Recent Developments in Dynamic Load Balancing

Ronak Buch, Kavitha Chandrasekar, Juan Galvez, Eric Mikida

rabuch2@illinois.edu

April 11, 2018

16th 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

Background

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

Improve performance of computation:

- Minimize maximum load of compute resources
- Improve resource utilization

Background

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

Improve performance of computation:

- Minimize maximum load of compute resources
- Improve resource utilization

Improve performance of communication:

- Place communicating objects near each other
- Minimize network load

Background

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

Improve performance of computation:

- Minimize maximum load of compute resources
- Improve resource utilization

Improve performance of communication:

- Place communicating objects near each other
- Minimize network load

$\Rightarrow$ **Improve application execution time**

Challenges

- Very large machines, will only get bigger
  - Hundreds of thousands of cores, millions of objects
- Strong scaling makes problem smaller, LB bigger
- Need to reduce costs of LB
  - Collecting statistics
    - Efficiency of statistics communication
    - Comm-aware LB requires comm statistics, but expensive to collect and use
  - Running strategies
  - Migrating objects
    - Move fewer objects
    - Move objects within same node or nearby is cheaper than elsewhere

What We're Working On

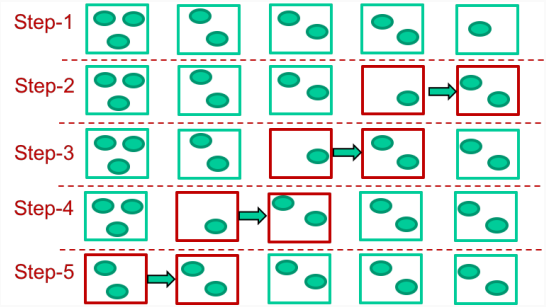
- New strategies aimed at preserving topological mapping
- Performance improvements for existing LBs
- Redesign of LB Framework
- LB for heterogeneous systems
- Miscellaneous features

Topological Strategies

Distributed Graph Refine Strategy

- Neighbor-based diffusion retains partitioning of communication graph
- Strategy
  - Each node finds neighbors in topology using Charm++ TopoManager
  - Each PE on node sends local load statistics to representative PE on the node
  - Representative PE on each node communicates load stats with neighbors
  - Object load tokens are sent/received to/from the neighbor PEs to reach a stable state where load is balanced
  - Load balancing is performed after this token passing stage
  - Since the load is transferred only to neighbors, communication cost increases by up to one hop per LB step

Distributed Graph Refine Strategy



- Diffusion-based scheme done iteratively
- Takes multiple iterations to reach stable state
- Avoids non-neighbor communication and has fewer migrations, similar to refinement based strategies
- Same step diffusion: Once tokens about load information have been exchanged, LB is performed

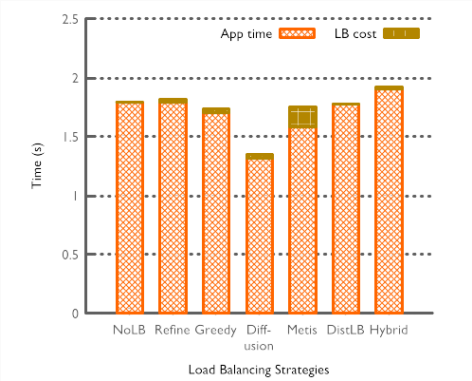
Communication Awareness

- Gain-value computation is performed to select objects to migrate
- For each object in the node:
  - Compute a gain value associated with owner node and neighbor node
  - Using heap of gain values of objects, get object with minimum gain value
    - Gain value for neighbors are taken into account
  - Once object is set to be moved, update gain values of other objects in node

$\Rightarrow$ **Keeps heavily communicating objects on the same node**

Evaluation with K-Neighbor

- *K*-Neighbor with varied stride for communication
  - Objects with low within node communication are more amenable to be migrated during LB
- 64 nodes using 8 cores per node on Blue Waters
- `DiffusionLB` minimizes communication, overall execution time
- `Metis` results in high communication, we are exploring this



Performance Improvements

Performance Improvements

Performance experiments have been done in various parts of the LB framework, but implementations have been *ad hoc* and not generalizable

Results are being incorporated into the new LB framework, which will provide benefits to all strategies

