A Scalable Architecture for Ordered Parallelism

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

MICRO 2015
Multicores Target Easy Parallelism
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
Regular: known tasks and data
Ordered tasks
Unordered tasks
≈ Load-balancing
Synchronization
Multicores Target Easy Parallelism

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 → 2 → C → 2 → B → 1 → D → 3 → E → 3 → source

B → 1 → D

C → 2

D → 3

E → 3
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

Order = Distance from source node

Tasks

A
Example: Parallelism in Dijkstra’s Algorithm

Finds shortest-path tree on a graph with weighted edges

![Graph Example]

**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

Task 1: Distance from source node

AO: Distance from source node

Source: A

Tasks

AO: Distance from source node

AO: 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

Source

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

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

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
Parallelism in Dijkstra’s Algorithm

Can execute independent tasks out of order

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

Can execute independent tasks out of order

Tasks

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

Can execute independent tasks out of order

Data dependences

Order = Distance from source node

Valid schedule
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

Data dependences

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
Insights about Ordered Parallelism

1. With perfect speculation, parallelism is plentiful
1. With perfect speculation, parallelism is plentiful

Ideal schedule
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

   Ideal schedule

   A → C → B → D → E

   Parallelism

   max 800x

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

2. Tasks are tiny: 32 instructions on average

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

Ideal schedule

Parallelism

max 800x

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
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**

   ![Diagram of ideal schedule]

   **Parallelism**

<table>
<thead>
<tr>
<th></th>
<th>max</th>
<th>800x</th>
</tr>
</thead>
<tbody>
<tr>
<td>window=64</td>
<td>26x</td>
<td></td>
</tr>
<tr>
<td>window=1k</td>
<td>180x</td>
<td></td>
</tr>
</tbody>
</table>

2. **Tasks are tiny:** 32 instructions on average

3. **Independent tasks are far away in program order**

   ![Diagram of N-task window]

   Can execute \( N \) tasks ahead of the earliest active task
Insights about Ordered Parallelism

1. With perfect speculation, parallelism is plentiful
   - Ideal schedule
   - Parallelism
     - max 800x
     - window=64 26x
     - window=1k 180x

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
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
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>
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>
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
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.
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**

![Graph showing speedup with Swarm and Software-only parallel for bfs, sssp, astar, msf, des, and silo algorithms.](image)
Swarm Mines Ordered Parallelism

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>1c</th>
<th>32c</th>
<th>64c</th>
</tr>
</thead>
<tbody>
<tr>
<td>bfs</td>
<td>1</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>sssp</td>
<td>1c</td>
<td>32c</td>
<td>64c</td>
</tr>
<tr>
<td>astar</td>
<td>1c</td>
<td>32c</td>
<td>64c</td>
</tr>
<tr>
<td>msf</td>
<td>1c</td>
<td>32c</td>
<td>64c</td>
</tr>
<tr>
<td>des</td>
<td>1c</td>
<td>32c</td>
<td>64c</td>
</tr>
<tr>
<td>silo</td>
<td>1c</td>
<td>32c</td>
<td>64c</td>
</tr>
</tbody>
</table>

**Speedup**: Swarm vs. Software-only parallel

- bfs: 117x
- sssp: 117x
- astar: 117x
- msf: 117x
- des: 117x
- silo: 117x
**Swarm Mines Ordered Parallelism**

- Execution model based on timestamped tasks
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 $\geq$ 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

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

\[ \text{swarm::enqueue(fptr, ts, args...);} \]

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

Tiled Multicore

Memory controller

Tile

L1I/D

L2 Cache

L1I/D

L1I/D

L1I/D

L1I/D

L2 Cache Bank

Router

L3 Cache Bank

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
Swarm Architecture Overview

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 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></td>
</tr>
<tr>
<td>8, R</td>
<td>8</td>
<td>3</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)

**New Task**
(timestamp=7, taskFn, args)

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

<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>

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]
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
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
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]

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
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
  - Timestamp order:
    - 0

- Core 1

- Core 2

Time

Timestamp order
Speculative Execution Example

0 → 1 → 3

Core 0: 0 → 1
Core 1: 3
Core 2

Timestamp order

Time
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
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
Speculative Execution Example

- 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
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 \(\rightarrow\) 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

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

![Diagram of the target system]

- 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 a chip with Memory controller, Router, L3 Cache Bank, L2 Cache, Core, Task Unit, 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

- Scalability experiments from 1-64 cores
  - Scaled-down systems have fewer tiles

- Memory controller
  - L3 Cache Bank: 16 MB shared L3 (1 MB/tile)
  - L2 Cache: 256 KB per-tile L2s
  - L1/D: 32 KB per-core L1s
  - 4096 task queue entries (64/core)
  - 1024 commit queue entries (16/core)
  - 256-byte, 8-way Bloom filters
Swarm vs. Software Versions

The diagram shows the speedup for different algorithms (bfs, sssp, astar, msf, des, silo) under Swarm and Software-only parallel conditions. The x-axis represents the number of cores (1c, 32c, 64c), and the y-axis represents the speedup. The Swarm results are represented by green lines, while the Software-only parallel results are represented by red dashed lines.

For example, in the bfs algorithm, the speedup for 64c cores is significantly higher under Swarm compared to Software-only parallel, indicating a potential advantage of Swarm in parallel computing.

The graph also highlights a 117x speedup for the sssp algorithm on 64c cores under Swarm, compared to Software-only parallel.
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

- Commit
- Abort
- Queue
- Stall

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>bfs</th>
<th>sssp</th>
<th>astar</th>
<th>msf</th>
<th>des</th>
<th>silo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Cycles (%)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Diagram shows the percentage distribution of core cycles across different algorithms and operations.
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

- Speculation adds moderate energy overheads:
  - 15% extra network traffic
  - Conflict check logic triggered in 9-16% of cycles

Most time spent executing tasks that commit

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
  Expressive execution model + large window =
  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