad\quad$ *bsp_send* (y_i, i) to the owner of y_i
9: *bsp_sync()*

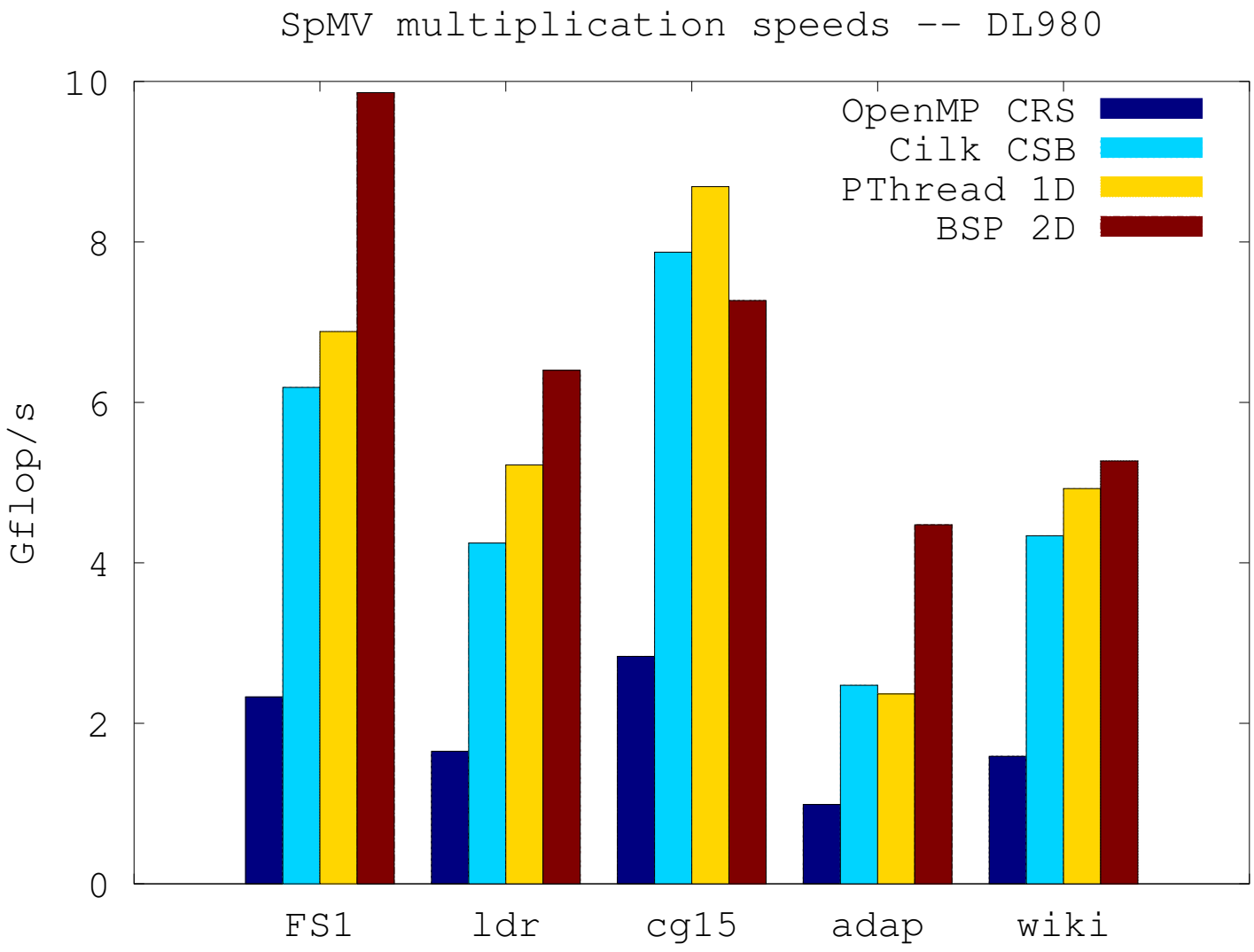
NUMA: Parallel 2D SpMV

Do sparse matrix partitioning as a **pre-processing** step. Then:

1: **for each** j s.t. $\exists a_{ij}$ local to s while x_j is not local **do**
2: $\quad$ *bsp_get* x_j from remote process
3: *bsp_sync()*
4: **for each** row i for which $\exists a_{ij}$ local to s **do**
5: $\quad$ **for each** a_{ij} that is local to s **do**
6: $\quad\quad$ add $a_{ij} \cdot x_j$ to y_i
7: $\quad$ **if** y_i is not local **then**
8: $\quad\quad$ *bsp_send* (y_i, i) to the owner of y_i
9: *bsp_sync()*
10: **while** *bsp_qsize()* > 0 **do**
11: $\quad$ $(\alpha, i) =$ *bsp_move()*
12: $\quad$ add α to y_i

everything is explicitly allocated!

Results


SpMV multiplication speeds -- DL980
Gflop/s
0
2
4
6
8
10
FS1
ldr
cg15
adap
wiki
OpenMP CRS
Cilk CSB
PThread 1D
BSP 2D

Fine-grained parallelism

Speedup

Definition (Speedup)

Let T_{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

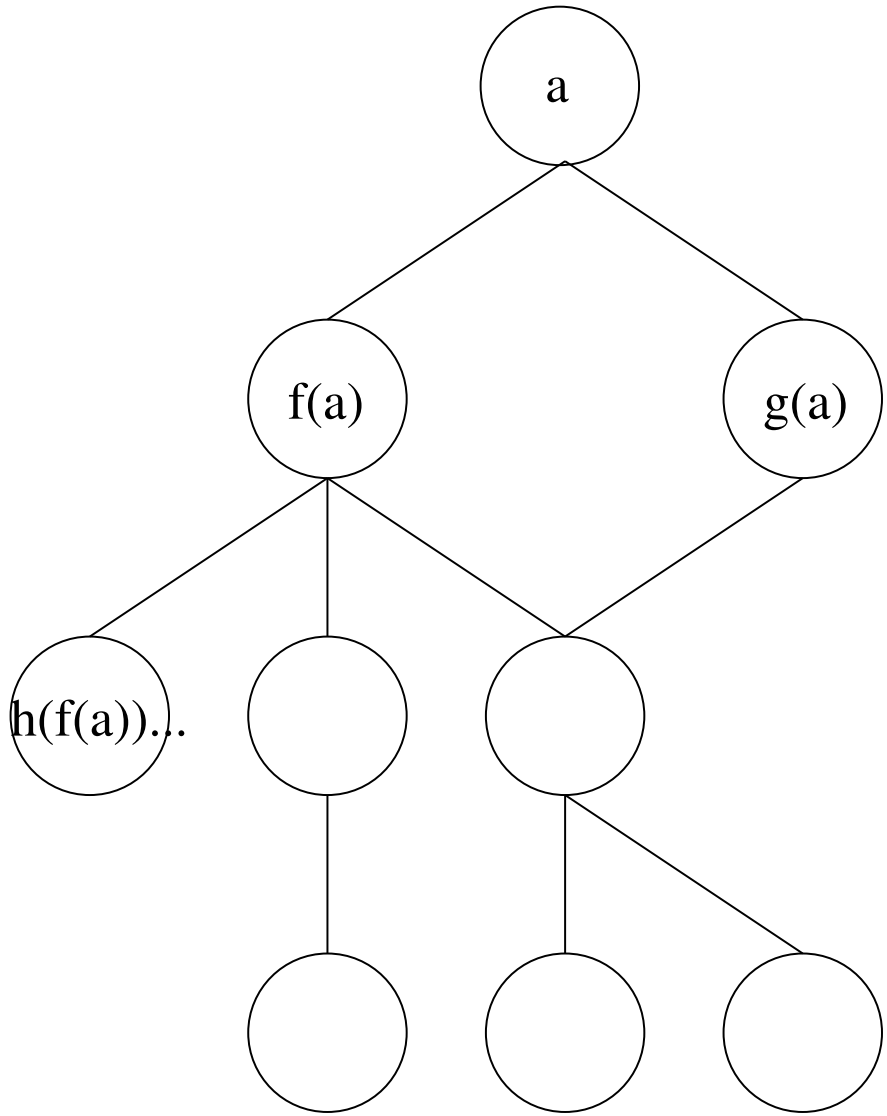
$$S = T_{\text{seq}} / T_p.$$

Scalable in time:

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

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.,

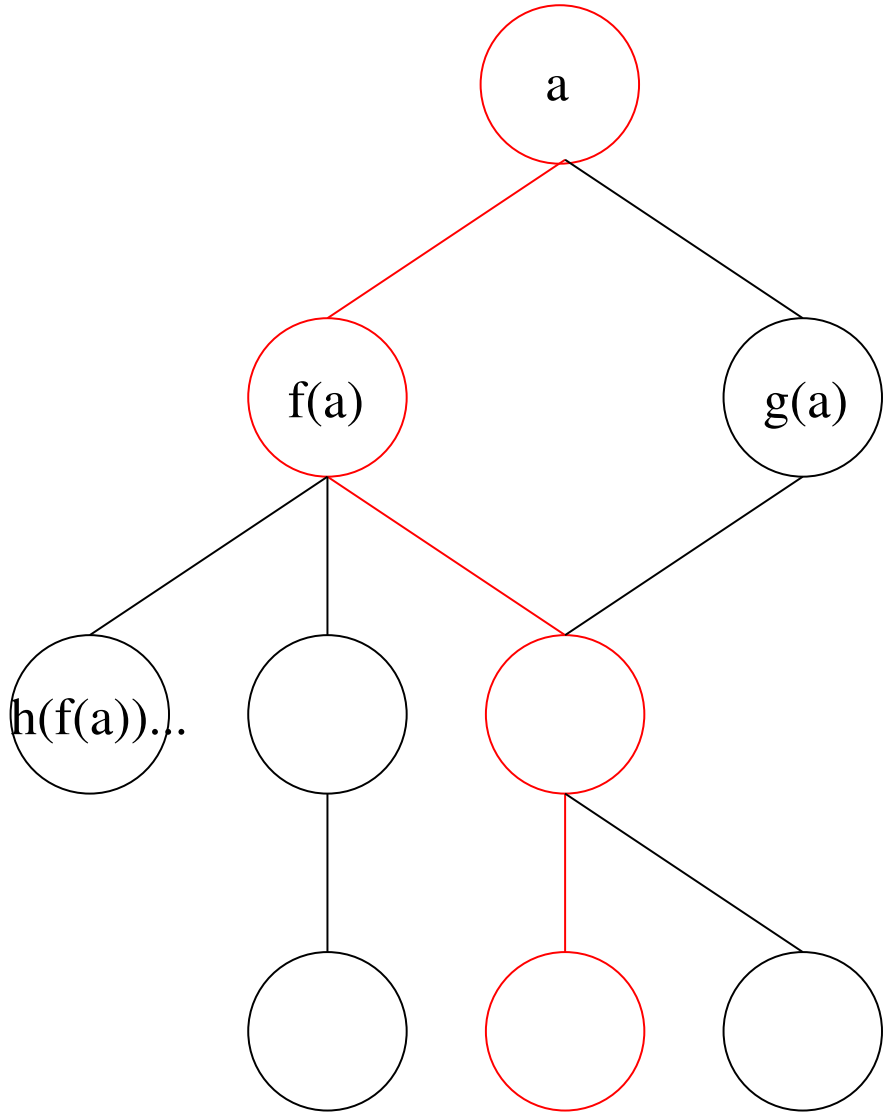
- Nodes correspond to data, edges indicate which data is combined to generate a certain output.

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_∞ 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_∞ 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**:

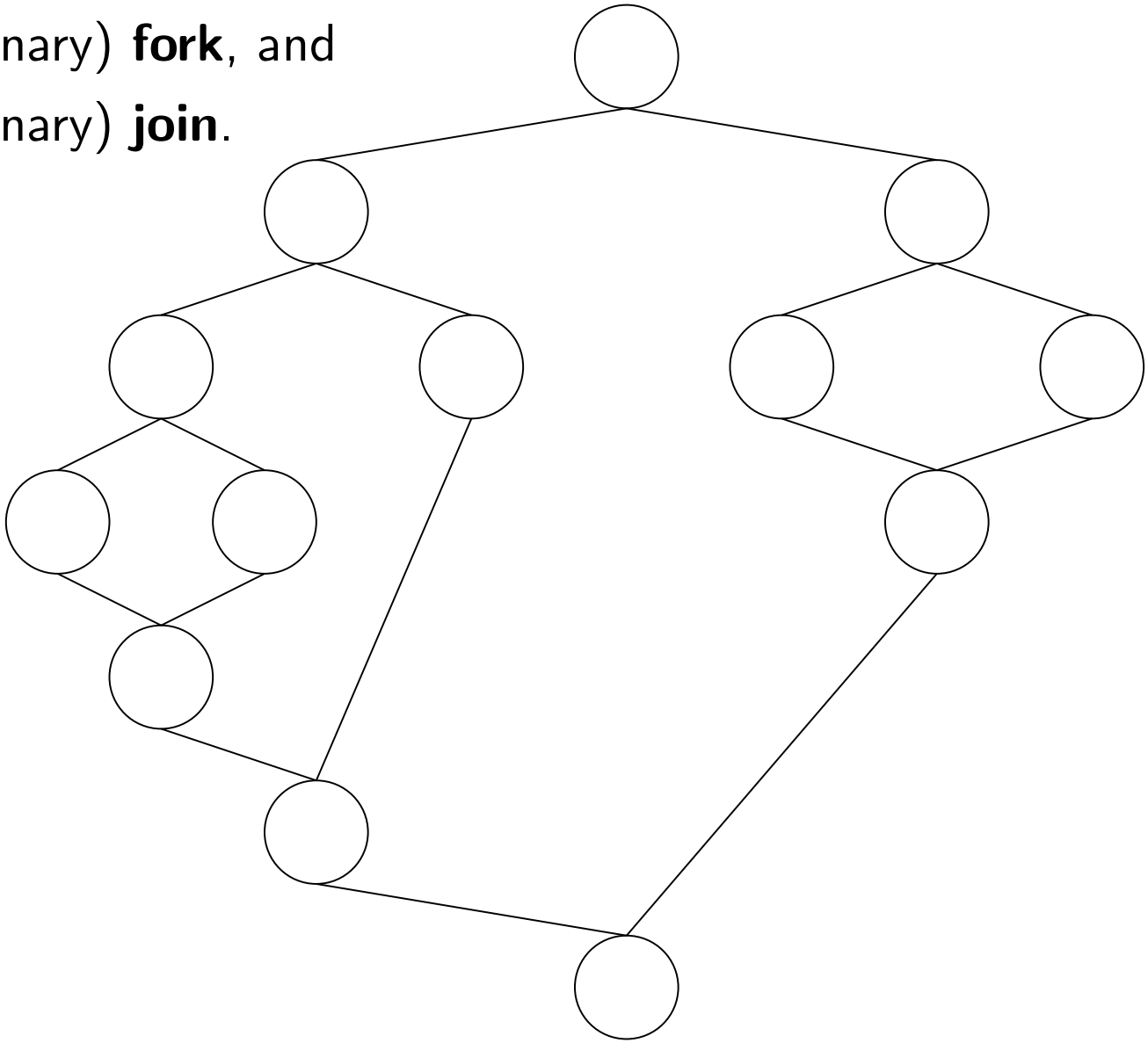
- Intel: Threading Building Blocks (TBB),
- **OpenMP**,
- Intel / MIT / Cilk Arts: **Cilk**,
- Google: **Pregel**,
- ...

