Placement de processus (MPI) sur architecture multi-cœur NUMA

Emmanuel Jeannot, Guillaume Mercier

LaBRI/INRIA Bordeaux Sud-Ouest/ENSEIRB

Runtime Team

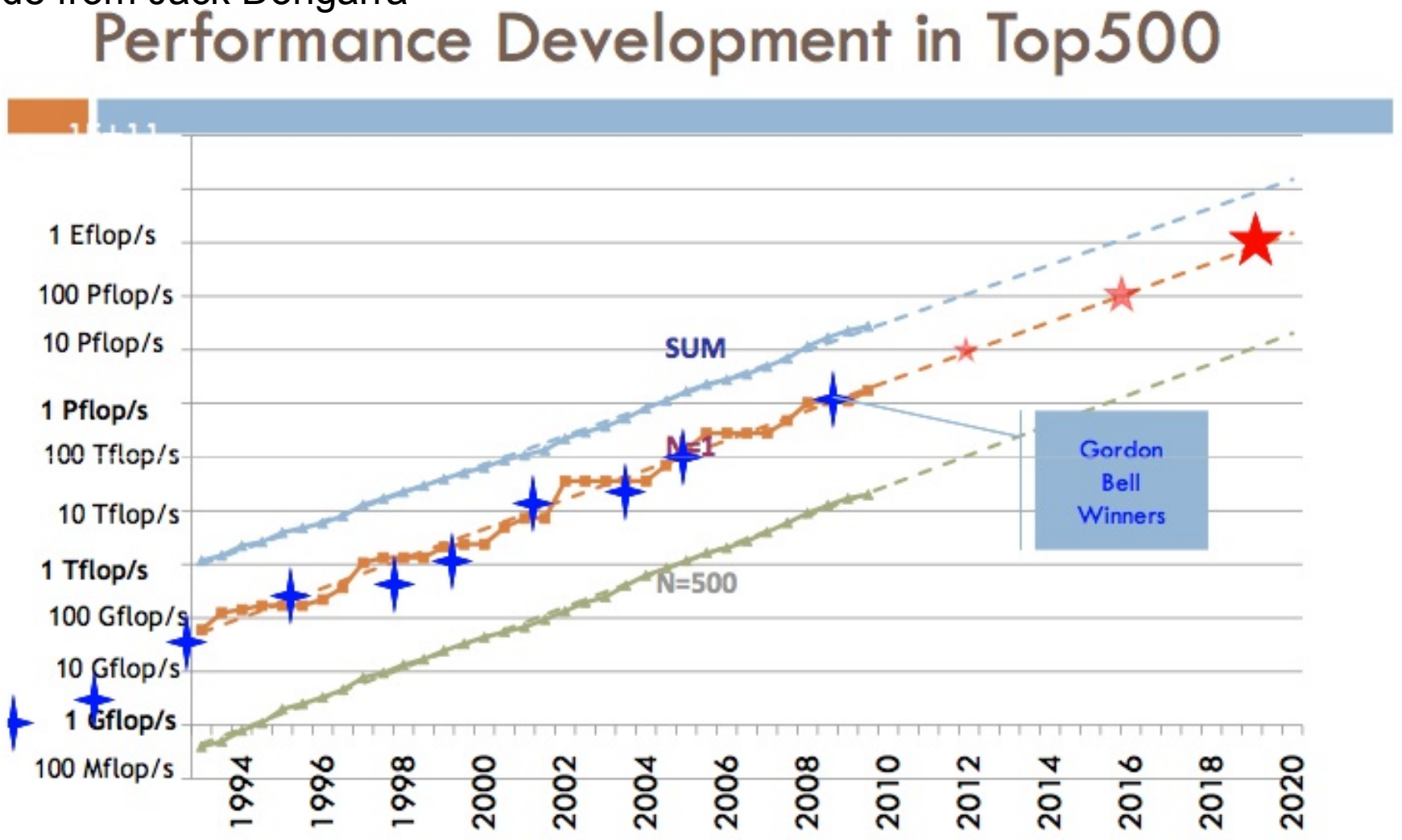
Lyon, journées groupe de calcul, november 2010

Emmanuel.Jeannot@inria.fr

Euro-Par 2011

Top 500

Slide from Jack Dongarra



Managing the memory in nowadays and future systems

Parallel systems toward exaflop…

More and more memory

More and more cores

Systems	2010	2018	Difference
System peak	2 Pflop/s	1 Eflop/s	
Power	6 MW	~20 MW	
System memory	0.3 PB	32 - 64 PB [.03 Bytes/Flop]	O(100)
Node performance	125 GF	1,2 or 15TF	
Node memory BW	25 GB/s	2 - 4TB/s [.002 Bytes/Flop]	
Node concurrency	12	O(1k) or 10k	O(100)-O(1000)
Total Node Interconnect BW	3.5 GB/s	200-400GB/s (1:4 or 1:8 from memory BW)	
System size (nodes)	18,700	O(100,000) or O(1M)	
Total concurrency	225,000	O(billion) [O(10) to O(100) for latency hiding]	
Storage	15 PB	500-1000 PB (>10x system memory is min)	
IO	0.2 TB	60 TB/s (how long to drain the machine)	
MTTI	days	O(1 day)	

The memory bus is already a bottleneck

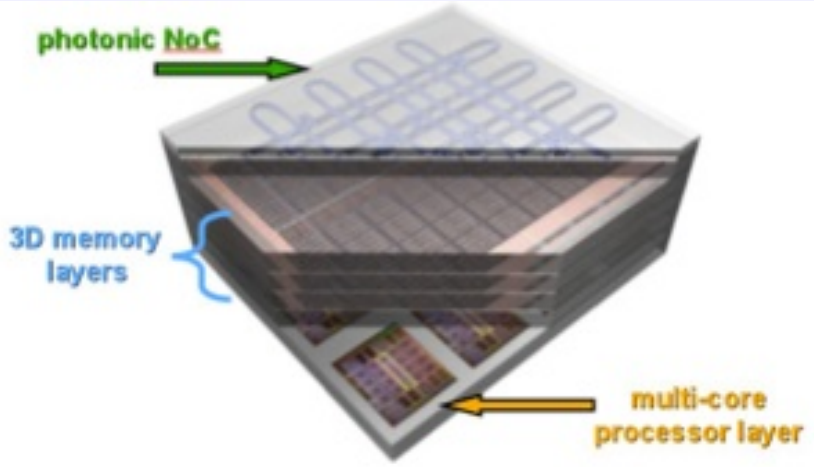
Feeding cache from memory

Already critical for sustaining performance in modern processor

Ex: gotoBLAS vs Atlas

Worst with 1000's of cores.

Solution: 3D stack memory?



Scalability

Strong scalability: gain performance by only adding new computing resources. Go faster on same problem size.

Weak scalability: when adding computing resources increase problem size. Increase the amount of processed data per core.

Most parallel programs only show weak scalability.

