A Scalable Architecture for Ordered Parallelism

Mark Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, Daniel Sanchez

MICRO 2015
Multicores Target Easy Parallelism
Regular: known tasks and data
Multicores Target Easy Parallelism

Regular: known tasks and data
**Multicores Target Easy Parallelism**

Regular: known tasks and data

Irregular: unknown tasks and data
Multicores Target Easy Parallelism

- **Regular**: known tasks and data
- **Irregular**: unknown tasks and data
- Unordered tasks
Multicores Target Easy Parallelism

Regular: known tasks and data

Irregular: unknown tasks and data

Unordered tasks

≈ Load-balancing Synchronization
Multicores Target Easy Parallelism

Regular: known tasks and data
Irregular: unknown tasks and data
Ordered tasks
Unordered tasks

≈ Load-balancing Synchronization

Irregular: unknown tasks and data
Multicores Target Easy Parallelism

Regular: known tasks and data
Irregular: unknown tasks and data
Ordered tasks
Unordered tasks

Load-balancing
Synchronization
Ordering is a simple and general form of synchronization
Multicores Target Easy Parallelism

Ordering is a simple and general form of synchronization

Support for order enables widespread parallelism
Outline

- Understanding Ordered Parallelism
- Swarm
- Evaluation
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

source

A → B: 3
B → C: 2
C → E: 3
C → D: 4
D → B: 1
D → E: 3

Graph diagram with weighted edges.
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Order = Distance from source node

Tasks
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Tasks

Order = Distance from source node
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Order = Distance from source node
Parallelism in Dijkstra’s Algorithm

Can execute independent tasks out of order

Order = Distance from source node

Tasks
Parallelism in Dijkstra’s Algorithm

Can execute independent tasks out of order

# Tasks

Order = Distance from source node

Data dependences
Parallelism in Dijkstra’s Algorithm

Can execute independent tasks out of order

Tasks

Order = Distance from source node

Valid schedule

Data dependences
Parallelism in Dijkstra’s Algorithm

Can execute independent tasks out of order

Tasks

Order = Distance from source node

Valid schedule

2x parallelism (more in larger graphs)

Tasks and dependences unknown in advance
Parallelism in Dijkstra’s Algorithm

Can execute independent tasks out of order

Tasks

Order = Distance from source node

Valid schedule

2x parallelism (more in larger graphs)

Tasks and dependences unknown in advance

Need speculative execution to elide order constraints
Insights about Ordered Parallelism
1. With perfect speculation, parallelism is plentiful
1. With perfect speculation, parallelism is plentiful

Ideal schedule

A → C → B → D → E
1. With perfect speculation, parallelism is plentiful

Insights about Ordered Parallelism

Ideal schedule

Parallelism
max 800x
Insights about Ordered Parallelism

1. With perfect speculation, parallelism is plentiful

2. Tasks are tiny: 32 instructions on average
1. With perfect speculation, parallelism is plentiful
   
   Ideal schedule

2. Tasks are tiny: 32 instructions on average

3. Independent tasks are far away in program order
1. With perfect speculation, parallelism is plentiful

2. Tasks are tiny: 32 instructions on average

3. Independent tasks are far away in program order

Can execute $N$ tasks ahead of the earliest active task
1. With perfect speculation, parallelism is plentiful

2. Tasks are tiny: 32 instructions on average

3. Independent tasks are far away in program order

Can execute $N$ tasks ahead of the earliest active task

**Parallelism**

<table>
<thead>
<tr>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>800x</td>
</tr>
</tbody>
</table>
1. With perfect speculation, parallelism is plentiful

2. Tasks are tiny: 32 instructions on average

3. Independent tasks are far away in program order

Can execute $N$ tasks ahead of the earliest active task
1. With perfect speculation, parallelism is plentiful

2. Tasks are tiny: 32 instructions on average

3. Independent tasks are far away in program order

Can execute $N$ tasks ahead of the earliest active task
Insights about Ordered Parallelism

1. With perfect speculation, parallelism is plentiful
   
   Ideal schedule

2. Tasks are tiny: 32 instructions on average

3. Independent tasks are far away in program order

   Can execute $N$ tasks ahead of the earliest active task

Need a large window of speculation

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>max</th>
<th>800x</th>
</tr>
</thead>
<tbody>
<tr>
<td>window=64</td>
<td></td>
<td>26x</td>
</tr>
<tr>
<td>window=1k</td>
<td></td>
<td>180x</td>
</tr>
</tbody>
</table>
Prior Work Can’t Mine Ordered Parallelism
Prior Work Can’t Mine Ordered Parallelism

- Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs
Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs.

