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

Richard M. Yoo[†]
richard.m.yoo@intel.com

Christopher J. Hughes[†]
christopher.j.hughes@intel.com

Changkyu Kim[†]
changkyu.kim@intel.com

Yen-Kuang Chen[†]
yen-kuang.chen@intel.com

Christos Kozyrakis[‡]
christos@ee.stanford.edu

[†]Parallel Computing Laboratory
Intel Labs
Santa Clara, CA 95054

[‡]Pervasive Parallelism Laboratory
Stanford University
Stanford, CA 94305

ABSTRACT

As we increase the number of cores on a processor die, the on-chip cache hierarchies that support these cores are getting larger, deeper, and more complex. As a result, non-uniform memory access effects are now prevalent even on a single chip. To reduce execution time and energy consumption, data access locality should be exploited. This is especially important for task-based programming systems, where a scheduler decides when and where on the chip the code segments, i.e., tasks, should execute. Capturing locality for structured task parallelism has been done effectively, but the more difficult case, unstructured parallelism, remains largely unsolved—little quantitative analysis exists to demonstrate the potential of locality-aware scheduling, and to guide future scheduler implementations in the most fruitful direction.

This paper quantifies the potential of locality-aware scheduling for unstructured parallelism on three different many-core processors. Our simulation results of 32-core systems show that locality-aware scheduling can bring up to *2.39x* speedup over a randomized schedule, and *2.05x* speedup over a state-of-the-art baseline scheduling scheme. At the same time, a locality-aware schedule reduces average energy consumption by *55%* and *47%*, relative to the random and the baseline schedule, respectively. In addition, our 1024-core simulation results project that these benefits will only increase: Compared to 32-core executions, we see up to 1.83x additional locality benefits. To capture such potentials in a practical setting, we also perform a detailed scheduler design space exploration to quantify the impact of different scheduling decisions. We also highlight the importance of locality-aware stealing, and demonstrate that a stealing scheme can exploit significant locality while performing load balancing. Over randomized stealing, our proposed scheme shows up to *2.0x* speedup for stolen tasks.

SPAA'13, June 23–25, 2013, Montréal, Québec, Canada.

Categories and Subject Descriptors

B.8.2 [**Hardware**]: Performance and Reliability—*performance analysis and design aids*; C.1.4 [**Computer Systems Organization**]: Processor Architectures—*parallel architectures*; D.1.3 [**Software**]: Programming Techniques—*concurrent programming*

Keywords

Task Scheduling; Task Stealng; Locality; Performance; Energy

1. INTRODUCTION

Limits on technology scaling highlights parallelism as the means to obtain sustainable performance. More cores are being packed on the same die, and the on-chip cache hierarchies that support these cores are getting larger, deeper, and more complex. As a result, non-uniform memory access (NUMA) effects are now common even on a single chip [4, 21]. Avoiding the high latency to access remote caches and main memory is increasingly critical for performance. The same holds for energy efficiency: Moving a word of data from a remote cache or from an off-chip memory requires 10 and 20 times, respectively, more energy than an arithmetic operation on that word [9]. Consensus exists [14, 7] that memory access locality should be exploited to reduce execution time and energy.

This is especially important for *task-based* programming systems [12, 25, 17, 5, 8, 26], where a computation is broken down into small code segments, *tasks*, and the underlying runtime *schedules* these tasks across threads for execution. Specifically, a scheduler generates a task schedule by *grouping* tasks to execute on the same thread, and by applying *ordering* across the tasks. For load balancing, the runtime may employ *stealing* to redistribute tasks from loaded threads to idle threads. To capture locality on a task-based system, the scheduling algorithm should be locality-aware.

The exact scheduling logic, however, depends on the type of parallelism exposed by the programming system: *structured* or *unstructured parallelism*. For *structured parallelism* (i.e., *task-parallel* programming systems [12, 31]), explicit data or control dependencies exist across tasks, and the runtime can leverage this information to exploit locality. A large body of work on capturing locality for structured parallelism exists [1, 13, 10, 29], and these schemes typically focus on exploiting producer-consumer locality (e.g., schedule consumer task close to producer).

On the contrary, for *unstructured parallelism* (i.e., *data-parallel*

programming systems [25]), for each parallel section, tasks are all independent—they may execute on any thread at any time, and the computation result will still be valid. While they represent a significant class of parallel applications, exploiting locality for unstructured parallelism has been quite difficult: First, the lack of dependency information implies the scheduler must obtain additional information from the workload to synthesize locality structure. Without understanding what the crucial information is, run-time and storage overheads for collecting the information can be significant. Second, the larger degrees of freedom in scheduling increases algorithmic complexity. Having many degrees of freedom implies many grouping and ordering choices, and enumerating all combinations is prohibitive. Third, the complexity of many-core cache hierarchies makes the process all the more complicated. Grouping and ordering decisions must optimize locality across all cache levels, whether the hierarchy being *shared* or *private*.

As a result, capturing the locality of the applications utilizing unstructured parallelism is not well understood. In fact, many runtime systems retrofit the simple scheduling heuristics meant for structured parallelism (e.g., FIFO or LIFO [3]) to unstructured parallelism, and hope that the schedule will capture significant locality. More importantly, little quantitative analysis exists to demonstrate the potential of locality-aware scheduling, and to guide future locality-aware scheduler implementations in the most fruitful direction. While some compiler efforts that map unstructured parallelism onto the cache hierarchy have been reported [18], they (1) apply only to grid-based workloads, and (2) do not address stealing.

The contributions of our paper are as follows: (1) We provide results that **quantitatively demonstrate the potential of locality-aware scheduling** for unstructured parallelism. Specifically, we develop a locality analysis framework and an offline scheduler that takes workload profile information as input and generates schedules that are optimized for the target cache hierarchy. We then evaluate the effectiveness of the scheduler on **three specific many-core cache hierarchies** that represent distinct and very different points in the many-core design space. Our 32-core simulation results verify the importance of locality-aware scheduling, as we observe up to **2.39x** speedup over a randomized schedule and **2.05x** speedup over a state-of-the-art baseline scheduling scheme [6, 22]. By increasing the hit rates in the caches closer to the cores, a locality-aware schedule also saves energy: It reduces the average energy consumption in the memory hierarchy beyond the L1 caches by **55%** relative to the random schedule, and **47%** relative to the baseline. We also perform **1024-core simulations** to verify that the performance advantages of locality-awareness only increase with more cores, and see up to **1.83x** additional performance improvement.

(2) To capture such potentials in a practical setting, we also **perform a scheduler design space exploration** to quantify the impact of different design decisions. In particular, by selectively applying task grouping and ordering, we show that proper grouping alone brings up to 2.23x speedup over a random schedule, using a good ordering gives up to 1.17x additional speedup, and applying grouping and ordering across multiple cache levels gives up to 1.52x speedup over a single-level schedule. These results identify the most crucial information necessary to develop practical locality-aware schedulers.

Additionally, while the conventional wisdom says there exists an inherent tradeoff between locality and load balancing, we **show load balancing can exploit significant locality**. By honoring the task grouping and ordering specified by the schedule as it transfers tasks across threads, our locality-aware stealing implementation reduces task execution time for stolen tasks by up to a factor of **2.0x** over randomized stealing.

Listing 1: **The core task-programming API.**

```
// Initialize task queue library
taskQInit(num_threads, max_tasks);

// Specify the parallel section by providing
// (1) task function and (2) task space
taskQEnqueue(task_fn, num_dims, size_arr);

// Execute the section in parallel
taskQWait();

// Finalize task queue library
taskQEnd();
```

These results also demonstrate that application domains that traditionally refuted dynamic scheduling in favor of locality—e.g., high-performance computing—may employ dynamic task management without losing significant locality. Even for a dedicated system, interference due to shared resources on a large-scale many-core chip (e.g., caches and memory controllers) can introduce significant load imbalance [22], and statically orchestrating all computation and communication will be increasingly challenging.

2. LOCALITY-AWARE TASK SCHEDULING

The key to obtaining the results summarized above is to decouple workload locality analysis and schedule generation, to manage complexity. We first perform a graph-based locality analysis to understand inherent workload locality, and then utilize the analysis results to map computation onto a target cache hierarchy. This decoupling allows to generate schedules for various cache hierarchies using a common framework, and to systematically alter scheduling decisions to perform design space exploration.

