Three hours

Two academic papers are provided for use with the examination.

UNIVERSITY OF MANCHESTER
SCHOOL OF COMPUTER SCIENCE

Designing for Parallelism and Future Multi-core Computing

Date: Monday 16th January 2017
Time: 09:45 - 12:45

Please answer the single Question provided

Two academic papers are attached for use with the examination.
Otherwise this is a CLOSED book examination.
The use of electronic calculators is NOT permitted.
1. Provide an analysis of one of the following two papers:

a) Scalable Architecture for Ordered Parallelism. MICRO 2015

or

b) Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. PACT 2016

by answering the following questions:

a) What is the problem being addressed? (10 marks)
b) What is the proposed solution? (12 marks)
c) What are the assumptions? (6 marks)
d) How is it evaluated? (12 marks)
e) What are the limitations? (6 marks)
f) Overall assessment of paper and possible improvements? (4 marks)

(Total 50)

END OF EXAMINATION
A Scalable Architecture for Ordered Parallelism

Mark C. Jeffrey† Suvinay Subramanian† Cong Yan† Joel Emer∗ Daniel Sanchez†
†MIT CSAIL ∗NVIDIA / MIT CSAIL
{mcj, suvinay, congy, emer, sanchez}@csail.mit.edu

ABSTRACT
We present Swarm, a novel architecture that exploits ordered irregular parallelism, which is abundant but hard to mine with current software and hardware techniques. In this architecture, programs consist of short tasks with programmer-specified timestamps. Swarm executes tasks speculatively and out of order, and efficiently speculates thousands of tasks ahead of the earliest active task to uncover ordered parallelism. Swarm builds on prior TLS and HTM schemes, and contributes several new techniques that allow it to scale to large core counts and speculation windows, including a new execution model, speculation-aware hardware task management, selective aborts, and scalable ordered commits.

We evaluate Swarm on graph analytics, simulation, and database benchmarks. At 64 cores, Swarm achieves 51–122× speedups over a single-core system, and outperforms software-only parallel algorithms by 3–18×.

Categories and Subject Descriptors
C.1.4 [Processor architectures]: Parallel architectures

Keywords
Multicore, ordered parallelism, irregular parallelism, fine-grain parallelism, synchronization, speculative execution

1. INTRODUCTION
Parallel architectures are now pervasive, but thread-level parallelism in applications is often scarce [19, 35]. Thus, it is crucial that we explore new architectural mechanisms to efficiently exploit as many types of parallelism as possible. Doing so makes parallel systems more versatile, easier to program, and, for many applications, it is the only way to improve performance.

We focus on ordered irregular parallelism [55], which is often abundant but hard to exploit. Programs with ordered irregular parallelism have three key features:

First, they consist of tasks that must follow a total or partial order. Second, tasks may have data dependences that are not known a priori. Third, tasks are not known in advance. Instead, tasks dynamically create children tasks and schedule them to run at a future time.

Ordered irregular parallelism is abundant in many domains, such as simulation, graph analytics, and databases. For example, consider a timing simulator for a parallel computer. Each task is an event (e.g., executing an instruction in a simulated core). Each task must run at a specific simulated time (introducing order constraints among tasks), and modifies a specific component (possibly introducing data dependences among tasks). Tasks dynamically create other tasks (e.g., a simulated memory access), possibly for other components (e.g., a simulated cache), and schedule them for a future simulated time.

Prior work has tried to exploit ordered parallelism in software [33, 34], but has found that, in current multicores, runtime overheads negate the benefits of parallelism. This motivates the need for architectural support.

To guide our design, we first characterize several applications with ordered irregular parallelism (Sec. 2). We find that tasks in these applications are as small as a few tens of instructions. Moreover, many of these algorithms rarely have true data dependences among tasks, and their maximum achievable parallelism exceeds 100×. It may seem that thread-level speculation (TLS) [28, 60, 66, 68], which speculatively parallelizes sequential programs, could exploit this parallelism. However, this is not the case due to two reasons (Sec. 3):

• Ordered irregular algorithms have little parallelism when written as sequential programs. To enforce order constraints, sequential implementations introduce false data dependences among otherwise independent tasks. For example, sequential implementations of timing simulators use a priority queue to store future tasks. Priority queue accesses introduce false data dependences that limit the effectiveness of TLS.

• To scale, ordered irregular algorithms need very large speculation windows, of thousands of tasks (hundreds of thousands of instructions). Prior TLS schemes use techniques that scale poorly beyond few cores and cannot support large speculation windows.

We present Swarm, an architecture that tackles these challenges. Swarm consists of (i) a task-based execution model where order constraints do not introduce false data dependences, and (ii) a microarchitecture that
leverages this execution model to scale efficiently (Sec. 4). Swarm is a tiled multicore with distributed task queues, speculative out-of-order task execution, and ordered task commits. Swarm adapts prior eager version management and conflict detection schemes [48, 79], and features several new techniques that allow it to scale. Specifically, we make the following novel contributions:

- An execution model based on tasks with programmer-specified timestamps that conveys order constraints to hardware without undue false data dependencies.
- A hardware task management scheme that features speculative task creation and dispatch, drastically reducing task management overheads, and implements a very large speculation window.
- A scalable conflict detection scheme that leverages eager versioning to, upon mispeculation, selectively abort the mispeculated task and its dependents (unlike prior TLS schemes that forward speculative data, which abort all later tasks).
- A distributed commit protocol that allows ordered commits without serialization, supporting multiple commits per cycle with modest communication (unlike prior schemes that rely on successor lists, token-passing, and serialized commits).

We evaluate Swarm in simulation (Sec. 5 and Sec. 6) using six challenging workloads: four graph analytics algorithms, a discrete-event simulator, and an in-memory database. At 64 cores, Swarm achieves speedups of 51–122× over a single-core Swarm system, and outperforms state-of-the-art parallel implementations of these algorithms by 2.7–18.2×. In summary, by making ordered execution scalable, Swarm speeds up challenging algorithms that are currently limited by stagnant single-core performance. Moreover, Swarm simplifies parallel programming, as it frees developers from using error-prone explicit synchronization.

2. MOTIVATION

2.1 Ordered Irregular Parallelism

Applications with ordered irregular parallelism have three main characteristics [33, 55]. First, they consist of tasks that must follow a total or partial order. Second, tasks are not known in advance. Instead, tasks dynamically create children tasks, and schedule them to run at a future time. Task execution order is different from task creation order. Third, tasks may have data dependencies that are not known a priori.

Ordered irregular algorithms are common in many domains. First, they are common in graph analytics, especially in search problems [33, 55]. Second, they are important in simulating systems whose state evolves over time, such as circuits [47], computers [12, 59], networks [37, 72], healthcare systems [39], and systems of partial differential equations [32, 44]. Third, they are needed in systems that must maintain externally-imposed order constraints, such as geo-replicated databases where transactions must appear to execute in timestamp order [14], or deterministic architectures [17, 45] and record-and-replay systems [36, 77] that constrain the schedule of parallel programs to ensure deterministic execution.

To illustrate the challenges in parallelizing these applications, consider Dijkstra’s single-source shortest paths (sssp) algorithm [15, 22]. sssp finds the shortest distance between some source node and all other nodes in a graph with weighted edges. Fig. 1(a) shows the sequential code for sssp, which uses a priority queue to store tasks. Each task operates on a single node, and is ordered by its tentative distance to the source node. sssp relies on task order to guarantee that the first task to visit each node comes from a shortest path. This task sets the node’s distance and enqueues all its children. Fig. 1(b) shows an example graph, and Fig. 1(c) shows the tasks that sssp executes to process this graph. Fig. 1(c) shows the order of each task (its distance to the source node) in the x-axis, and outlines both parent-child relationships and data dependences. For example, task A at distance 0, denoted (A, 0), creates children tasks (C, 2) and (B, 3); and tasks (B, 3) and (B, 4) both access node B, so they have a data dependence.

A distinctive feature of irregular parallel programs is that task creation and execution order are different: children tasks are not immediately runnable, but are subject to a global order influenced by all other tasks in the program. For example, in Fig. 1(c), (C, 2) creates (B, 4), but running (B, 4) immediately would produce the wrong result, as (B, 3), created by a different parent, must run first. Sequential implementations of these programs use scheduling data structures, such as priority or FIFO queues, to process tasks in the right order. These scheduling structures introduce false data dependences that restrict parallelism and hinder TLS (Sec. 3).

Order constraints limit non-speculative parallelism. For example, in Fig. 1(c), only (B, 4) and (D, 4) can run in parallel without violating correctness. A more attractive option is to use speculation to elide order
constraints. For example, Fig. 1(d) shows a speculative schedule for sssp tasks. Tasks in the same x-axis position are executed simultaneously. This schedule achieves 2× parallelism in this small graph; larger graphs allow more parallelism (Sec. 2.2). This schedule produces the correct result because, although it elides order constraints, it happens to respect data dependences. Unfortunately, data dependences are not known in advance, so speculative execution must detect dependence violations and abort offending tasks to preserve correctness.

Recent work has tried to exploit ordered parallelism using speculative software runtimes [33, 34], but has found that the overheads of ordered, speculative execution negate the benefits of parallelism. This motivates the need for hardware support.

2.2 Analysis of Ordered Irregular Algorithms

To quantify the potential for hardware support and guide our design, we first analyze the parallelism and task structure of several ordered irregular algorithms.

**Benchmarks:** We analyze six benchmarks from the domains of graph analytics, simulation, and databases:
- **bfs** finds the breadth-first tree of an arbitrary graph.
- **sssp** is Dijkstra’s algorithm (Sec. 2.1).
- **astar** uses the A* pathfinding algorithm [31] to find the shortest route between two points in a road map.
- **msf** is Kruskal’s minimum spanning forest algorithm [15].
- **des** is a discrete-event simulator for digital circuits.
- **silo** is an in-memory OLTP database [71].

Sec. 5 describes their input sets and methodology details.

**Analysis tool:** We developed a pintool [46] to analyze these programs in x86-64. We focus on the instruction length, data read and written, and intrinsic data dependences of tasks, excluding the overheads and serialization introduced by the specific runtime used.

The tool uses a simple runtime that executes tasks sequentially. The tool profiles the number of instructions executed and addresses read and written (i.e., the read and write sets) of each task. It filters out reads and writes to the stack, the priority queue used to schedule tasks, and other runtime data structures such as the memory allocator. With this information, the tool finds the critical path length of the algorithm: the sequence of data-dependent tasks with the largest number of instructions. The tool then finds the maximum achievable speedup by dividing the sum of instructions of all tasks by the critical path length [78] (assuming unbounded cores and constant cycles per instruction). Note that this analysis constrains parallelism only by true data dependences: task order dictates the direction of data flow in a dependence, but is otherwise superfluous given perfect knowledge of data dependences.

Table 1 summarizes the results of this analysis. We derive three key insights that guide the design of Swarm:

<table>
<thead>
<tr>
<th>Application</th>
<th>bfs</th>
<th>sssp</th>
<th>astar</th>
<th>msf</th>
<th>des</th>
<th>silo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum parallelism</td>
<td>3440×</td>
<td>793×</td>
<td>419×</td>
<td>158×</td>
<td>1440×</td>
<td>318×</td>
</tr>
<tr>
<td>Parallelism window=1K</td>
<td>827×</td>
<td>178×</td>
<td>62×</td>
<td>147×</td>
<td>198×</td>
<td>125×</td>
</tr>
<tr>
<td>Parallelism window=64</td>
<td>58×</td>
<td>26×</td>
<td>16×</td>
<td>49×</td>
<td>32×</td>
<td>17×</td>
</tr>
<tr>
<td>Instrs mean</td>
<td>22</td>
<td>32</td>
<td>195</td>
<td>40</td>
<td>296</td>
<td>1969</td>
</tr>
<tr>
<td>90th</td>
<td>47</td>
<td>70</td>
<td>508</td>
<td>40</td>
<td>338</td>
<td>2403</td>
</tr>
<tr>
<td>Reads mean</td>
<td>4.0</td>
<td>5.8</td>
<td>22</td>
<td>7.1</td>
<td>50</td>
<td>88</td>
</tr>
<tr>
<td>90th</td>
<td>8</td>
<td>11</td>
<td>51</td>
<td>7</td>
<td>57</td>
<td>110</td>
</tr>
<tr>
<td>Writes mean</td>
<td>0.33</td>
<td>0.41</td>
<td>0.26</td>
<td>0.03</td>
<td>10.5</td>
<td>26</td>
</tr>
<tr>
<td>90th</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>11</td>
<td>51</td>
</tr>
<tr>
<td>Max TLS parallelism</td>
<td>1.03×</td>
<td>1.10×</td>
<td>1.04×</td>
<td>158×</td>
<td>1.15×</td>
<td>45×</td>
</tr>
</tbody>
</table>

**Insight 1:** Parallelism is plentiful. These applications have at least 158× maximum parallelism (msf), and up to 3440× (bfs). Thus, most order constraints are superfluous, making speculative execution attractive.

