Selected Topics in Dynamic Load Balancing

Ronak Buch, Kavitha Chandrasekar, Juan Galvez, Hao Jin, Harshitha Menon, Michael Robson

rabuch2@illinois.edu

April 17, 2017

15th Annual Workshop on Charm++ and its Applications

Parallel Programming Laboratory, University of Illinois at Urbana-Champaign

Table of Contents

Introduction

Background

Load balancing is a key feature of Charm++, integral to achieving scalability and performance.

Charm++ load balancing has existed for a long time, but we are always looking to improve.

Background

Load balancing is a key feature of Charm++, integral to achieving scalability and performance.

Charm++ load balancing has existed for a long time, but we are always looking to improve.

New algorithms to improve performance.

Background

Load balancing is a key feature of Charm++, integral to achieving scalability and performance.

Charm++ load balancing has existed for a long time, but we are always looking to improve.

New algorithms to improve performance.

New approaches to tackle new hardware.

Overview

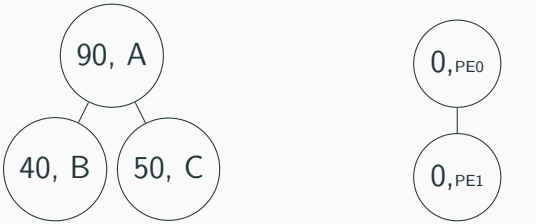
New features have been added to the load balancing infrastructure of Charm++, including:

- Improvements to greedy algorithm
- Topology aware load balancing
- Heterogeneous load balancing
- Load balancing library

Greedy Refine

Greedy Background

Basic `GreedyLB` in Charm++ places objects in max heap and PEs in min heap. Continuously pops heaviest object and maps to least loaded PE until object heap is empty.

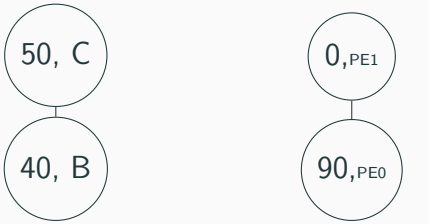


(a) Obj Heap **(b)** PE Heap

Figure: Step 0

Greedy Background

Basic `GreedyLB` in Charm++ places objects in max heap and PEs in min heap. Continuously pops heaviest object and maps to least loaded PE until object heap is empty.

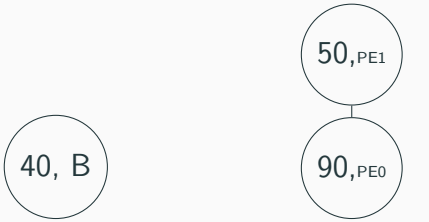


(a) Obj Heap **(b)** PE Heap

Figure: Step 1

Greedy Background

Basic `GreedyLB` in Charm++ places objects in max heap and PEs in min heap. Continuously pops heaviest object and maps to least loaded PE until object heap is empty.



(a) Obj Heap **(b)** PE Heap

Figure: Step 2

Greedy Background

Basic `GreedyLB` in Charm++ places objects in max heap and PEs in min heap. Continuously pops heaviest object and maps to least loaded PE until object heap is empty.

(a) Obj Heap

(b) PE Heap

Figure: Step 3

Greedy Issues

The greedy algorithm generally results in a good distribution of load.

However, it does not consider locality.

Often, this results in extremely high communication cost, with most objects in the system migrating at every load balancing step.

Greedy Issues

The greedy algorithm generally results in a good distribution of load.

However, it does not consider locality.

Often, this results in extremely high communication cost, with most objects in the system migrating at every load balancing step.

Can we do better?

Greedy Issues

The greedy algorithm generally results in a good distribution of load.

However, it does not consider locality.

Often, this results in extremely high communication cost, with most objects in the system migrating at every load balancing step.

Can we do better?
Yes, by modifying `GreedyLB` to limit the number of migrations.

Greedy Refine

New load balancer, `GreedyRefineLB`. Same as `GreedyLB`, but in each step decide whether to assign object to least loaded PE, or keep on same PE.