But, the memory per core is not going to increase any more

Forget strong scalability at the core level

(But still enjoy it at the machine level)

Consequences:

- Two levels of parallelism (core and machines)
- You can still use coarse-grain parallelism at machine level
- Need to find fine-grain parallelism in application to keep core busy (algorithmic challenge)
- Need to take care of access patterns to avoid cache misses and increase/improve pipelining of data (systolic algorithm)

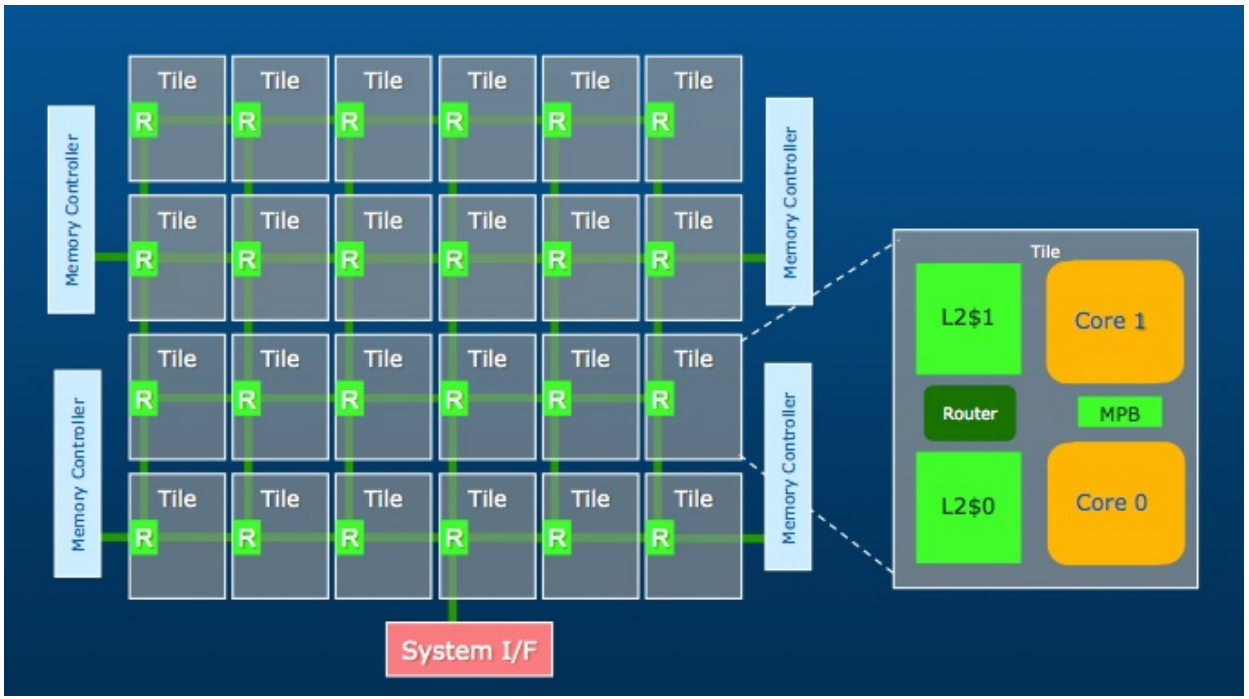
Possible solutions

- Hybrid programming
  - Mixing message passing and thread programming
  - Example: MPI+OpenMP
- Virtual memory
  - Memory abstraction that hides hierarchy and NUMA effects
  - Management of migration, access, thread placement/scheduling

But that's not over!

SCC: single chip cloud (Intel prototype)

"Cluster-on-die" architecture – 48 Pentium Processor cores (P54C - x87FP only)



Single Chip Cloud

Advanced power management:

- software control DVFS for core
- frequency control

Memory:

- Up to 64GB DDR3 via 4 memory controllers @ 21.3GB/s
- 16KB SRAM in each tile as Message Passing Buffer (MPB)

Caching:

- 32KB L1 per core (16KB I,D), 12MB L2 cache (256KB/core)
- **No HW cache-coherent shared memory**

Addressing:

- Core physical to system physical addresses in 16MB sections
- Memory mapped configuration & control registers

Cache Coherency

The cache is a copy of the main memory

The coherency between caches and memory is usually done by hardware

The application does not need to be aware of the cache hierarchy.

But not always:

- Cell Processors
- Future multicore processors?

Cache coherent HW mechanism do not scale:

- To be managed by the application?
- Need to take care on how data are accessed and when
- Split memory into independent chuck (like in distributed memory)?

Some research must done in the memory management

Future multicore processors will not make things simpler

Less memory per core (weak scalability)

End of HW cache coherent mechanism?

One issue: need to carefully manage the way applications access and process data.

(MPI) Process Placement

Computer-science research [Euro Par 2010]

INRIA Runtime team and Urbana Champaign (L. Kale/Charm++), BlueWaters project

Preliminary results (will probably not be implemented as such)

Introduction

- MPI is the main standard for programming parallel applications
- It provides portable code across platforms
- What about performance portability?

Performance of MPI programs

Depend on many factors:

- Implementation of the standard (e.g. collective com.)
- Parallel algorithm(s)
- Implementation of the algorithm
- Underlying libraries (e.g. BLAS)
- Hardware (processors, cache, network)
- etc.
- and …

Process placement

The MPI model makes little (no?) assumption on the way MPI processes are mapped to resources

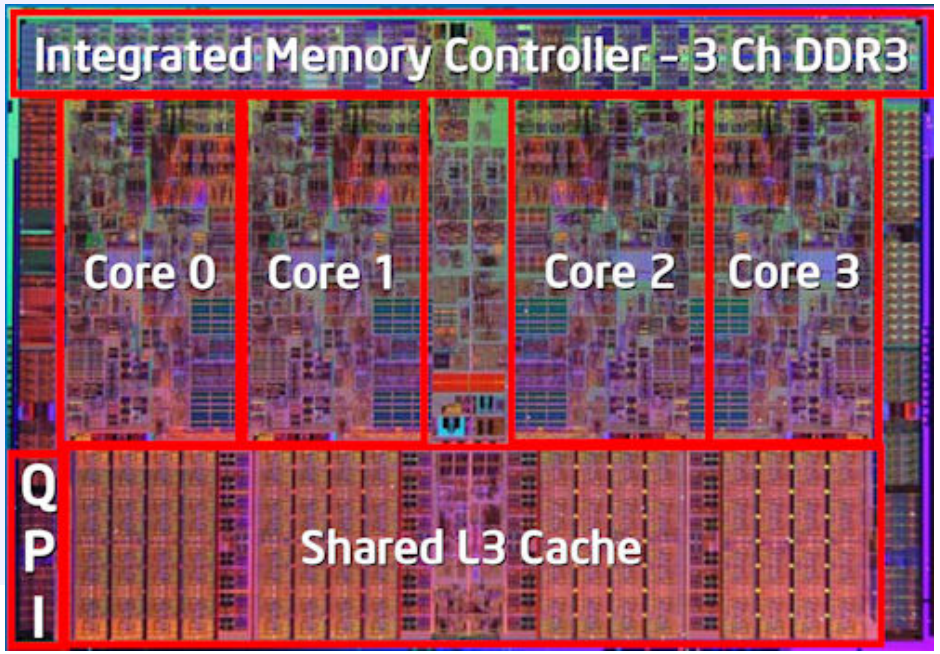
It is often assume that the network topology is flat and hence the process mapping has little impact on the performance

