A Scalable Architecture for Reprioritizing Ordered Parallelism

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

ISCA 2022
Ordered algorithms use priority schedules

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

Priority schedules accelerate convergence

Dijkstra’s SSSP  Breadth First Search

Residual Belief Propagation

Priority schedules are correct

Minimum Spanning Forest

KCore

Set Cover

Maximal Independent Set
Ordered algorithms use priority schedules

\[
pq = \text{init}();
\]

\[
\text{while } (!pq.\text{empty}())
\]

\[
\text{task, ts} = pq.\text{dequeueMin}()
\]

\[
\text{task(ts)}
\]

Priority schedules accelerate convergence

Priority schedules are powerful, but hard to parallelize

Priority schedules are correct

- Minimum Spanning Forest
- KCore
- 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);
}
```
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
Task-Based Execution Model

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

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

```cpp
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

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

Swarm’s execution model does not support priority updates

swarm::enqueue(
    fn,  //what to do
    ts,  //when to do it
    args //what to do it with);
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) {
            prios[nbr]--;
            pq.enqueue(nbr, prios[nbr]);
        }
}
```

Manual priority tracking

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

```java
PriorityQueue pq;

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

```java
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]);
    }
```
Updateable schedules are efficient

Swarm Task Graph

Input graph

Dependence Task

Updateable Task Graph

"Updates" enqueue a new Task

Priority = Remaining Degree

Updates change priority of a task

Priority = Remaining Degree
Updateable schedules are efficient

Enqueue-only schedule has 3 more tasks than updateable schedule

Swarm Task Graph

Dependence Task

Updateable Task Graph

“Updates”

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

Input graph

Enqueue - only schedule has 3 more tasks than updateable schedule

Priority = Remaining Degree

1
2
3

Priority = Remaining Degree

1
2
3

Updates change
Moot tasks outnumber useful dequeues

![Bar chart showing the comparison between Moot Tasks and Useful Task for different algorithms]

- KCore
- Set Cover
- BFS
- SSSP
- MSF
- MIS
- RBP

The chart indicates that the number of Moot Tasks significantly exceeds the number of Useful Tasks for each algorithm.
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);
    }
}
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);
    }
}
```

Update binds a task to an object and schedules it to run.

<table>
<thead>
<tr>
<th>Task</th>
<th>Object</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>hive::update(&amp;removeV, nbr, prev-1);</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>
Updating an occupied Hive object

```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);
    }
}
```

Object Table

- **Object Table**
  - A
  - B
  - C
  - D
  - E
  - F

![Diagram of nodes A, B, C, D, E, F with edges and a cross indicating a moot object](Image of diagram)

Moot
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);
    }
}

Updating an occupied Hive object

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>

No Priority Queue in Sequential Implementation
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

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

```plaintext
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
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 detected by comparing timestamps of parents
Hive extends the Swarm architecture

64-tile, 256-core chip

Tile organization

- Router
- GVT Arb. Node
- L1/D
- Core

Task unit structures

- L3 & Dir Bank
- L2
- L1/D
- Core

Task Send buffer

Commit queue

Task queue

Object map

Memory

A

B

C

D

E

F

+20B

Hive hardware additions

Swarm hardware additions

9% Task Unit Area Increase

3% Area of a Nehalem Processor
Evaluation
Methodology

Event-driven, Pin-based Simulator

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

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

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

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

System Size

Speedup

Hive
Swarm
Parallel
SW
Swarm scales well sometimes

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

Graphs showing speedup vs. system size for different algorithms and systems.
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

40%
Hive reduces queue pressure
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