Topology and affinity aware hierarchical and distributed load-balancing in Charm++

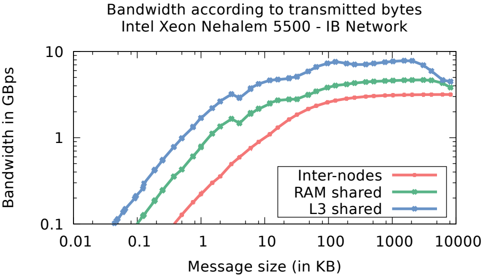
Emmanuel Jeannot, Guillaume Mercier, **François Tessier**

Inria - IPB - LaBRI - University of Bordeaux - Argonne National Lab.

November 18, 2016

Computing at Scale

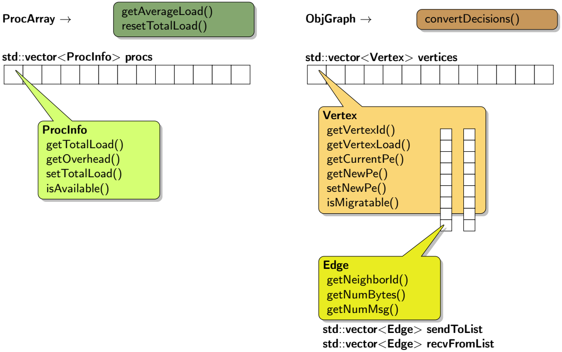
- Large-scale parallel simulations: climate, heart modelling, cosmology, etc.
- Increasing number of cores on supercomputers
  - The more parallelization, the bigger impact of load imbalance
  - Applications need to communicate even more
- Complex topologies: interconnection networks, memory hierarchy (NUMA effects)



What about combining CPU load-balancing and data locality to optimize performance of large-scale applications?

Charm++ and Load-balancing

- Parallel object-oriented programming language based on C++
- Fine-grained paradigm: cooperating objects called **chares**
- Plugable load balancing algorithms at launch time
- Load balancers able to natively **migrate** chares
- Adaptive runtime system supplying chares and cores statistics



ProcArray →

getAverageLoad()
resetTotalLoad()

std::vector<ProcInfo> procs

ProcInfo
getTotalLoad()
getOverhead()
setTotalLoad()
isAvailable()

ObjGraph →

convertDecisions()

std::vector<Vertex> vertices

Vertex
getVertexId()
getVertexLoad()
getCurrentPe()
getNewPe()
setNewPe()
isMigratable()

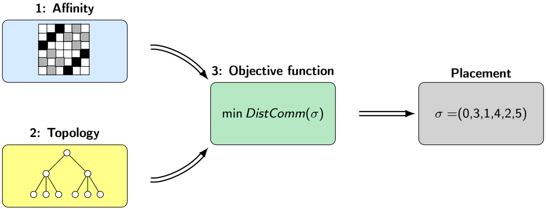
Edge
getNeighborId()
getNumBytes()
getNumMsg()

std::vector<Edge> sendToList
std::vector<Edge> recvFromList

Figure: Charm++ data structures

The TreeMatch Algorithm

- Algorithm and environment to compute processing entities placement based on their affinities and NUMA topology
- Requires tree topology, based on a qualitative approach
- **Input:**
  - The affinity pattern of the application
  - A model (tree) of the underlying architecture (qualitative approach)
- **Output:**
  - A processes permutation σ such that σ_i is the core number on which we have to bind the processing entity i
- **Goal:**
  - Minimize the sum of the communications between processes weighted by the number of hops (min $DistComm(\sigma)$)
- Combinatorial complexity with optimality to 128 processing entities then heuristic for larger input



Outline

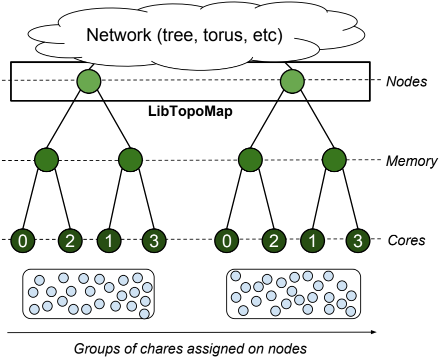
1. Context
2. Approach
3. Experimental Validation
4. Conclusion