Fast-CentralLB

- Optimizes all phases of centralized load balancing:
  1. Sending stats to central PE
  2. LB algorithm (GreedyRefine implementation faster than GreedyLB)
  3. Sending migration decisions
- Stats and migration messages are radically smaller than before
- Currently only using (Topo)GreedyRefine
- Observed speedup in balancing time is 4x-7x over CentralLB+GreedyLB

Fast-CentralLB Results

Stencil3d on Blue Waters (4 procs/node, 7 worker threads/process)

Load Balancer	LB Time (s)	Migration Time (s)	Total Time (s)
Dummy	5.8	-	22.3
Greedy	40.0	-	64.9
GreedyRefine	6.4	2.1	23.9
TopoGreedyRefine	5.7	1.2	20.0

Table: 1024 Nodes, 29k PEs, 229k Objects

Fast-CentralLB Results, Con't

Stencil3d on Blue Waters (4 procs/node, 7 worker threads/process)

Load Balancer	LB Time (s)	Migration Time (s)	Total Time (s)
Dummy	13.7	-	32.1
Greedy	136.5	-	171.5
GreedyRefine	29.3	4.5	53.3
TopoGreedyRefine	22.6	2.8	39.3

Table: 4096 Nodes, 115k PEs, 917k Objects

New LB Framework

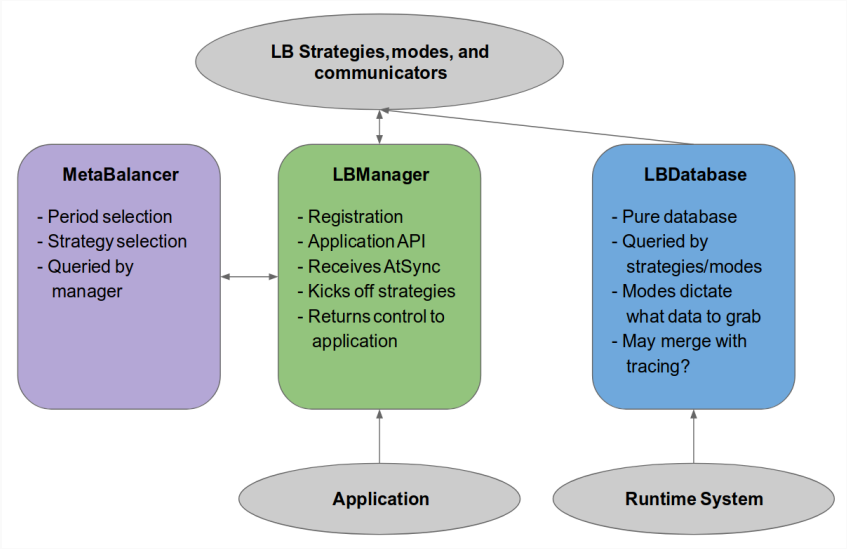
Existing Load Balancing Framework

- Current framework core consists of:
  - Database: LBDatabase, LBDBManager, LBDB
  - MetaBalancer
  - CentralLB
  - BaseLB
- No clear delineation of responsibility across all entities
- Unnecessary duplication of data structures
- A lot of core logic shows up in strategies
- Very little API accessible to application

Current Issues

- Strategies cannot easily be run with different metrics
  - Most strategies are agnostic of the semantics of the data, e.g. `GreedyLB` works equally well in balancing objects by size rather than load
- Strategies cannot be run on arbitrary domains
  - Should be able to run on the level of single process, single node, group of nodes, whole job, etc.
- Duplication of data in tracing and load balancing
- Strategy logic not well abstracted from LB infrastructure
- Statistics collection and usage inflexible and inextensible
- Hard to access and control LB from application at runtime

New LB Framework



New LB Framework

- Currently in progress, release expected later this year
- Solves all of the LB issues raised earlier
- Improves performance of load balancing infrastructure
- Cleanly separate the various components of the LB framework
- Component-based architecture allows for flexible and easy orchestration of the various parts of load balancing
- Writing strategies becomes simpler, more concise, and more flexible
  - e.g., Greedy from 157 LOC to 22 LOC

