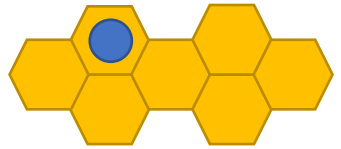
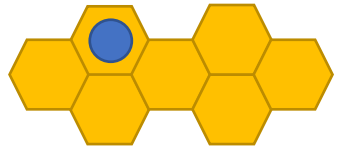
A Scalable Architecture for Reprioritizing Ordered Parallelism

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

ISCA 2022

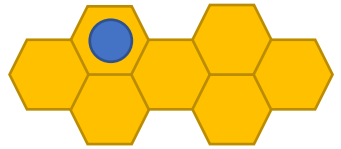
Ordered algorithms use priority schedules





Ordered algorithms use priority schedules

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



Ordered algorithms use priority schedules

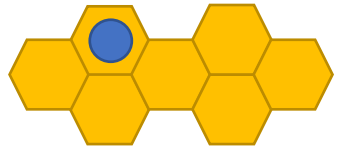
```
pq = init();
while (!pq.empty())
        task, ts = pq.dequeueMin()
        task(ts)
```

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

Breadth First Search

Residual Belief Propagation

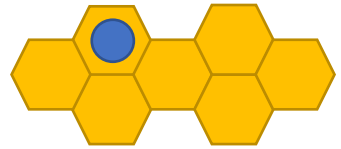
Priority schedules are correct

Minimum Spanning Forest

KCore

Set Cover

Maximal Independent Set



Ordered algorithms use priority schedules

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

Priority schedules accelerate convergence

Priority schedules are powerful, but hard to parallelize

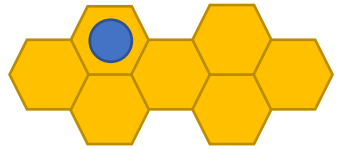
Priority schedules are correct

Minimum Spanning Forest

KCore

Set Cover

Maximal Independent Set

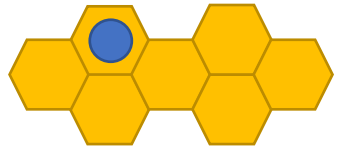


Hive parallelizes priority updates

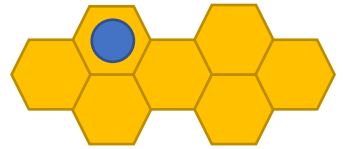
Hive builds on Swarm to provide a parallel **priority update** operation in speculative task-parallel hardware

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



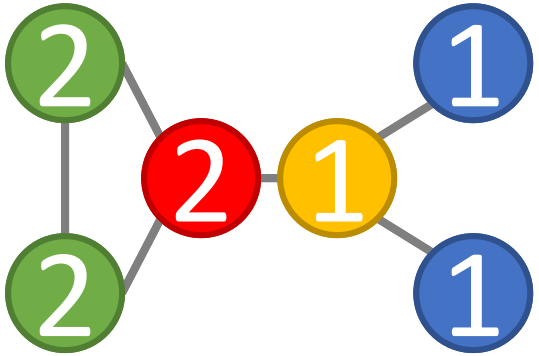
KCore requires priority updates

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)
}}
```

Input Graph

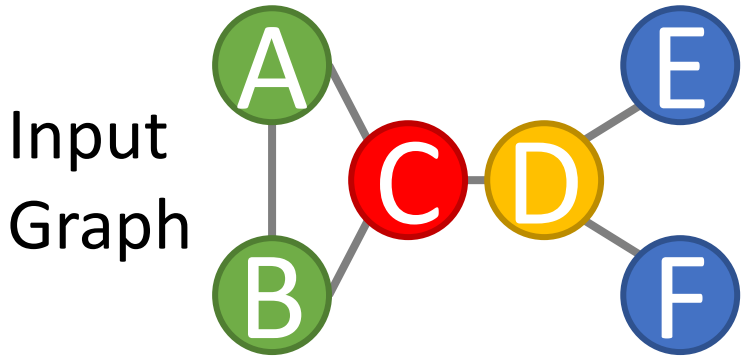


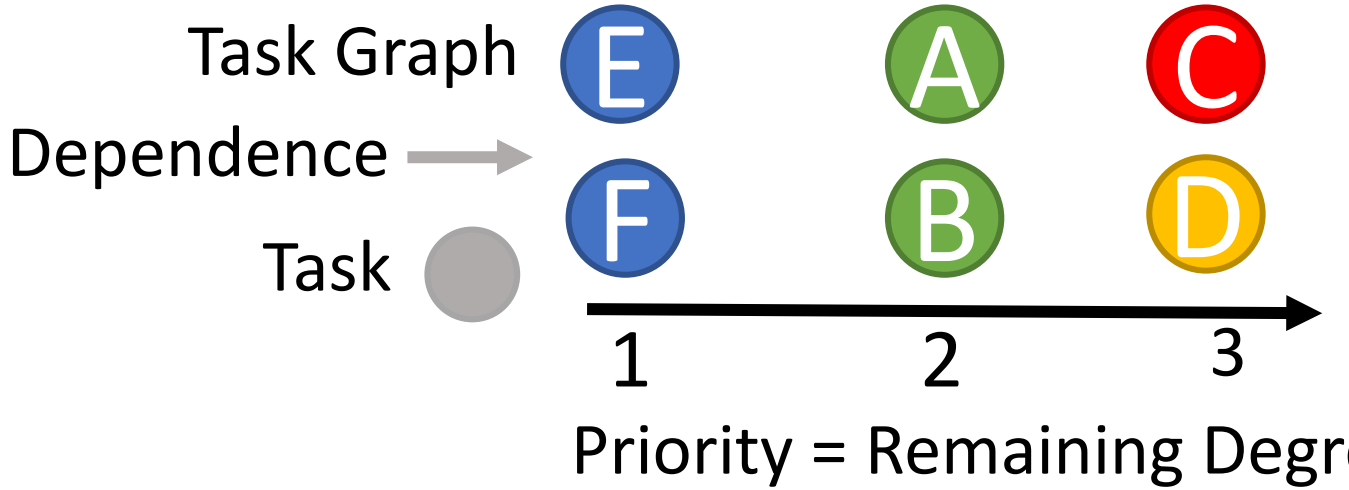
KCore requires priority updates

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)
}}
```





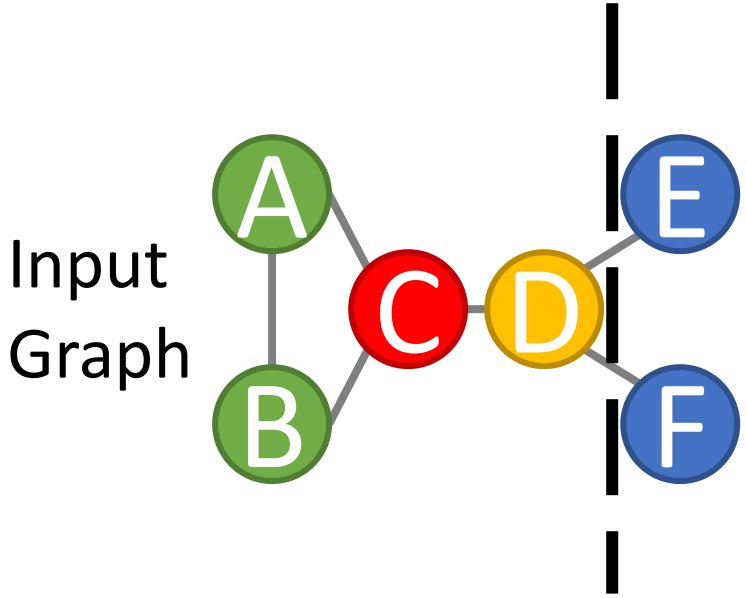
Priority = Remaining Degree

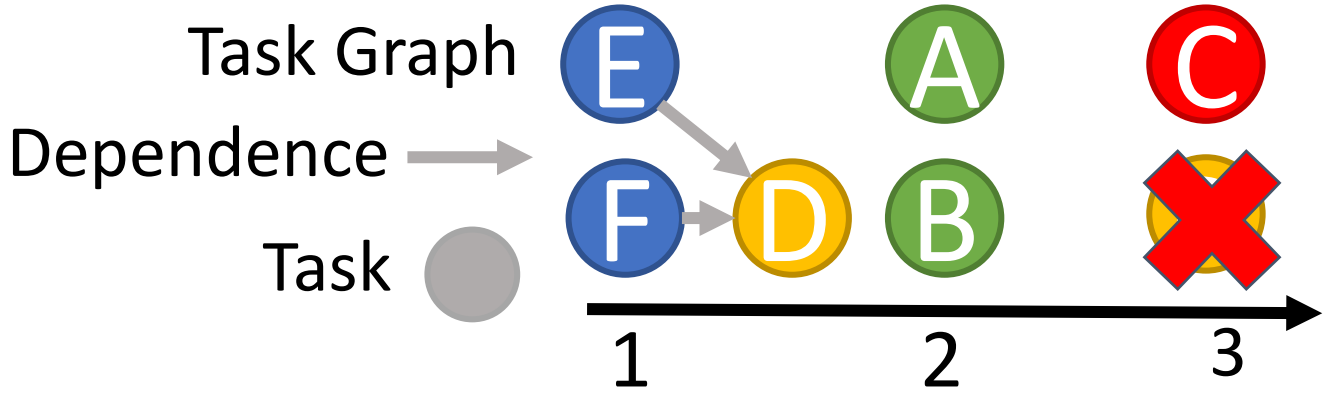
KCore requires priority updates

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)
}}
```





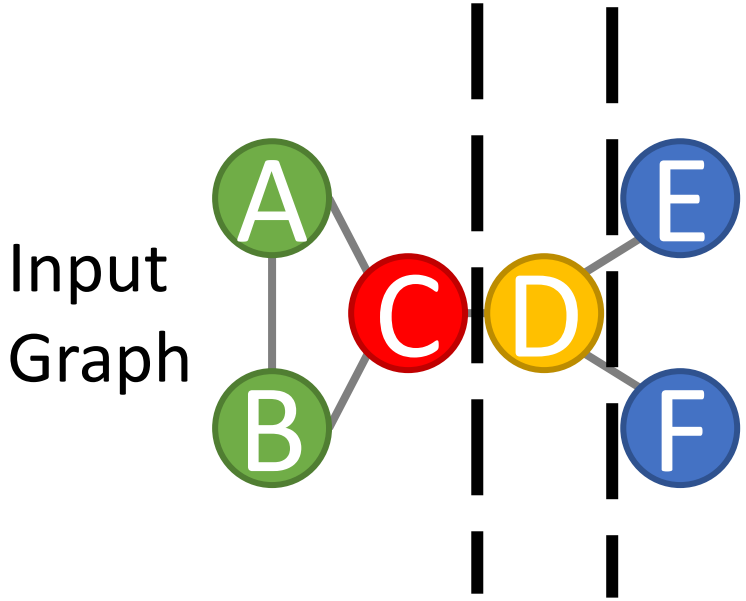
Priority = Remaining Degree

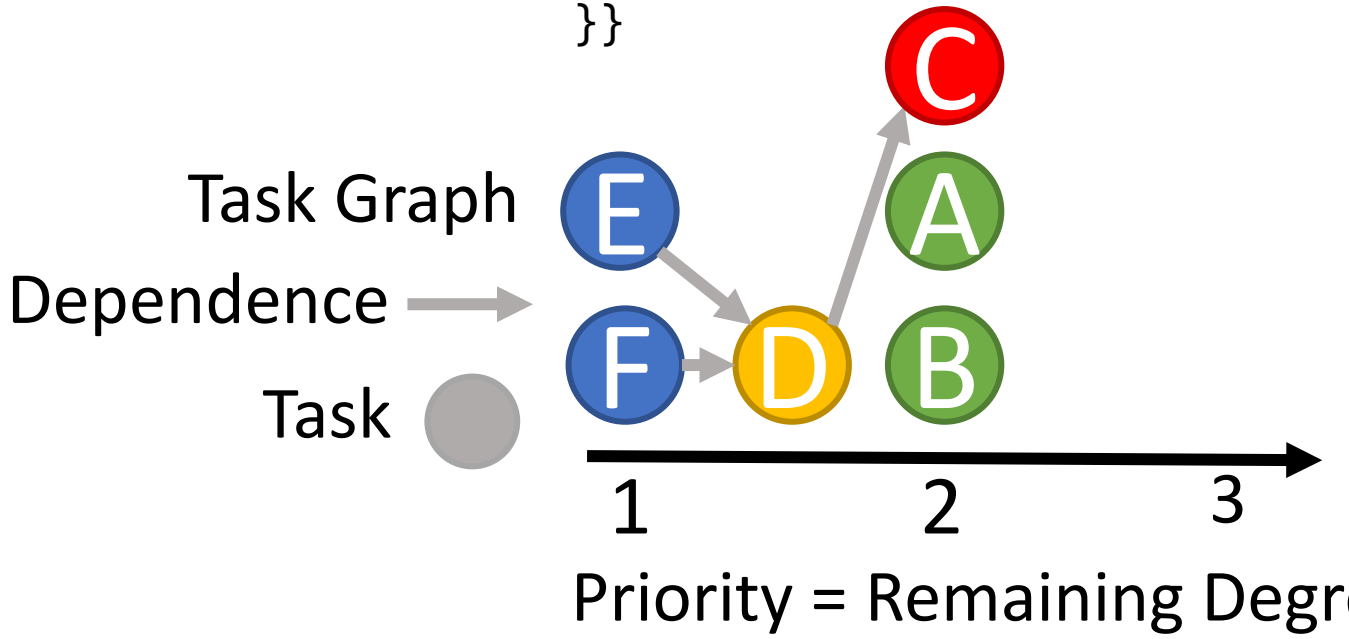
KCore requires priority updates

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)
}}
```