2.1 Unstructured Parallelism: API and Workloads

Listing 1 shows the core task-programming API we use in this study, which assumes a task queue-based, software task manager [25, 22]. It shows one parallel section; a program may have many. When the user application invokes **taskQInit()**, the manager spawns worker threads in its thread pool, and creates a task queue for each thread. A user then specifies the parallel section by providing the *task function* and a *task space*—a Cartesian space of integers—to **taskQEnqueue()**. For each coordinate in the task space, the manager bundles the task function with the coordinate to create a task, and schedules it by enqueueing to one of the queues.

Calling **taskQWait()** triggers the parallel execution of tasks. At first, each thread repeatedly dequeues a task from its own queue and executes it by invoking the task function with the coordinate as the argument. When its queue becomes empty, a thread tries to steal tasks from another queue. When the **taskQWait()** function returns, all the tasks have been executed, and the worker threads wait at the pool. Finally, **taskQEnd()** releases the allocated resources.

Note that the task manager assumes no dependencies among the tasks, and may arbitrarily group and order tasks across the queues; therefore, a task should be enqueued only after its dependencies have been satisfied. Since all the tasks in an unstructured parallel section are independent, large number of tasks can be created and enqueued in a single call to **taskQEnqueue()**.

Table 1 summarizes the workloads we ported to the above API. Most of them are real C/C++ workloads originally written to stress test a commercial processor [28]; they are optimized to capture decent intra-task locality. Variants appear in [22, 27, 20] as well. In particular, notice that the applications utilize relatively fine-grained tasks, and are not necessarily organized in a cache oblivious [11] or

Workload	Description	Task	Tasks that share data	Access pattern	# tasks	Task size	Input
hj	Probe phase of hash-join	Performs single row lookup	Look up same hash bucket (clustered sharing)	hash tables	4,096	157	4,460
bprj	Reconstruct a 3-D volume from 2-D images	Reconstructs a sub-volume	Work on sub-volumes mapping to overlapping pixels (clustered sharing)	3-D traversal	4,096	415	3,650
gjk	Collision detection	Operates on sets of object pairs	Work on overlapping object sets (clustered sharing)	pointer access	384	4,950	940
brmap	Map pixels in a 2-D image to another 2-D image	Maps pixels from a sub-image	Work on sub-images mapping to overlapping pixels (structured sharing)	trajectory-based	4,977	1,298	22,828
conv	2-D convolution filter (5x5) on image	Operates on square-shaped image block	Operate on overlapping pixels (structured sharing)	grid-based	1,056	9,487	26,400
mmm	Blocked matrix-matrix multiplication	Multiplies a pair of sub-matrices	Use common sub-matrix (clustered sharing)	grid-based	4,096	49,193	344,064
smvm	Sparse matrix-vector multiplication	Multiplies one row of matrix	Touch overlapping elements in a vector (clustered sharing)	sparse matrix	4,096	891	42,385
sp	Scalar pentadiagonal PDE solver	Solves a subset of equation lattice	Work on neighboring lattices (structured sharing)	grid-based	1,156	5,956	439

Table 1: **Workloads used in this study.** *Task size* **is the average dynamic instruction count.** *Input* **is the sum of all task footprints in KB.**

recursive manner. As discussed, execution order of tasks in these workloads does not affect the computation result.

2.2 Graph-Based Locality Analysis

A locality-aware schedule should map tasks to cores, taking into account both locality and load balance. Two techniques to construct such a schedule are *task grouping* and *ordering*. Executing a set of tasks (a *task group*) on cores that share one or more levels of cache captures data reuse across tasks. Similarly, executing tasks in an optimal order minimizes the reuse distance of shared data between tasks, which makes it easier for caches to capture the temporal locality. Generating a locality-aware schedule depends on understanding how task groups should be formed, and when ordering will matter.

To understand the *inherent* locality patterns of workloads, we develop a *graph-based* locality analysis framework. The framework proceeds as follows: (1) We first profile each workload to collect data access traces at cache line granularity, and discard ordering information to obtain read and write sets for each task. (2) Using the set information, we construct a *task sharing graph*. In a task sharing graph $G(V, E)$, a *vertex* represents a task, and an *edge* denotes sharing. A *vertex weight* is the task size in terms of number of dynamic instructions, and an *edge weight* is the number of cache lines shared between the two tasks connected by the edge. (3) We then partition the graph to form task groups, and observe some metrics to determine the 'right' task group size and the impact of ordering.

Even with profile information about each task's read and write sets, creating an 'optimal' set of task groups is an *NP-hard* problem. We therefore use a heuristic graph partitioning tool (METIS) [19] to generate quality task groups. METIS divides the vertices from a task sharing graph into a given number of groups, while trying to (a) maximize the sum of edge weights internal to each group (i.e., data sharing captured by a task group), and (b) equalize the sum of vertex weights in each group (i.e., balance load).

Table 2 shows the framework output for some of our workloads. Using these results, we try to answer: (1) How should a task group be formed? and (2) When does ordering matter? In addition, we also discuss the implications of task size on locality. For now we discuss locality assuming a single core with a single cache. We later extend the analysis to multiple cores with complex, multi-level cache hierarchies.

Q1: How should a task group be formed?

In Table 2, *sum of footprint* denotes the average sum of individual task footprints for each task group, while *union of footprint* denotes the average size of the union of individual task footprints (i.e., shared lines are counted only once)[1].

Intuitively, to maximize locality, a task group should be formed so that the *working set* of the group fits in cache. In that regard, the sum of footprint represents an upper bound on working set size (i.e., when a schedule fails to exploit any reuse across tasks), and the union of footprint represents a lower bound. For a fully-associative cache, the union of footprint should accurately track the working set. However, due to conflict misses and unaccounted runtime accesses, the actual working set size should be between the union and sum of footprint. Hence, to capture the working set, task groups should be formed so that the cache size falls between the union and sum of task footprints. For example, in Table 2, for **smvm**, when generating a schedule for a 32 KB cache, a task group should contain 8 tasks so that the cache size is between 21 KB (= union of footprint) and 82 KB (= sum of footprint).

However, strictly following this rule may lead to other inefficiencies: For example, grouping too few tasks together might introduce high scheduling overhead, and too many could introduce load imbalance. Such issues can be avoided by performing multi-level grouping, or by executing tasks within a task group in parallel (see Section 2.3).

Q2: When does ordering matter?

We consider both *task ordering* and *group ordering*. *Task ordering* specifies the traversal order of vertices for a task group. Assuming a task queue-based task management system, task order denotes the *dequeue order* of tasks within the same group. *Group ordering* specifies the execution order of task groups. Both task ordering and group ordering can maximize temporal locality when reuse distance is minimized.

To assess when ordering matters, we define two metrics, *sharing degree* and *cut cost*, that dictate the importance of reuse distance on certain data. In Table 2, given a shared cache line, the *sharing degree* denotes the average number of tasks within a task group that share it. The table reports normalized sharing degree, so a degree of 1 means all the tasks within the task group share the cache line.

In terms of locality, sharing degree indicates the potential impact of task ordering. Specifically, a high sharing degree implies that data is shared by a large fraction of tasks. For example, in Table 2a, it can be seen that for task groups that would fit in a 32 KB cache (8 tasks per group), about 90% of the **smvm** tasks within a group

[1]Since read sharing is dominant in our workloads, we give equal weight to read and write sharing in computing our metrics. It is straightforward to assign different weights to reflect different costs for read and write sharing.

Relative task group size	1	1/2	$1/2^2$	$1/2^3$	$1/2^4$	$\mathbf{1/2^5}$	$1/2^6$	$1/2^7$	$1/2^8$	$\mathbf{1/2^9}$	$1/2^{10}$
# tasks / group	4,096	2,048	1,024	512	256	**128**	64	32	16	**8**	4
Sum of footprint (KB)	42,385	21,192	10,596	5,298	2,649	**1,324**	662	331	165	**82**	41
Union of footprint (KB)	4,946	2,618	1,408	757	400	**211**	110	59	34	**21**	14
Sharing degree	0.03	0.03	0.04	0.06	0.11	**0.21**	0.38	0.58	0.75	**0.90**	0.99
Cut cost (E+07)	0.00	2.04	3.13	3.68	3.98	**4.15**	4.27	4.41	4.77	**4.97**	5.08

(a) Statistics collected for **smvm**.

