Locality-Aware Task Management for Unstructured Parallelism: A Quantitative Limit Study

Richard Yoo, Christopher Hughes,
Changkyu Kim, and Yen-Kuang Chen
Intel Corporation

Christos Kozyrakis
Stanford University

Notice and Disclaimers

- Notice: This document contains information on products in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information. Contact your local Intel sales office or your distributor to obtain the latest specification before placing your product order.

- INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications, product descriptions, and plans at any time, without notice.

- All products, dates, and figures are preliminary for planning purposes and are subject to change without notice.

- Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined.“ Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

- Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

- The Intel products discussed herein may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

- Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's website at http://www.intel.com.

Notice and Disclaimers (contd.)

- Intel® Itanium®, Intel® Xeon®, Xeon Phi™, Pentium®, Intel SpeedStep® and Intel NetBurst® , Intel®, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries. Copyright © 2012, Intel Corporation. All rights reserved.

- *Other names and brands may be claimed as the property of others.

- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

- Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice revision #20110804.

Shifting the Landscape of Computing

- Limits in technology scaling, power, and complexity
  - Parallelism now the norm to obtain sustainable performance
- More cores are being packed on the same die
  - Mainstream server processors contain 8 cores (16 threads)
  - 61 core co-processors already available in the market
- More cores increases memory demand
  - On-chip cache hierarchies are getting deeper and more complex

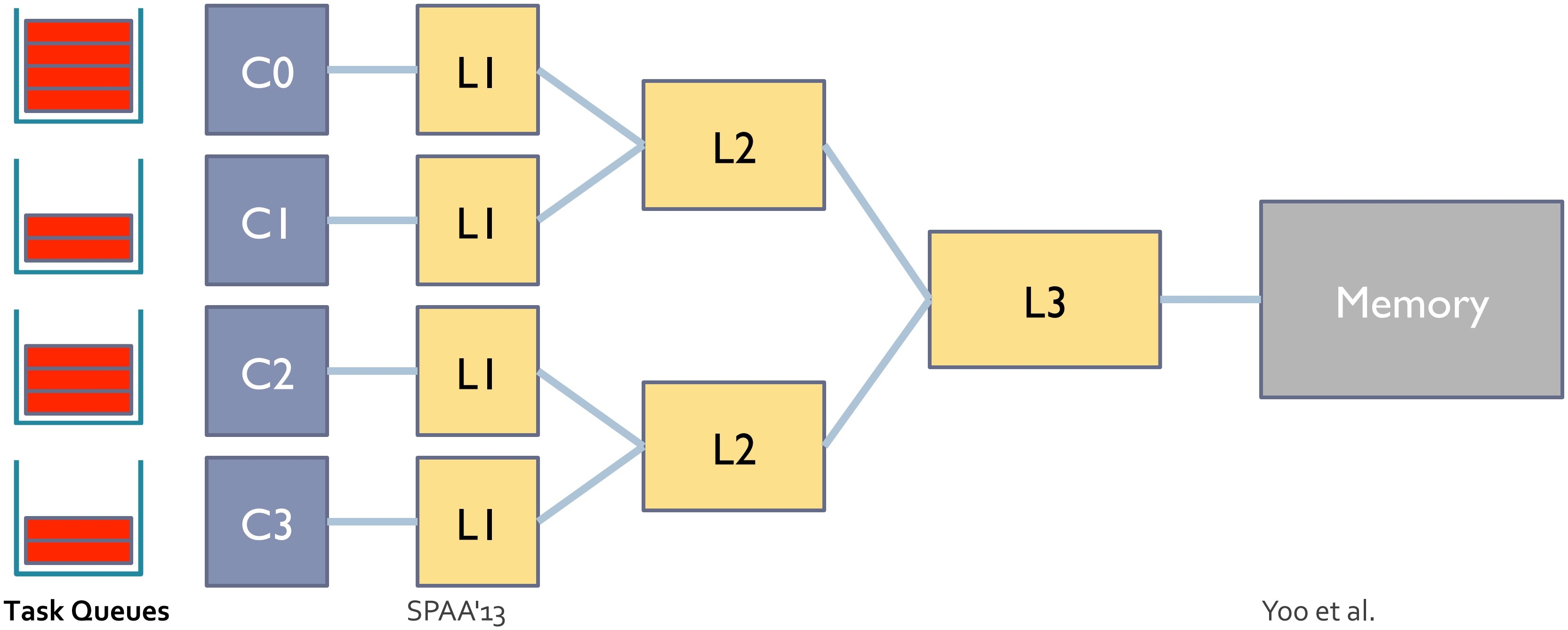
More Cores → Locality Matters More

- NUMA effects prevalent on a single chip
  - **Performance**: remote cache and memory accesses cause significantly higher latency
  - **Energy**: moving a word of data consumes up to 20x energy than an arithmetic op on that word [SC'09]
- Data access locality should be exploited
  - Avoid remote cache and memory accesses
  - Reduce execution time and energy consumption

[SC'09] W. Dally. "*The Future of GPU Computing.*" In 22nd SC, 2009.