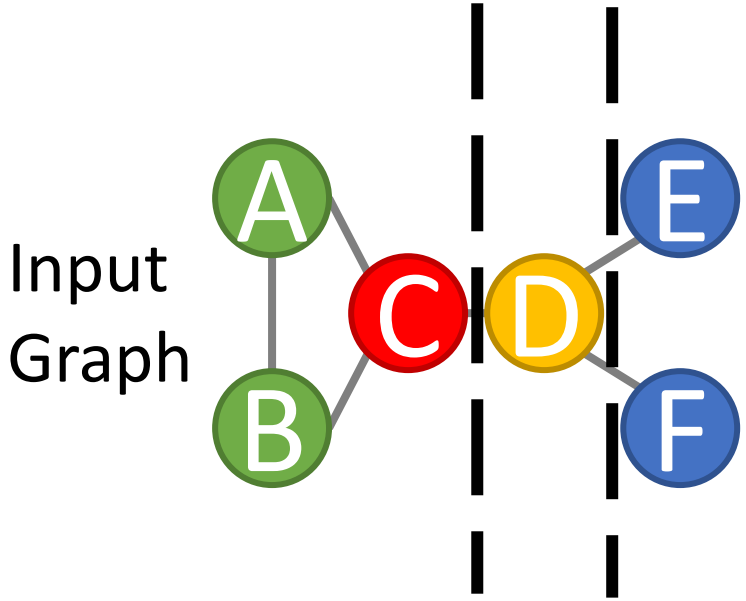
Priority = Remaining Degree

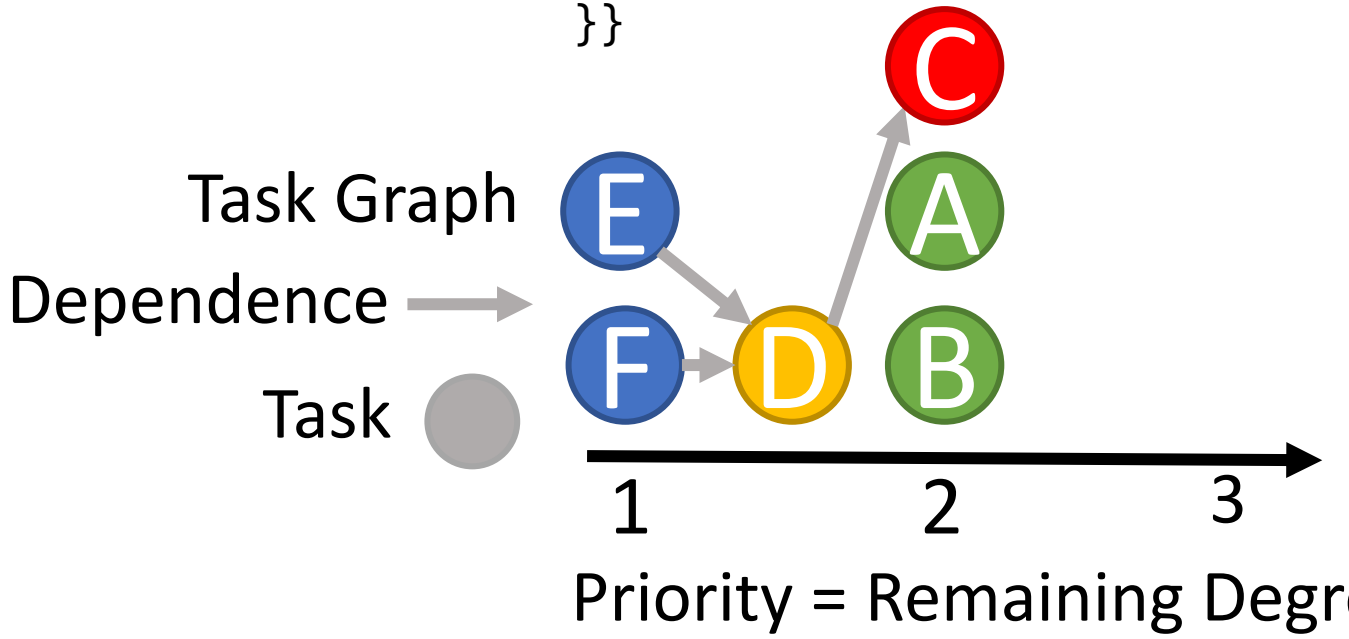
KCore requires priority updates

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

To find: repeatedly remove lowest degree vertex

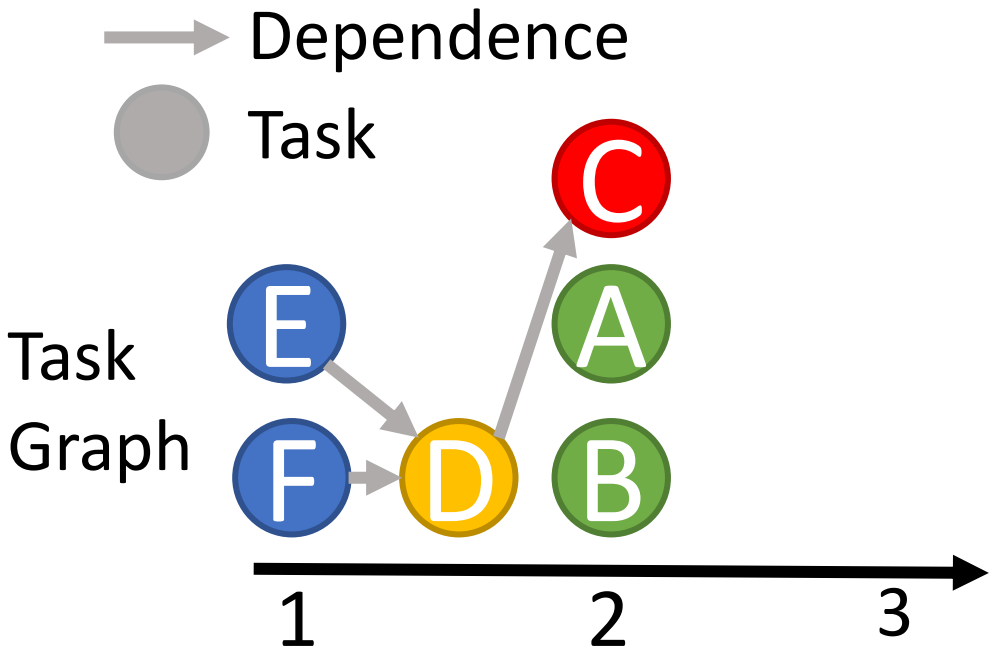
```
PriorityQueue pq;
for (int v: G.V)
  pq.enqueue(v, G.degree[v])
while (!pq.empty()) {
  int v, int prio = pq.dequeueMin();
  coreness[v] = prio;
  for (int nbr : G.edges[v])
    pq.decrementPrio(nbr)
}}
```





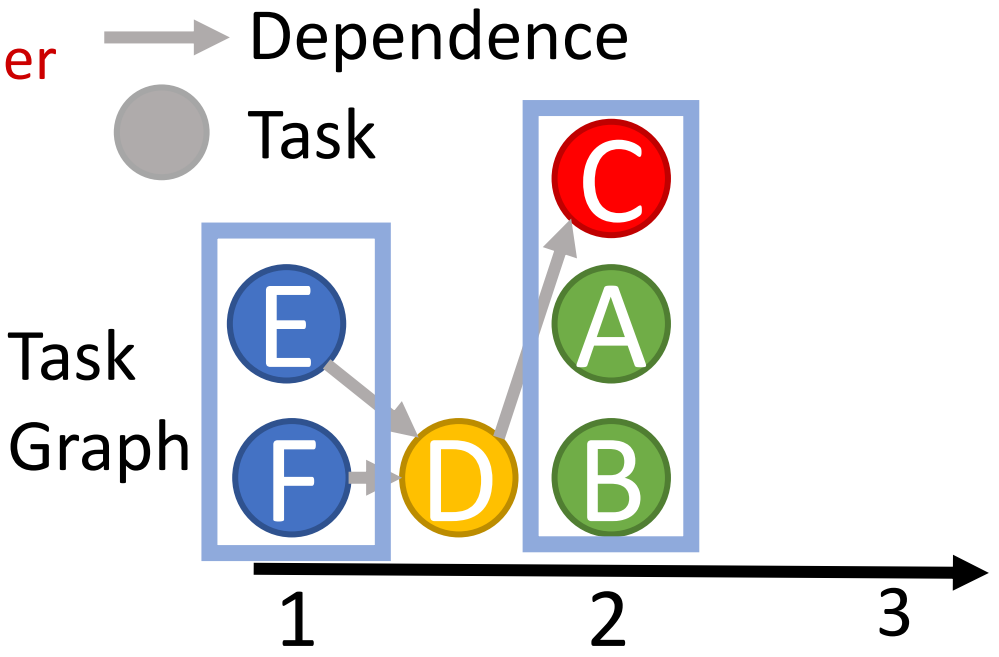
Priority = Remaining Degree

Where’s the parallelism in KCore?



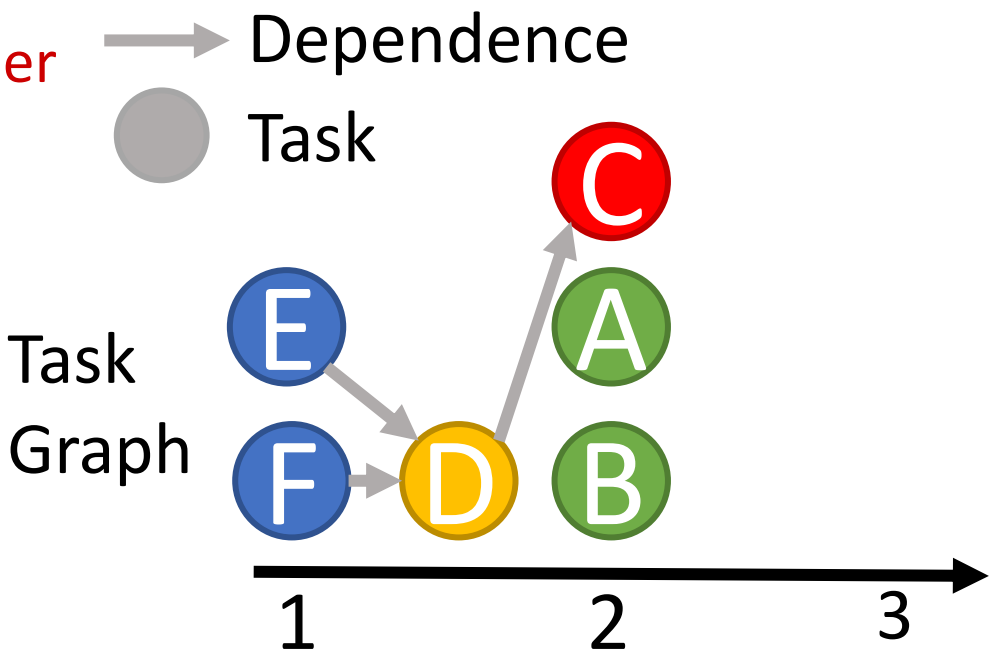
Where's the parallelism in KCore?

- 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



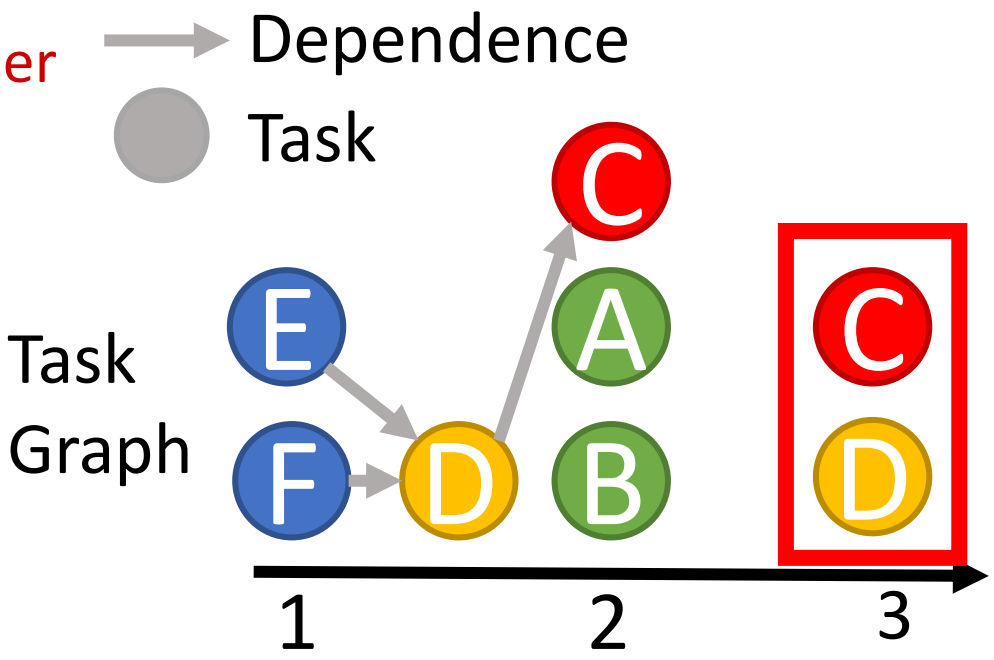
Where's the parallelism in KCore?

- 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



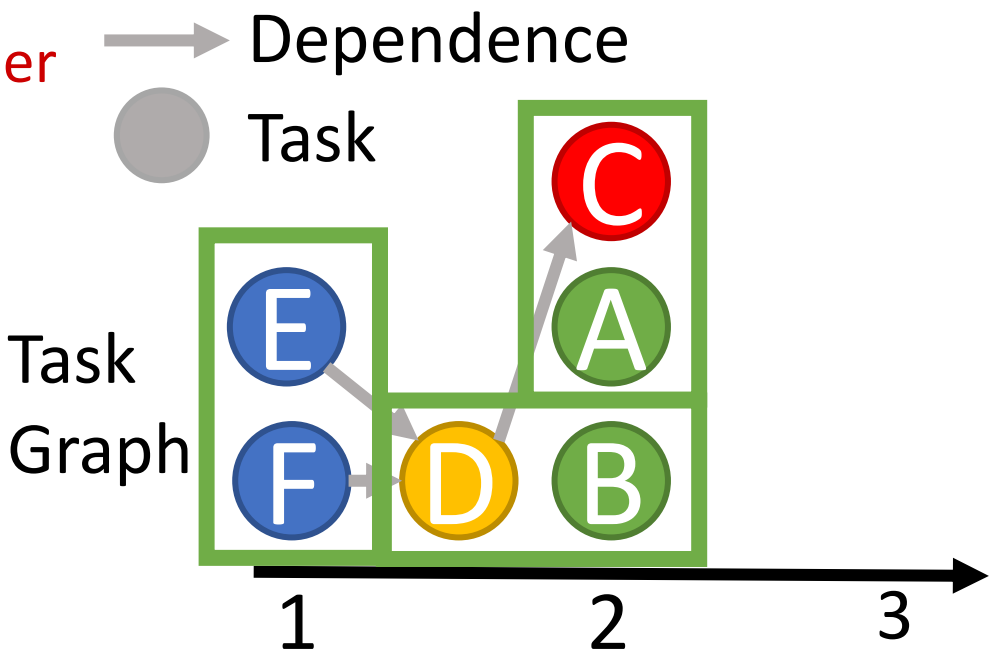
Where's the parallelism in KCore?

- 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?

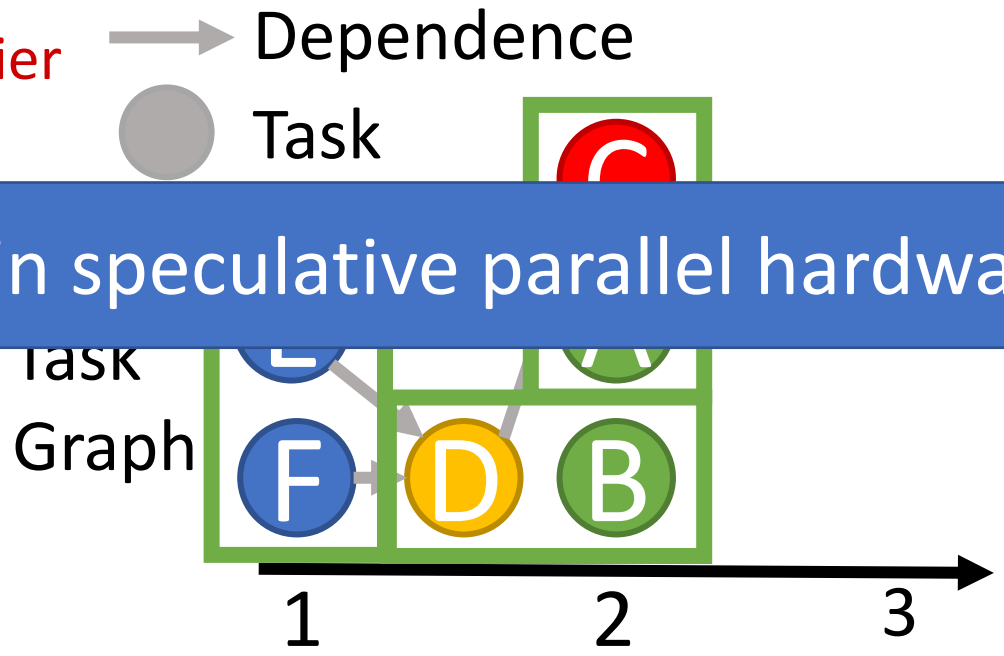
- Bulk-Synchronous [Dhulipala et al. SPAA'17] [Dadu et al. ISCA'21]
  - Effective when many tasks per barrier
  - Nearly sequential when few tasks per barrier
- Relaxed [Khan et al HPCA'22] [Yesil et al. SC'19] [Dadu et al. ISCA'21]
  - Can always find parallelism
  - loses efficiency as it scales
  - 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

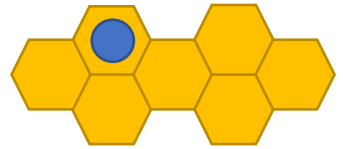


Where's the parallelism in KCore?

