9th Annual Workshop on CHARM++ and its Applications

Improving CHARM++ Performance with a NUMA-aware Load Balancer

Laércio Lima Pilla[1,2], Christiane Pousa[2], Daniel Cordeiro[2,3], Abhinav Bhatele[4], Philippe O. A. Navaux[1], Jean-François Méhaut[2], Laxmikant V. Kale[4]

[1]Federal University of Rio Grande do Sul – Porto Alegre, Brazil
[2]Grenoble University – Grenoble, France
[3]University of São Paulo – São Paulo, Brazil
[4]University of Illinois at Urbana-Champaign – Urbana, IL, USA

Summary

How we used NUMA architectural information to build a CHARM++ load balancer and obtained improvements on overall performance.

Agenda

NUMA

Our Load Balancer: NUMALB

Experimental Setup

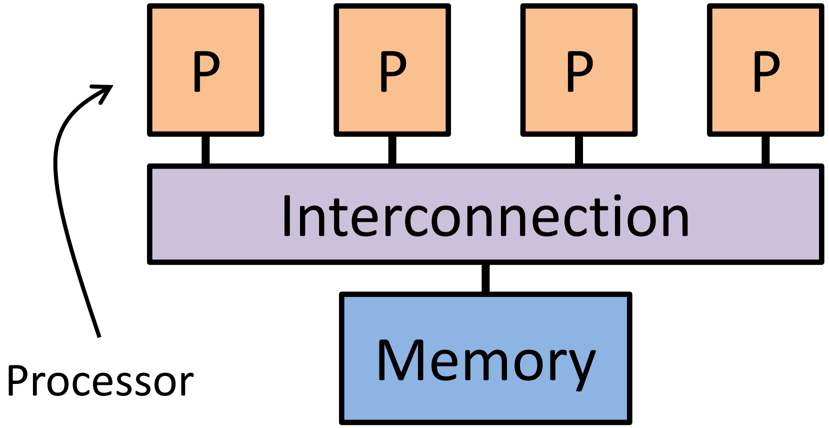
Results

Concluding Remarks

UMA x NUMA

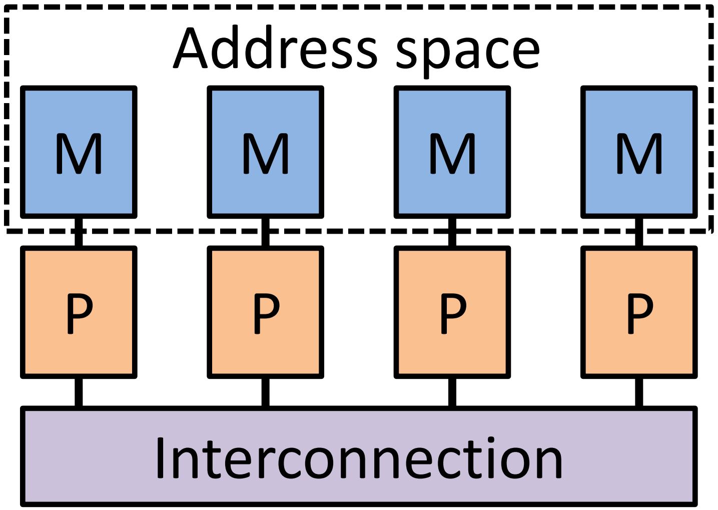
Uniform Memory Access

- Centralized shared memory
  - Uniform latencies
- Data placement does not matter



