A Scalable Architecture for Reprioritizing Ordered Parallelism

Gilead Posluns, Yan Zhu, Guowei Zhang, Mark C. Jeffrey

ISCA 2022
Ordered algorithms use priority schedules
Ordered algorithms use priority schedules

```java
pq = init();
while (!pq.empty())
    task, ts = pq.dequeueMin()
    task(ts)
```
Ordered algorithms use priority schedules

\[
pq = \text{init}();
\]
\[
\textbf{while} \ (\neg \ pq.\text{empty}())
\]
\[
\text{task, } ts = pq.\text{dequeueMin}()
\]
\[
\text{task}(ts)
\]

Priority schedules accelerate convergence

Dijkstra’s SSSP

Breadth First Search

Residual Belief Propagation
Ordered algorithms use priority schedules

pq = init();
while (!pq.empty())
    task, ts = pq.dequeueMin()
    task(ts)

Priority schedules accelerate convergence

Dijkstra’s SSSP  Residual Belief Propagation  Breadth First Search

Priority schedules are correct

KCore  Minimum Spanning Forest  Maximal Independent Set
Set Cover
Ordered algorithms use priority schedules

\[
pq = \text{init}(); \\
\textbf{while} (\neg pq.\text{empty}()) \\
\quad \text{task, ts} = pq.\text{dequeueMin}() \\
\quad \text{task}(ts)
\]

Priority schedules accelerate convergence

Priority schedules are powerful, but hard to parallelize

Priority schedules are correct

- KCore
- Minimum Spanning Forest
- Set Cover
- Maximal Independent Set
Hive parallelizes priority updates

Hive builds on Swarm to provide a parallel priority update operation in speculative task-parallel hardware

Hive speculates eagerly on data, control, and scheduler dependences

Hive achieves >100x speedup over parallel software, and up to 2.8x over Swarm at 256 cores
Understanding Priority Updates
KCore requires priority updates

Max core of a vertex ≈ “importance”  [Malliaros et al. VLDB ’20]
To find: repeatedly remove lowest degree vertex

PriorityQueue pq;
for (int v: G.V)
pq.enqueue(v, G.degree[v])
while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        pq.decrementPrio(nbr)
}
KCore requires priority updates

Max core of a vertex ≈ “importance”  [Malliaros et al. VLDB ‘20]

To find: repeatedly remove lowest degree vertex

```java
PriorityQueue pq;
for (int v: G.V)
    pq.enqueue(v, G.degree[v])
while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        pq.decrementPrio(nbr)
}
```

Input Graph

Task Graph

Dependence

Task

Priority = Remaining Degree

PriorityQueue pq;
for (int v: G.V)
pq.enqueue(v, G.degree[v])
while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        pq.decrementPrio(nbr)
}

To find: repeatedly remove lowest degree vertex

Max core of a vertex ≈ “importance”  [Malliaros et al. VLDB ‘20]
**KCore requires priority updates**

Max core of a vertex ≈ “importance” \cite{Malliaros et al. VLDB ’20}

To find: repeatedly remove lowest degree vertex

```java
PriorityQueue pq;
for (int v : G.V) pq.enqueue(v, G.degree[v])
while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        pq.decrementPrio(nbr)
}
```

**Input Graph**

**Task Graph**

**Task Dependence**

**Priority = Remaining Degree**

---

1

2

3

---

**PriorityQueue pq;**

**for (int v: G.V) pq.enqueue(v, G.degree[v])**

**while (!pq.empty()) {**

**int v, int prio = pq.dequeueMin();**

**coreness[v] = prio;**

**for (int nbr : G.edges[v]) pq.decrementPrio(nbr)**

**}**
KCore requires priority updates

Max core of a vertex ≈ “importance”  [Malliaros et al. VLDB ’20]

To find: repeatedly remove lowest degree vertex

```
PriorityQueue pq;
for (int v: G.V) 
pq.enqueue(v, G.degree[v])
```

```
while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        pq.decrementPrio(nbr)
}
```
KCore requires priority updates

Max core of a vertex ≈ “importance” \[\text{[Malliaros et al. VLDB ’20]}\]

To find: repeatedly remove lowest degree vertex

PriorityQueue pq;
for (int v: G.V)
pq.enqueue(v, G.degree[v])

while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    pq.decrementPrio(nbr)
}

Input
Graph

Task Graph
Dependence

Task

Priority = Remaining Degree

1

2

3

KCore requires priority updates

Max core of a vertex ≈ “importance” \[\text{[Malliaros et al. VLDB ’20]}\]

To find: repeatedly remove lowest degree vertex
Where’s the parallelism in KCore?
Where’s the parallelism in KCore?