- 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]
  - 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

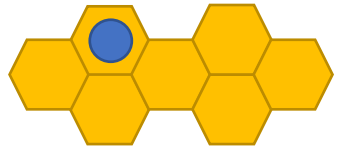
Our goal is to support priority updates in speculative parallel hardware





Swarm [Jeffrey et al. MICRO'15] speculates without updates

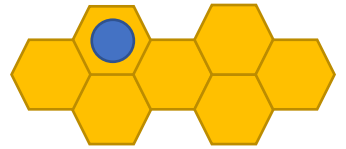
Task-Based Execution Model



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

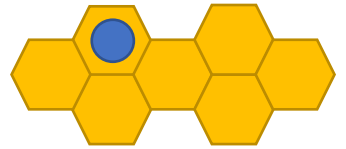


Swarm [Jeffrey et al. MICRO'15] speculates without updates

Task-Based Execution Model

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

- Programs consist of timestamp-ordered tasks
- Tasks appear to execute in timestamp order



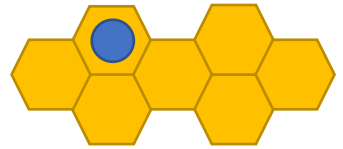
Swarm [Jeffrey et al. MICRO'15] speculates without updates

Task-Based Execution Model

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

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

```
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

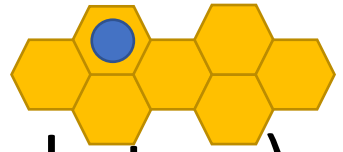
Task-Based Execution Model

- Programs consist of timestamp-ordered tasks

```
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)

```
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)

```
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)

```
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]);
}
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])
}}
```