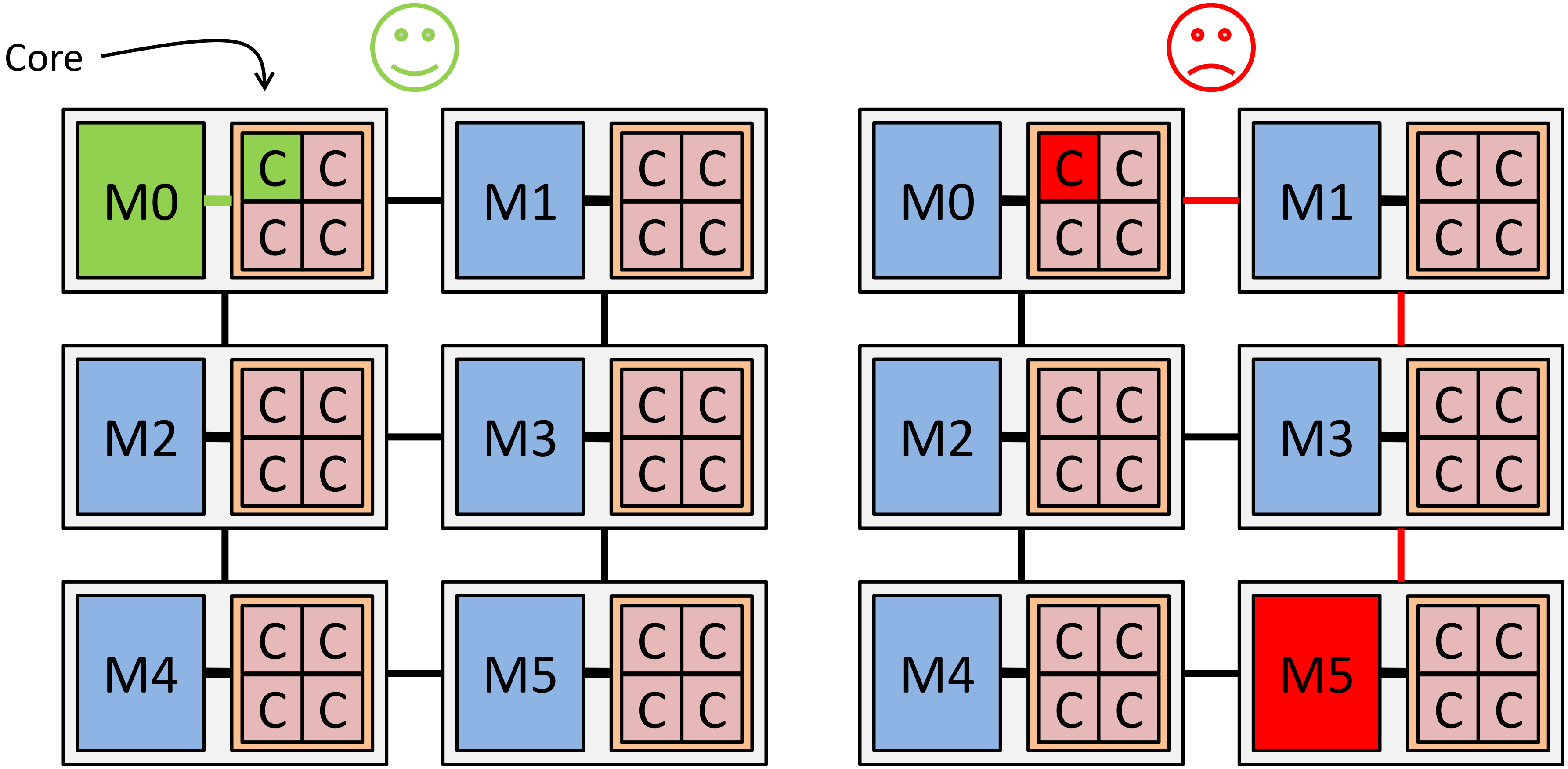
Non-Uniform Memory Access

- Distributed shared memory
  - Non-uniform latencies
- **Data placement matters**