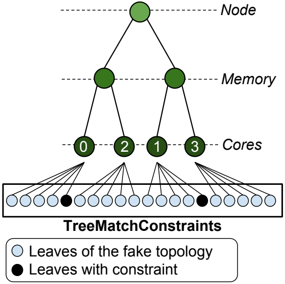
Outline

1. Context
2. **Approach**
3. Experimental Validation
4. Conclusion

TreeMatchLB

- Load balancing algorithm for communication-bound applications
  - Improve data locality dynamically (temporality)
  - CPU load-balancing based on refinement
- Hierarchical and distributed algorithm
  - Reorders groups of chares on nodes (LibTopoMap)
  - Reorders chares inside each node: TreeMatch with constraints
  - Each node in parallel



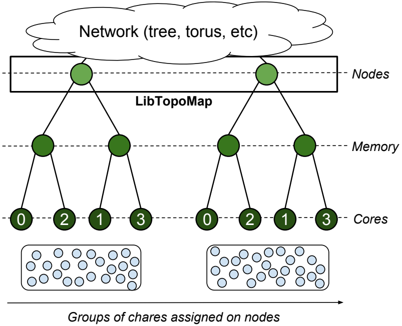


TreeMatchLB - Inter-node placement

- TreeMatch works only on tree topologies
- LibTopoMap: library able to place processes on any network topology

Example: 3D-Torus Cray Gemini network

- Algorithm steps
  1. Simple load refinement by swapping chares
  2. Convert the batch scheduler allocation to a readable format for LibTopoMap
  3. Apply network placement of groups of chares on nodes with LibTopoMap

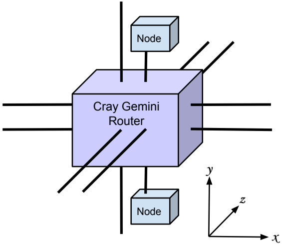


TreeMatchLB - Inter-node placement

- TreeMatch works only on tree topologies
- LibTopoMap: library able to place processes on any network topology

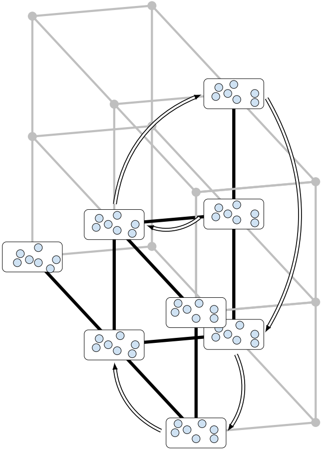
Example: 3D-Torus Cray Gemini network

- Algorithm steps
  1. Simple load refinement by swapping chares
  2. Convert the batch scheduler allocation to a readable format for LibTopoMap
  3. Apply network placement of groups of chares on nodes with LibTopoMap



TreeMatchLB - Inter-node placement

- TreeMatch works only on tree topologies
- LibTopoMap: library able to place processes on any network topology



Example: 3D-Torus Cray Gemini network

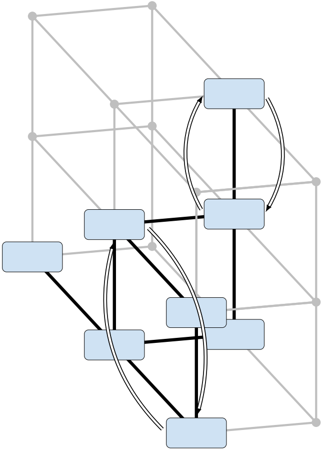
- Algorithm steps
  1. Simple load refinement by swapping chares
  2. Convert the batch scheduler allocation to a readable format for LibTopoMap
  3. Apply network placement of groups of chares on nodes with LibTopoMap

TreeMatchLB - Inter-node placement

- TreeMatch works only on tree topologies
- LibTopoMap: library able to place processes on any network topology

Example: 3D-Torus Cray Gemini network

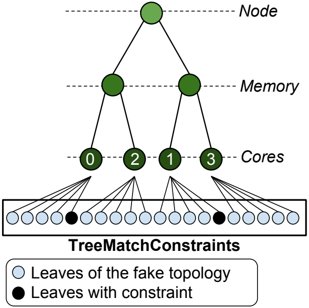
- Algorithm steps
  1. Simple load refinement by swapping chares
  2. Convert the batch scheduler allocation to a readable format for LibTopoMap
  3. Apply network placement of groups of chares on nodes with LibTopoMap



