An ML implementation of the MULTI-BSP model

Victor ALLOMBERT[1], Frédéric GAVA[2]

[1]LIFO - Université d'Orléans, France
[2]LACL - Université Paris Est, France

18 July 2018

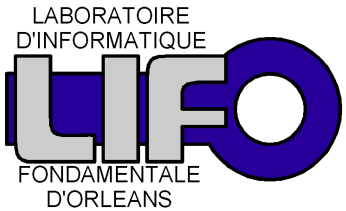


Table of Contents

Table of Contents

The world of parallel computing

Simulations:

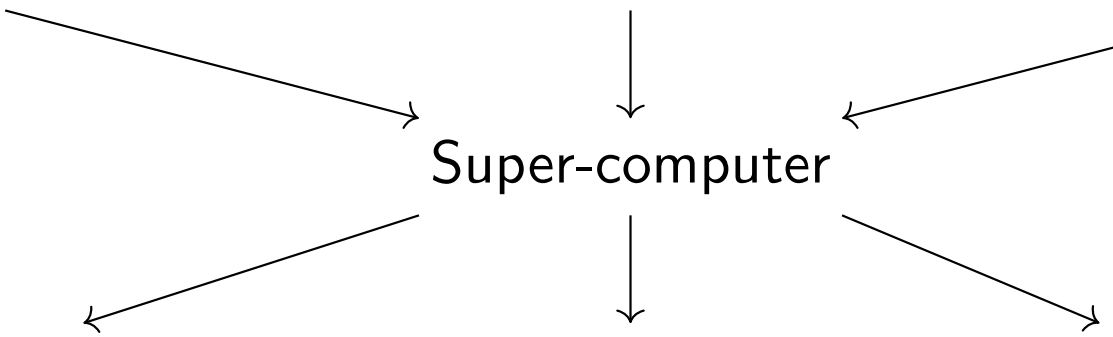
Fluid simulation
3D Visualisation

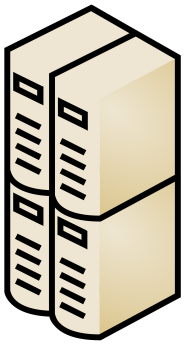
Big-Data:

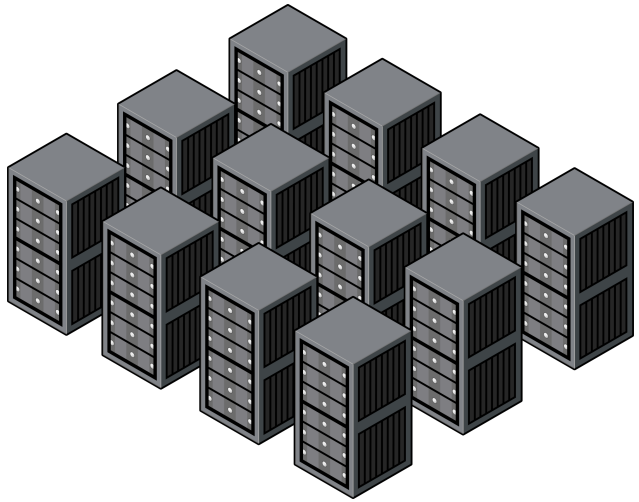
IoT
Social Networking
Data science

Symbolic computation:

Model-Checking
Formal computing







Bulk Synchronous Parallelism

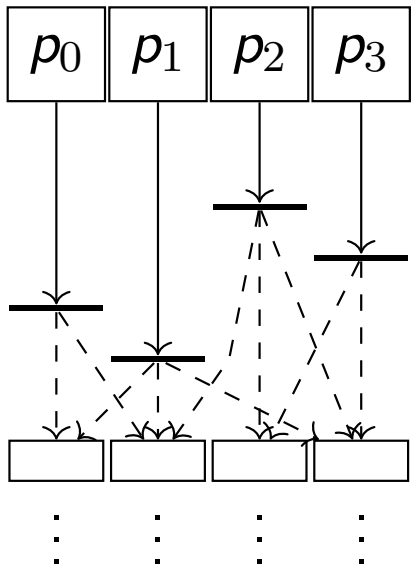
The BSP computer

Defined by:

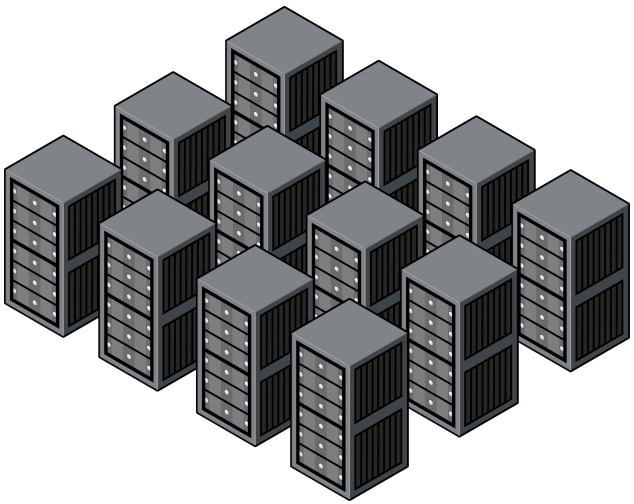
- $\mathbf{p}$ pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- Deadlock-free
- Predictable performances