• **Bulk-Synchronous** [Dhulipala et al. SPAA’17] [Dadu et al. ISCA’21]
  
  • Effective when many tasks per barrier
  • Nearly sequential when few tasks per barrier
Where’s the parallelism in KCore?

- **Bulk-Synchronous** [Dhulipala et al. SPAA’17] [Dadu et al. ISCA’21]
  - Effective when many tasks per barrier
  - Nearly sequential when few tasks per barrier

- **Relaxed** [Khan et al HPCA’22] [Yesil et al. SC’19] [Dadu et al. ISCA’21]
  - Can always find parallelism
  - Loses efficiency as it scales
  - Not always correct
Where’s the parallelism in KCore?

- **Bulk-Synchronous** [Dhulipala et al. SPAA’17] [Dadu et al. ISCA’21]
  - Effective when many tasks per barrier
  - Nearly sequential when few tasks per barrier

- **Relaxed** [Khan et al HPCA’22] [Yesil et al. SC’19] [Dadu et al. ISCA’21]
  - Can always find parallelism
  - Loses efficiency as it scales
  - Not always correct
Where’s the parallelism in KCore?

• **Bulk-Synchronous** [Dhulipala et al. SPAA’17] [Dadu et al. ISCA’21]
  • Effective when many tasks per barrier
  • Nearly sequential when few tasks per barrier

• **Relaxed** [Khan et al. HPCA’22] [Yesil et al. SC’19] [Dadu et al. ISCA’21]
  • Can always find parallelism
  • Loses efficiency as it scales
  • Not always correct

• **Speculation** [Blelloch et al. PPoPP’12] [Jeffrey et al. MICRO’15]
  • Always finds parallelism
  • Maintains strict ordering
  • SW speculation has high overheads
  • Existing HW systems do not support priority updates
Where’s the parallelism in KCore?

- **Bulk-Synchronous** [Dhulipala et al. SPAA’17] [Dadu et al. ISCA’21]
  - Effective when many tasks per barrier
  - Nearly sequential when few tasks per barrier

- **Relaxed** [Khan et al. HPCA’22] [Yesil et al. SC’19] [Dadu et al. ISCA’21]
  - Can always find parallelism
  - Loses efficiency as it scales
  - Not always correct

- **Speculation** [Blelloch et al. PPoPP’12] [Jeffrey et al. MICRO’15]
  - Always finds parallelism
  - Maintains strict ordering
  - SW speculation has high overheads
  - Existing HW systems do not support priority updates

Our goal is to support priority updates in speculative parallel hardware.
Swarm [Jeffrey et al. MICRO’15] speculates without updates

Task-Based Execution Model
Swarm [Jeffrey et al. MICRO’15] speculates without updates

Task-Based Execution Model

• Programs consist of timestamp-ordered tasks
• Tasks appear to execute in timestamp order
Swarm [Jeffrey et al. MICRO’15] speculates without updates

Task-Based Execution Model

• Programs consist of timestamp-ordered tasks

• Tasks appear to execute in timestamp order

while (!pq.empty())
    task, ts = pq.dequeueMin()
    task(ts)
Swarm [Jeffrey et al. MICRO’15] speculates without updates

Task-Based Execution Model

• Programs consist of timestamp-ordered tasks
• Tasks appear to execute in timestamp order
• Scheduler is **only** accessed with enqueues

```cpp
while (!pq.empty())
    task, ts = pq.dequeueMin()
    task(ts)

swarm::enqueue(
    fn, //what to do
    ts, //when to do it
    args //what to do it with);
```
Swarm [Jeffrey et al. MICRO’15] speculates without updates

