Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages

ANDI DREBES, Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, France
ANTONIU POP, University of Manchester, School of Computer Science
KARINE HEYDEMANN, Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, France
ALBERT COHEN, INRIA and École Normale Supérieure
NATHALIE DRACH, Sorbonne Universités, UPMC Univ Paris 06, CNRS, UMR 7606, LIP6, France

We present a joint scheduling and memory allocation algorithm for efficient execution of task-parallel programs on non-uniform memory architecture (NUMA) systems. Task and data placement decisions are based on a static description of the memory hierarchy and on runtime information about intertask communication. Existing locality-aware scheduling strategies for fine-grained tasks have strong limitations: they are specific to some class of machines or applications, they do not handle task dependences, they require manual program annotations, or they rely on fragile profiling schemes. By contrast, our solution makes no assumption on the structure of programs or on the layout of data in memory. Experimental results, based on the OpenStream language, show that locality of accesses to main memory of scientific applications can be increased significantly on a 64-core machine, resulting in a speedup of up to 1.63× compared to a state-of-the-art work-stealing scheduler.

Categories and Subject Descriptors: D.1.3 [**Concurrent Programming**]: Parallel Programming

General Terms: Performance, Languages

Additional Key Words and Phrases: FIFO queue, dynamic scheduling, work stealing, lock-free algorithm, weak memory model, dataflow programming, Kahn process network

ACM Reference Format:
Andi Drebes, Karine Heydemann, Nathalie Drach, Antoniu Pop, and Albert Cohen. 2014. Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages. ACM Trans. Architec. Code Optim. 11, 3, Article 30 (August 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2641764

1. INTRODUCTION

As the number of cores increases, shared memory machines now rely on complex, non-uniform memory architectures (NUMA) to reduce contention on memory controllers. Main memory is distributed over multiple *nodes* connected through large-scale coherent links. This distribution incurs high access cost for data stored on remote nodes. Performance thus highly depends on the exploitation of node affinity—that is, avoiding

This work was partly supported by the European FP7 projects PHARAON id. 288307 and TERAFLUX id. 249013.
Authors' addresses: A. Drebes, K. Heydemann, and N. Drach, 4 Place Jussieu, 75252 Paris Cedex 5, France; email: andi.drebes@lip6.fr; A. Pop, University of Manchester, School of Computer Science, Oxford Road, Manchester M13 9PL, United Kingdom; email: antoniu.pop@manchester.ac.uk; A. Cohen, Département d'Informatique, École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France; email: albert.cohen@inria.fr.

DOI: http://dx.doi.org/10.1145/2641764

expensive remote accesses by favoring accesses to local memory. Taking into account the topology of the target machine is a key enabling factor to efficient execution.

Efficiently exploiting NUMA is a nontrivial task. In addition to the optimization challenges for single-core memory hierarchies, new challenges arise, such as making efficient use of the interconnect bandwidth and avoiding contention [Dashti et al. 2013]. These problems can be addressed by locality-aware data and task placement [Best et al. 2011], keeping data as close as possible to the processing core. Several studies have shown that the execution time of independent, coarse-grained tasks can be reduced considerably by using schedulers that exploit the topology of the underlying architecture [Blagodurov et al. 2010; Zhuravlev et al. 2012]. Jiang et al. [2008] formally show that finding optimal co-schedules for more than two threads is already NP-complete. It is thus likely to be computationally intensive for many-core machines, and practical approaches are therefore based on heuristics.

Parallel programming using fine-grained tasks has become increasingly popular to harness the computing resources of large-scale parallel machines. In contrast to application-unaware schedulers and memory allocators, which are designed to work with independent processes, heuristics integrated into the parallel runtime system can take advantage of additional information, available during execution, such as the relationship between tasks and data. Such information is either implicitly assumed, as is the case for Cilk tasks [Blumofe et al. 1995] where data locality between sibling tasks in the task tree is a consequence of the frequent reliance on divide-and-conquer programming patterns, or explicitly provided by the programmer through code annotations, or derived automatically through static and dynamic analysis (profiling).

We present a resource-aware approach to jointly schedule and allocate memory for fine-grained tasks executing on large-scale, shared memory multicore machines. The main contribution of our algorithm resides in the way that it exploits locality and dependence information readily available in runtimes of task-parallel languages, and more specifically in runtimes of languages with support for explicit, point-to-point, intertask dependences. As a representative of such languages, we use OpenStream, an extension of OpenMP with dataflow tasks [Pop and Cohen 2013].

Information about intertask dependences and data reuse is preserved during compilation. It is thus available to the runtime system, capturing *accurate* information on data transfer between tasks. This allows the runtime to determine at execution time which processing units are contributors of data to any given task. Combining this information with dynamic information about data placement on NUMA nodes enables the scheduler to decide heuristically which core is the best candidate for execution of a task. In particular, this choice depends on which NUMA nodes are likely to contain the highest amount of task input data as well as the location of the task's output buffers.

A NUMA-aware data allocation mechanism gives the runtime fine-grained control over data placement and allows optimization of data placement in the course of the execution. Similar to the scheduler, the decision where data should be allocated can be driven by information about task dependences and the location of already allocated data. This way, data accesses can be optimized *before* execution of a task.

Our experiments show that the locality of data accesses of representative parallel benchmarks is increased significantly when exploiting data dependence and reuse information both by the scheduler and the memory allocator. For memory-bound benchmarks, increased locality translates into a significant reduction of execution time.

1.1. Outline

The presentation of our approach takes three successive steps. We first introduce a data- and communication-centric migration mechanism that triggers when all dependences conditioning the execution of a given task have been satisfied. This mechanism transfers the task to the core that fits best for execution for a given data placement. The

goal is to constantly react to the effective location of data, matching data locality with task ownership. Then, we introduce a modification of the scheduler itself, adapting the random work-stealing algorithm [Blumofe and Leiserson 1999] to prioritize steals from task queues of nearby cores, further reducing memory access latency.

Second, we propose a topology-aware and dependence-aware memory allocation mechanism that optimizes data placement before task execution. By analyzing producer–consumer relationships between tasks, this method prevents data regions for input and output data from being allocated on distant NUMA nodes and thus promotes short-distance read and write accesses.

Third, we study the reciprocal effects of the scheduling and allocation techniques and show that a joint approach using all of the proposed optimizations at once performs best.

The remainder of this article is organized as follows. First, we precisely define the problem that our approach solves and its requirements. In Section 2, we discuss complete random work stealing as well as two modifications of the scheduler with improved locality. Section 3 discusses the relationship between memory allocation and data locality and presents a dependence-aware allocation mechanism. Reciprocal effects of optimized scheduling and memory allocation are discussed in Section 4. Section 5 details the embedding of our optimizations into a modern task-parallel language. After a presentation of the evaluation methodology in Section 6, a quantitative evaluation of the implementation is given in Section 7. The article closes with a discussion of the most closely related work in Section 8, followed by a conclusion and an outlook in Section 9.

1.2. Problem Statement

Our objective in this work is to improve the performance of task-parallel applications executing on modern NUMA systems by reducing the latency of memory accesses through improved locality.

The behavior of parallel applications can depend on runtime parameters, such as the problem size and input data, but also on small differences in timing between runs, leading to different schedules. It can therefore be extremely difficult or even impossible to predict behavior based on static information available at compile time. Therefore, optimizations taking into account dynamic changes must operate at execution time. Our approach addresses the problem of optimization for memory hierarchies of parallel programs by providing an efficient task transfer mechanism, a topology-aware task scheduling algorithm, and a topology-aware and dependence-aware memory allocation mechanism designed to meet four major optimization goals:

(1) *Efficient load-balancing across cores* aims at distributing work over the cores of the machine to achieve high parallelism. Ideally, all processing units efficiently contribute to the execution of the parallel program.
(2) *Efficient load-balancing across memory controllers and interconnects* targets elimination of contention on individual memory controllers and interconnects by spreading data on different NUMA nodes.
(3) *Taking advantage of locality relative to NUMA nodes* consists in scheduling tasks on cores near the data processed during task execution, reducing the number of remote memory accesses and pressure on interconnects.
(4) *Optimized, active data placement* aims at allocating memory on NUMA nodes favorable for local memory accesses according to data exchanges of future computations.

Some of the goals are contradictory, and it might be difficult to meet all of them at once. For example, a set of communicating tasks generates traffic on the interconnect if executed on cores of different NUMA nodes, but executing all of them on the same node decreases parallel performance and increases contention on a single memory controller. Depending on behavior at runtime, a trade-off might thus be required.

Moreover, the optimization approach should be transparent to applications and also be portable across different machines and their memory hierarchies. Ideally, it adapts automatically to the target architecture without any manual intervention of the application programmer. We achieve this by using a runtime-based approach using a lightweight description of the memory hierarchy, which parametrizes the scheduler and allocator. It can either be provided by the system administrator or manufacturer or be generated automatically.

2. WORK STEALING AND LOCALITY

Randomized work stealing was originally designed as the load-balancing scheduler for the Cilk language for shared memory multiprocessors [Frigo et al. 1998]. Due to its advantages [Blumofe and Leiserson 1999], work stealing has been adopted in various parallel libraries and parallel programming environments, including the Intel TBB and compiler suite. Work-stealing variants have also been proposed for distributed clusters [Gautier et al. 2007] and heterogeneous platforms [Augonnet et al. 2011]. The scheduling strategy is intuitive:

—Each core uses a dynamic array as a deque holding tasks ready to be scheduled.
—Each core manages its own deque as a *stack*. It may only push and pop tasks from the `bottom` of its own deque.
—Other cores cannot push or pop from that deque; instead, they steal tasks from the `top` when their own deque is empty. The target deque for stealing is selected at random.
—Initially, one core starts with the *root* task of the parallel program in its deque, and all other deques are empty.