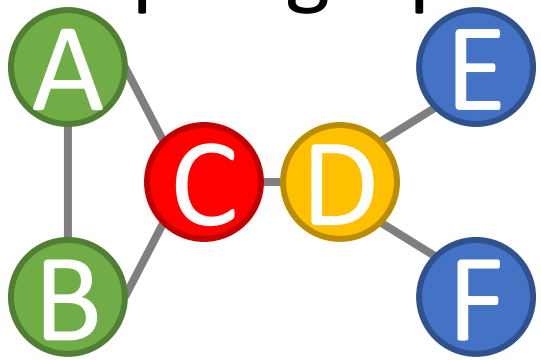
Manual priority tracking

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

Early exit for **moot** tasks

Updateable schedules are efficient

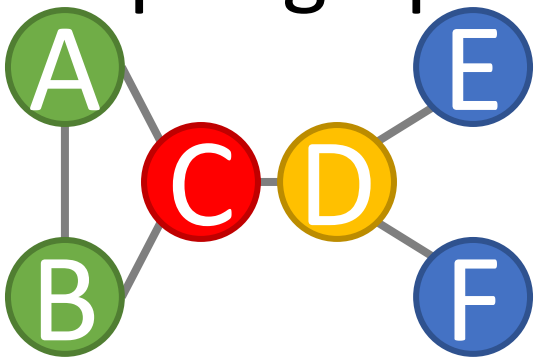
Input graph

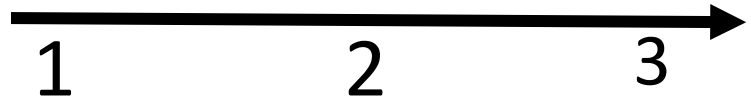


Updateable schedules are efficient

Updateable Task Graph

Input graph





Priority = Remaining Degree

Updateable schedules are efficient

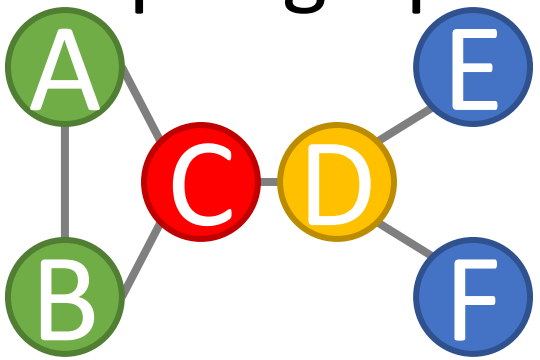
Swarm Task Graph

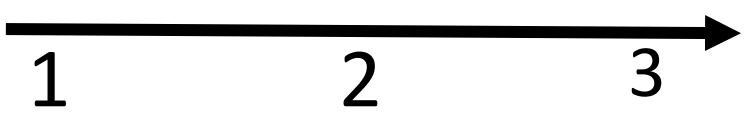
Dependence

Task

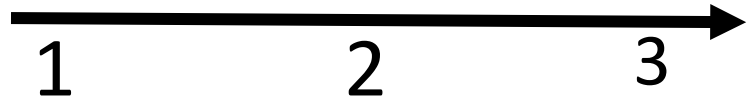
Updateable Task Graph

Input graph





Priority = Remaining Degree



Priority = Remaining Degree

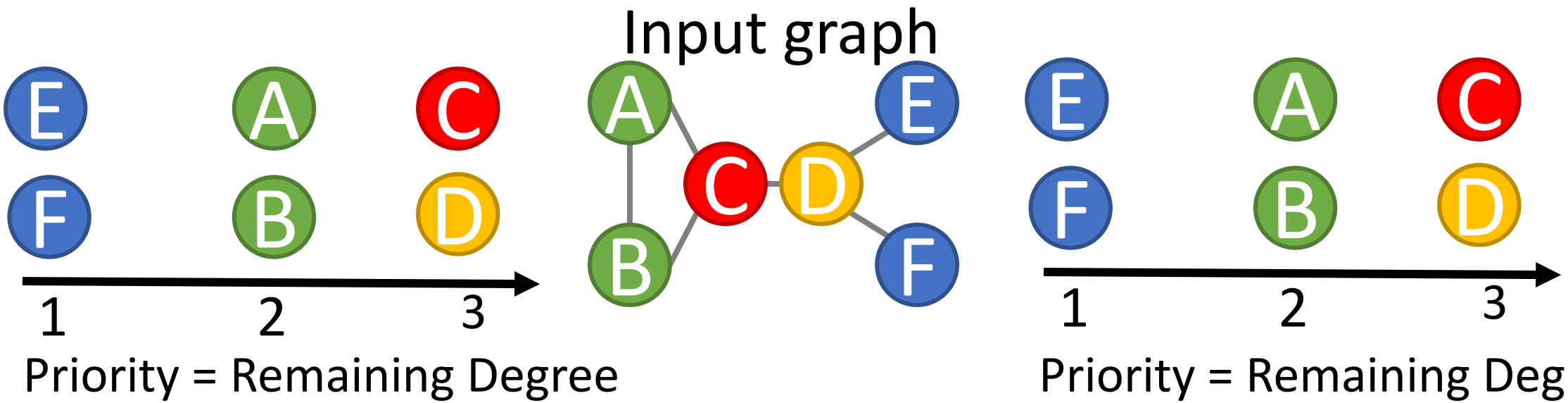
Updateable schedules are efficient

Swarm Task Graph

Dependence

Task

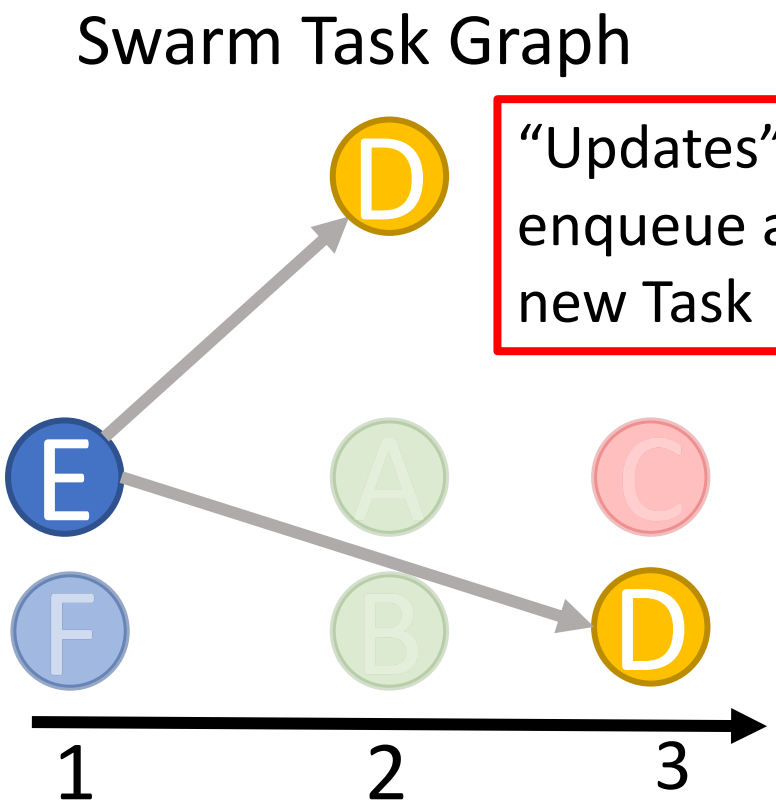
Updateable Task Graph



Priority = Remaining Degree

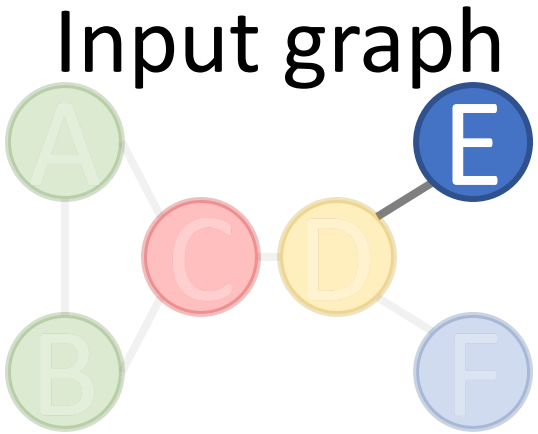
Priority = Remaining Degree

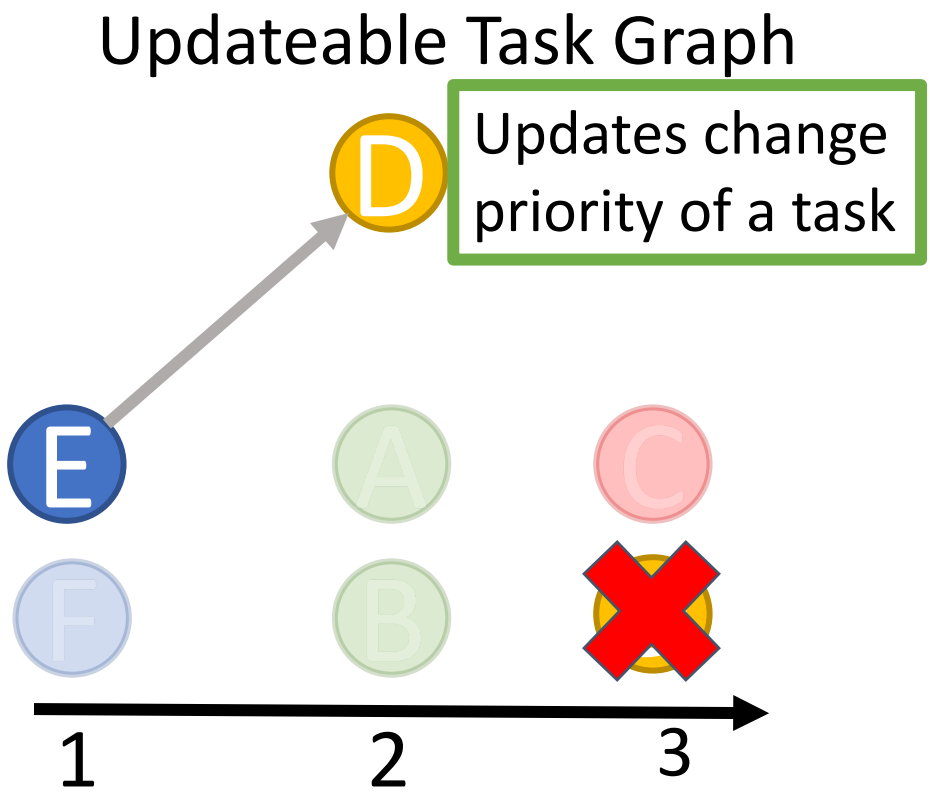
Updateable schedules are efficient



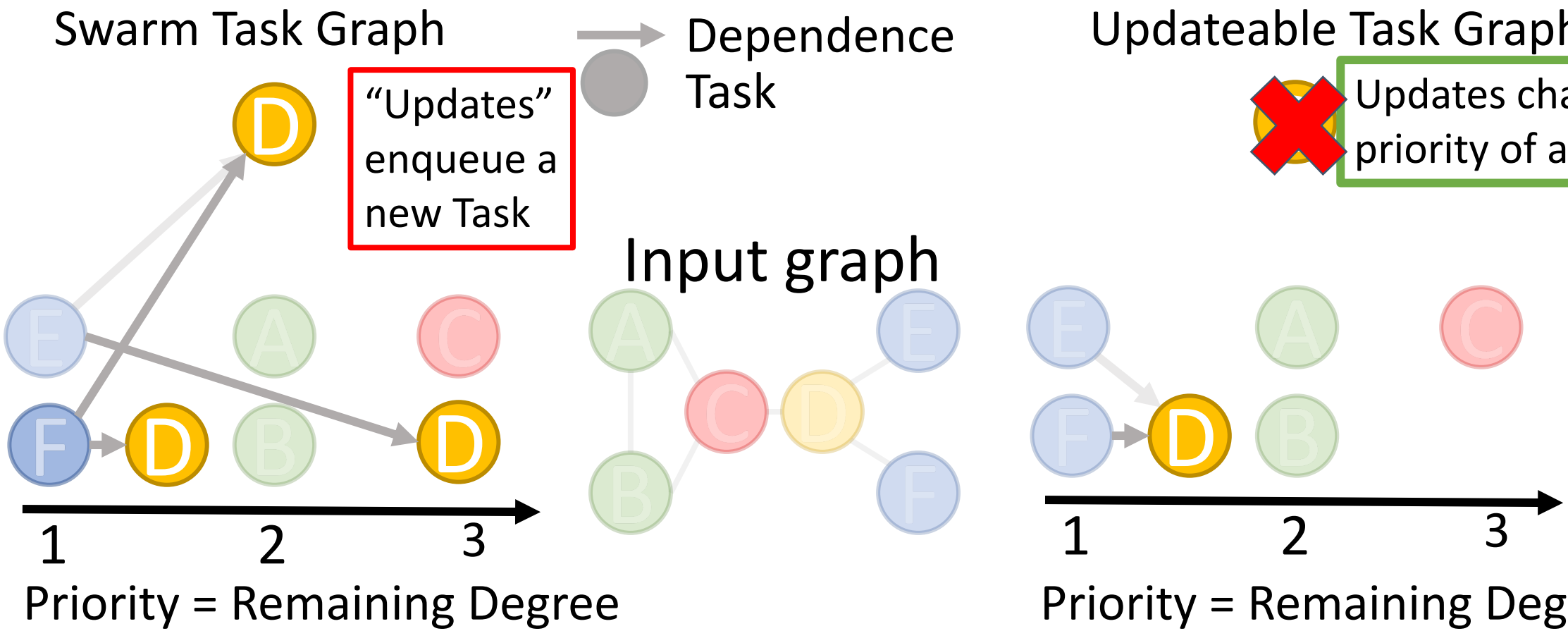
Dependence Task

"Updates" enqueue a new Task

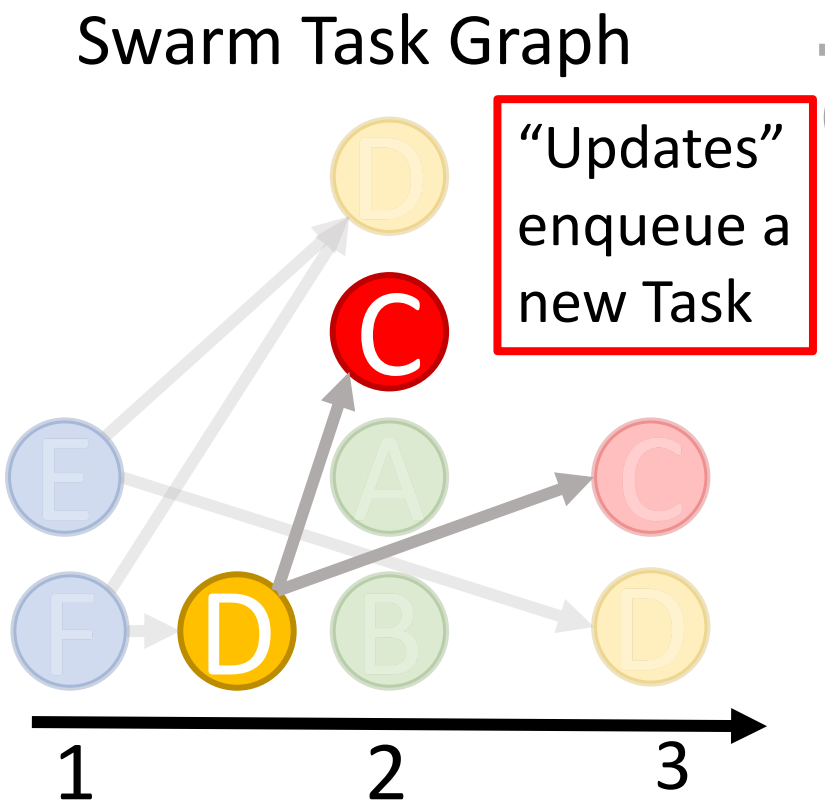


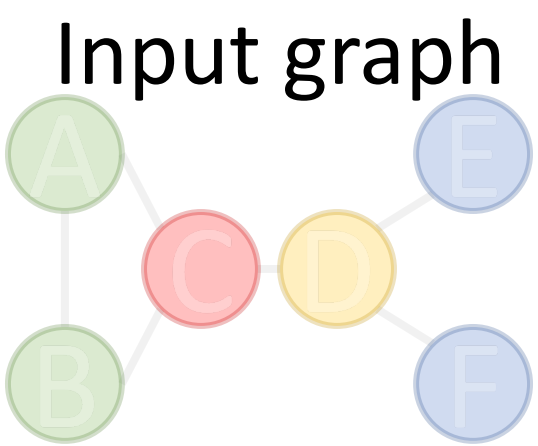


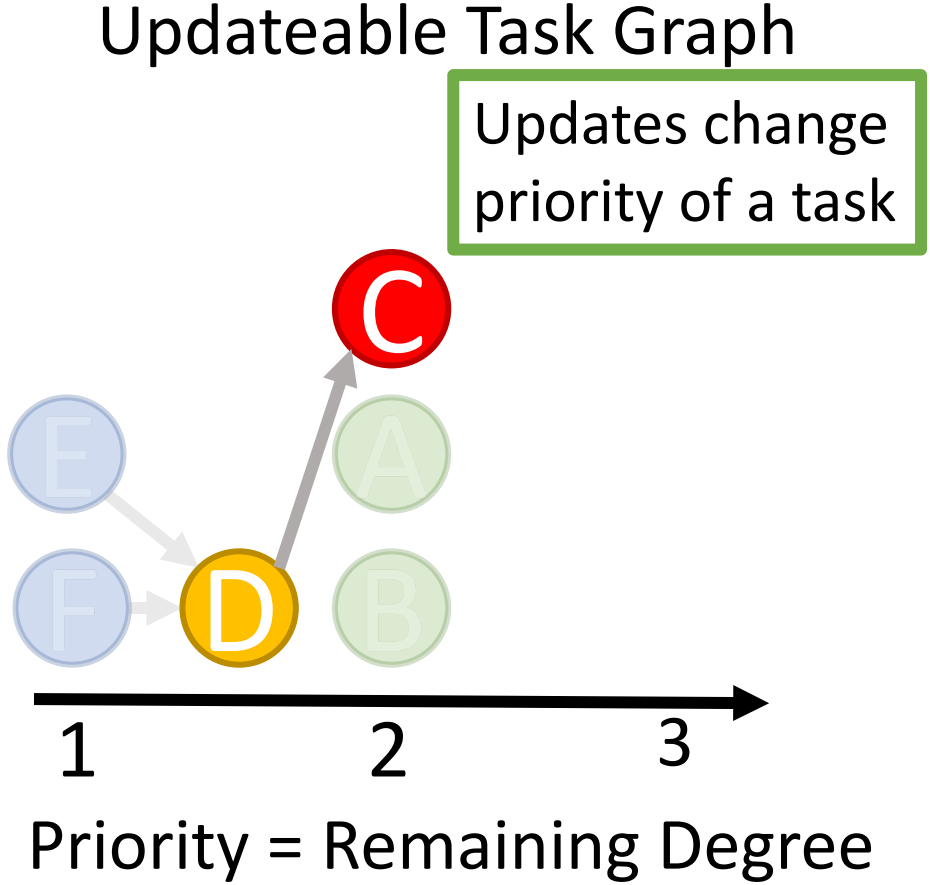
Updateable schedules are efficient



Updateable schedules are efficient

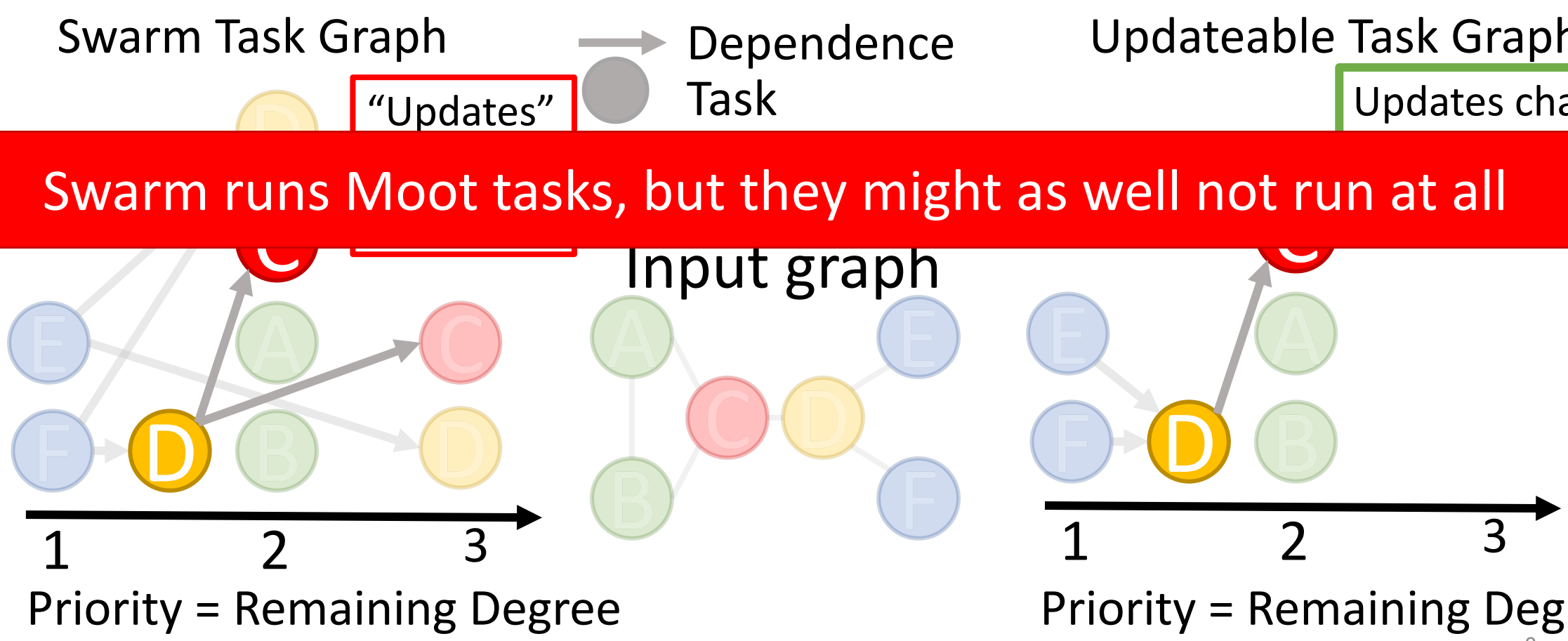




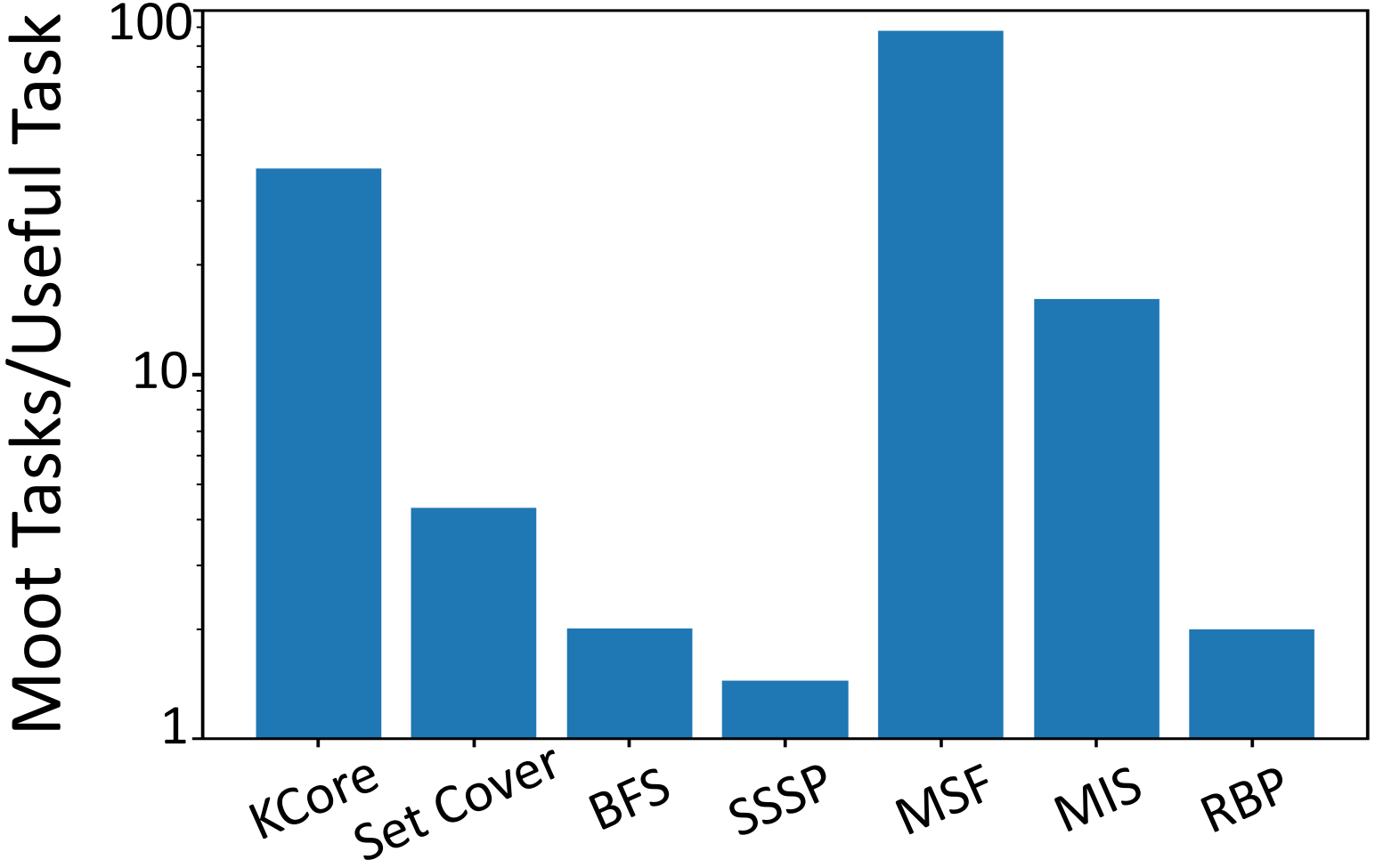


Updateable schedules are efficient

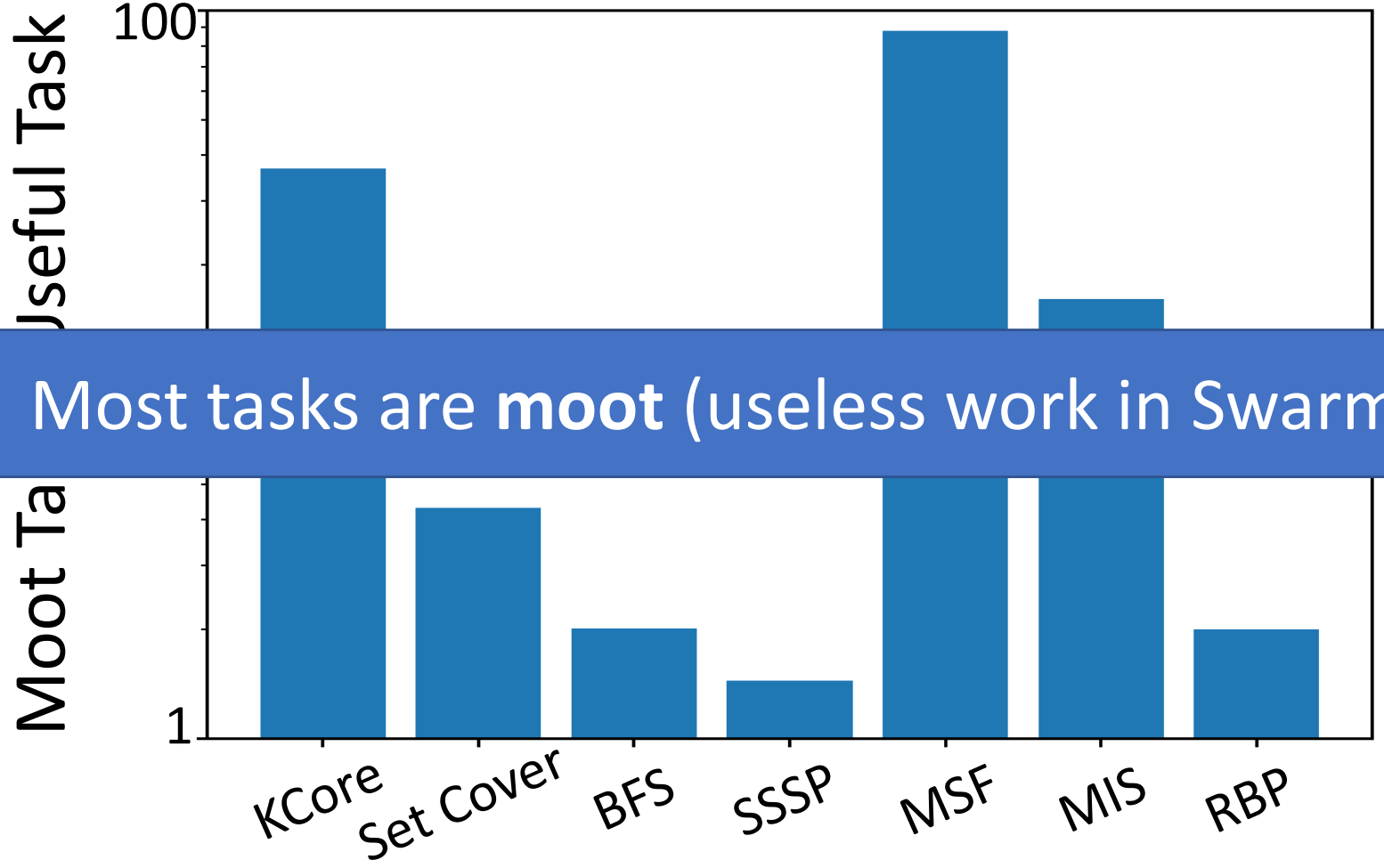
Enqueue-only schedule has 3 more tasks than updateable schedule



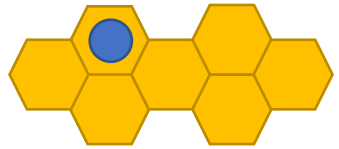
Moot tasks outnumber useful dequeues



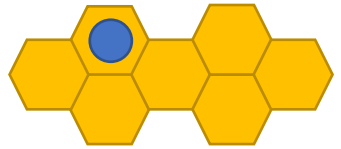
Moot tasks outnumber useful dequeues



Most tasks are **moot** (useless work in Swarm)



The Hive Execution Model



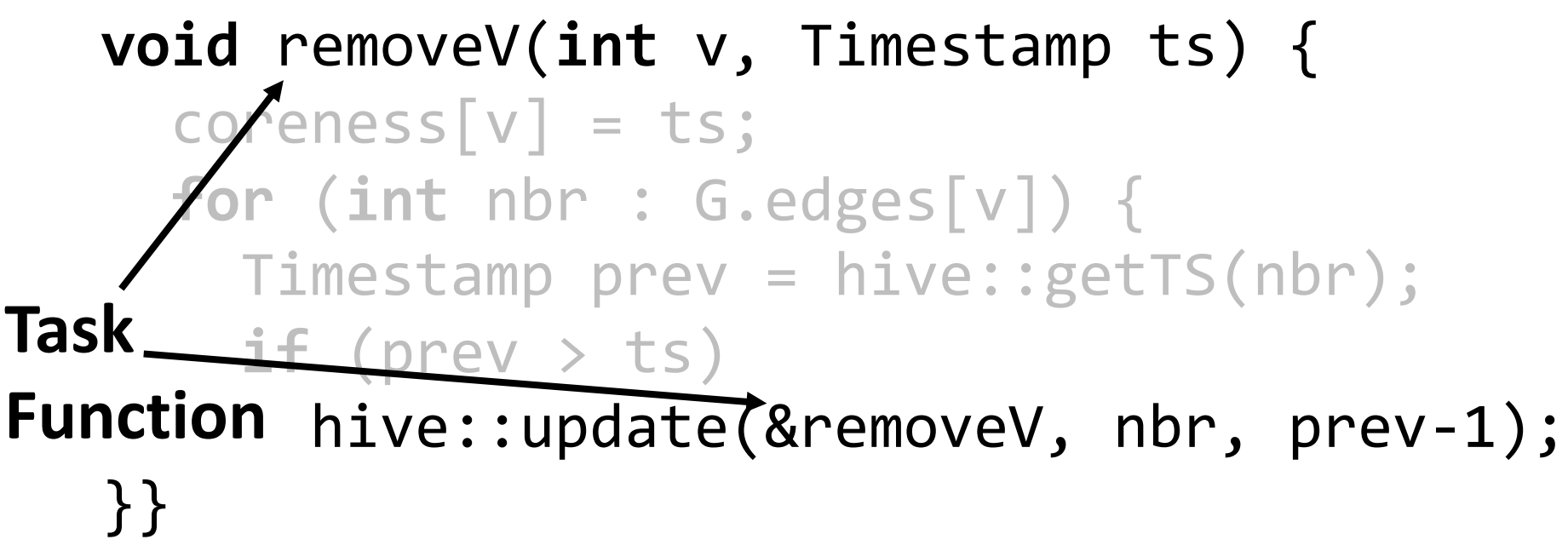
Understanding Hive tasks and objects

```
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);
}}
```