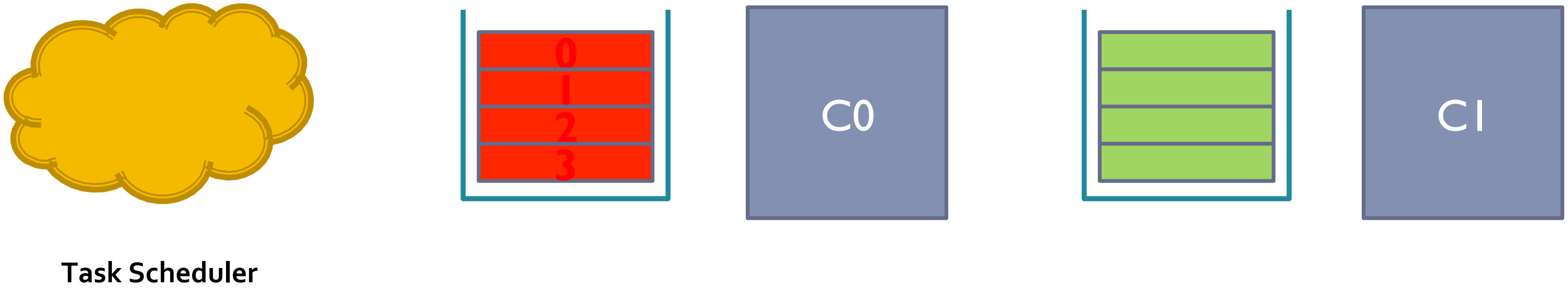
Locality on Task-Based Systems

- *Task-based programming systems* (e.g., Cilk, OpenMP, TBB)
  - A program is broken down into small *tasks*
  - Runtime *schedules* tasks across *task queues* for execution
  - Once a thread runs out of tasks, it may *steal* from other threads



Making Scheduler Locality-Aware

- To capture locality on a task-based system
  - The *scheduling algorithm* should be locality-aware
- Scheduler performs two things
  - Group a set of tasks to execute on the same core
  - Given those groups, determine the execution order
- Perform *grouping* and *ordering* in a locality-aware fashion



Making Scheduler Locality-Aware

- Exact scheduling logic depends on the parallelism model

1. Structured parallelism (= task-parallel systems)
   - Explicit data/control dependencies exit across tasks
   - Relatively easy to capture locality
     - Group producers and consumers
     - Follow the dependency order

2. Unstructured parallelism (= data-parallel systems)
   - No dependency exists (e.g., parallel for-all)
   - Lack of dependency info = larger solution space
   - Complex cache hierarchies also a challenge
   - Capturing locality is hard, remains largely unsolved

Contributions

- Propose a systematic approach to exploit locality
  1. Use a graph-based locality analysis framework
  2. Generate offline schedules that are optimized for target
- Perform limit study to quantify the potentials
  - Simulate 3 different multicore designs
  - Scale up to 1024 cores
- Conduct a thorough scheduler design space exploration
  - Capture the potentials in a practical setting
  - Provide a guideline for future scheduler implementations
- Show stealing can be made locality-compatible

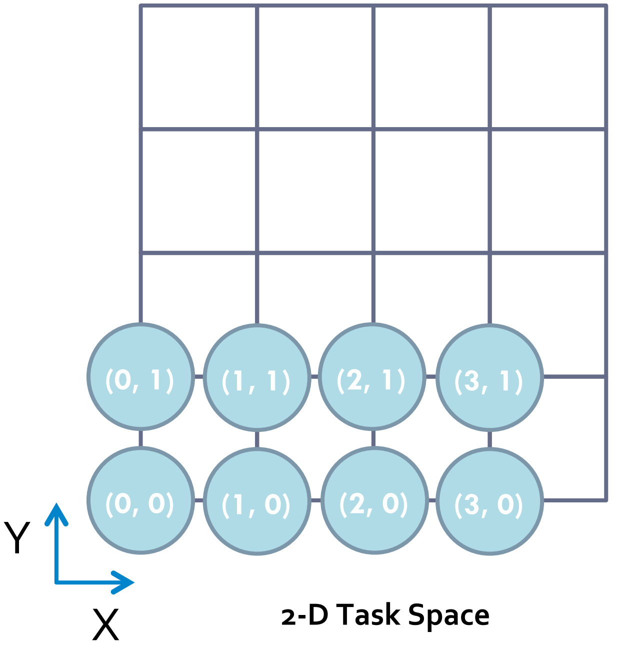
Agenda

- Motivation
- Locality-Aware Task Scheduling
  - Graph-Based Locality Analysis
  - Recursive Scheduling
- Performance Results
- Locality-Aware Task Stealing
- Summary