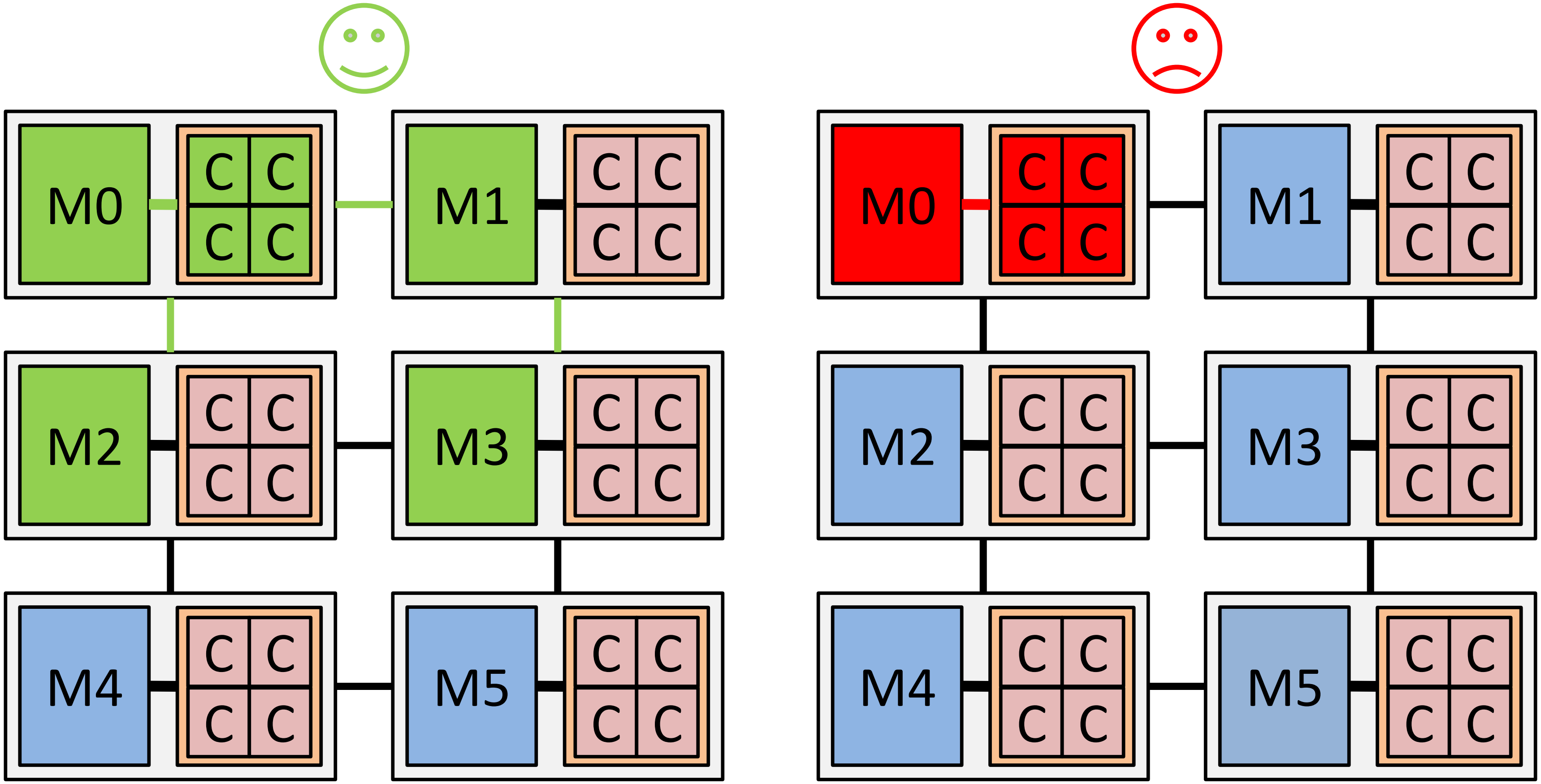
NUMA

Reduce latencies



NUMA

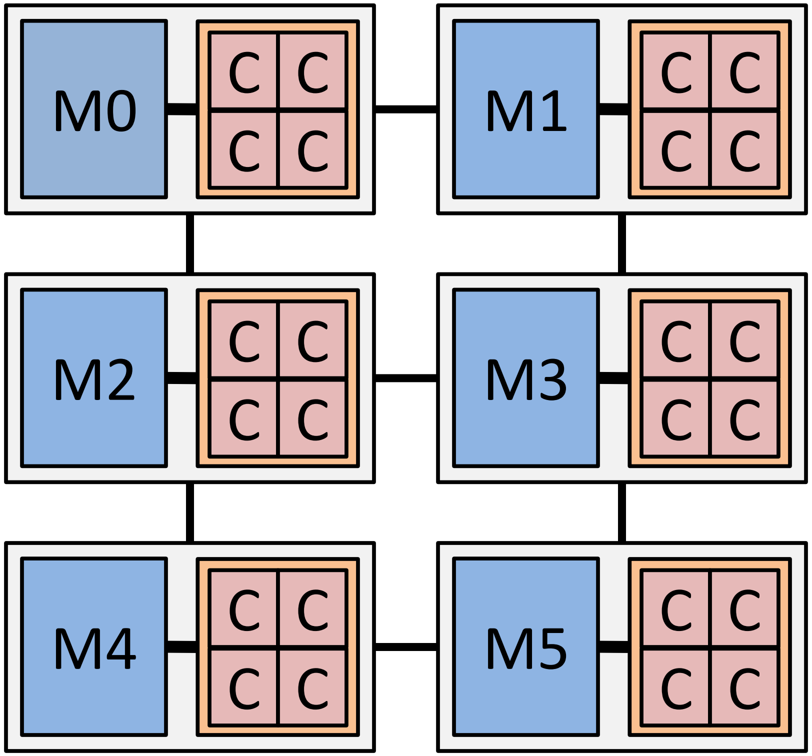
Reduce contention/improve bandwidth



NUMA

CHARM++ does not consider these characteristics

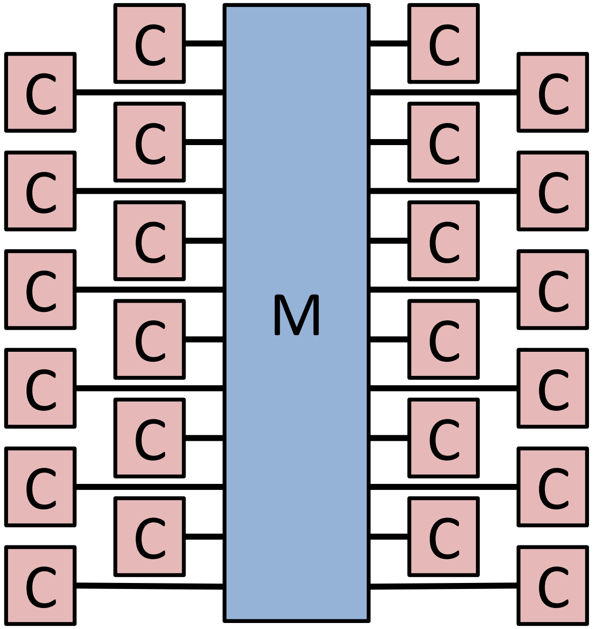
Physical organization



CHARM++'s vision (UMA)

No memory hierarchy

No locality



Agenda

~~NUMA~~

Our Load Balancer: NUMALB

Experimental Setup

Results

Concluding Remarks

Load Balancer

- Application data – CHARM++ LB framework
  - – Processor load: execution time
  - – Chare load: execution time
  - – Communication graph: size and number of messages
- NUMA topology – archTopology (our library)
  - – Core to NUMA node (socket) hierarchy mapping
  - – NUMA factor

$$\text{NUMA factor}(i, j) = \frac{\text{Read latency from } i \text{ to } j}{\text{Read latency on } i}$$

Load Balancer

- Heuristic
  - Task mapping is NP-Hard
  - No initial assumptions about the application
- List scheduling
  - Put tasks on a priority list by load
  - Assign tasks to the processor with the smallest cost on a greedy fashion
- Improve performance
  - by reducing unbalance
  - by reducing remote communication costs
  - while avoiding migrations (data movement costs)

Load Balancer

- Cost function

$$cost(c,p) = load(p) + a \times (r_{comm}(c,p) \times NUMA\ factor - l_{comm}(c,p))$$

Where

c: chare

p: core

$load(p)$: load (execution time) on core p

$r_{comm}(c,p)$: number of messages sent by chare c to chares on other NUMA node

$l_{comm}(c,p)$: number of messages sent by chare c to chares on the same NUMA node

a: communication weight

Load Balancer

NUMALB’s Algorithm

Input: *C* set of chares, *P* set of cores, *M* mapping

Output: *M’* mapping of chares to cores

```
$M' \leftarrow M$
while $c \neq \emptyset$ do                                      for the number of chares
    $c \leftarrow v \mid v \in \arg\max_{u \in C} load(u)$       take heaviest chare
    $C \leftarrow C \backslash \{c\}$
    $p \leftarrow q, q \in P \wedge \{(c,q)\} \in M$             get its core
    $load(p) \leftarrow load(p) - load(c)$                       remove its load from its core
    $M' \leftarrow M' \backslash \{(c,p)\}$                      remove from mapping
    $p' \leftarrow q \mid q \in \arg\min_{r \in P} cost(c,r)$    find core with smallest cost
    $load(p') \leftarrow load(p') + load(c)$                     add chare load to new core
    $M' \leftarrow M' \cup \{(c,p')\}$                           map to new core
```

