Distributed communication-aware load balancing with TreeMatch in Charm++

The 9th Scheduling for Large Scale Systems Workshop, Lyon, France

Emmanuel Jeannot    Guillaume Mercier    Francois Tessier

*In collaboration with the Charm++ Team from the PPL (UIUC, IL):*
Esteban Meneses-Rojas, Gengbin Zheng, Sanjay Kale

July 1, 2014
Scalable execution of parallel applications

- Number of cores is increasing
- But memory per core is decreasing
- Application will need to communicate even more than now

Our solution

- Process placement should take into account process affinity
- Here: load balancing in Charm++ considering:
  - CPU load
  - process affinity (or other communicating objects)
  - topology: network and intra-node
Charm++

Features

- Parallel object-oriented programming language based on C++
- Programs are decomposed into a number of cooperating message-driven objects called **chares**.
- In general we have more chares than processing units
- Chares are mapped to physical processors by an adaptive runtime system
- Load balancers can be called to **migrate** chares
- Charm++ is able to use MPI for the processes communications

Francois Tessier

TreeMatch in Charm++
Processes Placement

Why we should consider it

- Many current and future parallel platforms have several levels of hierarchy
- Application chares/processes do not exchange the same amount of data (affinity)
- The process placement policy may have impact on performance
  - Cache hierarchy, memory bus, high-performance network...

![Diagram of a tree-like structure with levels of hierarchy from Switch to Core.]
Problems

**Given**

- The parallel machine topology
- The application communication pattern

- Map application processes to physical resources (cores) to reduce the communication costs (NP-complete)
The TreeMatch Algorithm

- Algorithm and environment to compute processes placement based on processes affinities and NUMA topology

- **Input:**
  - The communication pattern of the application
  - Preliminary execution with a monitored MPI implementation for static placement
  - Dynamic recovery on iterative applications with Charm++
  - A model (tree) of the underlying architecture: Hwloc can provide us this.

- **Output:**
  - A processes permutation $\sigma$ such that $\sigma_i$ is the core number on which we have to bind the process $i$

TreeMatch can only work on tree topologies. How to deal with 3d torus?
Network placement

libtopomap

- Library that enables to map processes on various network topologies
- Used in TreeMatchLB to consider the Blue Waters 3d torus

*Figure:* 3d Torus and a Cray Gemini router
Load balancing

Principle

- Iterative applications
- Load balancer called at regular interval
- Migrate chares in order to optimize several criteria
- Charm++ runtime system provides:
  - Chares load
  - Chares affinity
  - Etc.

Constraints

- Dealing with complex modern architectures
- Taking into account communications between elements

Some other communication-aware load-balancing algorithms

- Some "built-in" Charm++ load balancers: RefineCommLB, GreedyCommLB...
Several issues raised

- Several issues raised!
- Scalability of TreeMatch
- How to deal with process mapping (user, core numbering)
  - Intel Xeon 5550 : 0,2,4,6,1,3,5,7
  - Intel Xeon 5550 : 0,1,2,3,4,5,6,7 (!!)
  - AMD Interlagos : 0,1,2,3,4,5,6,7...,30,31
- Need to find a relevant compromise between processes affinities and load balancing
- What about load balancing time?

The next slides will present our load balancer relying on TreeMatch and libtopomap which performs a parallel and distributed communication-aware load balancing.
First step: minimize communication cost on network

- libtopomap reorders processes from a communicator
- How to use it to reorder groups of processes (or chares)? Example: groups of chares on nodes
  - Charm++ uses MPI: full access to the MPI API
  - New MPI communicator with MPI_Comm_split
**TreeMatch load balancer**

- 1\(^{st}\) step: Remap groups of chares on nodes according to the communication on the network
  - libtopomap (example: part of 3d Torus)
- 2\(^{nd}\) step: Reorder chares inside each node (distributed)
  - Apply TreeMatch on the NUMA topology and the chares communication pattern
  - Bind chares according to their load (leveling on less loaded chares)
  - Each node carries out its own placement in parallel

```
CPU Load
3 6 8 9 12 14 15 16
```

Network (3d torus, hierarchical, …)
Strategy for Charm++ - Intra-node placement

TreeMatch load balancer

- 1st step: Remap groups of chares on nodes according to the communication on the network
  - libtopomap (example: part of 3d Torus)
- 2nd step: Reorder chares inside each node (distributed)
  - Apply TreeMatch on the NUMA topology and the chares communication pattern
  - Bind chares according to their load (leveling on less loaded chares)
  - Each node carries out its own placement in parallel

Figure: Part of a 3d Torus attributed by the resource manager
Strategy for Charm++ - Intra-node placement

TreeMatch load balancer

1\textsuperscript{st} step: Remap groups of chares on nodes according to the communication on the network
- libtopomap (example: part of 3d Torus)