Relative task group size	1	1/2	$1/2^2$	$1/2^3$	$1/2^4$	$\mathbf{1/2^5}$	$1/2^6$	$\mathbf{1/2^7}$	$1/2^8$	$1/2^9$	$1/2^{10}$
# tasks / group	4,096	2,048	1,024	512	256	**128**	64	**32**	16	8	4
Sum of footprint (KB)	3,650	1,825	912	456	228	**114**	57	**28**	14	7	3
Union of footprint (KB)	177	91	51	29	16	**9**	5	**3**	1	1	1
Sharing degree	0.01	0.01	0.02	0.04	0.07	**0.13**	0.22	**0.38**	0.58	0.78	0.92
Cut cost (E+06)	0.00	4.19	6.31	7.37	7.91	**8.19**	8.36	**8.48**	8.61	8.71	8.77

(b) Statistics collected for **bprj**.

Table 2: **Framework output for *smvm* and *bprj*. *Relative task group size* of 1 denotes the case where all the tasks are grouped into a single task group.** ***Sharing degree*** **of 1 means on average all the tasks within the task group share the cache line.**

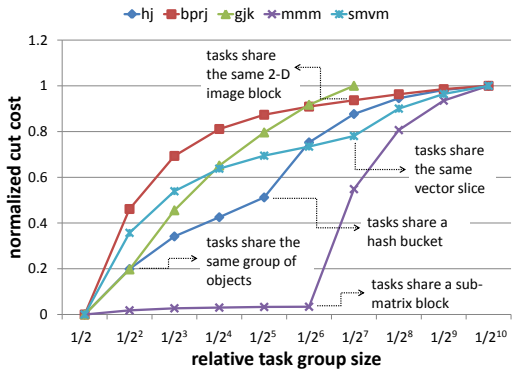


Figure 1: **Cut cost trend over different sizes of task groups. Cut costs are normalized to fit within the interval [0, 1].**

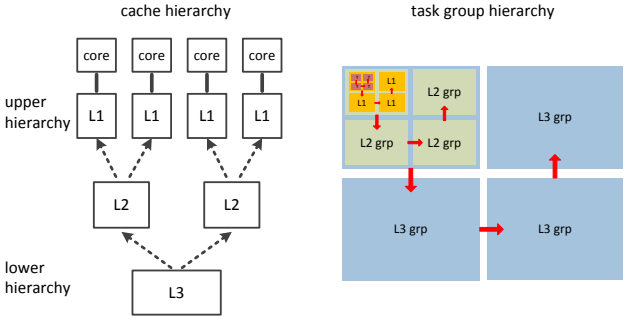


Figure 2: **Generating recursive task groups. Different levels of groups are sized to fit in a particular cache level. Colored arrows denote the group order determined over task groups.**

share any given shared cache line. Therefore, task ordering will have little impact on the reuse distance of shared data. On the other hand, when the sharing degree is low, task ordering could have a significant impact on locality. For **bprj** with 32 tasks per group, the sharing degree is 0.38, which is quite low compared to **smvm**. We can conjecture **bprj** is more sensitive to task ordering.

Next, in Table 2, the *cut cost* represents the sum of the edge weights for vertices in different groups (i.e., data sharing not captured within a single task group). Specifically, our workloads exhibited two distinct cut cost patterns: *clustered sharing* and *structured sharing*—relative importance of group ordering depends on this workload pattern.

Workloads with Clustered Sharing: When a task sharing graph is drawn for these workloads, the graph exhibits disjoint clusters or *cliques* of tasks that share the same data structure. For example, groups of **smvm** tasks share the same vector region, and **hj** tasks that access the same hash bucket.

Figure 1 plots the cut cost trend for the workloads of this type. As can be seen, these workloads exhibit an abrupt increase in cut cost—a *knee*—as we decrease the task group size. For some workloads, other cache lines are sporadically shared, so the knee is less visually striking; we determine knees by manual code analysis.

The sudden increase in cut cost means that the task group size became small enough that tasks sharing their key data structures have been separated into different groups. Ordering those task groups so that they execute consecutively will increase locality. For these workloads, assuming we group tasks so that each group fits in cache, the importance of group ordering depends on the *relative size of the cache to the task clique*. For example, we find that a task clique in **hj** exhibits a 128 KB footprint (where the knee is in Figure 1). On a 32 KB cache, **hj** would benefit from group ordering; on a 256 KB cache, group ordering would be less important.

Workloads with Structured Sharing: These workloads exhibit structured, regular sharing patterns. For example, for **conv**, a 2-D stencil operation, a task shares cache lines with its nearest 4 neighbors in the 2-D task space. For these workloads, cut cost is proportional to the number of groups, and no knee exists. Hence, group ordering is important *regardless of the cache size*; but a simple group ordering, such as assigning consecutive task groups to the same core, should capture reuse.

Now we discuss the implications of task size on a locality-aware schedule. In short, task size indirectly changes the relative importance of task grouping and ordering. For example, when the working set of a single task is larger than the cache, grouping tasks has little benefit, but ordering may help capture reuse from one task to the next. In general, smaller tasks give the scheduler more freedom. If a programmer breaks a large task into smaller tasks, and if the memory access pattern was originally suboptimal, better locality may be achieved via proper grouping and ordering.

2.3 Mapping Computation to Cache Hierarchy: Recursive Scheduling

We now leverage the framework analysis results to map computation onto an actual cache hierarchy. Specifically, we consider *recursive scheduling*, which (1) matches task group working sets and (2) applies ordering across all cache levels. The scheduling logic can be generically applied to arbitrary memory hierarchies; and by selectively applying task grouping and ordering, it can be used to perform scheduler design space exploration (see Section 3).

Creating an optimal order for tasks, however, is also an *NP-hard* problem. We therefore use a heuristic to provide high quality or-

dering. Specifically, we apply Prim's algorithm to construct a maximum spanning tree (MST), and use the order that the vertices are added to the MST. In architectural terms, Prim's algorithm accumulates the read and write sets of scheduled tasks, and picks the task whose read and write sets exhibit the maximum intersection with the cumulative sets as the next task to execute. To construct a task order, we apply MST on a task sharing graph. To construct a group order, we first map the task sharing graph to a *task group sharing graph*, where each *uber node* represents a task group; we then apply MST to the task group sharing graph.

Under recursive scheduling, to maximize the utility of every cache level, we start from the bottommost: We first group tasks so that the task group's working set fits in the last-level cache, and apply ordering over those groups. We then recursively apply this approach to each of the task groups, targeting one level up in the cache hierarchy each time. Figure 2 illustrates the procedure.

In the figure, we first perform grouping on the full set of tasks to create L3 groups, each of which matches the L3 size. Next, we order the L3 groups. For each L3 group, we then decompose it into tasks and create L2 groups to match the L2 size. Then we order the L2 groups. We proceed in this fashion until we finally generate L1 groups, order them, and order their component tasks.

Generating a schedule in this fashion results in a hierarchy of task groups. Moreover, since each task group also denotes a scheduling granularity, all the tasks in a group will be executed consecutively. Therefore, a task group will stay *resident* in its target cache from beginning to end. The existence of a hierarchy among these task groups guarantees that all the groups containing a given task stay resident at their corresponding level in the cache hierarchy, thus exploiting locality across all cache levels.

A slight complication arises when a system has private caches or caches shared by a subset of cores. In such a case, we should *pre-group* sets of task groups, so that groups with high sharing are assigned to the same cache. Specifically, recursive scheduling performs an additional operation whenever hitting a *branch* in the cache hierarchy: It first pre-groups the set of tasks according to the number of consumers one level above, and then performs grouping for each partition. For example, in Figure 2, with two L2 caches, before generating L2 groups it first divides the set of tasks in an L3 group into two partitions, one for each L2 cache. It then constructs a task sharing graph for each partition, and uses the graphs to create two sets of L2 groups. Each set of L2 groups is separately ordered. Due to the pre-grouping, the task schedule generated ensures that the tasks from a set of L2 groups go to the same L2—without this property, group ordering would not be effective. Pre-grouping can be done with the same graph partitioning algorithm that performs task grouping.

3. EVALUATION OF LOCALITY-AWARE TASK SCHEDULING

We now evaluate recursive schedules for specific many-core cache hierarchies that represent distinct points in the many-core design space, to quantify the potential of locality-aware scheduling and perform design space exploration.