Max parallelism | TLS parallelism
---|---
800x | 1.1x
Prior Work Can’t Mine Ordered Parallelism

- Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs

<table>
<thead>
<tr>
<th>Max parallelism</th>
<th>TLS parallelism</th>
<th>Execution order ≠ creation order</th>
</tr>
</thead>
<tbody>
<tr>
<td>800x</td>
<td>1.1x</td>
<td></td>
</tr>
</tbody>
</table>
Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs.

<table>
<thead>
<tr>
<th>Max parallelism</th>
<th>TLS parallelism</th>
<th>Execution order ≠ creation order</th>
<th>Task-scheduling priority queues introduce false data dependences</th>
</tr>
</thead>
<tbody>
<tr>
<td>800x</td>
<td>1.1x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs.

<table>
<thead>
<tr>
<th>Max parallelism</th>
<th>TLS parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>800x</td>
<td>1.1x</td>
</tr>
</tbody>
</table>

Execution order ≠ creation order. Task-scheduling priority queues introduce false data dependences.

Sophisticated parallel algorithms yield limited speedup.
Prior Work Can’t Mine Ordered Parallelism

- Thread-Level Speculation (TLS) parallelizes loops and function calls in sequential programs.

<table>
<thead>
<tr>
<th>Max parallelism</th>
<th>TLS parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>800x</td>
<td>1.1x</td>
</tr>
</tbody>
</table>

Execution order ≠ creation order.

Task-scheduling priority queues introduce false data dependences.

- Sophisticated parallel algorithms yield limited speedup.
Swarm Mines Ordered Parallelism

- Swarm
- Software-only parallel

Graphs showing speedup for different algorithms (bfs, sssp, astar, msf, des, silo) with varying processor counts (1c, 32c, 64c). The Swarm curve consistently outperforms the Software-only parallel curve, with the highest speedup of 117x for sssp.
Swarm Mines Ordered Parallelism

The diagram shows the speedup of different algorithms under Swarm and Software-only parallelism. The x-axis represents the number of cores (1c, 32c, 64c), and the y-axis represents speedup. The Swarm algorithm consistently outperforms the Software-only parallelism, with notable speedup gains in bfs, sssp, astar, msf, des, and silo. For example, bfs shows a speedup of 117x under Swarm compared to Software-only parallelism.
Execution model based on timestamped tasks
Swarm Mines Ordered Parallelism

- Execution model based on timestamped tasks
- Architecture executes tasks speculatively out of order
  - Leverages execution model to scale
Outline

- Understanding Ordered Parallelism
- Swarm
- Evaluation
Swarm Execution Model

Programs consist of timestamped tasks
Swarm Execution Model

Programs consist of timestamped tasks

- Tasks can create children tasks with $>= \text{timestamp}$
- Tasks appear to execute in timestamp order
Swarm Execution Model

Programs consist of timestamped tasks

- Tasks can create children tasks with $\geq$ timestamp
- Tasks appear to execute in timestamp order
- Programmed with implicitly-parallel task API

```cpp
swarm::enqueue(fptr, ts, args...);
```
Swarm Execution Model

Programs consist of timestamped tasks
- Tasks can create children tasks with $\geq$ timestamp
- Tasks appear to execute in timestamp order
- Programmed with implicitly-parallel task API