The state-of-the-art algorithm for the work-stealing deque is Chase and Lev's lock-free deque [Chase and Lev 2005] using an array with automatic, asynchronous growth.

The main difference in the execution of task-dependent programs, compared to more classical task-parallel models, is that tasks can be created before their dependences are satisfied. Such tasks are not yet ready to execute and therefore remain unknown to the scheduler.

A task may depend on data produced by an arbitrary set of tasks, each of which may produce an arbitrary amount of its input data. To keep track of these dependences, each task has a synchronization counter. Its value represents the amount of data missing before the task becomes ready to be scheduled. When a task generates input data for another task, the consuming task's synchronization counter is decreased accordingly, and when it reaches zero, the task becomes ready and is pushed into the deque of the worker that performed the last decrementation.

In this section, we first characterize two major weaknesses of randomized work stealing with respect to the locality of data accesses. In a second part, we present two complementary scheduling heuristics that improve data locality: (1) *work pushing* bypasses the work-stealing scheduler by transferring ownership of tasks based on effective runtime data locality; and (2) *topology-aware work stealing,* which attempts to steal tasks in a worker's incrementally widening neighborhood based on an abstract description of the memory hierarchy.

2.1. Randomized Work Stealing

In randomized work stealing, the target deque for stealing is selected based on a uniform distribution. Although this mechanism yields good global load balancing between workers, it only achieves average and in some cases even poor data locality. This is partially due to the mismatch between task and data placement, as well as a mismatch

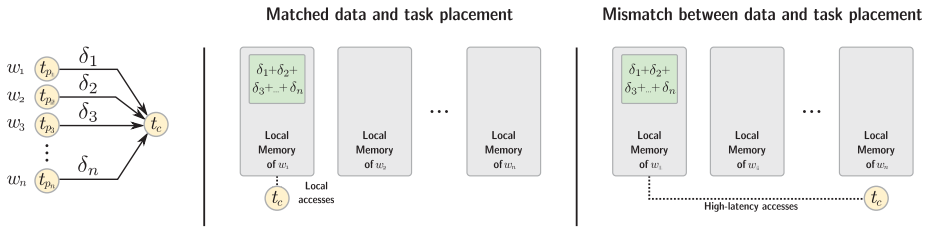

Fig. 1. Remote memory accesses due to mismatch between task and data placement.

between the machine's nonuniform topology and the uniform choice of target in the default work-stealing heuristic.

Mismatch between task and data placement. Consider a consumer task t_c whose input data is generated by n producer tasks $t_{p_1}, \ldots, t_{p_n}$ executed by workers $w_1, \ldots, w_n$, as shown at the left side of Figure 1. Independently from the number of input dependences, a consumer has a single task buffer in which *all* of its input data is stored. In this model, there are no producer-specific output buffers. Instead, the output buffers of a producer are the input buffers of its consumers. Hence, in the example, each of the workers writes a number $\delta_1, \ldots, \delta_n$ of bytes to t_c's input buffer and decreases the synchronization counter accordingly. When it reaches zero, t_c becomes ready and is added to the scheduler deque of the worker that satisfied the last dependence. Depending on the execution order of $t_{p_1}, \ldots, t_{p_n}$, this can be *any* of the workers $w_1, \ldots, w_n$.

Assume that t_c's buffer is located in the local memory of w_1. If w_1 satisfies the last dependence, t_c is added to the work deque of w_1 with matching task and data ownership. This scenario is illustrated in the center of Figure 1. However, assuming that all of the workers $w_1, \ldots, w_n$ or a large subset of workers execute on different nodes, the probability that t_c is placed in a deque of a worker on the same node as w_1 is small. Hence, it is unlikely that data and task ownership match, and situations such as those illustrated on the right side of Figure 1 with mismatched task and data placement are more frequent. In this case, input data has to be fetched from a remote site at execution of t_c, leading to high-latency, long distance data transfers and reduced performance of the application.

Steals from distant workers. Even if the data for all of the tasks in a worker's task deque is locally available on its node, random selection for a task steal leads to bad locality of data accesses: as the number of workers executing on a same node is smaller than the total number of workers, the probability of stealing a task from a worker on the same node is lower than to steal from a *remote* core.

Hence, solving these problems can have a significant impact on performance. The mismatch problem is addressed by a task transfer mechanism presented in the following section. A topology-aware victim selection strategy for steals is presented in Section 2.3.

2.2. Work Pushing

To reduce the mismatch between task and data placement, a task should be transferred to a worker whose node contains the task's input data. In the remainder of this article, the task transfer mechanism is referred to as work pushing. Deciding which worker satisfies this condition requires precise dataflow information, which is readily available in dataflow programming models like OpenStream. In particular, it is necessary to know which data will be read by a task *before* that task executes.

Efficient lock-free work-stealing deques [Chase and Lev 2005] cannot be used to remotely push tasks without changing the algorithm and incurring high synchronization

costs. As a solution to this problem, we propose to add a new work-sharing mechanism based on a multiproducer, single-consumer FIFO queue (MPSC FIFO). In addition to the work deque, each worker is provided with such an MPSC FIFO for task transfers.

ALGORITHM 1: last_dep_satisfied(w, t)

```
n_w ← local_node_of_worker(w)
n_t ← node_of_task_buffer(t)

if n_t ≠ n_w then
    w_dst ← random_worker_on_node(n_t)
    res ← push_back(w_dst.mpsc_fifo, t)

    if res = failure then
        insert_deque(t, w)
    end
else
    insert_deque(t, w)
end
```

ALGORITHM 2: empty_mpsc_fifo(w)

```
import ← true

while import = true do
    t ← pop_front(w.mpsc_fifo)

    if t ≠ null then
        insert_deque(t, w)
    else
        import ← false
    end
end
```

Algorithm 1 shows how a worker w, discovering that a task t is ready for execution, transfers the task to another worker if necessary. The identifier of the worker's local node is determined by calling *local_node_of_worker*. The result is assigned to n_w. The node containing the task's buffer is stored in n_t. If the nodes are different, data and task ownership must be restored by invoking the work-pushing mechanism. The target worker w_{dst} is selected randomly among the workers operating on n_t. The actual transfer is performed by a call to *push_back*, trying to insert the task into the target worker's MPSC FIFO. If the transfer fails (e.g., if the target MPSC FIFO is full), the task is simply added to the work deque of w as if no task transfer were performed at all.

Tasks received by the target worker cannot be scheduled as long as they are in the MPSC FIFO. Algorithm 2 shows the procedure *empty_mpsc_fifo* that transfers these tasks to the local work deque to make them available for execution. Its instructions are repeated each time before the worker selects a new task for execution. The transfer is very simple: while the MPSC FIFO is not empty, the front element is removed and added to the work deque. Using this order has an important side effect. The front of the MPSC FIFO contains the oldest tasks, whereas the back holds the most recent tasks. Thus, during the last iteration of the loop, the most recent task is added to the work deque and becomes the next task to be executed by the worker. Input data of the most recent task has the highest probability to be still present in parts of the memory hierarchy near to the executing core. The last task from the MPSC FIFO is therefore a good candidate for execution.

2.3. Topology-Aware Work Stealing

Before we discuss the topology-aware work-stealing algorithm, we present a lightweight static model for the representation of the memory hierarchy, which is used by the work-stealing algorithm to adapt to the topology of the target machine. The description can be broken down to the following parts:

—A set $C \subset \mathbb{N}$ of identifiers for processing units (e.g., $C = \{0, \ldots, 63\}$).
—An ordered set L containing the levels of the memory hierarchy from the cache nearest to the CPUs down to the different NUMA domains or memory controllers (e.g., $L = \langle L1, L2, L3, RAM \rangle$).

—A function $sibs : L \times C \to \mathbb{N}$ describing how many processing units share an instance of a hardware part at a given level. We refer to these processing units as siblings. For example, if four cores with identifiers 8, 9, 10, and 11 share a third-level cache, then $sibs(L3, 8) = sibs(L3, 9) = sibs(L3, 10) = sibs(L3, 11) = 4$. Passing a processing unit as a parameter to *sibs* allows the definition of asymmetric architectures.
—A function $nth\_sib : L \times C \times \mathbb{N} \to C$ describing which processing unit is the n-th sibling of another processing unit in ascending order of CPU identifiers at a given level. For example, if processing units 0 to 7 share a third-level cache, then $nth\_sib(L3, 0, 2) = 2$ and $nth\_sib(L3, 3, 2) = 5$.

By using L, *sibs*, *nth_sib,* and the order relation on L, the neighbors of a core at the different levels of the memory hierarchy can be determined easily for topology-aware work stealing.

The idea behind our optimized heuristic for work stealing is that instead of randomly selecting a steal victim, task deques of neighboring workers are favored, which leads to more local memory accesses when the stolen task is executed. However, the work deques of close workers might not always provide enough tasks to steal. To avoid poor load balancing, other attempts at higher levels in the memory hierarchy must be performed if work stealing fails on close deques.

Our topology-aware work-stealing technique is shown in Algorithm 3. At each level l beginning with the level nearest to the CPU, a number of steal attempts defined by $attempts(l)$ is performed until an attempt is successful or no level is left. In addition to the definitions of the memory hierarchy, the algorithm uses the following notations and functions:

—$rand(n)$ generates a random integer value in $[0; n]$ using a uniform distribution.
—$cpu : W \to C$ returns the processing unit a worker executes on with W being the set of workers.
—$attempts : L \to \mathbb{N} \cup \{0\}$ defines the maximal number of steal attempts at a given level of the memory hierarchy.