The network topology is not flat

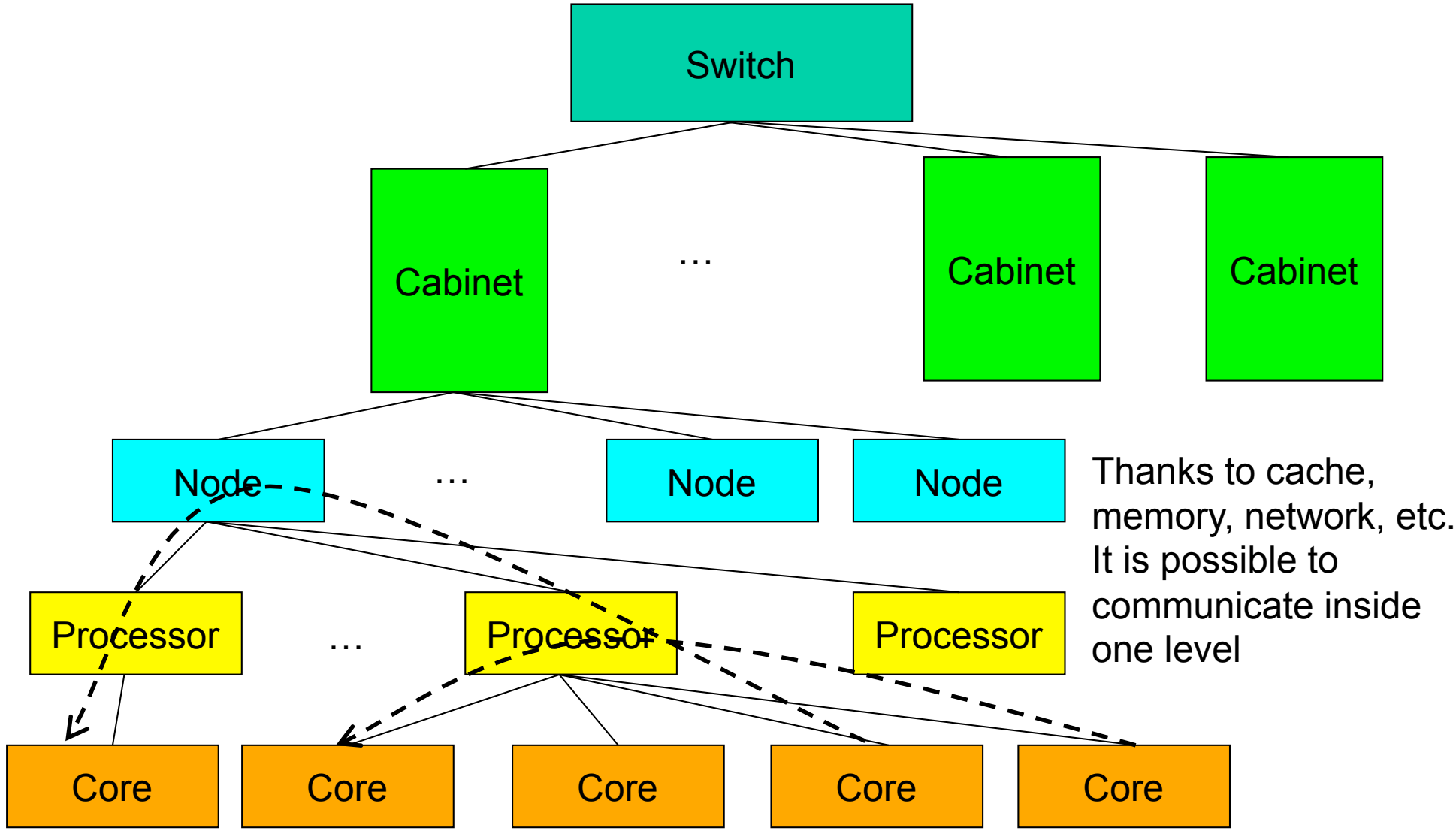
Due to multicore processors current and future parallel machines are hierarchical

Communication speed depend on:

- receptor and emitter
- Cache hierarchy
- Memory bus
- Interconnection network
- etc.



Example of typical parallel machine



Rationale

Not all the processes exchange the same amount of data

The speed of the communications, and hence performance of the application depend on the way processes are mapped to resources.

Process placement problem

Given:

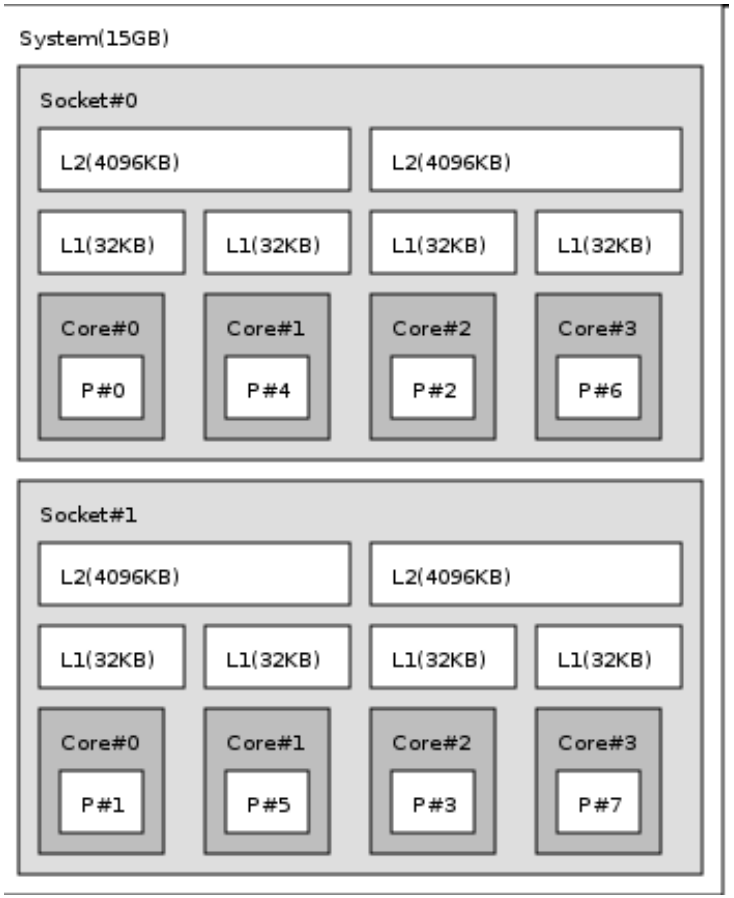
- The parallel machine topology
- The processes communication pattern

Map processes to resources (cores) to reduce the communication cost.

Obtaining the toplogy

HWLOC (portable hardware locality)

- Runtime and OpenMPI team
- portable abstraction (across OS, versions, architectures, ...)
- Hierarchical topology
- Modern architecture (NUMA, cores, caches, etc.)
- ID of the cores
- C library to play with
- etc.