**Insight 2:** Tasks are small. Tasks are very short, ranging from a few tens of instructions (bfs, sssp, msf), to a few thousand (silo). Tasks are also relatively uniform: 90th-percentile instructions per task are close to the mean. Tasks have small read- and write-sets. For example, sssp tasks read 5.8 64-bit words on average, and write 0.4 words. Small tasks incur large overheads in software runtimes. Moreover, order constraints prevent runtimes from grouping tasks into coarser-grain units to amortize overheads. Hardware support for task management can drastically reduce these overheads.

**Insight 3:** Need a large speculation window. Table 1 also shows the achievable parallelism within a limited task window. With a T-task window, the tool does not schedule an independent task until all work more than T tasks behind has finished. Small windows severely limit parallelism. For example, parallelism in sssp drops from 793× with an infinite window, to 178× with a 1024-task window, to 26× with a 64-task window. Thus, for speculation to be effective, the architecture must support many more speculative tasks than cores.

These insights guide the design of Swarm. Our goal is to approach the maximum achievable parallelism while incurring only moderate overheads.

3. BACKGROUND ON HW SUPPORT FOR SPECULATIVE PARALLELISM

Much prior work has investigated thread-level speculation (TLS) schemes to parallelize sequential programs [25, 28, 60, 61, 66, 69]. TLS schemes ship tasks from function calls or loop iterations to different cores, run them speculatively, and commit them in program order. Although TLS schemes support ordered speculative execution, we find that two key problems prevent them from exploiting ordered irregular parallelism:

1. **False data dependences limit parallelism:** To run under TLS, ordered algorithms must be expressed as sequential programs, but their sequential implementations have limited parallelism. Consider the code in Fig. 1(a), where each iteration dequeues a task from the priority queue and runs it, potentially enqueuing
more tasks. Frequent data dependences in the priority queue, not among tasks themselves, cause frequent conflicts and aborts. For example, iterations that enqueue high-priority tasks often abort all future iterations.

Table 1 shows the maximum speedups that an ideal TLS scheme achieves on sequential implementations of these algorithms. These results use perfect speculation, an infinite task window, word-level conflict detection, immediate forwarding of speculative data, and no communication delays. Yet parallelism is meager in most cases. For example, ssst has 1.1× parallelism. Only msf and silo show notable speedups, because they need no queues: their task orders match loop iteration order.

The root problem is that loops and method calls, the control-flow constructs supported by TLS schemes, are insufficient to express the order constraints among these tasks. By contrast, Swarm implements a more general execution model with timestamp-ordered tasks to avoid software queues, and implements hardware priority queues integrated with speculation mechanisms, avoiding spurious aborts due to queue-related references.

2. Scalability bottlenecks: Although prior TLS schemes have developed scalable versioning and conflict detection schemes, two challenges limit their performance with large speculation windows and small tasks: Forwarding vs selective aborts: Most TLS schemes find it is desirable to forward data written by an earlier, still-speculative task to later reader tasks. This prevents later tasks from reading stale data, reducing mispeculations on tight data dependences. However, it creates complex chains of dependences among speculative tasks. Thus, upon detecting mispeculation, most TLS schemes abort the task that caused the violation and all later speculative tasks [25, 28, 61, 66, 68]. TCC [29] and Bulk [11] are the exception: they do not forward data and only abort later readers when the earlier writer commits.

We find that forwarding speculative data is crucial for Swarm. However, while aborting all later tasks is reasonable with small speculative windows (2–16 tasks are typical in prior work), Swarm has a 1024-task window, and unselective aborts are impractical. To address this, we contribute a novel conflict detection scheme based on eager version management that allows both forwarding speculative data and selective aborts of dependent tasks. Commit serialization: Prior TLS schemes enforce inorder commits by passing a token among ready-to-commit tasks [28, 61, 66, 68]. Each task can only commit when it has the token, and passes the token to its immediate successor when it finishes committing. This approach cannot scale to the commit throughput that Swarm needs. For example, with 100-cycle tasks, a 64-core system should commit 0.64 tasks/cycle on average. Even if commits were instantaneous, the latency incurred by passing the token makes this throughput unachievable.

To tackle this problem, we show that, by adapting techniques from distributed systems, we can achieve inorder commits without serialization, token-passing, or building successor lists.

4. SWARM: AN ARCHITECTURE FOR ORDERED PARALLELISM

Fig. 2 shows Swarm’s high-level organization. Swarm is a tiled, cache-coherent chip multiprocessor (CMP). Each tile has a group of simple cores. Each core has small, private, write-through L1 caches. All cores in a tile share a per-core L2 cache, and each tile has a slice of a shared NUCA L3 cache. Each tile features a task unit that queues, dispatches, and commits tasks. Tiles communicate through a mesh NoC.

Key features: Swarm is optimized to execute short tasks with programmer-specified order constraints. Programmers define the execution order by assigning timestamps to tasks. Tasks can create children tasks with equal or later timestamps than their own. Tasks appear to execute in global timestamp order, but Swarm uses speculation to elide order constraints. Swarm is coherently designed to support a large speculative task window efficiently. Swarm has no centralized structures: each tile’s task unit queues runnable tasks and maintains the speculative state of finished tasks that cannot yet commit. Task units only communicate when they send new tasks to each other to maintain load balance, and, infrequently, to determine which finished tasks can be committed.

Swarm speculates far ahead of the earliest active task, and runs tasks even if their parent is still speculative. Fig. 3(a) shows this process: a task with timestamp 0 is still running, but tasks with later timestamps and several speculative ancestors are running or have finished execution. For example, the task with timestamp 51, currently running, has three still-speculative ancestors, two of which have finished and are waiting to commit (8 and 20) and one that is still running (40).

Allowing tasks with speculative ancestors to execute uncovers significant parallelism, but may induce aborts that span multiple tasks. For example, in Fig. 3(b) a new task with timestamp 35 conflicts with task 40, so 40 is aborted and child task 51 is both aborted and discarded. These aborts are selective, and only affect tasks whose speculative ancestors are aborted, or tasks that have read data written by an aborted task.

We describe Swarm in a layered fashion. First, we present Swarm’s ISA extensions. Second, we describe Swarm hardware assuming that all queues are unbounded. Third, we discuss how Swarm handles bounded queue sizes. Fourth, we present Swarm’s hardware costs.

![Figure 2: Swarm CMP and tile configuration.](image)
We design a low-level C++ API that uses these API:

1. The task queue holds task descriptors (function pointer, task queue, and arguments).
2. The commit queue holds the speculative state of tasks that have finished execution but cannot yet commit.

### 4.1 ISA Extensions and Programming Model

Swarm manages and dispatches tasks using hardware task queues. A task is represented by a descriptor with the following architectural state: the task’s function pointer, a 64-bit timestamp, and the task’s arguments.

Tasks appear to run in timestamp order. Tasks with the same timestamp may execute in any order, but run atomically—the system lazily selects an order for them.

A task can create one or more children tasks with an equal or later timestamp than its own. A child is ordered after its parent, but children with the same timestamp may execute in any order. Because hardware must track parent-child relations, tasks may create a limited number of children (8 in our implementation). Tasks that need more children enqueue a single task that creates them.

Swarm adds instructions to enqueue and dequeue tasks. The enqueue_task instruction accepts a task descriptor (held in registers) as its input and queues the task for execution. A thread uses the dequeue_task instruction to start executing a previously-enqueued task. dequeue_task initiates speculative execution at the task’s function pointer and makes the task’s timestamp and arguments available (in registers). Task execution ends with a finish_task instruction.

dequeue_task stalls the core if an executable task is not immediately available, avoiding busy-waiting. When no tasks are left in any task unit and all threads are stalled on dequeue_task, the algorithm has terminated, and dequeue_task jumps to a configurable pointer to handle termination.

API: We design a low-level C++ API that uses these mechanisms. Tasks are simply functions with signature:

```cpp
void taskFn(timestamp, args...) ...
```

Code can enqueue other tasks by calling:

```cpp
equeueTask(taskFn, timestamp, args...) ...
```

If a task needs more than the maximum number of task descriptor arguments, three 64-bit words in our implementation, the runtime allocates them in memory.

4.2 Task Queuing and Prioritization

The task unit has two main structures:

1. The task queue holds task descriptors (function pointer, timestamp, and arguments).
2. The commit queue holds the speculative state of tasks that have finished execution but cannot yet commit.
based implementations are also possible, but we find the small TCAMs to have a moderate cost (Sec. 4.8).

4.3 Speculative Execution and Versioning

The key requirements for speculative execution in Swarm are allowing fast commits and a large speculative window. To this end, we adopt eager versioning, storing speculative data in place and logging old values. Eager versioning makes commits fast, but aborts are slow. However, Swarm’s execution model makes conflicts rare, so eager versioning is the right tradeoff.

Eager versioning is common in hardware transactional memories [30, 48, 79], which do not perform ordered execution or speculative data forwarding. By contrast, most TLS systems use lazy versioning (buffering speculative data in caches) or more expensive multiversioning [11, 25, 28, 29, 56, 60, 61, 66, 68, 69] to limit the cost of aborts. Some early TLS schemes are eager [25, 80], and they still suffer from the limitations discussed in Sec. 3.

Swarm’s speculative execution borrows from LogTM and LogTM-SE [48, 63, 79]. Our key contributions over these and other speculation schemes are (i) conflict detection (Sec. 4.4) and selective abort techniques (Sec. 4.5) that leverage Swarm’s hierarchical memory system and Bloom filter signatures to scale to large speculative windows, and (ii) a technique that exploits Swarm’s large commit queues to achieve high-throughput commits (Sec. 4.6).

Fig. 6 shows the per-task state needed to support speculation: read- and write-set signatures, an undo log pointer, and child pointers. Each core and commit queue entry holds this state.

A successful dequeue_task instruction jumps to the task’s code pointer and initiates speculation. Since speculation happens at the task level, there are no register checkpoints, unlike in HTM and TLS. Like in LogTM-SE, as the task executes, hardware automatically performs conflict detection on every read and write (Sec. 4.4). Then, it inserts the read and written addresses into the Bloom filters, and, for every write, it saves the old memory value in a memory-resident undo log. Stack addresses are not conflict-checked or logged.

When a task finishes execution, it allocates a commit queue entry; stores the read and write set signatures, undo log pointer, and child pointers there; and frees the core for another task.

4.4 Virtual Time-Based Conflict Detection

Conflict detection is based on a priority order that respects both programmer-assigned timestamps and parent-child relationships. Conflicts are detected at cache line granularity.

**Unique virtual time:** Tasks may have the same programmer-assigned timestamp. However, conflict detection has much simpler rules if tasks follow a total order. Therefore, tasks are assigned a unique virtual time when they are dequeued for execution. Unique virtual time is the 128-bit tuple \((\text{programmer timestamp}, \text{dequeue cycle}, \text{tile id})\). The \((\text{dequeue cycle}, \text{tile id})\) pair is unique since at most one dequeue per cycle is permitted at a tile. Conflicts are resolved using this unique virtual time, which tasks preserve until they commit.

Unique virtual times incorporate the ordering needs of programmer-assigned timestamps and parent-child relations: children always start execution after their parents, so a parent always has a smaller dequeue cycle than its child, and thus a smaller unique virtual time, even when parent and child have the same timestamp.

**Conflicts and forwarding:** Conflicts arise when a task accesses a line that was previously accessed by a later-virtual time task. Suppose two tasks, \(t_1\) and \(t_2\), are running or finished, and \(t_2\) has a later virtual time. A read of \(t_1\) to a line written by \(t_2\) or a write to a line read or written by \(t_2\) causes \(t_2\) to abort. However, \(t_2\) can access data written by \(t_1\) even if \(t_1\) is still speculative. Thanks to eager versioning, \(t_2\) automatically uses the latest copy of the data—there is no need for speculative data forwarding logic [25].