Unstructured Parallelism

```
void
conv_task(int x_pos, int y_pos);
```

Task Function Signature

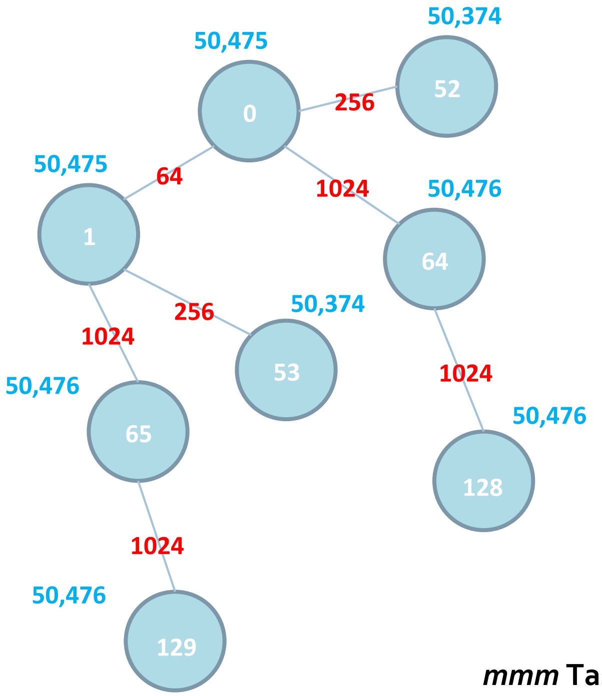


2-D Task Space

- A parallel section = (1) *task function* and (2) *task space*
  - Task = bundle of a coordinate and the task function
  - Execute = invoke the task function with the coordinate as the argument
- No dependencies among tasks
  - A task may execute on any thread, any time, without affecting correctness
  - A scheduler may arbitrarily group and order tasks
- Challenge
  - Too many ways to group/order tasks
  - Complexity of the underlying cache hierarchy

Graph-Based Locality Analysis

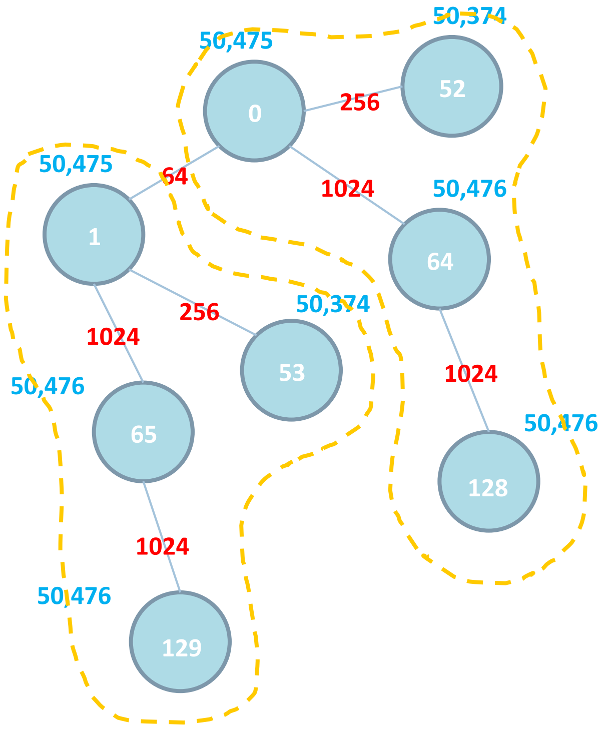
1. Profile each workload to collect read/write sets for each task
2. For each parallel section, construct a *task sharing graph*
3. Perform grouping and ordering to understand inherent locality



○ 0	Task (task id)
—	Sharing
Vertex Weight	Task size (# dynamic instrs)
Edge Weight	# cache lines shared

***mmm* Task Sharing Graph (Partial)**

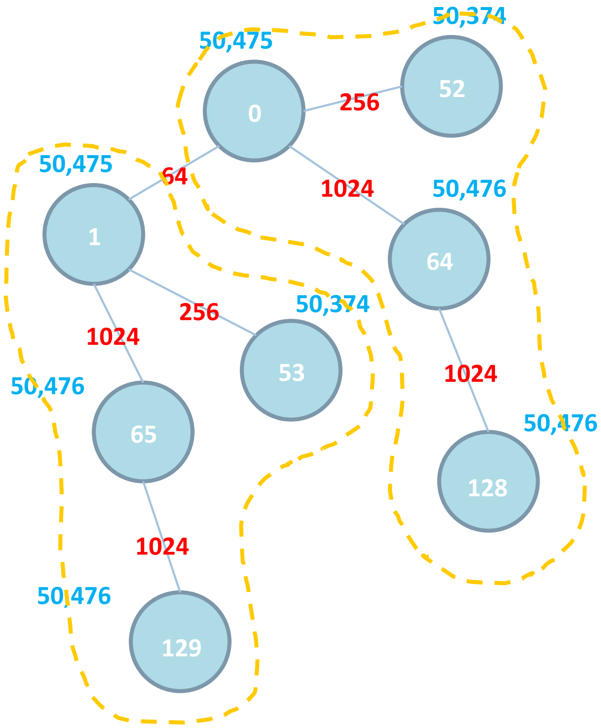
Locality Implications: Task Grouping



mmm **Task Sharing Graph (Partial)**

- Dispatch a group of tasks at a time
  - Amortize scheduling overheads
  - Capture data reuse across tasks
- *Partition* the task sharing graph s.t.
  - The sum of edge weights within each group is maximized (maximize locality)
  - Equalize the sum of vertex weights for each group (load balance)
  - Task group should be sized so that the working set fits in cache (from trace)
- NP-hard
  - Use a heuristic graph partitioner (METIS)