\texttt{swarm::enqueue(fptr, ts, args...);}

Conveys new work to hardware as soon as possible
Swarm Execution Model

Programs consist of timestamped tasks
- Tasks can create children tasks with \( \geq \) timestamp
- Tasks appear to execute in timestamp order
- Programmed with implicitly-parallel task API

```cpp
class swarm::enqueue(fptr, ts, args...);```

Conveys new work to hardware as soon as possible
void ssspTask(Timestamp dist, Vertex& v) {
    if (!v.isVisited()) {
        v.distance = dist;
        for (Vertex& u : v.neighbors) {
            Timestamp uDist = dist + edgeWeight(v, u);
            swarm::enqueue(&ssspTask, uDist, u);
        }
    }
}
void ssspTask(Timestamp dist, Vertex& v) {
  if (!v.isVisited()) {
    v.distance = dist;
    for (Vertex& u : v.neighbors) {
      Timestamp uDist = dist + edgeWeight(v, u);
      swarm::enqueue(&ssspTask, uDist, u);
    }
  }
}
void ssspTask(Timestamp dist, Vertex& v) {
    if (!v.isVisited()) {
        v.distance = dist;
        for (Vertex& u : v.neighbors) {
            Timestamp uDist = dist + edgeWeight(v, u);
            swarm::enqueue(&ssspTask, uDist, u);
        }
    }
}
Swarm Task Example: Dijkstra

```cpp
void ssspTask(Timestamp dist, Vertex& v) {
  if (!v.isVisited()) {
    v.distance = dist;
    for (Vertex& u : v.neighbors) {
      Timestamp uDist = dist + edgeWeight(v, u);
      swarm::enqueue(&ssspTask, uDist, u);
    }
  }
}