3.1 Experiment Settings

As described in Table 3, we simulate three different many-core chip configurations[2]. The first configuration is a throughput computing processor we refer to as *Throughput Processor*. Each core has a private L1 and L2, so all caches are private. The combined L2 capacity is 8 MB, and coherence is maintained through a directory-based protocol. The ISA includes 512-bit SIMD instructions, and the applications have been tuned to use them; simple spatial locality is already captured, and exploiting the remaining locality is more challenging. For this configuration, we use an industrial simulator that models a commercial processor [28].

Core	32 cores; dual issue in-order x86 16-wide 512-bit SIMD extensions Core-private 32 KB 8-way L1, 1-cycle Core-private 256 KB 16-way L2, 14-cycles Directory slice for L2 coherence
Interconnect	Ring network connects L2s, directory slices, and memory ctrls
Memory	4 memory ctrls, 120-cycles

(a) Throughput Processor Configurations

Core	32 cores; dual issue in-order SPARC v9 Core-private 32 KB 4-way L1, 1-cycle
Tile	4 cores per tile Tile-shared 4 MB 16-way banked L2, 10-cycles Directory slice, memory ctrl, and L3 bank
L3	16 MB per bank 16-way, 21-cycles
Interconnect	2-D flattened butterfly connects tiles
Memory	158-cycles

(b) Tiled Processor Configurations

Core	32 to 1024 cores; dual issue in-order x86 Core-private 32 KB 4-way L1, 1-cycle
Tile	1 core per tile Per-tile 512 KB 8-way L2, 12-cycles Directory slice and memory ctrl
Interconnect	2-D mesh connects tiles
Memory	100-cycles

(c) Futuristic Processor Configurations

Table 3: **Simulated system configurations.**

The second configuration is a tiled many-core processor we refer to as *Tiled Processor*. Each core has a private L1, and four cores form a tile. Each tile has a 4 MB L2 shared among the cores on the tile, and all tiles share a single L3 cache. We simulate this with the M5 simulator [2] coupled with the GEMS memory toolset [23].

In addition, to project the potential of locality-aware scheduling as the number of cores continues to increase, we employ a third configuration we refer to as *Futuristic Processor*. Each tile contains a core, core-private L1 and L2, and the tiles are connected through a mesh interconnect. We vary the number of cores from 32 to 1024. We model this configuration with a modified version of the Graphite parallel simulator [24].

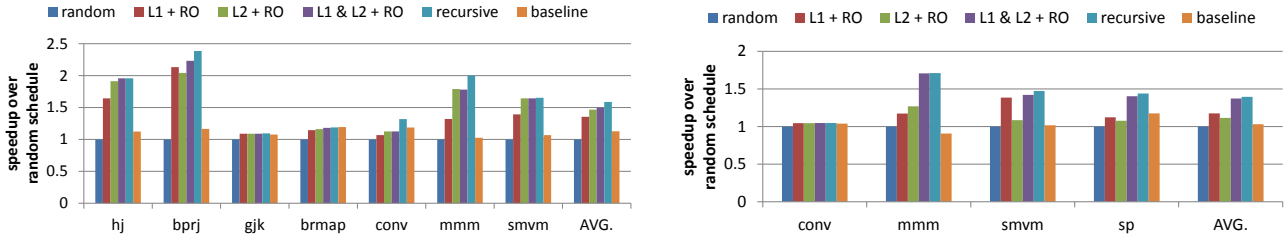
Due to ISA and toolchain issues, we evaluate the Throughput Processor on all benchmarks except **sp** from Table 1, the Tiled Processor on the bottom four benchmarks, and the Futuristic Processor on all benchmarks except **gjk**.

As described in Section 2.1, we utilize a task queue-based software task management system [25, 22, 27]. In particular, task queues are pre-populated with offline-generated schedules right before the start of each parallel section. By default, randomized task stealing [3] is performed across queues for load balancing. We further study the implications of stealing on locality in Section 4.

By evaluating schedules from different scheduling policies, we can quantify the impact of various scheduling decisions on locality. Specifically, in addition to recursive schedules, we evaluate (a) random and (b) baseline schedules. We obtain a random schedule by assigning each task to a random core and then randomly ordering the tasks for each core. We report the averages over 3 instances.

For the baseline schedule, we apply Parallel Depth First (PDF) scheduling [6]. Originally developed for structured parallelism, PDF hinges on the notion that many programs have been optimized for good sequential cache performance. Therefore, when a core completes a task, PDF assigns the task that the sequential program

[2]Instruction caches are modeled in all simulations. However, since we pass function pointer, not the code, to schedule tasks, disruption due to scheduling is minimal.



(a) Throughput Processor Performance

(b) Tiled Processor Performance

Figure 3: **Performance summary. Shows the speedup over a random schedule. For each workload, from left to right are: (1)** *random*, **(2)** *L1 grouping only*, **(3)** *L2 grouping only*, **(4)** *L1 and L2 grouping*, **(5)** *recursive schedule*, **and (6)** *baseline*. **Baseline represents the state-of-the-art PDF scheduling [6], and (2)∼(4) use random ordering (*RO*) instead of MST ordering.**

Workload	# L2 grps	Tasks / L2 grp	Tasks / L1 grp	Sharing degree	L2 cut cost	L1 cut cost
hj	32	128	16	0.91	4.0E+06	7.1E+06
bprj	32	128	32	0.38	8.2E+06	8.5E+06
gjk	32	12	12	0.36	20,010	20,010
brmap	128	39	5	0.53	22,609	61,099
conv	32	33	4	0.55	10,036	27,620
mmm	512	8	1	N/A	1.3E+08	1.4E+08
smvm	32	128	8	0.90	4.2E+07	5.0E+07

Table 4: **Task groups determined by the recursive scheduler and per group statistics. Sharing degree is for L1.**

Workload	L1 MPKI			L2 MPKI		
	random	recursive	baseline	random	recursive	baseline
hj	16.05	**6.63**	13.74	12.64	**6.61**	11.36
bprj	12.93	**3.29**	10.35	10.16	**2.72**	6.91
gjk	8.35	7.55	7.31	8.10	7.39	7.16
brmap	18.66	14.97	14.53	18.48	14.60	14.48
conv	8.54	7.73	7.62	8.37	**5.30**	6.47
mmm	28.37	**17.05**	28.38	27.08	**14.29**	26.07
smvm	136.30	132.22	133.86	52.93	**23.29**	48.46

Table 5: **Measured MPKIs over different schedules. Bold figures denote where recursive schedule improves over baseline.**

would have executed next. Since many parallel programming systems support both structured and unstructured parallelism [25, 5, 8, 26], the same schedule is often applied to unstructured parallelism. For unstructured parallelism, PDF linearizes the task space along the innermost loop, and then evenly divides the tasks into as many chunks as cores, such that consecutive tasks fall in the same chunk. The scheduler then assigns one chunk per core.

Throughout the section, to isolate locality measurements from task management overheads, we use *the sum of the execution time of the tasks* as our primary locality metric. Management overheads can be mitigated through proposed hardware or hybrid methods [22, 20, 27]. Since it uses a simpler cache hierarchy, we focus on the Throughput Processor performance results first. In Section 3.3, we contrast the Tiled Processor results to highlight where different memory hierarchies affect scheduling; in Section 3.4, we project how locality benefits will scale with more cores.

3.2 Throughput Processor Performance Results

In this section, we first summarize the performance and energy benefits of recursive scheduling. Then we isolate the benefits of each feature of the recursive schedule. In particular, we answer the following questions: (1) How much does locality-aware scheduling matter? (2) How much does grouping matter? (3) How much does ordering matter? (4) How does task size affect the schedule? (5) How do single-level schedules compare?

For each workload, Table 4 shows the recursive scheduler's task groups for the Throughput Processor, and the corresponding statistics. Table 2 also highlights L2 and L1 groups.

Q1: How much does locality-aware scheduling matter?

Figure 3a presents the speedup of various schedules over a random schedule, measured in terms of the sum of the execution time of the tasks. For each workload, from left to right, different schedules activate different aspects of grouping and ordering, to arrive at recursive schedule. Here we focus on the performance of recursive schedules; we explain the rest in the following sections. Table 5 reports the measured misses per thousand instructions (MPKI).