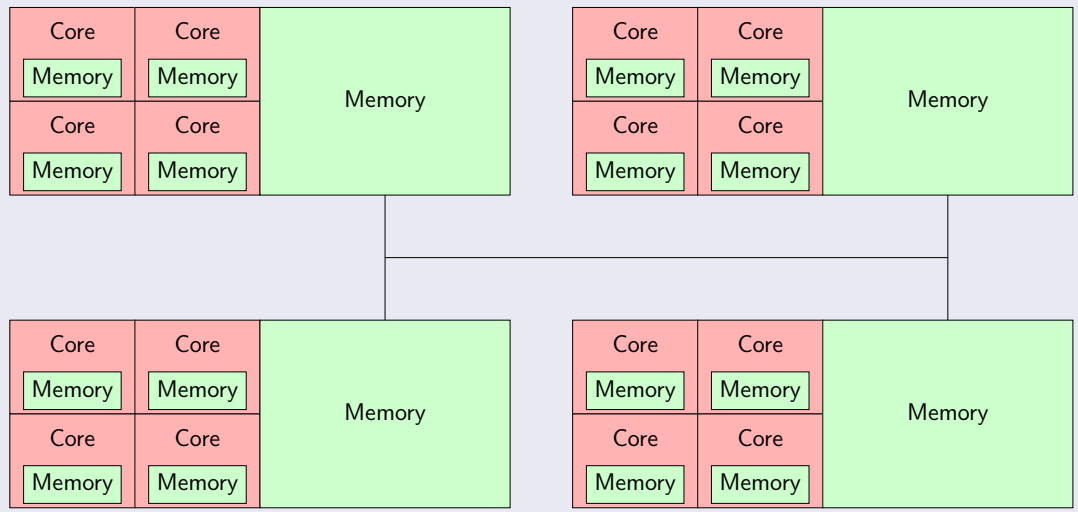


Hierarchical architectures



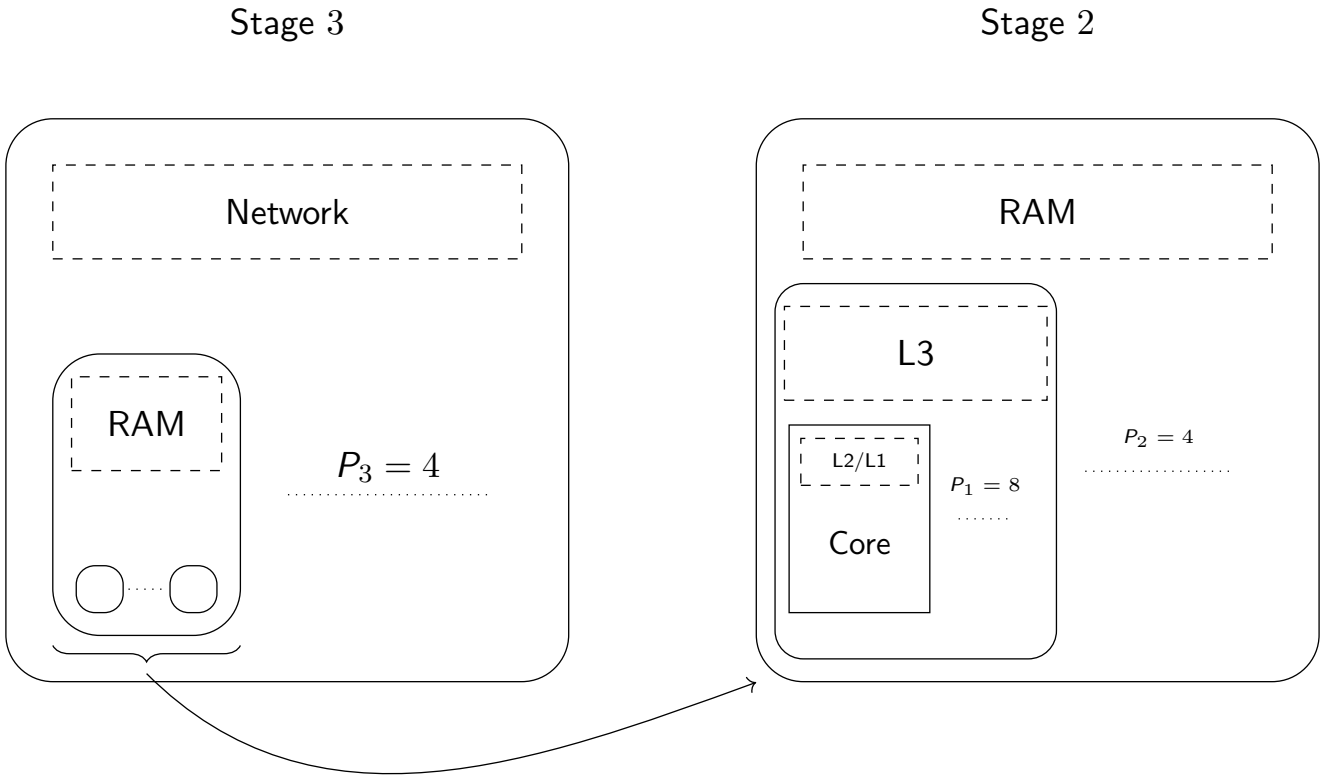
Characterised by:

- Interconnected units
- Both shared and distributed memories
- Hierarchical memories