For some values α and β:

llp: least loaded processor
prevPe: previous PE of *obj*
M: $\alpha \cdot$ maximum load of `GreedyLB` solution

Stay on *prevPe* if ($prevPe.load \leq llp.load * \beta$) and ($prevPe.load + obj.load \leq M$)
Else move to *llp*

Greedy Refine Results

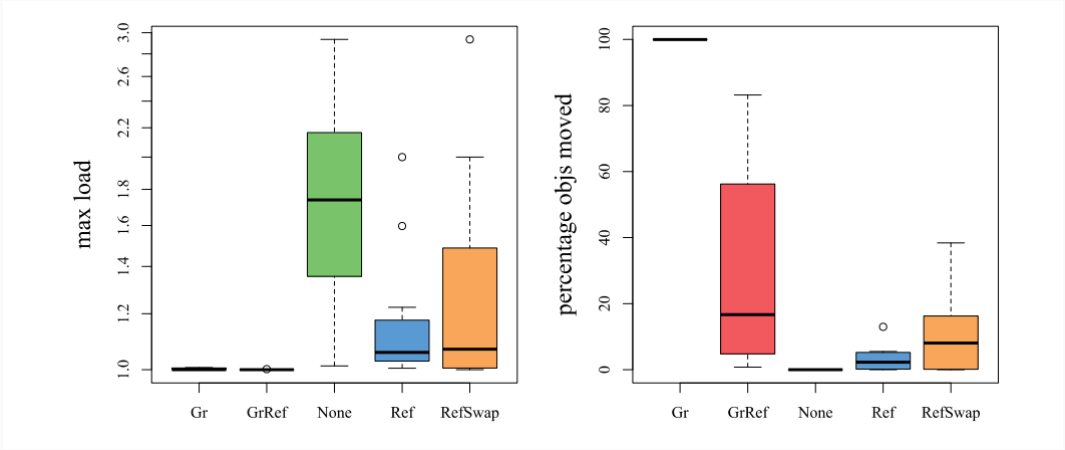


Figure: Load Balancing Results α_0, β_0

Greedy Refine Results (con't)

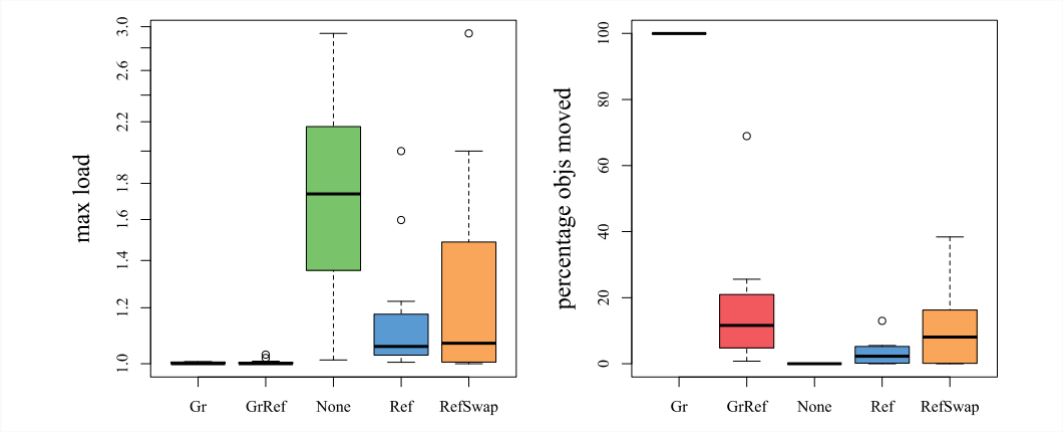


Figure: Load Balancing Results α_1, β_1

Greedy Refine Results (con't)