**Hierarchical conflict detection:** Swarm exploits the cache hierarchy to reduce conflict checks. Fig. 7 shows the different types of checks performed in an access:

1. The L1 is managed as described below to ensure L1 hits are conflict-free.
2. L1 misses are checked against other tasks in the tile (both in other cores and in the commit queue).
3. L2 misses, or L2 hits where a virtual time check (described below) fails, are checked against tasks in other tiles. As in LogTM [48], the L3 directory uses memory-backed sticky bits to only check tiles whose tasks may have accessed the line. Sticky bits are managed exactly as in LogTM.

Any of these conflicts trigger task aborts.

**Using caches to filter checks:** The key invariant that allows caches to filter checks is that, when a task with virtual time \(T\) installs a line in the (L1 or L2) cache, that line has no conflicts with tasks of virtual time \(T > T\). As long as the line stays cached with the right coherence permissions, it stays conflict-free. Because conflicts happen when tasks access lines out of virtual time order, if another task with virtual time \(U > T\) accesses the line, it is also guaranteed to have no conflicts.

However, accesses from a task with virtual time \(U < T\) must trigger conflict checks, as another task with intermediate virtual time \(X, U < X < T\), may have accessed the line. \(U\)’s access does not conflict with \(T\)’s, but may conflict with \(X\)’s. For example, suppose a task with virtual time \(X = 2\) writes line \(A\). Then, task \(T = 3\) in another core reads \(A\). This is not a conflict with \(X\)’s
write, so $A$ is installed in $T$’s L1. The core then finishes $T$ and dequeues a task $U = 1$ that reads $A$. Although $A$ is in the L1, $U$ has a conflict with $X$’s write. We handle this issue with two changes. First, when a core dequeues a task with a smaller virtual time than the one it just finished, it flushes the L1. Because LIs are small and write-through, this is fast, simply requiring to flash-clear the valid bits. Second, each L2 line has an associated canary virtual time, which stores the lowest task virtual time that need not perform a global check. For efficiency, lines in the same L2 set share the same canary virtual time. For simplicity, this is the maximum virtual time of the tasks that installed each of the lines in the set, and is updated every time a line is installed. Efficient commit queue checks: Although caches reduce the frequency of conflict checks, all tasks in the tile must be checked on every L2 access and on some global checks. To allow large commit queues (e.g., 64 tasks/queue), commit queue checks must be efficient. To this end, we leverage that checking a $K$-way Bloom filter only requires reading one bit from each way. As shown in Fig. 8, Bloom filter ways are stored in columns, so a single 64-bit access per way reads all the necessary bits. Reading and ANDing all ways yields a word that indicates potential conflicts. For each queue entry whose position in this word is set, its virtual time is checked; those with virtual time higher than the issuing task’s must be aborted.

### 4.5 Selective Aborts

Upon a conflict, Swarm aborts the later task and all its dependent tasks: its children and other tasks that have accessed data written by the aborting task. Hardware aborts each task $t$ in three steps:

1. Notify $t$’s children to abort and be removed from their task queues.
2. Walk $t$’s undo log in LIFO order, restoring old values. If one of these writes conflicts with a later-virtual-time task, wait for it to abort and continue $t$’s rollback.
3. Clear $t$’s signatures and free its commit queue entry.

Applied recursively, this procedure selectively aborts all dependent tasks, as shown in Fig. 10. This scheme has two key benefits. First, it reuses the conflict-detection logic used in normal operation. Undo-log writes (e.g., $A$’s second wr 0x10 in Fig. 10) are normal conflict-checked writes, issued with the task’s timestamp to detect all later readers and writers. Second, this scheme does not explicitly track data dependences among tasks. Instead, it uses the conflict-detection protocol to recover them as needed. This is important, because any task may have served speculative data to many other tasks, which would make explicit tracking expensive. For example, tracking all possible dependences on a 1024-task window using bit-vectors, as proposed in prior work [13, 58], would require $1024 \times 1023 \approx 1$ Mbit of state.

### 4.6 Scalable Ordered Commits

To achieve high-throughput commits, Swarm adapts the virtual time algorithm [38], common in parallel discrete event simulation [21]. Fig. 9 shows this protocol. Tiles periodically send the smallest unique virtual time of any unfinished (running or idle) task to an arbiter. Idle tasks do not yet have a unique virtual time and use $(timestamp, current cycle, tile id)$ for the purposes of this algorithm. The arbiter computes the minimum virtual time of all unfinished tasks, called the global virtual time (GVT), and broadcasts it to all tiles. To preserve ordering, only tasks with virtual time $<$ GVT can commit.

The key insight is that, by combining the virtual time algorithm with Swarm’s large commit queues, commit costs are amortized over many tasks. A single
GVT update often causes many finished tasks to commit. For example, if in Fig. 9 the GVT jumps from (80,100,2) to (98,550,1), all tasks with virtual time \((80,100,2) < t < (98,550,1)\) can commit. GVT updates happen sparingly (e.g., every 200 cycles) to limit bandwidth. Less frequent updates reduce bandwidth but increase commit queue occupancy.

In addition, eager versioning makes commits fast: a task commits by freeing its task and commit queue entries, a single-cycle operation. Thus, if a long-running task holds the GVT for some time, once it finishes, commit queues quickly drain and catch up to execution.

Compared with prior TLS schemes that use successor lists and token passing to reconcile order (Sec. 3), this scheme does not even require finding the successor and predecessor of each task, and does not serialize commits.

For the system sizes we evaluate, a single GVT arbiter suffices. Larger systems may need a hierarchy of arbiters.

### 4.7 Handling Limited Queue Sizes

The per-tile task and commit queues may fill up, requiring a few simple actions to ensure correct operation.

**Task queue virtualization:** Applications may create an unbounded number of tasks and schedule them for a future time. Swarm uses an overflow/underflow mechanism to give the illusion of unbounded hardware task queues [27,41,64]. When the per-tile task queue is nearly full, the task unit dispatches a special, non-speculative coalescer task to one of the cores. This coalescer task removes several non-speculative, idle task descriptors with high programmer-assigned timestamps from the task queue, stores them in memory, and enqueues a splitter task that will re-enqueue the overflowed tasks.

Note that only non-speculative task queue entries can be moved to software. These (i) are idle, and (ii) have no parent (i.e., their parent has already committed). When all entries are speculative, we need another approach.

**Virtual time-based allocation:** The task and commit queues may also fill up with speculative tasks. The general rule to avoid deadlock due to resource exhaustion is to always prioritize earlier-virtual time tasks, aborting other tasks with later virtual times if needed. For example, if a tile speculates far ahead, fills up its commit queue, and then receives a task that precedes all other speculative tasks, the tile must let the preceding task execute to avoid deadlock. This results in specific policies for the commit queue, cores, and task queue.

**Commit queue:** If task \(t\) finishes execution, the commit queue is full, and \(t\) precedes any of the tasks in the commit queue, it aborts the highest-virtual time finished task and takes its commit queue entry. Otherwise, \(t\) stalls its core, waiting for an entry.

**Cores:** If task \(t\) arrives at the task queue, the commit queue is full, and \(t\) precedes all tasks in cores, \(t\) aborts the highest-virtual time task and takes its core.

**Task queue:** Suppose task \(t\) arrives at a task unit but the task queue is full. If some tasks are non-speculative, then a coalescer is running, so the task waits for a free entry. If all tasks in the task queue are speculative, the enqueued request is NACK’d (instead of ACK’d as in

<table>
<thead>
<tr>
<th>Entries</th>
<th>Entry size</th>
<th>Size</th>
<th>Est. area</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task queue</td>
<td>256</td>
<td>51 B</td>
<td>12.75 KB</td>
</tr>
<tr>
<td>Commit filters</td>
<td>64</td>
<td>16×32 B</td>
<td>32 KB (2-port)</td>
</tr>
<tr>
<td>queue</td>
<td>other</td>
<td>64</td>
<td>36 B</td>
</tr>
<tr>
<td>Order queue</td>
<td>256</td>
<td>2×8 B</td>
<td>4 KB (TCAM)</td>
</tr>
</tbody>
</table>

Table 2: Sizes and estimated areas of main task unit structures.

Fig. 5) and the parent task stalls, and retries the enqueue using linear backoff. To avoid deadlock, we leverage that when a task’s unique virtual time matches the GVT, it is the smallest-virtual time task in the system, and cannot be aborted. This task need not keep track of its children (no child pointers), and when those children are sent to another tile, they can be overflowed to memory if the task queue is full. This ensures that the GVT task makes progress, avoiding deadlock.

### 4.8 Analysis of Hardware Costs

We now describe Swarm’s overheads. Swarm adds task units, a GVT arbiter, and modifies cores and L2s.

Table 2 shows the per-entry sizes, total queue sizes, and area estimates for the main task unit structures: task queue, commit queue, and order queue. All numbers are for one per-tile task unit. We assume a 16-tile, 64-core system as in Fig. 2, with 256 task queue entries (64 per core) and 64 commit queue entries (16 per core). We use CACTI [70] for the task and commit queue SRAM areas (using 32 nm ITRS-HP logic) and scaled numbers from a commercial 28 nm TCAM [3] for the order queue area. Task queues use single-port SRAMs. Commit queues use several dual-port SRAMs for the Bloom filters (Fig. 8), which are 2048-bit, 8-way in our implementation, and a single-port SRAM for all other state (unique virtual time, undo log pointer, and child pointers).

Overall, these structures consume 0.55 mm² per 4-core tile, or 8.8 mm² per chip, a minor cost. Enqueues and dequeues access the order queue TCAM, which consumes ~70pJ per access [50]. Moreover, queue operations happen sparingly (e.g. with 100-cycle tasks, one enqueue and dequeue every 25 cycles), so energy costs are small.

The GVT arbiter is simple. It buffers a virtual time per tile, and periodically broadcasts the minimum one.

Cores are augmented with enqueue/dequeue/finish-task instructions (Sec. 4.1), the speculative state in Fig. 6 (530 bytes), a 128-bit unique virtual time, and logic to insert addresses into Bloom filters and to, on each store, write the old value to an undo log. Finally, the L2 uses a 128-bit canary virtual time per set. For an 8-way cache with 64 B lines, this adds 2.6% extra state.

In summary, Swarm’s costs are moderate, and, in return, confer significant speedups.

### 5. EXPERIMENTAL METHODOLOGY

**Modeled system:** We use an in-house microarchitectural, event-driven, sequential simulator based on Pin [46] to model a 64-core CMP with a 3-level cache hierarchy. We use simple IPC-1 cores with detailed timing models for caches, on-chip network, and main memory (adapted from zsim [62]), and also model Swarm
features (e.g., conflict checks, aborts, etc.) in detail. Table 3 details the modeled configuration.

**Benchmarks**: We use the six benchmarks mentioned in Sec. 2.2: bfs, sssp, astar, msf, des, and silo. Table 4 details their provenance and input sets.

For most benchmarks, we use tuned serial and state-of-the-art parallel versions from existing suites (Table 4). We then port each serial implementation to Swarm. Swarm versions use fine-grain tasks, but use the same data structures and perform the same work as the serial version, so differences between serial and Swarm versions stem from parallelism, not other optimizations.

We wrote our own tuned serial and Swarm astar implementations. astar is notoriously difficult to par-
allelize—to scale, prior work in parallel pathfinding sac-
fices solution quality for speed [8]. Thus, we do not have a software-only parallel implementation.

We port silo to show that Swarm can extract ordered parallelism from applications that are typically consid-
ered unordered. Database transactions are unordered in silo. We decompose each transaction into many small ordered tasks to exploit intra-transaction parallelism. Tasks from different transactions use disjoint timestamp ranges to preserve atomicity. This exposes significant fine-grain parallelism within and across transactions.

**Input sets**: We use a varied set of inputs, often from standard collections such as DIMACS (Table 4). bfs operates on an unstructured mesh; sssp and astar use large road maps; msf uses a Kronecker graph; des simulates an array of carry-select adders; and silo runs the TPC-C benchmark on 4 warehouses.