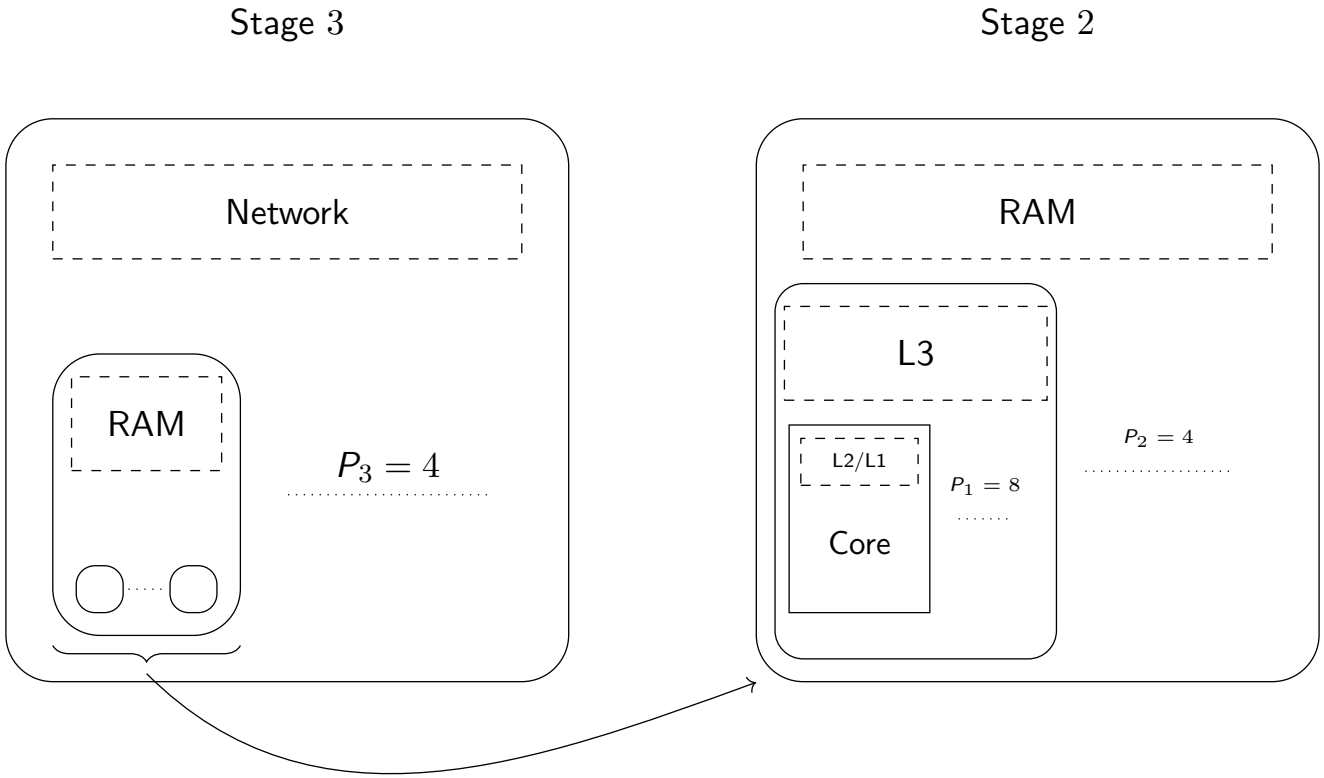
MULTI-BSP

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
4. With sub-synchronisation capabilities



MULTI-BSP

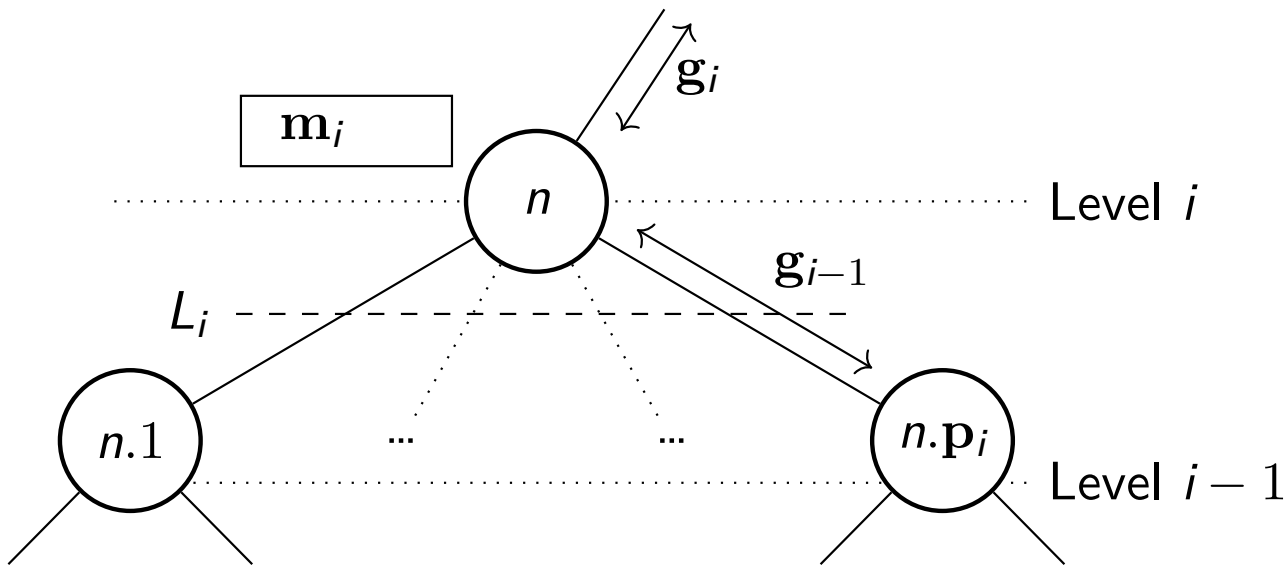
- Stage 3: 4 nodes with a network access
- Stage 2: one node has 4 chips plus RAM
- Stage 1: one chip has 8 cores plus L3 cache
- Stage 0: one core with L1/L2 caches