Agenda

~~NUMA~~

~~Our Load Balancer: NumaLB~~

Experimental Setup

Results

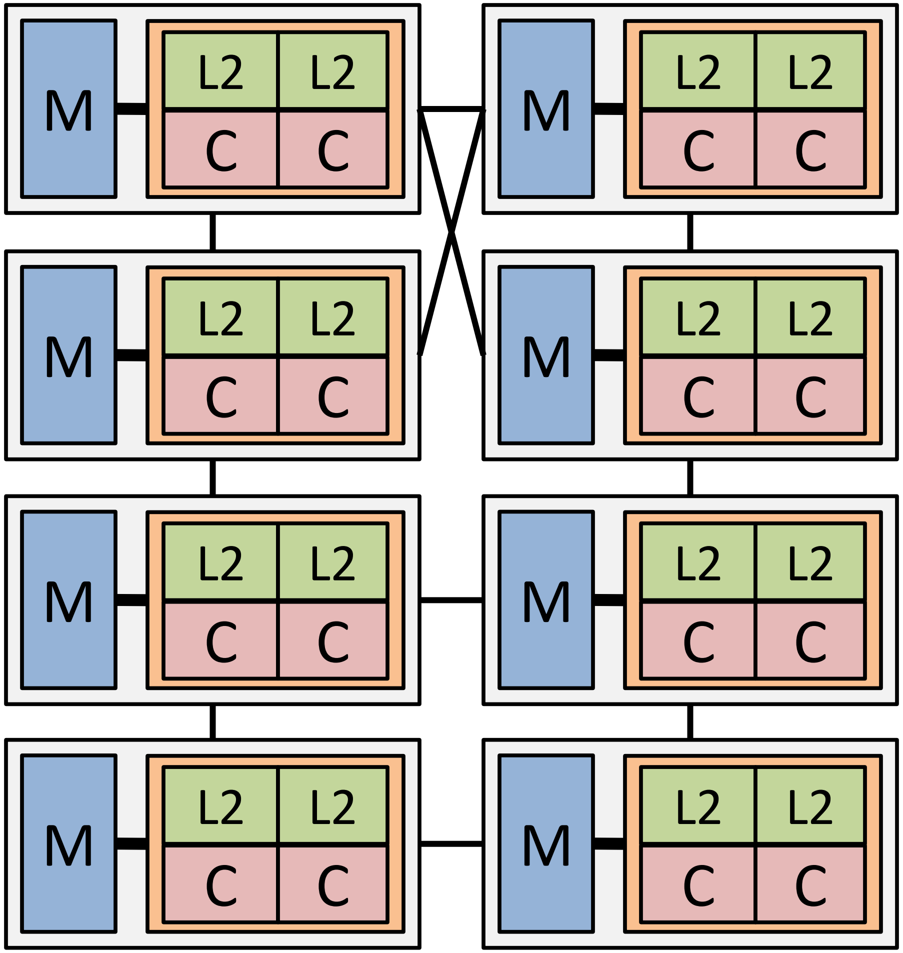
Concluding Remarks

Experimental Setup

- 2 NUMA machines
- 3 CHARM++ benchmarks
- 4 other CHARM++ load balancers
- Statistical confidence of 95%
  - 5% relative error
  - Student’s t-distribution
  - Minimum of 25 executions
- Performance
  - Gains: Average iteration time (baseline = no LB)
  - Costs: Load balancing overhead

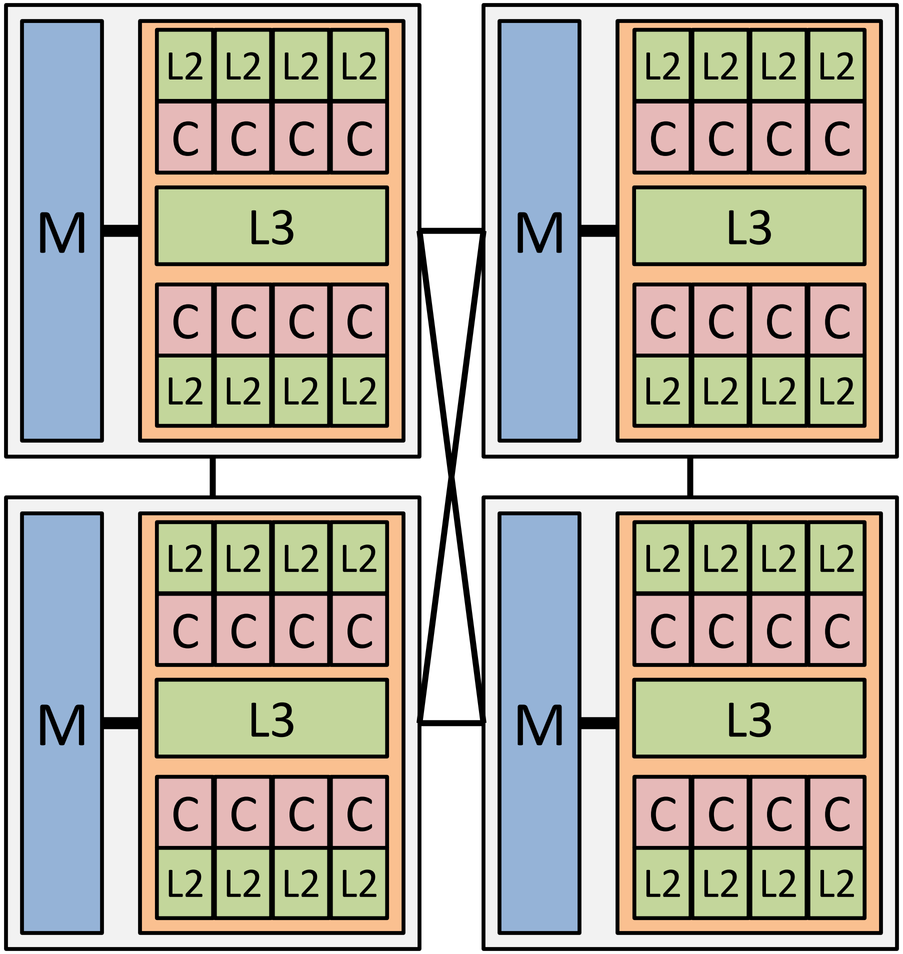
Experimental Setup: Machines

- NUMA16
  - AMD Opteron
  - 8×2 cores @ 2.2 GHz
  - 1 MB private L2 cache
  - 32 GB main memory
  - Low latency for memory access
  - Crossbar
  - NUMA factor: 1.1–1.5



Experimental Setup: Machines

- NUMA32
  - Intel Xeon X7560
  - 4×8 cores @ 2.27 GHz
  - 256 KB private L2
  - 24 MB shared L3
  - 64 GB main memory
  - QuickPath
  - NUMA factor: 1.36–3.6



Experimental Setup: Benchmarks

- kNeighbor
  - Synthetic iterative benchmark where a chare communicates with other *k* chares at each step
  - Completely I/O bound
  - 200 chares, 16 KB messages, *k* = 8
- lb_test
  - Synthetic unbalanced benchmark with different possible communication patterns
  - 200 chares, random communication graph, load between 50 and 200 ms
- jacobi2D
  - Unbalanced two-dimensional five-point stencil
  - 100 chares, 32^2 data array

Experimental Setup: LBs

- GreedyLB
  - Iteratively maps the most loaded chares to the least loaded cores
- RecBipartLB
  - Recursive bipartition of the communication graph
  - Breadth-first traversal until groups the required load
- MetisLB
  - Graph partitioning algorithms from METIS
- ScotchLB
  - Graph partitioning algorithms from SCOTCH
- Neither consider the current chare mapping