swarm::enqueue(ssspTask, 0, sourceVertex);
swarm::run();
```
Swarm Architecture Overview

![Diagram of Swarm Architecture Overview]

- Tiled Multicore
- Memory controller
- Tile Organization
  - L3 Cache Bank
  - Router
  - L2 Cache
  - L1I/D
  - L1I/D
  - L1I/D
  - L1I/D
  - Core
  - Core
  - Core
  - Core

Task Unit
Swarm Architecture Overview

Per-tile task units:

- **Task Queue**: holds task descriptors
- **Commit Queue**: holds speculative state of finished tasks
Per-tile task units:

- **Task Queue**: holds task descriptors
- **Commit Queue**: holds speculative state of finished tasks

Commit queues provide the window of speculation
**Task Unit Queues**

- **Task queue:** holds task descriptors
- **Commit Queue:** holds speculative state of finished tasks

<table>
<thead>
<tr>
<th>Task Queue</th>
<th>Cores</th>
<th>Commit Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>9, I</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10, I</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2, R</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>8, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3, F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Task States:** IDLE (I)  RUNNING (R)  FINISHED (F)
**Task Unit Queues**

- **Task queue**: holds task descriptors
- **Commit Queue**: holds speculative state of finished tasks

<table>
<thead>
<tr>
<th>Task Queue</th>
<th>Cores</th>
<th>Commit Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>7, I</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>9, I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10, I</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3, F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

New Task (timestamp=7, taskFn, args)
Task Unit Queues

- **Task queue**: holds task descriptors
- **Commit Queue**: holds speculative state of finished tasks

<table>
<thead>
<tr>
<th>Task Queue</th>
<th>Cores</th>
<th>Commit Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>7, R</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9, I</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>10, I</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2, F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3, F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Task States: IDLE (I) RUNNING (R) FINISHED (F)*
Task Unit Queues

- **Task queue**: holds task descriptors
- **Commit Queue**: holds speculative state of finished tasks

Task States: IDLE (I) RUNNING (R) FINISHED (F)

<table>
<thead>
<tr>
<th>Task Queue</th>
<th>Cores</th>
<th>Commit Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>7, F</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>9, I</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10, I</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2, F</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>8, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3, F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Task Unit Queues

- **Task queue**: holds task descriptors
- **Commit Queue**: holds speculative state of finished tasks

Task States: IDLE (I)  RUNNING (R)  FINISHED (F)

<table>
<thead>
<tr>
<th>Task Queue</th>
<th>Cores</th>
<th>Commit Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>7, F</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>9, R</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10, I</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>2, F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3, F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Task Unit Queues

- **Task queue**: holds task descriptors
- **Commit Queue**: holds speculative state of finished tasks

<table>
<thead>
<tr>
<th>Task States: IDLE (I)</th>
<th>RUNNING (R)</th>
<th>FINISHED (F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Queue</td>
<td>Cores</td>
<td>Commit Queue</td>
</tr>
<tr>
<td>7, F</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>9, R</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>10, I</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>2, F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3, F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Similar to a reorder buffer, but at the task level
Suppose 64-cycle tasks execute on 64 cores

- 1 task commit/cycle to scale
- TLS commit schemes (successor lists, commit token) too slow
Suppose 64-cycle tasks execute on 64 cores
- 1 task commit/cycle to scale
- TLS commit schemes (successor lists, commit token) too slow

We adapt “Virtual Time” [Jefferson, TOPLAS 1985]

- Tile 1
- Tile 2
- ... Tile N
- GVT Arbiter
High-Throughput Ordered Commits

- Suppose 64-cycle tasks execute on 64 cores
  - 1 task commit/cycle to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt “Virtual Time” [Jefferson, TOPLAS 1985]

Tiles periodically communicate to find the earliest unfinished task.

Tile 1

Tile 2

…

Tile N

GVT Arbiter
Suppose 64-cycle tasks execute on 64 cores

1 task commit/cycle to scale

TLS commit schemes (successor lists, commit token) too slow

We adapt “Virtual Time” [Jefferson, TOPLAS 1985]

Tiles periodically communicate to find the earliest unfinished task
Suppose 64-cycle tasks execute on 64 cores
- 1 task commit/cycle to scale
- TLS commit schemes (successor lists, commit token) too slow

We adapt “Virtual Time” [Jefferson, TOPLAS 1985]

Tiles periodically communicate to find the earliest unfinished task
Tiles commit all tasks that precede it
High-Throughput Ordered Commits

- Suppose 64-cycle tasks execute on 64 cores
  - 1 task commit/cycle to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt “Virtual Time” [Jefferson, TOPLAS 1985]

Tile 1 → Tile 2 → ... → Tile N

- Tiles periodically communicate to find the earliest unfinished task
- Tiles commit all tasks that precede it

With large commit queues, many tasks commit at once
High-Throughput Ordered Commits

- Suppose 64-cycle tasks execute on 64 cores
  - 1 task commit/cycle to scale
  - TLS commit schemes (successor lists, commit token) too slow
- We adapt “Virtual Time” [Jefferson, TOPLAS 1985]