The MULTI-BSP model

Execution model

A level i superstep is:

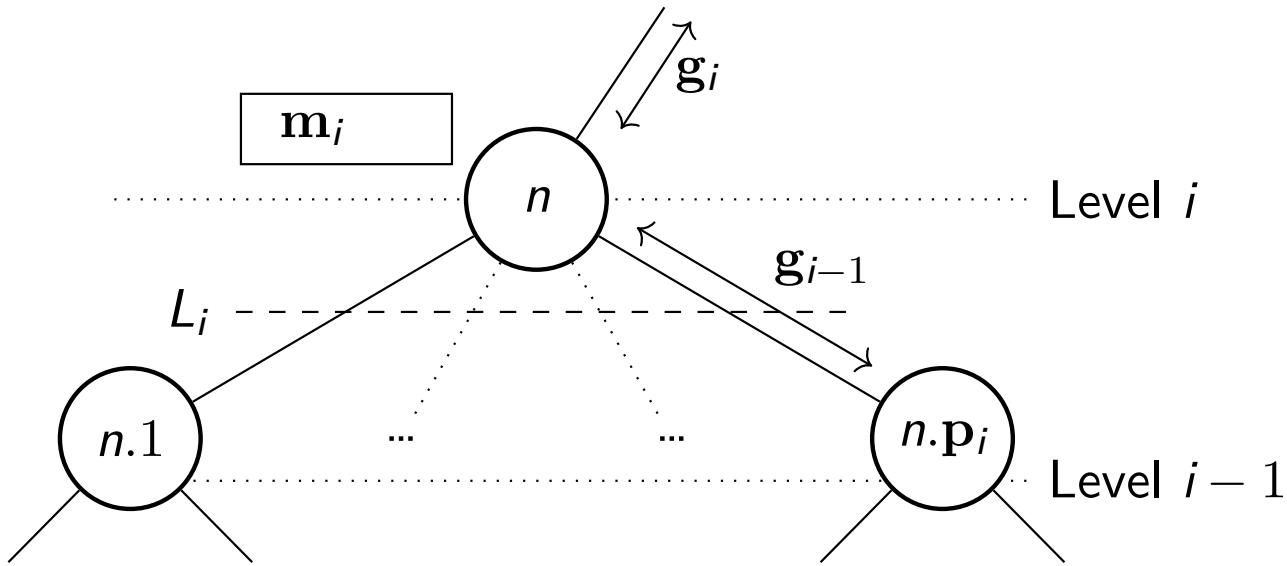


The MULTI-BSP model

Execution model

A level i superstep is:

- Level $i-1$ executes code independently

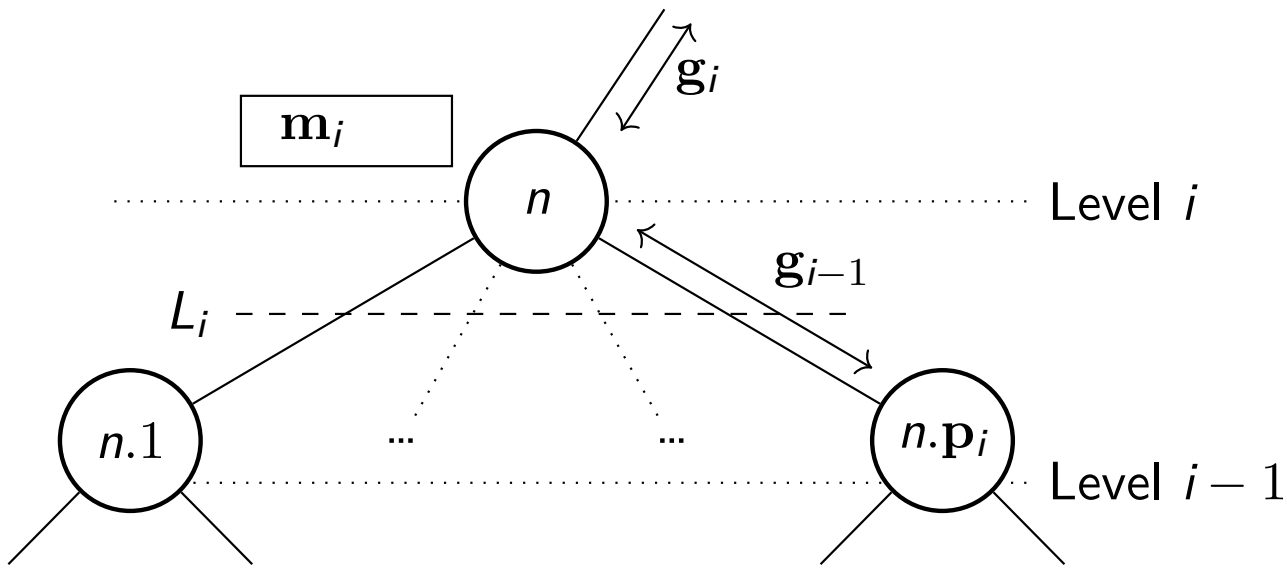


The MULTI-BSP model

Execution model

A level i superstep is:

- Level $i-1$ executes code independently
- Exchanges information with the m_i memory

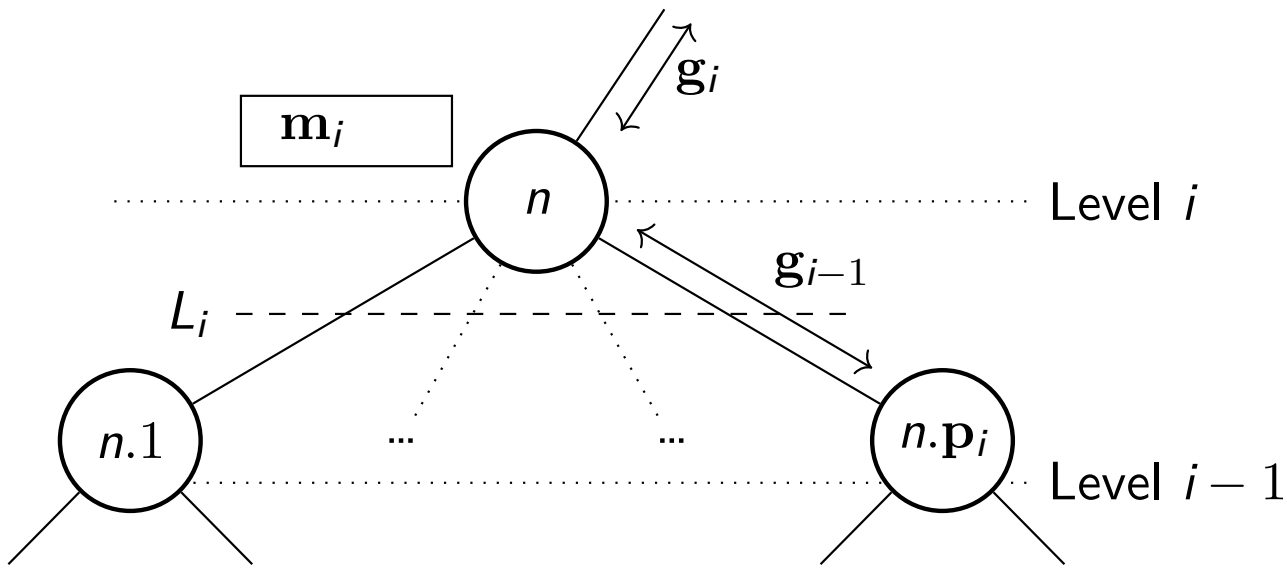


The MULTI-BSP model

Execution model

A level i superstep is:

- Level $i-1$ executes code independently
- Exchanges information with the m_i memory
- Synchronises