Task-Based Execution Model

- Programs consist of timestamp-ordered tasks
- Scheduler is only accessed with enqueues

```cpp
swarm::enqueue(
    fn,  // what to do
    ts,  // when to do it
    args // what to do it with);
```

while (!pq.empty())
    task, ts = pq.dequeueMin()
    task(ts)

Swarm’s execution model does not support priority updates
Swarm KCore is inefficient (i.e., without updates)

```java
PriorityQueue pq;

for (int v : G.V) {
    pq.enqueue(v, prios[v]);
}

while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();

    coreness[v] = prio;
    for (int nbr : G.edges[v])
        if (prios[nbr] > prio) {
            pq.enqueue(nbr, prios[nbr])
        }
}
Swarm KCore is inefficient (i.e., without updates)

```java
PriorityQueue pq;
int[] prios;
for (int v : G.V) {
    prios[v] = G.degree[v];
    pq.enqueue(v, prios[v]);
}
while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        if (prios[nbr] > prio) {
            prios[nbr]--;
            pq.enqueue(nbr, prios[nbr])
        }
}
```

Manual priority tracking
Swarm KCore is inefficient (i.e., without updates)

```java
PriorityQueue pq;
int[] prios;
for (int v : G.V) {
prios[v] = G.degree[v];
pq.enqueue(v, prios[v]);
}
while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    if (prios[v] < prio) continue;
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        if (prios[nbr] > prio) {
            prios[nbr]--;
pq.enqueue(nbr, prios[nbr])
        }
}
```

Manual priority tracking

Early exit for moot tasks
Swarm KCore is inefficient (i.e., without updates)

PriorityQueue pq;
int[] prios;
for (int v: G.V) {
    prios[v] = G.degree[v];
    pq.enqueue(v, prios[v]);
}

while (!pq.empty()) {
    int v, int prio = pq.dequeueMin();
    if (prios[v] < prio) continue;
    coreness[v] = prio;
    for (int nbr : G.edges[v])
        if (prios[nbr] > prio) {
            prios[nbr]--;
            pq.enqueue(nbr, prios[nbr])
        }
}

Tasks that exit early are **moot**: they might as well not run at all

Manual priority tracking

Early exit for **moot** tasks
Updateable schedules are efficient

Input graph

A → C → D → E → F

Task

Dependence
Updateable schedules are efficient

Input graph

<table>
<thead>
<tr>
<th>Task</th>
<th>Dependence</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>D, F</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
</tbody>
</table>

Updateable Task Graph

Priority = Remaining Degree
Updateable schedules are efficient

Swarm Task Graph

Dependence
Task

Updateable Task Graph

Input graph

Priority = Remaining Degree

1 2 3

1 2 3
Updateable schedules are efficient

Swarm Task Graph

Updateable Task Graph

Dependence

Task

Input graph

Priority = Remaining Degree

1

2

3

E

A

C

F

B

D

E

A

C

D

F

B

E

A

C

D

F

B

Priority = Remaining Degree

1

2

3
Updateable schedules are efficient

Swarm Task Graph

Dependence Task

Updateable Task Graph

“Updates” enqueue a new Task

Updates change priority of a task

Input graph

Priority = Remaining Degree

Priority = Remaining Degree
Updateable schedules are efficient

Swarm Task Graph

- Dependence Task

Priority = Remaining Degree

Input graph

- “Updates” enqueue a new Task

Updateable Task Graph

- Updates change priority of a task

Priority = Remaining Degree
Updateable schedules are efficient

Swarm Task Graph

Dependence Task

Updateable Task Graph

“Updates” enqueue a new Task

Priority = Remaining Degree

Input graph

Priority = Remaining Degree

Updates change priority of a task
Updateable schedules are efficient

Enqueue-only schedule has 3 more tasks than updateable schedule

Swarm Task Graph

Updateable Task Graph

“Updates”

Priority = Remaining Degree

Swarm runs Moot tasks, but they might as well not run at all

Input graph
Moot tasks outnumber useful dequeues

KCore
Set Cover
BFS
SSSP
MSF
MIS
RBP
Moot tasks outnumber useful dequeues

Most tasks are **moot** (useless work in Swarm)
The Hive Execution Model
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev-1);
    }
}
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev-1);
    }
}
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev-1);
    }
}
Understanding Hive tasks and objects