ALGORITHM 3: topology_aware_stealing(w)

```
cpu_w ← cpu(w)

for level ∈ L do
    num_siblings ← sibs(l, cpu_w)

    for attempt ← 1 to attempts(l) do
        n ← rand(num_siblings − 1)
        target_cpu ← nth_sib(l, cpu_w, n)

        if target_cpu ≠ cpu_w then
            target_worker ← cpu^−1(target_cpu)
            t ← steal(target_worker)

            if t ≠ null then
                return t
            end
        end
    end
end
return null
```

Fig. 2. Locality of write accesses in a chain of dependent tasks. (a) Tasks with buffers allocated on alternating nodes. (b) Task buffers allocated on the same node.

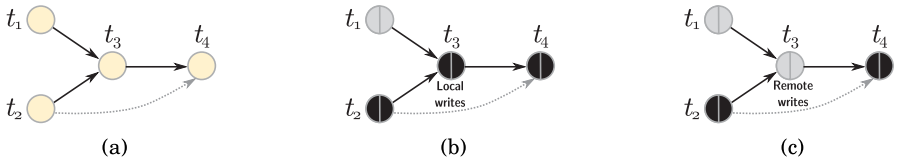

Fig. 3. (a) Indirect task creation. (b) Local allocation with local accesses. (c) Local allocation with remote accesses.

—*steal* : $W \to T \cup \{null\}$ is a function that performs a steal attempt on a target deque and returns the stolen task from the set of tasks T if the attempt is successful or null if the attempt fails.

2.4. Locality of Write Accesses

As work pushing operates with information about placement of task buffers and ignores intertask dependences, it can only optimize locality of *read accesses* of a task to its own buffer. *Write accesses* from a producer to a consumer's buffer remain unaffected by the optimization. This is illustrated in Figure 2. To visualize task and data ownership, each task is represented by two colored semicircles. The color of the right semicircle indicates the location of the task's buffer, whereas the color of the left semicircle shows the node to which the executing CPU belongs. For read accesses, ideally both colors match, indicating that accesses to input data are local as in Figure 2(a). However, write accesses in this example are entirely remote. A better placement consists in placing all task buffers on the same node as shown in Figure 2(b), where read *and* write accesses are local. In the next section, we show that the problem of remote write accesses can be addressed by the memory allocator.

3. MEMORY ALLOCATION AND LOCALITY

In this section, we study the relationship between buffer allocation and locality of data accesses. We first show why allocation on local nodes can result in poor data locality of write accesses. We then propose a dependence-aware memory allocation scheme that improves the locality of write accesses when used in conjunction with work pushing.

3.1. Local Allocation

As the write accesses during execution of a producer task address the task buffers of its consumers, the addresses of all consumer task buffers must be known before execution of the producer. Hence, a producer cannot be ready for execution before the buffers of the receiving tasks have been allocated. As a consequence, a task cannot allocate buffers of tasks that *directly* depend on it. Dynamic task graphs, however, require that buffers are allocated in the course of execution of the application, as new tasks are created dynamically. To circumvent the preceding restriction, tasks create indirect successors as shown in Figure 3(a). In the example, a direct dependence between t_3 and t_4 requires t_4 to be created by a predecessor of t_3 (e.g., t_2 at the bottom left).

By default, task buffers of new tasks are allocated on the local node of the allocating task. In the example of Figure 3(a), this leads to the allocation of t_4 on the node of t_2. Depending on the effective location of the task buffer of t_3, work pushing causes t_3 to be executed on the node of t_2 (Figure 3(b)) or on some other node, such as the one of t_1

ALGORITHM 4: allocate_task_buffer(w, t)

```
for n ∈ Nodes do
  | δ_n ← 0
end

δ_max ← 0
n_max ← null

for t_p ∈ producers(t) do
  | n_p ←
  |   node_of_task_buffer(t_p)
  | δ_{n_p} ← δ_{n_p} + δ(t_p, t)
  |
  | if δ_{n_p} > δ_max then
  |   | δ_max ← δ_{n_p}
  |   | n_max ← n_p
  | end
end

if n_max ≠ null then
  | n_dst ← n_max
else
  | n_dst ←
  |   local_node_of_worker(w)
end

buf ←
  alloc_on_node(n_dst, δ_max)
return buf
```

(Figure 3(c)). In the first case, write accesses of t_3 are local, and as t_4 is also executed on the same node, accesses to input data of t_4 are also local. In the other case, write accesses of t_3 are remote.

3.2. Dependence-Aware Allocation

The main issue of local allocation is the coupling of task placement and buffer allocation: the effective location at execution of the allocating task determines where the task buffer of a future task is allocated.

For local write accesses of tasks on a dependence path, buffer allocation and task execution should be performed on the same node. Hence, the location of a task's producers, not the location of the allocating tasks, should determine where its task buffer should be allocated. The procedure of Algorithm 4, allocating the task buffer of a new task t, achieves this by exploiting information about data dependences available to the runtime. The task buffer of t is allocated on the same node as the task buffers of its producers. If the producers are scattered across multiple nodes, the node with maximal contribution to t's input data is selected for allocation.

A variable δ_n counts for each node n how many bytes of input data will be written by producers of t whose task buffers are allocated on n. To compute this value, the algorithm iterates over the set of producers and then determines for each producer t_p the node n_{t_p} containing its task buffer and adds $\delta(t_p, t)$, the number of bytes produced by t_p and consumed by t, to δ_{n_p}. The node with maximal contribution is $n_{\max}$, producing $\delta_{\max}$ bytes of input data. The task buffer for t is finally allocated on this node. If no such node exists (e.g., if t does not have any input dependence), the buffer is allocated locally.

3.3. Implications for Different Dependence Patterns

For chain-like dependence patterns, as seen earlier in Figure 2, dependence-aware allocation causes the buffers of all tasks in the chain to be allocated on the same node. Placements favoring local writes are also achieved for asymmetric dependence patterns—that is, patterns containing tasks with multiple predecessors or successors producing or consuming different amounts of data. We refer to dependences associated with larger amounts of data in these patterns as *strong* dependences and those associated with fewer data as *weak* dependences. Dependence-aware allocation causes all buffers of tasks on paths with strong dependences to be allocated on the same node. This is illustrated in Figure 4, with thick lines indicating stronger dependences and thin lines indicating weaker dependences. Most read and write accesses of these tasks are local if they are executed by workers belonging to the buffers' node using work pushing.

For symmetric dependences, where all data exchanges are of the same size, the node for buffer allocation must be chosen among several maximal contributors. It is less likely that buffers on long dependence chains are allocated on the same node,

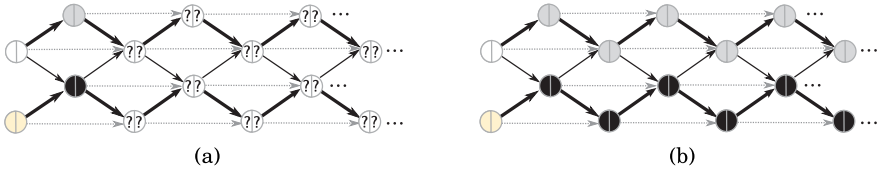

Fig. 4. Dependence-aware allocation keeps buffers of tasks with strong dependences on the same node. (a) Initial placement. (b) Resulting buffer placement.

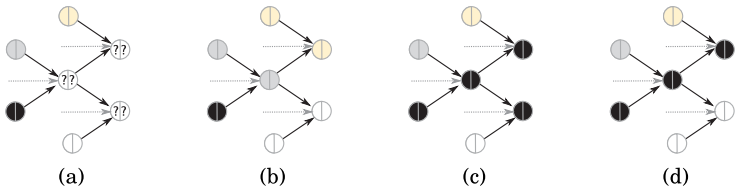

Fig. 5. Dependence-aware allocation resulting in improved locality of reads and writes for direct dependences. (a) Initial placement. (b–d) Possible buffer placements resulting from dependence-aware allocation.

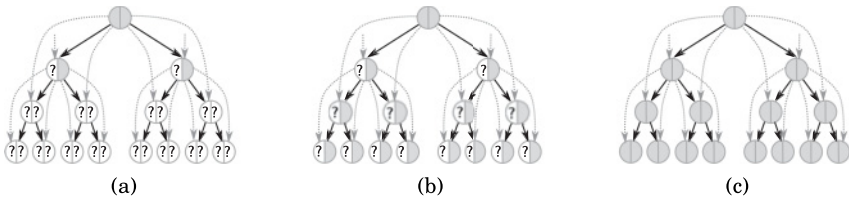

Fig. 6. Dependence-aware allocation and work pushing applied to a tree-like dependence pattern. (a) Initial placement. (b) Effects of dependence-aware allocation. (c) Final result due to work pushing.

but the algorithm guarantees that at least one input dependence of each task results in local writes for the associated producer. Figure 5 illustrates this property of the algorithm. Any selected maximal contributing producer in the initial placement at the left guarantees the locality of the write accesses of at least one dependence.

Allocating buffers of dependent tasks on the same node leads to higher locality of data accesses, but work pushing might diminish load balancing across computing units and memory controllers. An extreme case is tree-like dependence patterns, resulting in poor data distribution and low parallelism.

Figure 6 shows such a scenario. In the initial situation, the root task of the tree and its buffer are placed on the gray node. Dependence-aware allocation reserves the buffers of its children on the same node. This process repeats at each level of the tree, such that the buffers of the remaining tasks of the tree are allocated on the same node as well (Figure 6(b)). Work pushing transfers the tasks to workers of the gray node, resulting in poor load balancing (Figure 6(c)).

Consider Figure 7, which shows the same initial situation as in Figure 6 with the exception that one of the root's child tasks is stolen by a worker from a nearby node. Dependence-aware allocation still causes all buffers to be allocated on the same node, but repeated work stealing below the stolen task causes more tasks to be executed on the other node, leading to better load balancing. However, the memory accesses are remote, both for reads and writes, and pressure on the memory controller may be high.