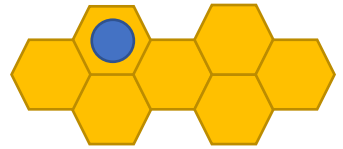
Understanding Hive tasks and objects

```
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



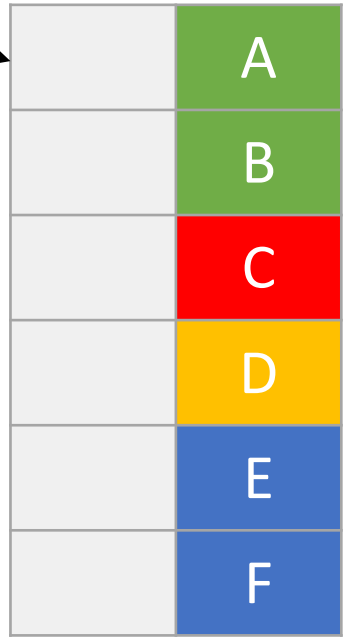

```
void removeV(int v, Timestamp ts) {
  coreness[v] = ts;
  for (int nbr : G.edges[v]) {
    Timestamp prev = hive::getTS(nbr);
    if (prev > ts)
      hive::update(&removeV, nbr, prev-1);
  }}
```



Understanding Hive tasks and objects

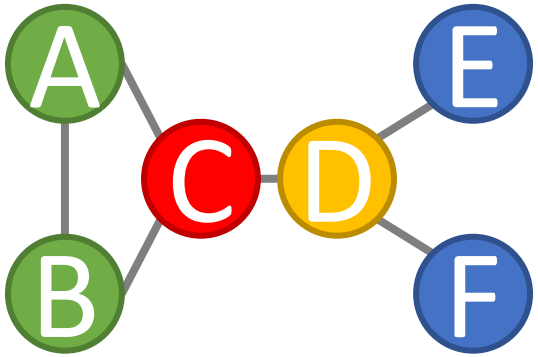
Object

Object Table

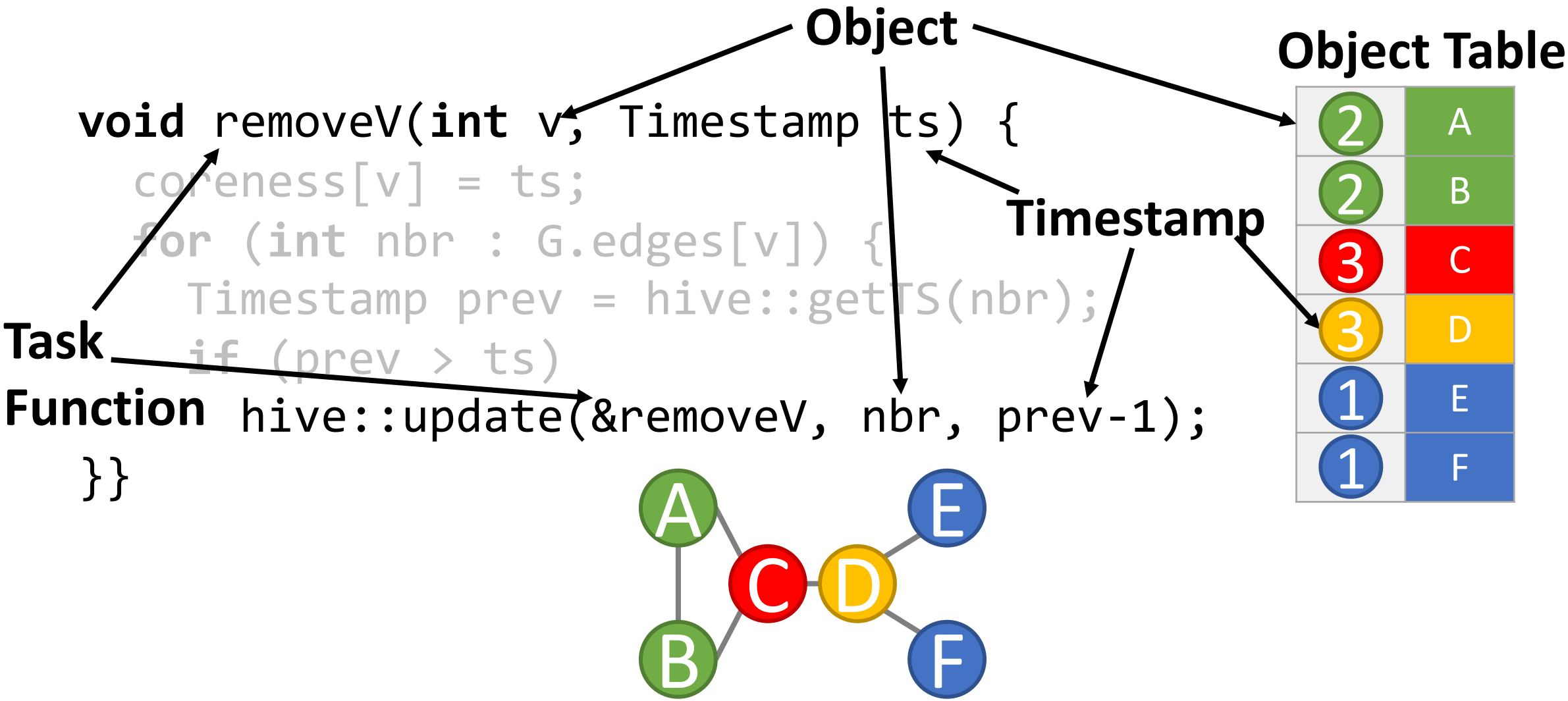


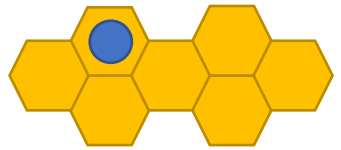

```
void removeV(int v, Timestamp ts) {
  coreness[v] = ts;
  for (int nbr : G.edges[v]) {
    Timestamp prev = hive::getTS(nbr);
    if (prev > ts)
      hive::update(&removeV, nbr, prev-1);
  }}
```

Task Function



Understanding Hive tasks and objects





Understanding Hive tasks and objects

Object

Object Table

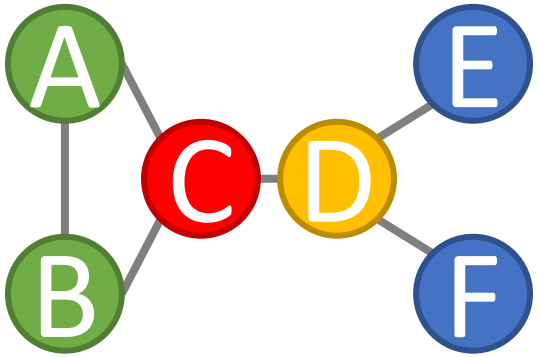
```
void removeV(int v, Timestamp ts) {
  coreness[v] = ts;
  for (int nbr : G.edges[v]) {
    if (prev > ts)
      hive::update(&removeV, nbr, prev-1);
  }}
```

Timestamp

Task Function

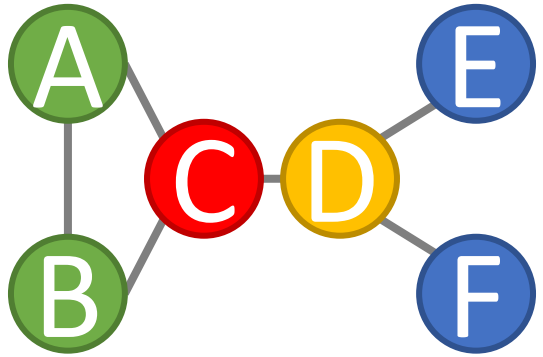
Update binds a task to an object and schedules it to run

2	A
2	B
1	E
1	F



Updating an occupied Hive object

```
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

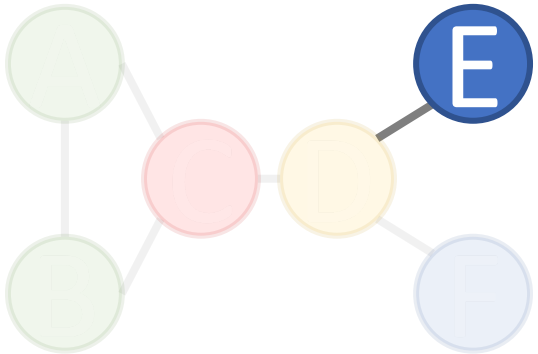
2	A
2	B
3	C
3	D
1	E
1	F

Updating an occupied Hive object

```
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);
}}
```

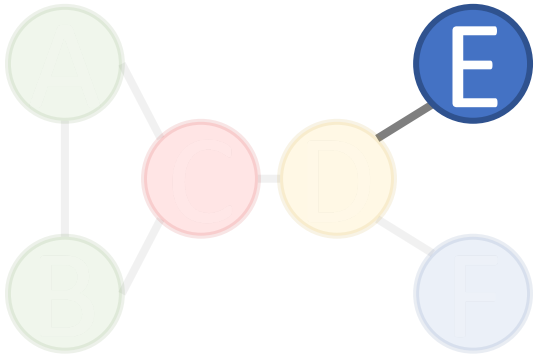
Object Table

2	A
2	B
3	C
3	D
1	E
1	F



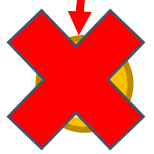
Updating an occupied Hive object

```
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

2	A
2	B
3	C
2	D
1	E
1	F

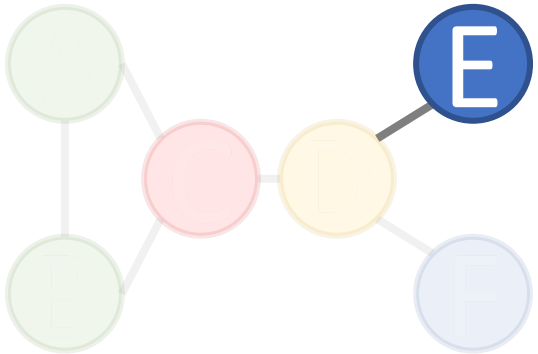


Moot

Updating an occupied Hive object

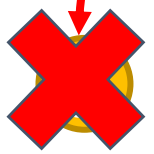
```
void removeV(int E, Timestamp ts) {
  coreness[E] = ts;
  for (int D : G.edges[E]) {
    if (prev > ts)
      hive::update(&removeV, D, prev-1);
}}
```

Hive doesn't waste time or space on **moot** tasks



Object Table

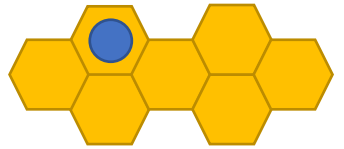
2	A
2	B
1	E
1	F



Moot

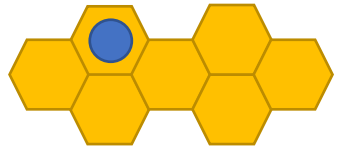
Hive supports many programming patterns

Benchmark	Increment	UpdateMin	Cancel	Update
KCore	✓			
Set Cover	✓			
Astar				
Breadth First Search				
SSSP				
Minimum Spanning Forest				
Maximal Independent Set				
Maximal Matching				
Residual Belief Propagation				



Hive supports many programming patterns

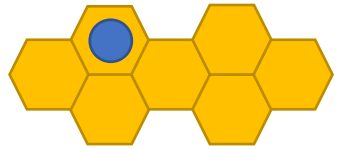
Benchmark	Increment	UpdateMin	Cancel	Update
KCore	✓			
Set Cover	✓	✓		
Astar		✓		
Breadth First Search		✓		
SSSP		✓		
Minimum Spanning Forest		✓	✓	
Maximal Independent Set			✓	
Maximal Matching			✓	
Residual Belief Propagation				✓



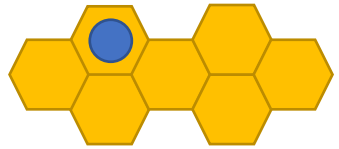
Hive supports many programming patterns

Benchmark	Increment	UpdateMin	Cancel	Update
KCore	✓			
Set Cover	✓	✓		
Astar		✓		
Breadth First Search		✓		
SSSP		✓		
Minimum Spanning Forest		✓	✓	
Maximal Independent Set			✓	
Maximal Matching			✓	
Residual Belief Propagation				✓

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

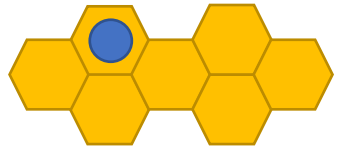
Hive speculates to run tasks in parallel

For each task, Hive speculates that:

- Eager data speculation: Predecessors have already performed their writes
- Eager control speculation: Its parent will not abort

The same as Swarm [Jeffrey et al. MICRO'15]

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

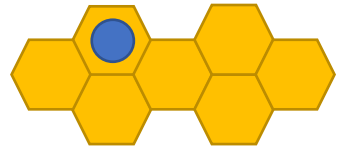


Priority updates are scheduler dependences

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

```
STR R5, [PC, #4]
ADD R1, R1, R1
```

- 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



Priority updates are scheduler dependences

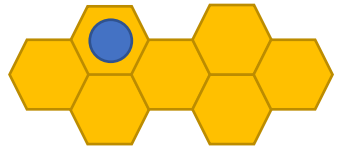
- The scheduler dependence is old
  - Found in self-modifying code [Wilkes and Renwick. ‘49]
- Created by priority updates