By contrast, BSP computing is **coarse-grained**.

Cilk

Only two parallel programming primitives:

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

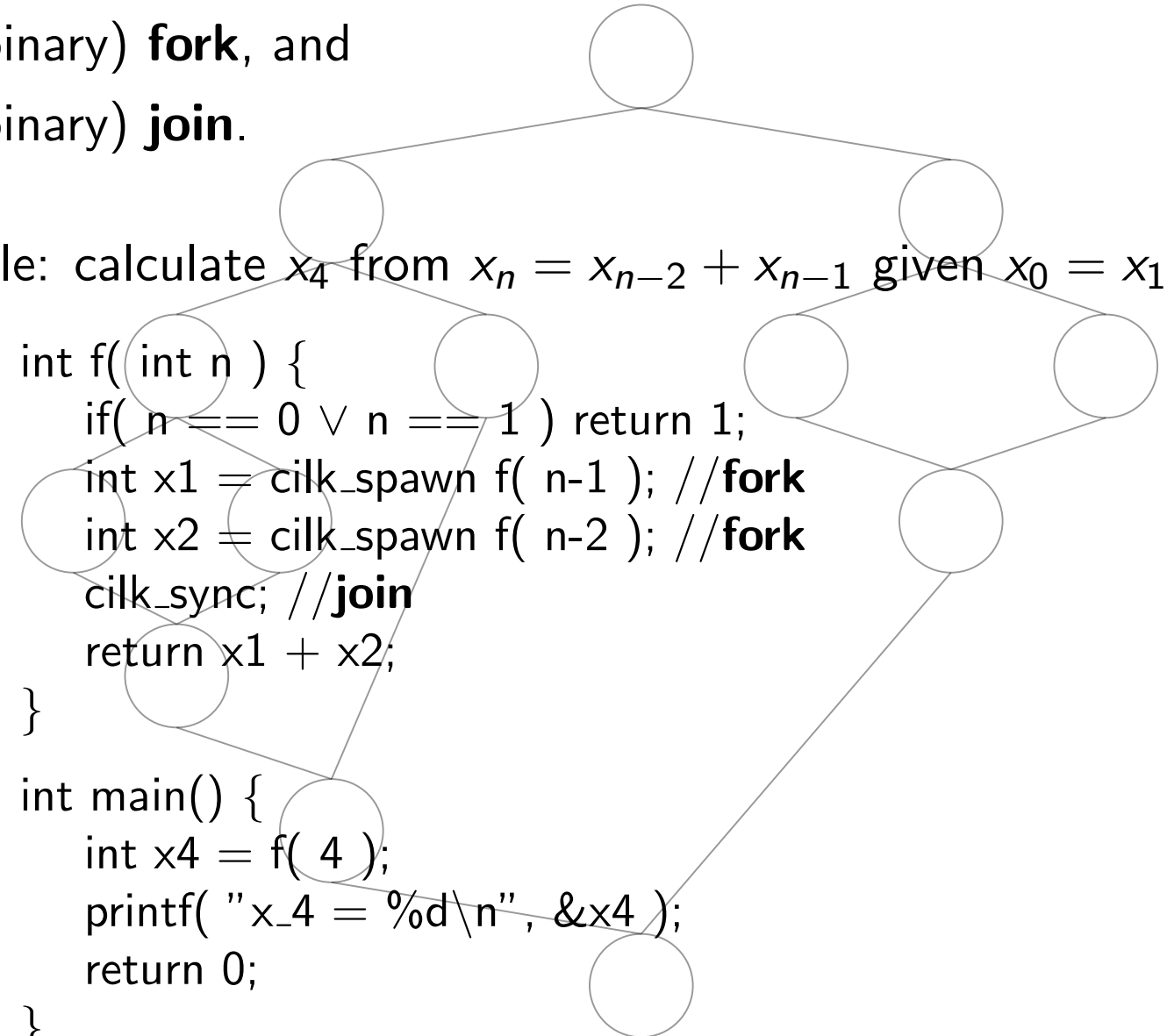


Cilk

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 $x_0 = x_1 = 1$:



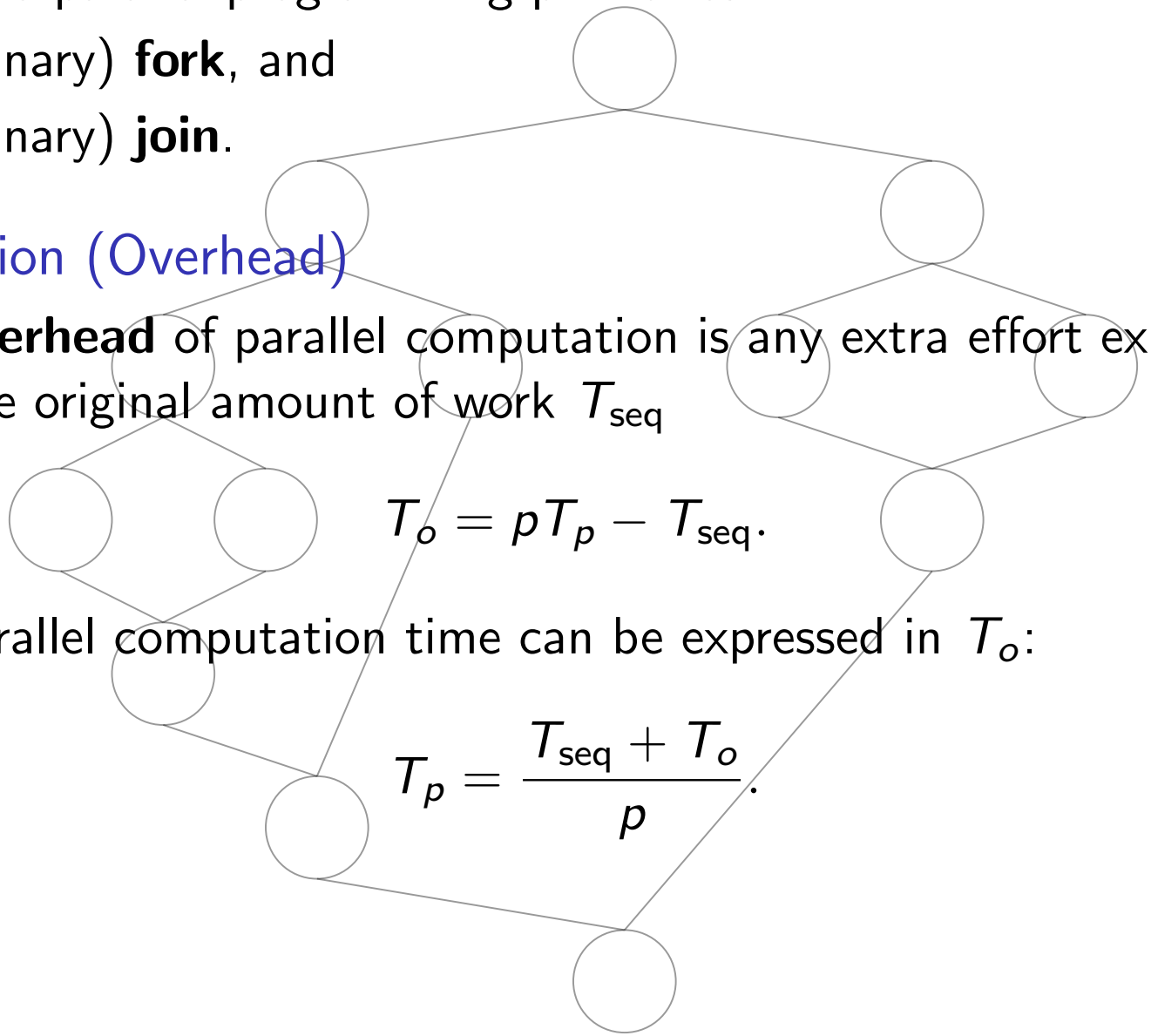
```
int f( int n ) {
    if( n == 0 ∨ n == 1 ) return 1;
    int x1 = cilk_spawn f( n-1 ); //fork
    int x2 = cilk_spawn f( n-2 ); //fork
    cilk_sync; //join
    return x1 + x2;
}

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}

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

The parallel computation time can be expressed in T_o:

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

Cilk

Only two parallel programming primitives:

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

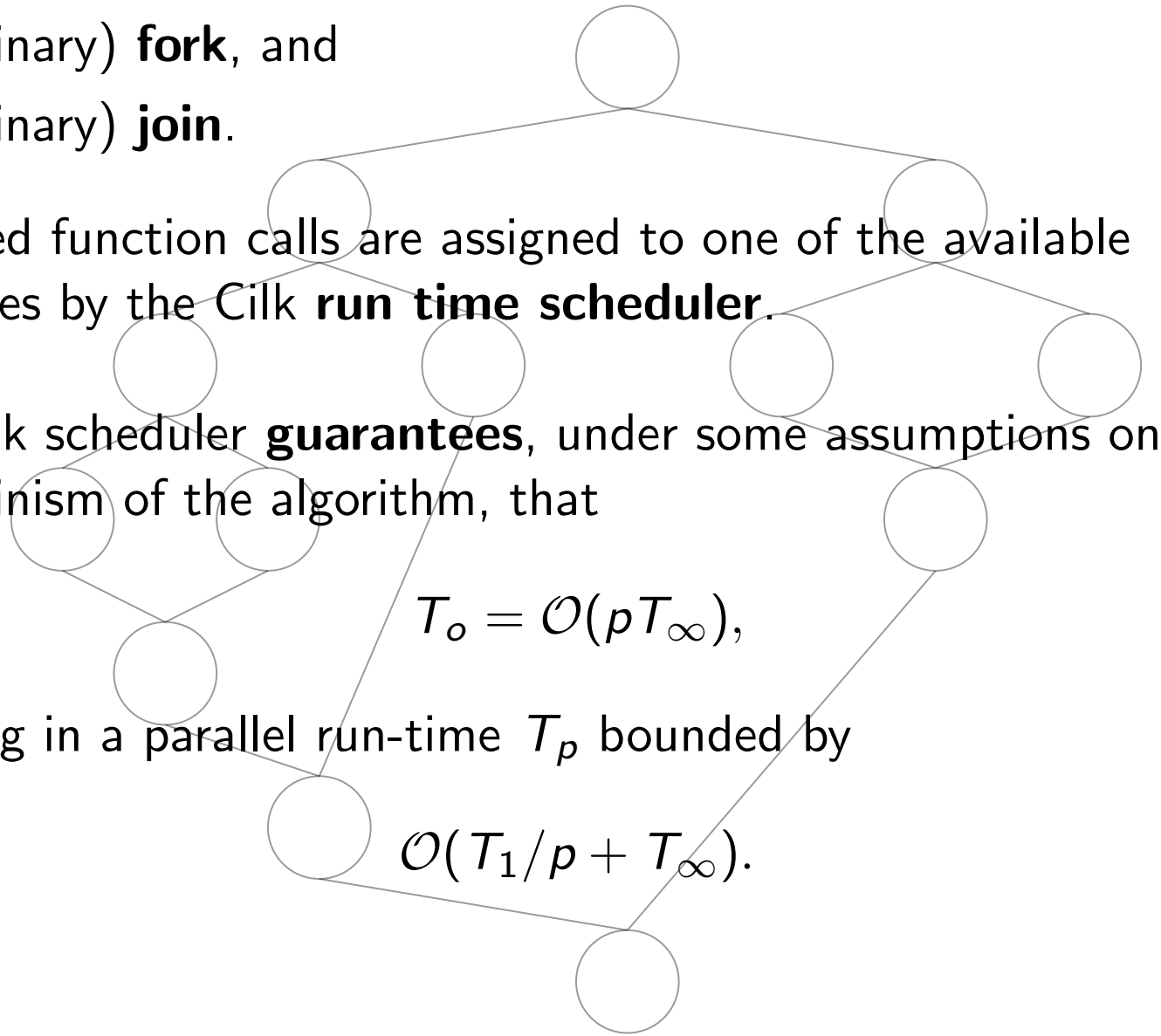
Spawned function calls are assigned to one of the available processes by the Cilk **run time scheduler**.

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

$$T_o = \mathcal{O}(pT_\infty),$$

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:

$$\text{map: } K \times V \to K \times V;$$

$$\text{reduce}(k)\text{: } \mathcal{P}(\{k\} \times V) \to K \times V, \quad k \in K.$$

- The map operation is **embarrasingly parallel**: every key-value pair is mapped to a new key-value pair, an entirely local operation.

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:

$$\text{map: } K \times V \to K \times V;$$

$$\text{reduce}(k)\text{: } \mathcal{P}(\{k\} \times V) \to K \times V, \quad k \in K.$$

- The map operation is **embarrasingly 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**.

MapReduce

Calculating $\alpha = x^T y$ using MapReduce:

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

1. Map: for each pair $(i, \{a, b\})$ write $(\text{partial}, a \cdot b)$. Applying this map adds $\{(\text{partial}, x_0 y_0), (\text{partial}, x_1 y_1), \ldots\}$ to S.
2. Reduce: for all pairs with key 'partial', combine their values by addition and store the result using key α. Applying this reduction adds $(\alpha, \sum_{i=0}^{n-1} x_i y_i)$ to S.
3. Done: S contains a single entry with key α 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_∞ 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_{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_{seq} be the sequential memory requirement and let M_p be the maximum memory requirement of a single parallel process. The parallel overhead in memory is

$$M_o = pM_p - M_{\text{seq}}, \text{ or, rewritten:}$$

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

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

Fine-grained summary

Answers:

- *Q: does more parallelism always mean better performance?*
- A: **no**.
  1. we typically have less than ∞ processors, and
  2. speedup versus the best available sequential algorithm may be disappointing.

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**...

Fine-grained summary

Answers:

- *Q: does more parallelism always mean better performance?*
- A: **no**.
  1. we typically have less than ∞ processors, and
  2. speedup versus the best available sequential algorithm may be disappointing.

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

$$S^{\text{odd-even sort}} = T_{\text{seq}}^{\textbf{qsort}} / T_p^{\text{odd-even sort}}.$$

Questions

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