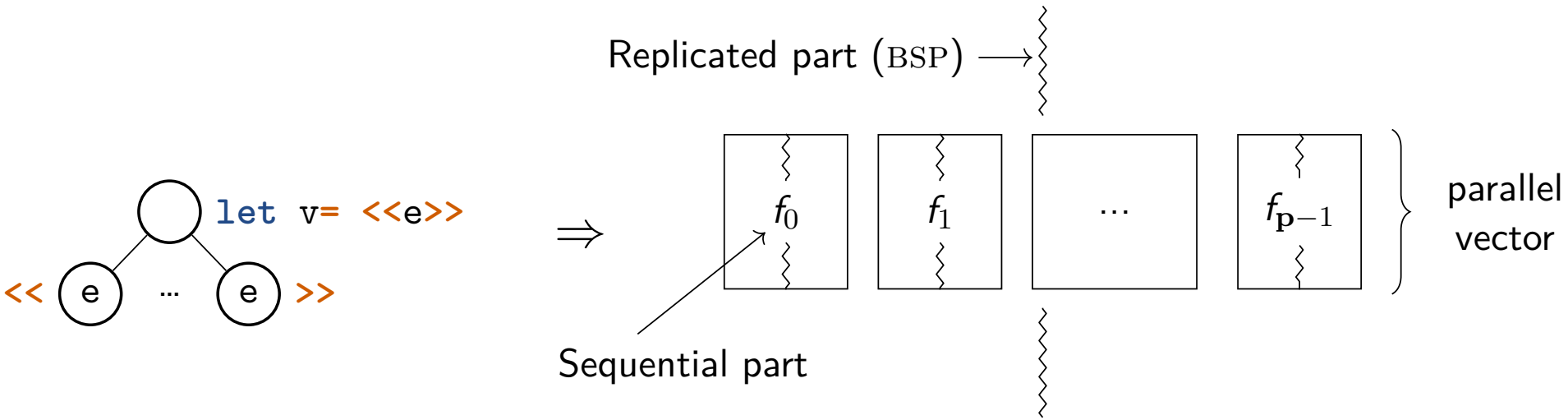
The MULTI-ML language

Basic ideas

The MULTI-ML language

Basic ideas

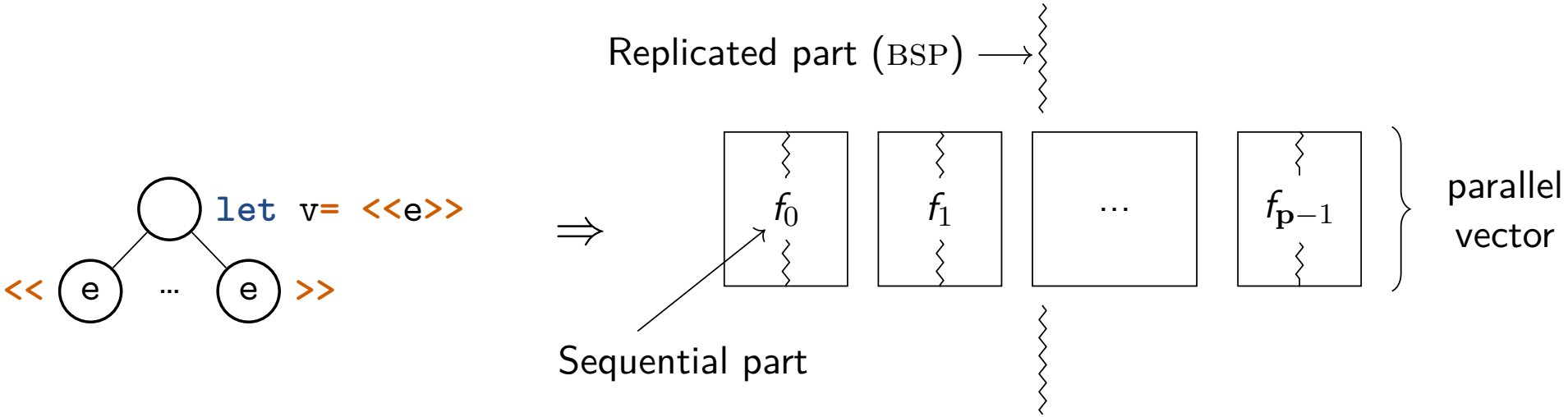
- BSML-like code on every stage of the MULTI-BSP architecture



The MULTI-ML language

Basic ideas

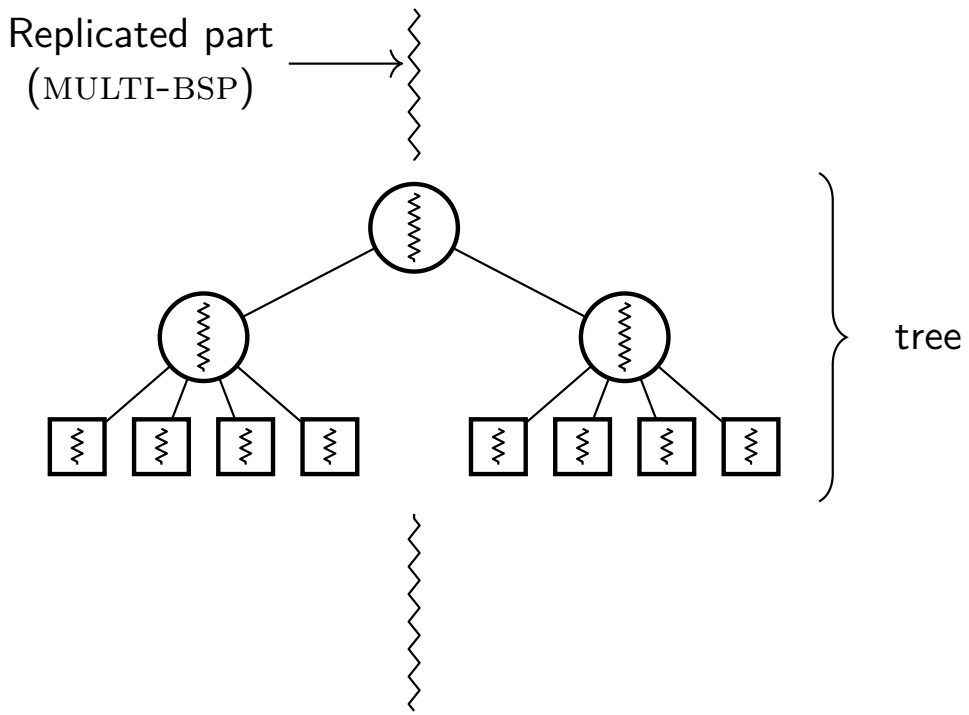
- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming



The MULTI-ML language

Basic ideas

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
- *Multi-functions* that recursively go through the MULTI-BSP tree