Locality Implications: Task Ordering

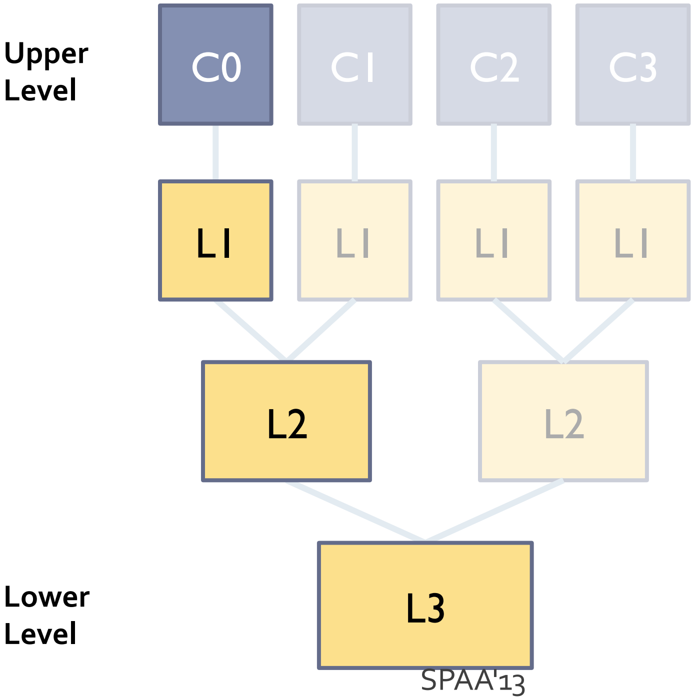


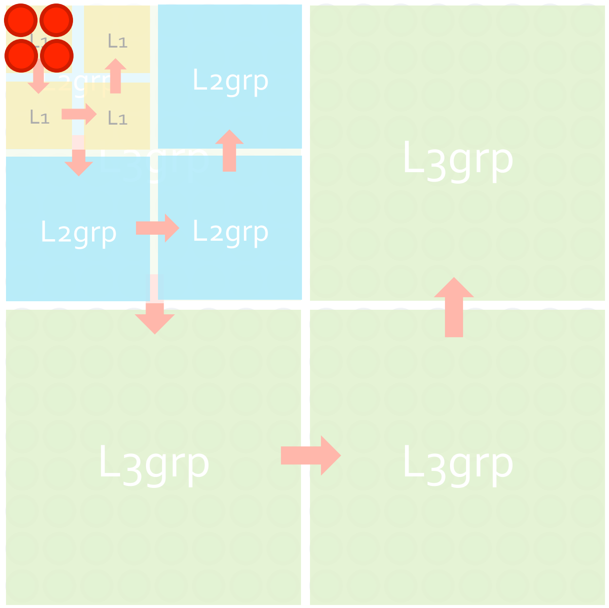
mmm Task Sharing Graph (Partial)

- Order tasks to minimize the *distance between reuses* of shared data
  - Improves temporal locality
- Obtain a *traversal order* of vertices s.t. it minimizes reuse distance
- NP-hard
  - Use Maximum Spanning Tree (MST) construction
    - Accumulate the history of scheduled tasks and pick the next task with maximum sharing
- Task groups can be similarly ordered

Matching the Cache Hierarchy: *Recursive Scheduling*

- Perform task grouping and ordering in a *recursive fashion*
  - Start from bottom: generate task groups and order them
  - Recursively apply the process, targeting one level up in the hierarchy each time
- Results in a hierarchy of task groups
  - Entire stack of task groups stay *resident* across the entire cache hierarchy





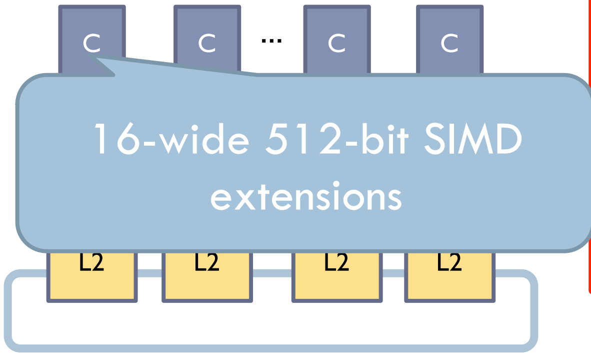
Tasks to Schedule

Agenda

- Motivation
- Locality-Aware Task Scheduling
  - Graph-Based Locality Analysis
  - Recursive Scheduling
- Performance Results
- Locality-Aware Task Stealing
- Summary

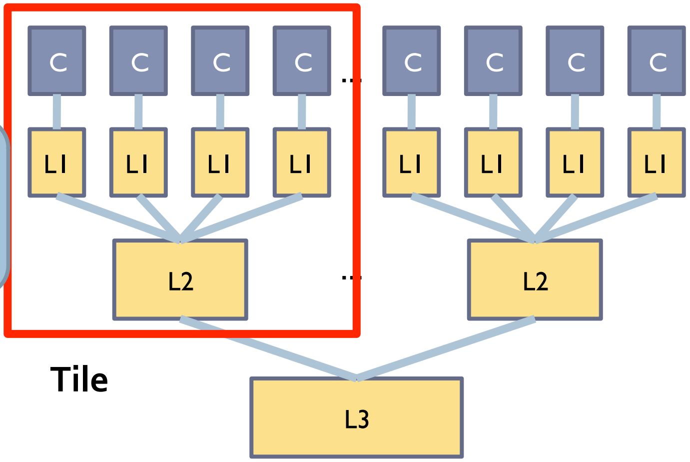
Experiment Settings

- Represent *various approaches* to many-core designs
  - Simulate three different 32-core designs
  - Scale up to 1024 cores (*Futuristic Processor*)