The figure shows that a locality-aware schedule (i.e., from the recursive scheduler) improves performance significantly. On average, the speedup over the random schedule is 1.60x, and over the baseline is 1.43x. In particular, **hj**, **bprj**, and **mmm** see large speedups of 1.96x, 2.39x, and 2.00x, respectively. Table 5 shows that this speedup is obtained by improving the behavior at both cache levels, verifying that multiple levels of scheduling is important. **conv** and **smvm** also see significant speedups of 1.32x and 1.65x, respectively, from improved L2 behavior.

While **gjk** and **brmap** are fairly memory intensive (judging from their MPKIs), they see little benefit from locality-aware scheduling. These workloads have simple locality patterns that the baseline schedule is able to capture—grouping consecutive tasks captures most of the locality.

We now compare the energy consumption of different schedules. Specifically, we compute the energy for the part of the memory hierarchy beyond the L1s[3], which includes the L2s, ring network, and memory: For each schedule we measure the total L2 accesses, network hops, and memory accesses, and use the model from [15] to derive energy. Figure 4 shows the results. For each workload, the first three pairs of bars show activity counts—L2 cache accesses, on-die interconnect hops, and memory accesses—and the last pair of bars shows energy consumption. Results are normalized to random schedule.

As expected, locality-aware schedule significantly reduces all three activity counts, and thus the energy consumption: On average, recursive schedule reduces energy by 55% relative to random schedule, and 47% relative to the baseline. Recursive schedules reduce L2 accesses by reducing the L1 miss rate (see Table 5), and likewise decrease on-die network and main memory activity by reducing the L2 miss rate. This shows that locality-aware scheduling, or *placing computation* near where data resides, could be a viable alternative to reducing energy through *migrating data* [16, 15] to the cores performing computation.

Q2: How much does grouping matter?

Recursive scheduling provides benefits through both grouping and ordering. Here, we isolate the benefits of grouping by disabling the ordering and pre-grouping parts of the recursive scheduler. We

[3][30] reports that on the chip level, Intel many-core processors spend 40% of its power on the uncore.

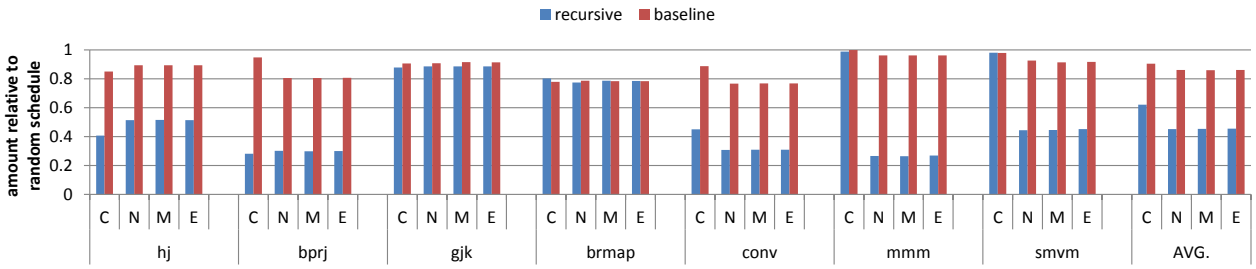


Figure 4: **Energy consumption and activity counts of the memory hierarchy beyond the L1 caches for various schedules.** *C*, *N*, **and** *M* **denotes activity counts for L2 cache accesses, network hops, and memory accesses, respectively.** *E* **denotes the energy consumption.**

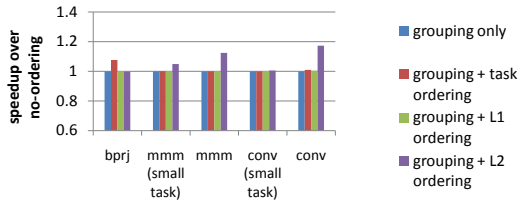


Figure 5: **Workload sensitivity to task, L1 group, and L2 group ordering. Speedup is relative to recursive grouping (with random ordering).**

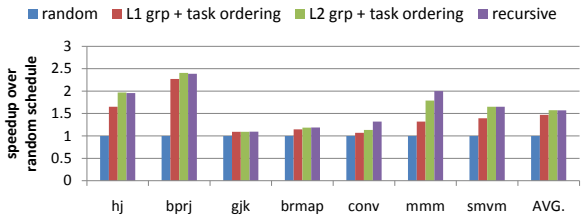


Figure 6: **Single-level schedule performance.**

also isolate the benefits of the two levels of grouping by disabling one of the two levels at a time.

Figure 3a shows the impact of various grouping policies. Compared to random schedules, performing both L1 and L2 grouping with random ordering (L1 & L2 + RO)—*recursive grouping*—provides 1.52x average speedup, capturing most of the benefit of full recursive schedules (1.60x). This shows that grouping captures significant locality, and ordering provides limited additional benefit on top of recursive grouping.

Due to its private-only cache hierarchy, on the Throughput Processor, applying only a single level of grouping can capture much of the locality benefits of recursive grouping: L1 grouping with random ordering (L1 + RO) or L2 grouping with random ordering (L2 + RO) provides 1.36x and 1.49x average speedup, respectively. Nevertheless, recursive grouping helps when different applications favor different cache levels.

Q3: How much does ordering matter?

We first consider ordering alone, and then when it can provide additional benefits over grouping. In a separate experiment where we applied MST ordering to each of the 32 task chunks generated by the baseline schedule, we observed 1.26x average speedup over random. So while ordering by itself does provide performance benefits, these are smaller than those from recursive grouping.

Next, in Figure 3a, comparing the performance of L1 and L2 grouping with random ordering (L1 & L2 + RO) against recursive scheduling shows that three workloads see benefits from ordering on top of grouping—1.07x, 1.17x, and 1.12x speedup on **bprj**, **conv**, and **mmm**, respectively.

Figure 5 shows **bprj**, **mmm**, and **conv** sensitivity to task, L1 group, and L2 group ordering. As can be seen, **bprj** benefits from task ordering; **mmm** and **conv** on the other hand, benefit from ordering L2 groups. L1 group ordering is not as effective.

bprj benefits from task ordering since its first level working set is slightly larger than expected (due to runtime accesses), and its sharing degree is low (see Table 4). With task ordering, L1 MPKI for **bprj** reduces from 4.97 to 2.27, while L2 MPKI reduces from 2.37 to 1.90. **mmm** is a clustered sharing workload—its task groups exhibit affinity for a small number of other groups (see Section 2.2). It also has high L2 cut cost (see Table 4), meaning the affinity between L2 groups is very strong. Therefore, it benefits from ordering L2 groups. **conv** is a structured sharing workload—its task groups exhibit stencil-like affinity. It thus benefits from L2 ordering.

Q4: How does task size affect the schedule?

For regular, grid-based workloads such as **mmm** and **conv**, task size can be easily adjusted by changing blocking parameters. For **mmm**, a single task actually overflows an L1. We shrink each **mmm** task so that a dozen tasks can fit in a single L1 group. Likewise, we make each **conv** task smaller so that it fits in an L1. Figure 5 shows that the sensitivity to L2 ordering reduces for small tasks (we label the modified versions of workloads with small task). The workloads perform the same computation, independent of the task size; hence, the locality to be captured should remain the same. Task size then *alters at which cache level the locality is captured*.

Also, as discussed in Section 2.2, task size can affect performance by changing the scheduler's freedom to exploit locality. For the experiment above, the performance improvement of recursive scheduling over random increased from 2.00x to 3.08x as we decreased **mmm** task size. Since a task group amounts to a *scheduler-determined optimal task size*, users should express their tasks in the finest granularity possible to maximize scheduling freedom. Task scheduling overheads may limit task granularity, but they may be reduced with hardware or hybrid methods [22, 27].

Q5: How do single-level schedules compare?

A single-level schedule denotes performing grouping and ordering at a single level only: i.e., L1 or L2-sized task groups with MST task ordering, but random ordering across groups. Figure 6 compares the performance of L1 and L2 single-level schedules against recursive scheduling. As expected, recursive scheduling provides the best all-around performance. Specifically, for **conv** and **mmm**, neither single-level schedule alone matches the performance of the recursive schedule. For the other workloads, however, an L2 single-level schedule is on par with the recursive schedule—due in part to the flat cache hierarchy and L1 latency hiding through SIMD.

