PPL: An Abstract Runtime System for Hybrid Parallel Programming

Alex Brooks\textsuperscript{1}, Hoang-Vu Dang\textsuperscript{1}, Nikoli Dryden\textsuperscript{1}, Marc Snir\textsuperscript{1,2}

\textsuperscript{1}University of Illinois at Urbana-Champaign
\textsuperscript{2}Argonne National Laboratory
Motivation

• Efficiency requires exploiting inter- and intra-node parallelism
  – Load balancing, latency hiding, communication management

• Requirements:
  – Lightweight threads (tasks)
  – RDMA
  – Support large numbers of communicating tasks
    • Close integration of threading and communication
  – Fine-grained remote synchronization
    • Efficient event-driven task scheduling
Outline

1. Design and Implementation
2. Runtime Evaluation
3. Application Evaluation
4. Future Work
PPL Design

• Modularized elements:
  – Communication Layer
  – Memory Management System
  – Threading Layer

• Implemented in C++11

• Provide necessary interaction between communication events and scheduling

• Modules are interoperable for minimizing runtime overhead

• **Result:** Flexible tool for analyzing & developing future runtime systems
Communication

• One-sided communication & active messages
  – Performed on global objects
• Two implementations
  – GASNet – general purpose
  – RDMAX – custom-built with InfiniBand Verbs API
• Dedicated communication threads
  – Offload work from communicating threads
  – 1 thread for communication, 1 for thread re-scheduling
Memory Management

• Partitioned Global Address Space (PGAS) Environment
  – Global and local heaps

• Global objects (gptr, gvar, and gvec)
  – Allocated on global heap, cached objects on local heap

• Software caching
  – Synergistic caching
  – No implicit coherence
  – Policies: nocache, noevict, evict
  – Different get operations specify interaction with the cache
    • lget, rget, get
Threading

- Support threads/futures which can yield
- Goals:
  - Flexible enough to use any underlying threading library
  - Support any threading model (tasks vs. kernel threads)
- Two implementations:
  - Argobots and Qthreads
- Primary synchronization primitive modeled as full/empty-bit semantics
  - Native in Qthreads
  - Custom-built atop Argobots
Outline

1. Design and Implementation
2. Runtime Evaluation
3. Application Evaluation
4. Future Work
Experiment Details

• Tested on Stampede Supercomputer at TACC
  – Mellanox FDR Infiniband
  – 2 Intel Xeon E5-2680 (8 C)
  – 6,400 Nodes

• Use the following module implementations unless otherwise noted:
  – RDMAX, Qthreads, noevict cache
Communication Overhead

- Perform `gptr` get operations with 64 threads (left graph) & single thread (right graph)
Caching Overhead

- Perform get operations with each of PPL’s caching policies
  - Compared against strictly performing get operations

<table>
<thead>
<tr>
<th></th>
<th>nocache</th>
<th>noevict</th>
<th>evict</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 B</td>
<td>7.4 %</td>
<td>15.3 %</td>
<td>17.7 %</td>
</tr>
<tr>
<td>1 KB</td>
<td>8.3 %</td>
<td>6.6 %</td>
<td>14.8 %</td>
</tr>
<tr>
<td>1 MB</td>
<td>0.8 %</td>
<td>0.2 %</td>
<td>0.6 %</td>
</tr>
</tbody>
</table>
Threading Overhead

• Perform thread creation and joining
  – Overhead added by PPL to Qthreads & Argobots

![Qthreads Overhead Graph](image1)

![Argobots Overhead Graph](image2)
Outline

1. Design and Implementation
2. Runtime Evaluation
3. Application Evaluation
4. Future Work
Barnes-Hut Results

- PPL implementation based off previous work\(^2\)

![Diagram showing force computation time vs. number of bodies for Qthreads and Argobots for different core counts.](image)

<table>
<thead>
<tr>
<th>Number of Bodies</th>
<th>(2^{16})</th>
<th>(2^{17})</th>
<th>(2^{18})</th>
<th>(2^{19})</th>
<th>(2^{20})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qthreads</td>
<td>1.3</td>
<td>2.9</td>
<td>4.9</td>
<td>6.5</td>
<td>5.1</td>
</tr>
<tr>
<td>Argobots</td>
<td>2.6</td>
<td>8.0</td>
<td>13.0</td>
<td>13.4</td>
<td>8.8</td>
</tr>
</tbody>
</table>

Monte Carlo Particle Transport

• Approximate the diffusion of particles through a structure
  – Involve frequent lookups of large, static cross-section data (100GB+)
• Energy Band Memory Server (EBMS) algorithm

![Graph showing execution time vs. number of cores](image)

<table>
<thead>
<tr>
<th>Number of cores</th>
<th>$2^8$</th>
<th>$2^9$</th>
<th>$2^{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPL (Qthreads)</td>
<td>1.09</td>
<td>1.15</td>
<td>1.16</td>
</tr>
<tr>
<td>PPL (Argobots)</td>
<td>1.04</td>
<td>1.11</td>
<td>1.19</td>
</tr>
</tbody>
</table>

Sparse Triangular Linear System Solver

- Implementation based off Charm++ implementation\(^4\)
  - PPL uses active messages to provide data-driven execution model

---

Outline

1. Design and Implementation
2. Runtime Evaluation
3. Application Evaluation
4. Future Work
Future Work

• Reduce overhead of threading module
  – Expect native implementation to improve

• Improve interactions between communication threads and communicating tasks

• Develop better performance models that focus on close communication and threading interaction

• Continued development
THANK YOU!

Contact info:
brooks8@illinois.edu | hdang8@illinois.edu | dryden2@illinois.edu | snir@illinois.edu

QUESTIONS?