Therefore, the dependence-aware allocation algorithm needs to react to steals: if the task calling the allocation function was obtained by work stealing, the task buffer of the newly created task is allocated locally—that is, without taking into account any

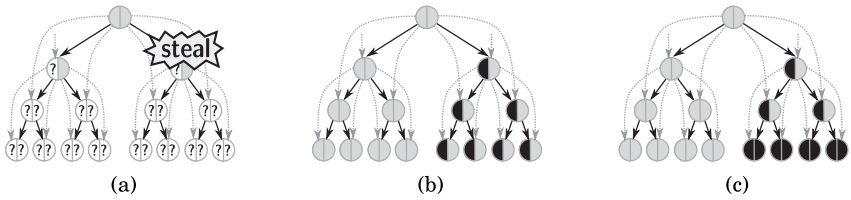

Fig. 7. Dependence-aware allocation and work pushing applied to a tree-like dependence pattern. (a) Initial placement with steal. (b) Resulting allocation and execution pattern. (c) Result for modified dependence-aware allocation.

task dependence. For the tree structure, this leads to a situation similar to Figure 7(c). Data accesses of the task that was stolen initially and those of its children are remote, but data locality can be restored for newly created tasks. As a consequence, global load balancing is improved, there are fewer task transfers, and contention on memory controllers decreases.

4. JOINT TASK SCHEDULING AND MEMORY ALLOCATION

The reciprocal effects of work pushing, topology-aware work stealing, and modified dependence-aware allocation can be summarized as follows:

—Work pushing restores the mismatch of task and data ownership explained in Section 2.1. It also acts as a complementary mechanism to dependence-aware allocation, exploiting the locality of data placement by transferring tasks to nodes containing their data.
—Dependence-aware allocation optimizes task buffer placement according to future data exchanges between tasks based on information about task dependences and the effective location of the corresponding task buffers.
—Topology-aware work stealing extenuates possible computational load-balancing and contention problems induced by dependence-aware allocation and work pushing. Remote read and write accesses are avoided by attempting to steal tasks from workers of the same node before stealing from remote nodes.

Altogether, these characteristics match the goals defined in Section 1.2:

(1) *Efficient load-balancing across cores* is achieved with topology-aware work stealing with an incrementally widening neighborhood ensuring *global* computational load balancing.
(2) *Efficient load balancing across memory controllers and interconnects* is implemented with work stealing and dependence-aware allocation.
(3) *NUMA node-relative locality* is addressed by dependence-aware allocation and work pushing.
(4) *Optimized, active data placement* is achieved through dependence-aware allocation, reducing the number of remote write accesses.

In Section 7, we show that the impact of these heuristics on locality and performance in a real-world hardware and software environment is significant. The next section describes how they were integrated into a runtime system for task-parallel applications with point-to-point dependences.

5. EMBEDDING INTO A TASK-PARALLEL LANGUAGE

OpenStream [Pop and Cohen 2013] is a dataflow programming language designed as an incremental extension to OpenMP. It allows expression of arbitrary dependence patterns between tasks in the form of task-level dataflow dependences. OpenStream allows

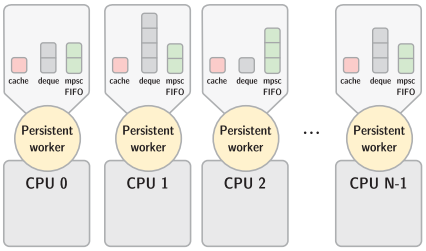

Fig. 8. OpenStream's per-worker data structures and worker placement.

exploitation of task, pipeline, and data parallelism. Programmers expose task parallelism through language annotations (pragmas), providing the compiler with intertask dependence information. These dependences are used to generate code that dynamically builds a graph of dependent tasks communicating and synchronizing through unbounded FIFO streams, implemented using task buffers as seen in the previous sections. Writes to streams result in writes to the buffers of the tasks consuming the data. Read accesses to streams by consumer tasks are translated to reads from their task buffers.

We have implemented the optimizations presented in this article into the OpenStream runtime. The language is compiled with the publicly available implementation[1] in GCC 4.7.1. Two important aspects of OpenStream are particularly relevant to the techniques studied here:

(1) OpenStream programs rely on programmer annotations, which make the flow of data between tasks explicit. This precise dataflow information is preserved during compilation and easily accessible in the runtime library. It allows efficient determination, at runtime, and *before* task execution, of how much data is exchanged by any given task, therefore enabling our work-pushing and dependence-aware allocation optimizations.
(2) The OpenStream scheduler uses a state-of-the-art implementation [Lê et al. 2013] of the Chase and Lev [2005] work-stealing deque. It originally relied on a randomized work-stealing policy.

In the OpenStream execution model, tasks whose dependences have been satisfied are executed by persistent worker threads. In addition to a local work-stealing deque, each worker has a single-entry software cache possibly containing a task ready to be executed as shown in Figure 8. If work pushing is enabled, each worker also owns an MPSC FIFO dedicated to the reception of remote tasks. In our experiments, we have placed one persistent worker on each core.

The software caching scheme is used to hide one task from theft attempts for a short time interval, between the moment the current task executed on the worker completes and the time where the worker needs more work. Indeed, when a worker completes a task, it decreases the synchronization counters of all tasks that depend on it, which can lead to some of these consumer tasks becoming ready to execute. New ready tasks are added either to the local work-stealing deque or to other workers' work-pushing FIFO queues. The most recently created task not pushed to another worker is kept locally in the software cache. If all ready tasks not transferred to another worker are added to the work-stealing deque, it is possible that all of them would be stolen and the worker would become idle, thus requiring stealing work from other workers. Thus, the caching

[1]http://www.openstream.info.

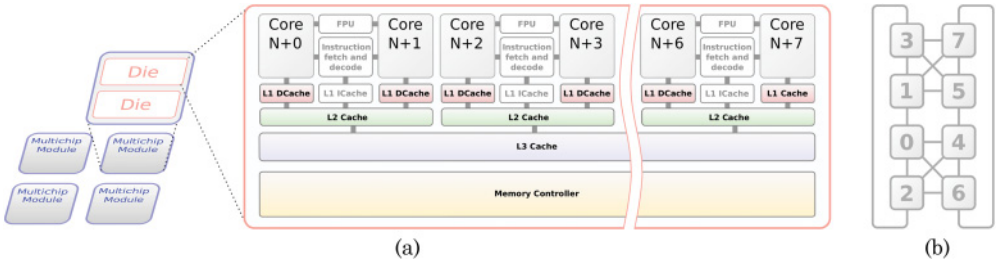

Fig. 9. Organization of the test system. (a) Shared and private caches. (b) Distance between NUMA nodes as reported by the NUMACTL tool.

technique allows reduction in the number of overall steals and reduction in the cost of the associated atomic operations.

The integration of dependence-aware allocation did not require any changes to the worker-specific data structures. Instead, the intertask dataflow dependences are derived from accesses to streams and captured in the existing data structures associated to each task. Then, these dependences are combined with dynamic information about the effective location of task buffers. Buffers of tasks that have terminated are returned to a slab allocator. Instead of freeing the memory immediately, the allocator adds it to a list of free buffers to be reused on future allocations. This common optimization greatly reduces the number of library calls related to memory management as well as the number of memory-related system calls. The implementation used for experimental evaluation uses one slab allocator per NUMA node, each of which manages buffers located within the associated memory region. The effective location of a buffer is determined at termination of the first task using it and cached in the buffer's metadata section for later reuse. This allows the runtime to efficiently determine to which slab allocator a buffer should be returned on termination of the associated task.

6. EXPERIMENTAL METHODOLOGY

To quantify the improvements of our optimizations over random work stealing, we have implemented the three strategies proposed in this work in the runtime of the OpenStream project. Using a set of different general-purpose, scientific applications, we have measured the impact of our optimizations on data locality, execution time, and speedup on a many-core NUMA system with 64 CPUs.

We first give an overview on the hardware and software environment used for experimentation followed by a presentation of the benchmarks used for evaluation. A detailed analysis of the results is given in the next section.

6.1. Experimental Setup

The following experiments were conducted on a quad-socket AMD Opteron 6282 SE running at a clock frequency of 2.6GHz. Figure 9(a) shows a hierarchical view of its basic components. At the coarsest level, the machine is composed of four physical packages called *multichip modules*. Each module contains two dies, each of which finally contains eight cores organized as pairs of cores sharing some resources.

At the core pair level, the floating point unit, the instruction fetcher and decoder, the first-level instruction cache, and the 2MB of second-level cache are shared. The third-level cache of 6MB and the memory controller are shared by all of the cores located on the same die. Among the private, per-core resources are the integer unit and the 16kB first-level data cache.

As implied by the sharing of memory controllers, main memory is divided into eight equally sized NUMA domains of 8GB so that the total amount of main memory available is 64GB. Their distances as reported by the NUMACTL tool [Kleen 2005] is visualized in Figure 9(b). For each domain, there are four neighbors at a distance of one hop and three neighbors at a distance of two hops.

According to the description scheme of the memory hierarchy of Section 2.3, there are six different levels for the testing machine and the definition of L is thus $L = \langle L1, L2, L3, \text{1hop}, \text{2hops}, \text{machine}\rangle$. The first three levels refer to the three levels of the cache hierarchy. All of the cores that share the third-level cache also share a memory controller; therefore, no additional level for cores at the same NUMA domain is modeled. Levels 1hop and 2hops represent cores at a NUMA distance of one (e.g., nodes 3 and 1 in Figure 9(b)) and two, respectively (e.g., nodes 3 and 0). The last level is used for complete random work stealing and contains all of the system's CPUs.