All benchmarks have serial run-times of over two billion cycles (Table 4). We have evaluated other inputs (e.g., random and scale-free graphs), and qualitative dif-
ferences are not affected. Note that some inputs can offer plentiful trivial parallelism to a software algorithm. For example, on large, shallow graphs (e.g., 10 M nodes and 10 levels), a simple bulk-synchronous bfs that op-
erates on one level at a time scales well [43]. But we use a graph with 7.1 M nodes and 2799 levels, so bfs must speculate across levels to uncover enough parallelism.

For each benchmark, we fast-forward to the start of the parallel region (skipping initialization), and report
results for the full parallel region.

**Idealized memory allocation**: Dynamic memory allocation is not simulated in detail, and a scalable solution is left to future work. Only des and silo tasks allocate memory frequently, and data dependences in the sys-
tem’s memory allocator serialize them. In principle, we could build a task-aware allocator with per-core memory pools to avoid serialization. However, building high-per-
formance allocators is complex [26, 65]. Instead, the simulator allocates and frees memory in a task-aware way. Freed memory is not reused until the freeing task commits to avoid spurious dependences. Each allocator operation incurs a 30-cycle cost. For fairness, serial and software-parallel implementations also use this allocator. We believe this simplification will not significantly affect des and silo results when simulated in detail.

6. EVALUATION

We first compare Swarm with alternative implement-
tations, then analyze its behavior in depth.

6.1 Swarm Scalability

Fig. 11 shows Swarm’s performance on 1- to 64-core systems. In this experiment, per-core queue and L2/L3 capacities are kept constant as the system grows, so systems with more cores have higher queue and cache capacities. This captures performance per unit area.

Each line in Fig. 11 shows the speedup of a single application over a 1-core system (i.e., its self-relative speedup). At 64 cores, speedups range from $51 \times (\text{msf})$ to $122 \times (\text{sssp})$, demonstrating high scalability. In addition to parallelism, the larger queues and L3 of larger systems also affect performance, causing super-linear speedups in some benchmarks (sssp, bfs, and astar). We tease apart the contribution of these factors in Sec. 6.3.
6.2 Swarm vs Software Implementations

Fig. 12 compares the performance of the Swarm and software-only versions of each benchmark. Each graph shows the speedup of the Swarm and software-parallel versions over the tuned serial version running on a system of the same size, from 1 to 64 cores. As in Fig. 11, queue and L2/L3 capacities scale with the number of cores.

Swarm outperforms the serial versions by 43–117x, and the software-parallel versions by 2.7–18.2x. We analyze the reasons for these speedups for each application. bfs: Serial bfs does not need a priority queue. It uses an efficient FIFO queue to store the set of nodes to visit. At 1 core, Swarm is 33% slower than serial bfs; however, Swarm scales to 43x at 64 cores. By contrast, the software-parallel version, PBFS [43], scales to 6.0x, then slows down beyond 24 cores. PBFS only works on a single level of the graph at a time, while Swarm speculates across multiple levels.

sssp: Serial sssp uses a priority queue. Swarm is 32% faster at one core, and 117× faster at 64 cores. The software-parallel version uses the Bellman-Ford algorithm [15]. Bellman-Ford visits nodes out of order to increase parallelism, but wastes work in doing so. Threads in Bellman-Ford communicate infrequently to limit overheads [33], wasting much more work than Swarm’s speculative execution. As a result, Bellman-Ford sssp scales to 14× at 64 cores, 8.1× slower than Swarm.

astar: Our tuned serial astar uses a priority queue to store tasks [15]. Swarm outperforms it by 2% at one core, and by 66× at 64 cores.

msf: The serial and software-parallel msf versions sort edges by weight to process them in order. Our Swarm implementation instead does this sort implicitly through the task queue, enqueuing one task per edge and using its weight as the timestamp. This allows Swarm to overlap the sort and edge-processing phases. Swarm outperforms the serial version by 70% at one core and 61× at 64 cores. The software-parallel msf uses software speculation via deterministic reservations [7], and scales to 19× at 64 cores, 3.1× slower than Swarm.

des: Serial des uses a priority queue to simulate events in time order. Swarm outperforms the serial version by 23% at one core, and by 57× at 64 cores. The software-parallel version uses the Chandy-Misra-Bryant (CMB) algorithm [47, 67]. CMB exploits the simulated communication latencies among components to safely execute some events out of order (e.g., if two nodes have a 10-cycle simulated latency, they can be simulated up to 9 cycles away). CMB scales to 21× at 64 cores, 2.7× slower than Swarm. Half of Swarm’s speedup comes from exploiting speculative parallelism, and the other half from reducing overheads.

des: Serial des runs database transactions sequentially without synchronization. Swarm outperforms serial des by 10% at one core, and by 57× at 64 cores. The software-parallel version uses a carefully optimized protocol to achieve high transaction rates [71]. Software-parallel des scales to 8.8× at 64 threads, 6.4× slower than Swarm. The reason is fine-grain parallelism: in Swarm, each task reads or writes at most one tuple. This exposes parallelism within and across database transactions, and reduces the penalty of conflicts, as only small, dependent tasks are aborted instead of full transactions.

Swarm’s benefits on des heavily depend on the amount of coarse-grain parallelism, which is mainly determined by the number of TPC-C warehouses. To quantify this effect, Fig. 13 shows the speedups of Swarm and software-parallel des with 64, 16, 4, and 1 TPC-C warehouses. With 64 warehouses, software-parallel des scales linearly up to 64 cores and is 4% faster than Swarm. With fewer warehouses, database transactions abort frequently, limiting scalability. With a single warehouse, software-parallel des scales to only 2.7×. By contrast, Swarm exploits fine-grain parallelism within each transaction, and scales well even with a single warehouse, by 49× at 64 cores, 18.2× faster than software-parallel des.

Overall, these results show that Swarm outperforms a wide range of parallel algorithms, even when they use application-specific optimizations. Moreover, Swarm implementations use no explicit synchronization and are simpler, which is itself valuable.

6.3 Swarm Analysis

We now analyze the behavior of different benchmarks in more detail to gain insights about Swarm.
Cycle breakdowns: Fig. 14 shows the breakdown of aggregate core cycles. Each set of bars shows results for a single application as the system scales from 1 to 64 cores. The height of each bar is the sum of cycles spent by all cores, normalized by the cycles of the 1-core system (lower is better). With linear scaling, all bars would have a height of 1.0; higher and lower bars indicate sub- and super-linear scaling, respectively. Each bar shows the breakdown of cycles spent executing tasks that are ultimately committed, tasks that are later aborted, spilling tasks from the hardware task queue (using coalescer and splitter tasks, Sec. 4.7), and stalled.

Swarm spends most of the cycles executing tasks that later commit. At 64 cores, aborted work ranges from 1% (bfs) to 27% (des) of cycles. All graph benchmarks spend significant time spilling tasks to memory, especially with few cores (e.g., 47% of cycles for single-core astar). In all benchmarks but msf, spill overheads shrink as the system grows and task queue capacity increases; msf enqueues millions of edges consecutively, so larger task queues do not reduce spills. Finally, cores rarely stall due to full or empty queues. Only astar and msf spend more than 5% of cycles stalled at 64 cores: 27% and 8%, respectively.

Fig. 14 also shows the factors that contribute to super-linear scaling in Fig. 11. First, larger task queues can capture a higher fraction of runnable tasks, reducing spills. Second, larger caches can better fit the working set, reducing the cycles spent executing committed tasks (e.g., silo). However, beyond 4–8 cores, the longer hit latency of the larger NUCA L3 counts against a large speculative window is crucial, as the analysis in Sec. 2.2 showed. The 4096-entry task queue can hold up to 1024 finished tasks before commitment (using coalescer and splitter tasks, Sec. 4.7), and stalled.

Swarm is able to mine 54× parallelism on average (46×–63×).

Table 5 shows the gmean speedups when these idealizations are progressively applied. The left and middle columns show 1- and 64-core speedups, respectively, over the 1-core baseline (without idealizations). While idealizations help both cases, they have a larger impact on the 1-core system. Therefore, the 64-core speedups relative to the 1-core system with the same idealizations (right column) are lower. With all idealizations, this speedup is purely due to exploiting parallelism; 64-core Swarm is able to mine 54× parallelism on average (46×–63×).

Queue Occupancies: Fig. 15 shows the average number of task queue and commit queue entries used across the 64-core system. Both queues are often highly utilized. Commit queues can hold up to 1024 finished tasks (64 per tile). On average, they hold from 216 in des to 821 in astar. This shows that cores often execute tasks out of order, and these tasks wait a significant time until they commit—a large speculative window is crucial, as the analysis in Sec. 2.2 showed. The 4096-entry task queues are also well utilized, with average occupations between 1157 (silo) and 2712 (msf) entries.

Network traffic breakdown: Fig. 16 shows the NoC traffic breakdown at 64 cores (16 tiles). The cumulative injection rate per tile remains well below the saturation injection rate (32 GB/s). Each bar shows the contributions of memory accesses (between the L2s and L3) issued during normal execution, tasks enqueues to other tiles, abort traffic (including child abort messages and rollback memory accesses), and GVT updates. Task enqueues, aborts, and GVT updates increase network traffic by 15% on average. Thus, Swarm imposes small overheads on traffic and communication energy.

Conflict detection energy: Conflict detection requires Bloom filter checks—performed in parallel over commit queue entries (Fig. 7)—and for those entries where the Bloom filter reports a match, a virtual time check to see whether the task needs to be aborted. Both
We conclude that 1024 entries strikes a good balance.

The default 2048-bit 8-way Bloom filters achieve perfor-

we find the falsely unfiltered checks are infrequent. At 64 cores, using precise per-line canary virtual times reduces global conflict checks by 10.3% on average, and improves application performance by less than 1%.

6.4 Sensitivity Studies

We explore Swarm’s sensitivity to several design pa-

Commit queue size: Fig. 17(a) shows the speedups of different applications as we sweep aggregate commit queue entries from 128 (8 tasks per tile) to unbounded; the default is 1024 entries. Commit queues are fundamental to performance: fewer than 512 entries degrade performance considerably. More than 1024 entries confer moderate performance boosts to some applications. We conclude that 1024 entries strikes a good balance between performance and implementation cost for the benchmarks we study.

Bloom filter configuration: Fig. 17(b) shows the rel-

The default 2048-bit 8-way Bloom filters achieve performance within 10% of perfect conflict detection. Smaller Bloom filters cause frequent false positives and aborts in silo and des, which have the tasks with the largest footprint. However, bfs, sssp, and msf tasks access little data, so they are insensitive to Bloom filter size.

Frequency of GVT updates: Swarm is barely sen-

as we vary the period between GVT updates from 50 cycles to 800 cycles (the default is 200 cycles), performance at 64 cores drops from 0.1% in sssp to 3.0% in msf.

6.5 Swarm Case Study: astar

Finally, we present a case study of astar running on a 16-core, 4-tile system to analyze Swarm’s time-varying behavior. Fig. 18 depicts several per-tile metrics, sampled every 500 cycles, over a 100 Kcycle interval: the breakdown of core cycles (top row), commit and task queue lengths (middle row), and tasks commit and abort events (bottom row). Each column shows these metrics for a single tile.

Fig. 18 shows that task queues are highly utilized throughout the interval. As task queues approach their capacity, coalescer tasks kick in, spilling tasks to memory. Commit queues, however, show varied occupancy. As tasks are executed out of order, they use a commit queue entry until they are safe to commit (or are aborted). Most of the time, commit queues are large enough to decouple execution and commit orders, and tiles spend the vast majority of time executing worker tasks.

Occasionally, however, commit queues fill up and cause the cores to stall. For example, tiles stall around the 40 Kcycle mark as they wait for a few straggler tasks to finish. The last of those stragglers finishes at 43 Kcycles, and the subsequent GVT update commits a large number of erstwhile speculative tasks, freeing up substantial commit queue space. These events explain astar’s sensitivity to commit queue size as seen in Fig. 17(a).

Finally, note that although queues fill up rarely, commits tend to happen in bursts throughout the run. This shows that fast commits are important, as they enable Swarm to quickly turn around commit queue entries.