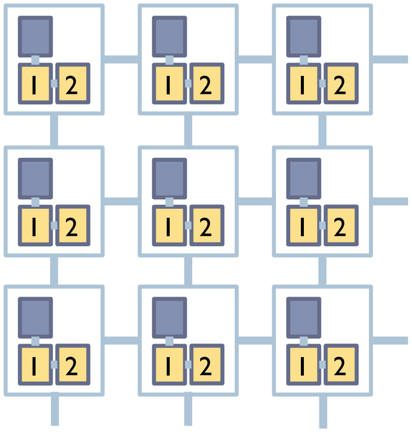


16-wide 512-bit SIMD extensions

Throughput Processor
(e.g., Intel Xeon Phi)
(All-Private Cache Hierarchy)



Tiled Processor
(Tree-Based Cache Hierarchy)



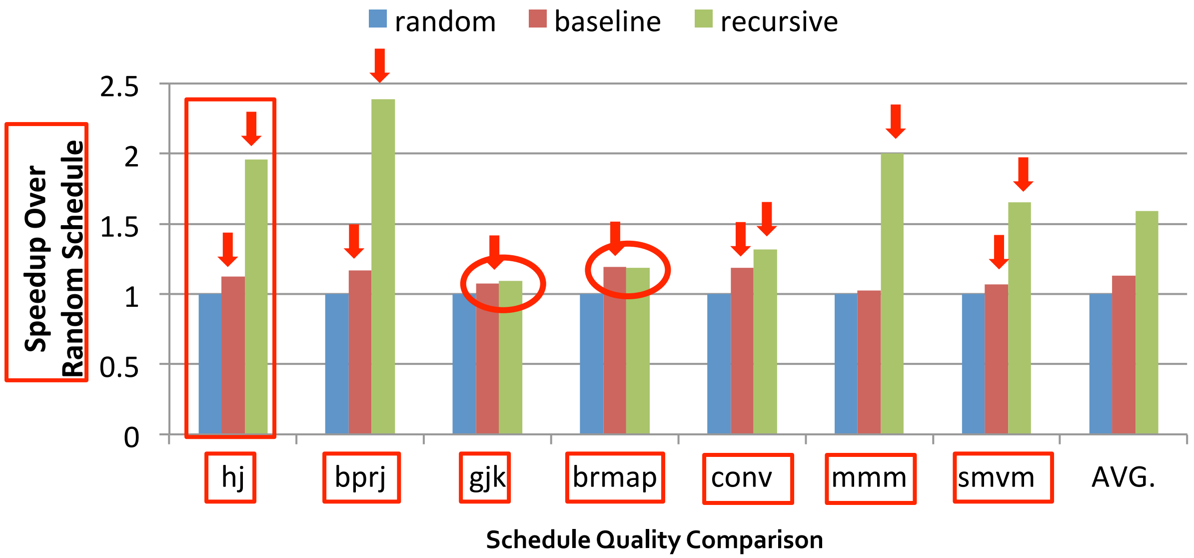
Futuristic Processor
(All-Private Cache Hierarchy)

Experiment Settings (contd.)

- Generate recursive schedules for *each* architecture, offline
  - Static scheduler reads in the schedule and populates queues
- Compare against different schedules
  - Recursive schedule (upper bound)
  - Random schedule (random bound)
    - Randomly group and order tasks into # threads
  - Baseline schedule (state-of-the-art)
    - Split the single-thread schedule into # threads
    - Exploits sequential locality across parallel execution [SPAA'07]
- Quality metric of a schedule: *sum of the execution time of tasks*
  - Will talk about the impact of load imbalance later
- Insights were similar
  - Start with Throughput Processor, and contrast the others

[SPAA'07] S. Chen et al. "*Scheduling threads for constructive cache sharing on CMPs.*" In Proc. of the 19th SPAA, 2007.