Obtaining the communication pattern

No automatic way so far

For now done through application monitoring

Left to future work (static code analysis?), memory/ object monitoring.

State of the art

Process placement fairly well studied problem:

- Graph Partitioning (Scotch/Metis): do not take hierarchy into account.
- [Träff 2002]: placement through graph embedding and graph partitioning
- MPIPP [Chen et al. 2006]: placement through local exchange of processes until no gain is achievable
- [Clet-Ortega & Mercier 09] : placement through graph renumbering

Example

T: topology matrix

0	100	100	10	1000	100	100	10
100	0	10	100	100	1000	10	100
100	10	0	100	100	10	1000	100
10	100	100	0	10	100	100	1000
1000	100	100	10	0	100	100	10
100	1000	10	100	100	0	10	100
100	10	1000	100	100	10	0	100
10	100	100	1000	10	100	100	0

Communication speed between processor 2 and processor 3

C: communication matrix

0	1000	10	1	100	1	1	1
1000	0	1000	1	1	100	1	1
10	1000	0	1000	1	1	100	1
1	1	1000	0	1	1	1	100
100	1	1	1	0	1000	10	1
1	100	1	1	1000	0	1000	1
1	1	100	1	10	1000	0	1000
1	1	1	100	1	1	1000	0

Amount of data exchanged between process 1 and process 3

Formal problem

Input: T and C two n by n matrices

Output: σ a permutation of size n

Constraint: minimize

$$\sum_{i \neq j} C(i,j) / T(\sigma_i, \sigma_j)$$

Example of solutions:

Round-robin:	0 1 2 3 4 5 6 7	241.3
Graph embedding:	3 7 4 0 6 2 5 1	210.52
Optimal (B&B):	0 4 1 5 2 6 3 7	29.08

Complexity of the problem

Finding the optimal permutation is NP-Hard

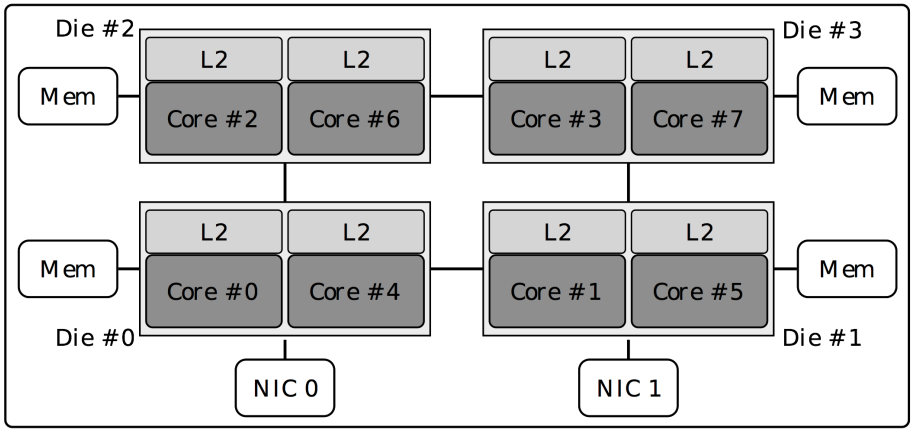
However, posed this way the problem does not take into account the hierarchy of the topology

Question: does taking the hierarchy into consideration help?

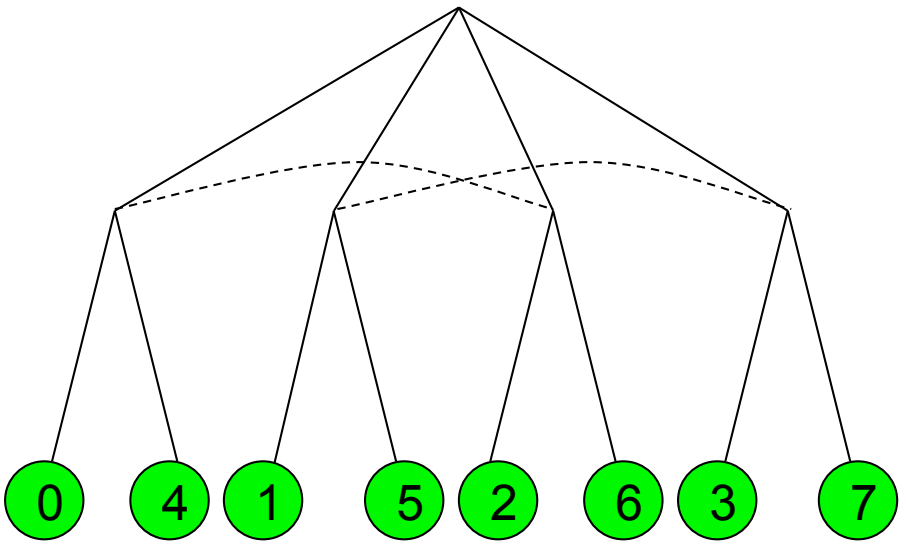
Taking into account the hierarchy

Topology matrix

0	100	100	10	1000	100	100	10
100	0	10	100	100	1000	10	100
100	10	0	100	100	10	1000	100
10	100	100	0	10	100	100	1000
1000	100	100	10	0	100	100	10
100	1000	10	100	100	0	10	100
100	10	1000	100	100	10	0	100
10	100	100	1000	10	100	100	0



HWLOC output



Mapping the communication matrix to the topology tree: the TreeMatch algorithm

Idea: for each level of the tree, group nodes to minimize remaining communication.

Group size should be equal to the arity of the considered level

Example

C: communication matrix

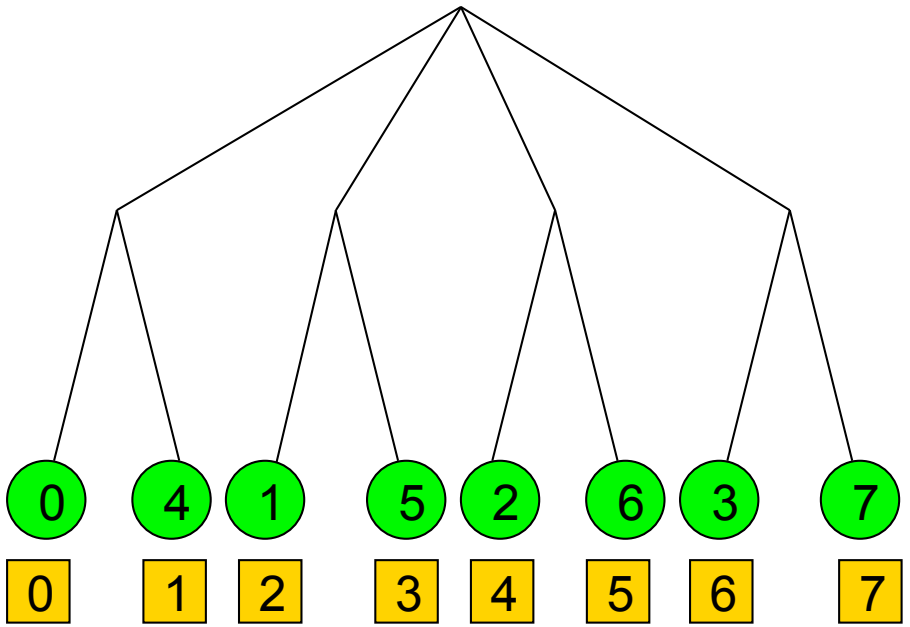
0	1000	10	1	100	1	1	1
1000	0	1000	1	1	100	1	1
10	1000	0	1000	1	1	100	1
1	1	1000	0	1	1	1	100
100	1	1	1	0	1000	10	1
1	100	1	1	1000	0	1000	1
1	1	100	1	10	1000	0	1000
1	1	1	100	1	1	1000	0