```
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;
}
```

Questions:

- what is T_1 (asymptotically)?
- what is T_∞ (asymptotically)?
- what is T_{seq}?
- is this trivial parallelisation a good idea?

Questions

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

```
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;
}
```

Answers:

- $T_1 = \Theta(\phi^n) = \mathcal{O}(2^n)$,
- $T_\infty = \Theta(n)$,
- $T_{\text{seq}} = n!$, so
- huge parallelism $\mathcal{O}(2^n/n)$, no usability.

Questions

In the bulk synchronous parallel setting, how do the concepts such as span and overhead translate?

Question:

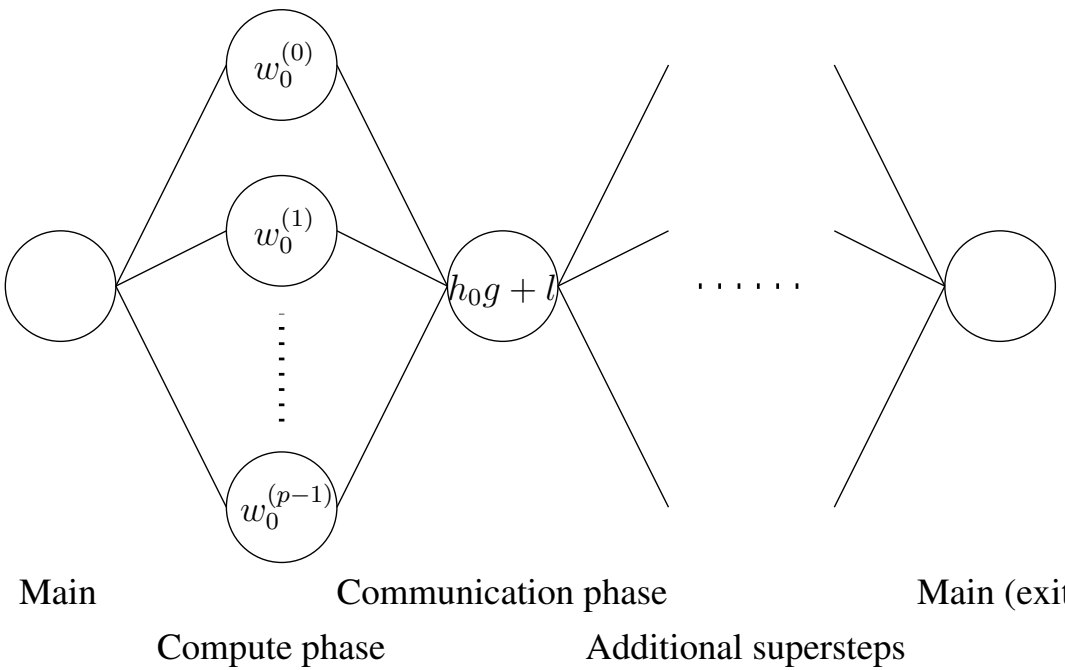
- how do the computation graph, T_∞, and T_o look for BSP?

Questions

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.



Questions

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

Span:

- Critical path follows compute phases with work $w_i^{(s)}$,
- communication costs are global and 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.

Questions

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

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