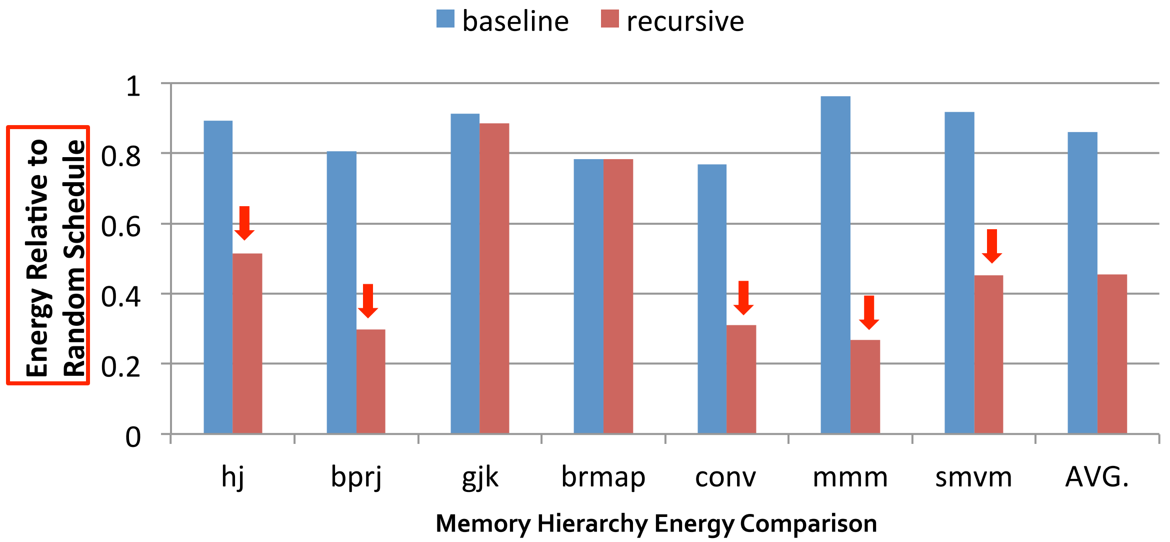
Performance Improvement



- Baseline captures *some* locality over random
- Recursive scheduling significantly improves performance
  - 1.41x over baseline (both L1 and L2 misses were reduced)
  - For **gjk** and **brmap**, baseline was good enough

- Much room for locality-aware task scheduling
  - Locality will be even more important on larger scales

Memory Hierarchy Energy Reduction

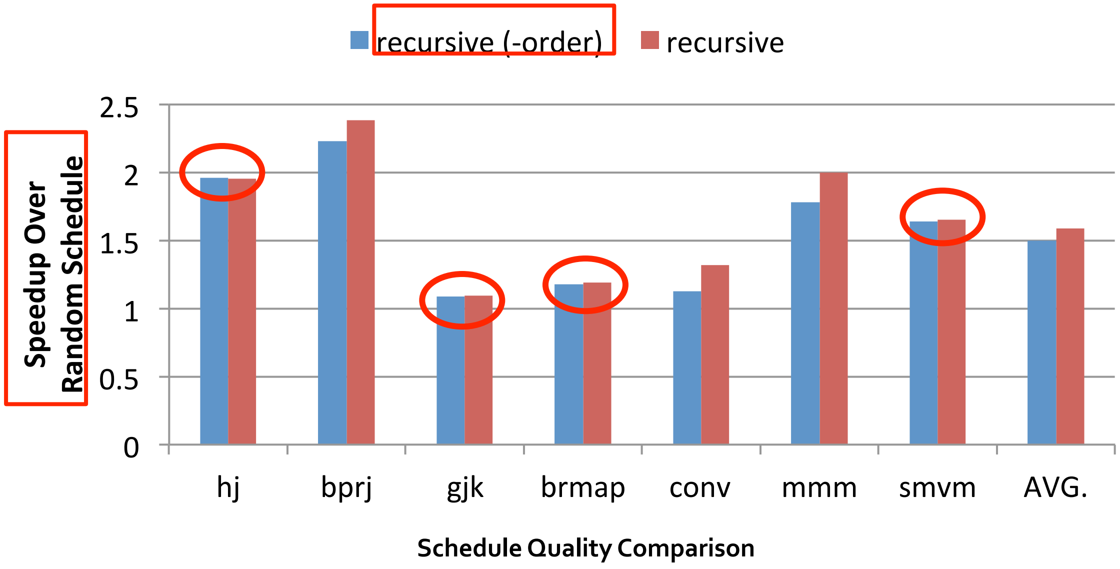


- Memory hierarchy energy from the model [Micro'10]
  - Activity counts for memory hierarchy beyond L1 + CACTI

- Reduction in cache misses translates into significant energy savings
  - 55% lower than random, and 47% lower than baseline

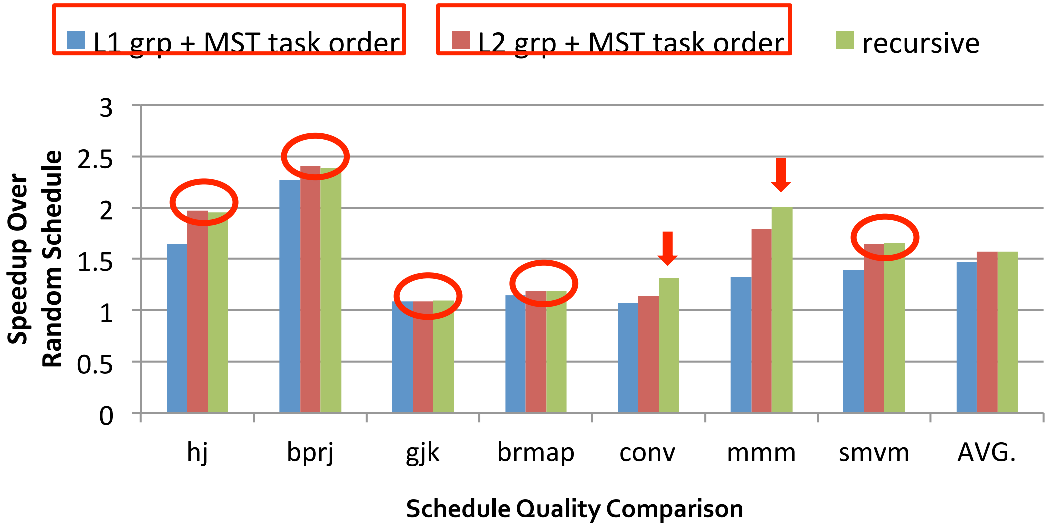
[Micro'10] C. Hughes et al. *"Performance and energy implications of many-core caches for throughput computing."* Micro, IEEE, 30(6), 2010.