```
STR R5, [PC, #4]
ADD R1, R1, R1
```

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

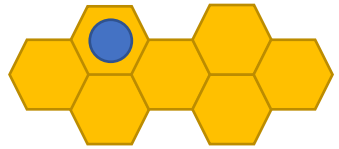
Scheduler speculation: Task versioning and Mootness detection



- 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

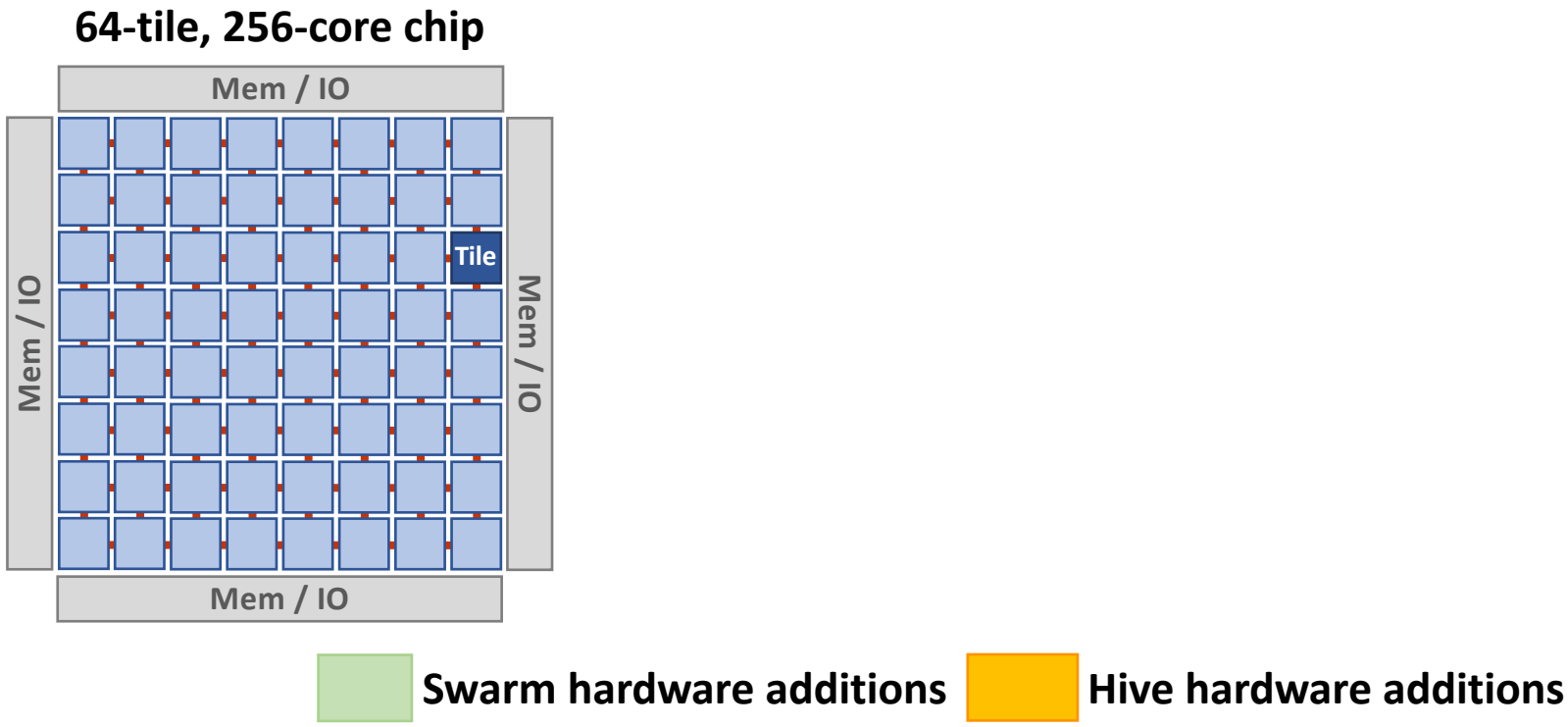


- Maintain multiple versions of each task
  - 1 for each speculative update + up to 1 non-speculative
- 1 task version is speculatively valid, all others are speculatively Moot
- When Mootness becomes non-speculative, discard the Moot version

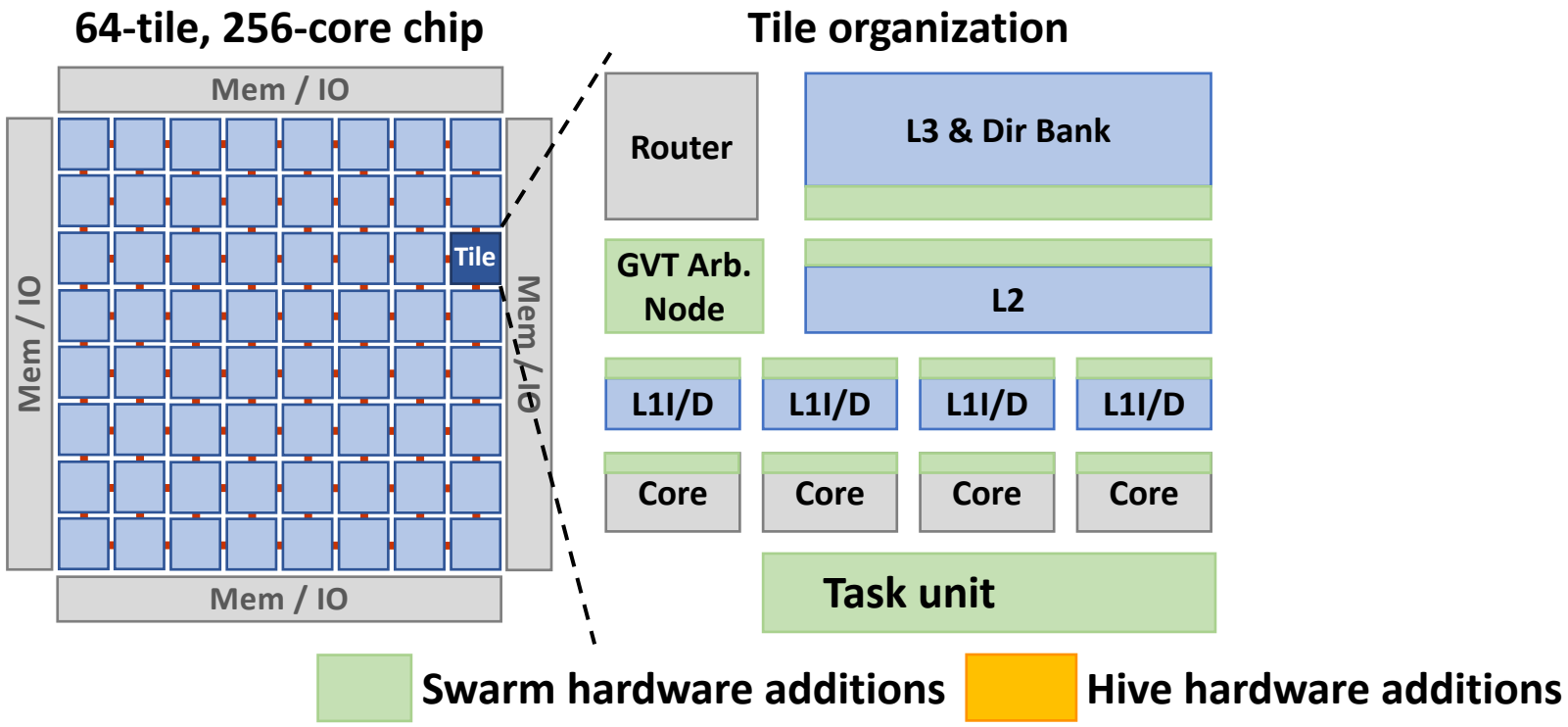
Hive avoids running moot tasks and reduces their speculative state

- Mootness can detected by comparing timestamps of parents

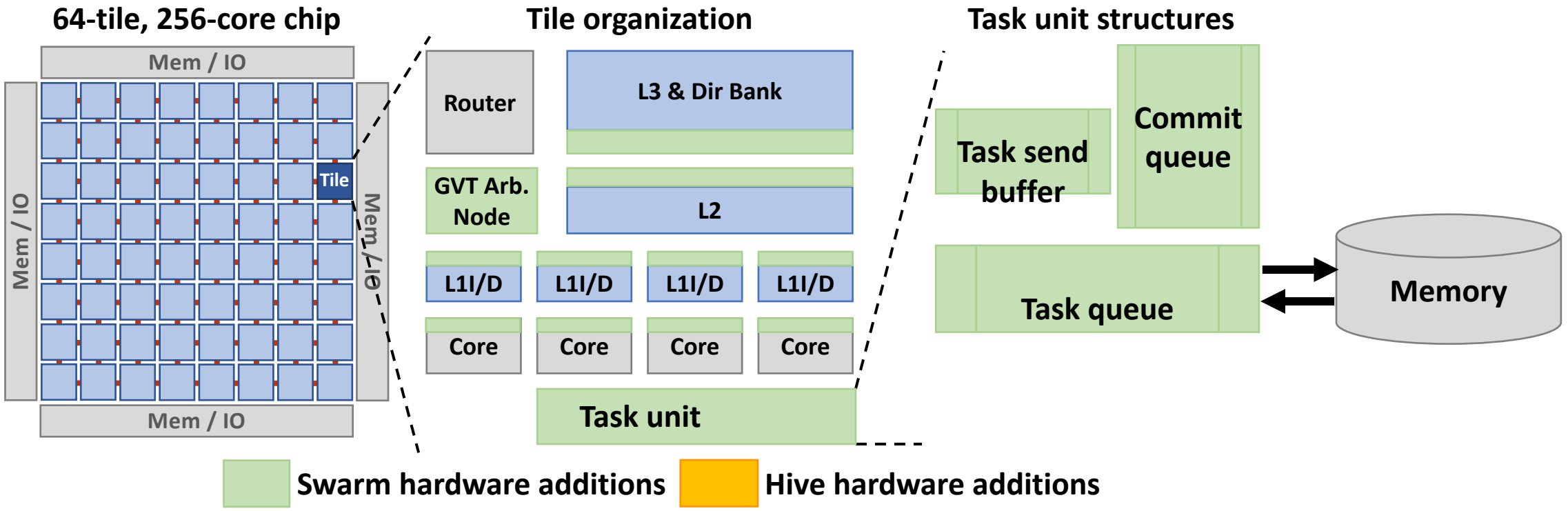
Hive extends the Swarm architecture