0 ↔ 1 2 ↔ 3 4 ↔ 5 6 ↔ 7

+

Grouped matrix

0	1012	202	4
1012	0	4	202
202	4	0	1012
4	202	1012	0



A more complex example

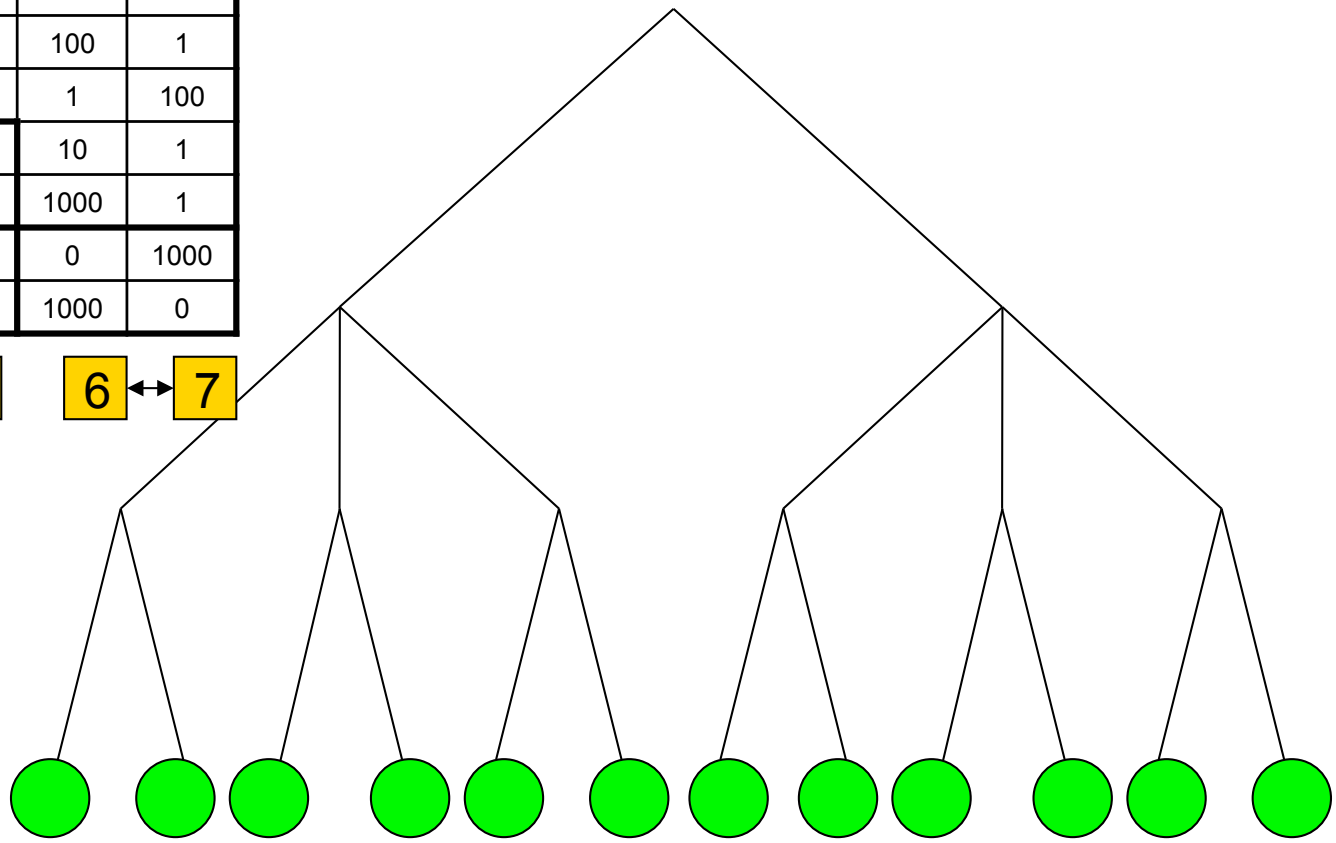
0	1000	10	1	100	1	1	1
1000	0	1000	1	1	100	1	1
10	1000	0	1000	1	1	100	1
1	1	1000	0	1	1	1	100
100	1	1	1	0	1000	10	1
1	100	1	1	1000	0	1000	1
1	1	100	1	10	1000	0	1000
1	1	1	100	1	1	1000	0