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

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

The Multi-BSP model

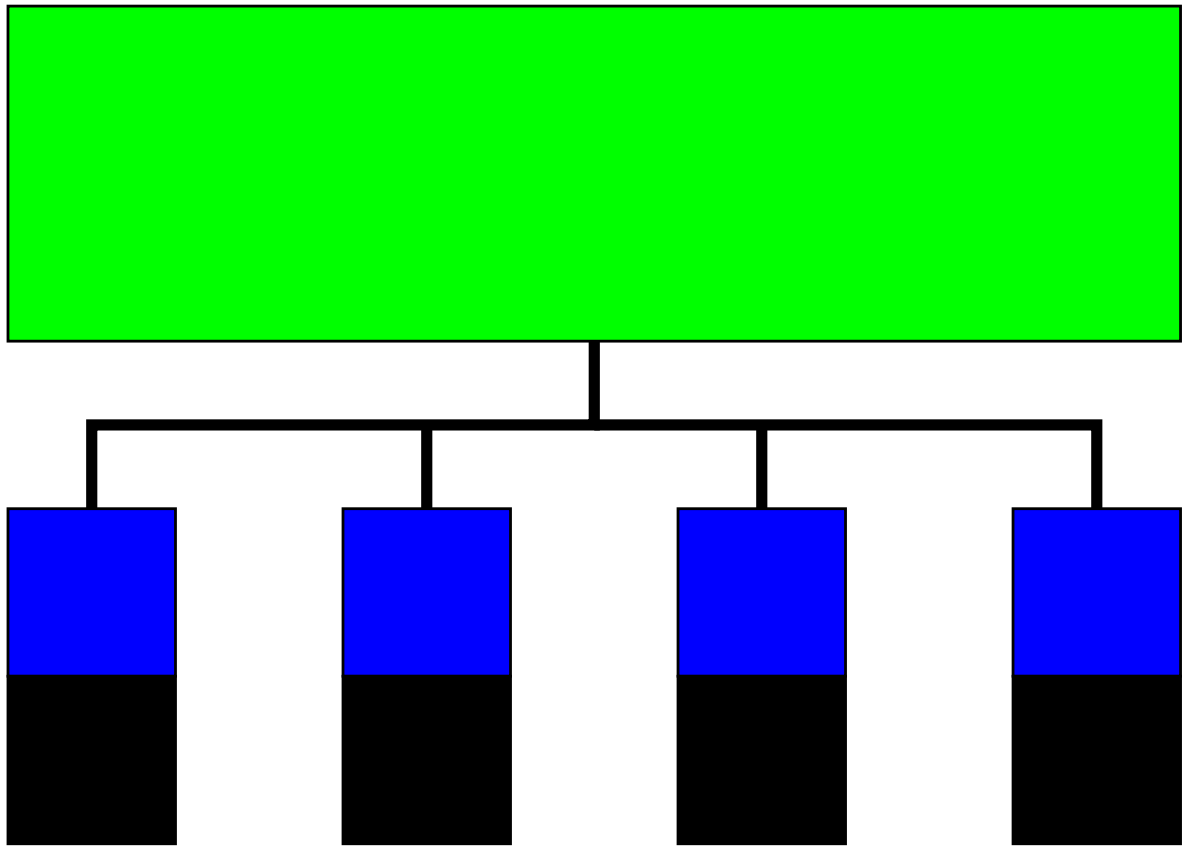
Three concepts

The Bulk Synchronous Parallel

1. computer,
2. algorithm,
3. cost model.

BSP computer

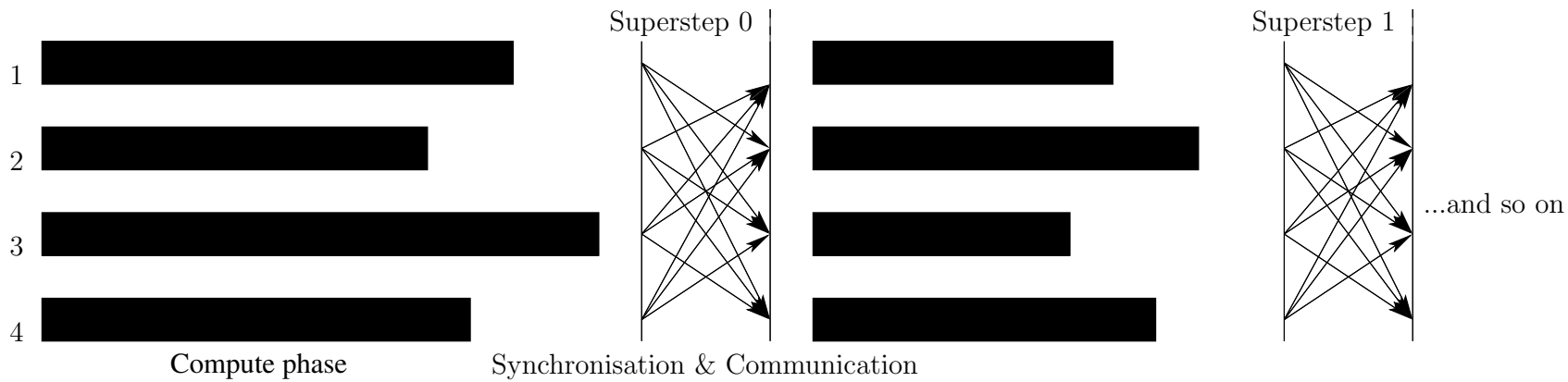
CPUs, memory, network.



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 ith superstep is

$$T_{\text{comp},i} = \max_s w_i^{(s)}.$$

The total **cost of communication** during the ith superstep is

$$T_{\text{comm,i}} = h_i g.$$

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

$$T = \sum_{i=0}^{N-1} \left(T_{\text{comp},i} + T_{\text{comm,i}} + l \right) = \sum_{i=0}^{N-1} \left(\max w_i^{(s)} + h_i g + l \right),$$

Multi-BSP

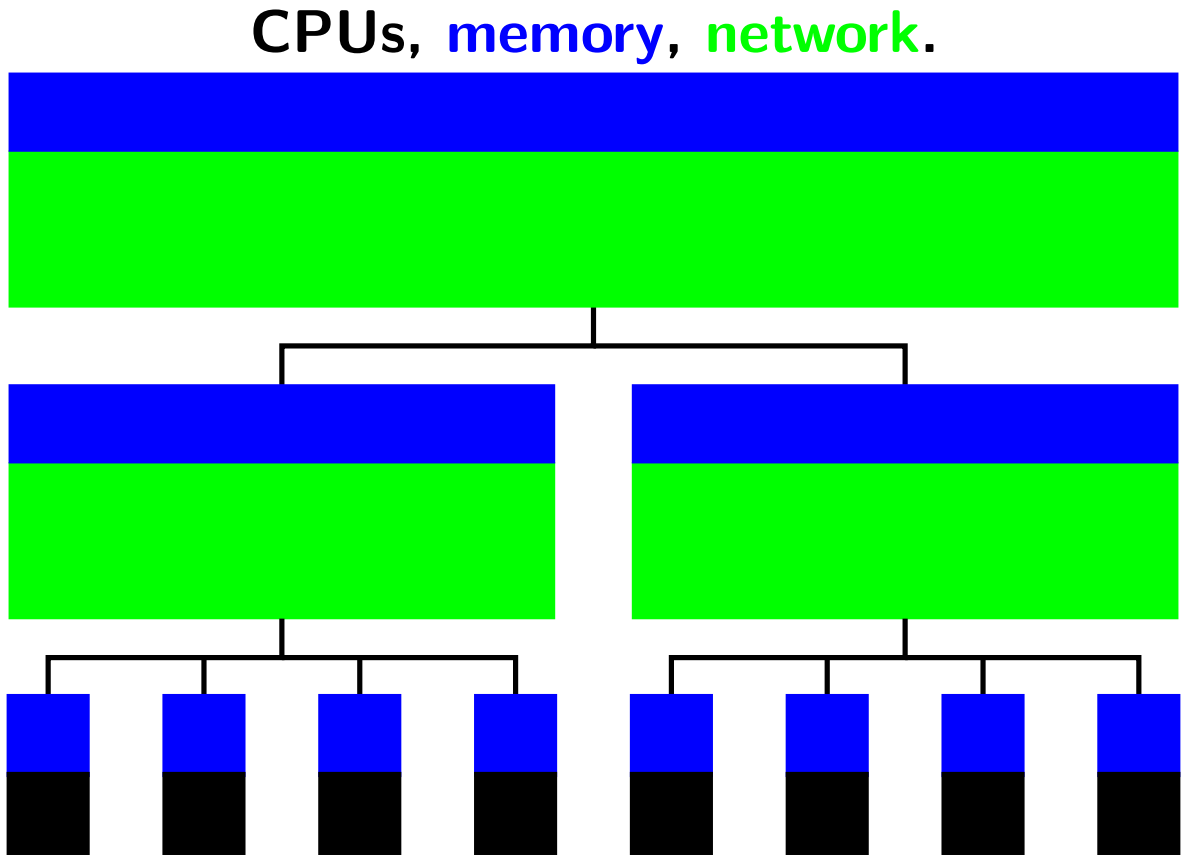
Multi-BSP is the recursive application of the BSP model.

- **BSP**: a computer consists of p CPUs/processors/cores/...
- **Multi-BSP**: a computer consists 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:

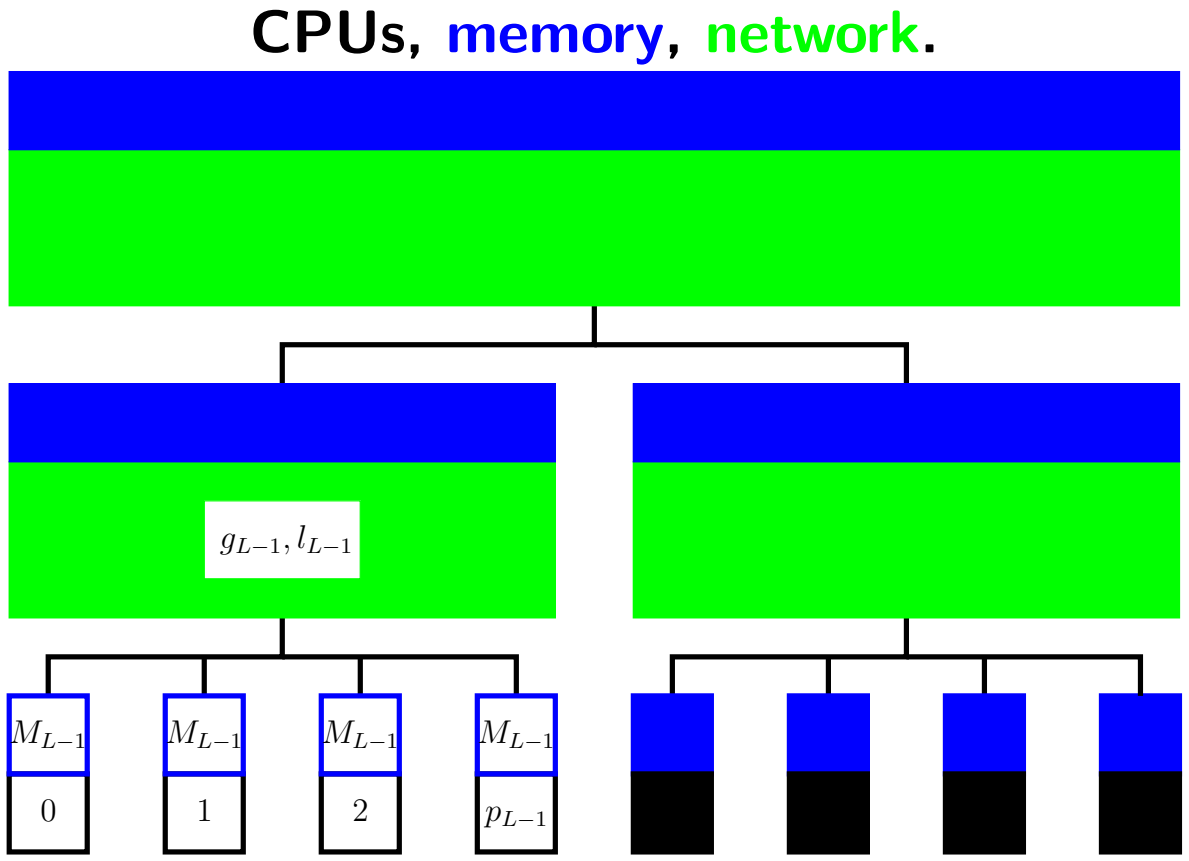
Valiant, Leslie G. "A bridging model for multi-core computing." Journal of Computer and System Sciences 77.1 (2011): 154-166.