Grouping vs. Ordering



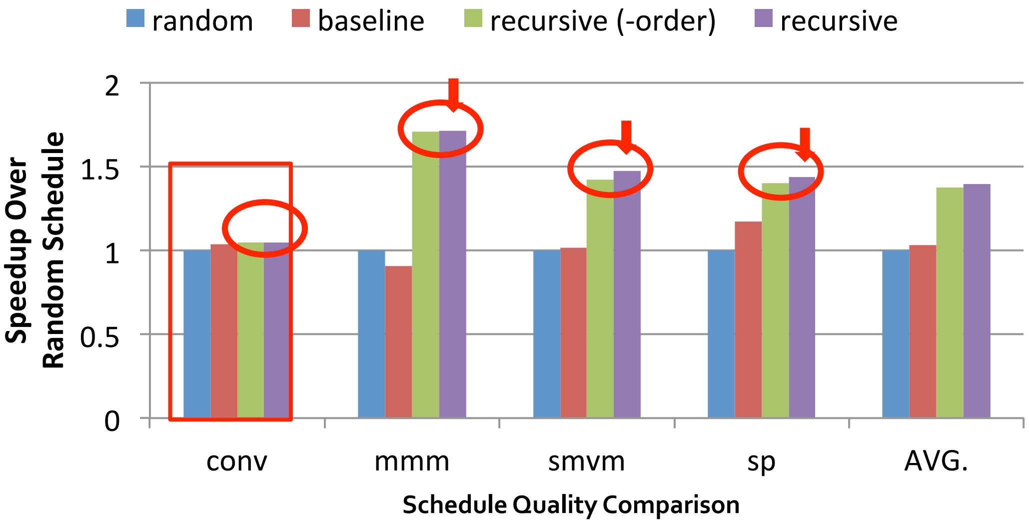
- Disable the ordering component of recursive scheduling
  - Group tasks into L2 granularity + subgroup into L1
  - Use random ordering throughout
- Grouping alone can capture significant locality
  - Ordering provides limited benefit, once task grouping is determined

Single-Level Schedules



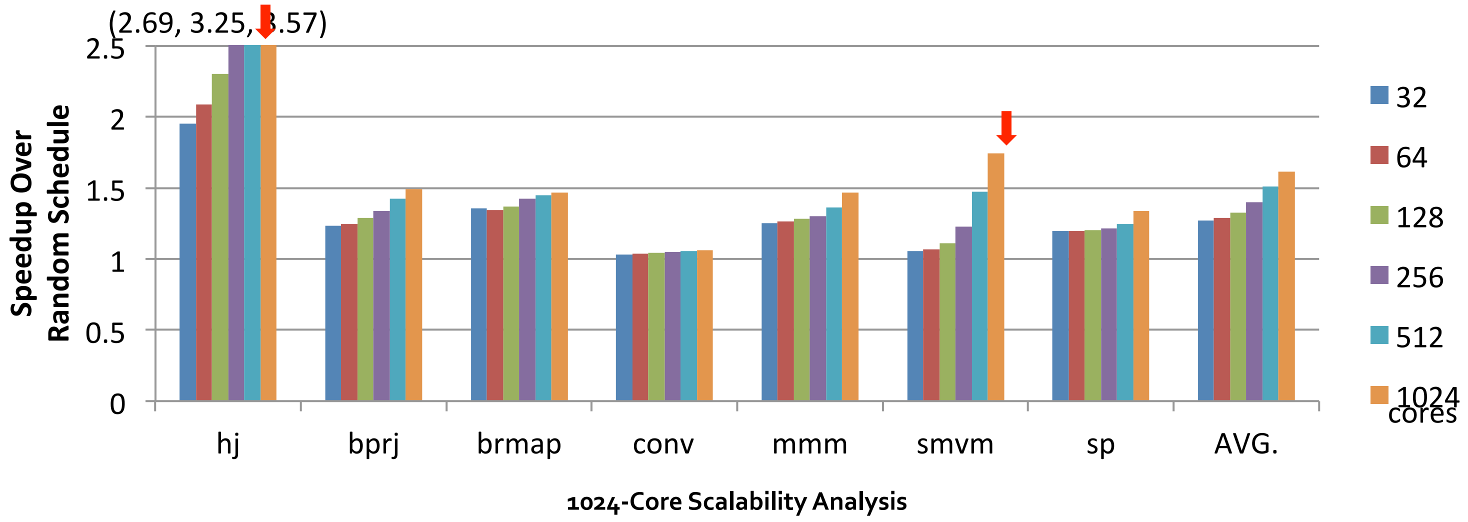
- Perform grouping and ordering at a single level
  - L1 or L2-sized task groups w/ MST task ordering
  - Random ordering across groups
- Due to flat cache hierarchy, single level schedules can capture significant locality

Tiled Processor Results



- Recursive scheduling provides significant performance improvement
  - 1.40x over random, 1.35x over baseline
  - Recursive scheduling can be generically applied to various cache hierarchies
- Unlike the Throughput Processor
  - Improvement due to ordering further diminished
  - Multi-level schedules provided larger gains
- Tree-based hierarchy: should match task group hierarchy to cache hierarchy