TreeMatchLB - Intra-node placement

- Intra-node placement based on the TreeMatch algorithm
  - Trade-off between CPU load and affinity
  - Oversubscribing (more chares than processing units)

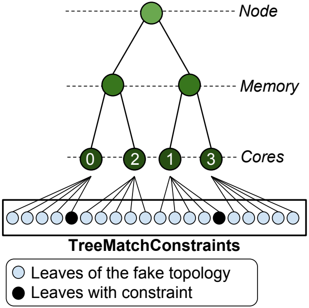
- Algorithm steps, for each node
  1. Extract each node communication pattern
  2. Extend the node topology
     - New level with $arity = \frac{\#chares}{\#proc}$
  3. Apply chares placement computed with TreeMatch
  4. Refinement: move chares to the least loaded core from cores as close as possible



TreeMatchLB - Intra-node placement

- Intra-node placement based on the TreeMatch algorithm
  - Trade-off between CPU load and affinity
  - Oversubscribing (more chares than processing units)

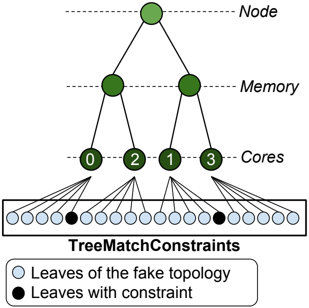
- Algorithm steps, for each node
  1. Extract each node communication pattern
  2. Extend the node topology
     - New level with $arity = \frac{\#chares}{\#proc}$
  3. Apply chares placement computed with TreeMatch
  4. Refinement: move chares to the least loaded core from cores as close as possible



TreeMatchLB - Intra-node placement

- Intra-node placement based on the TreeMatch algorithm
  - Trade-off between CPU load and affinity
  - Oversubscribing (more chares than processing units)

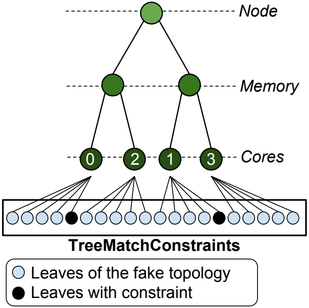
- Algorithm steps, for each node
  1. Extract each node communication pattern
  2. Extend the node topology
     - New level with $arity = \frac{\#chares}{\#proc}$
  3. Apply chares placement computed with TreeMatch
  4. Refinement: move chares to the least loaded core from cores as close as possible



TreeMatchLB - Intra-node placement

- Intra-node placement based on the TreeMatch algorithm
  - Trade-off between CPU load and affinity
  - Oversubscribing (more chares than processing units)

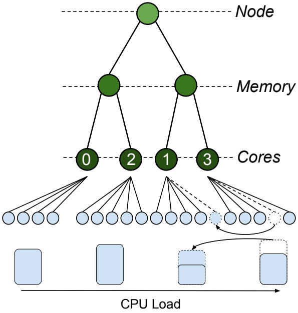
- Algorithm steps, for each node
  1. Extract each node communication pattern
  2. Extend the node topology
     - New level with *arity* $= \frac{\#chares}{\#proc}$
  3. Apply chares placement computed with TreeMatch
  4. Refinement: move chares to the least loaded core from cores as close as possible



TreeMatchLB - Intra-node placement

- Intra-node placement based on the TreeMatch algorithm
  - Trade-off between CPU load and affinity
  - Oversubscribing (more chares than processing units)

- Algorithm steps, for each node
  1. Extract each node communication pattern
  2. Extend the node topology
     - New level with *arity* $= \frac{\#chares}{\#proc}$
  3. Apply chares placement computed with TreeMatch
  4. Refinement: move chares to the least loaded core from cores as close as possible



TreeMatchLB - Intra-node placement

- Parallelized and distributed version of TreeMatchLB based on a master-worker scheme with two levels of parallelization
  - OpenMP to extract the communication pattern of each node and distribute the work
  - The Charm++ mechanisms for distribution