Multi-BSP computer model, $L = 2$



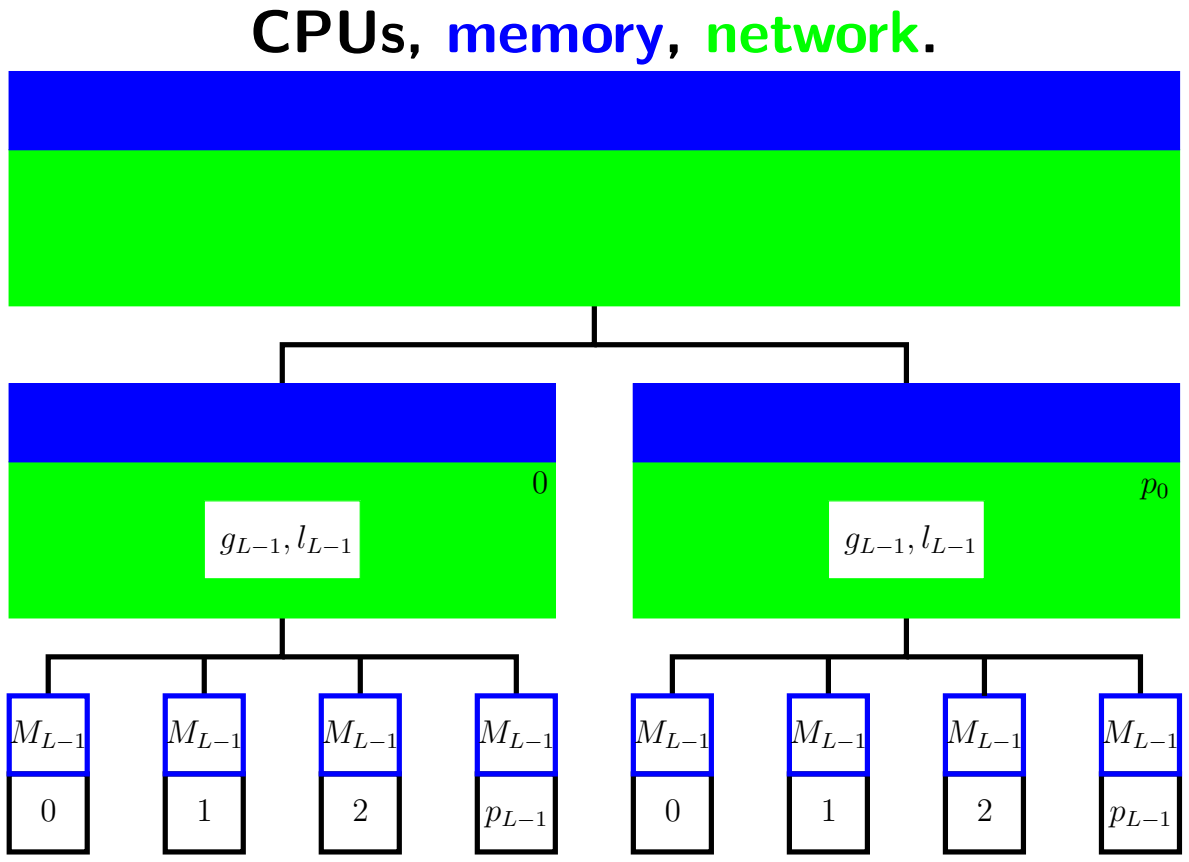
A BSP computer $(p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})$.

Multi-BSP computer model, $L = 2$



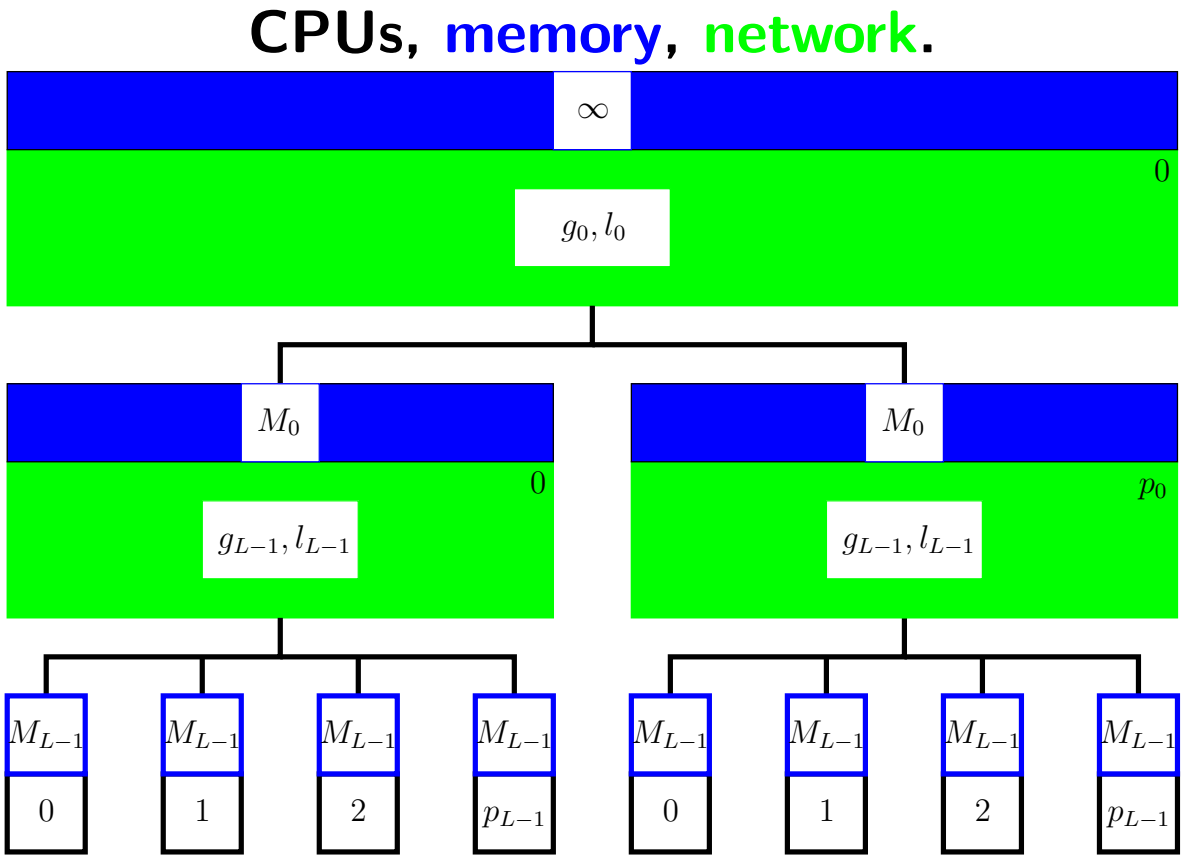
A BSP computer $(p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})$.

Multi-BSP computer model, $L = 2$



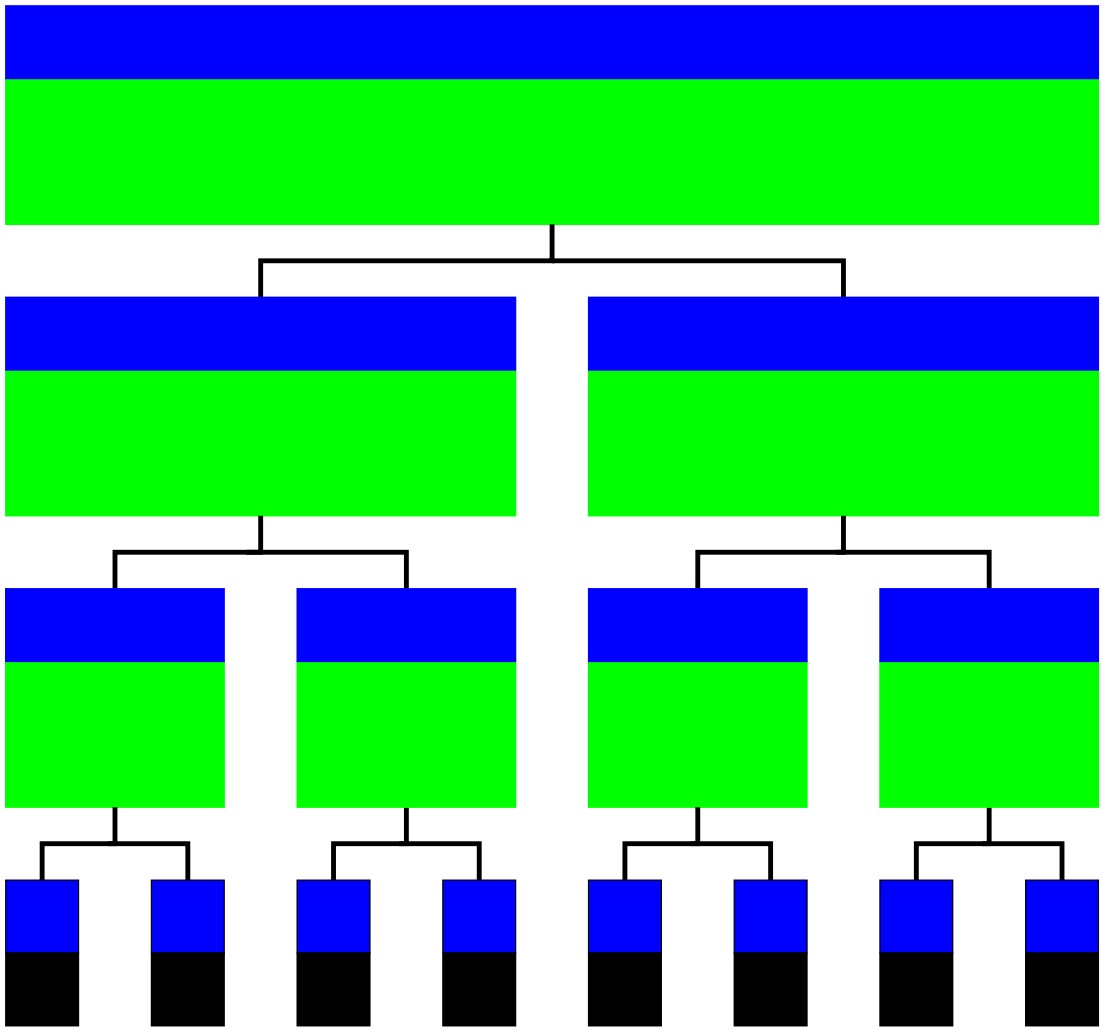
A BSP computer $(p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})$.

Multi-BSP computer model, $L = 2$

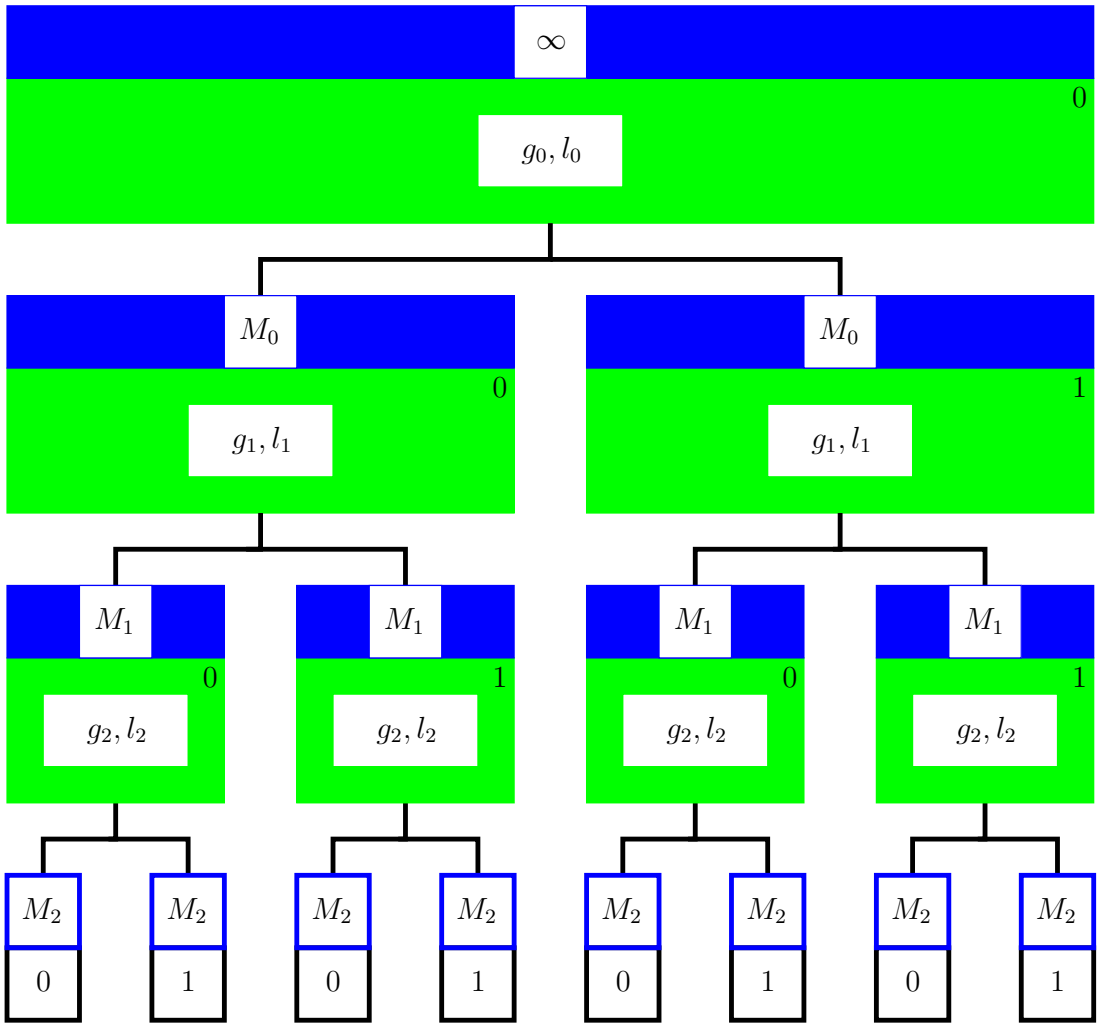


A BSP computer $(p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})$.

Multi-BSP computer model, $L = 3$



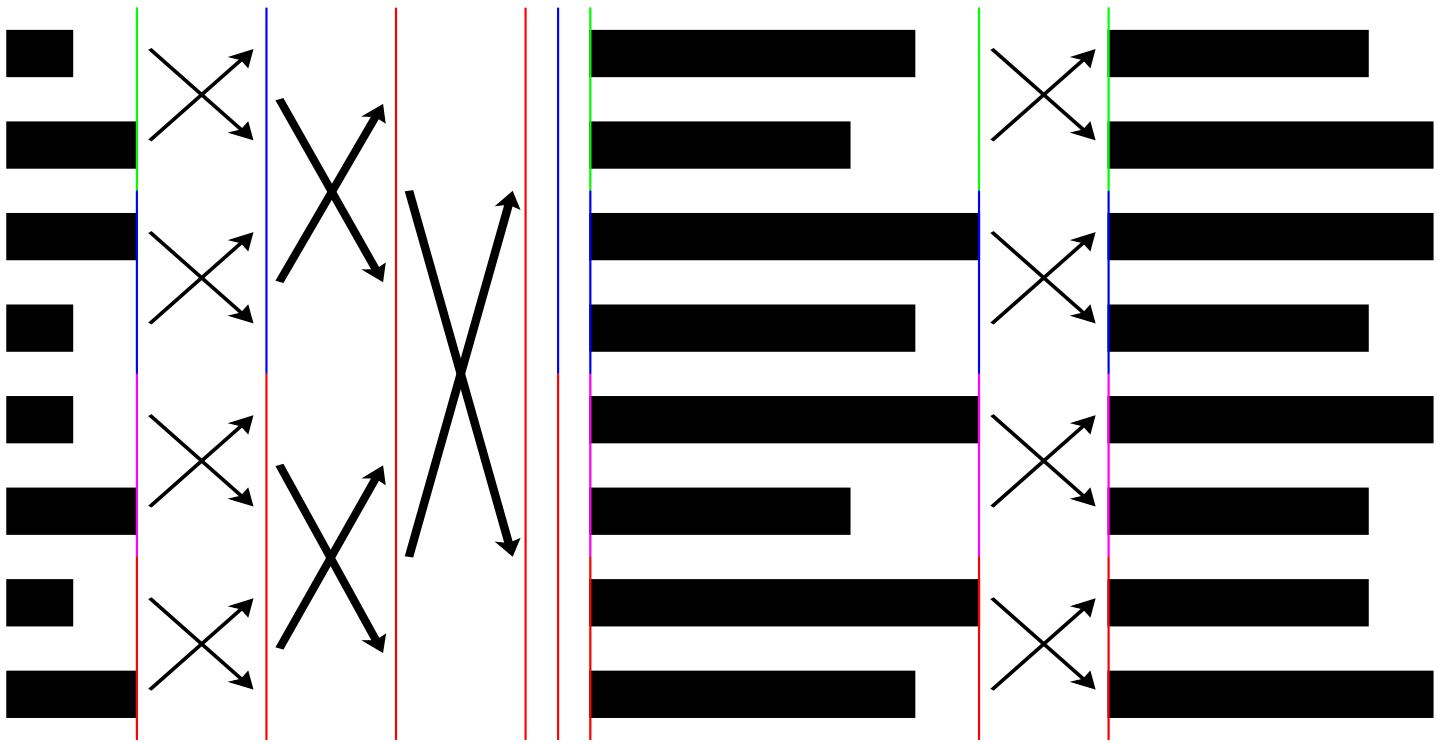
Multi-BSP computer model, $L = 3$



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 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 ith level,
- $h_{k,i}$: the **maximum of all h-relations** within the ith superstep on level k,
- $w_{k,i}$: the **maximum of all work** within the ith superstep on level k.

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

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

Motivation

- 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?

