Alternative parallel programming paradigms and frameworks

Albert-Jan Yzelman

10th of December, 2014

Flynn's Taxonomy

	Single Data	Multiple Data
Single Instruction	SISD	SIMD
Multiple Instruction	MISD	MIMD

Examples:

- classical von Neumann computing; program and data streamed to processor.
- vectorisation (MMX/AVX/...), vector computers.
- shared-memory programming, i.e.,
  - ccNUMA architectures,
  - symmetric multiprocessors,
  - non-cc NUMA (scratchpad memory, Cell Broadband Engine, ...)
- distributed-memory programming (classical, grids, clouds, ...).
- hybrid parallel programming.

Programming models

A class on parallel computing focuses on MIMD.

- Two flavours of MIMD:
  - Single Program, Multiple Data (SPMD), and
  - Multiple Program, Multiple Data (MPMD).
- Both SPMD and MPMD programs can be in **lockstep**.

Examples:

- GPUs often use **stream-based** programming. This is SPMD and employs groupwise lockstepping.
- Past and future architectures do allow and even **promote MPMD programming**: e.g., the Cell BE and the Epiphany archtictures.
- BSP is an SPMD paradigm.

Common HW/OS capabilities

The more common hardware and operating systems support for parallel computation include:

- thread and process execution and control:
  - spawn,
  - abort,
  - wait for,
  - exit.
- mutexes (locks),
- synchronisation barriers,
- transactional memory
  - readily available: atomics,
  - more complex technologies: speculative threading.

In-software equivalents of the above are possible (for instance, the use of spin-locks instead of barriers).

Common design patterns

Often-used techniques for writing parallel software:

- task-based: identify independently executable work and group them into tasks, then schedule these dynamically. This may include modeling inter-task dependencies in terms of input- and output-data. In the extreme: workflows.
- divide-and-conquer: concurrently conquer a divided problem.
- geometric decomposition: often used in spatial computational science simulations (e.g., mesh-based computations).
- pipelining: like task-based parallelism, but with serial-dependencies; this pattern sometimes appears in a subsection of a parallel program.
- master/worker: one process assignes work others.
- fork/join: after a fork a new process appears, starting at the same part of the program.

Task-based parallelisation

A task has three states: (1) wait for data, (2) ready, and (3) done. If $\mathcal{T}_i$ is the set of tasks in state i, then a simple scheduler runs a loop similar to:

- While $\mathcal{T}_2$ is empty, wait.
- Remove a task t from $\mathcal{T}_2$.
- Execute t.
- For all tasks $s \in \mathcal{T}_1$, if s waits on t,
- let s know t is done, and
- if s has no further waits, remove it from $\mathcal{T}_1$ and add it to $\mathcal{T}_2$.
- Insert t into $\mathcal{T}_3$.

This requires thread-safe datastructures (concurrent removals, insertions). **The orange statement is critical to performance**.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.

Remark: the sequential running time $T_{\text{seq}} = \sum_{v \in V} w(v)$.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.

Remark: let $P \subset V$ be a path from r to a **leaf vertex** x s.t.

$$\forall v \in V, \quad \nexists (x, v) \in E,$$

then $w(P) = \sum_{v \in P} w(v)$ is the **path weight**. The weighted **critical path** P_c has for all other possible paths P that $w(P_c) \geq w(P)$.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.
- find the weight of the weighted critical path (and try to minimise this for scalability).

Remark: if $p \to \infty$, the minimum amount of computation remains $w(P_c)$, **always**.

Task-based analysis

The analysis of a fine-grained (task-based) algortihm differs from what we have seen previously. Steps to perform the analysis:

- view the collection of tasks the program creates as a rooted directed graph $G = (V, E)$; V contain all tasks, $r \in V$ is the initial process of the algorithm, and an edge $e = (v, w) \in E$ indicates w depends on v.
- assign vertex weights $w(v)$ to vertices $v \in V$ such that $w(v)$ equals the amount of work of the task v.
- find the weight of the weighted critical path (and try to minimise this for scalability).

Remark: if $p \to \infty$, the minimum amount of computation remains $w(P_c)$, **always**. The parallel compute time T_p is bounded by

$$T_p = \mathcal{O}(w(P_c)/r + T_{\text{seq}}/p),$$

with r the computation speed of a single worker.

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** and **span**.

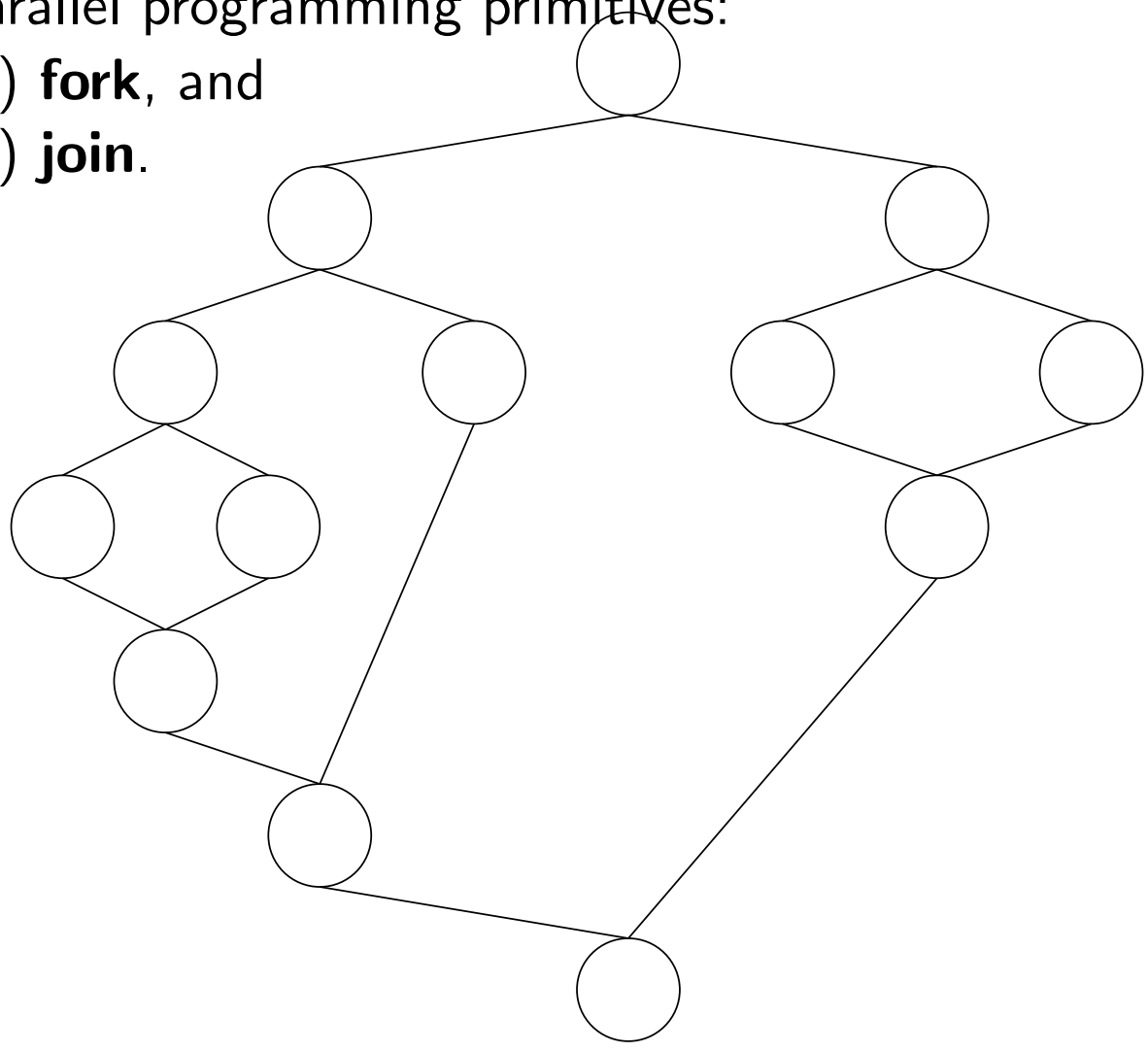
Algorithms can be implemented as graphs either explicitly or **implicitly**:

- Intel: Threading Building Blocks (TBB),
- **OpenMP** (remember the shared-memory SpMV example),
- Intel / MIT / Cilk Arts: **Cilk**,
- Google: **Pregel**,
- ...

