Harmonizing Speculative and Non-Speculative Execution in Architectures for Ordered Parallelism

MARK C. JEFFREY, VICTOR A. YING,
SUVINAY SUBRAMANIAN, HYUN RYONG LEE,
JOEL EMER, DANIEL SANCHEZ

MICRO 2018

There is a (false) dichotomy in parallelization

SPECULATIVE PARALLELIZATION	NON-SPECULATIVE PARALLELIZATION
Simplifies parallel programming	Lower overheads
Uncovers abundant parallelism	Parallel irrevocable actions

Current systems offer all-or-nothing speculation

Goal: Bring non-speculative execution to systems that support ordered parallelism

Espresso

- Expressive task-based execution model
- Coordinates concurrent speculative and non-speculative ordered tasks
- 256-core speedups up to **2.5x** vs. all-speculative

Capsules

- Let speculative tasks safely invoke software-managed speculation
- Enable important system services:
  e.g. memory allocator that improves performance up to **69x**

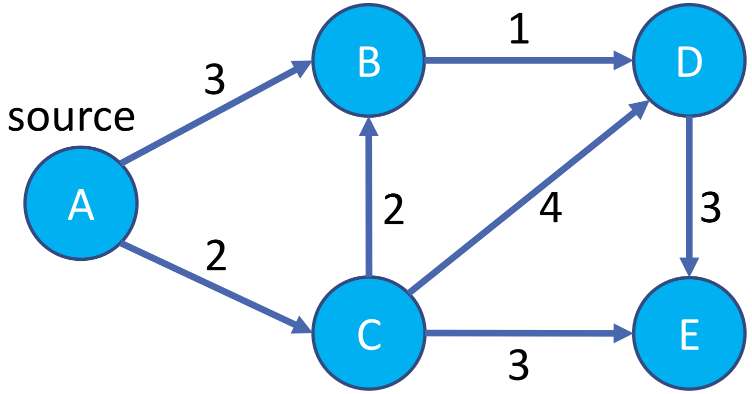
Espresso in action

THE NEED FOR SPECULATIVE AND NON-SPECULATIVE PARALLELISM

Example: Dijkstra’s algorithm

Finds shortest path tree on a graph with weighted edges

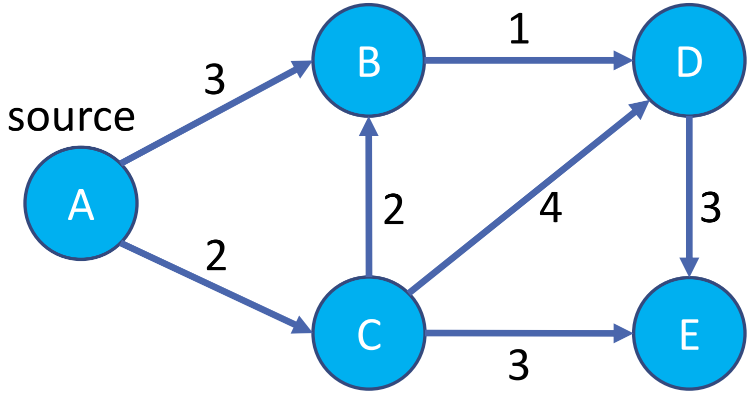
Input graph



Example: Dijkstra's algorithm

Finds shortest path tree on a graph with weighted edges

Input graph

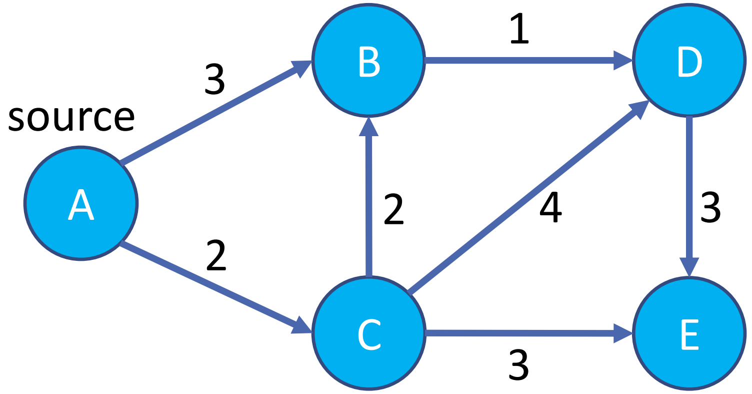


Task graph

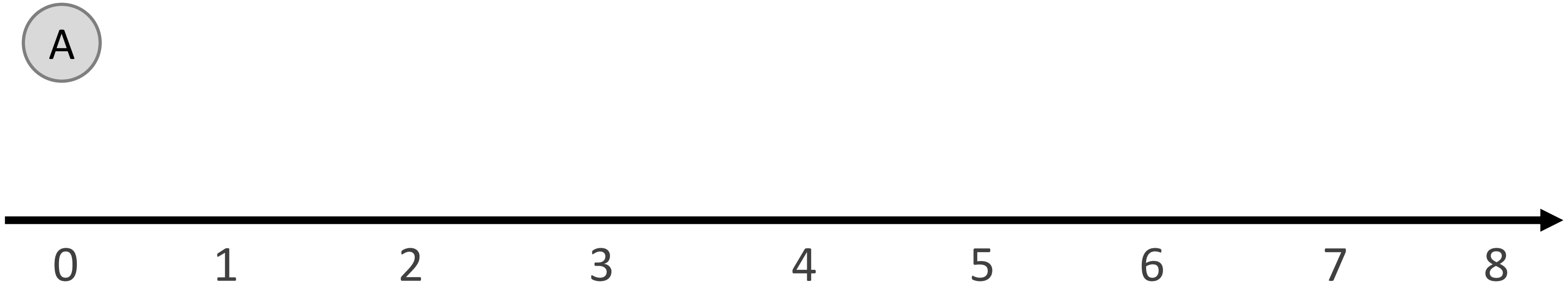
Example: Dijkstra's algorithm

Finds shortest path tree on a graph with weighted edges

Input graph



Task graph

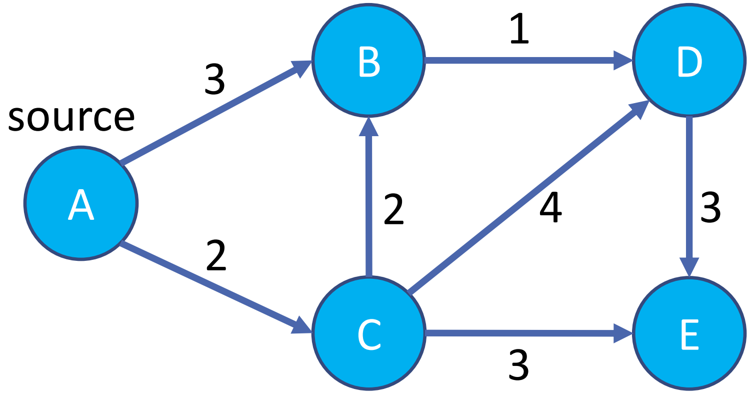