Futuristic Processor Results



- Scale # cores from 32 to 1024
  - Compare the performance of recursive schedules against random schedules
- Benefits from locality-awareness increases
  - Average 1.27x speedup (32 cores) to 1.61x speedup (1024 cores)
  - Magnitude depends on the workload pattern (e.g., **hj** and **smvm** w/ L1 sharing)

Summary:
Locality-Aware Task Scheduling

- Large potential exists for locality-aware task scheduling (both performance + energy)
  - All 3 systems benefitted from recursive scheduling
  - Only increases with larger core count

- Guidelines for practical schedulers
  - Grouping tends to be more effective than ordering
  - If only one, implement recursive grouping
  - For (mostly) private hierarchy, single-level schedule at last level is effective

Agenda

- Motivation
- Locality-Aware Task Scheduling
  - Graph-Based Locality Analysis
  - Recursive Scheduling
- Performance Results
- Locality-Aware Task Stealing
- Summary

Locality Analysis of Task Stealing

- Sources of dynamic load imbalance
  - Context switching due to multiprogramming
  - "Intelligent" runtimes share a chip
    - Generate schedules assuming they own the entire chip: *destructive oversubscription*

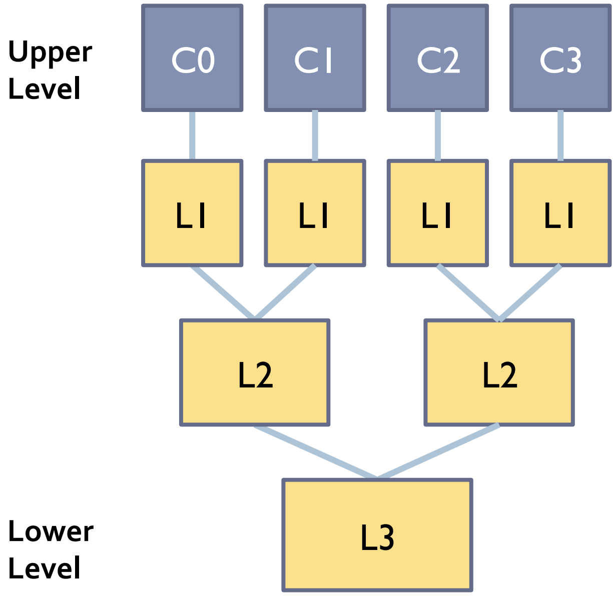
- Task stealing should actively redistribute tasks
  - Locality exploited in the original schedule can be lost

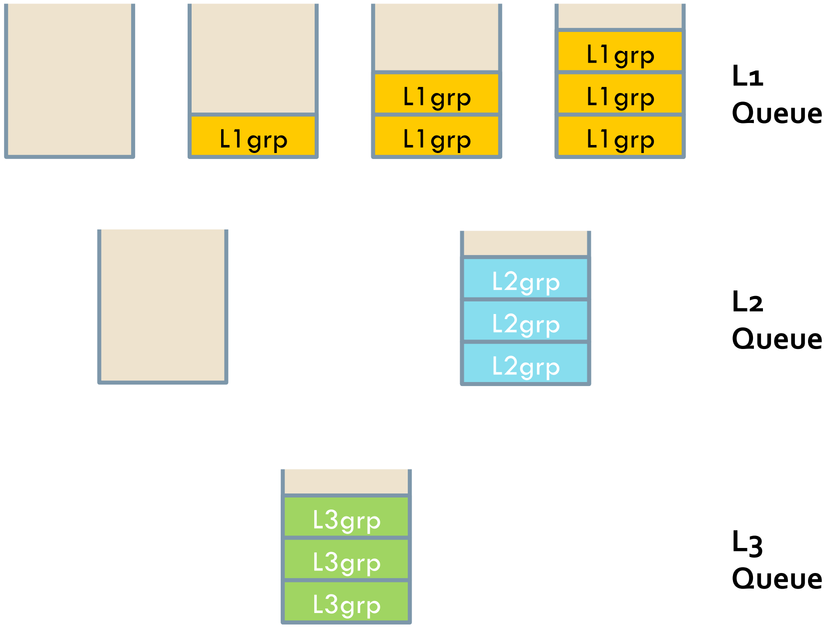
- Exploiting locality while stealing
  - *Preserve* the locality exploited in the original schedule
  - Honor the specified task groupings and orderings

Making Stealing Locality-Aware: *Recursive Stealing*

- Match the queue hierarchy to the cache hierarchy
  - Store order = group order specified by the recursive schedule
  - When transferred across levels, a task group is broken down

- Recursively steal a task group at a time
  - Up to 2.0x task performance improvement for stolen tasks (Throughput Processor)





Agenda

- Motivation
- Locality-Aware Task Scheduling
    - Graph-Based Locality Analysis
    - Recursive Scheduling
- Performance Results
- Locality-Aware Task Stealing
- Summary

Summary

- Provided a systematic approach to exploit locality from unstructured parallel tasks
  - Graph-based analysis framework + offline scheduler
- Limit study demonstrates significant potential
  - Performance/energy benefits on 3 different systems
  - Benefits only increase with larger core count
- Design space exploration
  - Grouping tends to be more effective than ordering
  - For (mostly) private hierarchy, single-level schedule at last level is effective
- Stealing can be made locality-compatible
  - By honoring the original schedule while redistributing