```cpp
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev - 1);
    }
}
```
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev-1);
    }
}
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev - 1);
    }
}

Update binds a task to an object and schedules it to run
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev-1);
    }
}
Updating an occupied Hive object

```c
void removeV(int E, Timestamp ts) {
    coreness[E] = ts;
    for (int D : G.edges[E]) {
        Timestamp prev = hive::getTS(D);
        if (prev > ts)
            hive::update(&removeV, D, prev-1);
    }
}
```

Object Table

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>E</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>
Updating an occupied Hive object

```cpp
void removeV(int E, Timestamp ts) {
    coreness[E] = ts;
    for (int D : G.edges[E]) {
        Timestamp prev = hive::getTS(D);
        if (prev > ts)
            hive::update(&removeV, D, prev-1);
    }
}
```

Object Table

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Moot
void removeV(int v, Timestamp ts) {
    coreness[v] = ts;
    for (int nbr : G.edges[v]) {
        Timestamp prev = hive::getTS(nbr);
        if (prev > ts)
            hive::update(&removeV, nbr, prev - 1);
    }
}

Hive doesn’t waste time or space on moot tasks
Hive supports many programming patterns

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Increment</th>
<th>UpdateMin</th>
<th>Cancel</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>KCore</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Cover</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Astar</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth First Search</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum Spanning Forest</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximal Independent Set</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximal Matching</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Residual Belief Propagation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hive supports many programming patterns

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Increment</th>
<th>UpdateMin</th>
<th>Cancel</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>KCore</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Cover</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Astar</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth First Search</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum Spanning Forest</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Maximal Independent Set</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximal Matching</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Residual Belief Propagation</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Hive supports many programming patterns

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Increment</th>
<th>UpdateMin</th>
<th>Cancel</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>KCore</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set Cover</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Astar</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breadth First Search</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSSP</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Minimum Spanning Forest</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Maximal Independent Set</td>
<td>No Priority Queue in Sequential Implementation</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Maximal Matching</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Residual Belief Propagation</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Parallelizing Priority Updates
Hive speculates to run tasks in parallel

For each task, Hive speculates that:

• Eager data speculation: Predecessors have already performed their writes

• Eager control speculation: Its parent will not abort

• Eager scheduler speculation: It will not be replaced by an update
Hive speculates to run tasks in parallel

For each task, Hive speculates that:

• Eager data speculation: Predecessors have already performed their writes

• Eager control speculation: Its parent will not abort

• Eager scheduler speculation: It will not be replaced by an update

The same as Swarm [Jeffrey et al. MICRO’15]
Priority updates are scheduler dependences

• The scheduler dependence is old
  • Found in self-modifying code [Wilkes and Renwick. ‘49]

• Created by priority updates
  • When a task replaces a later-scheduled task, it creates a scheduler dependence

• Can be predicated into data and control dependences
  • Moot tasks are like predicated instructions in straight-line code

```
STR R5, [PC, #4]
ADD R1, R1, R1
```
Priority updates are scheduler dependences

- The scheduler dependence is old
  - Found in self-modifying code [Wilkes and Renwick. ‘49]
- Created by priority updates

Updates have a different dependence, they need different speculation

- Can be predicated into data and control dependences
  - Moot tasks are like predicated instructions in straight-line code