Order = Distance from source node

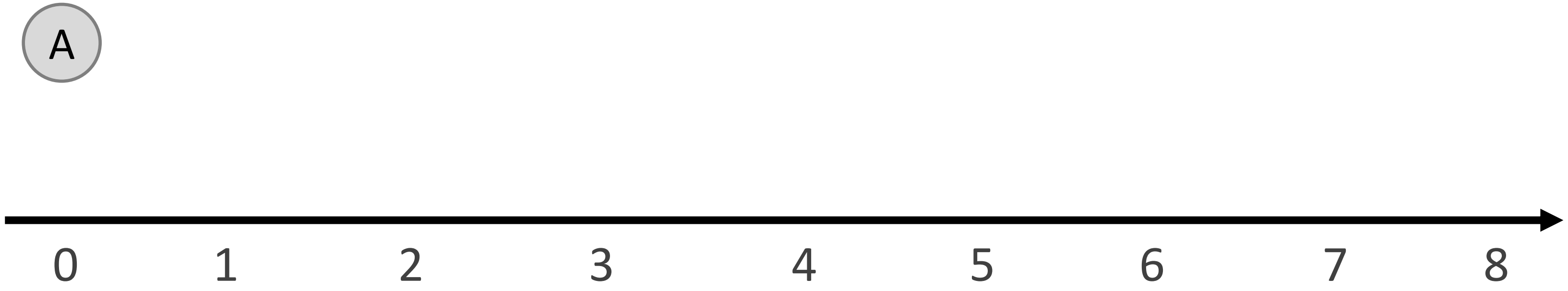
Example: Dijkstra's algorithm

Finds shortest path tree on a graph with weighted edges

Input graph



Task graph



To be processed

First to visit vertex

Vertex already visited

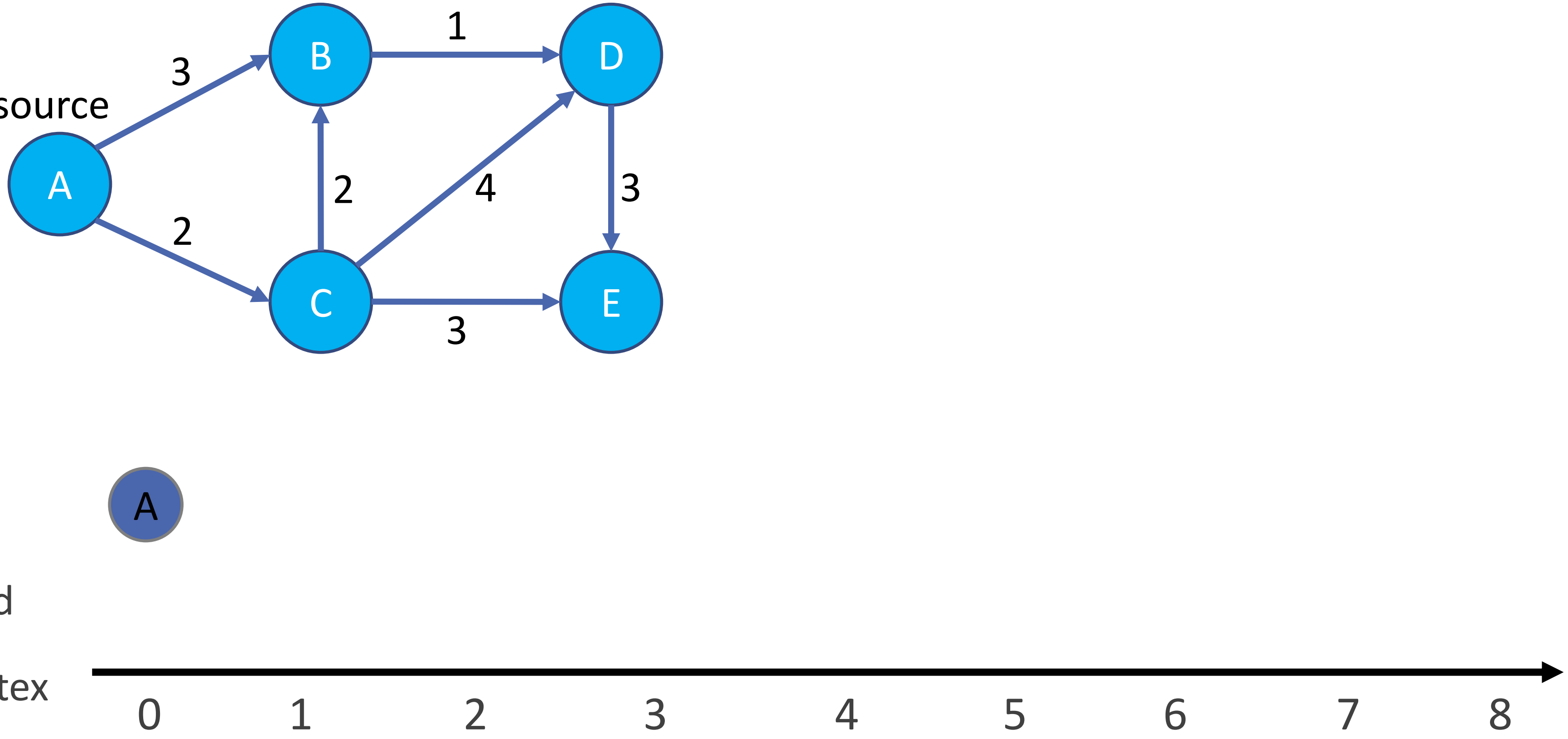
Order = Distance from source node

Example: Dijkstra's algorithm

Finds shortest path tree on a graph with weighted edges

Input graph

Task graph



To be processed

First to visit vertex

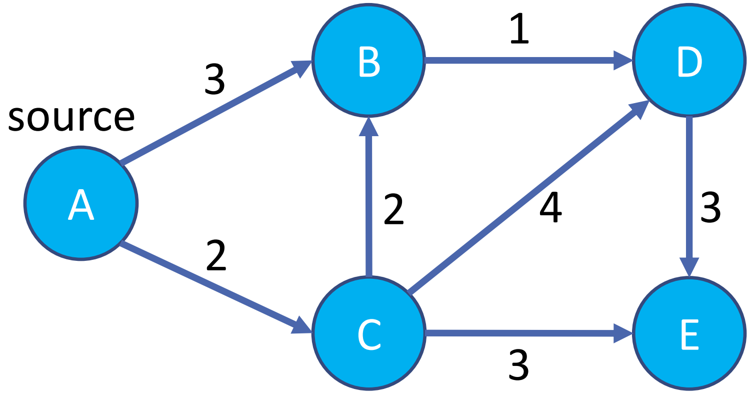
Vertex already visited

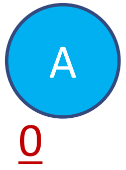
Order = Distance from source node

Example: Dijkstra's algorithm

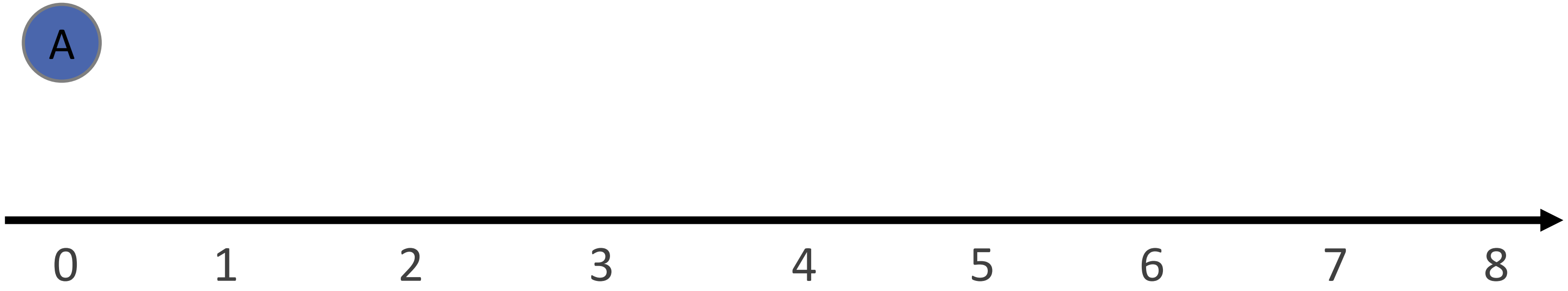
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