7. ADDITIONAL RELATED WORK

Prior work has studied the limits of instruction-level parallelism under several idealizations, including a large or infinite instruction window, perfect branch prediction and memory disambiguation, and simple program transformations to remove unnecessary data dependences [4, 9, 18, 20, 24, 42, 49, 57, 74]. Similar to our limit study, these analyses find that parallelism is often plentiful (>1000×), but very large instruction windows are needed to exploit it (>100K instructions [42, 57]). Our oracle tool focuses on task-level parallelism, so it misses intra-task parallelism, which is necessarily limited with short tasks. Instead, we focus on removing superfluous dependences in scheduling data structures, uncovering large amounts of parallelism for irregular applications.

Several TLS schemes expose timestamps to software for different purposes, such as letting the compiler schedule loop iterations in Stampede [68], speculating across
barriers in TCC [29], and supporting out-of-order spawn of speculative function calls in Renau et al. [61]. These schemes work well for their intended purposes, but cannot queue or buffer tasks with arbitrary timestamps—they can only spawn new work if there is a free hardware context. Software scheduling would be required to sidestep this limitation, which, as we have seen, would introduce false data dependences and limit parallelism.

Prior work in fine-grain parallelism has developed a range of techniques to reduce task management overheads. Active messages lower the cost of sending tasks among cores [52,73]. Hardware task schedulers such as Carbon [41] lower overheads further for specific problem domains. GPUs [76] and Anton 2 [27] feature custom schedulers for non-speculative tasks. By contrast, Swarm implements speculative hardware task management for a different problem domain, ordered parallelism.

Prior work has developed shared-memory priority queues that scale with the number of cores [2,75], but they do so by relaxing priority order. This restricts them to benchmarks that admit order violations, and loss of order means threads often execute useless work far from the critical path [33,34]. Nikas et al. [51] use hardware transactional memory to partially parallelize priority queue operations, accelerating sssp by 1.8× on 14 cores. Instead, we dispense with shared-memory priority queues: Swarm uses distributed priority queues, load-balanced through random enqueues, and uses speculation to maintain order.

Our execution model has similarities to parallel discrete-event simulation (PDES) [21]. PDES events run at a specific virtual time and can create other events, but cannot access arbitrary data, making them less general than Swarm tasks. Moreover, state-of-the-art PDES engines have overheads of tens of thousands of cycles per event [6], making them impractical for fine-grain tasks. Fujimoto proposed the Virtual Time Machine (VTM), tailored to the needs of PDES [23], which could reduce these overheads. However, VTM relied on an impractical memory system that could be indexed by address and time.

8. CONCLUSIONS

We have presented Swarm, a novel architecture that unlocks abundant but hard-to-exploit irregular ordered parallelism. Swarm relies on a novel execution model based on timestamped tasks that decouples task creation and execution order, and a microarchitecture that performs speculative, out-of-order task execution and implements a large speculation window efficiently. Programs leverage Swarm’s execution model to convey new work to hardware as soon as it is discovered rather than in the order it needs to run, exposing a large amount of parallelism. As a result, Swarm achieves order-of-magnitude speedups on ordered irregular programs, which are key in emerging domains such as graph analytics, data mining, and in-memory databases [34,55,71]. Swarm hardware could also support thread-level speculation and transactional execution with minimal changes.

Swarm also opens several research avenues. First, Swarm’s techniques may benefit a broader class of applications. For instance, Swarm could be applied to automatically parallelize general-purpose programs more effectively than prior TLS systems. Second, we have shown that co-designing the execution model and microarchitecture is a promising approach to uncover parallelism. Investigating new or more general execution models may expose additional parallelism in other domains. Third, Swarm shows that globally sequenced execution can be scalable even with fine-grained tasks. With additional work, Swarm’s techniques could be scaled to multi-chip and multi-machine systems. These topics are the subject of our ongoing and future work.

9. ACKNOWLEDGMENTS

We sincerely thank Nathan Beckmann, Harshad Kasture, Anurag Mukkara, Li-Shiuan Peh, Po-An Tsai, Guowei Zhang, and the anonymous reviewers for their helpful feedback. M. Amber Hassaan and Donald Nguyen graciously assisted with Galois benchmarks. This work was supported in part by C-FAR, one of six SRC STARnet centers by MARCO and DARPA, and by NSF grant CAREER-1452994. Mark Jeffrey was partially supported by a MIT EECS Jacobs Presidential Fellowship and an NSERC Postgraduate Scholarship.

10. REFERENCES
Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management

Andi Drebes  
The University of Manchester  
School of Computer Science  
Oxford Road  
Manchester M13 9PL  
United Kingdom  
andi.drebes@manchester.ac.uk

Antoniu Pop  
The University of Manchester  
School of Computer Science  
Oxford Road  
Manchester M13 9PL  
United Kingdom  
antoniu.pop@manchester.ac.uk

Karine Heydemann  
Sorbonne Universités  
UPMC Univ Paris 06  CNRS, UMR 7606, LIP6  
4, Place Jussieu  
F-75252 Paris Cedex 05, France  
karine.heydemann@lip6.fr

Albert Cohen  
INRIA and École Normale Supérieure  
45 rue d’Ulm  
F-75005 Paris  
France  
albert.cohen@inria.fr

Nathalie Drach  
Sorbonne Universités  
UPMC Univ Paris 06  CNRS, UMR 7606, LIP6  
4, Place Jussieu  
F-75252 Paris Cedex 05, France  
nathalie.drach-temam@upmc.fr

ABSTRACT

Dynamic task-parallel programming models are popular on shared-memory systems, promising enhanced scalability, load balancing and locality. Yet these promises are undermined by non-uniform memory access (NUMA). We show that using NUMA-aware task and data placement, it is possible to preserve the uniform abstraction of both computing and memory resources for task-parallel programming models while achieving high data locality. Our data placement scheme guarantees that all accesses to task output data target the local memory of the accessing core. The complementary task placement heuristic improves the locality of task input data on a best effort basis. Our algorithms take advantage of data-flow style task parallelism, where the privatization of task data enhances scalability by eliminating false dependences and enabling fine-grained dynamic control over data placement. The algorithms are fully automatic, application-independent, performance-portable across NUMA machines, and adapt to dynamic changes. Placement decisions use information about inter-task data dependences readily available in the run-time system and placement information from the operating system. We achieve 94% of local memory accesses on a 192-core system with 24 NUMA nodes, up to 5× higher performance than NUMA-aware hierarchical work-stealing, and even 5.6× compared to static interleaved allocation. Finally, we show that state-of-the-art dynamic page migration by the operating system cannot catch up with frequent affinity changes between cores and data and thus fails to accelerate task-parallel applications.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PACT ’16, September 11-15, 2016, Haifa, Israel  
© 2016 ACM. ISBN 978-1-4503-4121-9/16/09...$15.00  
DOI: http://dx.doi.org/10.1145/2967938.2967946

Keywords

Task-parallel programming; NUMA; Scheduling; Memory allocation; Data-flow programming.

1. INTRODUCTION

High-performance systems are composed of hundreds of general-purpose computing units and dozens of memory controllers to satisfy the ever-increasing need for computing power and memory bandwidth. Shared memory programming models with fine-grained concurrency have successfully harnessed the computational resources of such architectures [3, 33, 28, 30, 31, 10, 19, 8, 9, 7, 35]. In these models, the programmer exposes parallelism through the creation of fine-grained units of work, called tasks, and the specification of synchronization that constrains the order of their execution. A run-time system manages the execution of the task-parallel application and acts as an abstraction layer between the program and the underlying hardware and software environment. That is, the run-time is responsible for bookkeeping activities necessary for the correctness of the execution (e.g., the creation and destruction of tasks and their synchronization), interfacing with the operating system for resource management (e.g., allocation of data and meta-data for tasks, scheduling tasks to cores) and efficient exploitation of the hardware.

This concept relieves the programmer from dealing with details of the target platform and thus greatly improves productivity. Yet it leaves issues related to efficient interaction with system software, efficient exploitation of the hardware, and performance portability to the run-time. On today’s systems with non-uniform memory access (NUMA), memory latency depends on the distance between the requesting cores and the targeted memory controllers. Efficient resource usage through task scheduling needs to go hand in hand with the optimization of memory accesses through the placement of physical pages. That is, memory accesses must
be kept local in order to reduce latency and data must be distributed across memory controllers to avoid contention.

The alternative of abstracting computing resources only and leaving NUMA-specific optimization to the application is far less attractive. The programmer would have to take into account the different characteristics of all target systems (e.g., the number of NUMA nodes, their associated amount of memory and access latencies), to partition application data properly and to place the data explicitly using operating system-specific interfaces. For applications with dynamic, data-dependent behavior, the programmer would also have to provide mechanisms that constantly react to changes throughout the execution as an initial placement with high data locality at the beginning might have to be revised later on. Such changes would have to be coordinated with the run-time system to prevent destructive performance interference, introducing a tight and undesired coupling between the run-time and the application.

On the operating system side, optimizations are compelled to place tasks and data conservatively [13, 24], unless provided with detailed affinity information by the application [5, 6], high-level libraries [26] or domain specific languages [20]. Furthermore, as task-parallel run-times operate in user-space, a separate kernel component would add additional complexity to the solution; this advocates for a user-space approach.

This paper shows that it is possible to efficiently and portably exploit dynamic task parallelism on NUMA machines without exposing programmers to the complexity of these systems, preserving a simple, uniform abstract view for both memory and computations, yet achieving high locality of memory accesses. Our solution exploits the task-parallel data-flow programming style and its transparent privatization of task data. This allows the run-time to determine a task’s working set, enabling transparent, fine-grained control over task and data placement.

Based on the properties of task-private data, we propose a dynamic task and data placement algorithm to ensure that input and output data are local and that interacts constructively with work-stealing to provide load-balancing across both cores and memory controllers:

- Our memory allocation mechanism, called deferred allocation, avoids making early placement decisions that could later harm performance. In particular, the memory to store task output data is not allocated until the task placement is known. The mechanism hands over this responsibility to the producer task on its local NUMA node. This scheme guarantees that all accesses to task output data are local. Control over data placement is obtained through the privatization of task output data.
- To enhance the locality of read memory accesses, we build on earlier work [14] and propose enhanced work-pushing, a work-sharing mechanism that interacts constructively with deferred allocation. Since the inputs of a task are outputs of another task, the location of input data is determined by deferred allocation when the producer tasks execute. Enhanced work-pushing is a best-effort mechanism that places a task according to these locations before task execution and thus before allocating memory for the task’s outputs.

This combination of enhanced work-pushing and deferred allocation is fully automatic, application-independent, portable across NUMA machines and transparently adapts to dynamic changes at run time. The detailed information about the affinities between tasks and data required by these techniques is either readily available or can be obtained automatically in the run-times of task-parallel programming models with inter-task dependences, such as StarSs [30], OpenMP 4 [28], SWAN [33] and OpenStream [31], which allow the programmer to make inter-task data dependences explicit. While specifying the precise task-level data-flow rather than synchronization constraints alone requires more initial work for programmers, this effort is more than offset by the resulting benefits in terms of performance and performance portability.

The paper is organized as follows. Section 2 presents the principles of enhanced work-pushing. For a more complete discussion of our solutions, we propose multiple heuristics taking into account the placement of input data, output data or both. Section 3 introduces deferred allocation, including a brief outline of the technical solutions employed for fine-grained data placement. Sections 4 and 5 present the experimental methodology and results. A comparison with dynamic page migration is presented in Section 6. Section 7 discusses the most closely related work, before we conclude in Section 8.

2. TASK SCHEDULING WITH ENHANCED WORK-PUSHING

Let us start with terminology and hypotheses about the programming and execution models.

2.1 An abstract model for task parallelism

Our solutions are based on shared memory task-parallel programming models with data dependences. Each task is associated with a set of incoming data dependences and a set of outgoing data dependences, as illustrated by Figure 1. Each dependence is associated with a contiguous region of memory, called input buffer and output buffer for incoming and outgoing dependences, respectively. The addresses of these buffers are collected in the task’s frame, akin to the activation frame storing a function’s arguments and local variables in the call stack. While the frame is unique and allocated at the task’s creation time, its input and output buffers may be placed on different NUMA nodes and allocated later in the life cycle of the task, but no later than the beginning of the execution of the task, reading from input buffers and writing into output buffers. Buffer placement and allocation time have a direct influence on locality and task-data affinity. Since we ought to offer a uniform abstraction of NUMA resources, we assume input and output buffers are managed by the run-time system rather than explicitly allocated by the application. This is the case of programming models such as OpenStream [31] and CnC [8], but not of StarSs [30] and OpenMP 4.0; see Section 7 for further discussion. We say that a task \( t_i \) depends on another task \( t_p \) if \( t_p \) has an outgoing dependence associated to a buffer \( b \) and if \( t_i \) has an incoming dependence associated to \( b \). In this scenario the task \( t_p \) is referred to as the producer and \( t_i \) is the consumer.