```
STR R5, [PC, #4]
ADD R1, R1, R1
```
Scheduler speculation:
Task versioning and Mootness detection

• Maintain multiple versions of each task
  • 1 for each speculative update + up to 1 non-speculative
• 1 task version is speculatively valid, all others are speculatively Moot
  • Speculatively Moot task versions are not runnable
• When Mootness becomes non-speculative, discard the Moot version

• Mootness can detected by comparing timestamps of parents
Scheduler speculation: Task versioning and Mootness detection

- Maintain multiple versions of each task
  - 1 for each speculative update + up to 1 non-speculative
  - 1 task version is speculatively valid, all others are speculatively Moot

Hive avoids running moot tasks and reduces their speculative state
- When Mootness becomes non-speculative, discard the Moot version

- Mootness can be detected by comparing timestamps of parents
Hive extends the Swarm architecture

64-tile, 256-core chip

- Swarm hardware additions
- Hive hardware additions
Hive extends the Swarm architecture

- 64-tile, 256-core chip
- Tile organization:
  - Router
  - GVT Arb. Node
  - L3 & Dir Bank
  - L2
  - L1/D
  - Core
  - Task unit

Swarm hardware additions

Hive hardware additions
Hive extends the Swarm architecture
Hive extends the Swarm architecture
Hive extends the Swarm architecture
Hive extends the Swarm architecture

64-tile, 256-core chip

Tile organization
- Router
- GVT Arb. Node
- L1I/D
- L1I/D
- L1I/D
- L1I/D
- Core
- Core
- Core
- Core
- L2
- L3 & Dir Bank

Task unit structures
- Task unit
- Task send buffer
- Task queue
- Commit queue
- Object map

Object Table

Swarm hardware additions
Hive hardware additions

Memory
Hive extends the Swarm architecture

64-tile, 256-core chip

Tile organization
- Router
- GVT Arb. Node
- L3 & Dir Bank
- L2
- L1/D
- Core
- Task unit

Task unit structures
- Task send buffer
- Commit queue
- Task queue
- Object map

Object Table

Swarm hardware additions
Hive hardware additions

9% Task Unit Area Increase
3% Area of a Nehalem Processor

Memory

+20B
Evaluation
Methodology

Event-driven, Pin-based Simulator\textsuperscript{1}

Scalability experiments up to 256 cores
• Smaller systems have fewer tiles

64 Tiles, 256 Cores

32kB L1 per core
1MB L2 per tile
256MB LLC
4 In-order, single-issue scoreboarded cores/tile
64 Task Queue entries/core
16 Commit Queue entries/core

9 applications: KCore, Setcover, astar, BFS, SSSP, MSF, MIS, MM, RBP

1: https://github.com/SwarmArch/sim
Software struggles to scale beyond 100c

**Diagram Description:**

- The diagram illustrates the speedup of various software systems as the system size increases.
  - The x-axis represents the system size, ranging from 1 to 256.
  - The y-axis shows the speedup, ranging from 1 to 256.
  - The graphs show the performance of different software categories:
    - kcore
    - setcover
    - astar
    - bfs
    - sssp
    - msf
    - mis
    - mm
    - rbp

**Legend:**

- **Hive**: Green line
- **Swarm**: Orange line
- **Parallel**: Dark red line
- **SW**: Maroon line

The graphs indicate that while Hive and Swarm maintain a consistent speedup, Parallel and SW show varying degrees of scalability, with SW struggling to scale beyond 100 cores.
Swarm scales well sometimes
Hive is faster than Swarm

- **kcore**
- **setcover**
- **astar**
- **bfs**
- **sssp**
- **msf**
- **mis**
- **mm**
- **rbp**

### Graphs

- **Hive** (green line)
- **Swarm** (orange line)
- **Parallel** (red line)
- **SW** (brown line)

**System Size**

- **Speedup**

<table>
<thead>
<tr>
<th>System Size</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>274x</td>
</tr>
<tr>
<td>256</td>
<td>270x</td>
</tr>
</tbody>
</table>
Hive is faster than Swarm

Hive is up to 2.8x faster than Swarm

- Hive is up to 2.8x faster than Swarm.
Breaking down Hive vs. Swarm at 256 cores
Hive does less work
Hive does less work

Normalized execution time

<table>
<thead>
<tr>
<th>Dataset</th>
<th>kcore</th>
<th>setcover</th>
<th>astar</th>
<th>bfs</th>
<th>sssp</th>
<th>msf</th>
<th>mis</th>
<th>mm</th>
<th>rbp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spill</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Empty</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

40% increase
Hive reduces queue pressure

Normalized execution time

Commit | Abort | Spill | Stall | Empty

S H S H S H S H S H S H S H S H

kcore setcover astar bfs sssp msf mis mm rbp

27
Conclusions and Q+A

• Priority updates are useful operations for ordered algorithms
• The scheduler dependences created by these updates require task versioning and mootness detection for speculation
• Hive extracts parallelism by speculating on data, control, and scheduler dependences

Gilead Posluns, Yan Zhu, Guowei Zhang, Mark C. Jeffrey
ISCA 2022