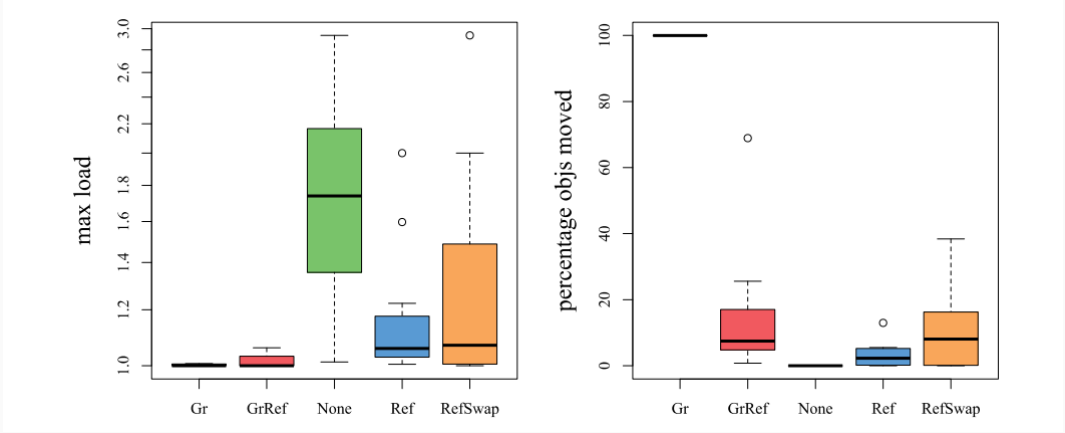


Figure: Load Balancing Results α_2, β_2

Greedy Refine Implementation

How do we pick α and β?

Finding good values is difficult, so, instead, we try multiple values and select the best result.

Charm++ had no way to compute and compare the results of multiple candidate load balancing solutions in parallel, so we implemented a new *concurrent mode*. To enable, set `concurrent` flag to `true` and then call `ApplyDecision()` on the PE of the best solution.

For example, `GreedyRefineLB` computes 225 different candidate solutions and chooses the best one (fewest migrations with max load under a threshold).

Topology Aware Load Balancing

Topology Aware Mapping

Topology aware mapping has been used to great effect in the past to improve the performance of MPI and Charm++ programs.

Reducing the distance messages must travel reduces the number of network hops and the likelihood of contention.

Rather than mapping ranks or PEs based on topology, we can dynamically map objects based on topology, load, and communication.

More sophisticated than static mapping, can cope with changing communication patterns and load.

GTopoCommLB

GTopoCommLB is a load balancer that combines load information with communication and topology information.

Performs multi-objective optimization, tries to *minimize hopbytes* while satisfying a *maximum load tolerance constraint*.

Optimal load tolerance is difficult to ascertain. Too low, and it can produce an infeasible problem or harm hopbytes, but too high, and it can allow load to be poorly balanced.

$\Rightarrow$ Use *concurrent mode* to run multiple instances with different maximum load tolerances.

GTopoCommLB **Results**

For Stencil3d on Blue Waters, 1024 nodes, 16k processors, 64k objects. Instrumentation enabled at iteration 10, one LB step at iteration 15.

GTopoCommLB maximum load 2% higher than GreedyLB.

GTopoCommLB hopbytes 7x smaller than GreedyLB.

GTopoCommLB performance 49% higher than DummyLB. (GreedyLB performs very badly due to communication performance)

Heterogeneous Load Balancing

Heterogeneous Loads

With the advent of GPUs and other accelerator devices to HPC, standard CPU load balancing is no longer sufficient to distribute work.

There are many challenges:

- Different hardware may have different performance for the same work.
- Data movement overhead may outweigh hardware performance gains.
- The execution model varies depending on the hardware target.

Some programs may have CPU only and GPU only work, while others may have work that can execute on any device, and yet others may have both sorts.

Vector Load Balancing

First, we consider programs with CPU only and GPU only work.

To load balance, we use the Charm++ runtime to measure load data for each hardware target. Thus, an object has a two-dimensional load vector, $< cpuload, gpuload >$.

The work should be distributed such that both dimensions are minimized.

Currently, we are using greedy strategies to place objects. We are in the process of exploring other algorithms.