Current implementation

Current approach

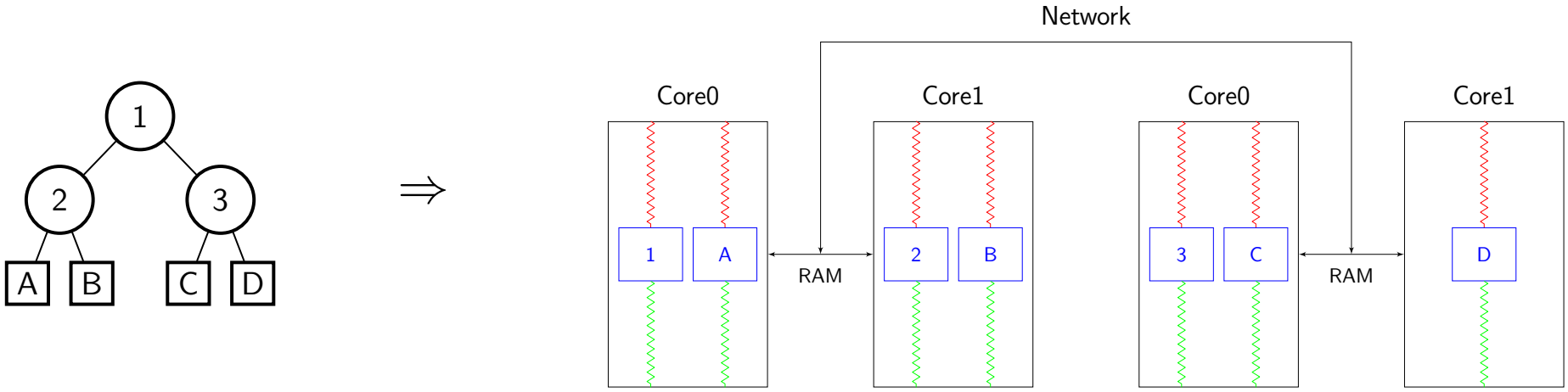
- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

Current version

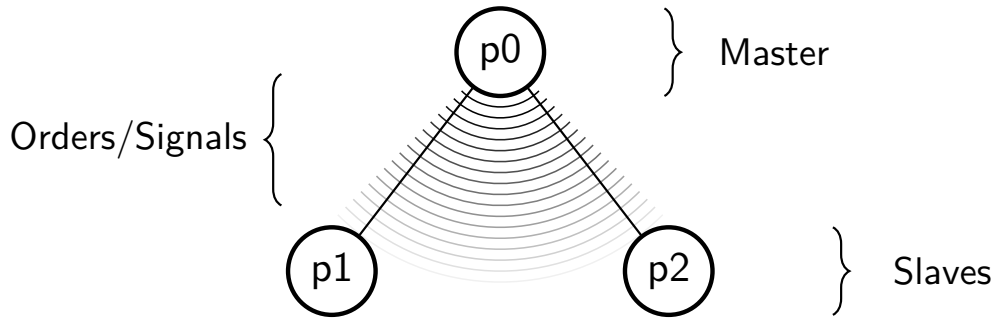
- Based on MPI
- SPMD
- One process for each nodes/leaves
- Distributed over physical cores
- Shared/Distributed memory optimisations