The machine was running Scientific Linux 6.2 with kernel 3.10.1. Micro-architectural events, such as cache misses and requests to main memory, were measured by sampling hardware performance counters using PAPI [Terpstra et al. 2010].

6.2. Benchmarks

We evaluate the impact of our technique on six applications. Each application is available in an optimized sequential implementation as well as tuned parallel implementations using OpenStream. The experimental evaluation studies the impact of our optimizations on locality and performance, compared to a parallel OpenStream baseline:

—*Seidel* simulates heat transfer using the Gauss-Seidel method, which iterates a 5-point stencil over a two-dimensional array. We used a resolution of $2^{14} \times 2^{14}$ points, divided in blocks of size $S_B \times S_B = 2^8 \times 2^8$, performing 60 iterations.
—*Bitonic* implements a bitonic sorting network [Batcher 1968], sorting a sequence of arbitrary 64-bit integer values. The input sequence is partitioned into blocks for parallel processing. Experiments were conducted using an array of 2^{28} 64-bit keys divided into blocks of $S_B = 2^{16}$ elements.
—*Kmeans* is a data-mining benchmark that partitions a set of n multidimensional points into k clusters using the K-means clustering algorithm. For evaluation, 40,960,000 points with 10 floating-point dimensions, generated from random walks around 11 centers have been clustered. For parallel processing, points are divided into blocks of $S_B = 10{,}000$ elements.
—*Sparse-LU* calculates the LU decomposition of a block-sparse matrix. It has been derived from an earlier version written in StarSs [Planas et al. 2009]. In our experiments, the matrix size is $2^{13} \times 2^{13}$, with blocks of size $2^7 \times 2^7$. Empty blocks—that is, blocks that only contain zero values—are not allocated. The number of empty blocks varies during execution, which leads to irregular parallelism in this benchmark.
—*Cholesky* calculates the lower triangular matrix L of a symmetric, positive definite matrix A, such that $A = L \cdot L^T$. We used dense matrices of size $2^{14} \times 2^{14}$ elements with blocks of $2^8 \times 2^8$ elements. In the parallel implementation, operations on individual blocks are carried out by highly tuned functions from LAPACK [Anderson et al. 1999] and BLAS [Blackford et al. 2001]. The sequential version uses the LAPACK implementation for Cholesky factorization.
—*FMradio* is a benchmark derived from the GNUradio[2] project and performs frequency demodulation on ADC samples. The input data stream used for our experiment consists of 46MB, which we process in batches of $g = 1{,}024$ complex samples.

[2]http://gnuradio.org/.

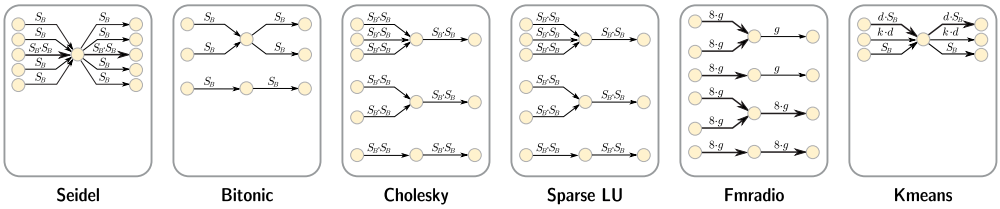

Fig. 10. Relevant types of input and output data dependences of the benchmarks.

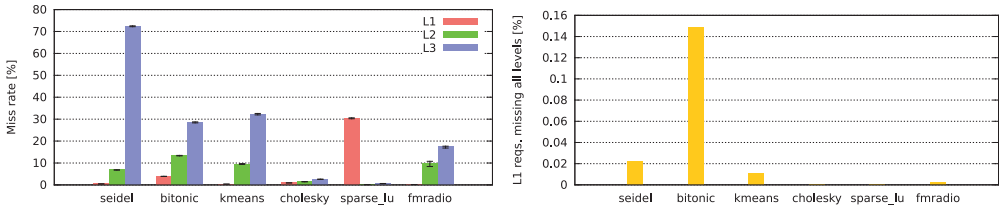

Fig. 11. (a) First-, second-, and third-level data cache miss rates of the parallel baseline (error bars indicate minimum and maximum values for 50 runs). (b) Estimation of L1 requests missing all cache levels.

6.2.1. Data Dependence Patterns. Figure 10 shows the relevant producer–consumer patterns for the dependent tasks of the benchmarks. These can be divided into two groups with different implications for work pushing, topology-aware work stealing, and topology-aware and dependence-aware allocation as seen in Sections 2, 3, and 4: asymmetric dependences (cf. *Seidel* or *Kmeans*) and symmetric dependences (cf. *Bitonic*, *Cholesky*, *Sparse-LU*, or *FMradio*). The behavior of our scheduling and allocation heuristics on these patterns are referenced in the experimental evaluation.

6.2.2. Characterization of Memory Accesses. Tile and block size parameters of the benchmarks were chosen carefully to take advantage of caches. However, efficiency of cache memory also depends on the pattern, the frequency, and the timing of memory accesses during execution of a benchmark, leading to more or fewer cache misses for a given block size. Based on the cache miss rates, benchmarks can be categorized as *compute-bound* applications or *memory-bound* applications.

Compute-bound benchmarks have a low cache miss rate, such that only few memory accesses result in accesses to main memory. Hence, their performance primarily depends on the performance of cache memory. In contrast to this, memory-bound applications frequently miss the data cache, resulting in frequent accesses to main memory. Although the latency of main memory accesses has only little influence on compute-bound benchmarks, it is crucial for the performance of memory-bound benchmarks. Thus, memory-bound benchmarks are far more sensitive to NUMA-related optimizations than compute-bound benchmarks. *Seidel* and *Bitonic* are memory bound, whereas *Kmeans*, *Cholesky*, *Sparse-LU*, and *FMradio* are compute bound.

Figure 11(a) shows the cache miss rates at the different cache levels for parallel execution of the baseline with neither optimizations for scheduling nor optimized memory allocation. *Sparse-LU* has the highest L1 miss rate, with about 30%. But unlike *Seidel* and *Bitonic*, *Sparse-LU* only generates very few misses on the other levels of the cache hierarchy. *Cholesky* and *FMradio* have high L2 and L3 miss rates but have a very high L1 hit rate, and *Cholesky* generates only few misses at all levels. Figure 11(b) summarizes this information, showing the product of the miss rates as an estimation

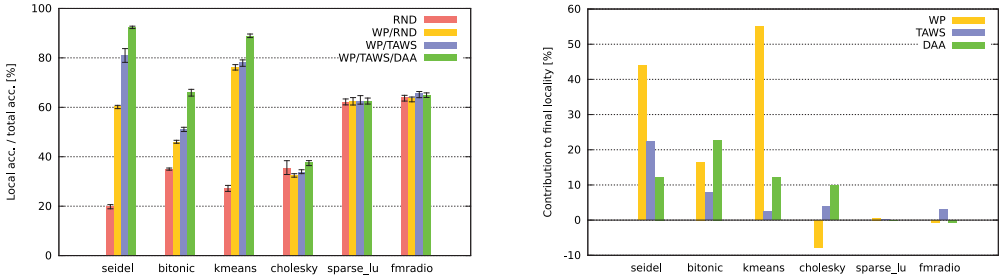

Fig. 12. (a) Ratio of requests to memory of local nodes to the total number of requests to main memory (error bars indicate minimum and maximum values for 50 runs). (b) Contribution of each optimization to the final locality.

of the fraction of L1 requests that miss all levels of the cache hierarchy and thus result in accesses to main memory.

7. EXPERIMENTAL RESULTS

In the following comparison of the optimization techniques, we focus on the influence on the locality of data accesses before studying the impact on execution time. The following acronyms are used to refer to different configurations: *RND* (random work stealing), *WP* (work pushing), *TAWS* (topology-aware work stealing), and *DAA* (modified dependence-aware allocation).

7.1. Data Locality

The block sizes for all benchmarks were chosen carefully to achieve high data reuse during execution of a single task. For the actual values, this implies that a task's working data occupies a large portion of the second- and third-level cache. Furthermore, as there are more tasks than cores, several other tasks might be executed between a producing task and its consumer. It is thus highly unlikely that a huge fraction of a task's input data is still present in a cache at the beginning of its execution. Hence, improvement on data locality with respect to caches is expected to be less important than improvement of data locality concerning main memory.

The following analysis is therefore divided into two parts. The first part studies the locality of accesses to main memory—that is, accesses to memory controllers located on the different NUMA nodes. The second part focuses on accesses to caches.

7.1.1. Locality of Accesses to Main Memory. Figure 12(a) presents the ratio of requests to local memory controllers to the total number of requests to main memory for the different benchmarks. Each bar shows the result for particular combination of optimizations: random work stealing as the baseline, work pushing in conjunction with random work stealing, work pushing and topology-aware work stealing, and all three optimizations proposed in this article applied at once. To highlight the influence of each optimization, Figure 12(b) shows their respective contribution to the final locality—that is, the increase in locality achieved by adding the optimization divided by the final rate of local requests. The following conclusions can be drawn from the results.