Synthetic Vector Load Balancing Results

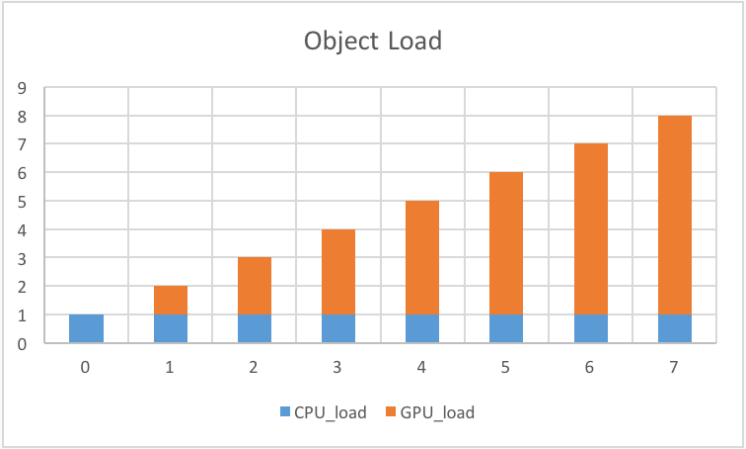


Figure: Object Loads

Synthetic Vector Load Balancing Results (con't)

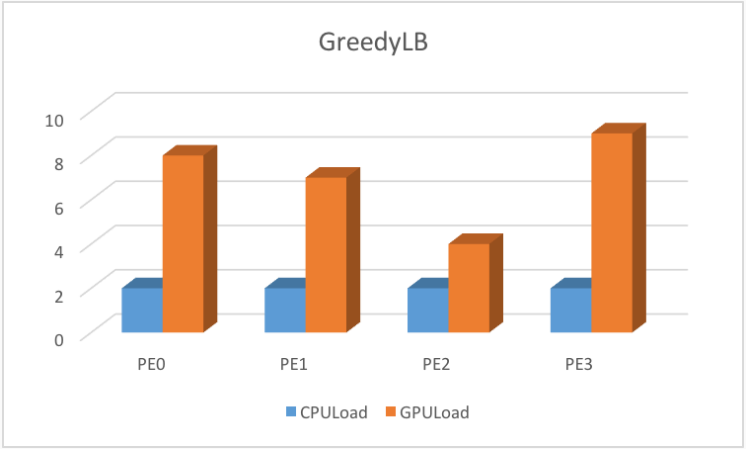


Figure: Regular `GreedyLB` Results

Synthetic Vector Load Balancing Results (con't)

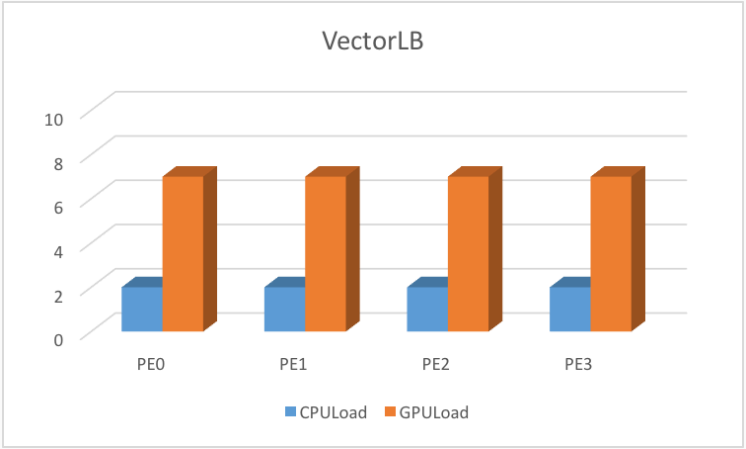


Figure: `VectorLB` Results

Production Vector Load Balancing Results

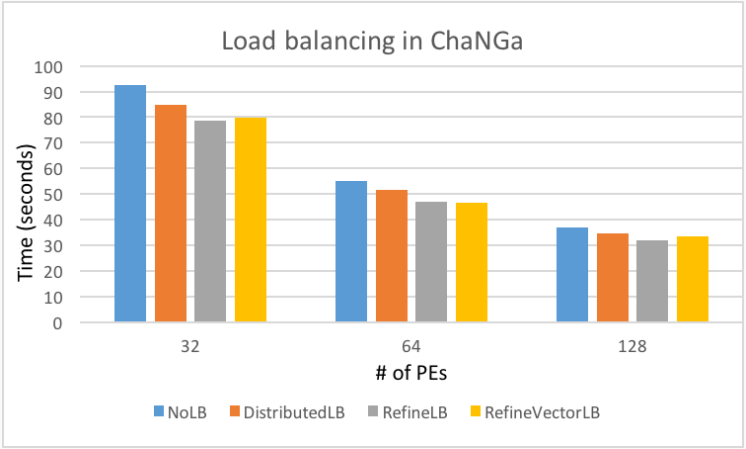


Figure: ChaNGa Load Balancing Results

Accel Load Balancing

Some programs have work that can be retargeted to different hardware devices.