Motivation

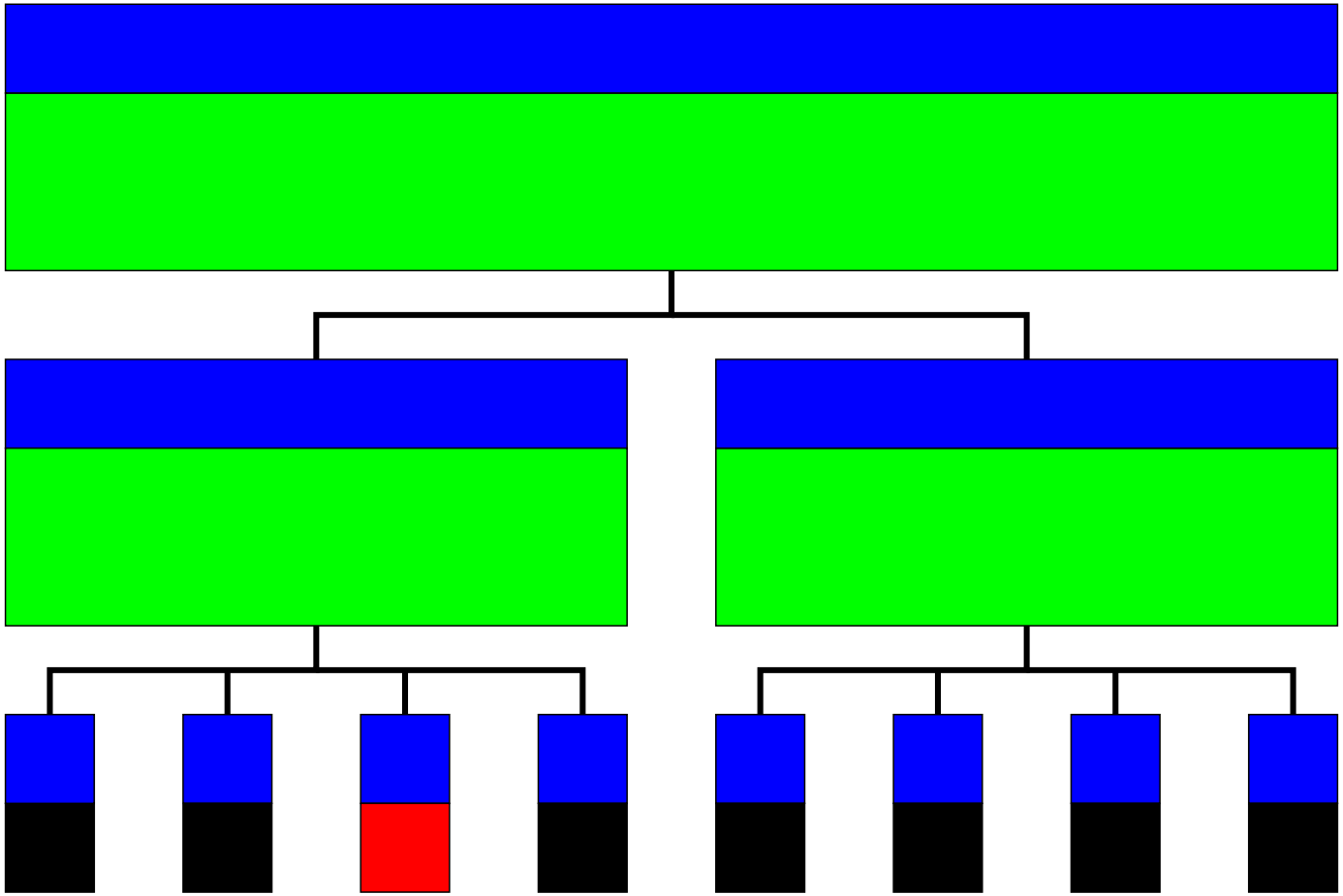
- Multi-BSP is **a better model** for modern parallel architectures. It closely resembles
  - contemporary shared-memory multi-socket machines,
  - multi-level shared and private cache architectures, and
  - multi-level network topologies (e.g., fat trees).
- Hierarchical modeling also has **drawbacks**. It is more difficult to
  - **prove optimality** of hierarchical algorithms, and
  - **portably** implement hierarchical algorithms.

Would you like to:

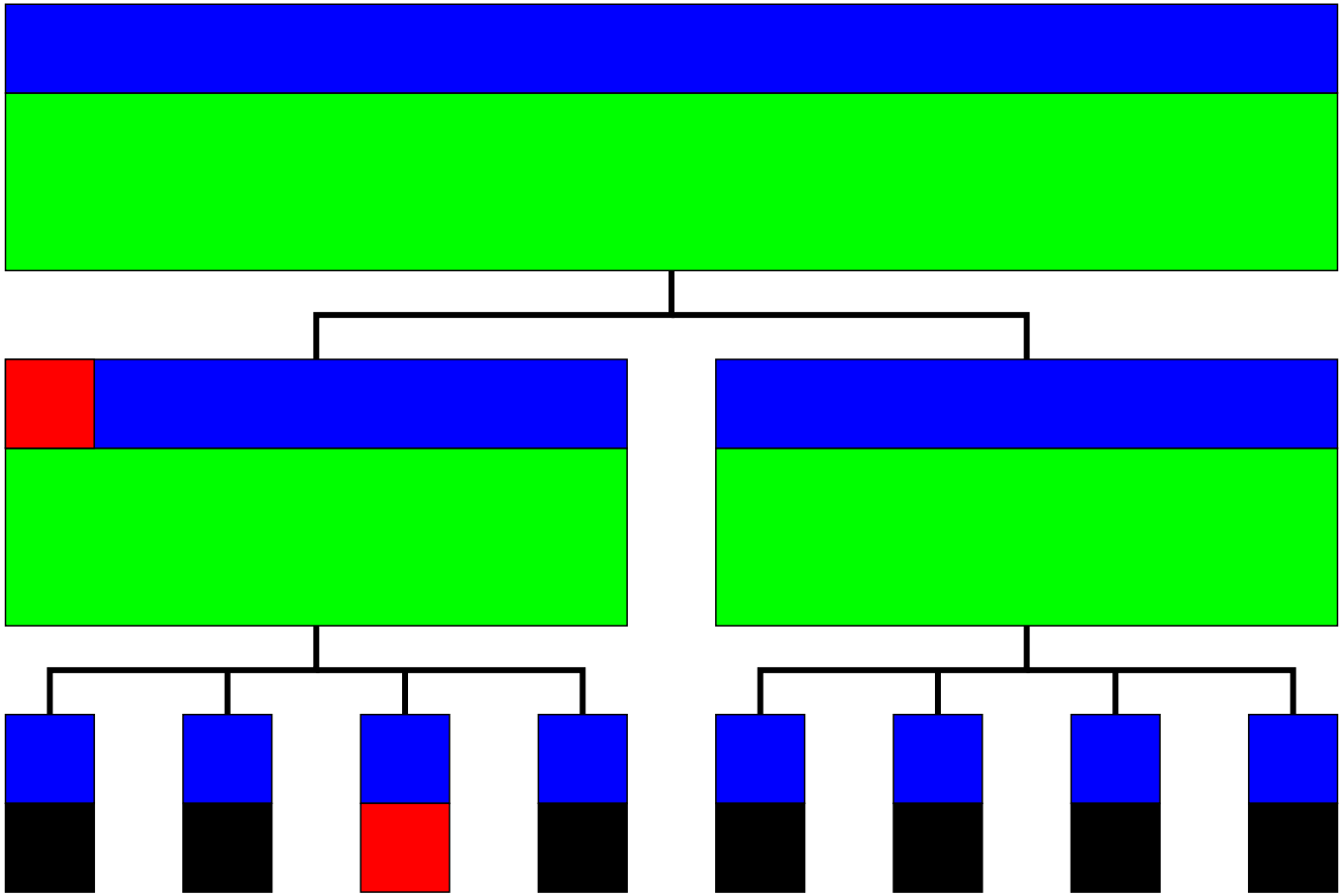
- prove optimality of an algorithm in 16 parameters?
- develop algorithms for a four-level machine?

Multi-BSP can actually **simplify** these issues!

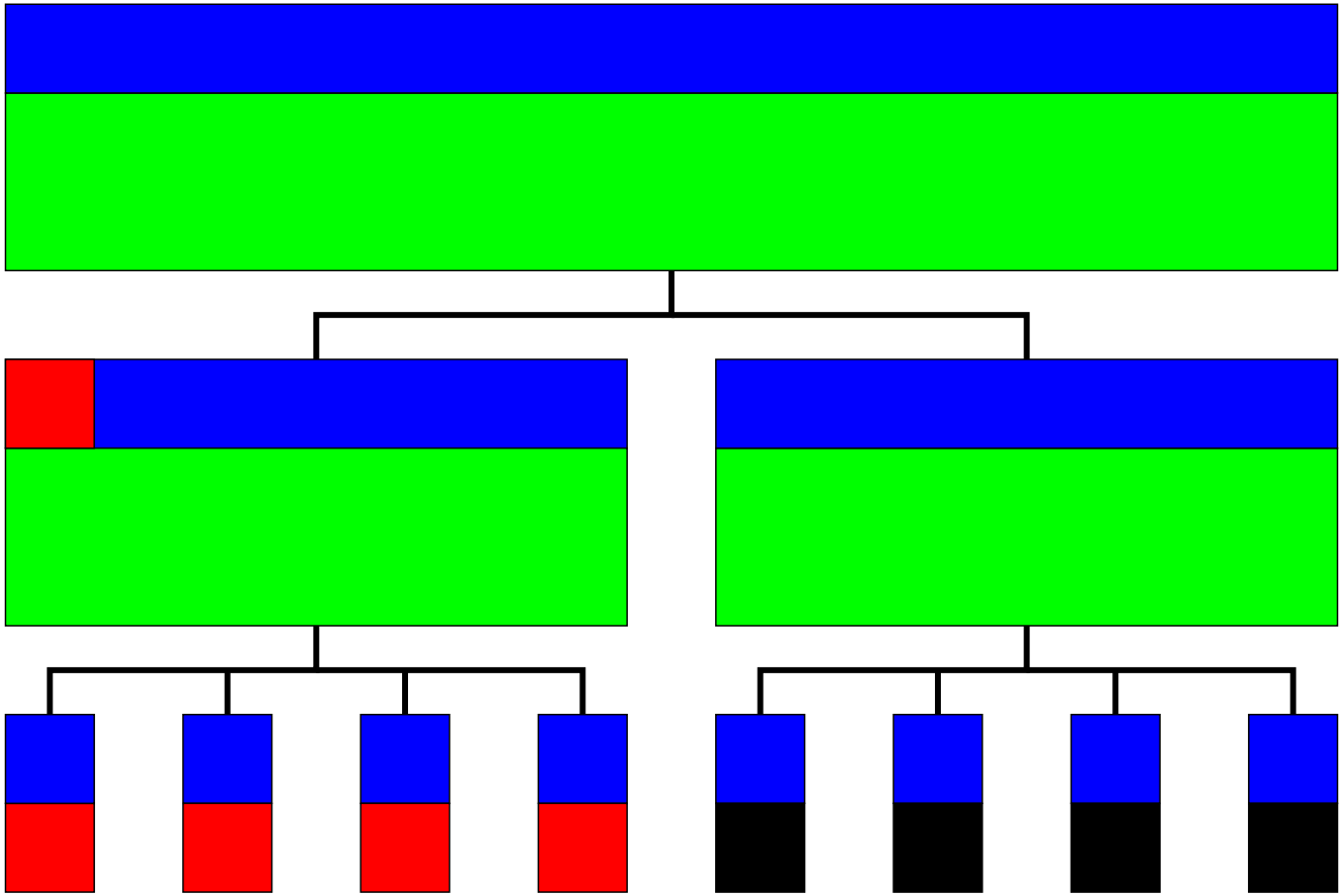
Multi-BSP: broadcasting



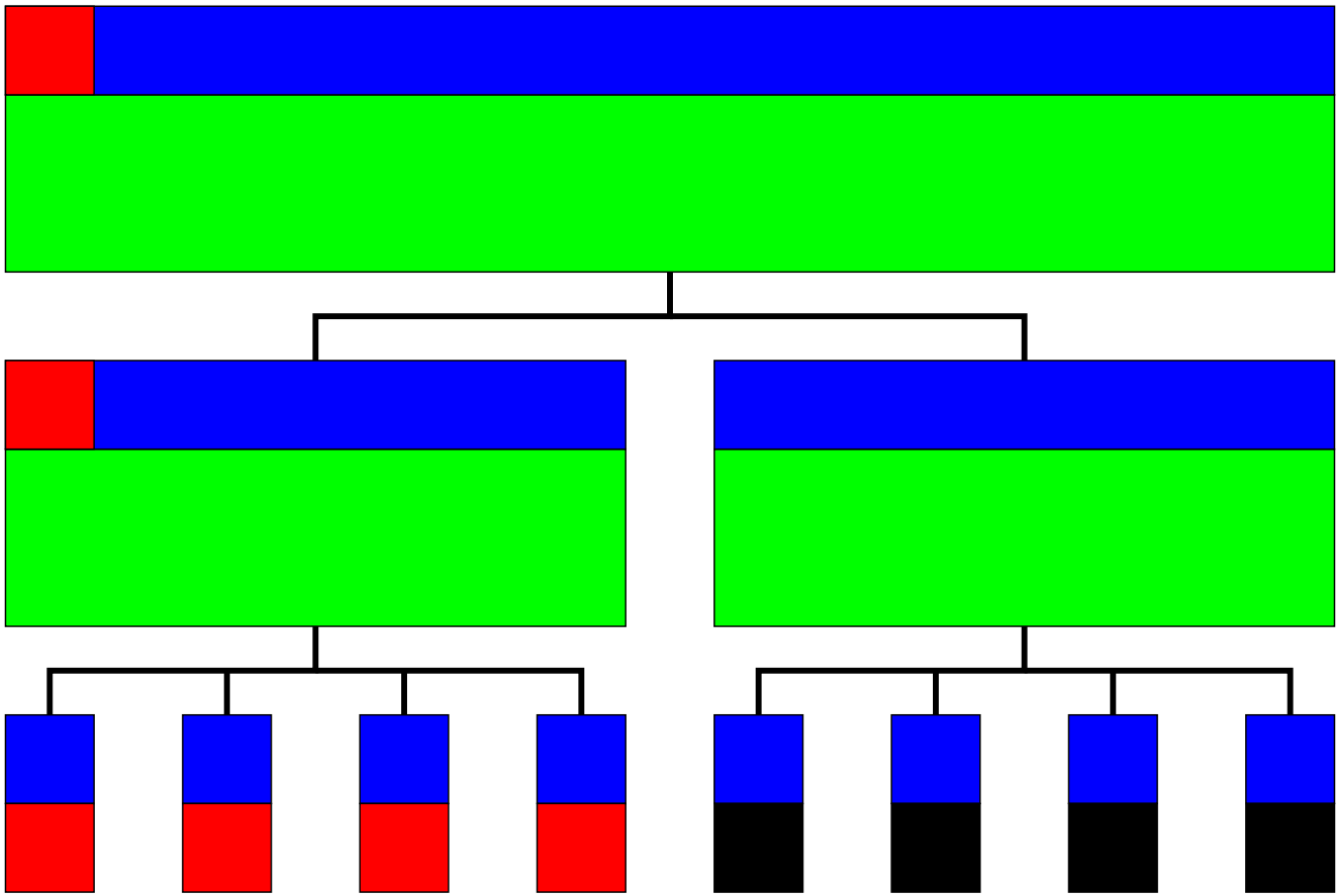
Multi-BSP: broadcasting



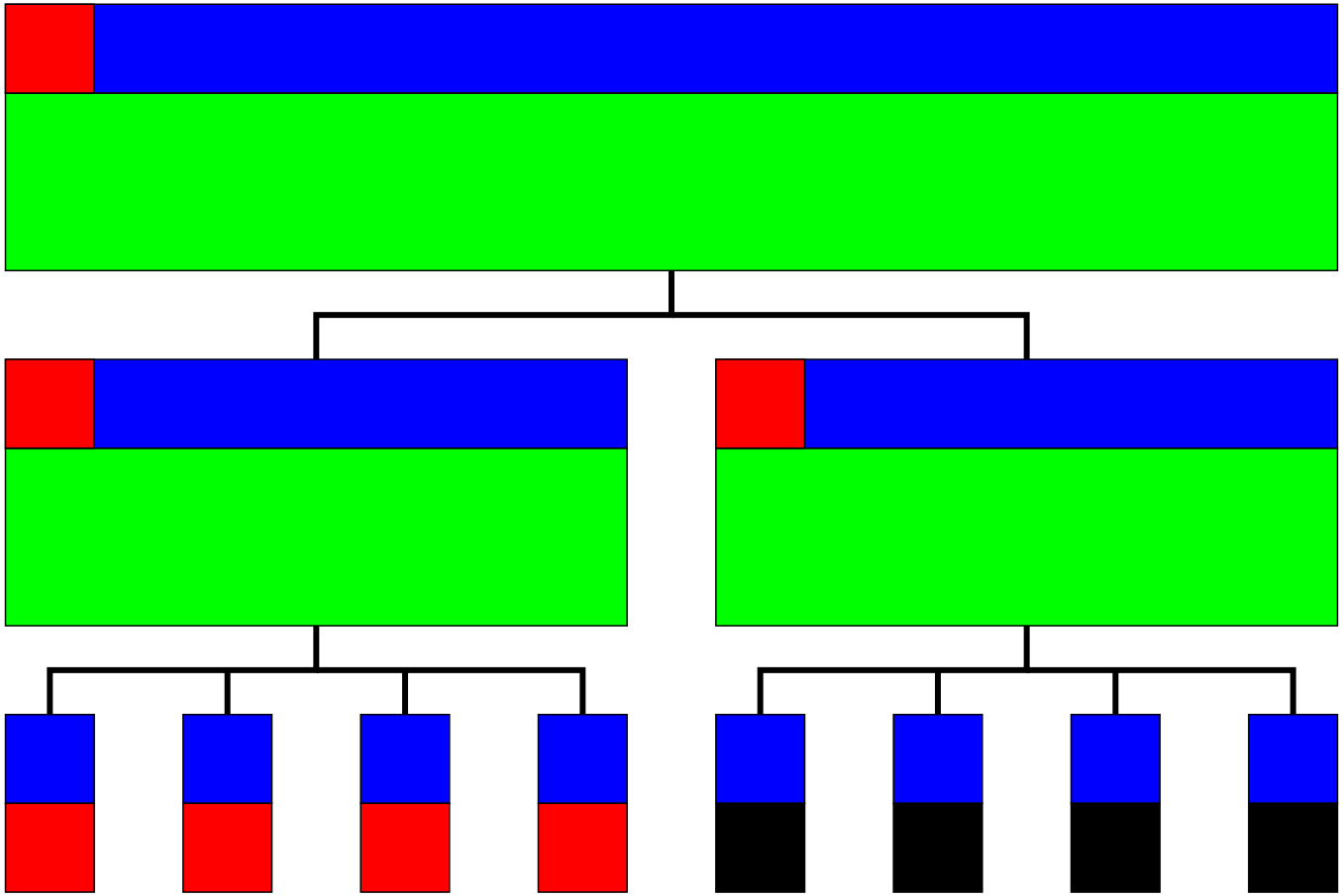
Multi-BSP: broadcasting



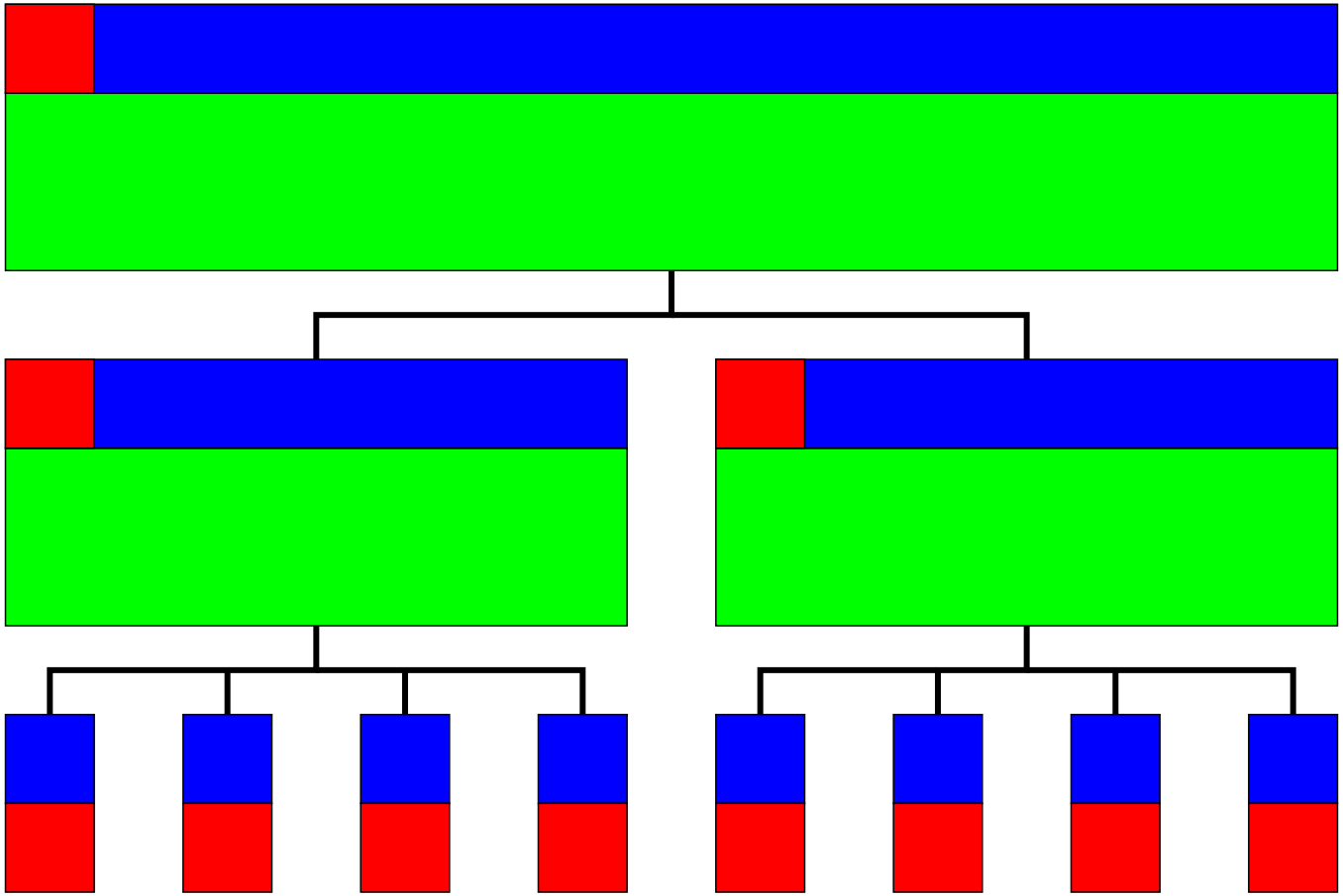
Multi-BSP: broadcasting



Multi-BSP: broadcasting



Multi-BSP: broadcasting



Multi-BSP: broadcasting

Question: can we do better?

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.