Although this is not a fundamental requirement in our work, we will assume for simplicity that a task executes from beginning to end without interruption. A task becomes ready for execution when all of its dependences have
Figure 1: Most general case for a task $t$: $n$ inputs of size $\delta_0^t, \ldots, \delta_{n-1}^t$, $m$ outputs of size $\delta_0^o, \ldots, \delta_{m-1}^o$, data placed on $n + m$ NUMA nodes $N_0^t, \ldots, N_{n-1}^t, N_0^o, \ldots, N_{m-1}^o$.

been satisfied, i.e., when its producers of input data have completed and when its consumers have been created with the addresses of their frames communicated to the task. The working set of a task is defined as the union of all memory addresses that are accessed during its execution. Note that the working set does not have to be identical to the union of a task’s input and output buffers (e.g., a task may access globally shared data structures). However, since our algorithms require accurate communication volume information, we assume that the bulk of the working set of each task is constituted by its input and output data. This is the case for all of the benchmarks studied in the experimental evaluation.

A worker thread is responsible for the execution of tasks on its associated core. Each worker is provided with a queue of tasks ready for execution from which it pops and executes tasks. When the queue is empty, the worker may obtain a task from another one through work-stealing [4]. A task is pushed to the queue by the worker that satisfies its last remaining dependence. We say that this worker activates the task and becomes its owner.

The execution of a task-parallel program starts with a root task derived from the main function of a sequential process. New tasks are created dynamically. The part of the program involved in creating new tasks is called the control program. If only the root task creates other tasks we speak of a sequential control program, otherwise of a parallel control program.

2.2 Weaknesses of task parallelism on NUMA systems

Whether memory accesses during the execution of a task target the local memory controller of the executing core or some remote memory controller depends on the placement of the input and output buffers and on the worker executing the task. These affinities are highly dynamic and can depend on many factors, such as:

- the order of task creations by the control program;
- the execution order of producers;
- the duration of each task;
- work-stealing events;
- or resource availability (e.g., available memory per node).

In earlier work [14], we showed that some of these issues can be mitigated by using work-pushing. Similar to the abstract model discussed above, the approach assumes that tasks communicate through task-private buffers. However, it also assumes that all input data of a task is stored in a single, contiguous memory region rather than multiple input buffers. As a consequence, the task’s input data is entirely located on a single node. This property is used by the work-pushing technique, in which the worker activating a task only becomes its owner if the task’s input data is stored on the worker’s node. If the input data is located on another node, the task is transferred to a random worker associated to a core on that node. The approach remains limited however: (1) as it assumes that all inputs are located on the same node it is ill-suited for input data located on multiple nodes, and (2) it does not optimize for outgoing dependences.

Below, we first present enhanced work-pushing, a generalization of work-pushing, capable of dealing with input data distributed over multiple input buffers potentially placed on different nodes. This technique serves as a basis for the complementary deferred allocation technique, presented in the next section, that allows the run-time to improve the placement of output buffers. We introduce three work-pushing heuristics that schedule a task according to the placement of its input or output data or both. This complements the study of the effects of work-pushing on data locality. And experimentally in Section 5, it enables us to show the limitations of NUMA-aware scheduling alone, limited to passive reactions to a given data placement.

2.3 Enhanced work-pushing

The names of the three heuristics for enhanced work-pushing are input-only, output-only and weighted. The first two heuristics take into account incoming and outgoing dependences only, respectively. The weighted heuristic takes into account all dependences, but associates different weights to incoming and outgoing dependences to honor the fact that read and write accesses usually do not have the same latency.

Algorithm 1 shows how the heuristics above are used by the function activate, which is called when a worker $w$ activates a task $t$ (i.e., when the task becomes ready). Lines 1 to 3 define variables $u_{in}$ and $u_{out}$, indicating which types of dependences should be taken into account according to the heuristic $h$. The variables used to determine whether the newly activated task needs to be transferred to a remote node are initialized in lines 5 to 8: the data array stores the cumulated size of input and output buffers of $t$ for each of the $N$ nodes of the system, $D_{in}$ stands for the incoming dependences of $t$ and $D_{out}$ for its outgoing dependences.

The for loop starting at Line 10 iterates over a list of triples with a set of dependences $D_{cur}$, a variable $u_{cur}$ indicating whether the set should be taken into account, and a weight $w_{cur}$ associated to each type of dependence. During the first iteration, $D_{cur}$ is identical with $D_{in}$ and during the second iteration identical with $D_{out}$. For each dependence in $D_{cur}$, a second loop in Line 12 determines the buffer $b$ used by the dependence, the size $s_b$ of the buffer as well as the node $n_b$ containing the buffer. The node on which a buffer is placed might be unknown if the buffer has not yet been placed by the operating system, e.g., using the first-touch scheme and the buffer has been allocated but not yet written. If this is the case, its size is added to the total size $s_{out}$, but not included into the per-node statistics. Otherwise, the data array is updated accordingly by multiplying $s_{in}$ with the weight $w_{cur}$.

Once the total size and the weighted number of bytes per node have been determined, the procedure checks whether the task should be pushed to a remote node. Tasks whose overall size of dependences is below a threshold are added
Algorithm 1: activate(w, t)

1 if h = input only then (u_{in}, u_{out}) ← (true, false)
2 else if h = output only then (u_{in}, u_{out}) ← (false, true)
3 else if h = weighted then (u_{in}, u_{out}) ← (true, true)
4
5 data[0,...,N-1] ← ⟨0,...,0⟩
6 D_{in} ← inc_deps(t)
7 D_{out} ← out_deps(t)
8 s_{init} ← 0
9
10 for (D_{cur}, u_{cur}, w_{cur}) in
11     ⟨(D_{in}, u_{in}, w_{in}), (D_{out}, u_{out}, w_{out})⟩ do
12         if u_{cur} = true then
13             for d ∈ D_{cur} do
14                 s_{b} ← size_of(buffer_of(d))
15                 n_{b} ← node_of(buffer_of(d))
16                 s_{init} ← s_{init} + s_{b}
17                 if n_{b} ≠ unknown then
18                     data[n_{b}] ← data[n_{b}] + w_{cur} · s_{b}
19                 end
20             end
21         end
22
23 if s_{init} < threshold then
24     add_to_local_queue(w, t)
25 else
26     n_{min} ← node_with_min_access_cost(data)
27     if n_{min} ≠ local_node_of_worker(w) then
28         w_{max} ← random_worker_on_node(n_{min})
29         res ← transfer_task(t, w_{max})
30         if res = failure then
31             add_to_local_queue(w, t)
32         end
33     else
34         add_to_local_queue(w, t)
35     end
36 end
37

This algorithm avoids the overhead of a remote push cannot be compensated by the improvement on task execution time. For tasks with longer dependences, the run-time determines the node n_{min} with the minimal overall access cost (Line 29). The access cost for a node N_i is estimated by summing up the access costs to each node N_j containing at least one of the buffers, which in turn can be estimated by multiplying the average latency between N_i and N_j. If n_{min} is different from the local node n_{local} of the active worker, the run-time tries to transfer t to a random worker on n_{min}. If this fails, e.g., if the data structure of the targeted worker receiving remotely pushed tasks is full, the task is added to the local queue (Line 32).

2.4 Limitations of enhanced work-pushing

The major limitation of enhanced work-pushing is that, regardless of the heuristic, it can only react passively to a given data placement. This implies that data must already be well-distributed across memory controllers if all tasks should take advantage of this scheduling strategy. For poorly distributed data, e.g., if all data is placed on a single node, a subset of the workers receives a significantly higher amount of tasks than others. Work-stealing redistributes tasks among the remaining workers and thus prevents the system from load imbalance, but cannot improve overall data locality if the initial data distribution was poor. Classical task parallel run-times allocate buffers during task creation [8, 33, 31]; hence data distribution mainly depends on the control program. A sequential control program leads to poorly placed data, while a parallel control program lets work-stealing evenly distribute task creation and buffer allocation. However, writing a parallel control program is already challenging in itself, even for programs with regularly-structured task graphs. Additionally ensuring an equal distribution of data across NUMA nodes through the control program is even more challenging or infeasible, especially for applications with less regularly-structured task graphs (e.g., if the structure of the graph depends on input data). Such optimizations also reject efficient exploitation of NUMA to the programmer and are thus contrary to the idea of abstraction from the hardware by the run-time.

In the following section, we introduce a NUMA-aware allocator that complements the input only work-pushing heuristic and that decouples data locality from the control program, leaving efficient exploitation to the run-time.

3. DEFERRED ALLOCATION

NUMA-aware allocation controls the placement of data on specific nodes. Our proposed scheme to make these decisions transparent relies on per-node memory pools to control the placement of task buffers.

3.1 Per-node memory pools

Per-node memory pools combine a mechanism for efficient reuse of blocks of memory with the ability to determine on which nodes blocks are placed. Each NUMA node has a memory pool that is composed of k free lists L_0 to L_{k-1}, where L_i contains blocks of size 2^{k_{min}+i} bytes. When a worker allocates a block of size s, it determines the corresponding list L_j with 2^{k_{min}+j} < s ≤ 2^{k_{min}+j+1} and removes the first block of that list. If the list is empty, it allocates a larger chunk of memory from the operating system, removes the first block from the chunk and adds the remaining parts to the free list.

A common allocation strategy of operating systems is first-touch allocation, composed of two steps. The first step referred to as logical allocation is triggered by the system call used by the application to request additional memory and only extends the application’s address space. The actual physical allocation is triggered upon the first write to the memory region and places the corresponding page on the same node as the writing core. Hence, a block that originates from a newly allocated chunk is not necessarily placed on any node.

However, when a block is freed, it has been written by a producer and it is thus safe to assume that the block has been placed through physical allocation. The identifier of the containing node can be obtained through a system call, which enables the run-time to return the block to the correct memory pool. To avoid the overhead of a system call each time a block is freed, information on the NUMA node containing a block is cached in a small meta-data section associated to the block. This memory pooling mechanism provides three fundamental properties for deferred allocation presented below. First, it ensures that allocating a block
from a memory pool always returns a block that has not been placed yet or a block that is known to be placed on the node associated to the memory pool. Second, data can be placed on a specific node with very low overhead. Finally, the granularity for data placement is decoupled from the usual page granularity as a block may be sized arbitrarily.

3.2 Principles of deferred allocation

The key idea of deferred allocation is to delay the allocation and thus the placement of each task buffer until the node executing the producer that writes to the buffer is known. This guarantees that accesses to output buffers are always local. The classical approach in run-times for dependent tasks is to allocate input buffers upon the creation of a task [8, 33, 31] or earlier [30]. Instead, we propose to let each input buffer for a consumer task $t_c$ be allocated by the producer task $t_p$, writing into it, immediately before task $t_c$ starts execution. Since the input buffer of $t_c$ is an output buffer of $t_p$, the location of input data in $t_c$ is effectively determined by its producer(s). In the following, we use the term immediate allocation to distinguish the default allocation scheme in which input buffers are allocated upon creation from deferred allocation.

Figure 2a shows the implications of immediate allocation on data locality for a task $t$. All input buffers of $t$ are allocated on the node $N_c$ on which the creator of $t$ operates. The same scheme applies to the creators $t^{0}_{i,o}$ to $t^{m-1}_{i,o}$, causing the input buffers of the tasks $t^0_i$ to $t^{m-1}_i$ to be allocated on nodes $N^0_i$ to $N^{m-1}_i$, respectively. In the worst case for data locality, $t$ is stolen by a worker operating on neither $N_c$ nor $N^0_i$ to $N^{m-1}_i$ and all memory accesses of $t$ target memory on remote nodes.