0 ↔ 1 2 ↔ 3 4 ↔ 5 6 ↔ 7

\+

Grouped matrix

0	1012	202	4
1012	0	4	202
202	4	0	1012
4	202	1012	0



A more complex example

0 ↔ 1 2 ↔ 3 4 ↔ 5 6 ↔ 7

Grouped matrix

0	1012	202	4
1012	0	4	202
202	4	0	1012
4	202	1012	0

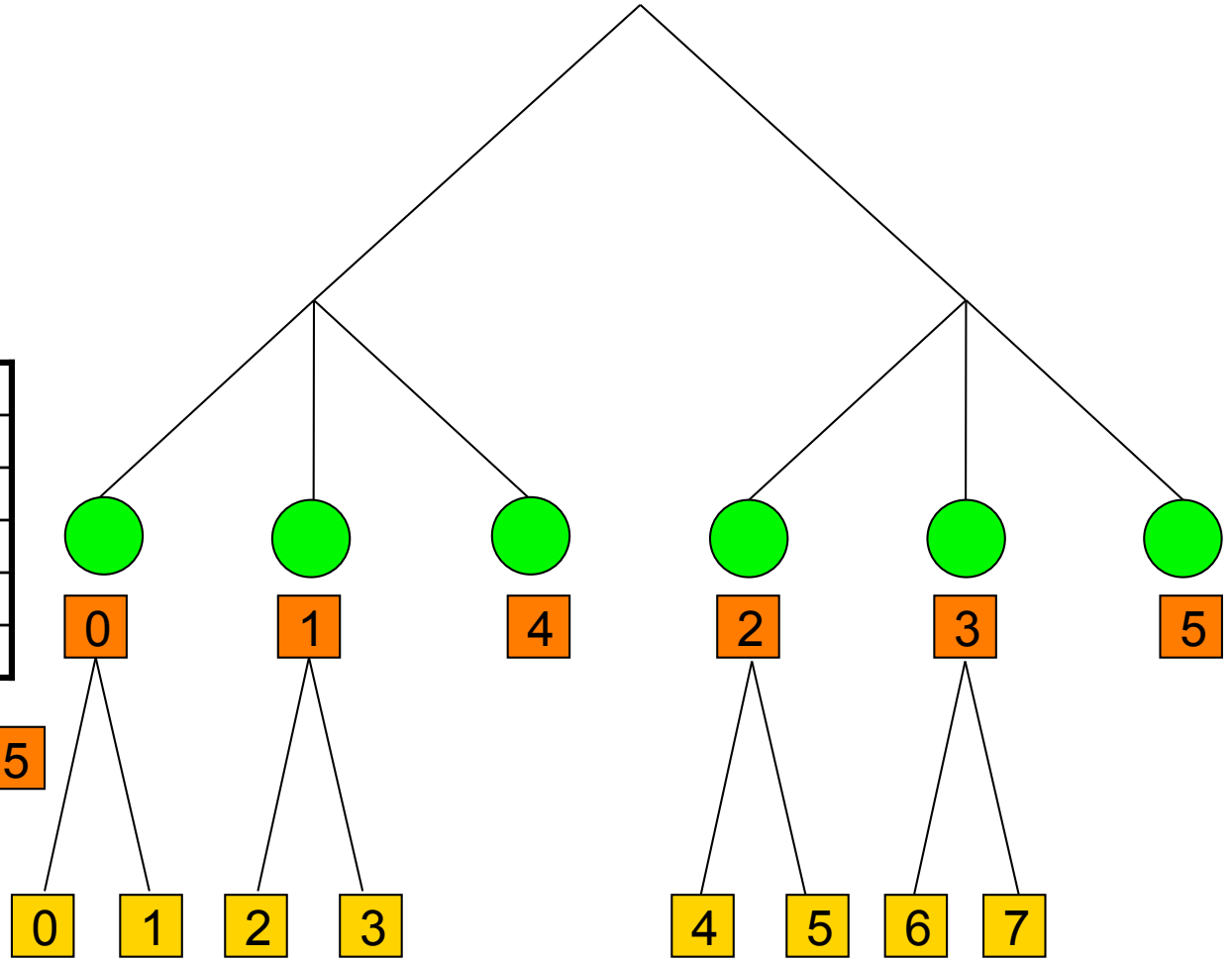
Extended grouped matrix

0	1012	202	4	0	0
1012	0	4	202	0	0
202	4	0	1012	0	0
4	202	1012	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

0 ↔ 1 ↔ 4 2 ↔ 3 ↔ 5

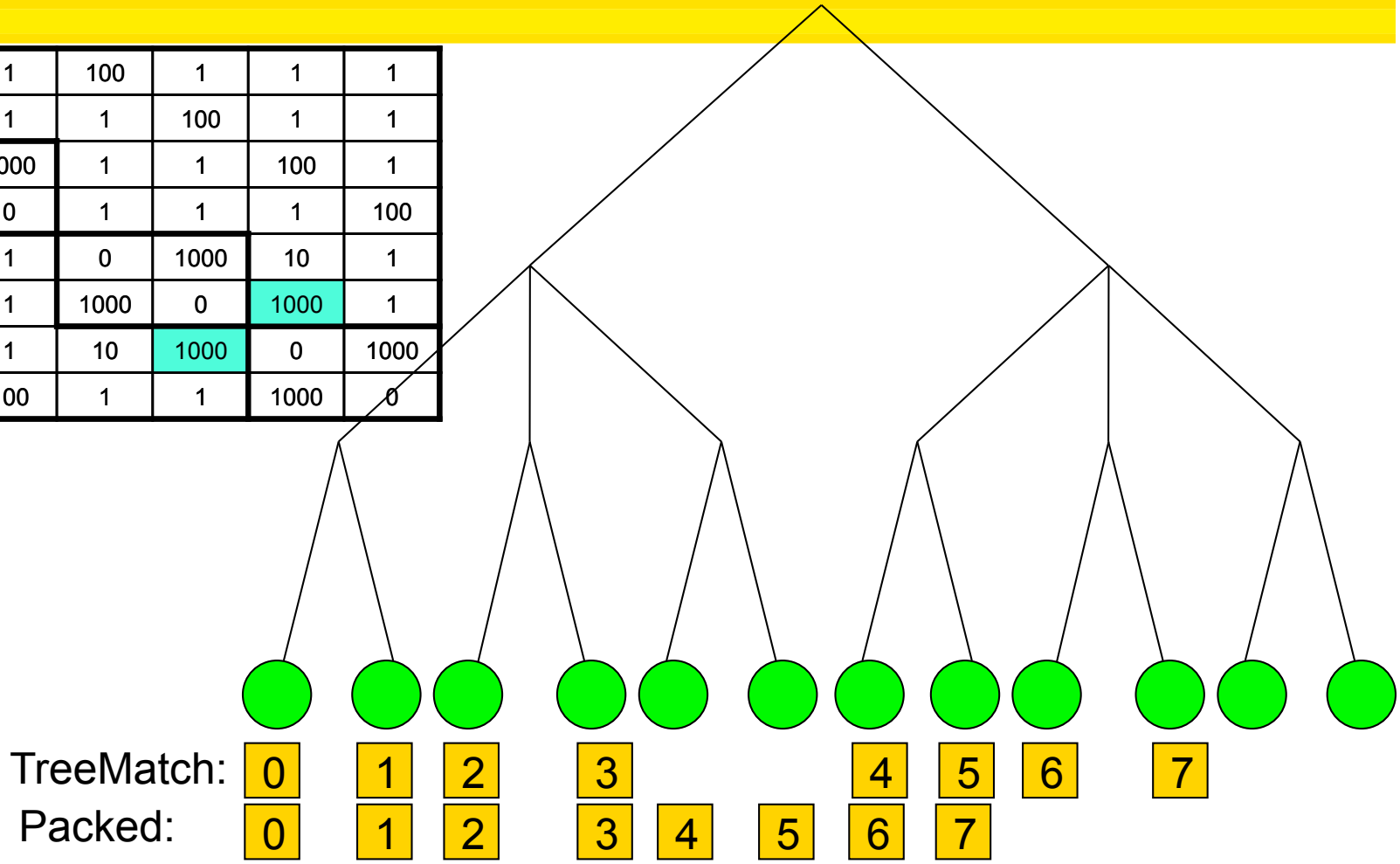
Grouped matrix

0	412
412	0



A more complex example

0	1000	10	1	100	1	1	1
1000	0	1000	1	1	100	1	1
10	1000	0	1000	1	1	100	1
1	1	1000	0	1	1	1	100
100	1	1	1	0	1000	10	1
1	100	1	1	1000	0	1000	1
1	1	100	1	10	1000	0	1000
1	1	1	100	1	1	1000	0