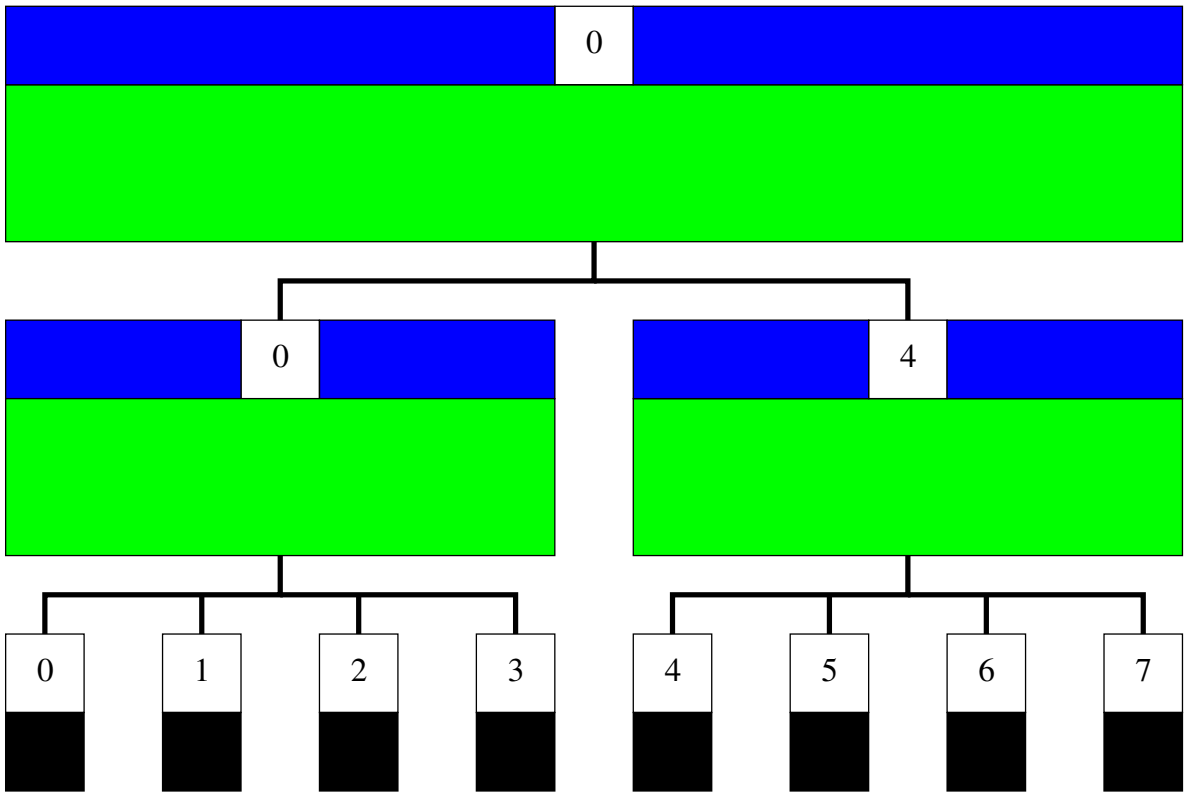
For any non-trivial Multi-BSP algorithm,

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

Question: is **storing** the broadcast value $2p - 1$ times mandatory?

Multi-BSP: broadcasting

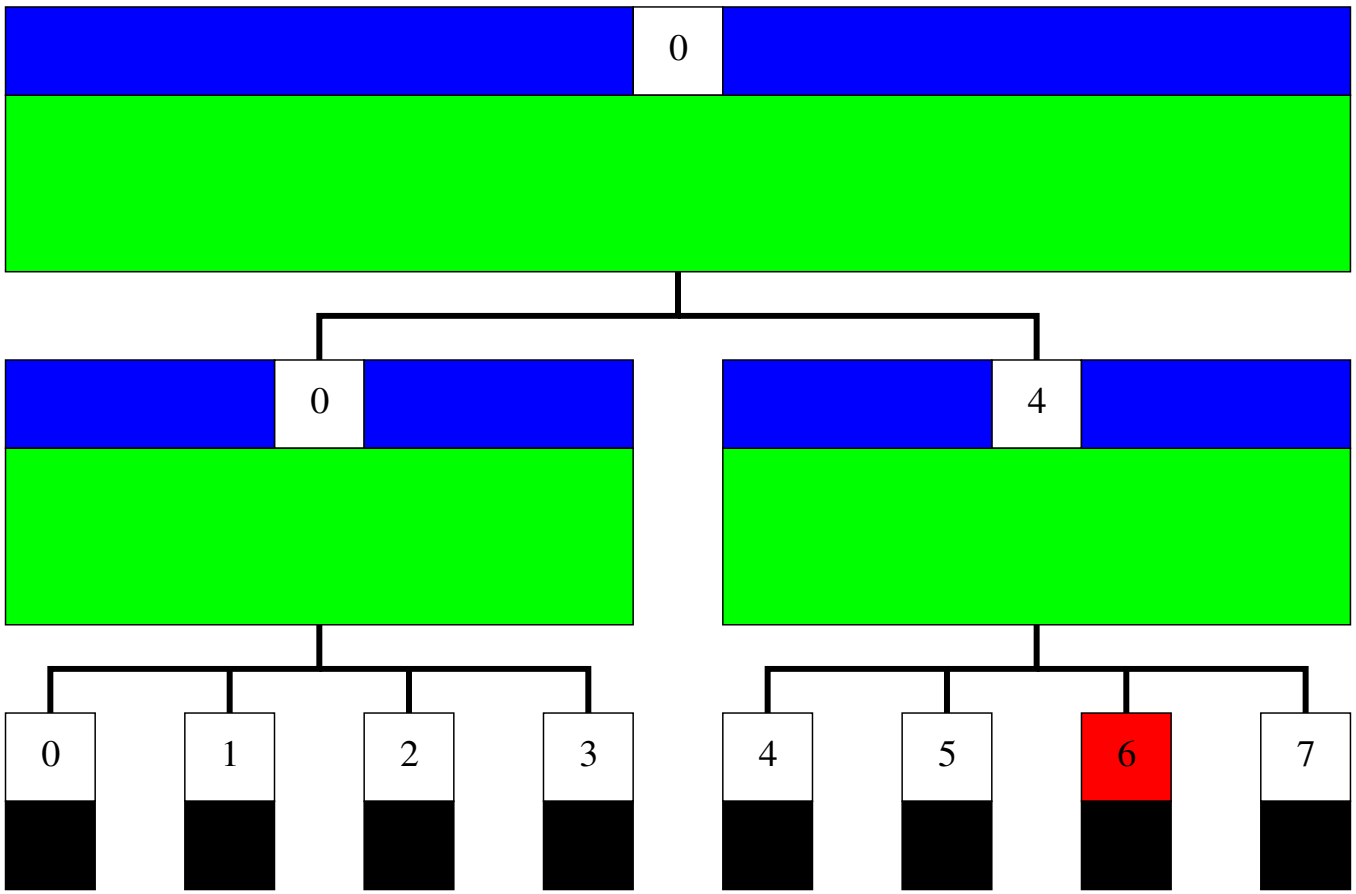
Memory embedding (shared address space):



Memory requirements are now bounded from below as $M_p \geq M_{\text{seq}}/p + p$.

Multi-BSP: broadcasting

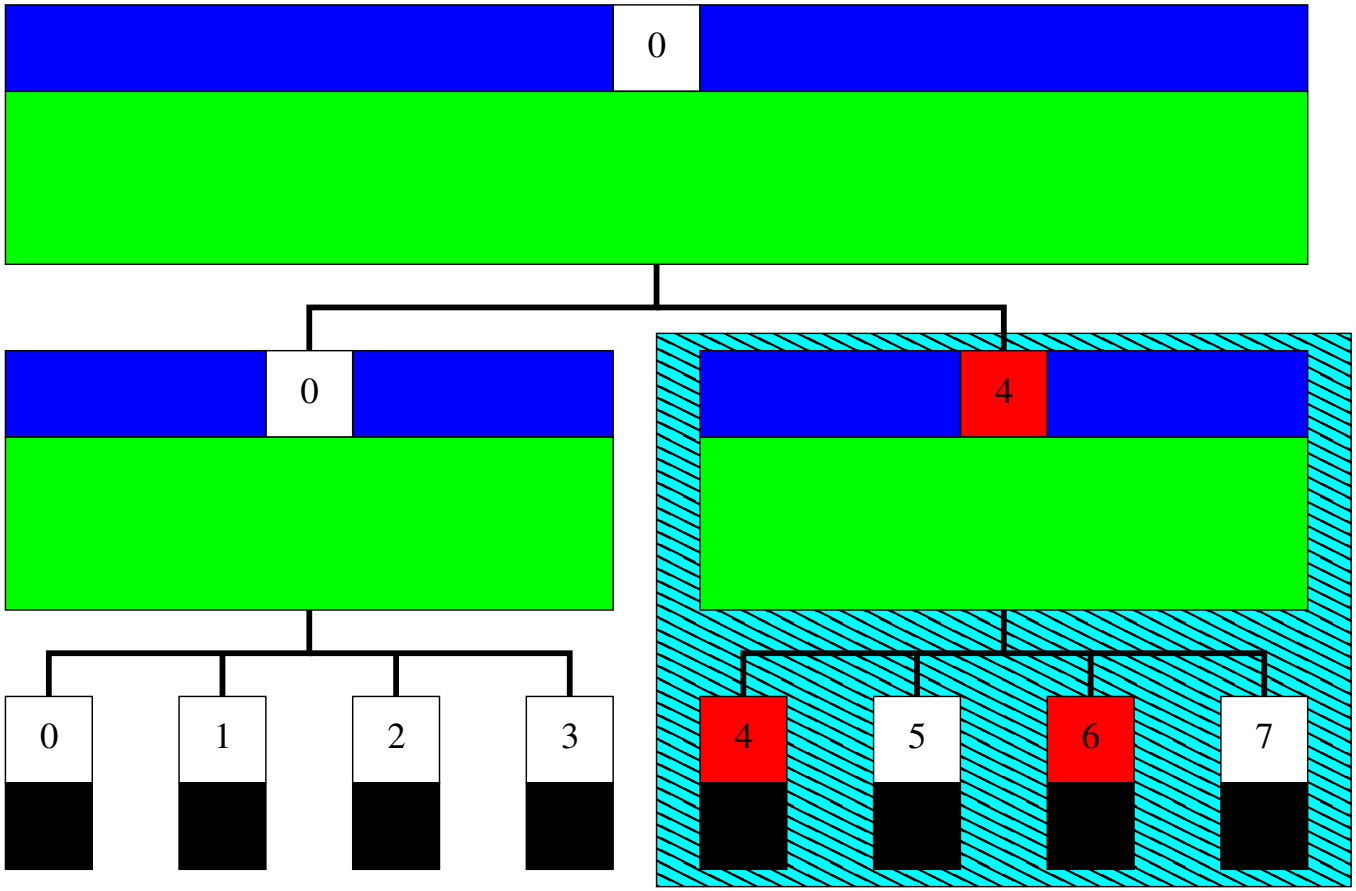
Optimal algorithm with embedding (shared address space):



Start SPMD section, entry at leaf level, leaf 6 is source.

Multi-BSP: broadcasting

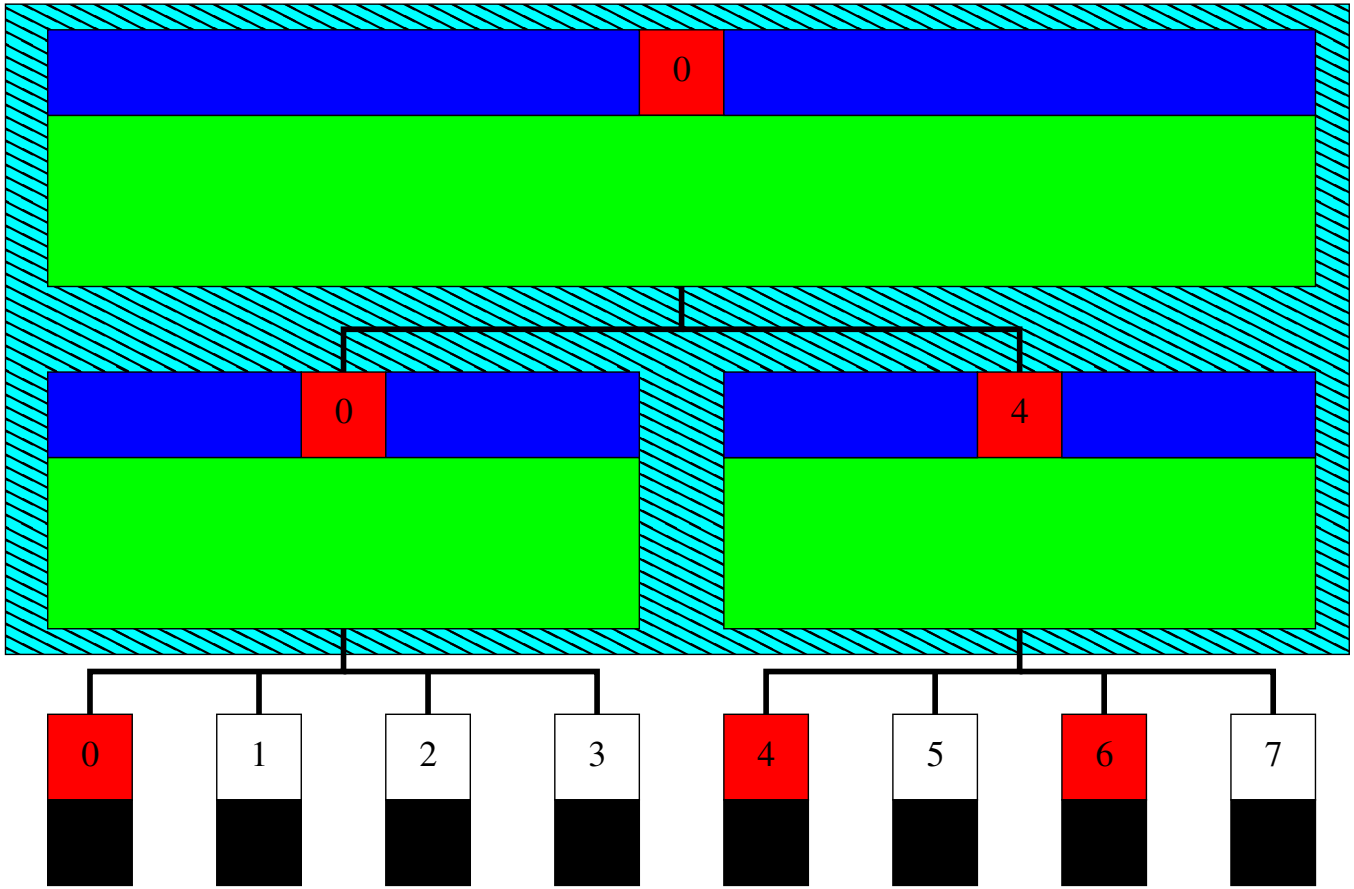
Optimal algorithm with embedding (shared address space):



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

Multi-BSP: broadcasting

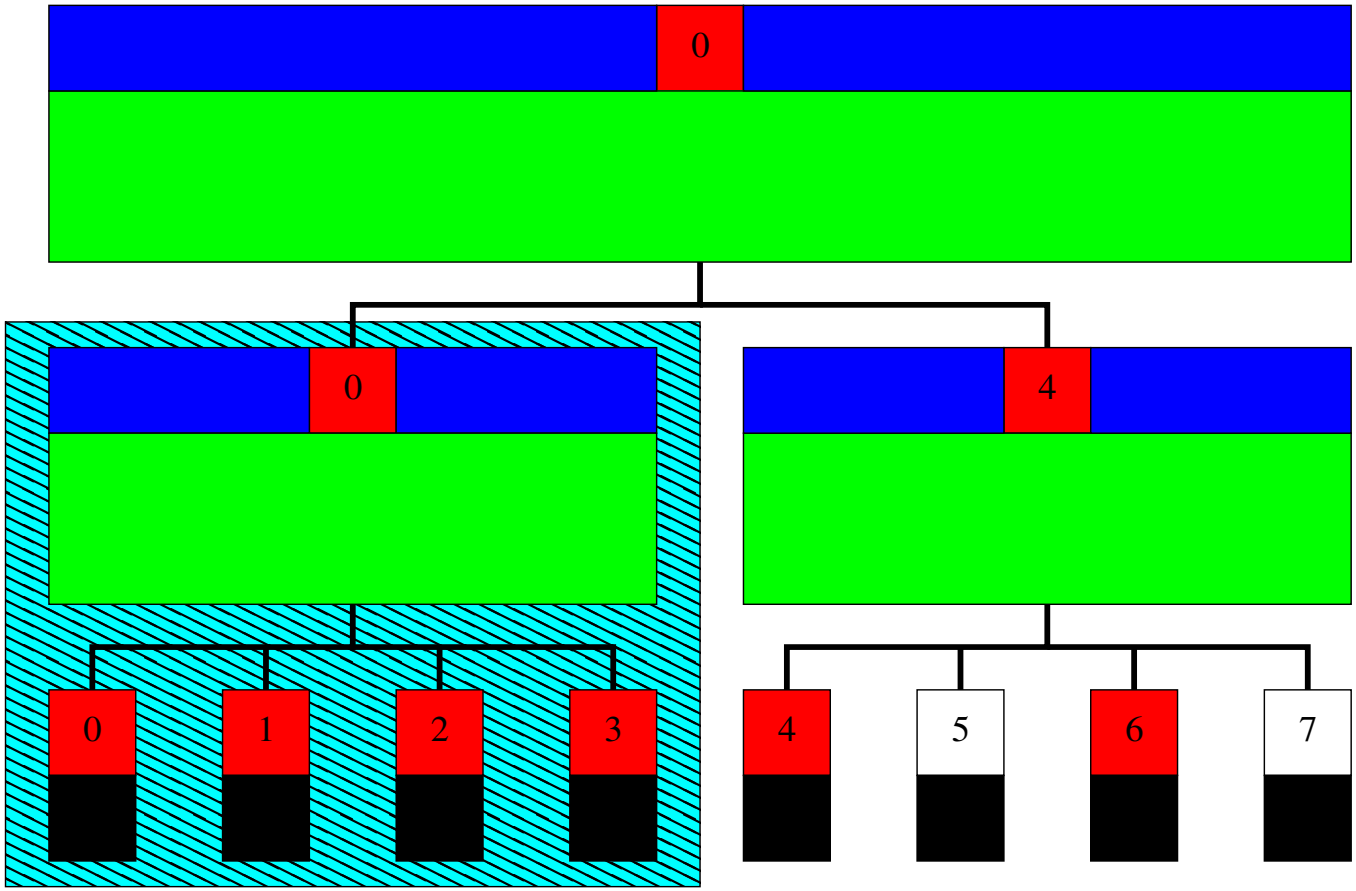
Optimal algorithm with embedding (shared address space):



On this upper level, send *val* to PID 0, **broadcast**, and **move down**.

Multi-BSP: broadcasting

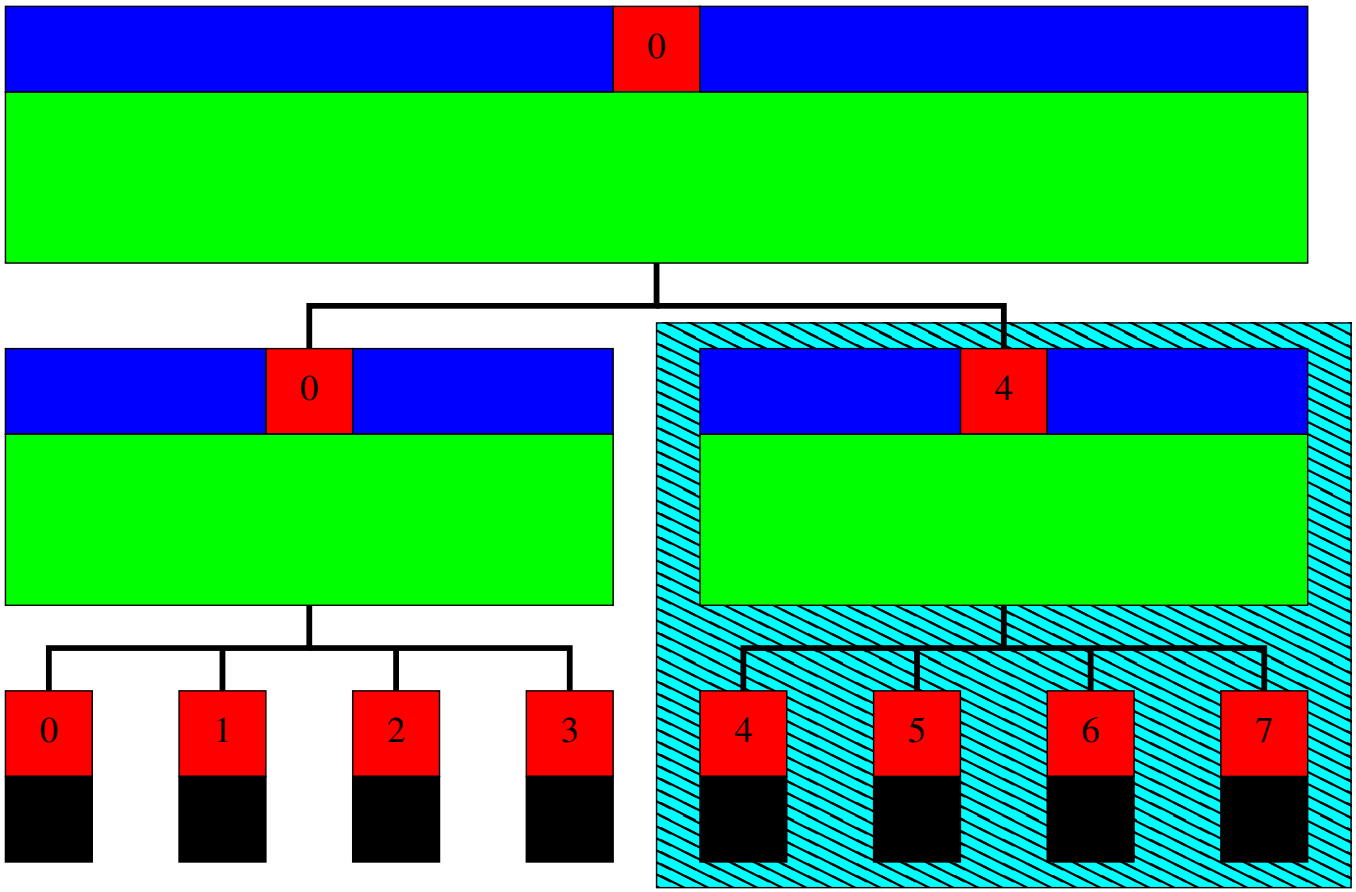
Optimal algorithm with embedding (shared address space):



On this level, only PID 0 has *val*; **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.

do
    if $val \neq \emptyset$
        bsp_put val into PID 0
    bsp_sync
while *bsp_up*

do
    if $bsp\_pid \neq 0$
        bsp_get val from PID 0
    bsp_sync
while *bsp_down*

return *val*

New Multi-BSP primitives

To control the flow up and down the Multi-BSP tree:

- `bsp_up()`, and
- `bsp_down()`.

These functions return *true* when successfully having moved up or down the Multi-BSP tree, and *false* otherwise.

Relevant run-time 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.

We are closing in on a final API.

Multi-BSP summary

- Non-trivial Multi-BSP algorithms require that

$$T_p = \Omega(T_{\text{seq}}/p + \sum_{k=0}^{L-1}(g_k + l_k));$$

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

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

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’ BSPlib.

This allows reuse of existing BSP codes.

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 *uses less than p processors*, thus resulting in 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

$$T_p = \Omega(T_{\text{seq}}/p + \sum_{k=0}^{L-1}(g_k + l_k));$$

- 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.