Comparing Parallel Programming Models Using GRAMPS

Daniel Sanchez, David Lo, Jeremy Sugerman, Richard Yoo, Christos Kozyrakis

Overview

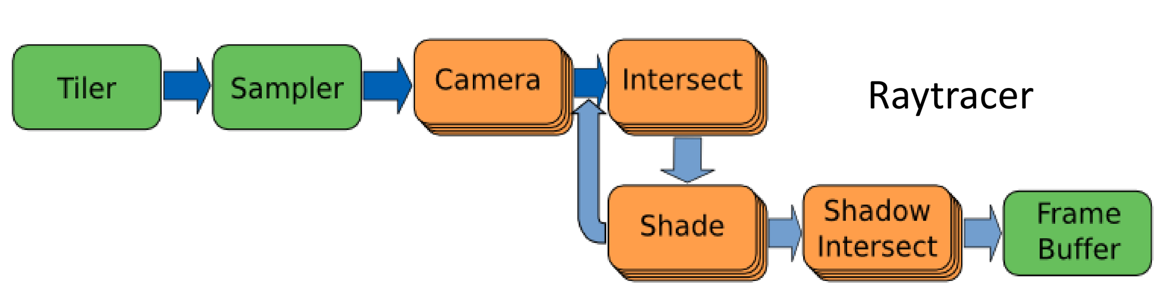
- Modern parallel programming models
  - Different constructs for expressing parallelism
  - Automatically managed communication and scheduling
- How to compare?
  - Different implementations, architectures etc.
  - Focus on resource management, not syntax/language
- Our approach
  - Develop runtime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

Programming 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 though queues
  - Stages:
    - Shader (stateless, automatically instanced)
    - Thread (stateful)
  - Queues: Bounded size, operate in packets



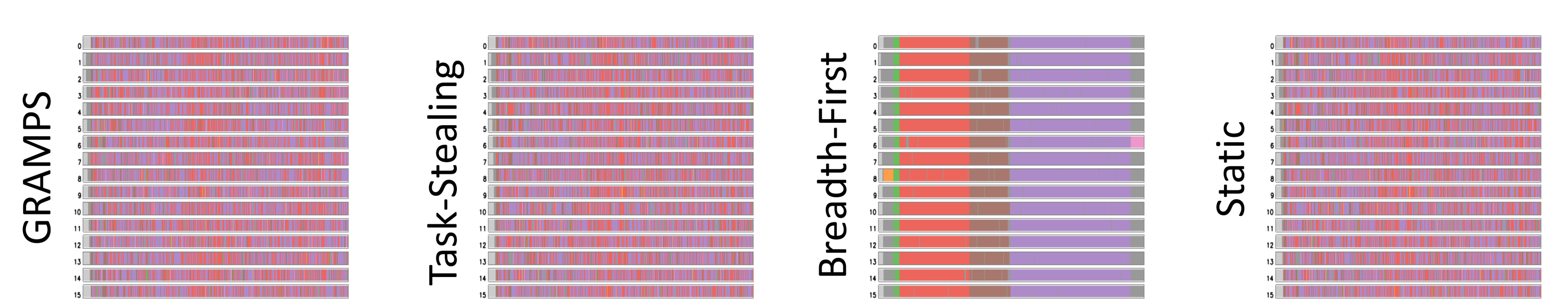
- Dynamic scheduling
  - Maximize utilization while keeping working sets small
  - Exploit producer-consumer locality

Model Differences

	Data-parallel Shaders	Producer-Consumer	Hierarchical Work	Adaptive Scheduling
Task-Stealing	No	No	No	Yes
Breadth-First	Yes	No	Yes	No
Static	Yes	Yes	Yes	No
GRAMPS	Yes	Yes	Yes	Yes

GRAMPS Implementation

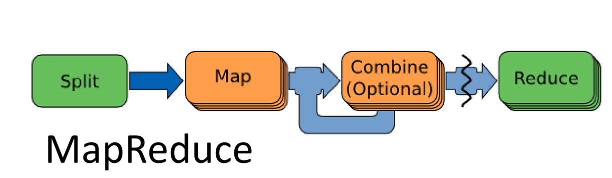
- x86 pthreads + atomics
- GRAMPS scheduling
  - Application-specified queue sizes
  - Work-stealing task priority queues
  - Static per-stage priorities
  - Preemption after a low watermark
- Implementing other models
  - Task-stealing: unbounded queues, no priorities, preempt to child tasks (depth-first)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