3.3 Tiled Processor Performance Results

In this section, we highlight where different cache hierarchies affect a locality-aware schedule. Specifically, we ported **conv**, **mmm**,

# cores	32	64	128	256	512	1024
Cache-to-cache	124.32	134.39	146.38	166.38	192.18	228.94
Mem-to-cache	184.61	186.70	199.13	208.93	220.83	246.30

Table 6: **Cache-to-cache and memory-to-cache transfer latency of a single cache line.**

smvm, and **sp** to the Tiled Processor[4], and conducted the same set of experiments.

Figure 3b summarizes the results. Similar to the Throughput Processor, recursive scheduling brings about a significant speedup: On average, the speedup over random is 1.40x, and over baseline is 1.35x. This demonstrates that a processor with a shared cache organization has similar potential for locality-aware scheduling.

However, shared caches *alter the relative importance of grouping and ordering*. Similar to Figure 3a, Figure 3b compares different grouping schemes. In contrast to the Throughput Processor where L2 grouping alone (L2 + RO) provided most of the grouping benefits, for the Tiled Processor neither L1 nor L2 single-level grouping (i.e., L1 + RO and L2 + RO) consistently matches recursive grouping. With a complex cache organization, matching the task group hierarchy to the cache hierarchy becomes important.

For this hierarchy, we also see that ordering provides less benefit over grouping—the performance of L1 and L2 grouping with random ordering (L1 & L2 + RO) is very similar to recursive schedule. In particular, **conv** and **mmm**, which exhibit sensitivity to ordering on the Throughput Processor, now barely benefit from ordering. This can be attributed to the increased importance of recursive grouping: Recursive grouping amounts to applying *coarse ordering* over smaller groups, limiting the benefits of additional ordering.

As on the Throughput Processor, the benefits of ordering alone (i.e., applying MST to each task chunk generated by the baseline) are significant, but smaller than grouping alone: 1.15x speedup over random, compared to 1.37x.

3.4 Futuristic Processor Performance Results

As we add more cores on a processor die, the size of the on-die network increases, which results in larger access latencies for remote cache and memory. To quantify the impact of core scaling on locality-awareness, we vary the number of cores on the Futuristic Processor from 32 to 1024, and compare the performance of random and recursive schedules. Workload inputs were re-adjusted to fully utilize up to 1024 cores.

Table 6 first shows the measurements from a pointer-chasing microbenchmark; a producer core populates each cache line-wide entry of a list with a pointer to the next entry, then the consumer core chases the chain of pointers. The table reports the average latency to transfer a cache line from (1) one core's L2 cache to another core's L1, and (2) memory to an L1, as we increase the core count. Home nodes of the cache lines are spread uniformly, and the cache-to-cache transfer is between the cores farthest apart. As can be seen, remote cache access latency increases from 124 cycles at 32 cores to 229 cycles at 1024 cores. For the same configuration, the memory access cost increases from 185 cycles to 246 cycles.

Such an increase in latency in turn amplifies the impact of locality-aware scheduling. Figure 7 shows the speedup of recursive schedules over random schedules across varying core counts. With 32 cores, recursive schedule provides 1.27x average speedup over random (1.20x speedup over the baseline). As the number of cores increases, the benefit of locality-awareness increases across all workloads (at 1024 cores, 1.61x average speedup over random).

However, the exact degree depends on the workload locality pattern. In particular, for **hj** and **smvm**, which exhibit high L1 sharing degree (see Table 4), random schedules' poor task grouping generate many cache-to-cache transfers; and as the transfer latency increases, give recursive schedules significant performance advantage (compared to 32-core executions, 1.83x and 1.65x additional locality benefits with 1024 cores, respectively).

[4]Vector instructions were replaced by scalar loops.

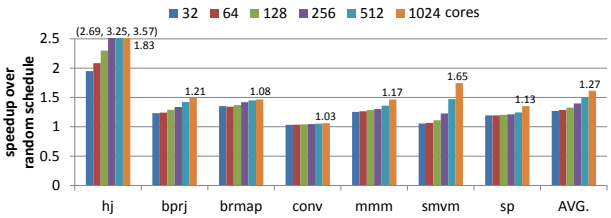


Figure 7: **Performance scalability up to 1024 cores. At each core count, speedup is over a random schedule. For each workload, the number on the rightmost bar denotes the additional locality benefits of 1024-core execution when compared to 32-core.**

3.5 Summary: Guidelines for Practical Locality-Aware Task Schedulers

Locality-aware task scheduling can provide significant performance and energy efficiency improvements for unstructured parallelism, both on private and shared cache organizations. The importance of locality-awareness will only increase with larger core count. The relative importance of task grouping and ordering, however, is a function of the workload and the underlying cache hierarchy. Nevertheless, if a locality-aware scheduler were to implement only one scheme, it should be recursive grouping—recursively matching task group working set size across all cache levels. For a processor with (mostly) private cache hierarchy, a single-level schedule at the last-level cache can capture most of the locality.

4. LOCALITY-AWARE TASK STEALING

Dynamic task management comprises two components: (1) task scheduling, or initial assignment of tasks to threads, and (2) task stealing, or balancing load by transferring tasks from a loaded thread to an idle thread. In Sections 2 and 3, we explored locality-aware task scheduling. Here, we explore locality-aware task stealing.

4.1 Motivation

Intuitively, task stealing will benefit most from being locality-aware when many tasks are stolen. One major source of large load imbalance is multiprogramming: Software threads from applications compete for hardware contexts, and potentially large number of tasks may be stolen from a switched-out thread. Even for a dedicated system, interference due to shared resources on a many-core chip (e.g., caches and memory controllers) can introduce significant load imbalance [22].

Previously proposed stealing schemes, however, are *locality-oblivious*. The most widely adopted scheme, *randomized stealing* [3], chooses a victim at random, and steals one or more tasks (stealing multiple amortizes stealing overheads). It provides good characteristics such as even load distribution and theoretically bounded execution time, but its randomness renders it inherently locality-oblivious. In fact, if the task schedule is also locality-oblivious, we expect locality-oblivious stealing to have little impact on cache behavior. However, for a locality-aware schedule, this stealing policy may significantly decrease the performance of stolen tasks.

We verify this by inducing large amounts of task stealing. Specifically, we emulate context switching: After producing a task schedule for 32 threads (on the Throughput Processor), we offline a sub-

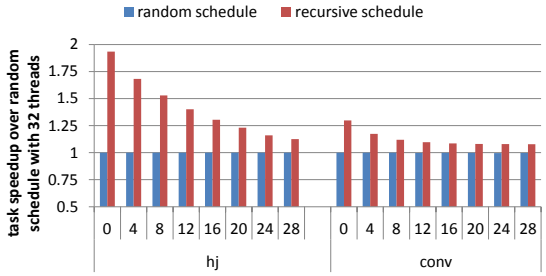


Figure 8: **Impact of locality-oblivious stealing on a locality-aware schedule. The numbers along the x-axis denote the number of threads offlined.**

set of the threads, and rely on stealing to redistribute tasks from the offlined threads.

Figure 8 shows the task performance trend of a random schedule and a recursive schedule for two workloads, normalized to the performance of a random schedule when no threads are offlined. We use the same randomized stealing policy for both schedules—it randomly selects a victim and tries to steal half of the victim's queue with a prescribed upper bound (an empirical value of 8 was used [22]). If it fails to steal anything, it visits the other potential victims in a round-robin fashion.

The case where no threads are offlined is the same data presented earlier. However, the benefit of recursive scheduling decreases as more threads are offlined, since the locality captured in the schedule is disrupted by randomized stealing. On the other hand, the performance with a random schedule is independent of the number of threads offlined. When tasks are scheduled in a locality-aware fashion, *locality-aware stealing* becomes important.

4.2 Locality Analysis of Task Stealing

We approach task stealing using a similar analysis methodology we used for scheduling. For this discussion, we assume a locality-aware task schedule created from a recursive scheduler. We explore the impact on locality of the two key design decisions for task stealing: (1) *Which* tasks to steal? and (2) *How many* tasks to steal?

Q1: Which tasks to steal?