To be processed

First to visit vertex

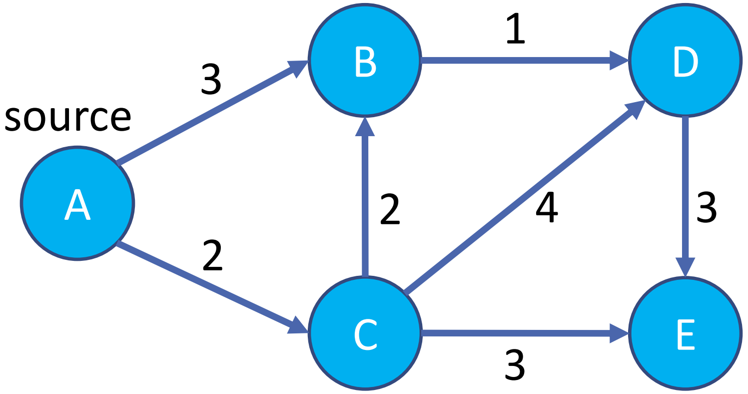
Vertex already visited

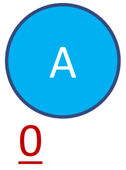
Order = Distance from source node

Example: Dijkstra's algorithm

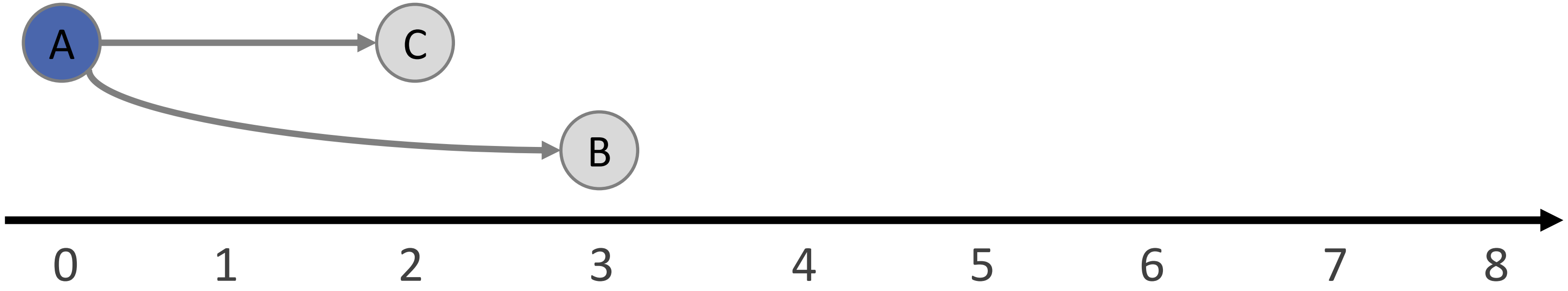
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



Order = Distance from source node

To be processed

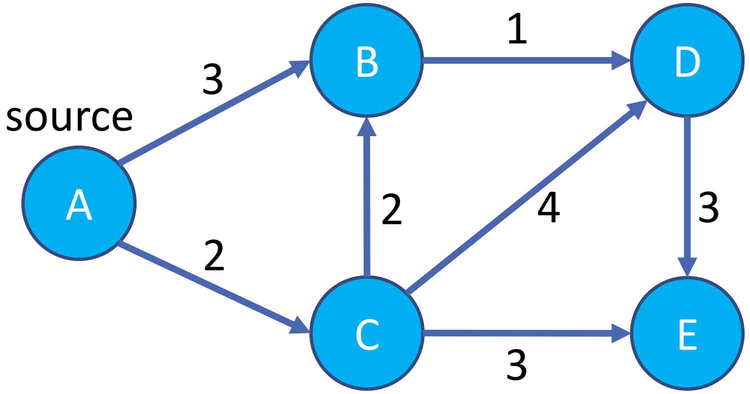
First to visit vertex

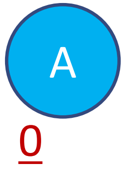
Vertex already visited

Example: Dijkstra's algorithm

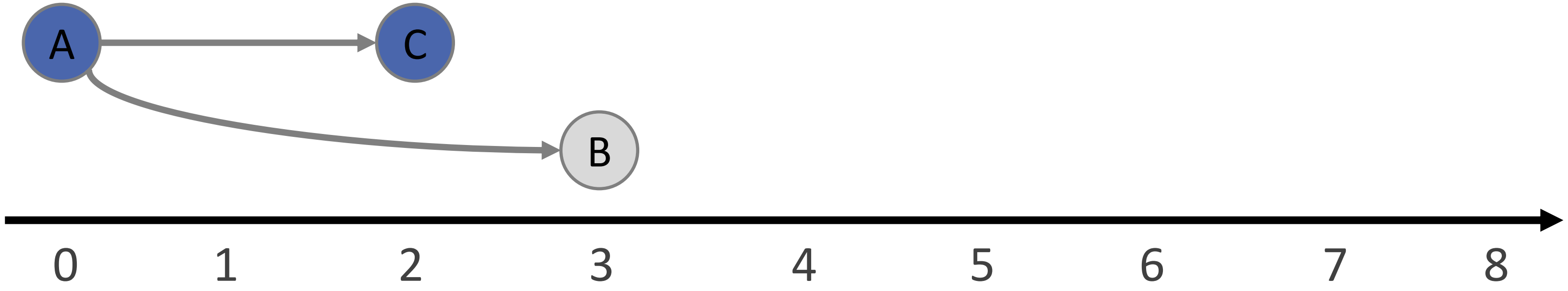
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