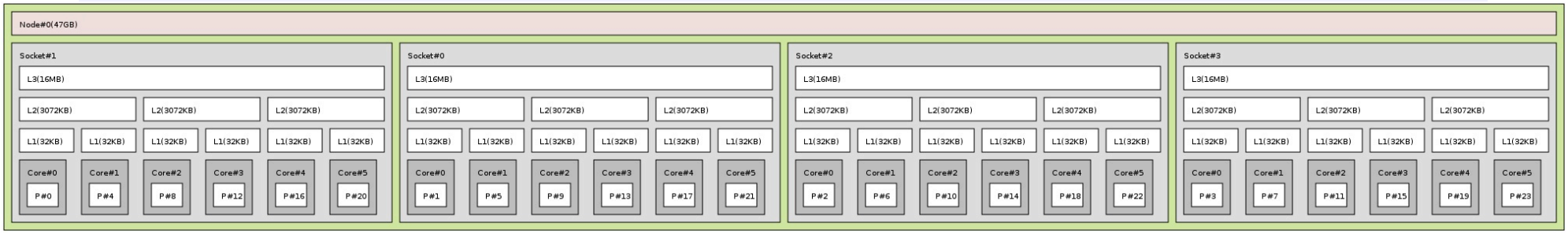


Packed solution worst than the TreeMatch one because there is a large communication between processes 5 and 6

Experiments

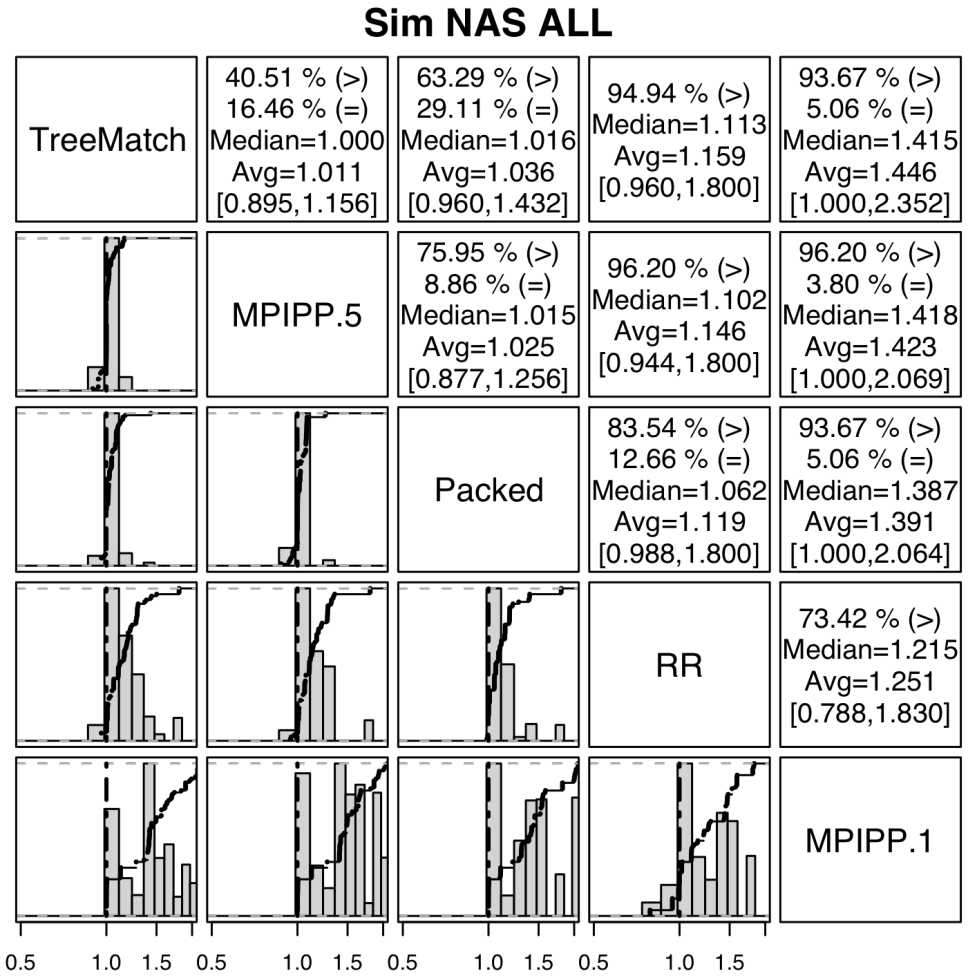
We use the NAS benchmarks:

- All the kernels
- Class: A,B,C,D
- Size: 16, 32/36, 64
- On highly NUMA machine (4 nodes of 4 Xeon quad-core Dunnington)
- Comparison with : MPIPP [Chen et al. 2006] (two versions), Packed (by sub-tree), Round-Robin (process i to core i).



Simulation Results

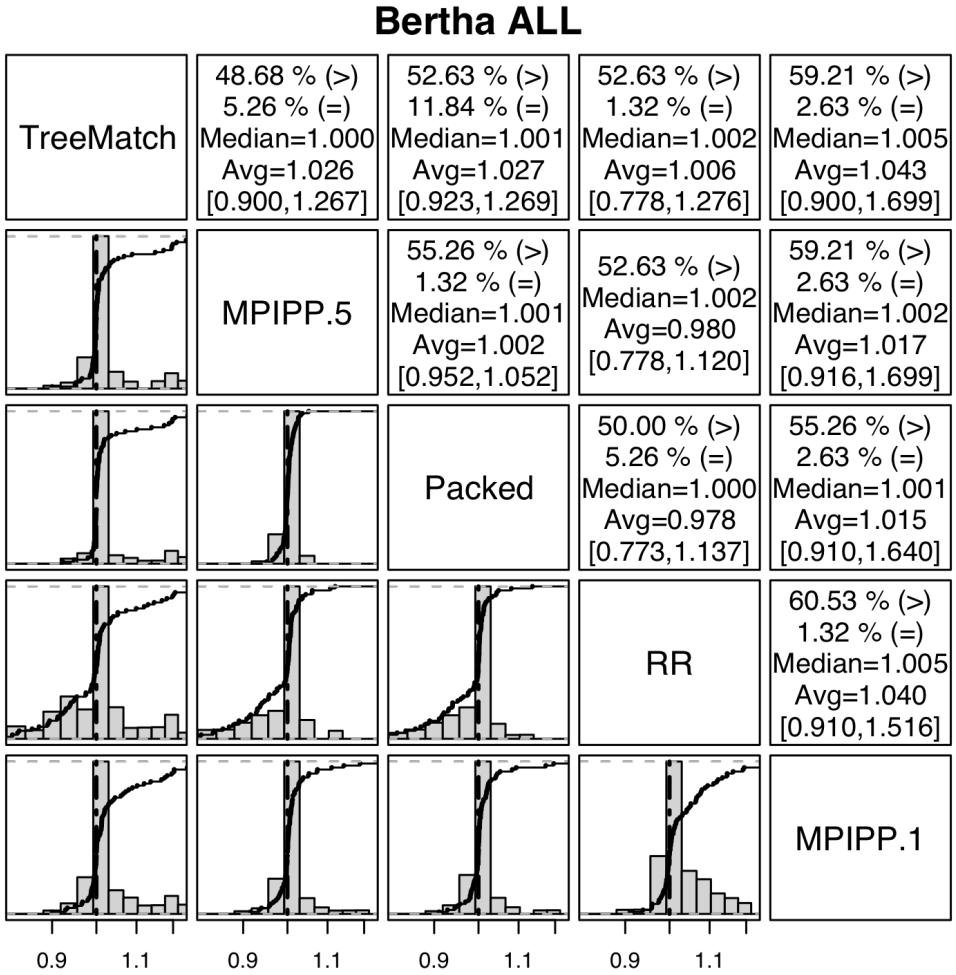
We simulate the execution time using our model