Random victim selection fails to capture the locality between the tasks that have already executed and those that are to be stolen. Assuming a multi-level memory hierarchy, it would be the best if such locality is exploited through the highest-level cache (i.e., L1). If no such tasks are available, victim tasks should be chosen among those that will give sharing through the next level (i.e., L2), and so forth. In essence, a thief should look for tasks in a top-to-bottom fashion, so that *stealing scope* gradually increases as we lower the cache level where sharing will take place.

Q2: How many tasks to steal?

The other locality to consider is the locality among the stolen tasks. A natural steal granularity that would provide good locality among victim tasks is a task group—after all, this is how recursive scheduling constructs groups. Stealing a task group at a time amortizes steal overheads, as well.

Stealing a *fixed amount or portions* of tasks each time may under- or overshoot a task group boundary, to break the group. Stealing an already-stolen task (i.e., *secondary stealing*) breaks locality within stolen task groups as well; secondary stealing from a group with strong internal sharing, e.g., an L1 group, may impair performance.

On the other hand, the level of task group stolen affects load balancing. Stealing a coarser granularity task group preserves more locality among stolen tasks, but could increase load imbalance, assuming we prohibit secondary stealing. One way to emulate the

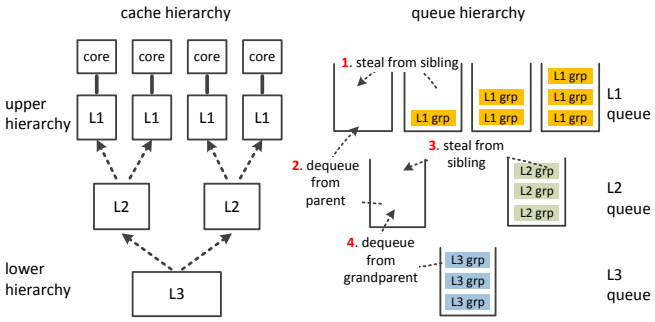


Figure 9: **Recursive stealing. The top-level queues (that hold tasks) are not shown. Colored numbers indicate the order that the leftmost core visits queues.**

locality of stealing a coarser task group while maintaining flexibility is to steal smaller groups but enforce *steal ordering*: If a thread steals again, it follows the specified group order.

4.3 Making Stealing Locality-Aware: Recursive Stealing

In this section we present a reference locality-aware stealing scheme, *recursive stealing*. Our discussion so far suggests that (1) stealing scope should recursively expand through the memory hierarchy, and that (2) stealing should be performed at the granularity of task groups. Figure 9 illustrates the scheme.

We maintain a hierarchy of queues, where a queue exists for each cache; these queues may be implemented as software or hardware components. A queue at a specific level holds task groups that fit in the cache for that level: an L3 queue holds L3 groups, an L2 queue holds L2 groups, etc. The order these task groups are stored reflects the *group order* determined by the recursive schedule. Not shown are the queues that hold actual tasks; for the example hierarchy, one task queue exists per L1 queue. Once a task group is dequeued and moved to an upper-level queue, it is logically broken down into upper-level groups. For example, when an L2 group is transferred to an L1 queue, it is decomposed into L1 groups.

To exploit as much locality as possible from the original recursive schedule, recursive stealing *interleaves* regular dequeues and steal operations. In our example (see Figure 9), tasks are replenished as follows. Once a task queue is empty, a thread attempts to dequeue from its L1 queue; if the L1 queue is empty, it attempts to steal from the sibling L1 queue (i.e., before it tries a regular dequeue from the L2 queue). We interleave steals with dequeues in this example because the L2 caches are shared: If a thread steals from a sibling L1 queue, it grabs an L1 group that shares data in the L2 cache with (a) the tasks it just executed, and (b) the tasks the sibling core(s) are currently executing; thus, we exploit the shared cache. If the L2 queue is empty as well, it climbs down the hierarchy and repeats the process: It attempts to steal from the sibling L2 queue, and then visits the L3 queue. When stealing, a thread grabs the task group at the tail of the victim queue, in the same way randomized stealing operates [3].

In addition, we do not allow stealing across task queues: The minimum steal granularity is an L1 group, and a stolen L1 group cannot be stolen again. For our workloads, this does not impose significant load imbalance, since a typical L1 group has 4 to 8 tasks.

In essence, recursive stealing exploits locality through two features: (1) by performing *recursive victim selection* to exploit locality across potentially multiple levels of shared caches, and (2) by stealing at minimum a whole L1 group to guarantee locality among the stolen tasks.

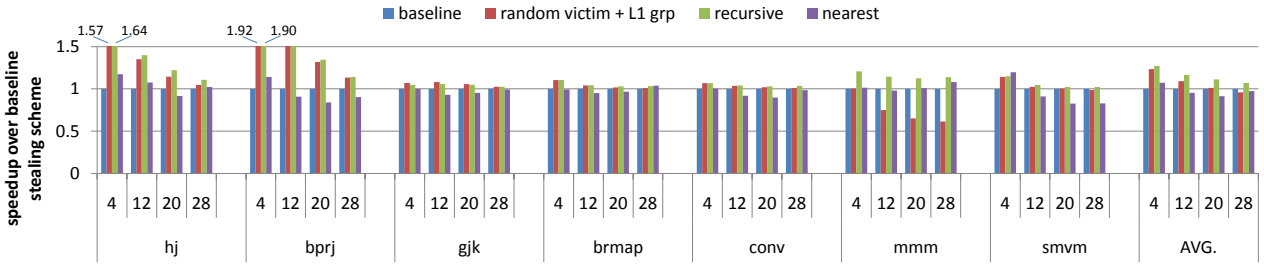


Figure 10: **Performance improvement of stolen tasks.** *baseline*, *recursive*, **and** *nearest* **represents randomized, recursive, and nearest-neighbor stealing, respectively.** *random victim + L1 grp* **disables the recursive victim selection in recursive scheduling. The numbers along the x-axis indicate the number of threads offlined.**

4.4 Performance Results

We first present the performance summary, and then isolate each feature of recursive stealing. In particular, we answer the following questions: (1) How beneficial is locality-aware stealing? (2) How much does victim selection matter? (3) How much does steal granularity matter?

Q1: How beneficial is locality-aware stealing?

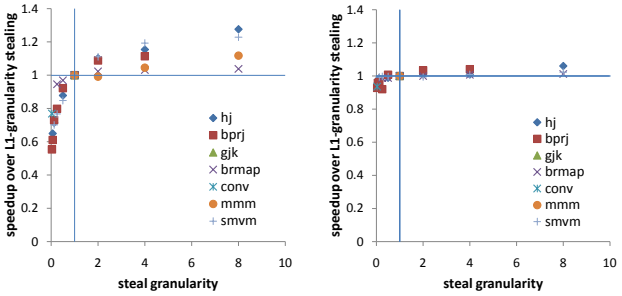
We implemented the recursive stealing scheme in Section 4.3 as a software library for our Throughput Processor configuration, and compared its performance against the baseline randomized stealing (see Section 4.1). For a given workload, we use the same recursive schedule in all experiments.

Figure 10 compares the performance of *stolen* tasks over various stealing schemes, as we offline some threads. Looking at the average, recursive stealing provides benefit across all numbers of offlined threads, but the average benefit decreases as more threads are offlined. When 4 threads are offlined, the average speedup over randomized stealing is 1.27x. The reason for the speedup decrease is that in general, executing the tasks on fewer cores (i.e., spreading an application's data across fewer caches) naturally captures more sharing. This is especially true for applications with small working sets. For this case, there is less potential for improving locality.

When we look at individual workload performance, we can see that those workloads that benefit the most from recursive scheduling, i.e., **hj**, **bprj**, and **mmm**, significantly benefit from recursive stealing. **hj** and **bprj**'s L1 groups are significantly larger than 8 tasks (the upper bound for the baseline stealing scheme), so stealing an L1 group at a time gives significant locality boost. When 28 threads are offlined, recursive stealing reduces the L1 miss rate by 1.28x and 1.39x, respectively. For **mmm**, however, an L1 group contains only a single task. Recursive stealing exploits locality through the L2 instead, and reduces the L2 miss rate by 1.56x (with 28 threads offlined). This verifies that both steal granularity and victim selection contribute to the benefits of recursive stealing.

smvm presents an interesting case. The workload benefits significantly from recursive scheduling, but the performance improvement due to recursive stealing is not as profound. By coincidence, random stealing's upper bound of 8 tasks matches the number of tasks within an L1 group. However, the baseline performs worse due to secondary stealing—**conv** behaves similarly. On the other hand, for **gjk** and **brmap**, which exhibit sharing through consecutive tasks, maintaining the task order is good enough to preserve most of the locality.