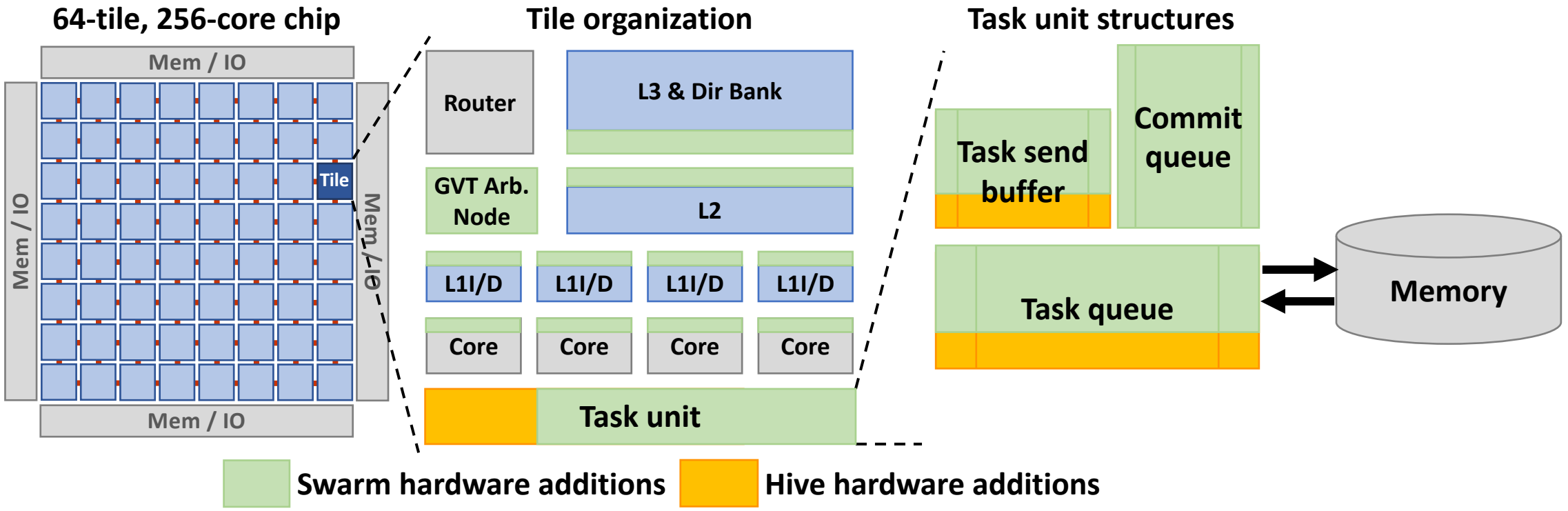
Hive extends the Swarm architecture



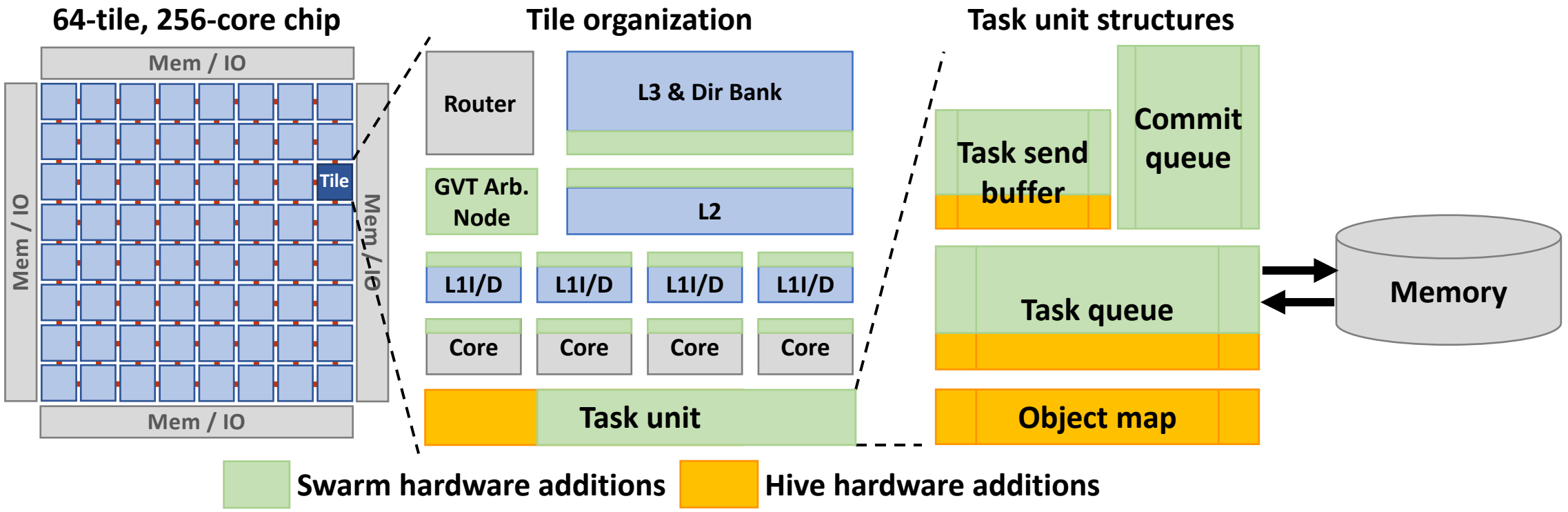
Hive extends the Swarm architecture



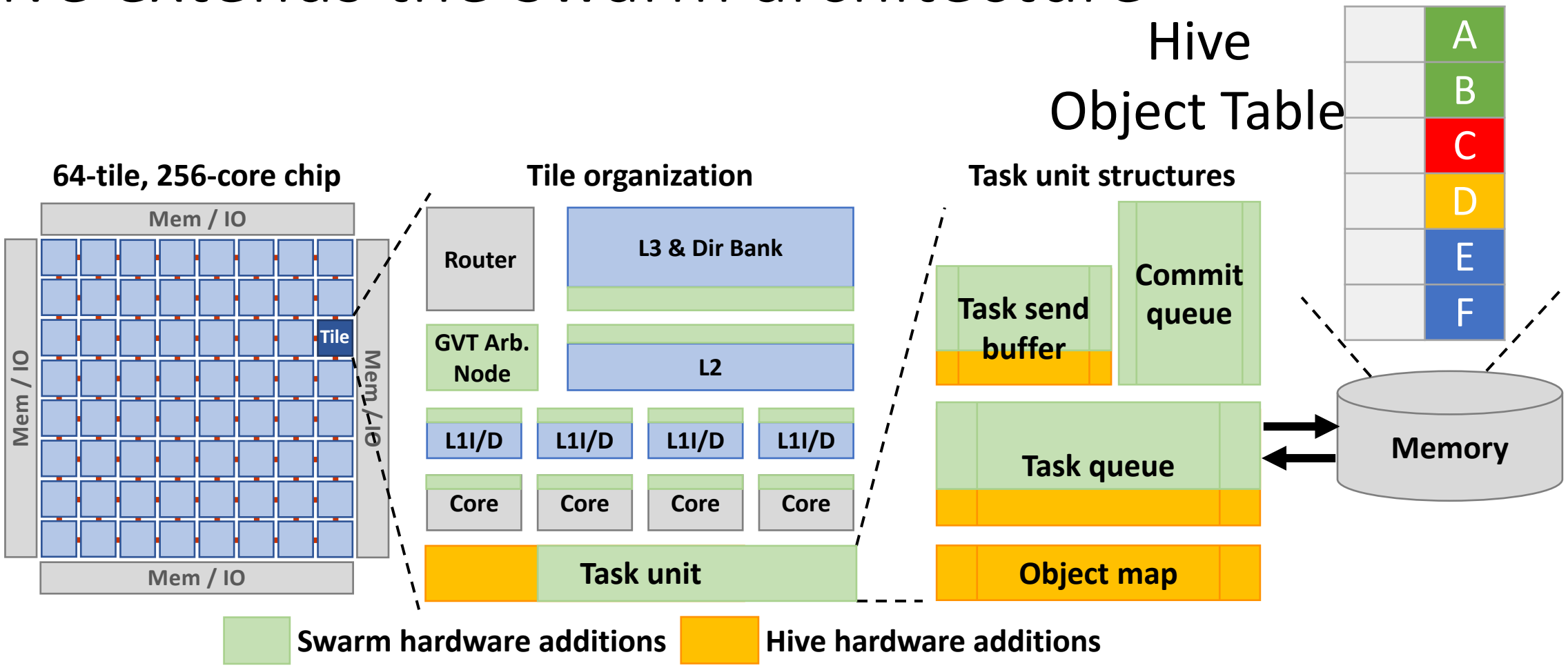
Hive extends the Swarm architecture



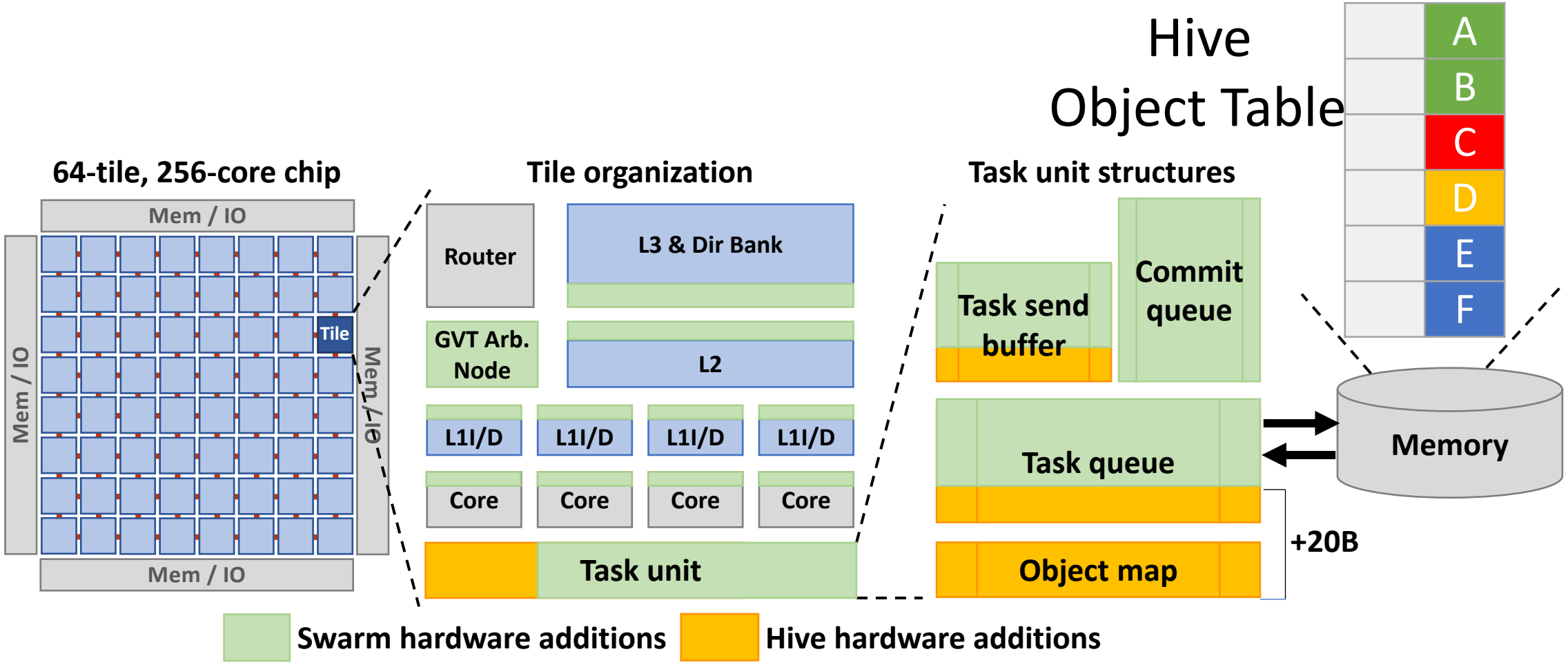
Hive extends the Swarm architecture



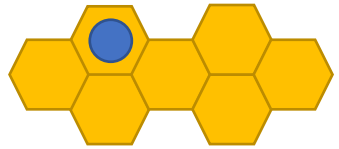
Hive extends the Swarm architecture



Hive extends the Swarm architecture



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

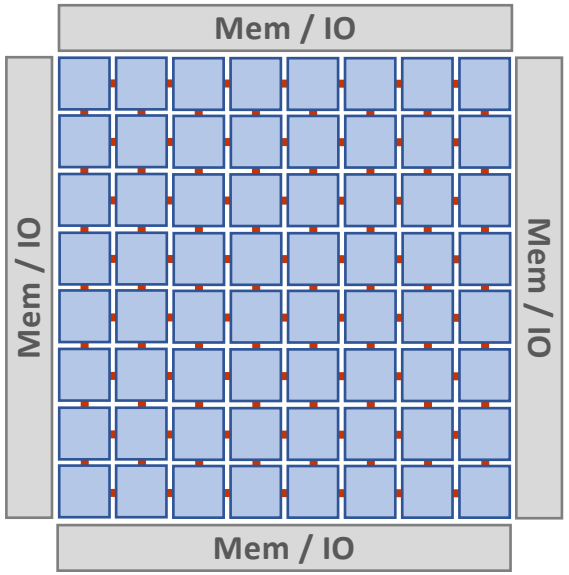


Evaluation

Methodology

Event-driven, Pin-based Simulator[1]

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

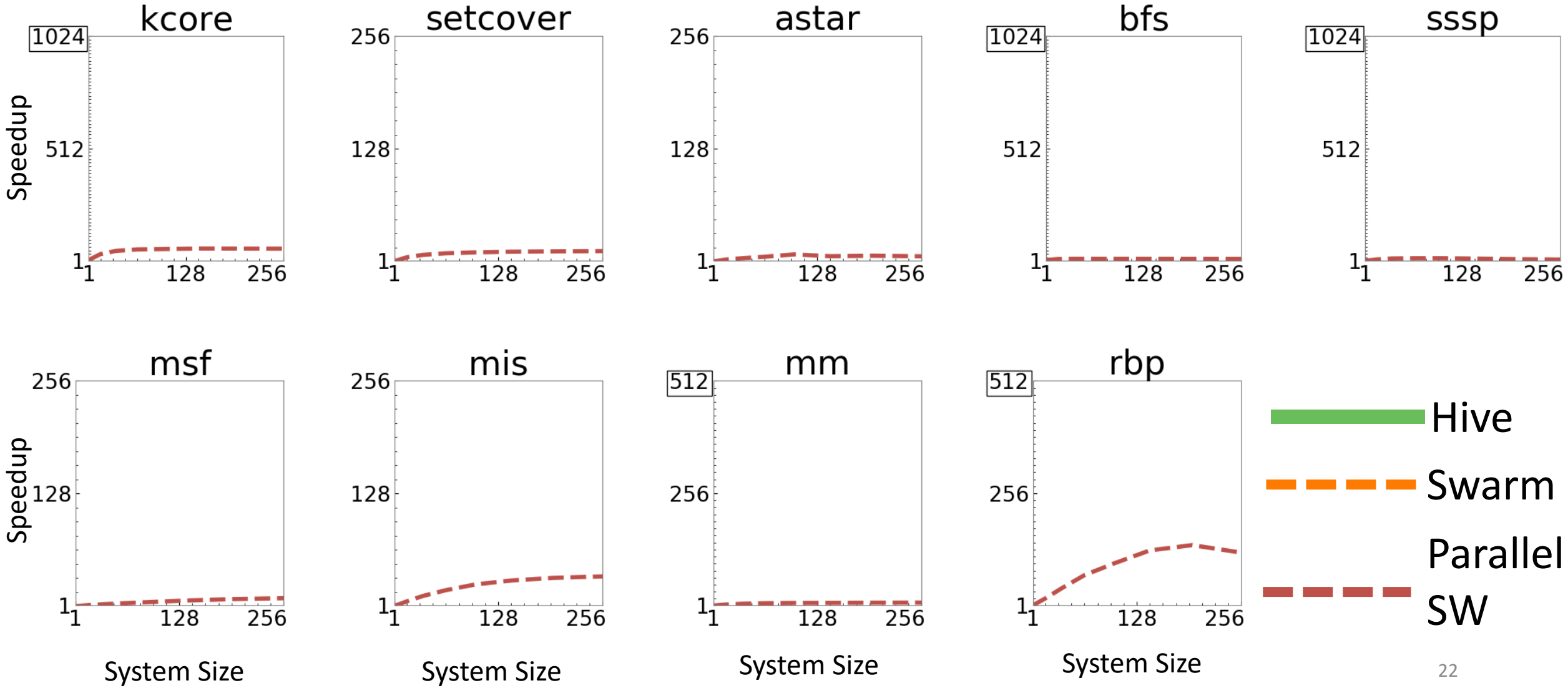
Scalability experiments up to 256 cores

- Smaller systems have fewer tiles

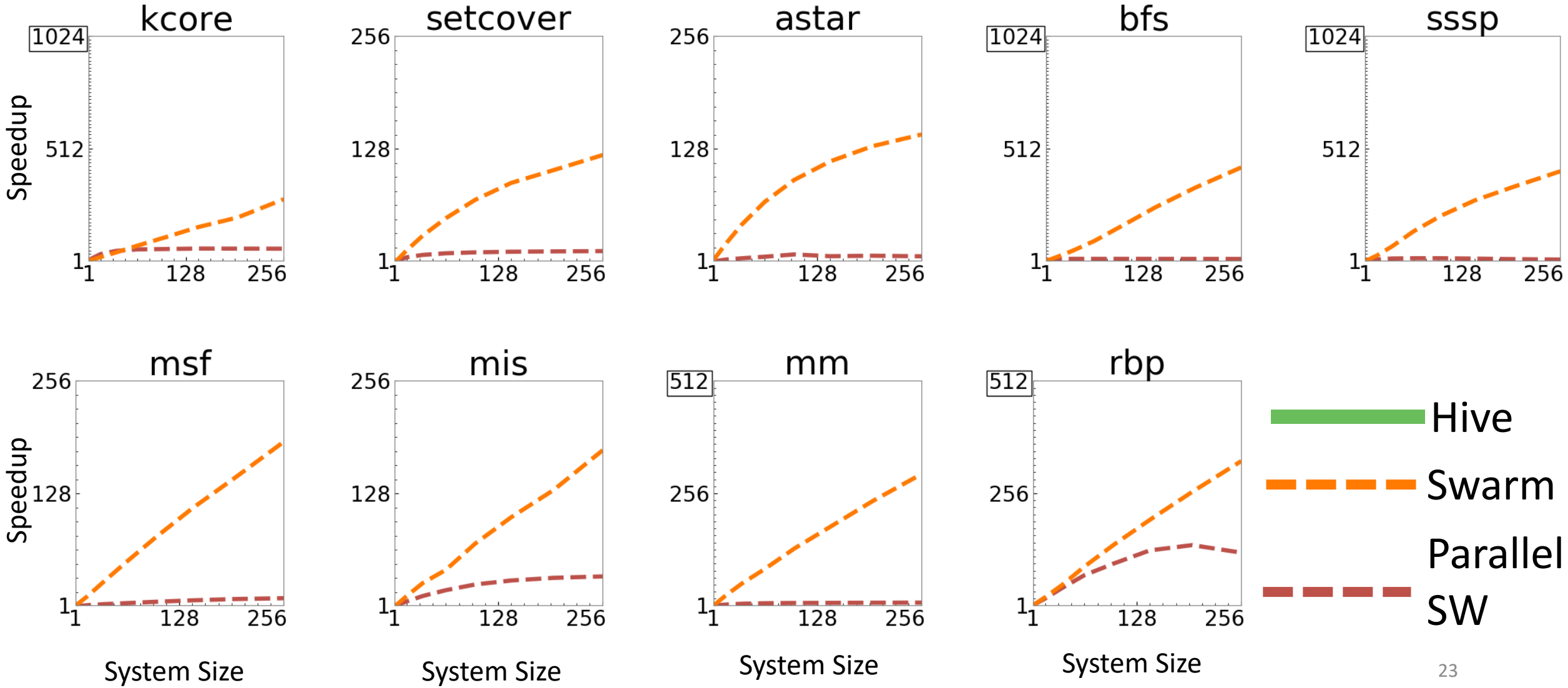