Order = Distance from source node

To be processed

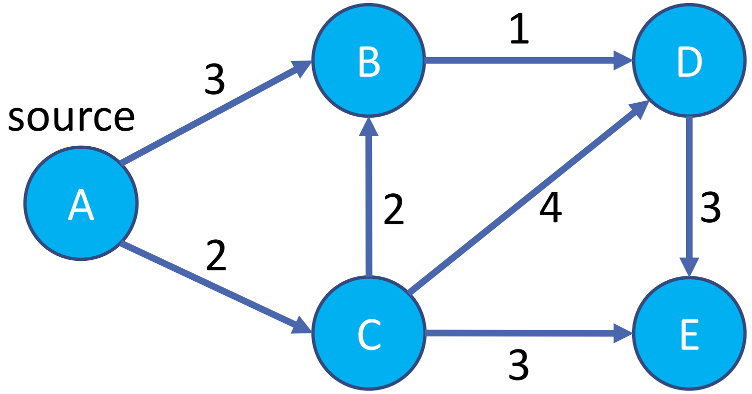
First to visit vertex

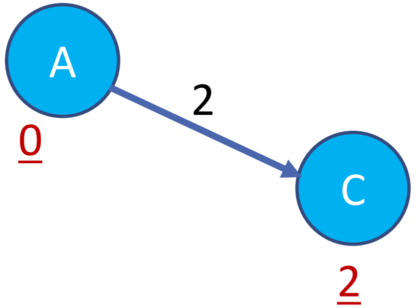
Vertex already visited

Example: Dijkstra's algorithm

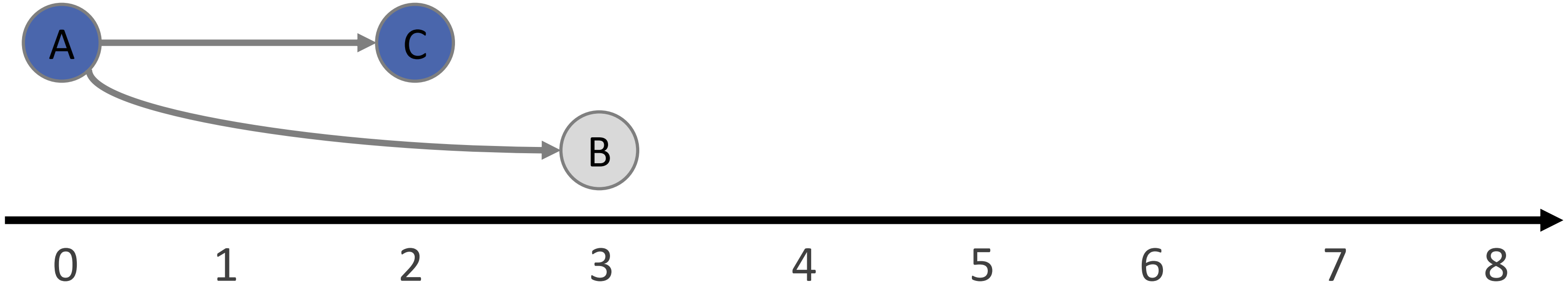
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