Seidel. Activation of work pushing causes a huge increase of local accesses (from 20% to 60%). As intended, work pushing greatly improves locality of read accesses. But as a side effect, an important fraction of write accesses *also* results in local accesses. The principles are shown in Figure 13. Figure 13(a) shows the initial situation in which the first task and its consumer are both created by the root task. By default,

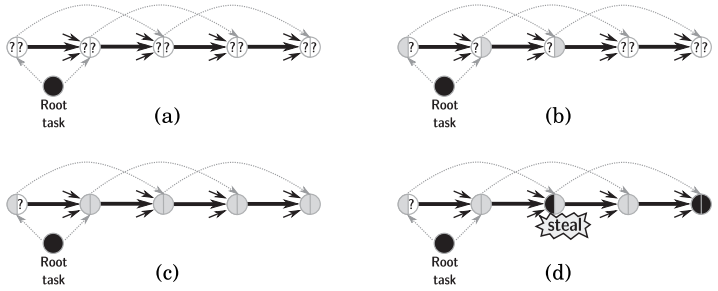

Fig. 13. Side effects of work pushing due to delayed physical page allocation in *Seidel*. (a) Logical allocation by the root task. (b) Physical allocation on write access to the task buffer. (c) Resulting pattern. (d) Alternating pattern resulting from work stealing.

these allocations happen in the local memory of the worker executing the root task. However, when a task buffer is allocated for the first time—that is, using a system call and without reusing a buffer from the runtime's slab allocator described earlier—its pages are not yet allocated *physically*. Physical allocation is triggered by a copy-on-write mechanism on the first write access to a page. At that moment, the operating system decides on which node the page is to be allocated. Hence, *initial* allocation of task buffers does not result in physical allocation of the buffers' pages. As input data is generated by the initial task with a high probability to execute on a different node than the root task due to work stealing, physical allocation of the consumer's buffer is done on a different node than logical allocation (cf. Figure 13(b)). The initial task allocates the task buffer of the consumer's successor, hence establishing a stable chain of tasks with strong dependences with buffers allocated on the same node (cf. Figure 13(c)). However, work stealing easily disturbs this pattern and leads to alternating allocations (cf. Figure 13(d)). Topology-aware work stealing reduces the probability for such a situation as steals of tasks from the same node are favored, hence the additional increase of local accesses to 81%. But once an alternating pattern is established, it is unlikely to disappear. Dependence-aware allocation helps in breaking the alternation and further increases the ratio of requests to local memory to 92%.

Bitonic. The ratio of requests to local memory during execution using random work stealing is higher than for *Seidel*, at approximately 35%. The impact of work pushing (increase to 46%) and topology-aware work stealing (increase to 51%) is smaller due to the symmetric dependences of the benchmark, which disable the side effect for allocation as described for *Seidel*. Therefore, dependence-aware allocation is the key optimization for *Bitonic*. With all optimizations enabled, about 66% of the requests to main memory are satisfied by a local controller.

Kmeans. This application has very similar characteristics to *Seidel* with the same side effect for buffer allocation. The initial ratio of 27% increases to 76% with work pushing, to 78% with both work pushing and topology-aware work stealing and to 89% when all optimizations are enabled.

Cholesky. The optimizations fail to increase the ratio of local accesses due to imprecise information on the effective location of buffers. The problem is illustrated in Figure 14. The pages of task buffers are physically allocated on up to three different nodes (Figure 14(a) and 14(b)). However, for task buffer reuse based on a slab allocator per NUMA node, the runtime must associate the buffer to a *single* node. Subsequent allocations with a hit in the slab allocator will not return buffers allocated *entirely* on the requested node. As no precise information about page placement is modeled, the

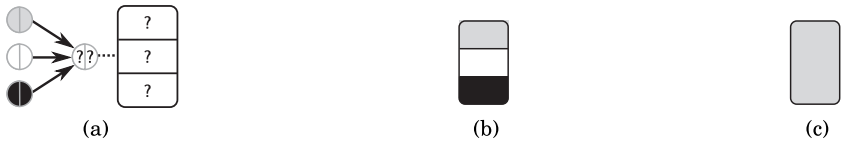

Fig. 14. Dependence patterns in *Cholesky* leading to imprecise information about page distribution. (a) Data dependences and initially unknown physical distribution. (b) Physical page distribution. (c) Runtime view on the task buffer.

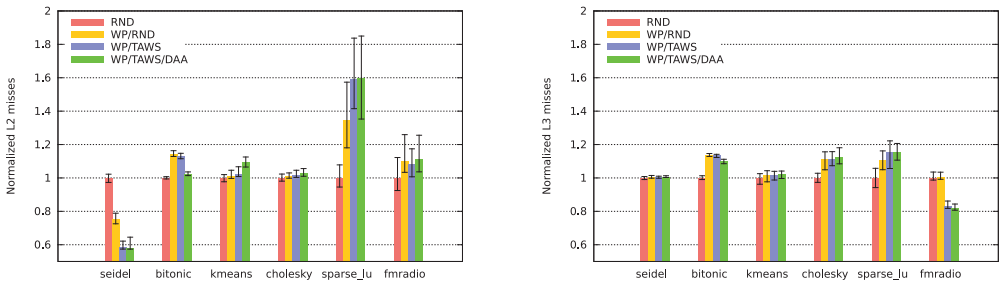

Fig. 15. Cache misses normalized to unoptimized parallel execution. Error bars indicate minimum and maximum values for 50 executions.

runtime assumes that the whole buffer resides on a single node (Figure 14(c)). Optimization decisions are hence based on imprecise information. Dynamic page migration on buffer reuse or explicit physical placement at logical allocation can solve the problem of imprecise information but requires elaborated load-balancing mechanisms for buffer placement of initial task. Evaluation of these mechanisms is outside the scope of this article and will be the subject of future work.

Sparse-LU and FMradio. All tasks in these benchmarks are created by the root task during a short period at the beginning of the execution. During this period, only a small fraction of tasks terminates, hence only few buffers can be reused on allocation. The physical location of most of the buffers is therefore unknown until physical allocation on write accesses of producers. These writes cause the pages to be allocated on the writing node and are thus local, resulting in a high initial ratio of local accesses.

Work pushing is not beneficial for small task buffers. The work-pushing mechanism therefore is invoked only for tasks whose buffers are greater than an experimentally determined threshold. The task buffers of *FMradio* do not exceed this threshold, such that work pushing is not triggered. As the other optimizations depend on work pushing, neither improve locality. However, the ratio of requests to local memory does not decrease, showing that there is no degradation even for this unfavorable case.

Sparse-LU cannot benefit from the optimizations due to imprecise information about the physical location of buffers as described for *Cholesky*, but as for *FMradio*, no degradation is observable.

7.1.2. Cache Misses. Figure 15 shows the number of cache misses for the last two levels of the cache hierarchy normalized to parallel execution without any optimization. For the L2 cache, the difference varies from a reduction of about 40% to an increase of 60%. Variation of L3 cache misses is below 20% in both directions. Given the low initial miss rates of compute bound benchmarks, the impact on performance of these variations is negligible. For memory-bound benchmarks, the impact of improved locality of

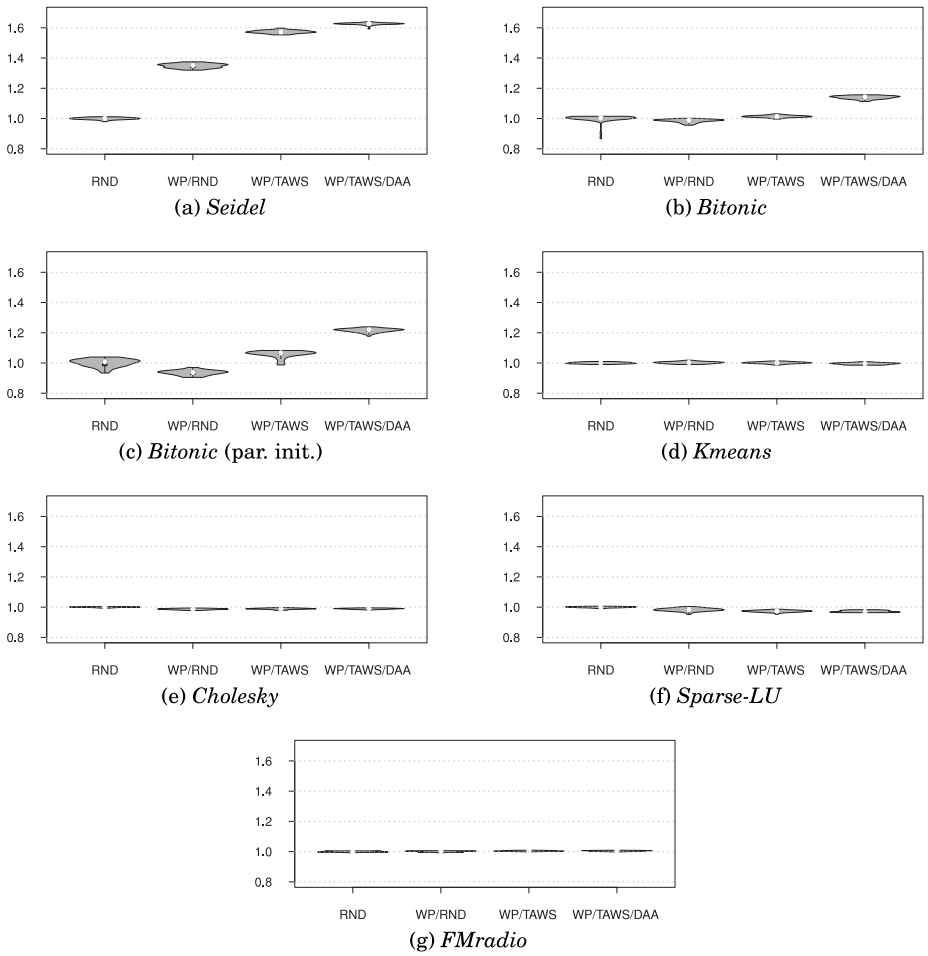

Fig. 16. Violin plots showing the speedup over random work stealing of 50 runs for each configuration.

accesses to main memory outweighs the variations of cache misses, leading to better performance. Both cases are shown in the next section.