9 applications: KCore, Setcover, astar, BFS, SSSP, MSF, MIS, MM, RBP

1: https://github.com/SwarmArch/sim

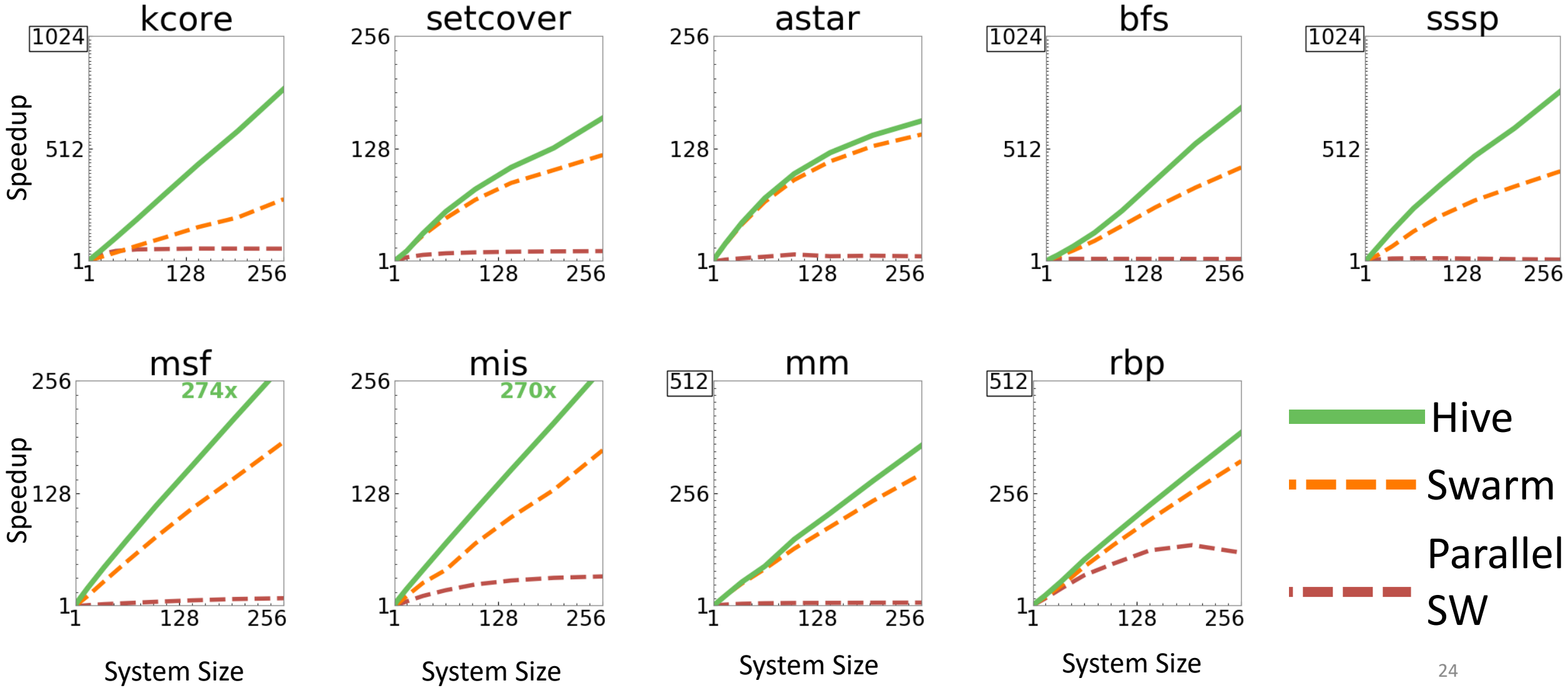
Software struggles to scale beyond 100c



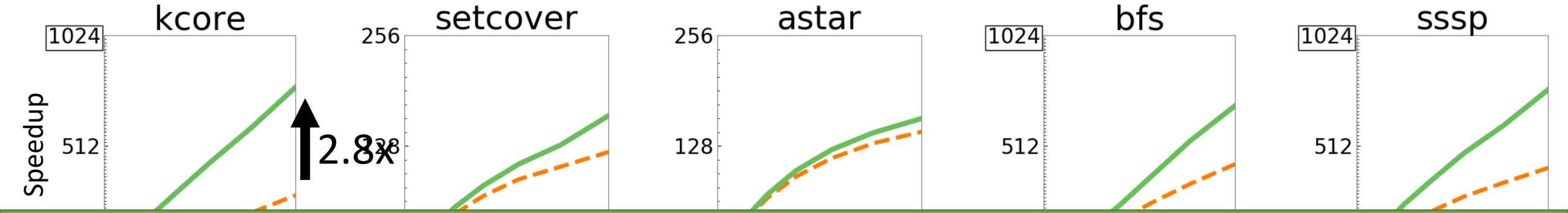
Swarm scales well sometimes



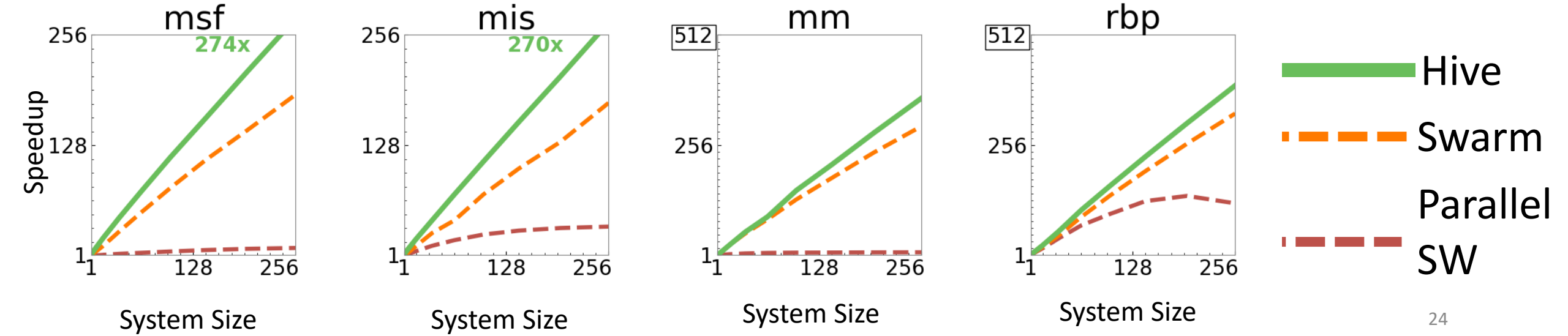
Hive is faster than Swarm



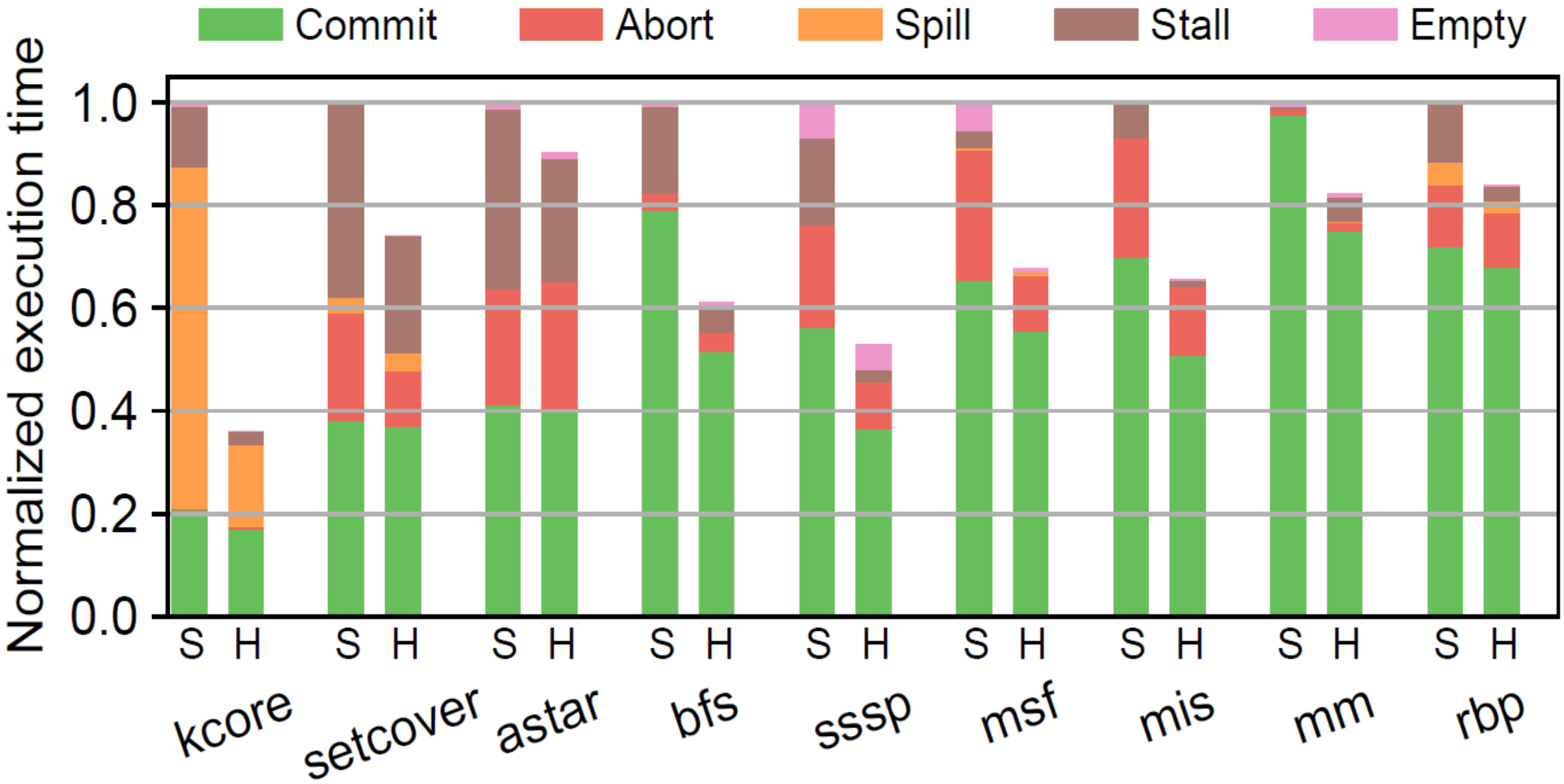
Hive is faster than Swarm



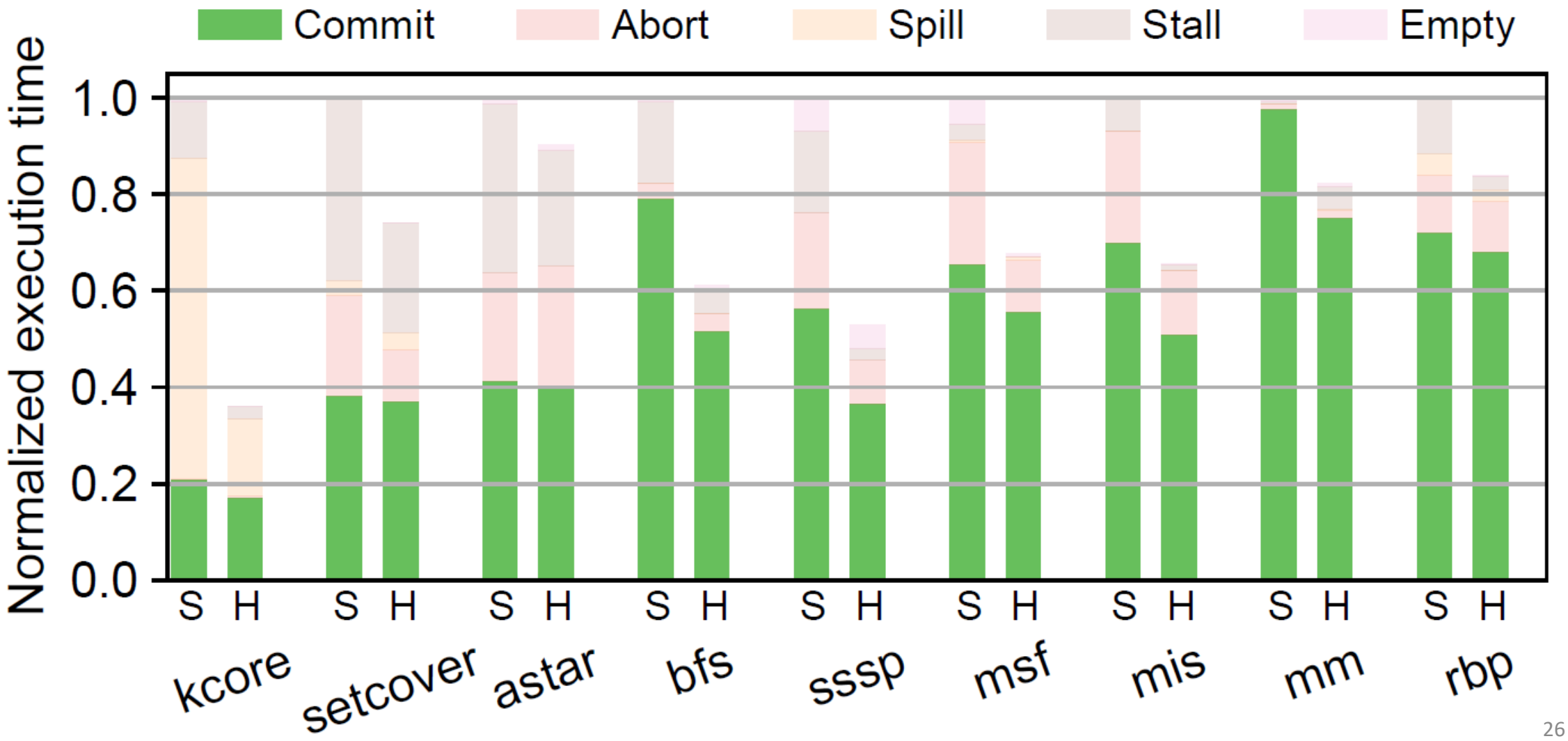
Hive is up to 2.8x faster than Swarm



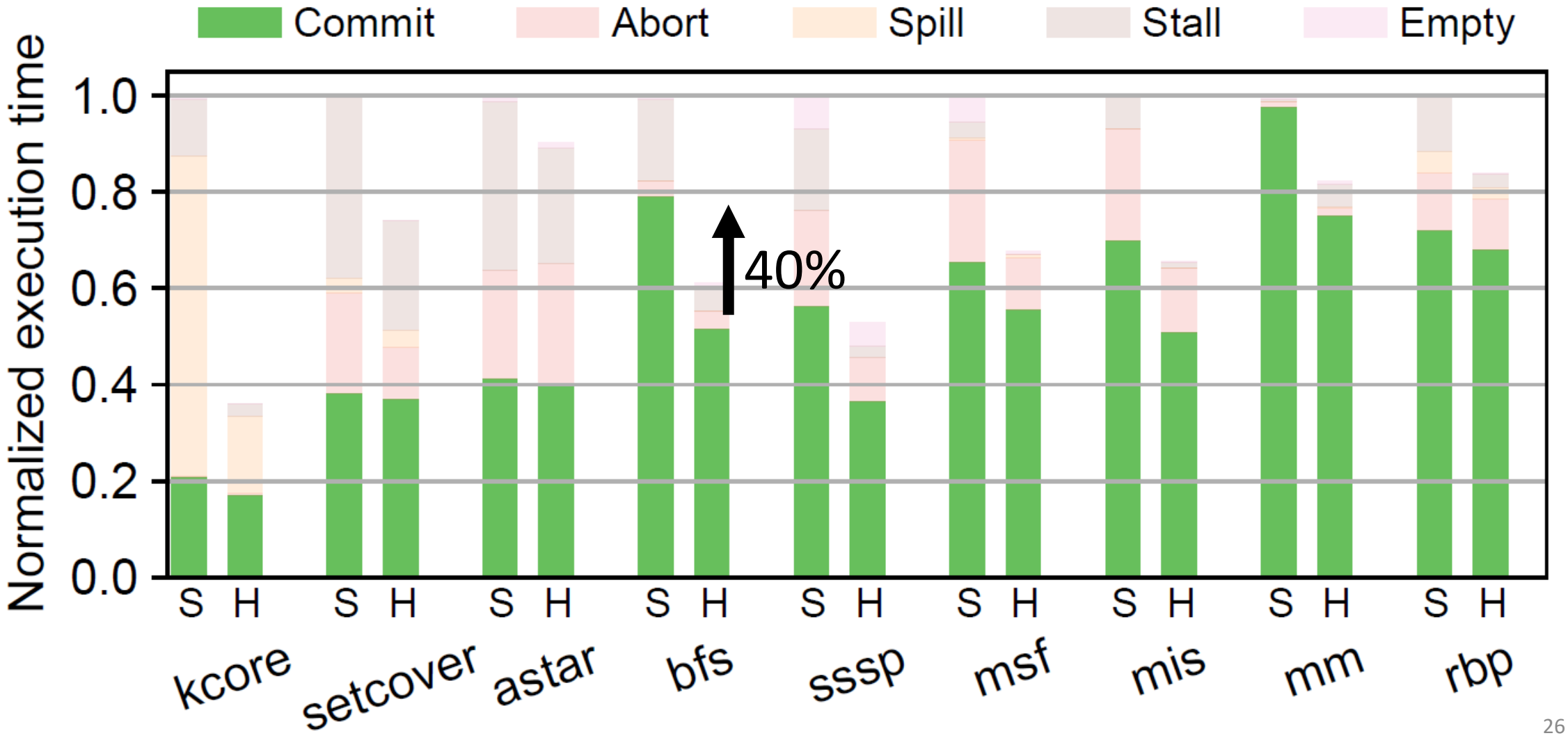
Breaking down Hive vs. Swarm at 256 cores



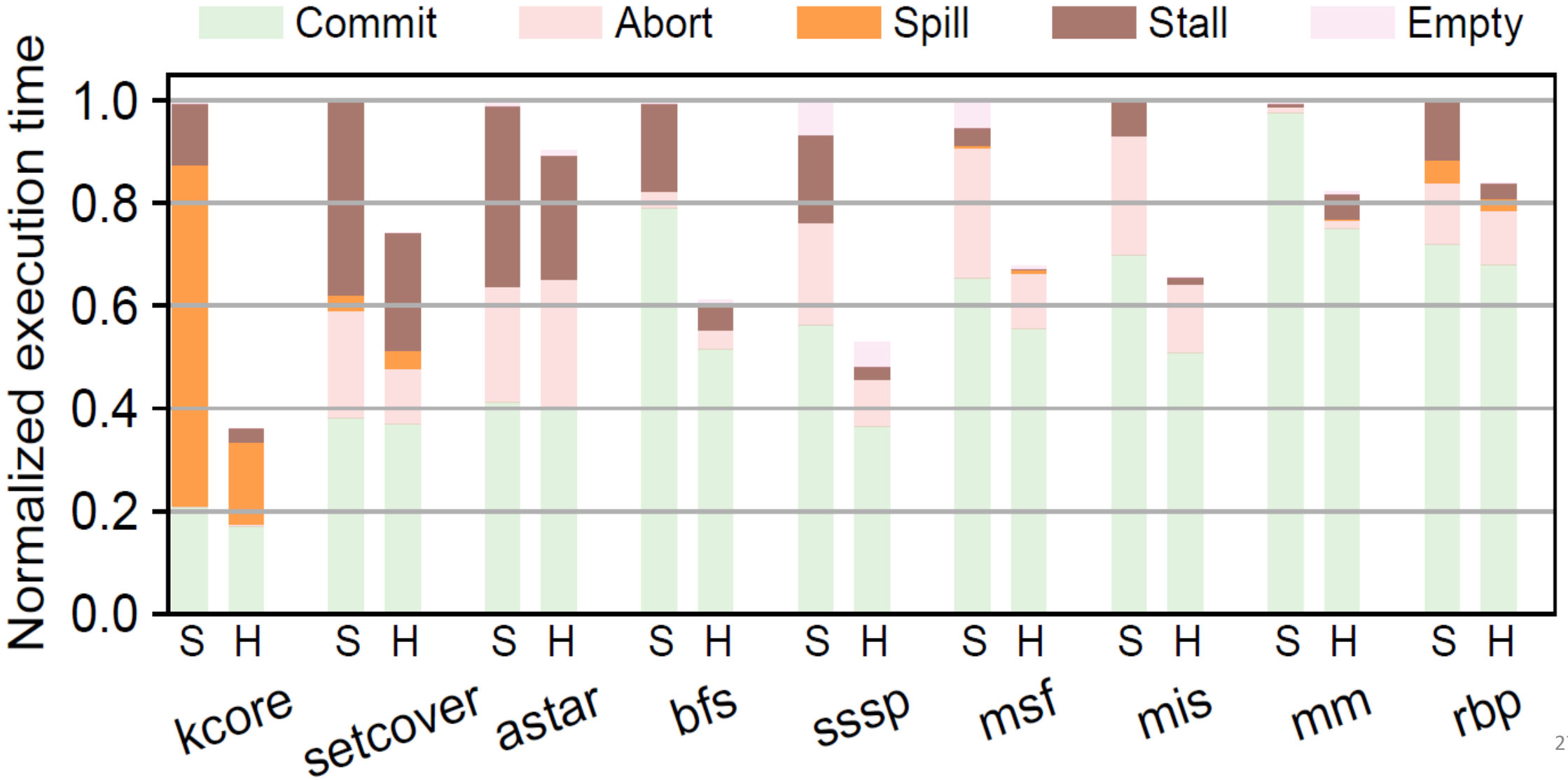
Hive does less work

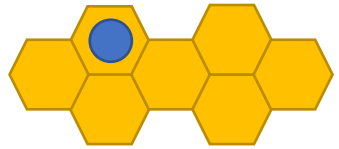


Hive does less work



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

UNIVERSITY OF TORONTO

HUAWEI