To be processed

First to visit vertex

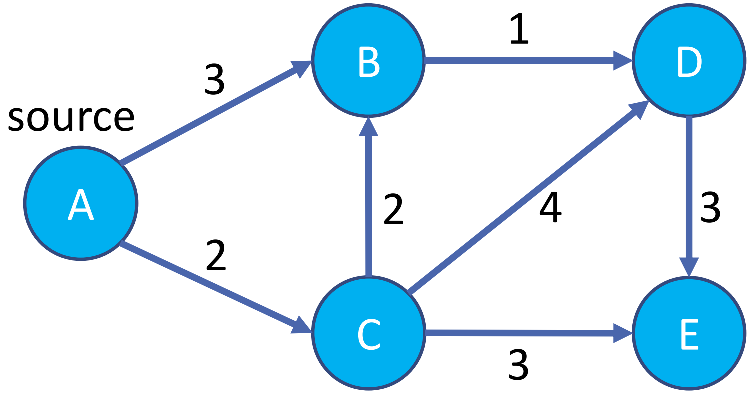
Vertex already visited

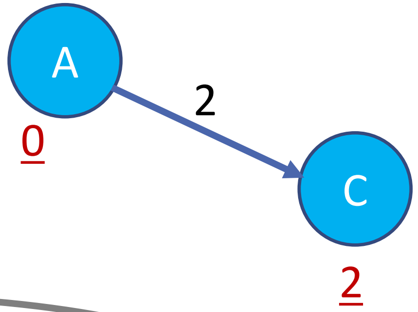
Order = Distance from source node

Example: Dijkstra’s algorithm

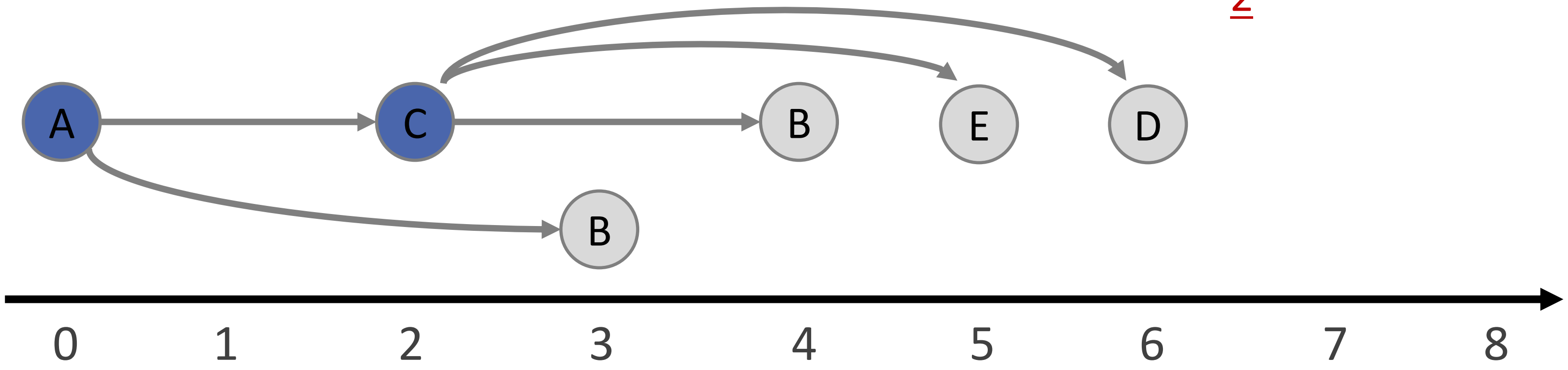
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



To be processed

First to visit vertex

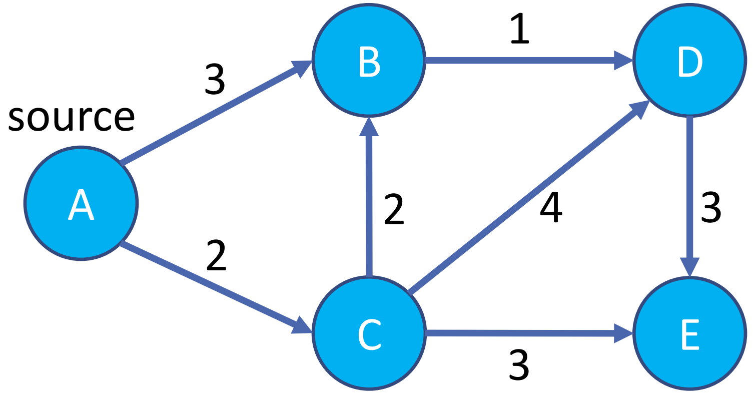
Vertex already visited

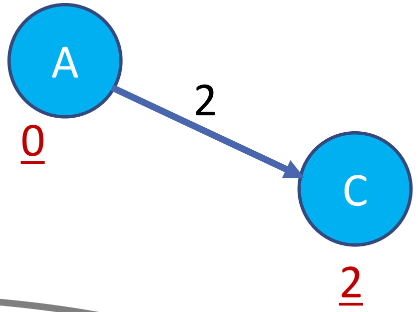
Order = Distance from source node

Example: Dijkstra's algorithm

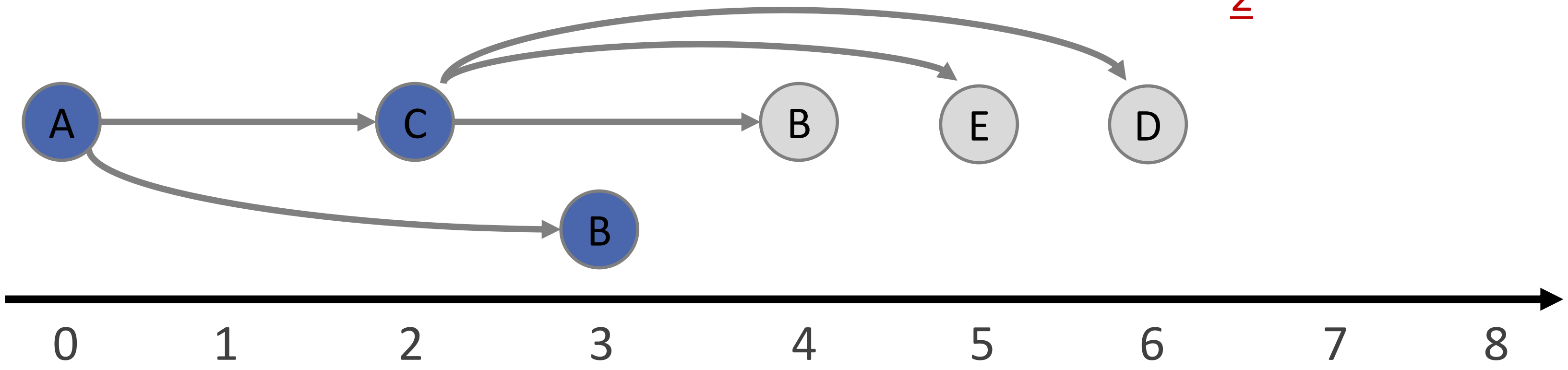
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