The source of randomized stealing's poor behavior is not that it chooses its victims at random, but that its victim selection is locality-oblivious. To demonstrate this, in Figure 10, we also show the performance of a nearest-neighbor stealing scheme. This scheme is the same as the baseline, except a thief always chooses its nearest-neighbor as the first victim. As can be seen, the scheme, which is also locality-oblivious, performs as poorly as randomized stealing.



(a) High contention (4 threads off, 28 threads on)

(b) Low contention (28 threads off, 4 threads on)

Figure 11: **Application sensitivity to steal granularity. Vertical and horizontal lines denote x = 1 and y = 1, respectively.**

Q2: How much does victim selection matter?

To isolate the benefits from recursive victim selection, we implement a stealing scheme which (1) selects a victim at random, but (2) steals an L1 group at a time. The random victim + L1 grp in Figure 10 shows its performance.

Most of our applications see the same performance from the random victim + L1 grp policy as recursive stealing. However, for workloads exhibiting strong sharing through L2, recursive victim selection is able to capture the locality. **mmm** shows this effect strongly, since its L1 group amounts to a single task—locality must be captured across L1 groups. In particular, as more threads are offlined, random victim selection exhibits deteriorating performance; offlining more threads means more victims to choose from, making it more likely that random victim selection fails to capture locality. While the baseline also uses random victim selection, it steals 8 tasks at a time, and captures some locality through L2.

Q3: How much does steal granularity matter?

Intuitively, if load balance is not an issue, stealing larger chunks of tasks will better exploit locality. Conversely, stealing a smaller number of tasks will fail to preserve the locality specified in the schedule. However, we find that sensitivity to steal granularity is regulated by the degree to which thieves contend over victim tasks.

In Figure 11, the x-axis denotes normalized steal granularity, where steal granularity of 1 denotes the case when a single L1 group is stolen at a time. The y-axis is the performance of a stolen task, normalized to the same case. When multiple L1 groups are stolen, they are stolen in an atomic fashion, and are not subject to secondary stealing.

Figure 11a shows the performance under high contention. In this configuration, 4 threads are offlined, and 28 online threads compete over the tasks assigned to the 4 offlined threads. When the contention is high, it becomes hard to preserve locality across multiple steal operations from the same thread. Therefore, performance is relatively sensitive to steal granularity. Specifically, we can see that

reducing steal granularity below an L1 group results in a significant loss in locality, since the strong sharing within an L1 group is now broken. Conversely, increasing the steal granularity beyond a single L1 group gives sizeable locality improvements.

On the contrary, Figure 11b shows performance under low contention, when only 4 threads are online. These threads rarely contend over the tasks originally assigned to the 28 offlined threads, and recursive stealing improves inter-steal locality by preserving group order (see Section 4.2). In fact, recursive stealing effectively collects smaller task groups to emulate the effect of executing a larger granularity group. As a result, sensitivity to steal granularity is much smaller than the high contention case.

4.5 Summary

To preserve the locality exploited in task schedules while load balancing, task stealing should be made locality-aware. Two types of locality need to be captured: (a) locality between the tasks that have executed and that are to be stolen, and (b) locality among the stolen tasks. By adhering to the task grouping and ordering specified by the original schedule while transferring tasks, a stealing scheme can be made locality-compatible.

5. CONCLUSION

This paper provides a quantitative analysis of exploiting task locality for unstructured parallelism. Through a graph-based locality analysis framework and a generic, recursive scheduling scheme, we demonstrate that significant potential exists for locality-aware scheduling. Specifically, our simulation results of three distinct 32-core systems show significant performance improvement (up to 2.05x over a state-of-the-art baseline) and energy reduction (47% average reduction from the baseline). In addition, 1024-core simulation results project that with an increasing number of cores, benefits from locality-awareness will only increase (up to 1.83x additional benefits compared to 32-core executions). To capture this potential, we also explore the scheduler design space in detail. While we find the performance contributions of different scheduling decisions to be the function of the workload and the underlying cache hierarchy, matching the task group hierarchy to the cache hierarchy provides the most benefit. We also highlight the importance of locality-aware stealing when the tasks are scheduled in a locality-aware fashion, and demonstrate that a recursive stealing scheme can effectively exploit significant locality while load balancing (up to 2.0x speedup over randomized stealing).

6. ACKNOWLEDGMENTS

Richard Yoo was supported in part by a David and Janet Chyan Stanford Graduate Fellowship. This work was supported in part by the Stanford Pervasive Parallelism Laboratory.

7. REFERENCES

[1] Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. The data locality of work stealing. In *Proc. of the 12th SPAA*, pages 1–12, 2000.

[2] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[3] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. of the 35th Annual Symposium on Foundations of Computer Science*, pages 356–368, 1994.

[4] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *Proc. of the 8th OSDI*, pages 43–57, 2008.

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proc. of the 20th OOPSLA*, pages 519–538, 2005.

[6] Shimin Chen, Phillip B. Gibbons, Michael Kozuch, Vasileios Liaskovitis, Anastassia Ailamaki, Guy E. Blelloch, Babak Falsafi, Limor Fix, Nikos Hardavellas, Todd C. Mowry, and Chris Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *Proc. of the 19th SPAA*, pages 105–115, 2007.

[7] Computing Community Consortium. 21st century computer architecture: A community white paper. 2012.

[8] Cray. Chapel Language Specification 0.796, 2010.

[9] William Dally. The future of GPU computing. In *the 22nd Annual Supercomputing Conference*, 2009.

[10] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proc. of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[11] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. *Annual IEEE Symposium on Foundations of Computer Science*, 0:285–297, 1999.

[12] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proc. of the 1998 PLDI*, pages 212–223.

[13] Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proc. of the 2010 PPoPP*, pages 341–342.

[14] Mark Hill and Christos Kozyrakis. Advancing computer systems without technology progress. In *DARPA / ISAT Workshop*, 2012.

[15] Christopher J. Hughes, Changkyu Kim, and Yen-Kuang Chen. Performance and energy implications of many-core caches for throughput computing. *Micro, IEEE*, 30(6):25–35, 2010.

[16] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. *IEEE TPDS*, 18:1028–1040, 2007.

[17] Intel. Threading building blocks, `http://www.threadingbuildingblocks.org`.

[18] Mahmut Kandemir, Taylan Yemliha, Sai Prashanth Muralidhara, Shekhar Srikantaiah, Mary Jane Irwin, and Yuanrui Zhang. Cache topology aware computation mapping for multicores. In *Proc. of the 2010 PLDI*, pages 74–85.

[19] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. In *Proc. of the 24th International Conference on Parallel Processing*, pages 113–122, 1995.

[20] John H. Kelm, Daniel R. Johnson, Matthew R. Johnson, Neal C. Crago, William Tuohy, Aqeel Mahesri, Steven S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *Proc. of the 36th ISCA*, pages 140–151, 2009.

[21] Changkyu Kim, Doug Burger, and Stephen W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. *IEEE Micro*, 23:99–107, 2003.

[22] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proc. of the 34th ISCA*, pages 162–173, 2007.

[23] Milo M. K. Martin, Daniel J. Sorin, Bradford M. Beckmann, Michael R. Marty, Min Xu, Alaa R. Alameldeen, Kevin E. Moore, Mark D. Hill, and David A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33:92–99, 2005.

[24] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proc. of the 16th HPCA*, pages 1–12, 2010.

[25] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 3.1, 2011.

[26] Oracle. The Fortress Language Specification Version 1.0, 2008.

[27] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proc. of the 15th ASPLOS*, pages 311–322, 2010.

[28] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, pages 18:1–18:15.

[29] Janis Sermulins, William Thies, Rodric Rabbah, and Saman Amarasinghe. Cache aware optimization of stream programs. In *Proc. of the LCTES 05*, pages 115–126.

[30] Avinash Sodani. Race to exascale: Opportunities and challenges. In *the 44th Annual IEEE / ACM International Symposium on Microarchitecture*, 2011.

[31] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proc. of the 11th CC*, pages 179–196, 2002.