Cilk

Only two parallel programming primitives:

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



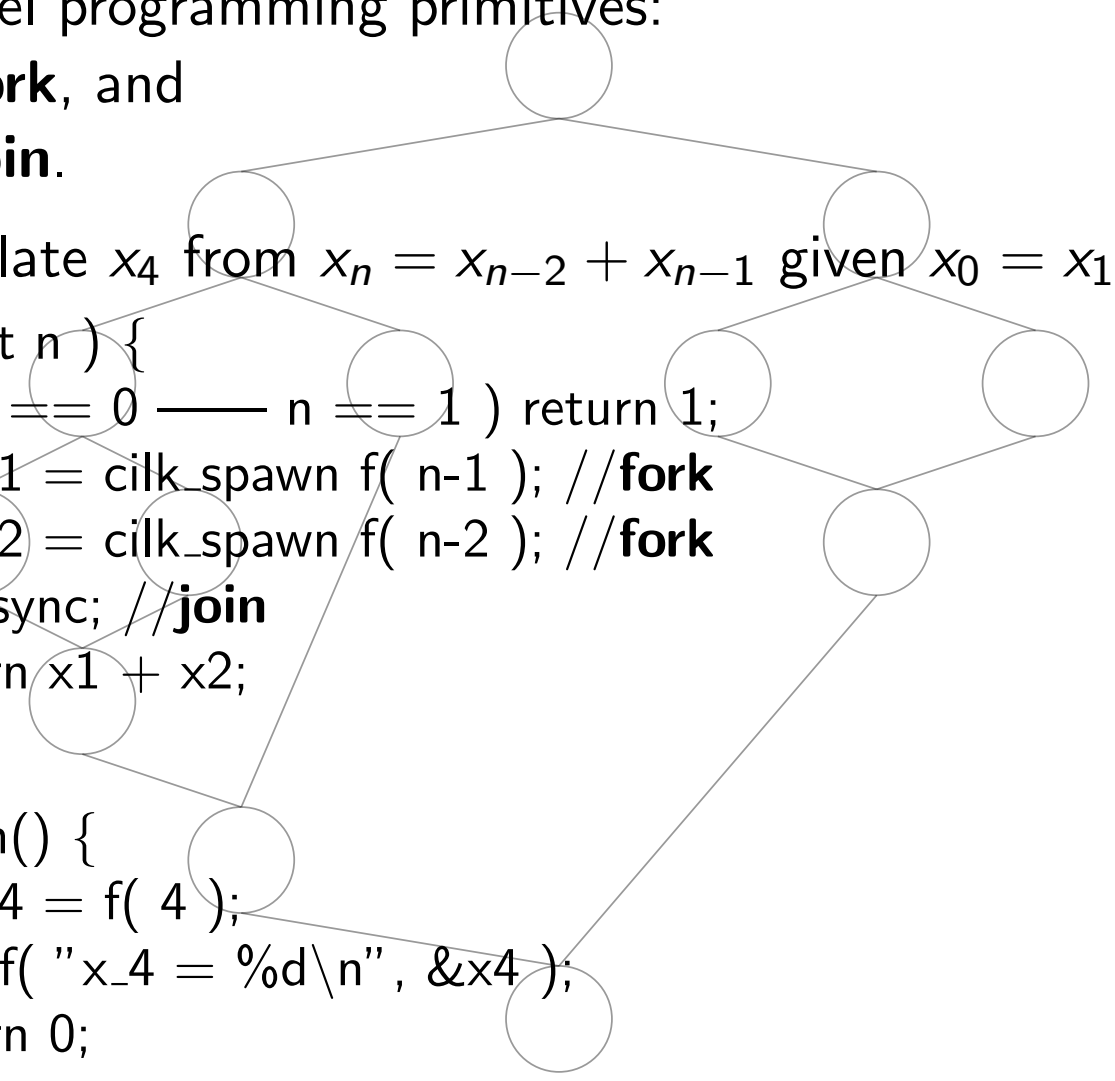
Cilk

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 $x_0 = x_1 = 1$:

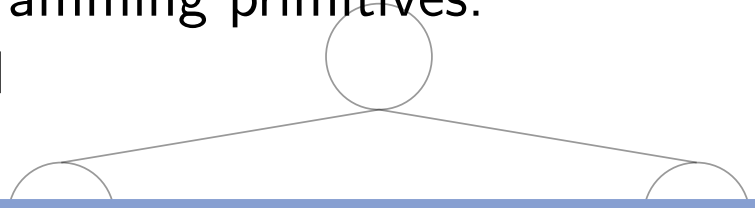
```
int f( int n ) {
    if( n == 0 —— n == 1 ) return 1;
    int x1 = cilk_spawn f( n-1 ); //fork
    int x2 = cilk_spawn f( n-2 ); //fork
    cilk_sync; //join
    return x1 + x2;
}

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**.



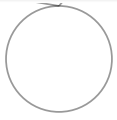
Definition (Overhead)

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:

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

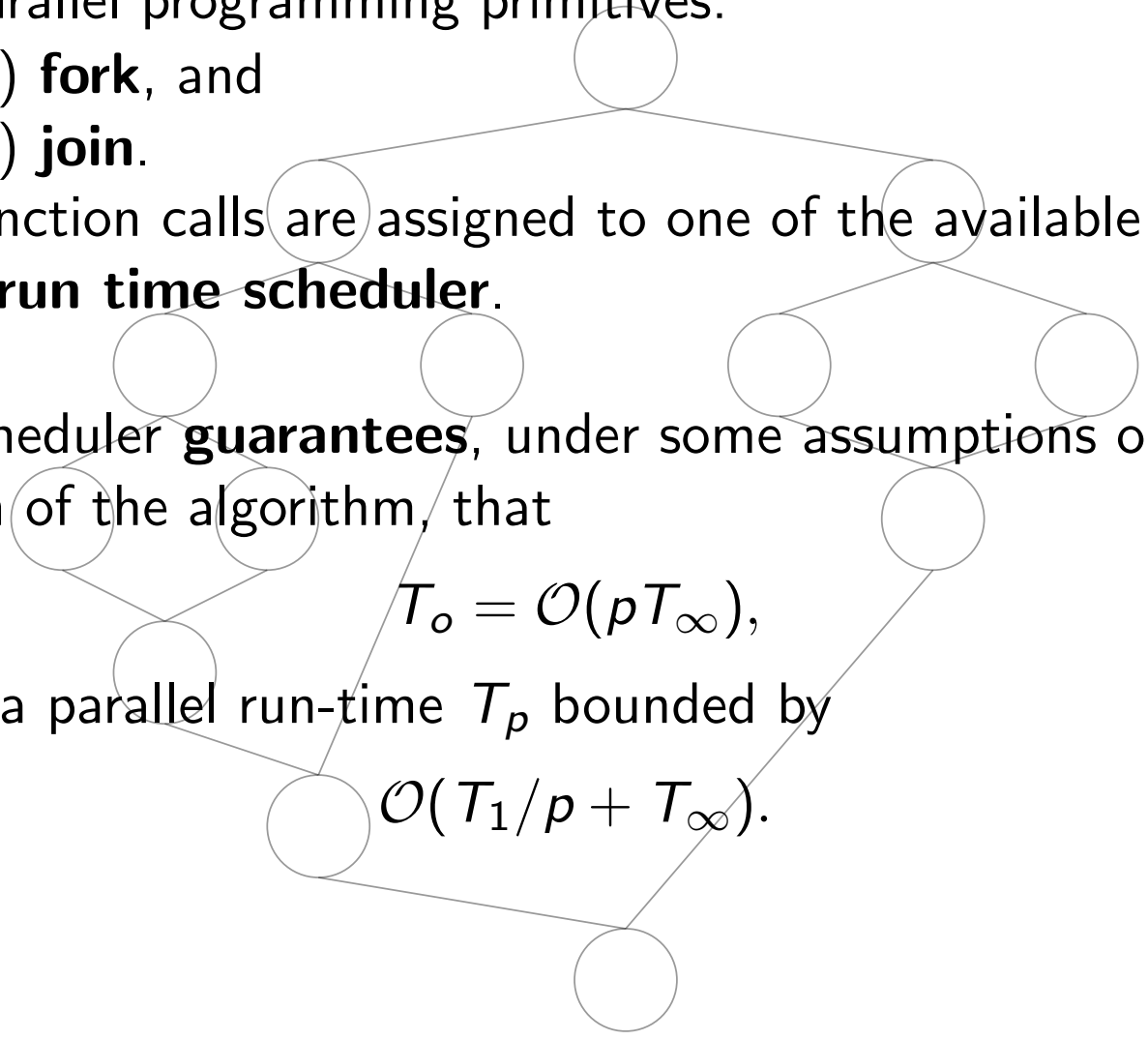


Cilk

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

$$\mathcal{O}(T_1/p + T_\infty).$$

Coarse-grained parallisation: communicating processes

Popular ways for inter-process communication:

- message passing (MPI, BSP),
- direct remote memory access (BSP, MPI),
- collectives (MPI),
- global arrays / global address space (GA/PGAS/UPC).

These approaches can be

- Blocking or non-blocking,
- One-sided or two-sided (and up to p-sided).

Classically,

- MPI is a two-sided blocking message passing (send/receive pairs) paradigm, while
- BSP is a one-sided non-blocking direct remote memory access paradigm.

Naturally, both paradigms have evolved beyond this strict classification.

History of MPI

- 1994: Message Passing Interface (MPI) became available as a standard interface for parallel programming in C and Fortran 77.
- Designed by a committee called the MPI Forum consisting of computer vendors, users, computer scientists.
- Based on sending and receiving messages by a pair of processors. One processor sends; the other receives. Both are active in the communication.
- Underlying model: communicating sequential processes (CSP) proposed by Hoare in 1978.
- MPI itself is not a model. BSP is a model.
- MPI is an interface for a communication library, like BSPlib.

Recent history of MPI

- 1997: MPI-2 standard defined. Added functionality:
  - one-sided communications (put, get, sum)
  - dynamic process management
  - parallel input/output
  - languages C++ and Fortran 90
- 2003: first full implementations of MPI-2 arrive, namely MPICH (Argonne National Labs) and LAM/MPI (Indiana University).
- 2004–: Open MPI. Open-source project, merges 3 MPI implementations: LAM/MPI, FT-MPI (University of Tennessee), LA-MPI (Los Alamos National Laboratory).
- Many users still use MPI-1, particularly its latest version MPI-1.3.
- 2012 MPI-3. Major update. More one-sided communications, nonblocking collective communications.

Why use MPI?

- It is available on almost every parallel computer, often in an optimised version provided by the vendor. Thus MPI is the most portable communication library.
- Many libraries are available written in MPI, such as the numerical linear algebra library ScaLAPACK.
- You can program in many different ways using MPI, since it is highly flexible.

Why not?

- It is huge: the full standard has about 300 primitives. The user has to make many choices.
- It is not so easy to learn. Usually one starts with a small subset of MPI. Full knowledge of the standard is hard to attain.
- The one-sided communications of MPI-2 and MPI-3 are rather complicated. If you like one-sided communications you may want to consider BSPlib as an alternative.

Ping pong benchmark

- The cost of communicating a message of length n is

$$T(n) = t_{\text{startup}} + nt_{\text{word}}.$$

  Here, t_{startup} is a fixed startup cost and t_{word} is the additional cost per data word communicated.
- Communication of a message (in its blocking form) synchronises the sender and receiver. This is pairwise synchronisation, not global.
- Parameters t_{startup} and t_{word} are usually measured by sending a message from one processor to another and back: ping pong.
- The message length is varied in the ping pong benchmark.
- There is only one ping pong ball on the table.

Send and receive primitives