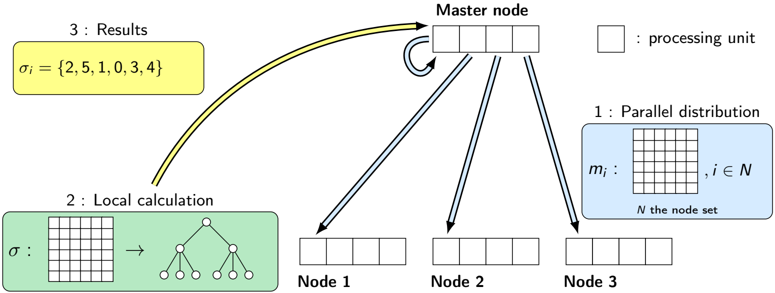


Figure: Master-worker scheme used in our topology-aware load-balancer

Outline

1. Context
2. Approach
3. Experimental Validation
4. Conclusion

TreeMatchLB - Experiments

Experimental Conditions

- Two different architectures

PlaFRIM

- Intel Xeon Nehalem X5550 (2.66 GHz, 8 cores / node)
- 24 GB of 1.33GHz DDR3 RAM / node
- 8 MB of L3 cache / 4 cores
- Infiniband fat-tree network

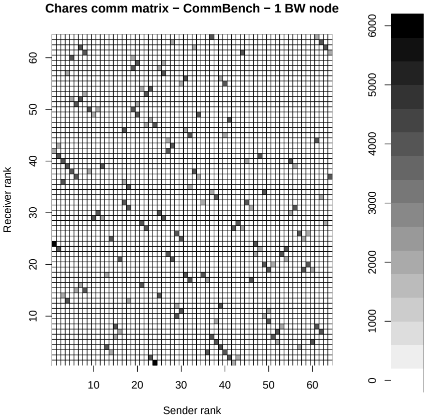
Blue Waters

- Peak perf.: 13.34 Petaflops
- Cray XE6 nodes: AMD 6276 Interlagos processors (32 cores / node)
- 64 GB of main memory / node
- Cray Gemini 3D-Torus

- Benchmarks and applications
  - **CommBench**: benchmark simulating irregular communications
  - **ChaNGa**: large-scale cosmological simulation
  - Ondes3D: simulator of three-dimensional seismical wave propagation

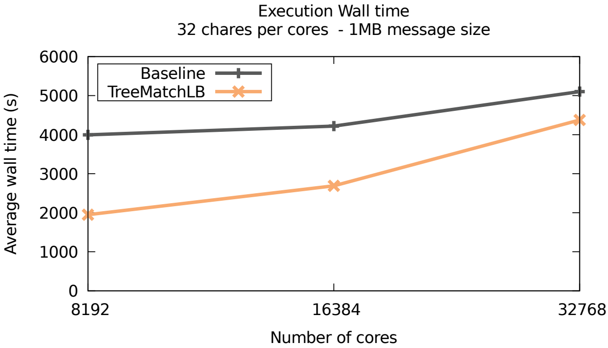
TreeMatchLB - commBench

- Benchmark simulating irregular communications
- Scalability: 32K cores (256 XE6 nodes) on Blue Waters
- Up to 1M chares, i.e. 32 chares/core
- Native Charm++ load balancers do not work at such scale
- 16.6% of improvement compared to baseline on the largest case



TreeMatchLB - commBench

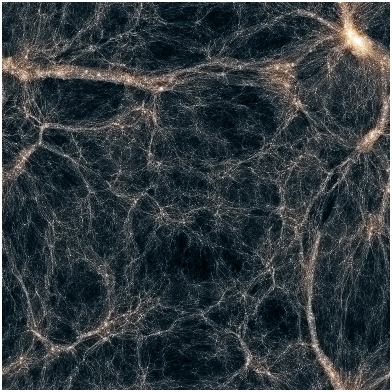
- Benchmark simulating irregular communications
- Scalability: 32K cores (256 XE6 nodes) on Blue Waters
- Up to 1M chares, i.e. 32 chares/core
- Native Charm++ load balancers do not work at such scale
- 16.6% of improvement compared to baseline on the largest case



TreeMatchLB - ChaNGa

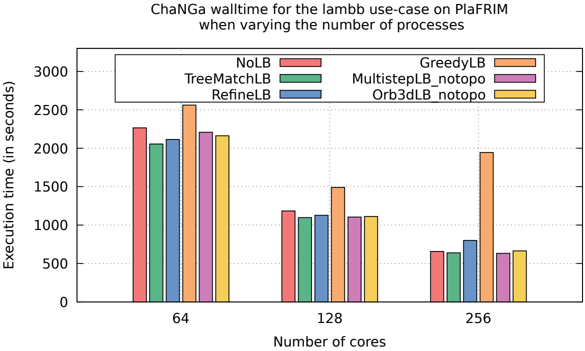
- Large-scale cosmological application designed to perform collisionless N-body simulation
- Lambb use case designed for up to 1024 cores: 80M particles represented a 70 Mpc^3 (Megaparsec) volume. Computes the mass function of dark matter halos.

