Vector Load Balancing in Charm++

Ronak Buch
Parallel Programming Laboratory, University of Illinois at Urbana-Champaign

October 18, 2021
19th Annual Workshop on Charm++ and Its Applications
Dynamic Load Balancing

- Adaptively arrange work on PEs to maximize performance
- Execution time often determined by maximum load on a PE
- Enabled by migratable objects, load measurement
- Necessary for scaling all but very regular, static applications
What is Load?

- **Load** is a proxy value used to represent performance
  - Metric measuring utilization of a resource over a period
- Real goal is to minimize execution time, not balance load
- Traditionally, balancing for equal CPU time per PE by itself has been sufficient for high performance
- However, can we do better by considering a richer set of metrics?
Vector Load Balancing

- Rather than being a single scalar value, load is now a vector of multiple values
- Composed of things like:
  - Various resource measurements, e.g. CPU/GPU/network/memory/IO
  - Timings of separate phases of an iteration
  - Application specific parameters, e.g. number of particles
Measuring Vector Loads

- Features and APIs to add vector load measurement have been added to Charm++
  - Application can add call to indicate phase boundaries, RTS will automatically measure per-phase load
  - Runtime flags to automatically add communication load (msgs, bytes sent)
  - Can specify load vector explicitly
  - GPU load, memory use, etc. in the works
Vector Balancing

• Extra dimensionality makes vector load balancing computationally difficult
• Objects can no longer be totally ordered
• Want to minimize the “maximum” over all dimensions simultaneously
  ◦ Single variable optimization is now multivariable
• New LB strategies are needed
Vector Strategies

• A simple strategy finds the object with maximum load across all dimensions and places it on PE with minimum load in that dimension
  ○ Only works well when object has load in only one dimension, e.g. \((0, 0, 0, l, 0)\)

• For more realistic cases, have to consider vector holistically
Holistic Vector Strategies

- Place objects based on largest load in vector as before, then refine partitions to improve balance (used by METIS)
- Find object with maximum norm and place on PE with minimum norm after placement
  - Works well, but computationally expensive
  - PE “weight” varies with object, i.e. \( \| (2, 0) \|_2 < \| (0, 3) \|_2 \), but when adding object with \( (3, 0) \), \( \| (5, 0) \|_2 > \| (3, 3) \|_2 \)
NormLB - Exhaustive

- Initial implementation orders objects by norm and then does exhaustive search across all PEs for placement
  - Quality is exactly as desired
  - Performance is very poor ($\Theta(p \cdot o)$)

<table>
<thead>
<tr>
<th>Method</th>
<th>Makespan</th>
<th>Strategy Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>1965.83</td>
<td>0.32</td>
</tr>
<tr>
<td>Norm</td>
<td>1674.86</td>
<td>22.72</td>
</tr>
</tbody>
</table>

Table: Greedy vs Norm (1e4 PEs, 1e6 objs)
To improve performance, we use a \( k \)-d tree to guide PE selection

- Arbitrary dimension space partitioning tree
- Allows PE search to be bounded as candidates are found

\( k \)-d works well for searching in static point set, but here, tree updated after every assignment

- Costly update operations
- Pattern of updates often results in unbalanced tree

Can be worse than the naïve exhaustive version!
Random Relaxed $k$-d

- Random Relaxed $k$-d trees help solve these problems; two key differences from standard $k$-d:
  - **Relaxed** Instead of cycling through discriminants, $1, 2, \ldots, k, 1, \ldots$, each node stores arbitrary discriminant $j \in \{1, 2, \ldots, k\}$
  - **Random** Discriminant is uniformly randomly chosen and each insertion has some probability of becoming the root, or root of subtree, \ldots
Random Relaxed $k$-d

Figure: $k$-d

Figure: $rk$-d
NormLB - $rk$-d

- These low-cost arbitrary updates and stochastic balancing improve LB (all provide same results)

<table>
<thead>
<tr>
<th>Method</th>
<th>Strategy</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1e4 PEs, 1e5 objs</td>
<td>1e4 PEs, 1e6 objs</td>
</tr>
<tr>
<td>Exhaustive</td>
<td>2.18</td>
<td>21.54</td>
</tr>
<tr>
<td>Standard $k$-d</td>
<td>0.93</td>
<td>27.55</td>
</tr>
<tr>
<td>Relaxed $k$-d</td>
<td>0.57</td>
<td>7.96</td>
</tr>
</tbody>
</table>

Table: Performance of Norm-Based Strategies
Vector LB Performance - AMPI

Time in Microseconds

0 100,000,000 200,000,000 300,000,000

PE 0
(57, 57)

PE 1
(57, 57)

PE 2
(57, 57)

PE 3
(57, 57)

AMPI - No Load Balancing
Vector LB Performance - AMPI

Time in Microseconds

0 100,000,000 200,000,000 300,000,000

PE 0
(73, 57)

PE 1
(73, 57)

PE 2
(73, 57)

PE 3
(73, 57)

AMPI - Regular Load Balancing
Vector LB Performance - AMPI

AMPI - Vector Load Balancing

Time in Microseconds

0  100,000,000  200,000,000  300,000,000

PE 0
(77, 57)

PE 1
(77, 57)

PE 2
(77, 57)

PE 3
(77, 57)
Vector LB Performance - AMPI

LB Off

Phase Unaware (1.44x speedup)

Phase Aware (1.67x speedup)
Vector LB Performance

Timeline of phase-based application:

![Timeline of phase-based application](image-url)
Vector LB Performance

No LB
Vector LB Performance

Scalar LB
Vector LB Performance

Vector LB

Vector Load Balancing in Charm++
Locality in LB

• Vector loads give performance insight with increased nuance and detail

• However, performance may also vary based on the location of objects
  ◦ The distance between communicating objects changes latency, load on links, routers
  ◦ Balanced via graph partitioners, geometric strategies

• Currently captured via RTS communication graph or application provided positions
  ◦ For vector: first application positions, then comm graph
LB Position API

- Geometric strategies use *ad hoc* data passing
  - e.g. ChaNGa uses `LBRegisterObjUserData` to pass in `void*`
  - Each application needs its own custom LB strategies
- Adding standardized LB position API to Charm++
  - `setObjPosition(const vector<LBRealType>& pos)`
  - Allows positions of arbitrary dimension
  - Load balancers can opt-in for positions at registration time
- Allows for generic, application agnostic strategies
- Fully implemented, no results yet, currently testing with ChaNGa and other applications, slated in 7.1
Vector Geometric Strategies

• Currently using orthogonal recursive bisection with position API
• In scalar world, find split coordinate that minimizes differences in load between both halves
• In vector world, things are more complicated
  ◦ Each dimension may have a different split coordinate
  ◦ Select by taking average, minimizing square difference, etc.
  ◦ Rather than splitting at a single coordinate, allow objects in some neighborhood to go to either half
  ◦ Still topic of active experimentation
Future Vector LB Work

- Dimensionality reduction to simplify problem
- Performance can still be an issue
  - Have bounded versions of Norm LBs to tradeoff quality and performance
  - Further optimizations of search space are possible
  - Can use relaxation and approximation to tune
- Add support for constraint based objective functions rather than always minimizing everything
- Support for GPU, cache, memory, I/O load, comm graph
Conclusions

- Complex, modern applications need sophisticated performance measurement
- Combining different metrics into a vector has been shown to improve the quality of LB
- New techniques must maintain communication locality to be useful for certain class of applications