```
if (s==2)
  MPI_Send(x,5,MPI_DOUBLE,3,0,MPI_COMM_WORLD);
if (s==3)
  MPI_Recv(y,5,MPI_DOUBLE,2,0,MPI_COMM_WORLD,
           &status);
```

- Processor $P(2)$ sends 5 doubles to $P(3)$.
- $P(2)$ reads the data from its array `x`. After transmission, $P(3)$ writes these data into its array `y`.
- The integer '0' is a tag for distinguishing between different messages from the same source processor to the same destination processor.
- `MPI_Send` and `MPI_Recv` are of fundamental importance in MPI.

Communicator: the whole processor world

```
if (s==2)
  MPI_Send(x,5,MPI_DOUBLE,3,0,MPI_COMM_WORLD);
if (s==3)
  MPI_Recv(y,5,MPI_DOUBLE,2,0,MPI_COMM_WORLD,
           &status);
```

- A communicator is a subset of processors forming a communication environment with its own processor numbering.
- `MPI_COMM_WORLD` is the communicator consisting of all the processors.

Send/Receive considered harmful

- 1968: Edsger Dijkstra, guru of structured programming, considered the Go To statement harmful in sequential programming.
- Go To was widely used in Fortran programming in those days. It caused spaghetti code: if you pull something here, something unexpected moves there.
- No one dares to use Go To statements any more.
- Send/Receive in parallel programming has the same dangers, and even more, since several diners eat from the same plate.
- Pull here, pull there, nothing moves: deadlock.
- Deadlock may occur if $P(0)$ wants to send a message to $P(1)$, and $P(1)$ to $P(0)$, and both processors want to send before they receive.

Inner product program mpiinprod

```
int main(int argc, char **argv){

    int p, s, n;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&p);
    MPI_Comm_rank(MPI_COMM_WORLD,&s);

    if (s==0){
        printf("Please enter n:\n");
        scanf("%d",&n);
        if(n<0)
            MPI_Abort(MPI_COMM_WORLD,-1);
    }
    MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
```

Collective communication: broadcast

```
MPI_Bcast(&n, 1,    MPI_INT, 0, MPI_COMM_WORLD);

MPI_Bcast(buf,count,datatype,root,communicator);
```

- Broadcast `count` data items of a certain `datatype` from processor `root` to all others in the `communicator`, reading from location `buf` and also writing it there.
- All processors of the communicator participate.
- Extensive set of collective communications available in MPI. Using these reduces the size of program texts, and allows hardware vendors to provide optimised algorithms.

Inner product program mpiinprod (cont'd)

```
...
nl= nloc(p,s,n);       //local vector length
x = vecallocd(nl);     //allocate and initialise x
for (i=0; i<nl; i++) {
    iglob= i*p+s;
    x[i]= iglob+1;
}

MPI_Barrier(MPI_COMM_WORLD); // global sync for timing
time0=MPI_Wtime();           // wall clock time
...
```

Inner product program mpiinprod (cont'd)

```
...

//do calculate inner product
alpha= mpiip(p,s,n,x,x);

//sync for timing
MPI_Barrier(MPI_COMM_WORLD);
time1=MPI_Wtime();

...
MPI_Finalize();
exit(0);
```

Inner product function mpiip

```
double mpiip(int p,int s,int n,
             double *x,double *y){

    double inprod, alpha;
    int i;

    inprod= 0.0;
    for (i=0; i<nloc(p,s,n); i++)
        inprod += x[i]*y[i];
    MPI_Allreduce(&inprod,&alpha,1,MPI_DOUBLE,
                  MPI_SUM,MPI_COMM_WORLD);

    return alpha;
}
```

Collective communication: reduce

```
MPI_Allreduce(&inprod, &alpha,  1,  MPI_DOUBLE,
              MPI_SUM,   MPI_COMM_WORLD);