To be processed

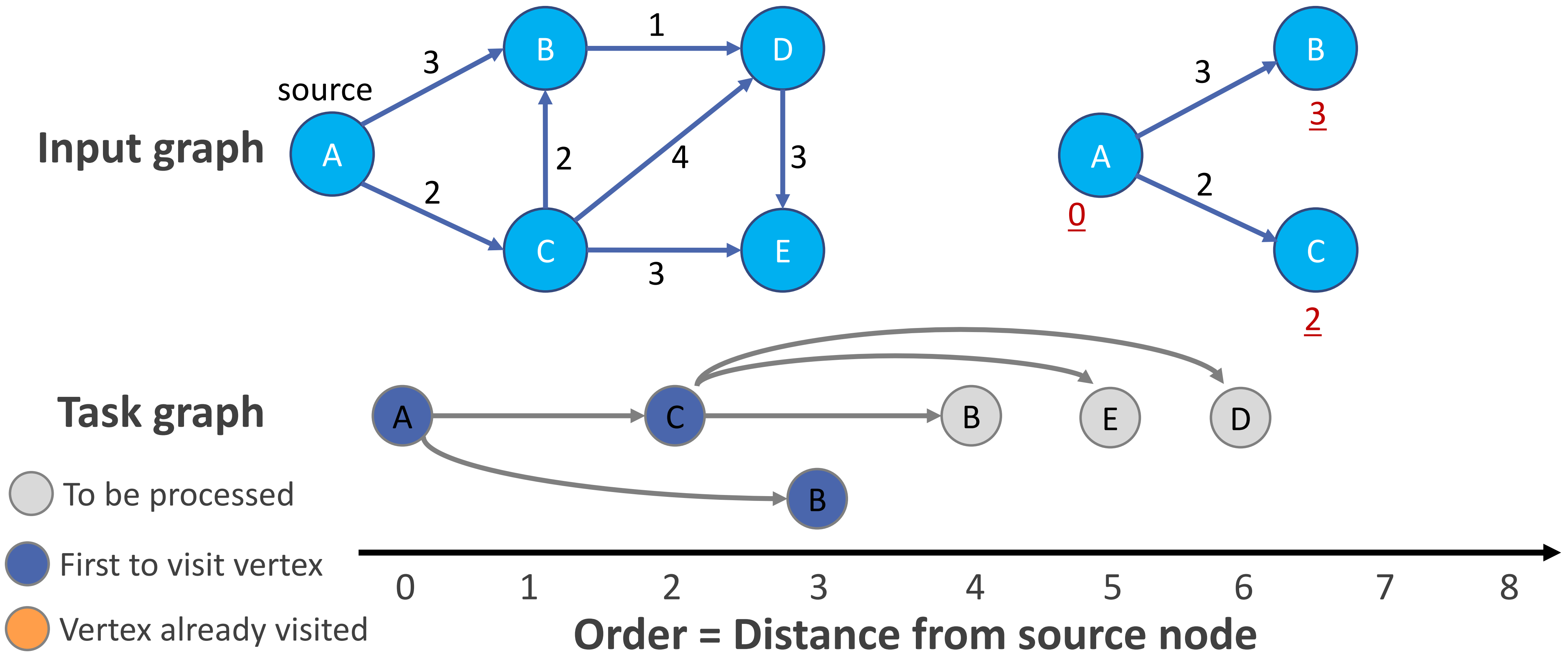
First to visit vertex

Vertex already visited

Order = Distance from source node

Example: Dijkstra's algorithm

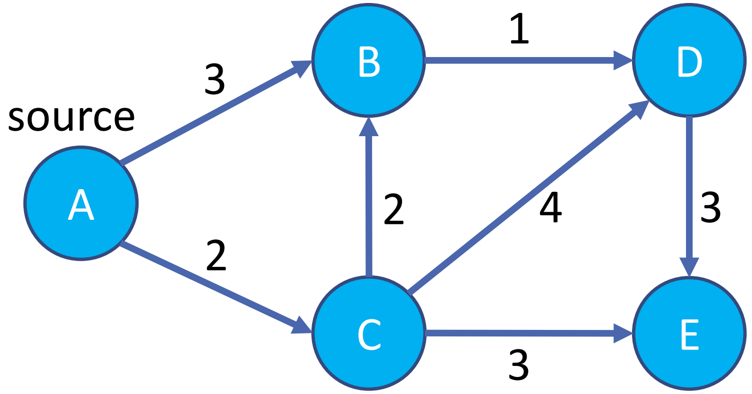
Finds shortest path tree on a graph with weighted edges