7.2. Performance Impact

Figure 16 shows the speedup over random work stealing grouped by benchmark. The results are presented using the `R` *vioplot* library. These violin plots combine box plots and kernel density estimators. The width represents the density of data points for a given value, with a logarithmic bias. Solid black bars represent the 95% confidence intervals for the median, whereas the white dot within each violin represents the median of the distribution.

As noted in Section 6.2.2, only *Seidel* and *Bitonic* are memory bound. Hence, the increased ratio of requests to local memory for these benchmarks translates into a significant improvement over random work stealing, as shown in Figure 16(a) and (b). Initially executing about 13.5× faster than the sequential version, the speedup for *Seidel* can be increased to about 21.9×. The speedup for the bitonic sorting network slightly drops when using work pushing only due to poor load balancing and remote

Table I. Speedups over Sequential Execution and over Random Work Stealing

	Seidel	*Bitonic*	*Bitonic* (par. init)	*Kmeans*	*Cholesky*	*Sparse-LU*	*FMradio*	*Geometric Mean (a)*[1] */ (b)*[2]
RND	13.5	16.8	16.6	36.9	60.8	49.0	52.5	33.1/33.0
WP/RND	18.2	16.6	15.5	37.0	60.0	48.2	52.5	34.5/34.1
WP/TAWS	21.2	17.0	17.5	37.0	60.1	47.7	52.7	35.5/35.7
WP/TAWS/DAA	**21.9**	**19.2**	**20.2**	36.7	60.1	47.6	52.7	36.4/36.7
Speedup over RND	**1.63**	**1.14**	**1.22**	1.00	0.99	0.97	1.00	1.10/1.11

[1]Including *Bitonic*. [2]Including *Bitonic (par. init.)*.

steals. However, adding topology-aware work pushing already outperforms random work stealing, and all three optimizations raise the initial speedup of 16.8× to 19.2×.

During inspection of the execution traces of *Bitonic* using the *Aftermath* trace-based visualization and analysis tool [Drebes et al. 2014], we discovered that the creation of initial tasks by a single worker is not fast enough to provide work for all cores. This problem only occurs for optimized execution with dependence-aware allocation. To compensate for this lack of parallelism, we have derived another version of *Bitonic*, creating initial tasks in parallel. The increased parallelism during initialization can now be exploited as shown in Figure 16(c). The speedup for random work stealing is slightly lower compared to sequential creation of initial tasks. However, the final speedup for topology-aware work stealing, dependence-aware allocation, and work pushing is more than 20.2-fold instead of 19.2× for sequential task creation.

Performance of compute-bound benchmarks stays approximately constant with a variation of less than 3% on average, showing that there is no significant negative impact on benchmarks with few cache misses despite increased cache miss rates. Table I summarizes the average speedups over sequential execution and the speedup for all optimizations enabled over the parallel baseline.

7.3. Summary

The results show that work pushing is the key optimization for data locality (cf. Figure 12(b)) for benchmarks with asymmetric dependences, greatly eliminating the mismatch between task and data ownership. Topology-aware work stealing is complementary to work pushing and further increases locality, but its contribution is smaller than work pushing. Dependence-aware allocation is beneficial for benchmarks with symmetric dependences, such as *Bitonic*.

The results confirm that the optimizations techniques are complementary. Best results for the locality of data accesses are achieved with a combination of topology-aware work stealing, dependence-aware allocation, and work pushing.

Memory-bound applications benefit most from improved locality, resulting in increased performance. For compute-bound applications, neither data locality nor performance are degraded, even in unfavorable cases with imprecise information about buffer placement or sequential task creation during initialization.

8. RELATED WORK

Lightweight task parallelism is a cornerstone of many parallel programming environments. Optimizations for scheduling such tasks were developed in different areas, from event-driven systems to Cilk runtime implementations, and OpenMP. To our knowledge, there is no prior topology-aware and dependence-aware approach combining both scheduling of lightweight tasks and memory allocation. In the following, we discuss the most closely related work on locality-aware scheduling.

Approaches for Cilk-based implementations. A multitude of dynamic, load-balancing schedulers have been evaluated, from load-sharing [Rudolph et al. 1991] to work stealing [Blumofe and Leiserson 1999] and work dealing [Hendler and Shavit 2002]. Although locality and task/data affinity may be a concern of these techniques, none takes advantage of the memory access pattern of the application.

On the contrary, Chen et al. [2012] proposed CATS, a profile-guided two-level hierarchical cache-aware work-stealing algorithm designed for multisocket multicore architectures. Based on an estimation of the working set size of tasks during the first iteration, task graphs of subsequent iterations are partitioned into inter- and intrasocket tasks, which allows grouping neighboring tasks on the same socket without exceeding its shared cache capacity. Data accesses are assumed to happen only in leaf tasks of iterations. In contrast to this approach, we also support noniterative applications, and data accesses can happen in any task. Furthermore, we do not rely on profiling, our topology-aware work-stealing algorithm supports more than two levels for hierarchical scheduling, and we also address topology-aware and dependence-aware memory allocation.

Custom frameworks. Chandra et al. [1993] propose a locality-aware scheduler for concurrent object-oriented language (COOL) based on affinity hints provided by the programmer. The memory hierarchy is assumed to have three levels—private caches, local DRAM, and remote DRAM—but is not modeled explicitly. So-called task and object affinities can be specified for objects, where task affinities form task sets identified by the referenced object and object affinities relate to the memory location referenced by a task. The scheduler executes tasks with the same object affinity on the core owning the object and groups tasks of the same set to execute one after the other. The former technique reduces the number of remote memory accesses, whereas the latter provides better reuse of cached data. However, the approach heavily relies on annotations and therefore requires expert knowledge on the application.

A locality-guided work-stealing approach for applications in which locality of tasks in the computation graph does not reflect locality of data accesses is presented in Acar et al. [2000]. Based on affinities of tasks for cores, every newly created task is put both into the queue of the creating core and into a FIFO queue called *mailbox* of the core defined by the affinity. When a worker runs out of tasks, it first considers its mailbox before attempting to steal from someone else. We use a similar concept for task transfers, but instead of relying on explicit affinities, we determine locality fully automatically from dynamic producer–consumer relationships of tasks.

StarPU is primarily designed for heterogeneous platforms with discrete accelerators [Augonnet et al. 2011], but its generic support for dynamic, dependent tasks and its modular scheduler design are also unique among schedulers for shared memory architectures. In particular, a variety of scheduling policies can be and have been implemented in StarPU. Many of these integrate locality and work/communication metrics. But although the toolkit itself is not restricted, the implemented strategies have mostly been static. They cover a variety of performance models, as well as application-specific priorities for classical LAPACK kernels.

SLAW [Guo et al. 2010] is a scheduler with support for the Habanero-Java language and uses *places* as the central abstraction for locality-aware scheduling. Associating a task to a place limits its execution to a specific subset of workers threads. Steals only occur between workers of the same place. To allow task creation on remote places, each place is provided with a mailbox dedicated to task reception. The mapping of tasks to places, however, fully relies on hints provided by the programmer.

HotSLAW [Min et al. 2011] is a library for UPC [UPC Consortium 2005], providing support for locality-aware schedulers. As a prototype, the authors propose a

hierarchical work-stealing mechanism with a configurable number of steal attempts at each level, similar to our topology-aware work-stealing approach. Memory accesses during execution of tasks are assumed to target local memory of the victim from which a task is stolen. Data exchanges between tasks are therefore not modeled explicitly.

The optimizations proposed in Yoo et al. [2013] address unstructured parallelism—that is, parallel sections with independent tasks that can be scheduled in any order. Data sharing is determined through profiling of the application and is captured in a task sharing graph. Analysis of this graph allows the formation of task groups that are then scheduled over a hierarchy of work queues organized according to the topology of the machine. The topology-aware work-stealing mechanism uses an incrementally widening neighborhood for steal attempts similar to our approach but also deals with task group decomposition as queues contain task groups rather than individual tasks.

Event-driven systems. The MELY runtime [Gaud et al. 2010] for event-driven systems modifies the central function that generates the list of cores to steal from of the work-stealing algorithm of LIBASYNC-SMP [Zeldovich et al. 2003] in a way that steals from cores that are close in the memory hierarchy are favored. Locality-aware work stealing of MELY focuses on reducing the cost of queue accesses during a steal rather than the cost of memory accesses during execution of an event handler. Another strategy presented in this article takes into account the size of an event's dataset and avoids migration of tasks with large datasets. However, improvement of data locality fully relies on passive work stealing without proactive remote transfers.

OpenMP runtimes. The FORRESTGOMP runtime [Broquedis et al. 2010] for OpenMP implements a hierarchical, resource-aware scheduler using the BUBBLESCHED framework [Thibault et al. 2007]. A NUMA-aware memory allocator associates the amount and the location of allocated memory to the allocating thread. Threads accessing nearby data are grouped in bubbles that also form the basic scheduling entities. By scheduling these bubbles over an explicit hierarchical representation of the machine's resources, the scheduler keeps threads accessing the same data region closely together with respect to the memory hierarchy. Data placement is addressed by dynamic migrations at runtime using a next-touch policy and initially by programmer hints on data distribution on allocation. The approach works particularly well for applications that partition the data among the descendants of threads by using nested parallelism.

LIBKOMP [Broquedis et al. 2012] uses the concept of memory regions accessed by a task modeled as partitionings of multidimensional arrays. The programmer specifies how tasks access these regions, allowing the runtime to calculate dataflow dependences. These dependences are only used to express task dependences and thus to evaluate if a task is ready for execution. The scheduler does not use this information to group tasks according to their data accesses.

OMPSS [Duran et al. 2008] is the OpenMP incarnation of the StarSs [Planas et al. 2009] parallel programming environment and relies on the Nanos runtime library [Teruel et al. 2007]. Similarly to LIBKOMP, it allows expressing task dependences through annotations on memory regions accessed by tasks. The Nanos scheduler offers two flavors, *breadth-first* and *work-first*, which do not factor in NUMA characteristics. The work-first scheduler is similar to Cilk's and has the same advantages and drawbacks: it generally provides good cache locality on divide-and-conquer parallelization patterns, but it does not take into account application runtime behavior.