Greedy in Existing LB Framework (Excerpt of 1 of 3 files)

```
#include <algorithm>
#include "charm++.h"
#include "ckgraph.h"
#include "cklists.h"
#include "GreedyLB.h"

using namespace std;

CreateLBFunc_Def(GreedyLB, "always␣assign␣the␣heaviest␣obj␣onto␣\
lightest␣loaded␣processor.")

GreedyLB::GreedyLB(const CkLBOptions &opt): CBase_GreedyLB(opt)
{
  lbname = "GreedyLB";
  if (CkMyPe()==0)
    CkPrintf("[%d]␣GreedyLB␣created\n",CkMyPe());
}

bool GreedyLB::QueryBalanceNow(int _step)
{
  return true;
}

class GreedyLB::ProcLoadGreater {
  public:
    bool operator()(const ProcInfo &p1, const ProcInfo &p2) {
      return (p1.getTotalLoad() > p2.getTotalLoad());
    }
};

class GreedyLB::ObjLoadGreater {
  public:
    bool operator()(const Vertex &v1, const Vertex &v2) {
      return (v1.getVertexLoad() > v2.getVertexLoad());
    }
};

void GreedyLB::work(LDStats* stats)
{
```

```
      CmiAbort("GreedyLB␣cannot␣handle␣nonmigratable␣object␣on␣an␣un
    continue;
  }
  double load = oData.wallTime * stats->procs[pe].pe_speed;
  objs.push_back(Vertex(obj, load, stats->objData[obj].migratable, s
}

// max heap of objects
sort(objs.begin(), objs.end(), GreedyLB::ObjLoadGreater());
// min heap of processors
make_heap(procs.begin(), procs.end(), GreedyLB::ProcLoadGreater());

if (_lb_args.debug()>1)
  CkPrintf("[%d]␣In␣GreedyLB␣strategy\n",CkMyPe());


  // greedy algorithm
int nmoves = 0;
for (obj=0; obj < objs.size(); obj++) {
  ProcInfo p = procs.front();
  pop_heap(procs.begin(), procs.end(), GreedyLB::ProcLoadGreater());
  procs.pop_back();

  // Increment the time of the least loaded processor by the cpuTime
  // the 'heaviest' object
  p.totalLoad() += objs[obj].getVertexLoad() / p.pe_speed();

  //Insert object into migration queue if necessary
  const int dest = p.getProcId();
  const int pe   = objs[obj].getCurrentPe();
  const int id   = objs[obj].getVertexId();
  if (dest != pe) {
    stats->to_proc[id] = dest;
    nmoves ++;
    if (_lb_args.debug()>2)
      CkPrintf("[%d]␣Obj␣%d␣migrating␣from␣%d␣to␣%d\n", CkMyPe(),obj
  }
```

Greedy in New LB Framework

```
#ifndef _GREEDY_ALG_H
#define _GREEDY_ALG_H
#include <algorithm>
#include <queue>
#include <vector>

template <typename O, typename P, typename S>
class GreedyAlg {
public:
  void solve(vector<O> &objs, vector<P> &procs, S &solution, bool objsSorted)
  {
    if (!objsSorted) sort(objs.begin(), objs.end(), lb::CmpLoadGreater<O>());
    priority_queue<P, vector<P>, lb::CmpLoadGreater<P> > procHeap(lb::CmpLoadGreater<P>(), procs);
    for (int i = 0; i < objs.size(); i++) {
      const O &o = objs[i];
      P p = procHeap.top(); procHeap.pop();
      ptr(p)->assign(o);     // update load of processor
      solution.assign(o, p); // update solution
      procHeap.push(p);
    }
  }
};
#endif
```

Strategies in New LB Framework

- Most strategies consider list of “objects” (with loads), and “processors” (with background loads)
- Objects do not necessarily have to be individual chares, and processors do not have to be cores
- LB strategies simplified so they merely assign objects to processors to minimize max processor load
- Such strategies can be used at any level of the hierarchy that deals with generic “objects” and “processors”
- The framework can change what is passed to the strategy if strategy is generic enough

Hierarchical Infrastructure Redesign

- New hierarchical load balancing infrastructure replaces current CentralLB and HybridLB infrastructure
- Implements mentioned performance improvements
- Pluggable at each tree level, supports many different kinds of strategies
- How each tree level aggregates stats, makes LB decisions, etc. can be changed
- Arbitrary hierarchies are supported, e.g. a four level tree of cores $\rightarrow$ processes $\rightarrow$ nodes $\rightarrow$ entire job

Heterogeneous Load Balancing

Heterogeneous Loads

As machines get more diverse, many FLOPs come from non-CPU devices

Standard CPU load balancing in Charm++ is no longer sufficient

There are many challenges:

- How do we balance multiple metrics at once?
- How do we measure load on multiple devices?
- How do we account for data movement?