2\textsuperscript{nd} step: Reorder chares inside each node (distributed)
- Apply TreeMatch on the NUMA topology and the chares communication pattern
- Bind chares according to their load (leveling on less loaded chares)
- Each node carries out its own placement in parallel

Network (3d torus, hierarchical, …)

Groups of chares assigned to cores

CPU Load
Strategy for Charm++ - Intra-node placement

**TreeMatch load balancer**

- **1\textsuperscript{st} step**: Remap groups of chares on nodes according to the communication on the network
  - libtopomap (example: part of 3d Torus)
- **2\textsuperscript{nd} step**: Reorder chares inside each node (distributed)
  - Apply TreeMatch on the NUMA topology and the chares communication pattern
  - Bind chares according to their load (leveling on less loaded chares)
  - Each node carries out its own placement in parallel
TreeMatch load balancer

1\textsuperscript{st} step: Remap groups of chares on nodes according to the communication on the network
- libtopomap (example: part of 3d Torus)

2\textsuperscript{nd} step: Reorder chares inside each node (distributed)
- Apply TreeMatch on the NUMA topology and the chares communication pattern
- Bind chares according to their load (leveling on less loaded chares)
- Each node carries out its own placement in parallel
Strategy for Charm++ - Intra-node placement

**TreeMatch load balancer**

- 1\textsuperscript{st} step: Remap groups of chares on nodes according to the communication on the network
  - libtopomap (example: part of 3d Torus)
- 2\textsuperscript{nd} step: Reorder chares inside each node (distributed)
  - Apply TreeMatch on the NUMA topology and the chares communication pattern
  - Bind chares according to their load (leveling on less loaded chares)
  - Each node carries out its own placement in parallel
Results

**commBench**

- Benchmark designed to simulate irregular communications
- Experiments on 16 nodes with 32 cores on each (AMD Interlagos 6276) - Blue Waters Cluster
- 1 MB messages - 100 iterations - 2 distant receivers for each chare

![Bar chart showing comparison of DummyLB, RefineCommLB, and TreeMatchLB]

commBench on 512 cores
8192 elements – 1MB message size

Average time of one iteration in ms
Results

**commBench**
- 1 MB messages - 100 iterations - 2 distant receivers for each chare
- TreeMatch applied on a chares communication matrix

![Chares comm matrix – CommBench – 1 PlaFRIM node](image)

**Figure:** $\sigma(i) = 0, 8, 4, 5, 12, 1, 9, 6, 14, 2, 3, 13, 7, 10, 11, 15$
Results

**kNeighbor**

- Benchmarks application designed to simulate regular intensive communication between processes
- Experiments on 8 nodes with 8 cores on each (Intel Xeon 5550) - PlaFRIM Cluster
- Particularly compared to RefineCommLB
  - Takes into account load and communication
  - Minimizes migrations

![Graph 1](image1)

**Graph 1:**
- kNeighbor on 64 cores
- 128 elements – 1MB message size

![Graph 2](image2)

**Graph 2:**
- kNeighbor on 64 cores
- 256 elements – 1MB message size
**kNeighbor**

- Experiments on 16 nodes with 8 cores on each (Intel Xeon 5550) - PlaFRIM Cluster
- 1 MB messages - 100 iterations - 7-Neighbor

**Execution time versus shares by core**

![Graph showing execution time versus shares by core](image)

**Graph Details**

- **Y-axis**: Average time for each 7-kNeighbor iteration (in ms)
- **X-axis**: Number of shares by core

**Legend**

- DummyLB 0-7
- TreeMatchLB
- DummyLB 0,2,4,6,1,3,5,7
What about the load balancing time?

- Comparison between the sequential and the distributed versions of TreeMatchLB
- The master node distributes the data to each node which will compute its own shares placement. This data distribution can be done in parallel (around 20% of improvements)
Results

What about the load balancing time?

- Comparison between the sequential and the distributed versions of TreeMatchLB
- The master node distributes the data to each node which will compute its own shares placement. This data distribution can be done in parallel (around 20% of improvements)
Results

What about the load balancing time?

- Linear trajectory while the number of chares is doubled
- TreeMatchLB is slower than the other Greedy strategies
- RefineCommLB which provides some good results for communication-bound applications is not scalable (fails from 8192 chares)

---

**Figure:** Load balancing time of the different strategies vs. number of chares for the KNeighbor application.
Future work and Conclusion

The end

- Topology is not flat!
- Processes affinities are not homogeneous
- Take into account these information to map chares give us improvement
- Algorithm adapted to large problems (Distributed)
- Published at IEEE Cluster 2013

Future work

- Find a better way to gather the topology (Hwloc?)
- Improve network part (BGQ routing?)
- Perform more large scale experiments
- Evaluate our solution on other applications (CFD?)
Thanks for your attention!
Any questions?