Comparing Parallel Programming Models Using GRAMPS
Daniel Sanchez, David Lo, Jeremy Sugerman, Richard Yoo, Christos Kozyrakis

Overview

- Modern parallel programming models
  - Different constructs for expressing parallelism
  - Automatically managed communication and scheduling
- How to compare?
  - Different implementations, architectures etc.
  - Focus on resource management, not syntax/language
- Our approach
  - Develop runtime for a rich programming model (GRAMPS)
  - Modify runtime to express other models

Model Differences

<table>
<thead>
<tr>
<th></th>
<th>Data-parallel Shaders</th>
<th>Producer-Consumer</th>
<th>Hierarchical Work</th>
<th>Adaptive Scheduling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task-Stealing</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Breadth-First</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Static</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>GRAMPS</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Programmed Models

- Task-Stealing (Cilk, TBB / multicore)
  - Sea of “tasks”, task queues with work-stealing
  - Low overhead with fine granularity tasks
  - No producer-consumer locality or aggregation
- Breadth-First (CUDA, OpenCL / GPGPU)
  - Pipeline / DAG of “kernels”; data-parallel shaders
  - Simple scheduler
  - No producer-consumer, no pipeline parallelism
- Static (StreamIt / Streaming)
  - Graph of “stages” and “streams”; offline scheduling
  - No runtime scheduler overheads; complex schedules
  - Cannot adapt to irregular workloads

GRAMPS

- Graph of stages communicating through queues
  - Stages:
    - Shader (stateless, automatically instanced)
    - Thread (stateful)
  - Queues: Bounded size, operate in packets
- Dynamic scheduling
  - Maximize utilization while keeping working sets small
  - Exploit producer-consumer locality

GRAMPS Implementation

- x86 pthreads + atomics
- GRAMPS scheduling
  - Application-specific queue sizes
  - Work-stealing task priority queues
  - Static per-stage priorities
  - Preemption after a low watermark
- Implementing other models
  - Task-stealing: unbounded queues, no priorities, preempt to child tasks (depth-first)
  - Breadth-first: unbounded queues, one stage at a time
  - Static: unbounded queues, offline schedule using SAS/SGMS

Methodology

- System: 2-socket Core i7 (8 cores, 16 threads)
- Applications: Multiple sources
  - GRAMPS: Raytracer, Spheres
  - MapReduce: Hist, LR, PCA
  - Cilk: Mergesort
  - CUDA: Gaussian, SRAD
  - StreamIt: FM, TDE

Conclusions

- Adaptive scheduling is the obvious choice for multicore
  - Better load-balance / handling of irregularity
- Task-Stealing
  - For fine-grained apps and apps without task structure
- GRAMPS
  - Uses high level info to balance utilization with memory footprint
  - Should be increasingly important with more cores & complex apps

Evaluation

- Small overheads for dynamic scheduling
- Breadth-First has larger idle time (no pipeline parallelism)
- Queue overheads can be large (Task-Stealing increases contention)

- Breadth-First has worst-case data footprint (no pipeline parallelism)
- Task-Stealing has significantly larger footprint than GRAMPS

- System: 2-socket Core i7 (8 cores, 16 threads)