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

Current Status

Experimental versions of multiple heterogeneous execution paradigms exist:

- Vector load balancing for balancing applications with separate CPU and GPU (or other) routines
- `Accel` load balancing for balancing application with work that can be dynamically targeted to CPUs or GPUs (or others)

Currently working on making these production ready, to be released with new LB framework

Adding LB measurement to GPU Manager (talk tomorrow)

Vector Load Balancing

- Rather than a single value for load, use a vector of different load metrics
  - For example, $< cpuload, gpuload >$ or $< phase1, phase2, phase3 >$
- Different dimensions may have different optimization functions, e.g. minimize, maximize, constraint
- Not necessarily just for heterogeneous hardware, can also be used by applications with unbalanced phases
- Change strategies to consider all dimensions of vector while making decisions

Vector LB Algorithms

Currently experimenting with different algorithms for vector LB

- `GreedyMaxLB` uses maximum of the vector as load, using standard greedy assignment
- `MultiGreedyLB` creates a separate heap for each dimension of the vector and greedily assigns
  - Different from `GreedyMaxLB` because it maps to PE with minimum value for object's highest load dimension rather than PE with smallest maximum value
- `GreedyNormLB` places objects to minimize some vector norm

Vector LB Results

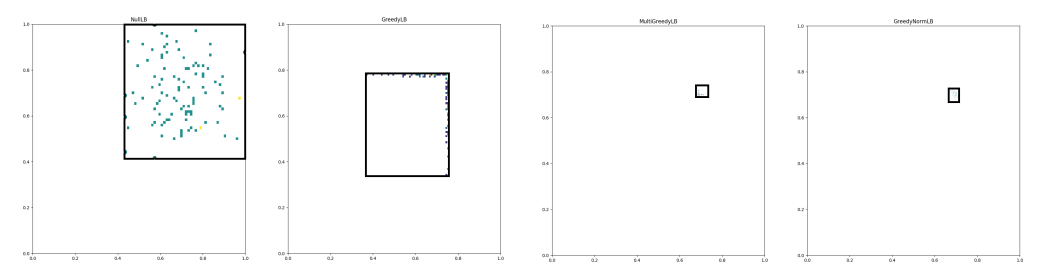


Figure: NullLB vs. GreedyMaxLB vs. MultiGreedyLB vs. GreedyNormLB

Vector LB Performance

Load Balancer	Time
NullLB	0.020 s
GreedyLB	0.357 s
MultiGreedyLB	0.319 s
GreedyNormLB	12.088 s

Figure: LB Strategy Time (512 PEs, 10k objects, 2 dimensions, simulation)

Vector LB Performance

Load Balancer	Time
NullLB	0.020 s
GreedyLB	0.357 s
MultiGreedyLB	0.319 s
GreedyNormLB	12.088 s

Figure: LB Strategy Time (512 PEs, 10k objects, 2 dimensions, simulation)

$\Rightarrow$Tradeoffs between balance quality and performance still unclear

Miscellaneous

Miscellaneous

- `MetaBalancer`:
  - System that uses ML to automatically select optimal LB frequency
  - Adding automatic selection of LB strategy
  - Currently the model is pretrained and fixed, working on adding a way to allow it to be trained by end users and for arbitrary strategies
- `TopoGreedyRefine`:
  - Extension of `GreedyRefine` (see last year's LB talk) that constrains migrations to some local topological neighborhood
  - To be released with new framework
- `DistributedLB`
  - Migration decisions are now done in multiple phases
  - Earlier phases prioritize the most overloaded PEs, which increases the probability that they migrate work

Summary

Summary

- New topological diffusion strategy optimizes for communication costs and scalability
- Performance improvements in the LB infrastructure
- New LB framework will improve performance, usability, flexibility, and applicability
- Heterogeneous load measurement and strategies address CPU/GPU and other complex imbalance

Questions?