Agenda

~~NUMA~~

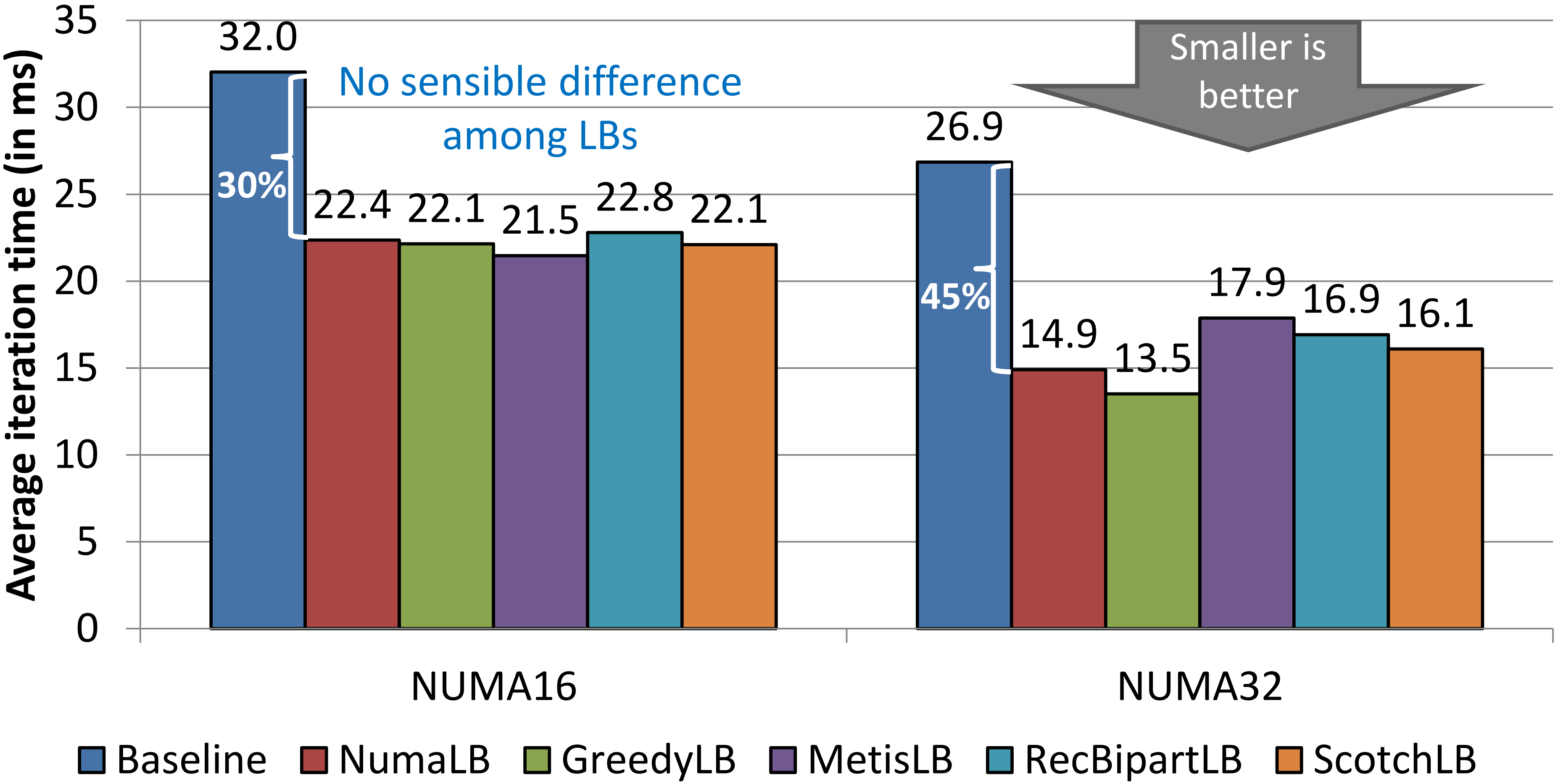
~~Our Load Balancer: NumaLB~~

~~Experimental Setup~~

Results

Concluding Remarks

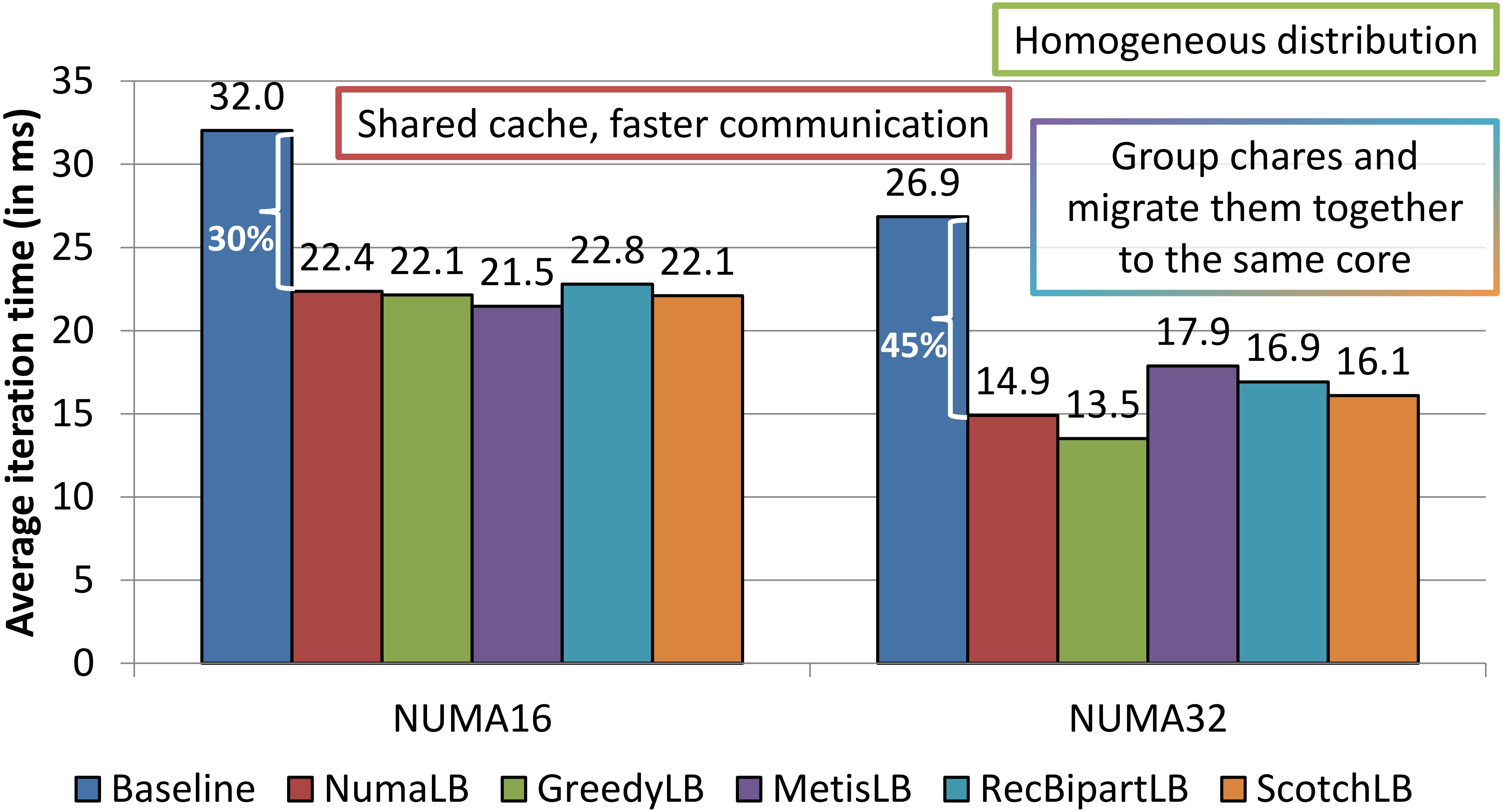
Results: kNeighbor



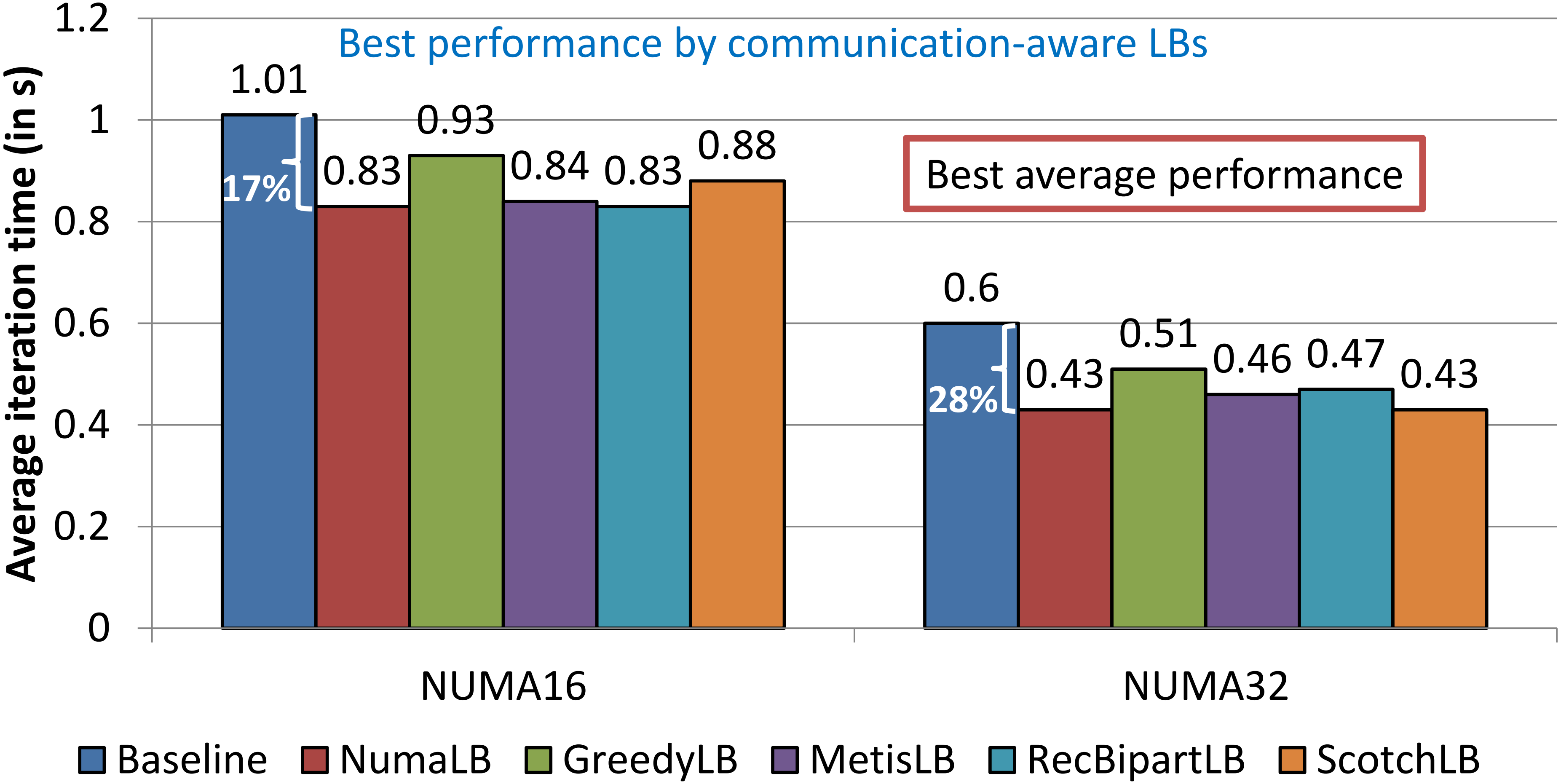
No sensible difference among LBs