Tiles periodically communicate to find the earliest unfinished task.
Tiles commit all tasks that precede it.

With large commit queues, many tasks commit at once.
Amortizes commit costs among many tasks.
Speculative Execution Example

- Core 0
- Core 1
- Core 2

Time

Timestamp order
Speculative Execution Example

- Core 0: 0 → 1
- Core 1: 3
- Core 2: 3

Timestamp order:
Core 0: 0 → 1
Core 1: 3
Core 2: 3

Time line:
Tasks can execute even if parent is still speculative

Uncovers more parallelism
Speculative Execution Example

- Tasks can execute even if parent is still speculative
- Uncovers more parallelism
Speculative Execution Example

- Tasks can execute even if parent is still speculative
- Uncovers more parallelism
Speculative Execution Example

- Tasks can execute even if parent is still speculative
- Uncovers more parallelism
Speculative Execution Example

- Tasks can execute even if parent is still speculative
  - Uncovers more parallelism
  - May trigger cascading (but selective) aborts
Tasks can execute even if parent is still speculative
- Uncovers more parallelism
- May trigger cascading (but selective) aborts
Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window → Small per-task speculative state
Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window → Small per-task speculative state

- Eager versioning + timestamp-based conflict detection
  - Bloom filters for cheap read/write sets [Yen, HPCA 2007]
Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window $\rightarrow$ Small per-task speculative state

- Eager versioning + timestamp-based conflict detection
  - Bloom filters for cheap read/write sets [Yen, HPCA 2007]
  - Uses hierarchical memory system to filter conflict checks
Swarm Speculation Mechanisms

- Key requirements for speculative execution:
  - Fast commits
  - Large speculative window ➔ Small per-task speculative state

- Eager versioning + timestamp-based conflict detection
  - Bloom filters for cheap read/write sets [Yen, HPCA 2007]
  - Uses hierarchical memory system to filter conflict checks

- Enables two helpful properties
  1. **Forwarding** of still-speculative data
  2. On rollback, corrective writes abort dependent tasks only
Outline

- Understanding Ordered Parallelism
- Swarm
- Evaluation
Evaluation Methodology

- Event-driven, sequential, Pin-based simulator
- Target system: 64-core, 16-tile chip

16 MB shared L3 (1MB/tile)
256 KB per-tile L2s
32 KB per-core L1s
4096 task queue entries (64/core)
1024 commit queue entries (16/core)
256-byte, 8-way Bloom filters
Evaluation Methodology

- Event-driven, sequential, Pin-based simulator

- Target system: 64-core, 16-tile chip

![Diagram of the target system]

- Memory controller
- Tile
- Core
- L1/D
- L2 Cache
- L3 Cache Bank
- Router
- Task Unit

- 16 MB shared L3 (1 MB/tile)
- 256 KB per-tile L2s
- 32 KB per-core L1s
- 4096 task queue entries (64/core)
- 1024 commit queue entries (16/core)
- 256-byte, 8-way Bloom filters
Evaluation Methodology

- Event-driven, sequential, Pin-based simulator

- Target system: 64-core, 16-tile chip

- Scalability experiments from 1-64 cores
  - Scaled-down systems have fewer tiles
Swarm vs. Software Versions

![Graph showing speedup comparisons between Swarm and Software-only parallel versions for different benchmarks: bfs, sssp, astar, msf, des, and silo. The graphs display speedup values for 1c, 32c, and 64c configurations.]
Swarm vs. Software Versions

43x – 117x faster than serial versions
Swarm vs. Software Versions

43x – 117x faster than serial versions
3x – 18x faster than parallel versions
**Swarm vs. Software Versions**

- **43x – 117x faster than serial versions**
- **3x – 18x faster than parallel versions**

*Simple implicitly-parallel code*
Swarm Uses Resources Efficiently

The diagram shows the core cycles (%) for different algorithms: bfs, sssp, astar, msf, des, and silo. The y-axis represents the core cycles in percentage, ranging from 0 to 100. The categories are Commit, Abort, Queue, and Stall. The diagram visualizes how each algorithm utilizes resources efficiently by distributing these categories across the core cycles.
Swarm Uses Resources Efficiently

Most time spent executing tasks that commit
**Swarm Uses Resources Efficiently**

Most time spent executing tasks that commit

Swarm speculates 200-800 tasks ahead on average.
Swarm Uses Resources Efficiently

- Most time spent executing tasks that commit
- Speculation adds moderate energy overheads:
  - 15% extra network traffic
  - Conflict check logic triggered in 9-16% of cycles

Swarm speculates 200-800 tasks ahead on average
Conclusions

- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general
Conclusions

- Swarm exploits ordered parallelism efficiently
  - Necessary to parallelize many key algorithms
  - Simplifies parallel programming in general

- Conventional wisdom: Ordering limits parallelism
Conclusions

- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general

- Conventional wisdom: Ordering limits parallelism
  Expressive execution model + large window =
  Only true data dependences limit parallelism
Conclusions

- Swarm exploits ordered parallelism efficiently
  - **Necessary** to parallelize many key algorithms
  - **Simplifies** parallel programming in general

- Conventional wisdom: Ordering limits parallelism
  Expressive execution model + large window =
  Only true data dependences limit parallelism

- Conventional wisdom: Speculation is wasteful
Conclusions

- Swarm exploits ordered parallelism efficiently
  - Necessary to parallelize many key algorithms
  - Simplifies parallel programming in general

- Conventional wisdom: Ordering limits parallelism
  Only true data dependences limit parallelism

- Conventional wisdom: Speculation is wasteful
  Speculation unlocks plentiful ordered parallelism
  Can trade parallelism for efficiency (e.g., simpler cores)
Thanks for your attention!
Questions?

A Scalable Architecture for Ordered Parallelism
Mark Jeffrey, Suvinay Subramanian, Cong Yan,
Joel Emer, Daniel Sanchez