When using deferred allocation, the input buffers of $t$ are not allocated before its producers start execution and the output buffers of $t$ are not allocated before $t$ is activated (Figure 2b). When $t$ becomes ready, all of its input buffers have received input data from the producers of $t$ and have been placed on up to $n$ different nodes $N^0_i$ to $N^{m-1}_i$ (Figure 2c). The data locality impact of deferred allocation is illustrated in Figure 2d, showing the placement at the moment when the worker executing $t$ has been determined. Regardless of any possible placement of $t$, all of its output buffers are placed on the same node as the worker executing the task. Hence, using deferred allocation, write accesses are guaranteed to target local memory. Furthermore, this property is independent from the placement of the creating tasks $t_c$ and $t^0_{i,o}$ to $t^{m-1}_{i,o}$, which effectively decouples data locality from the control program. Even for a sequential control program, data is distributed over the different nodes of the machine according to work-stealing. This way, work-stealing does not only take the role of a mechanism responsible for computational load balancing, but also the role as a mechanism for load balancing across memory controllers.

An important side effect of deferred allocation is a significant reduction of the memory footprint. With a sequential control program, all tasks are created by a single “root” task. This causes a large number of input buffers to be allocated early on, while the actual working set of live buffers might be much smaller. A parallel control program can mitigate the effects of early allocation, e.g., by manually throttling task creation as shown in Figure 3b. However, this requires significant programmer effort and hurts the separation of concerns that led to the delegation of task management to the run-time.

Thanks to deferred allocation, buffers allocated for early tasks can be reused at a later stage. The difference is shown in Figures 3a and 3c. In the first case, all three buffers $b_{i}, b_{i+1}$ and $b_{i+2}$ are allocated before the dependent tasks $t_i$ to $t_{i+3}$ are executed. In the latter case, the buffer used by $t_i$ and $t_{i+1}$ can be reused as the input buffer of $t_{i+3}$. Parallel control programs also benefit from deferred allocation as the minimal number of buffers along a path of dependent tasks can be decreased by one (e.g., in Figure 3b only $b_i$ and $b_{i+1}$ are simultaneously live and $b_i$ can be reused for $b_{i+2}$ when using deferred allocation).

3.3 Compatibility with work-pushing

Deferred allocation guarantees local write accesses, but it does not influence the locality of read accesses. By combining deferred allocation with the input-only heuristic of enhanced work-pushing, it is possible to optimize for both read and write accesses.

It is important to note that neither the output-only heuristic nor the weighted heuristic can be used since the output buffers of a task are not determined upon task activation when the work-pushing decision is taken.

4. EXPERIMENTAL SETUP

For the experimental evaluation we implemented enhanced work-pushing and deferred allocation in the run-time system of the OpenStream project [29]. We start with an overview of the software and hardware environment used in our experiments, followed by a presentation of the selected benchmarks.

4.1 Software environment

OpenStream [31] is a task-parallel, data-flow programming model implemented as an extension to OpenMP. Arbitrary dependence patterns can be used to exploit task, pipeline and data parallelism. Each data-flow dependence is semantically equivalent to a communication and synchronization event within an unbounded FIFO queue referred to as a stream. Pragmatically, this is implemented by compiling dependences as accesses to task buffers dynamically allocated at execution time: writes to streams result in writes to the buffers of the tasks consuming the data, while read accesses to streams by consumer tasks are translated to reads from their own, task-private buffers.

We implemented the optimizations presented in this paper into the publicly available run-time of OpenStream [29]. Crucially, we rely on the fact that OpenStream programs are written with programmer annotations explicitly describing the flow of data between tasks. This precise data-flow information is preserved during compilation and made available to the run-time library. We leverage this essential semantic information to determine, at run-time and before task execution, how much data is exchanged by any given task.

OpenStream programs are dynamically load-balanced, worker threads use hierarchical work-stealing to acquire and execute tasks whose dependences have been satisfied. If work-pushing is enabled, workers can also receive tasks in a dedicated multi-producer single-consumer queue [14]. Our experiments use one worker thread per core.
4.2 Hardware environment

The experiments were conducted on two many-core systems.

**Opteron-64** is a quad-socket system with four AMD Opteron 6282 SE processors running at 2.6GHz, using Scientific Linux 6.2 with kernel 3.10.1. The machine is composed of 4 physical packages, with 2 dies per package, each die containing 8 cores organized in pairs. Each pair shares the first-level instruction cache as well as a 2MiB L2 cache. An L3 cache of 6MiB and the memory controller are shared by the 8 cores on the same die. The 16KiB L1 cache is private to each core. Main memory is 64GiB, equally divided into 8 NUMA domains. For each NUMA node, 4 neighbors are at a distance of 1 hop and 3 neighbors are at 2 hops.

**SGI-192** is an SGI UV2000 with 192 cores and 756GiB RAM, distributed over 24 NUMA nodes, and running SUSE Linux Enterprise Server 11 SP3 with kernel 3.0.101-0.46-default. The system is organized in *blades*, each of which contains two Intel Xeon E5-4640 CPUs running at 2.4GHz. Each CPU has 8 cores with direct access to a memory controller. The cache hierarchy consists of 3 levels: a core-private L1 with separate instruction and data cache, each with a capacity of 32KiB; a core-private, unified L2 cache of 256KiB; and a unified L3 cache of 20MiB, shared among all 8 cores of the CPU. *Hyperthreading* was disabled for our experiments. Each blade has a direct connection to a set of other blades and indirect connections to the remaining ones. From a core’s perspective, a memory controller can be either local if associated to the same CPU, at 1 hop if on the same blade, at 2 hops if on a different blade that is connected directly to the core’s blade or at 3 hops if on a remote blade with an indirect connection.

**Latency of memory accesses and NUMA factors.**

We used a synthetic benchmark to measure the latency of memory accesses as a function of the distance in hops between a requesting core and the memory controller that satisfies the request. It allocates a buffer on a given node using LibNUMA, initializes it and measures execution time for a sequence of memory accesses to this buffer from a core on a specific node. Each sequence traverses the whole buffer from beginning to end in steps of 64 bytes, such that each cache line is only accessed once. The buffer size was set to 1GiB to ensure data is evicted from the cache before it is reused and thus to measure only memory accesses that are satisfied by the memory controller and not by the hierarchy of caches.

Tables 1 and 2 indicate the total execution time as a function of the number of hops and the access mode of the synthetic benchmark for both systems. The results show that latency increases with the distance between the requesting core and the targeted memory controller and that writes are significantly slower than reads. The rightmost column of each table shows the access time normalized to accesses targeting local memory. For reads on the Opteron-64 system, these values range from 1.81 for on-package accesses to a memory controller at a distance of one hop to a factor of 4.34 for off-package accesses at a distance of two hops. For writes, these values are lower—1.2 to 2.48—due to the higher latency of local writes. Not surprisingly, the factors for both reads and writes are significantly higher on the larger SGI-192 system (up to 7.48 for reads at three hops). This suggests that locality optimizations will have a higher impact on SGI-192 and that the locality of writes will have the greatest impact.

4.3 Benchmarks

We evaluate the impact of our techniques on nine benchmarks, each of which is available in an optimized sequential implementation and two tuned parallel implementations using OpenStream.

The first parallel implementation uses task-private input and output buffers as described in Section 2.1 and thus enables enhanced work-pushing and deferred allocation. Data from input buffers is only read and never written, while data in output buffers is only written and never read. Hence, tasks cannot perform in-place updates and results are writ-
ten to a different location than the input data. We refer to this implementation as DSA (dynamic single assignment).

The second parallel implementation, which we refer to as SHM, uses globally shared data structures and thus does not expose information on memory accesses to the run-time. However, the pages of the data structures are distributed across all NUMA nodes in a round-robin fashion using interleaved allocation. We use this implementation to compare our solutions to classical static NUMA-aware optimizations that require only minimal changes to the application. The benchmarks are the following.

- **Jacobi-1d**, **jacobi-2d** and **jacobi-3d** are the usual one-, two- and three-dimensional Jacobi stencils iterating over arrays of double precision floating point elements. At each iteration, the algorithm averages for each matrix element the values of the elements in its Von Neumann neighborhood using the values from the previous iteration.

- **Seidel-1d**, **seidel-2d** and **seidel-3d** employ a similar stencil pattern but use values from the previous and the current iteration for updates.

- **Kmeans** is a data-mining benchmark that partitions a set of n d-dimensional points into k clusters using the K-means clustering algorithm. Each vector is represented by d single precision floating point values.

- **Blur-roberts** applies two successive image filters on double precision floating point elements [22]: a Gaussian blur filter on each pixel’s Moore neighborhood followed by the Roberts Cross Operator for edge detection.

- **Bitonic** implements a bitonic sorting network [1], applied to a sequence of arbitrary 64-bit integers.

Table 3 summarizes the parameters for the different benchmarks and machines. The size of input data was chosen to be significantly higher than the total amount of cache memory and low enough to prevent the system from swapping. This size is identical on both machines, except for **blur-roberts**, whose execution time for images of size $2^{15} \times 2^{15}$ is too short on SGI-192 and which starts swapping for size $2^{16} \times 2^{16}$ on Opteron-64. To amortize the execution of auxiliary tasks at the beginning and the end of execution of the stencils, we set the number of iterations to 60.

The block size has been tuned to minimize the execution time for the parallel implementation with task-private input and output data (DSA) on each machine. To avoid any bias in favor of our optimizations, enhanced work-pushing and deferred allocation have been disabled during this tuning phase. In this configuration, the run-time only relies on optimized work-stealing [23] extended with hierarchical work-stealing [14] for computational load balancing. We refer to this baseline for our experiments as DSA-BASE. Identical parameters for the block size and run-time have been used for the experiments with the shared memory versions of the benchmarks (SHM), which we refer to as SHM-BASE.

All benchmarks were compiled using the OpenStream compiler based on GCC 4.7.0. The compilation flags for **blur-roberts** as well as the **jacobi** and **seidel** benchmarks were -O3 -fno-fast-math, while **kmeans** uses -O3 and **bitonic** uses -O2.

The parallel implementations are provided with a parallel control program to prevent sequential task creation from becoming a performance bottleneck. To avoid memory controller contention, the initial and final data are stored in global data structures allocated using interleaved allocation across all NUMA nodes.

### Data dependence patterns

The relevant producer-consumer patterns shown in Figure 4 can be divided into three groups with different implications for our optimizations: unbalanced dependences (e.g., one input buffer accounting for more than 90% of the input data) with long dependence paths (jacobi-1d, jacobi-2d, jacobi-3d, seidel-1d, seidel-2d, seidel-3d, kmeans), unbalanced dependences with short dependence paths (blur-roberts) and balanced dependences (bitonic). The behavior of our heuristics on these patterns is referenced in the experimental evaluation. All of the benchmarks have non-trivial, connected task graphs, i.e., none of the benchmarks represents an embarrassingly parallel workload.

### Characterization of memory accesses

The benchmarks were carefully tuned (block sizes and tiling) to take advantage of caches. However, the effectiveness of the cache hierarchy also depends on the pattern, the frequency and the timing of memory accesses during the execution of a benchmark, leading to more or fewer cache misses for a given block size. Figure 5 shows the cache miss rates at the last level of cache (LLC) on SGI-192 of DSA-BASE, which is a good proxy for the rate of requests to main memory for each benchmark. For all bar graphs in this paper, error bars indicate standard deviation. As the focus of our optimizations is on the locality of memory accesses, we expect a higher impact for benchmarks exhibiting higher LLC miss rates. For this reason, seidel and blur-roberts are expected to benefit the most from our optimizations, followed by the jacobi benchmarks and bitonic. **Kmeans** has a very low LLC miss rate and is not expected to show significant improvement.

### 4.4 Experimental baseline

To demonstrate the effectiveness of our optimizations, our principal point of comparison is DSA-BASE. We validate
the soundness of this baseline by comparing its performance on Cholesky factorization against the two-state-of-the-art linear algebra libraries PLASMA/QUARK [35] and Intel’s MKL [18]. Figure 6 shows the execution times of Cholesky factorization on a matrix of size $8192 \times 8192$ running on the Opteron-64 platform with four configurations: DSA-BASE, Intel MKL, PLASMA and finally optimized OpenStream (DSA-OPT), the latter using our run-time implementing the optimizations presented in this paper. This validates the soundness of our baseline, which achieves similar performance to Intel MKL, while also showcasing the effectiveness of our optimizations, automatically and transparently matching the performance of PLASMA without any change in the benchmark’s source code.

5. RESULTS

We now evaluate enhanced work-pushing and deferred allocation, starting with the impact on memory access locality, and following on with the actual performance results.