Smaller is better

Results: kNeighbor



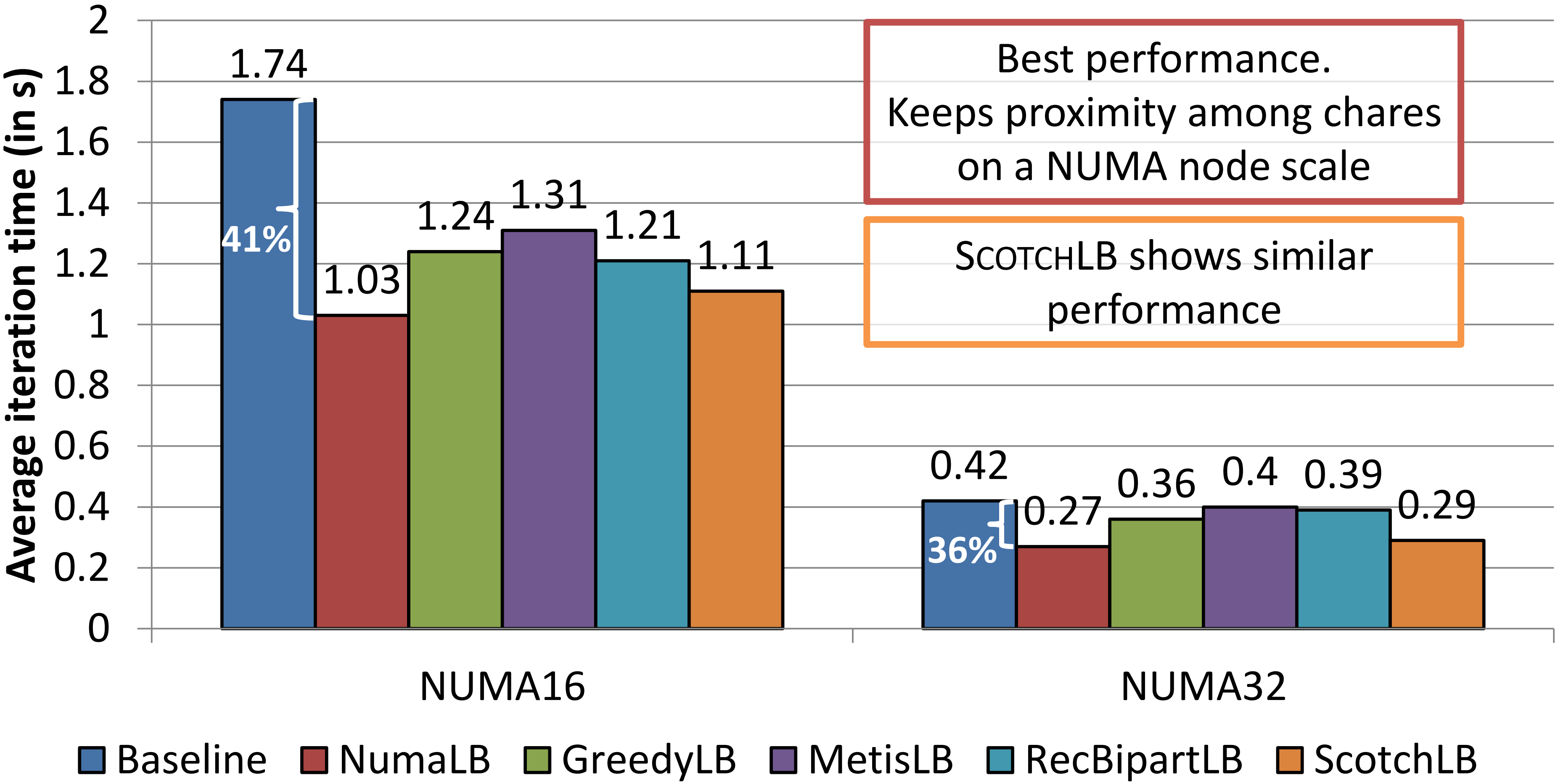
Results: lb_test



Best performance by communication-aware LBs

Best average performance

Results: jacobi2D



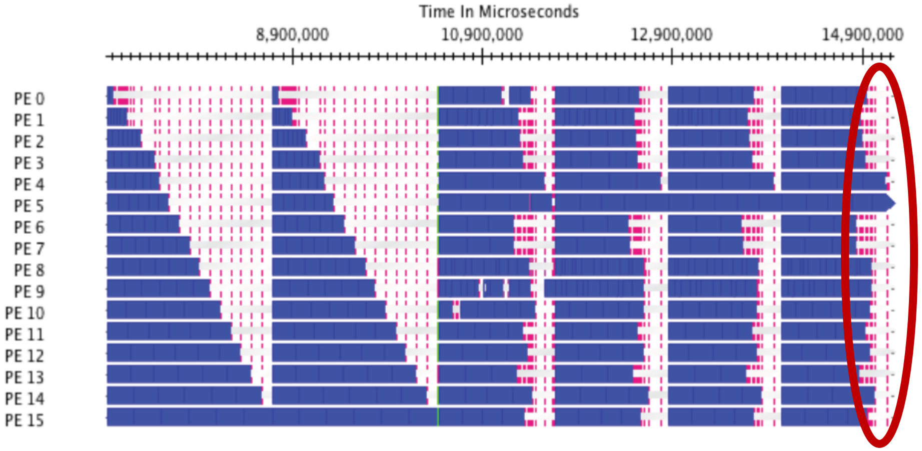
Best performance.