MPI_Allreduce(sendbuf, recvbuf, count, datatype,
              operation, communicator);
```

- The reduction operation by `MPI_Allreduce` sums the double-precision local inner products `inprod`, leaving the result `alpha` on all processors.
- One can also do this for an array instead of a scalar, by changing the parameter 1 to the array size `count`, or perform other operations, such as taking the maximum, by changing `MPI_SUM` to `MPI_MAX`.

Collectives

More collective operations:

- broadcast,
- scatter,
- gather,
- reduction (also reduce-scatter),
- all-gather, and
- all-to-all.

For a nice graphical overview, see
`www.mpi-forum.org/docs/mpi-1.1/mpi-11-html/node64.html`

- MPI directly integrates these patterns by providing primitives: MPI_BCAST, MPI_GATHER, MPI_SCATTER, ...
- MPI also supports collectives on vectors instead of single entities (MPI_GATHERV. MPI_SCATTERV, ...), thus enabling optimisations such as the two-phase broadcast.

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:

$$\text{map: } V \to \mathcal{P}(K \times V);$$

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

- 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:

$$\text{map: } V \to \mathcal{P}(K \times V);$$

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

- 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 (may) have the same key, 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?

Multi-BSP

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

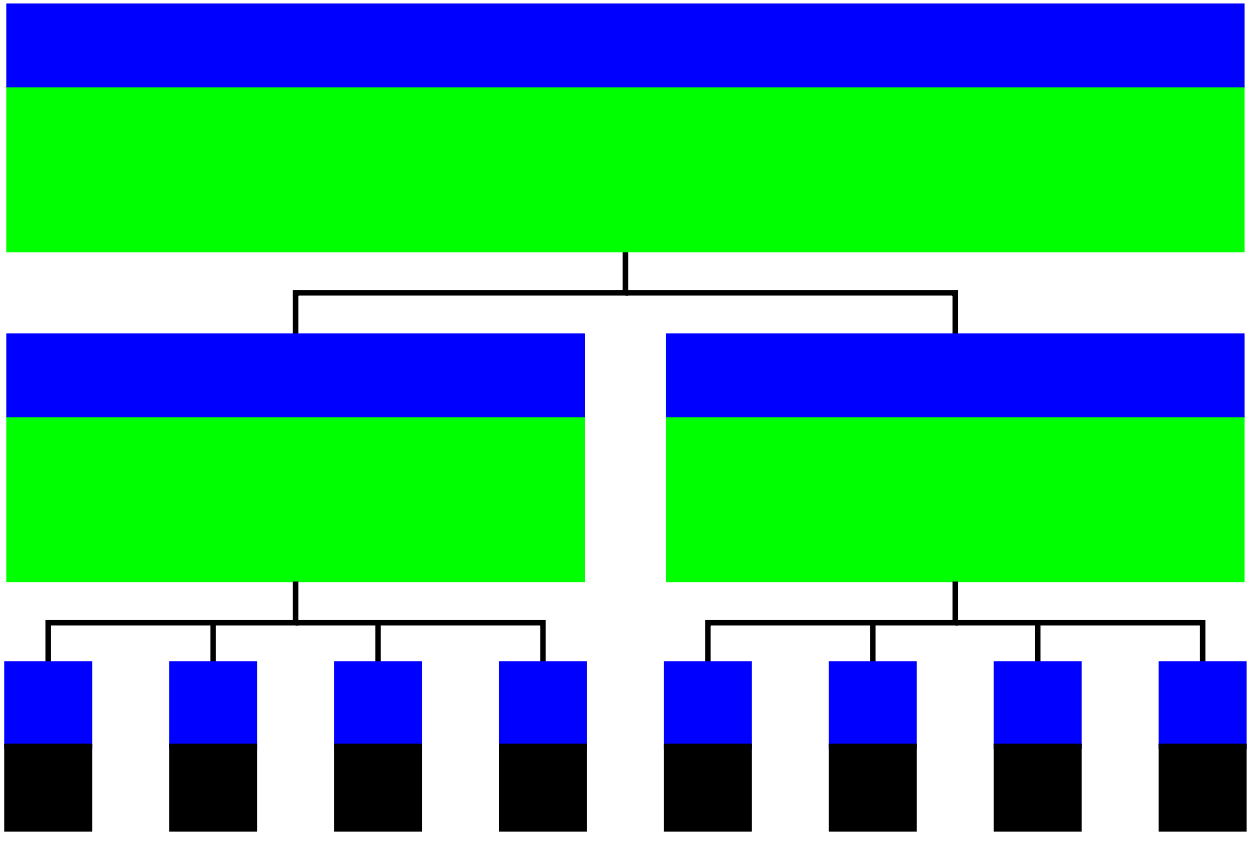
- **BSP**: a computer consists out of p CPUs/processors/cores/...
- **Multi-BSP**: a computer consists out of
  1. p *other* Multi-BSP subcomputers (recursively), **or**
  2. p units of execution (leaves).
- Each Multi-BSP computer:
  - connects its subcomputers or leaves via a network, and
  - provides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

Reference:

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

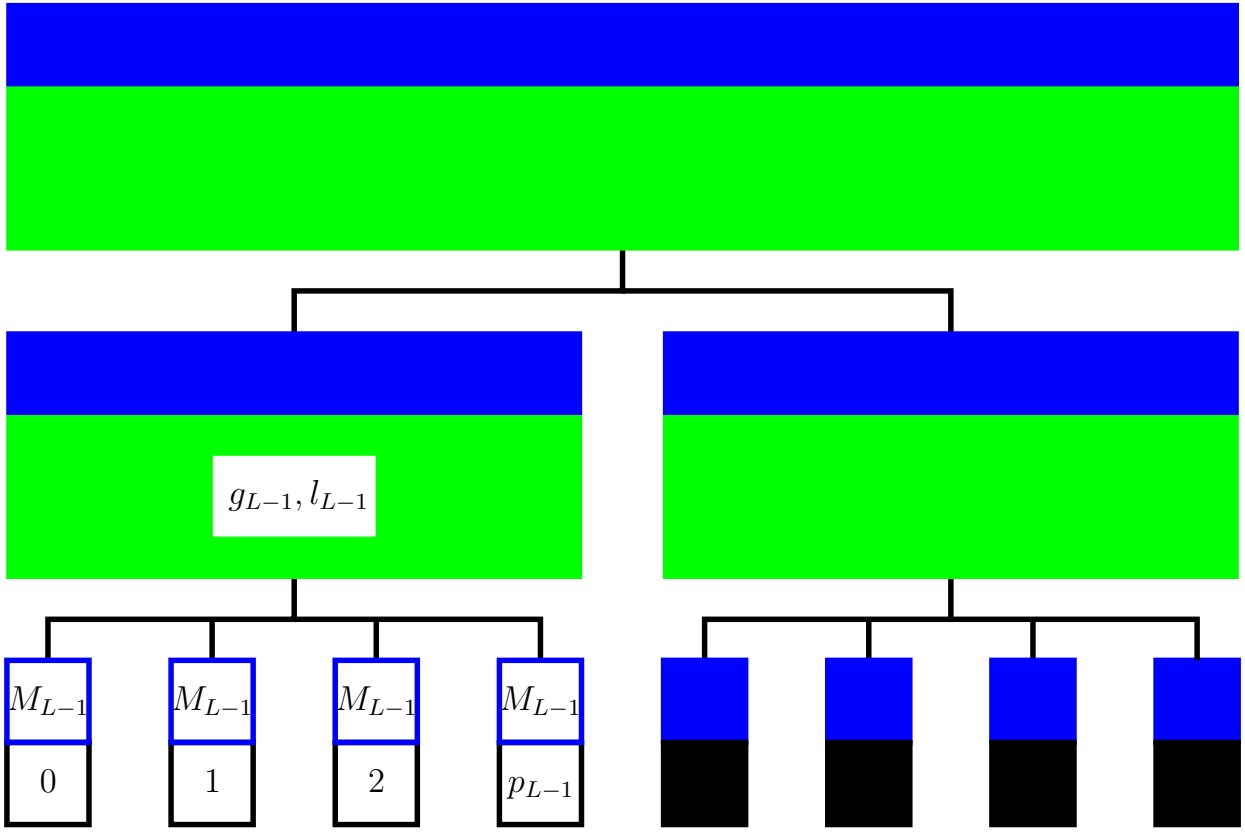
CPUs, memory, network.



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

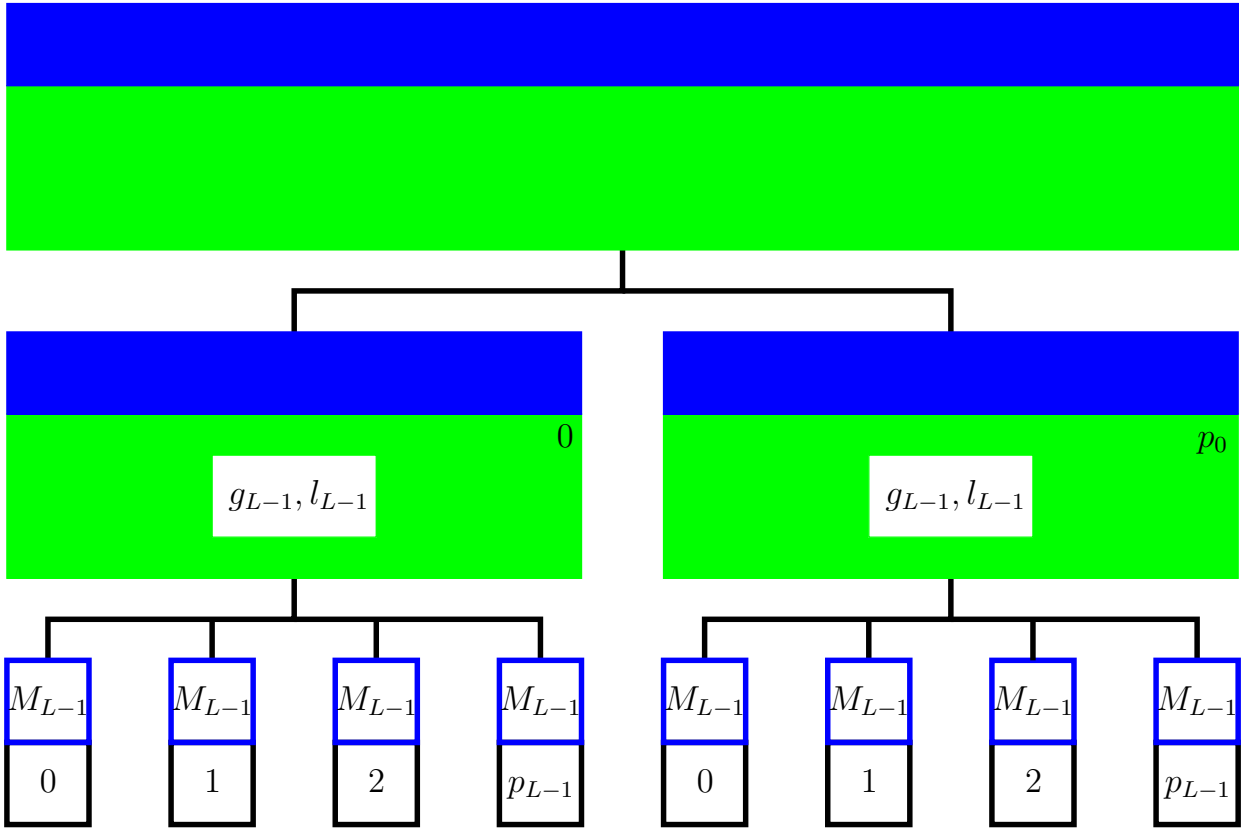
CPUs, memory, network.



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

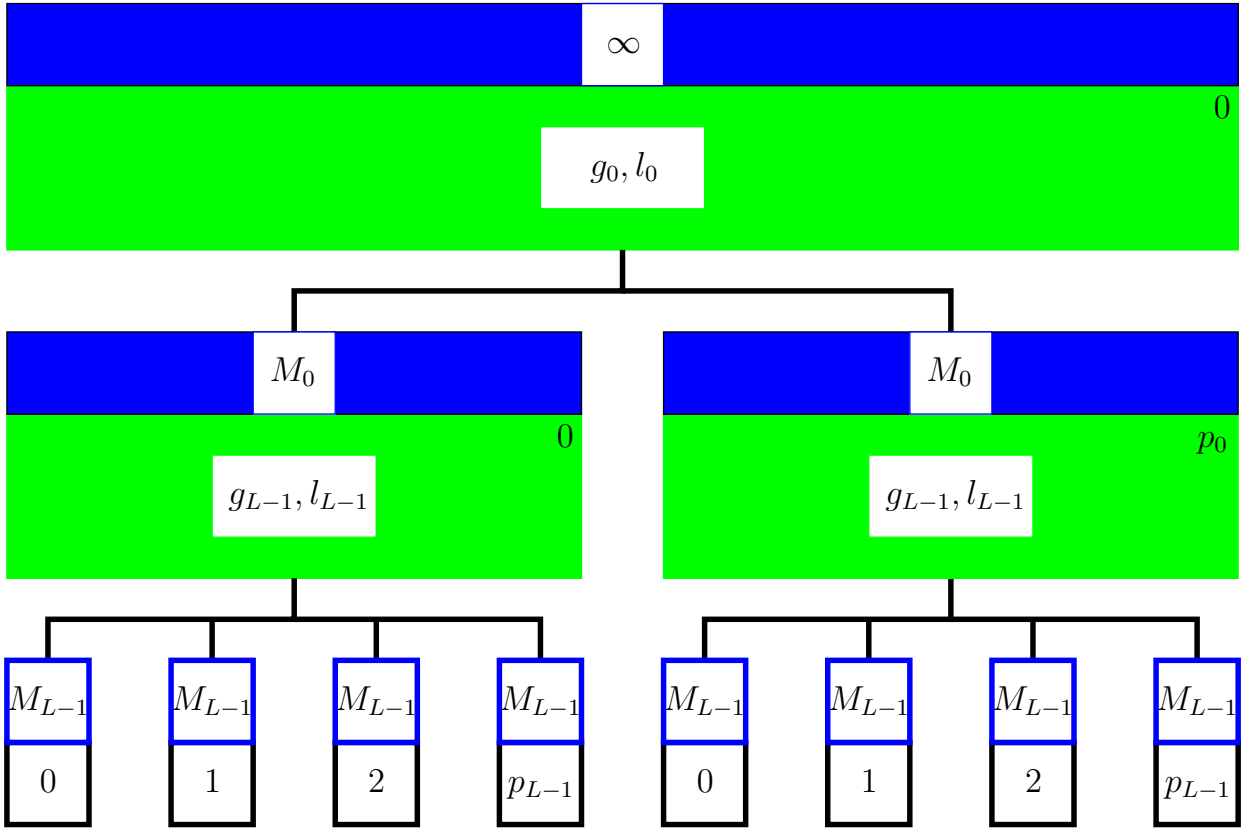
CPUs, memory, network.



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

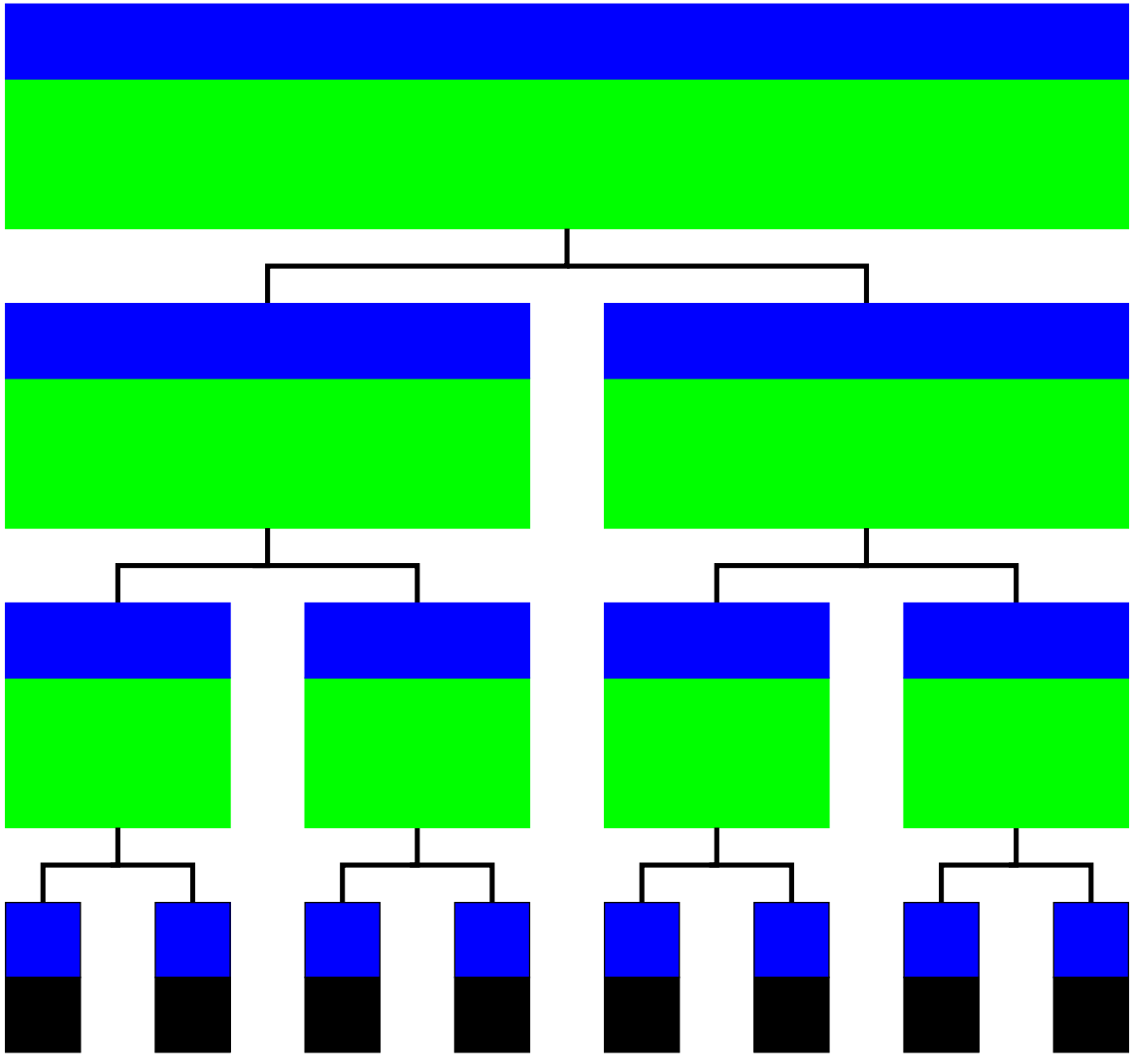
Multi-BSP computer model

CPUs, memory, network.

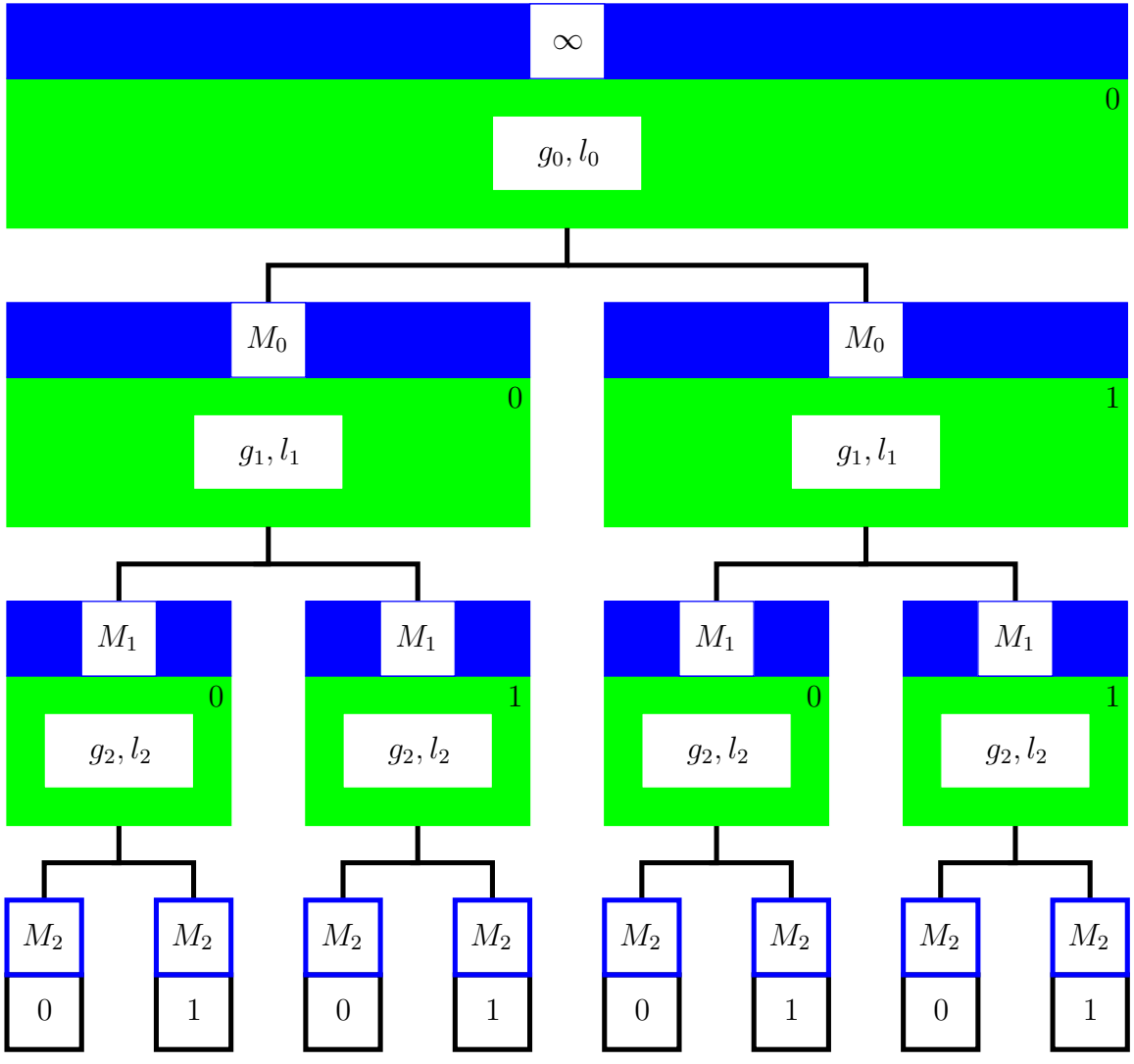


A BSP computer $(p_0, g_0, l_0, M_0) \cdots (p_{L-1}, g_{L-1}, l_{L-1}, M_{L-1})$.

Multi-BSP computer model



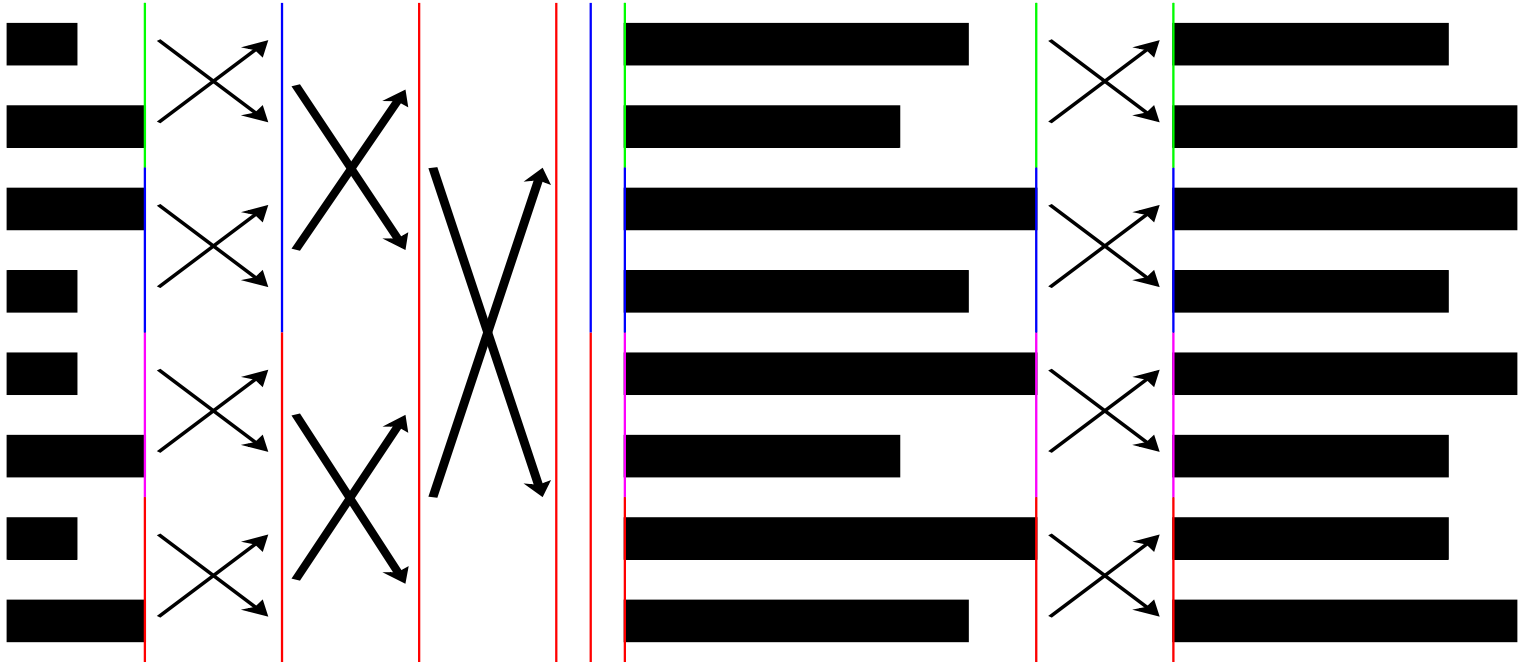
Multi-BSP computer model



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 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**:

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

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.