- TreeMatchLB compared to different load balancers
  - **RefineLB**: migrates chares from overloaded cores to underloaded cores to reach an average load (few migrations)
  - **GreedyLB**: re-assign all the chares by mapping the highest loaded chare to the least loaded core
  - **MultistepLB**: load balancing based on predictions made from previous timesteps
  - **Orb3dLB**: recursive bisection to find a balanced state



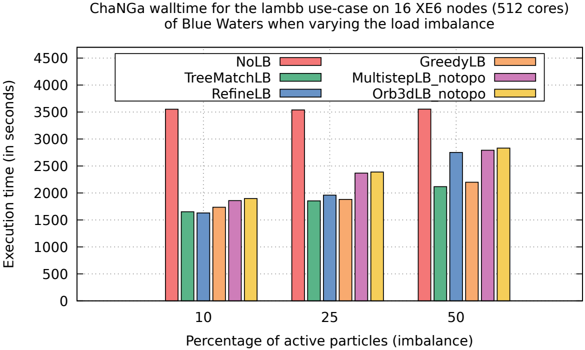
TreeMatchLB - ChaNGa

- **PlaFRIM**: 8, 16 and 32 nodes
  - 64, 128 and 256 cores
  - 512, 1024 and 2048 chares
- TreeMatchLB is better or on par with other strategies
- 600ms to compute the new chares placement while less than 50ms for the other methods
- Benefits in term of performance counterbalance this additional cost



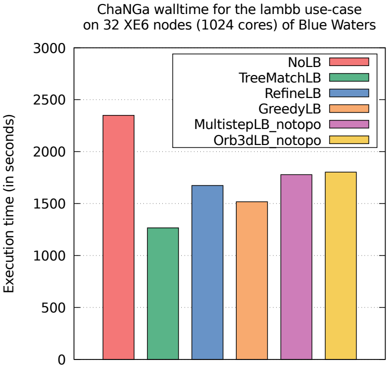
TreeMatchLB - ChaNGa

- **Blue Waters**: 16 and 32 nodes
  - 512 and 1024 cores
  - 4096 and 8192 chares
- On 16 nodes, variation of the percentage of active particles (change load imbalance)
- Equalize at worst the performance obtained with the best solution
- On 32 nodes, TreeMatchLB outperforms GreedyLB by 17%



TreeMatchLB - ChaNGa

- **Blue Waters**: 16 and 32 nodes
    - 512 and 1024 cores
    - 4096 and 8192 chares
- On 16 nodes, variation of the percentage of active particles (change load imbalance)
- Equalize at worst the performance obtained with the best solution
- On 32 nodes, TreeMatchLB outperforms GreedyLB by 17%



Outline

1. Context
2. Approach
3. Experimental Validation
4. Conclusion

Conclusion

- Load-balancing algorithm taking into account the data locality
    - Application independent (communication-bound applications)
    - Based on LibTopoMap (inter-node) and TreeMatch (intra-node)
    - Distributed and hierarchical
- Outperforms by 17% the native load balancers on 32 Blue Waters nodes and a real application
- Scales up to 1M processing entities

Future work

- Evaluate the impact of the routing policy
- Adaptive hierarchical approach: let TreeMatchLB chose the two levels of hierarchy to balance

Acknowledgments

- JLESC for allocations on Blue Waters
- PPL for the support

Conclusion

Thank you for your attention!

ftessier@anl.gov

TreeMatchLB - Behavior faced with the initial placement

What is the sensitivity of TreeMatchLB to initial placement?

- Application (kNeighbor) for which the optimal placement is known
- Testbed: Intel Xeon Nehalem X5550 (8 cores)
  - Physical core numbering: 0, 2, 4, 6, 1, 3, 5, 7
- TreeMatchLB VS optimal placement VS default placement
  - The initial mapping may vary according to the core numbering
- No sensitivity of TreeMatchLB to initial placement
- Converge to the optimal placement