Keeps proximity among chares
on a NUMA node scale

ScotchLB shows similar
performance

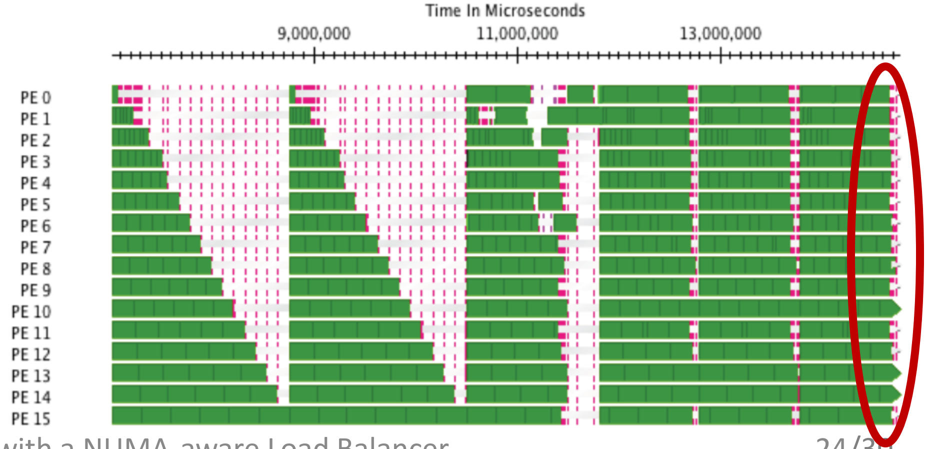
Results: jacobi2D - Projections

- jacobi2D on NUMA16
  - 2 steps before LB
  - 4 steps after LB
- The smaller the idle parts, the higher the efficency