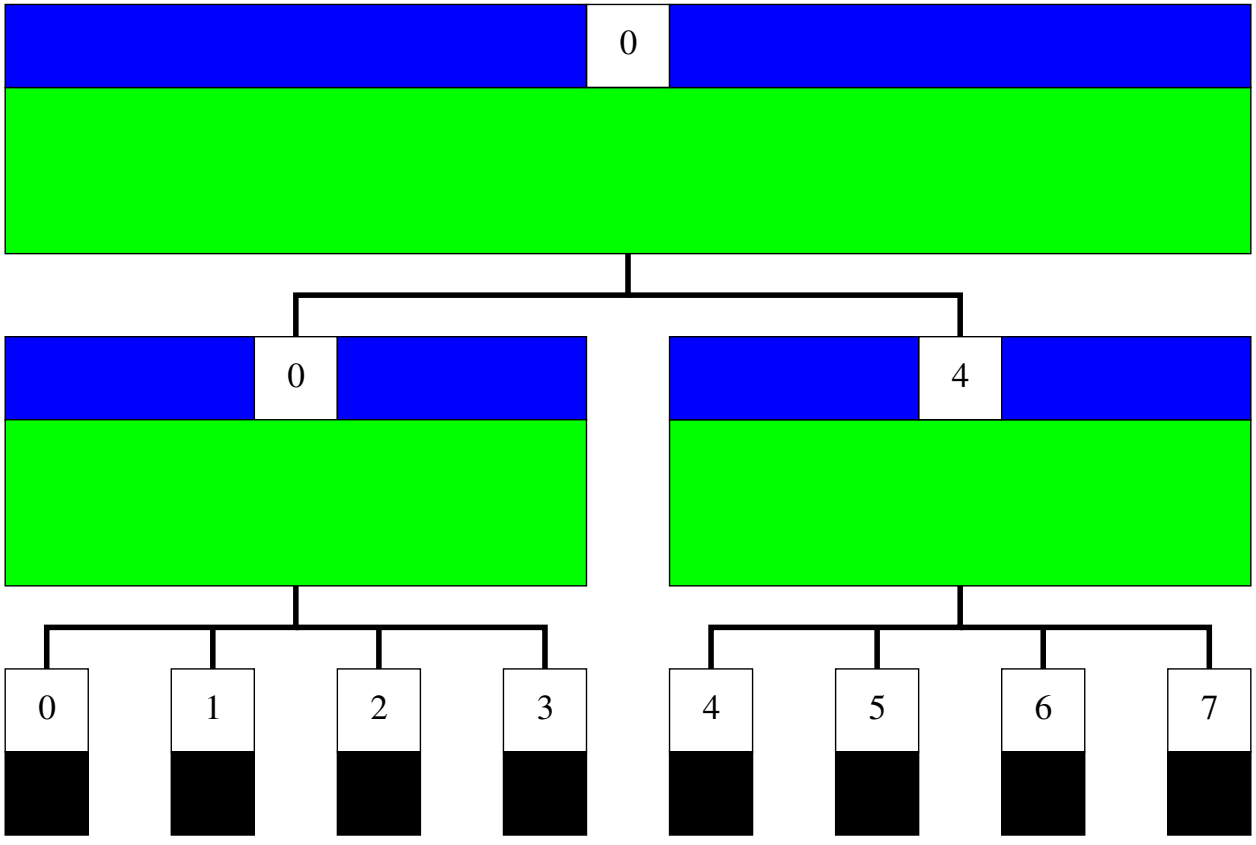
Would you like to:

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

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

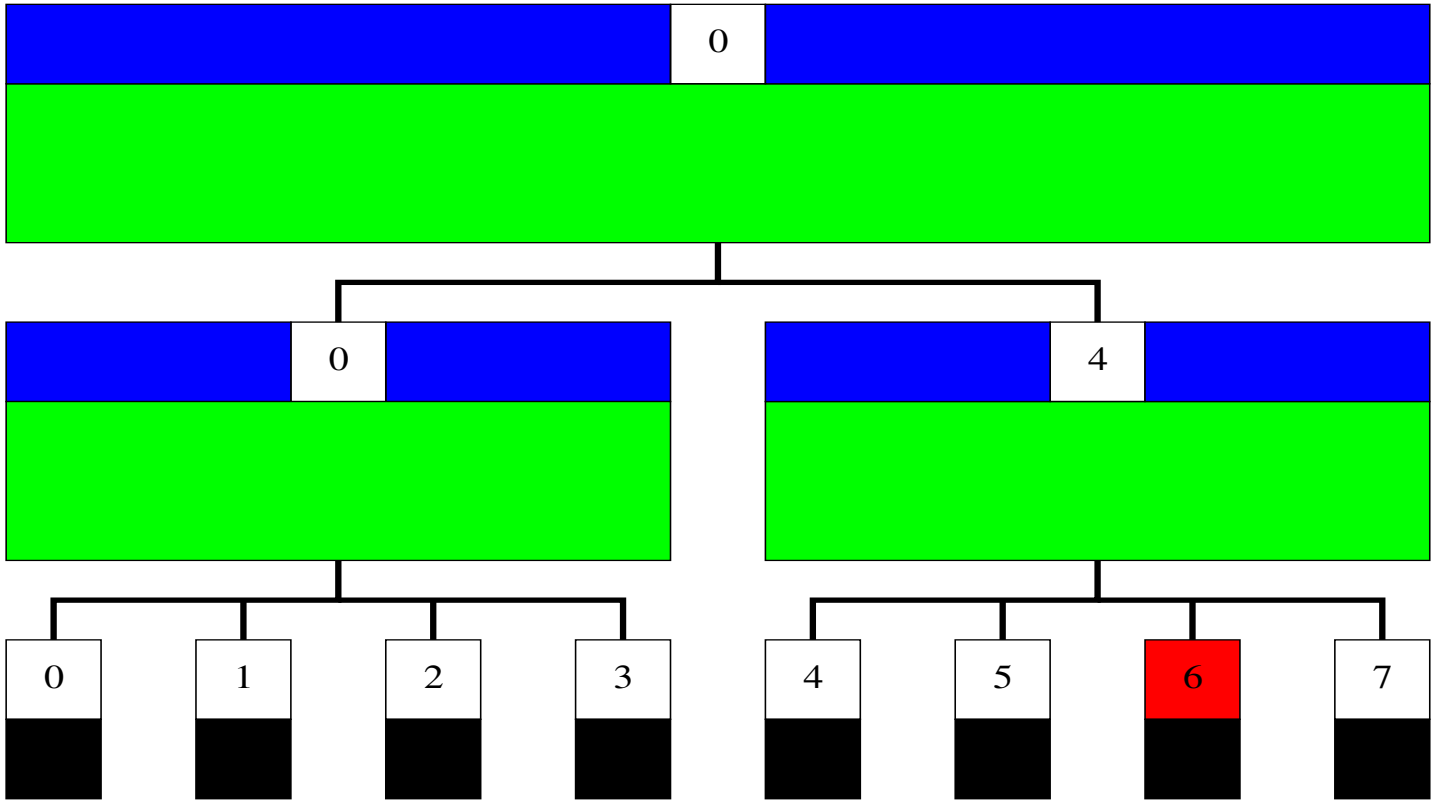
Multi-BSP: broadcasting

Memory embedding (shared address space):



Multi-BSP: broadcasting

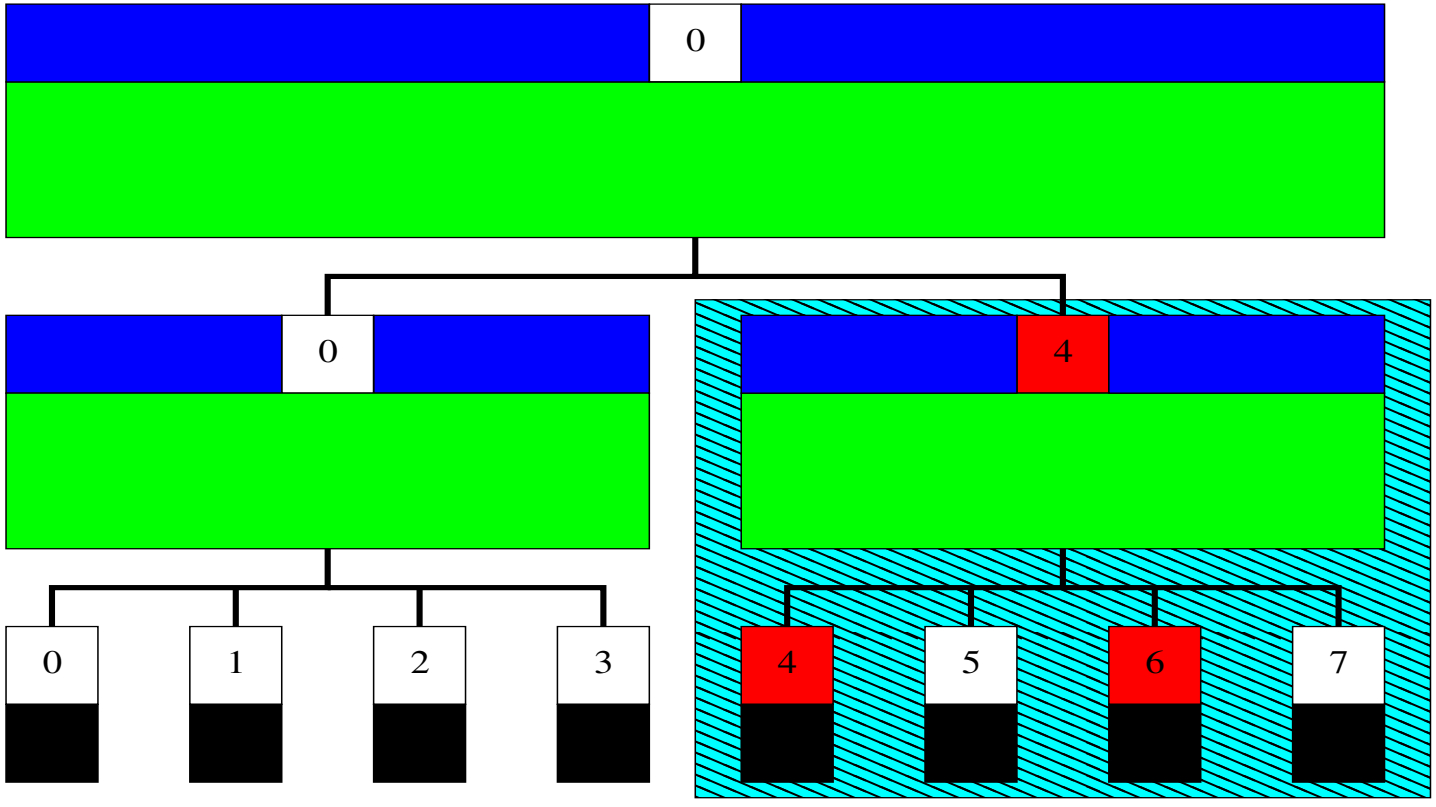
Optimal algorithm with embedding (shared address space):



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

Multi-BSP: broadcasting

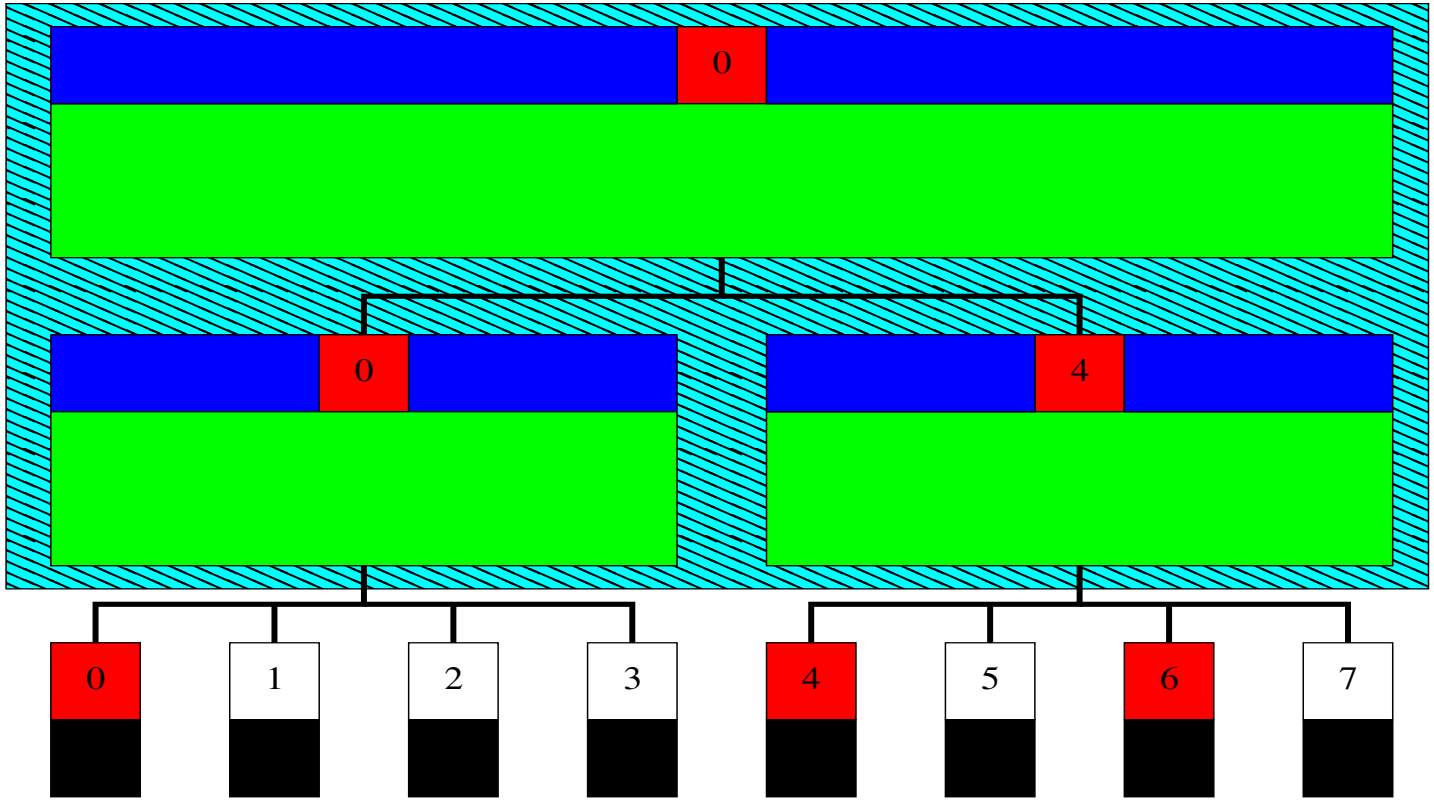
Optimal algorithm with embedding (shared address space):



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

Multi-BSP: broadcasting

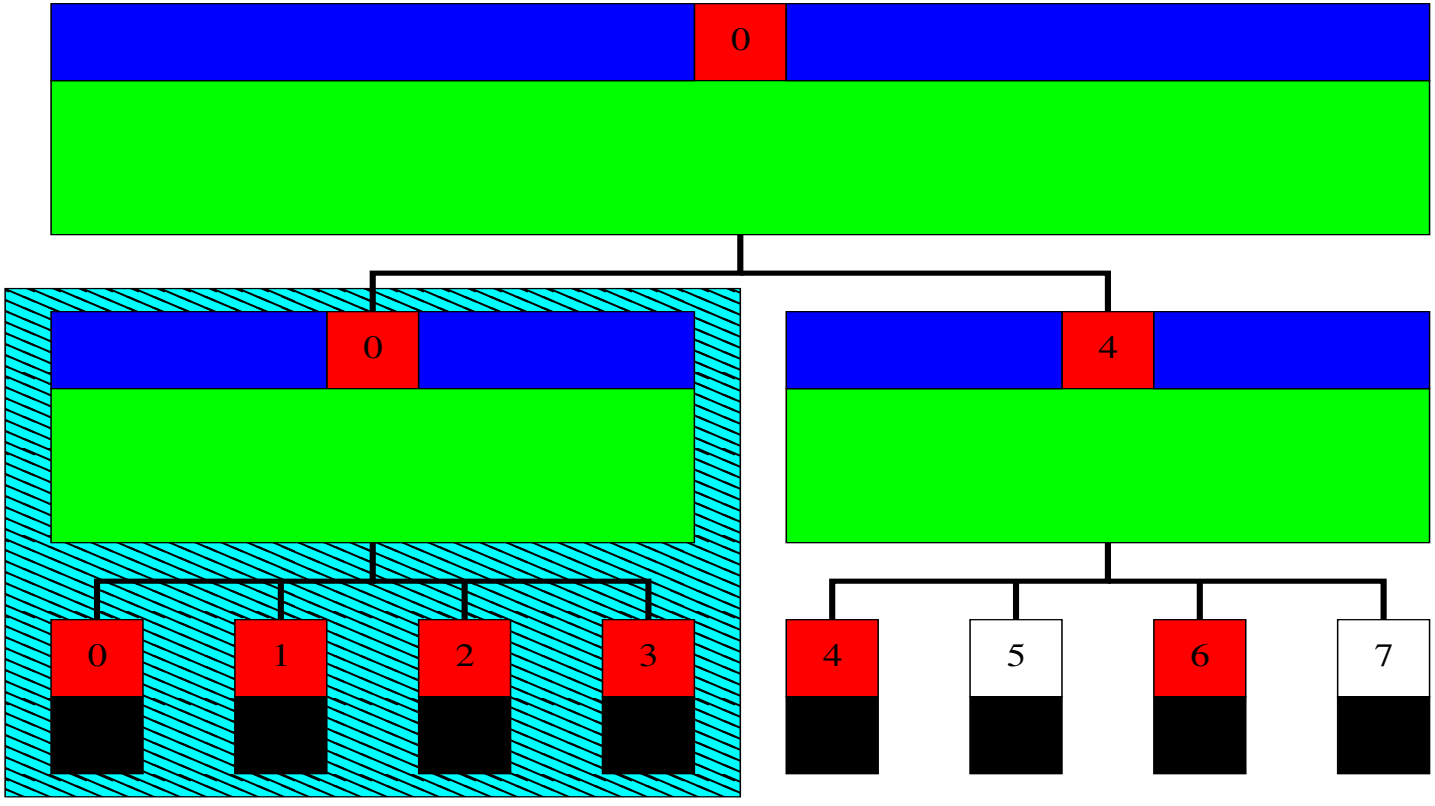
Optimal algorithm with embedding (shared address space):



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

Multi-BSP: broadcasting

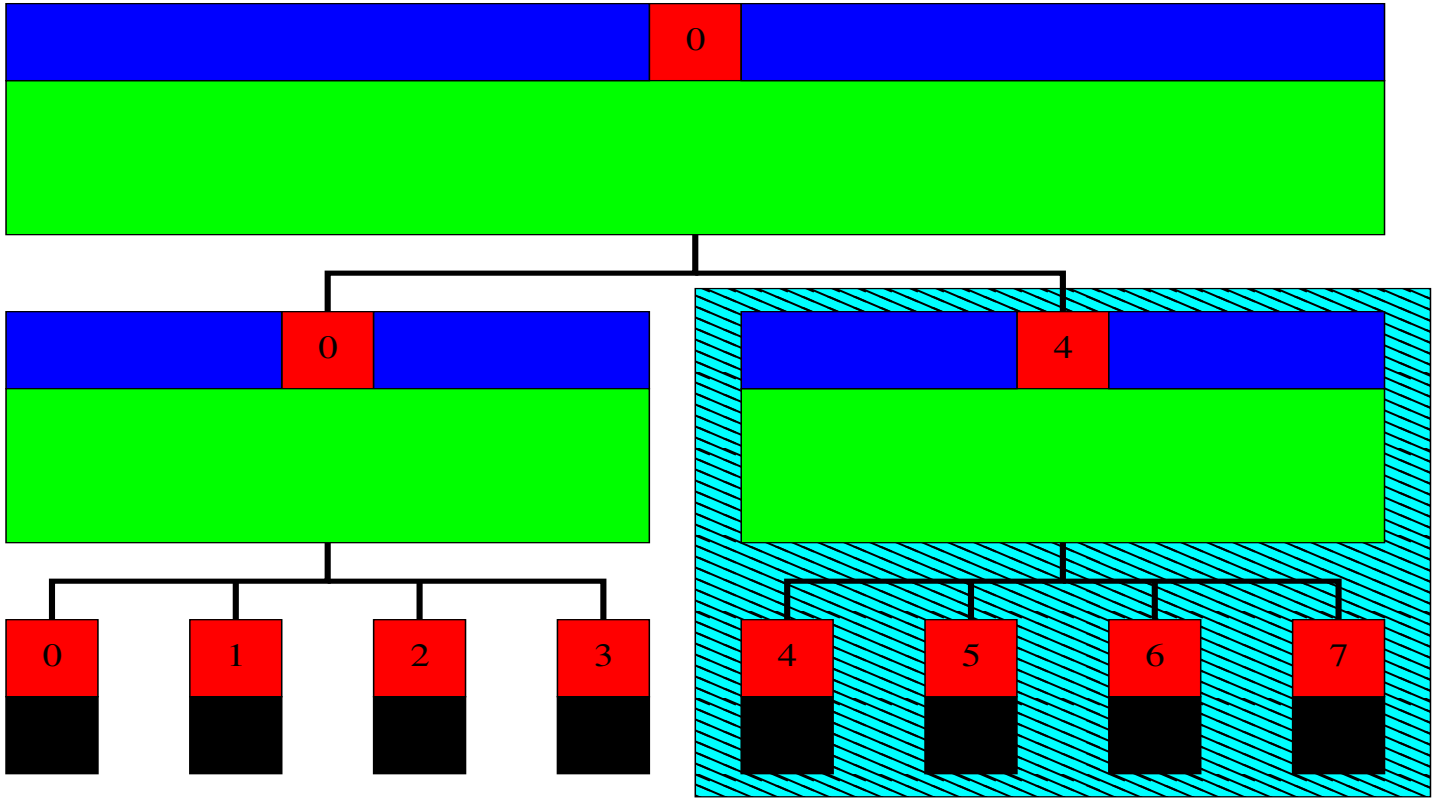
Optimal algorithm with embedding (shared address space):



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.

if $val \neq \emptyset$ **then**
  set $source = true$
else
  set $source = false$
while not on the Multi-BSP root **do**
  if *source* and $bsp\_pid() \neq 0$ **then**
    send *val* to PID 0
  move upwards in the Multi-BSP tree
while not on a leaf node **do**
  if $bsp\_pid() = 0$ **then**
    for $k = 1$ to $bsp\_nprocs()$ **do**
      send *val* to PID k
  move downwards in the Multi-BSP tree
return *val*

Summary

We have seen

- various ways on exploiting parallelism on current hardware,
- a high-level overview of parallel programming paradigms,
- common design patterns used within these paradigms,
- what the hardware and OS can do for you,
- an overview of communication paradigms,
- how to analyse fine-grained parallel applications,
- a short introduction to the Message Passing Interface, and programming using MPI, and
- what the future of BSP might look like.