To wrap up, let us recall that our work differs from most prior art in that it exploits precise information about data transfers among dependent tasks, and the information is derived automatically from a dynamic dataflow language.

9. CONCLUSION

In this article, we investigated the performance anomalies of random work stealing and topology-unaware memory allocation on the data locality of task-parallel applications. We presented three complementary locality optimizations to alleviate these weaknesses. *Work pushing* transfers tasks to workers executing on NUMA nodes that contain the tasks' input data, hence optimizing for read accesses. *Topology-aware work stealing* uses a lightweight, static description of the memory hierarchy to improve the process of work stealing by favoring steals in an incrementally widening neighborhood of the stealing core. *Dependence-aware memory allocation* optimizes the locality of write accesses by allocating buffers of dependent tasks on the same node. The three optimizations rely only on explicit point-to-point task dependence information and the dynamic monitoring of task and data placement. We showed that they can be efficiently integrated in the runtime system of a modern task-parallel language—OpenStream.

Our experimental results confirm that for a set of general-purpose and scientific applications, the locality of accesses to main memory can be increased significantly. This effect naturally yields performance improvements on memory-intensive benchmarks.

The results also show that our approach does not impact the performance of compute-bound applications. However, as the optimizations do not bring any benefit in these cases, we plan to include online detection of compute-bound algorithms in future work to disable them when appropriate.

Further investigation is needed on how the optimizations can be enabled for (sequential) task creation and imprecise information about initial task buffer placement, which is present in some of the compute-bound OpenStream applications.

In future work, we also plan to extend the applicability of this work to other state-of-the-art task-parallel languages such as X10 [Charles et al. 2005], Habanero Java and C [Cavé et al. 2011], CnC [Budimlicć et al. 2010], Chapel [Chamberlain et al. 2007], or StarSs [Planas et al. 2009]. In these languages, the main difficulty to overcome will be recovering missing runtime information on precise dependences or on communication intensity between tasks.

REFERENCES

Umut A. Acar, Guy E. Blelloch, and Robert D. Blumofe. 2000. The data locality of work stealing. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*. ACM, New York, NY, 1–12. DOI:http://dx.doi.org/10.1145/341800.341801

Edward Anderson, Zhaojun Bai, Christian Bischof, Laura Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, A. McKenney, and Danny Sorensen. 1999. *LAPACK Users' Guide* (3rd ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA.

Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience* 23, 2, 187–198. DOI:http://dx.doi.org/10.1002/cpe.1631

Kenneth E. Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference (AFIPS'68)*. ACM, New York, NY, 307–314. DOI:http://dx.doi.org/10.1145/1468075.1468121

Micah Best, Shane Mottishaw, Craig Mustard, Mark Roth, Parsiad Azimzadeh, Alexandra Fedorova, and Andrew Brownsword. 2011. Schedule data, not code. In *Proceedings of the 3rd USENIX Workshop on Hot Topics on Parallelism (HotPar'11)*.

L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roldan Pozo, Karin Remington, and R. Clint Whaley. 2001. An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software* 28, 2, 135–151.

Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. 2010. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems* 28, 4, Article No. 8. DOI:http://dx.doi.org/10.1145/1880018.1880019

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*. ACM, New York, NY, 207–216. DOI:http://dx.doi.org/10.1145/209936.209958

Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM* 46, 5, 720–748. DOI:http://dx.doi.org/10.1145/324133.324234

François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. 2010. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming* 38, 5, 418–439. DOI:http://dx.doi.org/10.1007/s10766-010-0136-3

François Broquedis, Thierry Gautier, and Vincent Danjean. 2012. LIBKOMP, an efficient OpenMP runtime system for both fork-join and data flow paradigms. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World (IWOMP'12)*. Springer-Verlag, Berlin, Heidelberg, 102–115. DOI:http://dx.doi.org/10.1007/978-3-642-30961-8_8

Zoran Budimlìc, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Taşirlar. 2010. Concurrent collections. *Scientific Programming* 18, 3–4, 203–217. http://portal.acm.org/citation.cfm?id=1938482.1938486

Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ'11)*. ACM, New York, NY, 51–61. DOI:http://dx.doi.org/10.1145/2093157.2093165

Bradford L. Chamberlain, David Callahan, and Hans P. Zima. 2007. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications* 21, 3, 291–312. DOI:http://dx.doi.org/10.1177/1094342007078442

Rohit Chandra, Anoop Gupta, and John L. Hennessy. 1993. Data locality and load balancing in COOL. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'93)*. ACM, New York, NY, 249–259. DOI:http://dx.doi.org/10.1145/155332.155358

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*. ACM, New York, NY, 519–538. DOI:http://dx.doi.org/10.1145/1094811.1094852

David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'05)*. ACM, New York, NY, 21–28. DOI:http://dx.doi.org/10.1145/1073970.1073974

Quan Chen, Minyi Guo, and Zhiyi Huang. 2012. CATS: Cache aware task-stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*. ACM, New York, NY, 163–172. DOI:http://dx.doi.org/10.1145/2304576.2304599

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, New York, NY, 381–394. DOI:http://dx.doi.org/10.1145/2451116.2451157

Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-Temam. 2014. Aftermath: A graphical tool for performance analysis and debugging of fine-grained task-parallel programs and run-time systems. In *Proceedings of the 7th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG'14)*.

Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. Evaluation of OpenMP task scheduling strategies. In *OpenMP in a New Era of Parallelism*, Rudolf Eigenmann and Bronis R. Supinski (Eds.). Lecture Notes in Computer Science, Vol. 5004. Springer-Verlag, Berlin, Heidelberg, 100–110. DOI:http://dx.doi.org/10.1007/978-3-540-79561-2_9

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI'98)*. ACM, New York, NY, 212–223. DOI:http://dx.doi.org/10.1145/277650.277725

Fabien Gaud, Sylvain Genevès, Renaud Lachaize, Baptiste Lepers, Fabien Mottet, Gilles Muller, and Vivien Quéma. 2010. Efficient workstealing for multicore event-driven systems. In *Proceedings of the 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS'10)*. IEEE, Los Alamitos, CA, 516–525. DOI:http://dx.doi.org/10.1109/ICDCS.2010.55

Thierry Gautier, Xavier Besseron, and Laurent Pigeon. 2007. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multi-processors. In *Proceedings of the 2007 In-*

ternational Workshop on Parallel Symbolic Computation (PASCO'07). ACM, New York, NY, 15–23. DOI:http://dx.doi.org/10.1145/1278177.1278182

Yi Guo, Jisheng Zhao, Vincent Cave, and Vivek Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler for multi-core systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*. ACM, New York, NY, 341–342. DOI:http://dx.doi.org/10.1145/1693453.1693504

Danny Hendler and Nir Shavit. 2002. Work dealing. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'02)*. ACM, New York, NY, 164–172. DOI:http://dx.doi.org/10.1145/564870.564900

Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. 2008. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM, New York, NY, 220–229. DOI:http://dx.doi.org/10.1145/1454115.1454146

Andreas Kleen. 2005. A NUMA API for Linux. Retrieved July 25, 2014, from http://halobates.de/numaapi3.pdf.

Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'13)*. ACM, New York, NY.

Seung-Jai Min, Costin Iancu, and Katherine Yelick. 2011. Hierarchical work stealing on manycore clusters. In *Proceedings of the 5th Conference on Partitioned Global Address Space Programming Models (PGAS'11)*.

Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. 2009. Hierarchical task-based programming with StarSs. *International Journal on High Performance Computing Architecture* 23, 3, 284–299.

Antoniu Pop and Albert Cohen. 2013. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article No. 53. DOI:http://dx.doi.org/10.1145/2400682.2400712

Larry Rudolph, Miriam Slivkin-Allalouf, and Eli Upfal. 1991. A simple load balancing scheme for task allocation in parallel machines. In *Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'91)*. ACM, New York, NY, 237–245. DOI:http://dx.doi.org/10.1145/113379.113401

Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. 2010. Collecting performance data with PAPI-C. In *Tools for High Performance Computing 2009*, Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel (Eds.). Springer-Verlag, Berlin, Heidelberg, 157–173.

Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. 2007. Support for OpenMP tasks in Nanos v4. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research (CASCON'07)*. IBM Corp., Riverton, NJ, 256–259. DOI:http://dx.doi.org/10.1145/1321211.1321241

Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2007. Building portable thread schedulers for hierarchical multiprocessors: The bubblesched framework. In *Euro-Par 2007 Parallel Processing*, Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol (Eds.). Lecture Notes in Computer Science, Vol. 4641. Springer-Verlag, Berlin, Heidelberg, 42–51. DOI:http://dx.doi.org/10.1007/978-3-540-74466-5_6

UPC Consortium. 2005. *UPC Language Specifications, v1.2*. Technical Report LBNL-59208. Lawrence Berkeley National Lab. Available at http://www.gwu.edu/~upc/publications/LBNL-59208.pdf.

Richard M. Yoo, Christopher J. Hughes, Changkyu Kim, Yen-Kuang Chen, and Christos Kozyrakis. 2013. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the 25th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'13)*. ACM, New York, NY, 315–325. DOI:http://dx.doi.org/10.1145/2486159.2486175

Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and Frans Kaashoek. 2003. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX'03)*.

Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. 2012. Survey of scheduling techniques for addressing shared resources in multicore processors. *Computing Surveys* 45, 1, Article No. 4. DOI:http://dx.doi.org/10.1145/2379776.2379780

Received June 2013; revised April 2014; accepted June 2014