METISLB: 75% efficiency



NUMALB: 93.5% efficiency



Results: overheads

Average number of chares migrated

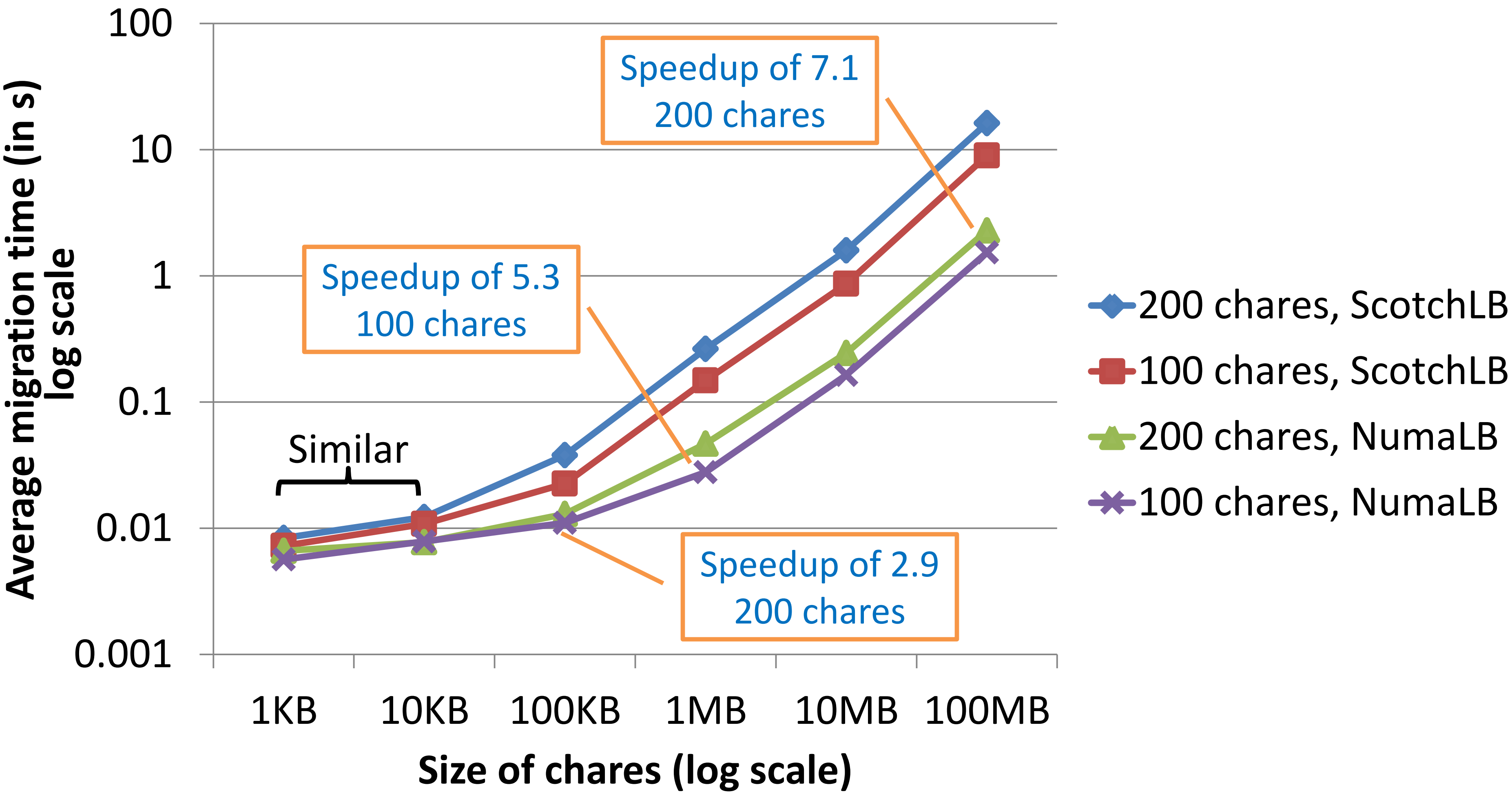
Benchmark	Machine	Load Balancer				
		NumaLB	GreedyLB	MetisLB	RecBipartLB	ScotchLB
kNeighbor	NUMA16	25	189	188	176	185
	NUMA32	57	194	195	185	194
lb_test	NUMA16	40	188	187	184	184
	NUMA32	48	194	194	192	192
jacobi2D	NUMA16	26	94	94	91	93
	NUMA32	33	97	96	93	98

Maximum migrations = 33% ☺

Minimum migrations = 88% ☹

All load balancers took less than 7 ms for their algorithms.

Results: migration times for NUMA16



Speedup of 7.1
200 chares

Speedup of 5.3
100 chares

Speedup of 2.9
200 chares

Agenda

~~NUMA~~

~~Our Load Balancer: NumaLB~~

~~Experimental Setup~~

~~Results~~

Concluding Remarks

Conclusions

- Multi-core machines with NUMA design introduce new challenges for their efficient use
- CHARM++ does not consider NUMA asymmetries
- With our NUMA-aware LB we obtained
  - An average speedup of 1.51 over the baseline
    - Transparent to the user, no previous knowledge
  - 10% improvement over most LBs
  - Migration overheads up to 7 times smaller
    - Migrating at most 33% of all chares

Future Work

- Multi-core load balancer
  - UMA and NUMA machines
  - Communication latencies among cores
  - Use HWLOC representation of cache hierarchy
- Distributed multi-core load balancer
  - For clusters of multi-core machines
- Gather and organize communication information
  - Latencies, bandwidth
  - Provide this data to other libraries (like SCOTCH)

9th Annual Workshop on CHARM++ and its Applications

Improving CHARM++ Performance with a NUMA-aware Load Balancer

Thank you.

Laércio Lima Pilla

Contact: llpilla@inf.ufrgs.br