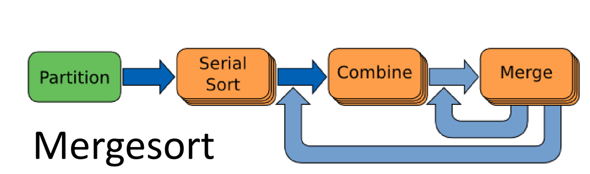


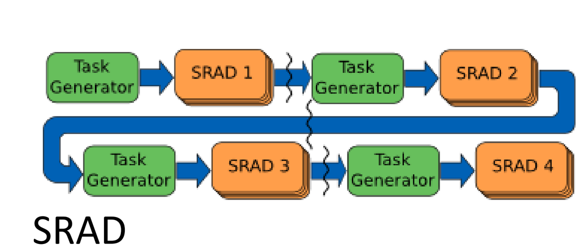
Raytracer Schedules

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

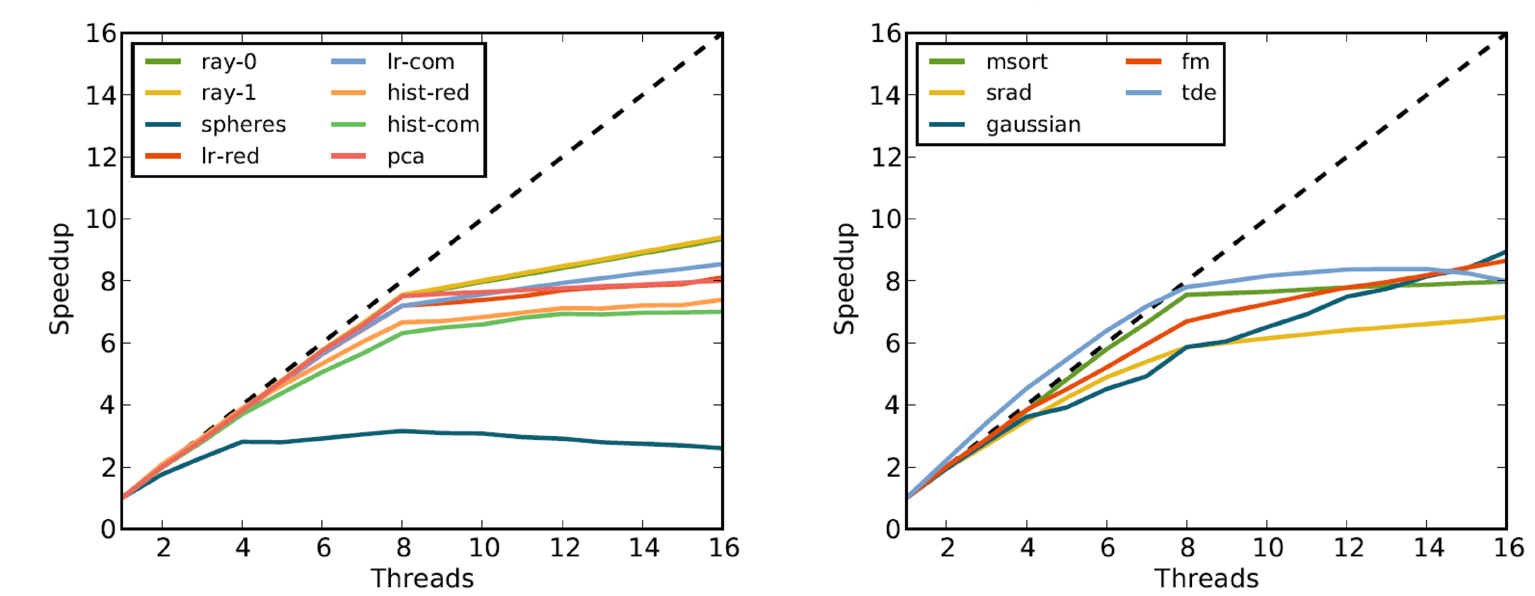


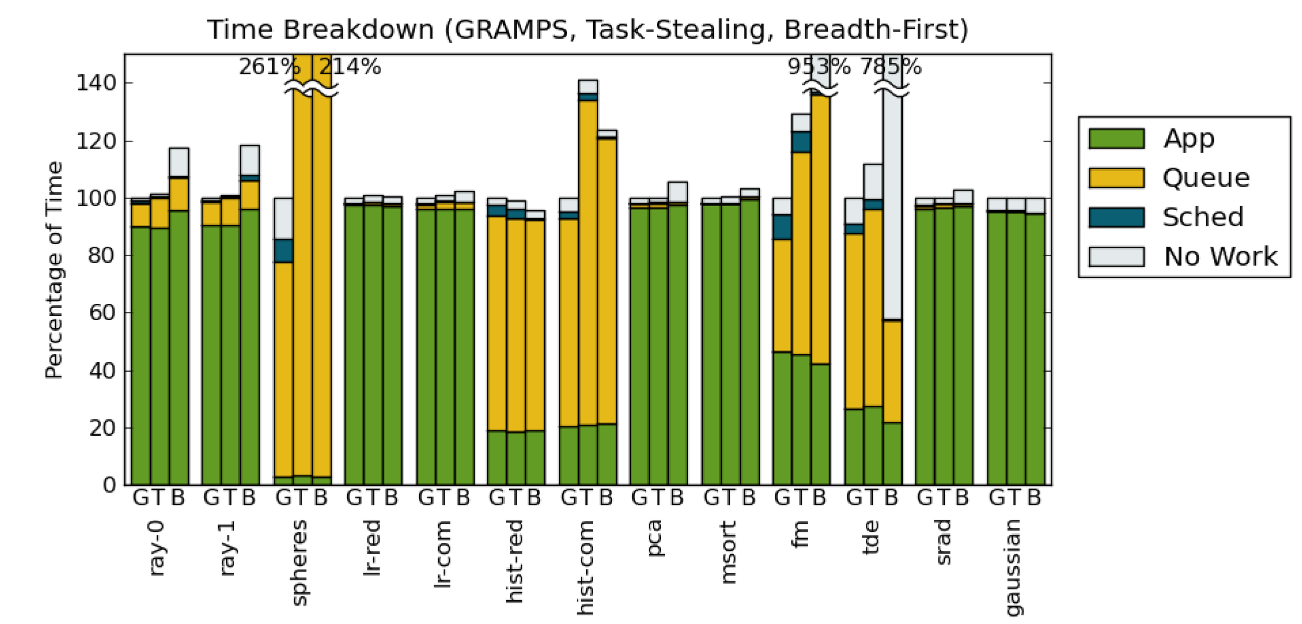




Evaluation

GRAMPS Runtime Scalability





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

App	Avg Data Queue footprint			Avg Task Queue footprint	
	GRAMPS	Task-Stealing	Breadth-First	GRAMPS	Task-Stealing
ray-0	79.2 KB	94.2 KB	21797.9 KB	0.9 KB	1.2 KB
ray-1	93.9 KB	140.0 KB	30056.6 KB	1.0 KB	1.7 KB
spheres	193.5 KB	193.7 KB	163.6 KB	6.7 KB	8.1 KB
lr-red	3.7 KB	3.2 KB	2.1 KB	0.6 KB	0.9 KB
lr-com	21.2 KB	21.2 KB	12.7 KB	0.7 KB	1.0 KB
hist-red	4013.5 KB	4002.6 KB	4094.4 KB	2.9 KB	3.2 KB
hist-com	736.2 KB	736.3 KB	4773.2 KB	3.9 KB	4.4 KB
pca	0.6 KB	0.7 KB	131.8 KB	0.8 KB	1.0 KB
msort	2.6 KB	2.6 KB	9.6 KB	2.7 KB	2.4 KB
fm	1156.0 KB	1657.4 KB	24895.6 KB	1.9 KB	3.5 KB
tde	2144.4 KB	2202.6 KB	4078.5 KB	1.1 KB	1.0 KB
srad	0.7 KB	0.9 KB	40.0 KB	0.7 KB	1.0 KB
gaussian	0.6 KB	0.8 KB	1.9 KB	0.6 KB	0.9 KB

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

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