5.1 Impact on the locality of memory accesses

On the Opteron platform, we use two hardware performance counters to count the requests to node-local memory\(^2\) and to remote nodes,\(^3\) respectively. We consider the locality metric $R_{\text{loc}}^{\text{HW}}$, defined as the ratio of local memory accesses to total memory accesses, shown in Figure 7. We could not provide the corresponding figures for the SGI system due to missing support in the kernel. However, the OpenStream run-time contains precise information on the working set of tasks and on the placement of input buffers, which can be used to provide a second locality metric $R_{\text{loc}}^{\text{RT}}$ that precisely accounts for accesses to data managed by the run-time, i.e., associated to task dependences.

5.2 Impact on performance

Figure 9 shows the speedup achieved over DSA-BASE. The best performance is achieved by combining work-pushing and deferred allocation, with a global maximum of $2.5 \times$ on Opteron-64 and $5.0 \times$ on SGI-192. Generally, the speedups

---

\(^2\) CPU\_IO\_REQUESTS\_TO\_MEMORY\_TO\_LOCAL, CPU\_IO\_TO\_LOCAL\_MEM

\(^3\) CPU\_IO\_REQUESTS\_TO\_MEMORY\_TO\_LOCAL, CPU\_IO\_TO\_REMOTE\_MEM
are higher on SGI-192, showing that our optimizations have a higher impact on machines with higher penalties for remote accesses. These improvements result from better locality as well as from the memory footprint reduction induced by deferred allocation. Note that the input-only heuristic did not perform well with *jacobi*, highlighting the importance of considering both input and output flows, and of proactively distributing buffers through deferred allocation rather than reactively adapting the schedule only.

### 5.3 Comparison with interleaved allocation

Figure 10 shows the speedup of the DSA parallel baseline over the implementations using globally shared data structures distributed over all NUMA nodes using interleaved allocation (SHM-BASE). The optimizations achieve up to 3.1× speedup on Opteron-64 and 5.6× on SGI-192. The best performance is systematically obtained by the combined work-pushing and deferred allocation strategy. These results clearly indicate that taking advantage of the dynamic data-flow information provided in modern task-dependent languages allows for more precise control over the placement of data leading to improved performance over static schemes unable to react to dynamic behavior at execution time. In the case of interleaved allocation, the uniform access pattern of the benchmarks evaluated in this work yields good load balancing across memory controllers, but poor data locality.

### 6. COMPARISON WITH DYNAMIC PAGE MIGRATION

While our study focused on the application and run-time, the reader may wonder how kernel-level optimizations fare in our context. Recent versions of Linux comprise the *balancer-numa* patchset [12] for dynamic page migration and a transparent policy migrating pages during the execution of a process based on its memory accesses. The kernel periodically scans the address space of the process and changes the protection flags of the scanned pages such that an access causes an exception. Upon an access to such a page, the exception handler checks whether the page is misplaced with respect to NUMA and migrates it towards the node of the accessing CPU.

We first evaluate the influence of page migration on a synthetic benchmark to determine under which conditions the mechanism is beneficial and show that these conditions do not meet the requirements for task-parallel programs. We then study the impact of dynamic page migration on the SHM parallel baseline with globally shared data (SHM-BASE).

#### 6.1 Parametrization of page migration

For all experiments, we used version 4.3.0 of the Linux Kernel. As the SGI test platform requires specific kernel patches and is shared among many users, we conducted these experiments on Opteron-64 only. The migration mechanism is configured through the procfs pseudo filesystem, as follows:

- Migration can be globally enabled or disabled by setting `numa_balancing` to 1 or 0, respectively.
- The parameter `numa_balancing_scan_delay_ms` indicates the minimum execution time of a process before page migration starts. In our experiments, we have set this value to 0 to enable migration as soon as possible. Page migration during initialization is prevented using appropriate calls to `mbind`, temporarily imposing static placement.
- How much of the address space is examined in one scan is defined by `numa_balancing_scan_size_mb`. In the experiments, this parameter has been set to 0 to prevent the system from scanning only a subset of the pages.

In the following evaluation, we calculate the ratio of the median wall clock execution time with dynamic migration (`numa_balancing` set to 1) divided by the median time with-
out migration (numa_balancing set to 0) for 10 runs of a synthetic benchmark.

### 6.2 Evaluation of a synthetic benchmark

The synthetic benchmark has been designed specifically to evaluate the potential of dynamic page migration for scenarios with clear relationships between data and computations and without interference. It is composed of the following steps:

1. Allocate $S$ sets of $T$ 64MiB buffers, distributed in a round-robin fashion on the machine's NUMA node. That is, the $i$-th buffer of each set is allocated on NUMA node $(i \mod N)$, with $N$ being the total number of NUMA nodes.

2. Create $T$ threads and pin the $i$-th thread on the $i$-th core.

3. Assign exactly one buffer of the current set to each thread, with thread $i$ being the owner of the $i$-th buffer.

4. Synchronize all threads using a barrier and let each thread traverse its buffer $I$ times linearly by adding a constant to the first 8-byte integer of every cache line of the buffer.

5. Change the affinity $A$ times by repeating steps 3 and 4 a total of $A$ times.

6. Synchronize all threads with a barrier and print the time elapsed between the moments in which the first thread passed the first and the last barrier, respectively.

On Opteron-64, the number of 8 cores per NUMA node is equal to the total number of nodes. Thus, the allocation scheme above causes every NUMA node to access every node of the system at the beginning of each affinity change, which
allows for load-balancing on the system’s memory controllers and thus factors out contention arising from the initial distribution. We have set the total number of affinity changes \( A \) to 8.

Figure 11 shows the speedup for a varying number of iterations \( I \) before each affinity change. We found that the preferred page size for the buffers has a strong influence on migration overhead. Therefore, we evaluate three configurations: \textit{default} does not impose any specific page size and leaves the choice to the operating system, while \textit{small} and \textit{huge} force the use of 4KiB and 2MiB pages, respectively.

In order to match the performance of the SHM baseline with static placement, at least 12 iterations are necessary for \textit{default} and \textit{huge}. For small pages, at least 70 iterations must be performed. In our benchmarks with task-private input and output buffers, the bulk of the input data of each task is accessed only up to 7 times. The temporal data locality is thus far below these thresholds, which lets us expect that dynamic page migration cannot improve performance.

### 6.3 Evaluation of OpenStream benchmarks

Let us now study the impact of page migration on the SHM parallel baseline with globally shared data structures. In contrast to task-private buffers, in which each input buffer at each iteration potentially uses a different set of addresses, each block of data of the globally shared data structures is associated to a fixed set of addresses for all iterations with very high temporal locality. As in the previous experiments, we used hierarchical work-stealing, initial interleaved allocation, and the parameters for the benchmarks described in Table 3.

Figure 12 shows the speedup of dynamic page migration over the median execution time without migration. For none of the benchmarks does page migration improve performance. In the best case (\textit{jacobi-3d}, \textit{seidel-2d}, \textit{seidel-3d}, \textit{kmeans}, and \textit{blur-roberts}) performance is almost identical with small variations. In many other cases performance degrades substantially (\textit{jacobi-1d}, \textit{jacobi-2d}, \textit{seidel-1d}, and \textit{bitonic}).

The first reason for this degradation is that dynamic page migration is not able to catch up with frequent affinity changes between cores and data. Second, in contrast to task-private buffers, data blocks of the shared data structures do not necessarily represent contiguous portions of the address space and a single huge page might contain data of more than one block, accessed by different cores. In conclusion, page migration only reacts to changes in the task-data affinity while our scheme proactively binds them together and page granularity may also not be appropriate for data placement in task-parallel applications.

### 7. RELATED WORK

Combined NUMA-aware scheduling and data placement can be split into general methods operating at the thread level—usually implemented in the operating system at page granularity, and task-oriented methods operating at the task level in parallel programming languages—typically in userland.

Starting with userland methods, it is possible to statically place arrays and computations to bring NUMA awareness to OpenMP programs [27, 2, 34, 32] or applications using TBB [26]. Such approaches are well suited to regular data structures and involve target-specific optimizations by the programmer. This is viable in some application areas, but generally not consistent with the performance portability and dynamic concurrency of task-parallel models.

ForestGOMP [5, 6] is an OpenMP run-time with a resource-aware scheduler and a NUMA-aware allocator. It introduces three concepts: grouping of OpenMP threads into bubbles, scheduling of threads and bubbles using a hierarchy of run-queues, and migrating data dynamically upon load balancing. Although the affinity between computations and data remains implicit, ForestGOMP performs best when these affinities are stable over time, and when the locality of unstable affinities can be restored through scheduling without migration. LAWS [11] brings NUMA awareness to task-private input and output buffers, the bulk of the input data of each task is accessed only up to 7 times. The temporal data locality is thus far below these thresholds, which lets us expect that dynamic page migration cannot improve performance.

Among dependent task models, one of the most popular is the StarSs project [30], whose OMPSs variant led to dependent tasks in OpenMP 4.0. This model does provide an explicit task-data association. Yet we preferred OpenStream as it facilitates the privatization of data blocks communicated across tasks, and its first-class streams expressing dependences across concurrently created tasks accelerate the creation of complex graphs on large scale NUMA systems.

Our analysis of NUMA-aware placement and scheduling and the proposed algorithms would most likely fit other models with similar properties such as CnC [8] or KAAPi [16].

In earlier work [14], we introduced the work-pushing technique for task-parallel application that we extended in this paper. Furthermore, we proposed dependence-aware allocation, a NUMA-aware memory allocation technique that examines inter-task data dependences and speculatively allocates memory for the input data of a task on the node whose cores are likely to execute its producers. However, the techniques presented in this paper achieve better locality and performance since in some cases the prediction might fail and work-stealing might lead to remote write accesses.

Alternatively, one could argue that the operating system should be in charge of providing a NUMA-oblivious abstraction for parallel programs. Operating systems typically follow a first-touch strategy by default, in which a page of physical memory is allocated on the node associated to the core that first writes to the page. A common optimization is to migrate a page dynamically to the node performing the next write access. This strategy, referred to as affinity-or migrate-on-next-touch can be transparent [17] or controlled through system calls [25]. Dashti et al. proposed Carrefour [13], a kernel-level solution whose primary goal is to avoid congestion on memory controllers and interconnects. Carrefour detects affinities between computations and pages dynamically using hardware performance counters. Based on these affinities it combines allocation of pages near the accessing node, interleaved allocation and page replication for read-only data. However, the solution is suited for long-running processes as it uses sampling hardware counters [15]...
to determine data affinities, requiring high reuse of data for statistical confidence without high overhead [24]. Similar to Carrefour, AsymSched [24]—a combined user-space-kernel solution for data and thread placement—focuses on bandwidth rather than locality, taking into account the asymmetry of interconnects on recent NUMA systems. However, despite a highly efficient migration mechanism, the placement granularity is bound to a page and may not meet the requirements of task-parallel applications.

8. CONCLUSION

We showed that parallel languages with dependent tasks can achieve excellent, scalable performance on large scale NUMA machines without exposing programmers to the complexity of these systems. One key element of the solution is to implement communications through task-private data buffers. This allows for the preservation of a simple, uniform abstract view for both memory and computations, yet achieving high data locality. Inter-task data dependences provide precise information about affinities between tasks and data in the run-time, improving the accuracy of NUMA-aware scheduling. We proposed two complementary techniques to exploit this information and to manage task-private buffers: enhanced work-pushing and deferred allocation. Deferred allocation guarantees that all accesses to task output data are local, while enhanced work-pushing improves the locality of accesses to task input data. By combining hierarchical work-stealing with enhanced work-pushing, we ensure that no processor remains idle, unless no task is ready to execute. Deferred allocation provides an additional level of load-balancing, addressing contention on memory controllers. We showed that our techniques achieve up to 5× speedup over state of the art NUMA-aware solutions, in presence of dynamic task creation and changing dependence patterns. In a comparison with transparent static and dynamic allocation techniques by the operating system, we showed that our solution is up to 5.6× faster than interleaved allocation and that dynamic page migration is unable to cope with the fine-grained concurrency and communication of task-parallel applications.

Acknowledgments

Our work was partly supported by the grants EU FET-HPC ExaNoDe I2H2012-671578 and UK EPSRC EP/M004880/1. A. Pop is funded by a Royal Academy of Engineering University Research Fellowship.

9. REFERENCES