NAS on the real machine

Best strategy:
TreeMatch

Some very bad results against round-robin

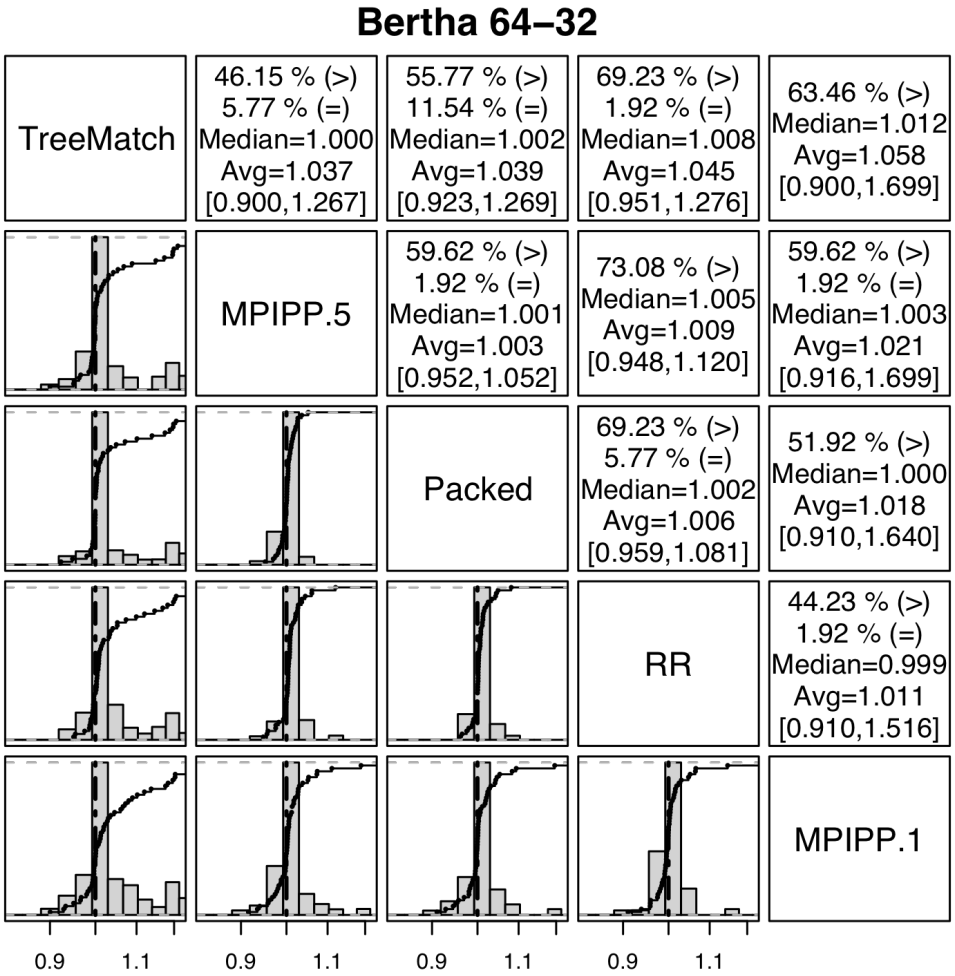


32-64 processes

Several nodes are used

Best: TreeMatch (up to 27% improvement).

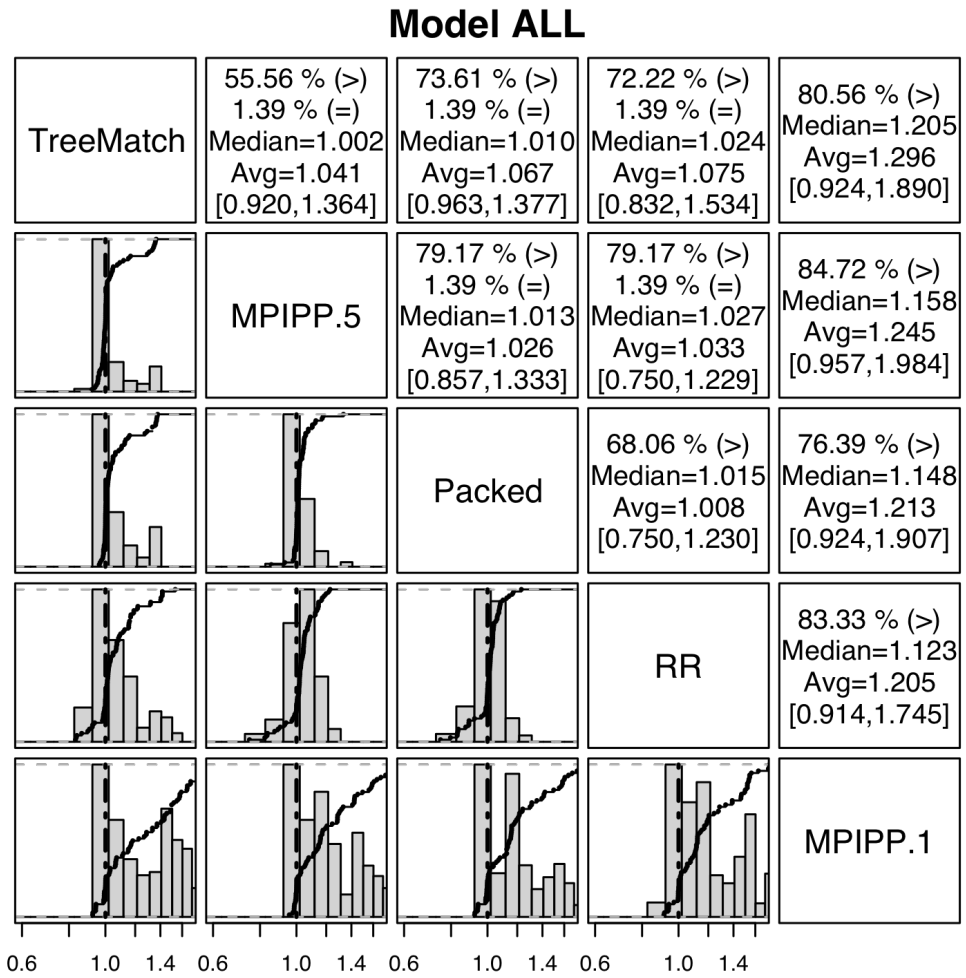
Comparable to MPIPIP.5 (but faster runtime)



Communication Only Application

We extract the communication pattern of the NAS

Up to 37% improvement



Conclusion

Mapping processes can help to reduce the communication cost

TreeMatch: an algorithm to perform such mapping

- Bottom-up
- Fast
- Does not require that the number process equals the number of cores/ processors
- Optimal in some cases

Early results:

- TreeMatch: best method on average
- Works well when more than one node is used
- Difference between model and reality

Future work

On going work

Future work:

- Top Down?
- Improve model (NUMA effect)
- Hybrid case
- Dymamic adaptation
- Automation
- Process topology interface of MPI 2.2 (With J. L. Träff).