This is a different problem than before since the static division of labor no longer exists.

Now, we must find the optimal "split" point to equally divide the work between the CPU and GPU.

- Different workloads react in different ways to different hardware.
- Unknown *a priori*.

To do this, the runtime dynamically assigns work to different hardware targets based on some strategy. The user can specify static assignment strategies or dynamic strategies where the runtime searches for the optimal split.

Accel Load Balancing Results

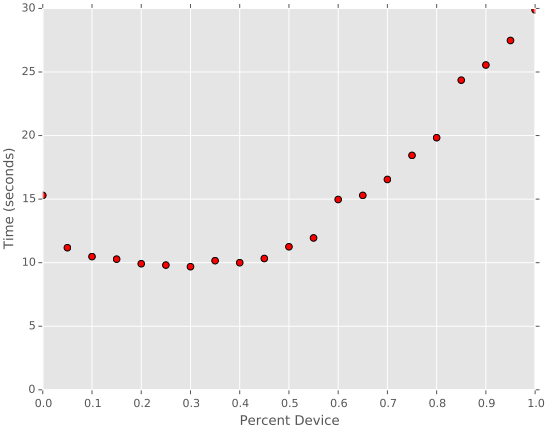


Figure: Accel Stencil3d Load Balancing Results

Accel Load Balancing Results (con't)

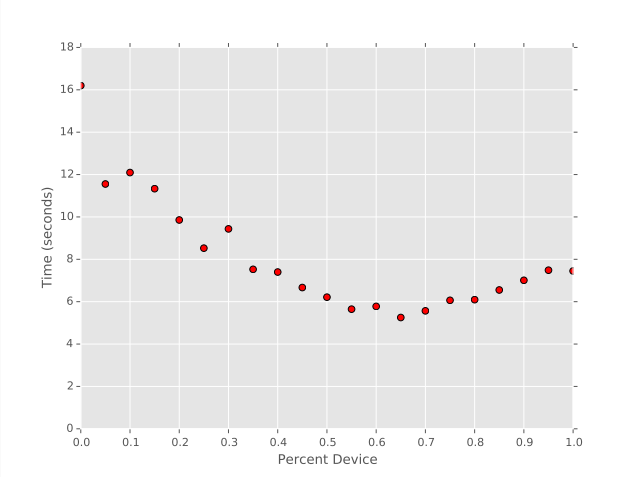


Figure: Accel MD Load Balancing Results

Future Work

We plan on combining the vector load balancing and accel load balancing strategies in the future, providing a more holistic response capability to shifting load.

Currently, data movement costs are not considered. We plan on adding cognizance of whether the data is located on the CPU or GPU.

Additional constraints will be added to the vector load balancer, such as a maximum memory threshold per PE or even application specific parameters.

Load Balancing Library

Load Balancing Library

We have decoupled load balancing strategies from Charm++ and released them as a library at
`https://charm.cs.illinois.edu/gerrit/gitweb?p=MPI_LB_Library.git`

It can easily be linked into an MPI program and be used to make object placement decisions.

The library includes a feature to automatically measure computation and communication for MPI programs. These data can also be manually specified.

No mechanism for migration is included; the end user is responsible for that.

Conclusion

Summary

- `GreedyRefineLB` provides results close to `GreedyLB` with a tunable number of migrations.
- `GTopoCommLB` provides an application of topology aware mapping to load balancing, reducing communication contention.
- Various new strategies for heterogeneous load balance were introduced, both for static and dynamic heterogeneity.
- Charm++'s load balancing strategies are now available independently of the runtime system for use with MPI and other applications.

Questions?