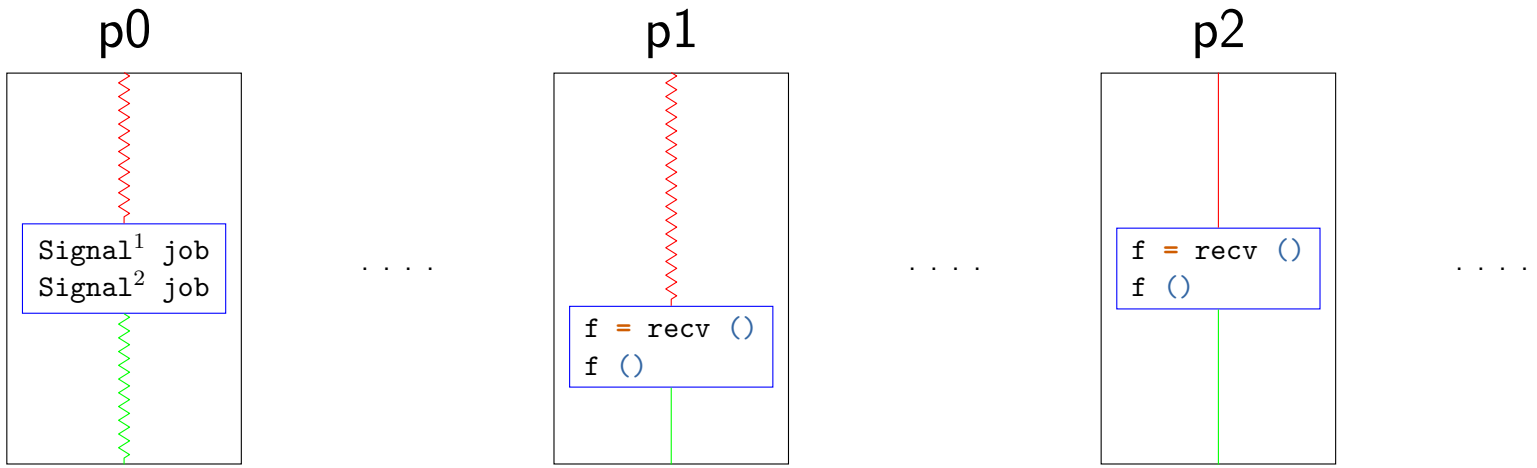
Current implementation

- One process per leaf/node
- Distributed over physical cores



Current implementation





Drawback of current implementation

- Closure communication
- Unnecessary communication
- Unnecessary processes

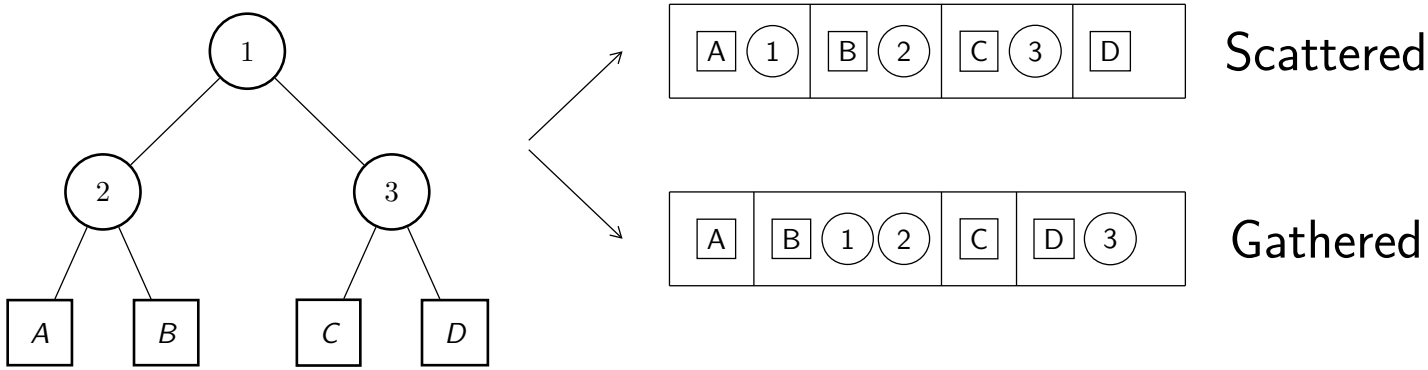
Table of Contents

Why a new implementation ?

- Minimise the number of daemons
- Abstract compilation scheme

The execution scheme

- The code is duplicated on each cores



Execution scheme

The execution scheme

- The code is duplicated on each cores
- MULTI-BSP code is executed sequentially everywhere

Execution scheme

The execution scheme

- The code is duplicated on each cores
- MULTI-BSP code is executed sequentially everywhere
- A multi-function is, basically, a scheduler

Execution scheme

The execution scheme

- The code is duplicated on each cores
- MULTI-BSP code is executed sequentially everywhere
- A multi-function is, basically, a scheduler
- Schedulers receives "continue (and daemon spawn)" or "end" execution

Execution scheme

The execution scheme

- The code is duplicated on each cores
- MULTI-BSP code is executed sequentially everywhere
- A multi-function is, basically, a scheduler
- Schedulers receives "continue (and daemon spawn)" or "end" execution
- Schedulers waits primitives instructions

Execution scheme

The execution scheme

- The code is duplicated on each cores
- MULTI-BSP code is executed sequentially everywhere
- A multi-function is, basically, a scheduler
- Schedulers receives "continue (and daemon spawn)" or "end" execution
- Schedulers waits primitives instructions
- Leaves are running sequential code

Primitive abstraction

Low level routines

- `Signal and rcv`: asynchronous communications
- `up`: communicate upward
- `run`: spawn daemon
- `WakeUpChildren and WakeUpAll`: broadcast values

Correctness

Theorem 1. *For a program* $P = e_1;;\cdots;;e_n$ *and if* $\mathcal{WF}\_(e_i)$,

- *If* $\mathcal{M} \vdash e_i \Downarrow_p^{\mathcal{L}} v_i$ *for each* e_i *then:*
  $\langle\langle \{\mathcal{E}_0, \lhd [\![P]\!]_{\text{M}} \rhd\}, \ldots, \{\mathcal{E}_{\boldsymbol{p_c}}, \lhd [\![P]\!]_{\text{M}} \rhd\} \rangle\rangle \Rightarrow^*$
  $\langle\langle \{\mathcal{E}'_0, \lhd v_0 \rhd\}, \ldots, \{\mathcal{E}'_{\boldsymbol{p_c}}, \lhd v_{\boldsymbol{p_c}} \rhd\} \rangle\rangle$
- *If* $\mathcal{M}, \vdash e_i \Downarrow_p^{\mathcal{L}} \infty$ *for one* e_i *then:*
  $\langle\langle \{\mathcal{E}_0, \lhd [\![P]\!]_{\text{M}} \rhd\}, \ldots, \{\mathcal{E}_{\boldsymbol{p_c}}, \lhd [\![P]\!]_{\text{M}} \rhd\} \rangle\rangle \Rightarrow^\infty$

Then, we get $\langle\langle e, ..., e \rangle\rangle \Rightarrow_{safe}$, *where* $\Rightarrow_{safe} \equiv \Rightarrow^* \cup \Rightarrow^\infty$.

Performances

Execution platform

Mirev3:

- 4 nodes with 2 octo-cores (INTEL xeon E5 – 2650 at 2.6Ghz)
- 64GB of memory per node
- 10Gbit/s network

Algorithm	i7	mirev3_1	mirev3	mirev3_{ht}
TDS	690	2070	8625	16100
FFT	972	2916	12150	22680

Functions closures overhead (in Bytes)

Table of Contents

Conclusion

Presented work

- Abstracts implementation
- Gives greater confidence on the compilation
- Reduces runtime overhead
- Reduces communications

Future work

- Prove the compilation scheme using the COQ proof assistant
- Extend approach to all OCAML features

Thank you for your attention ☺

Questions ?