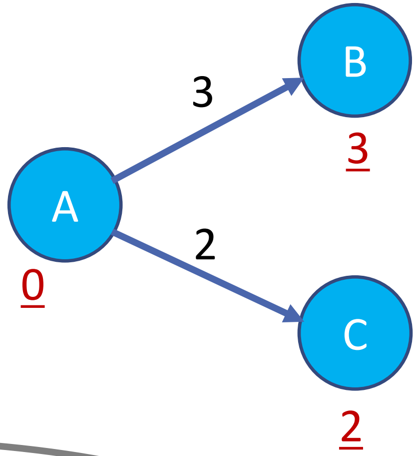


Example: Dijkstra's algorithm

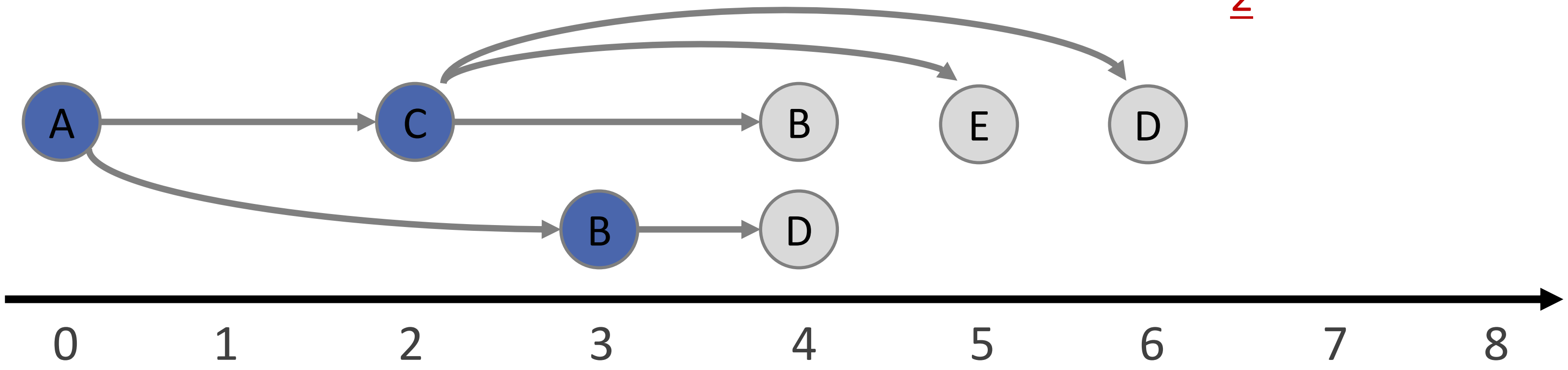
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



Order = Distance from source node

To be processed

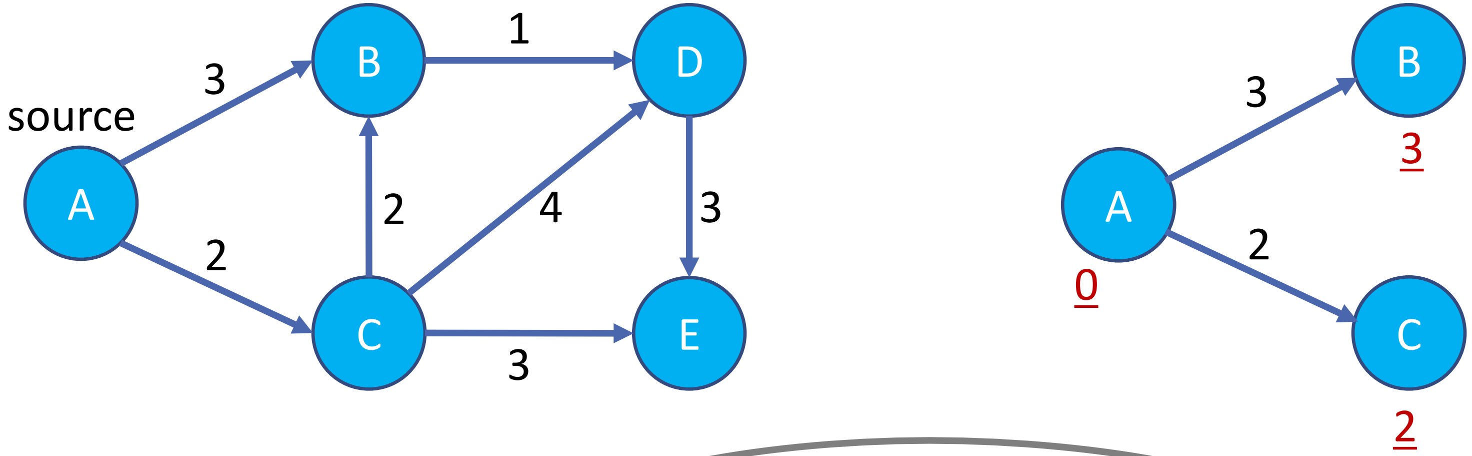
First to visit vertex

Vertex already visited

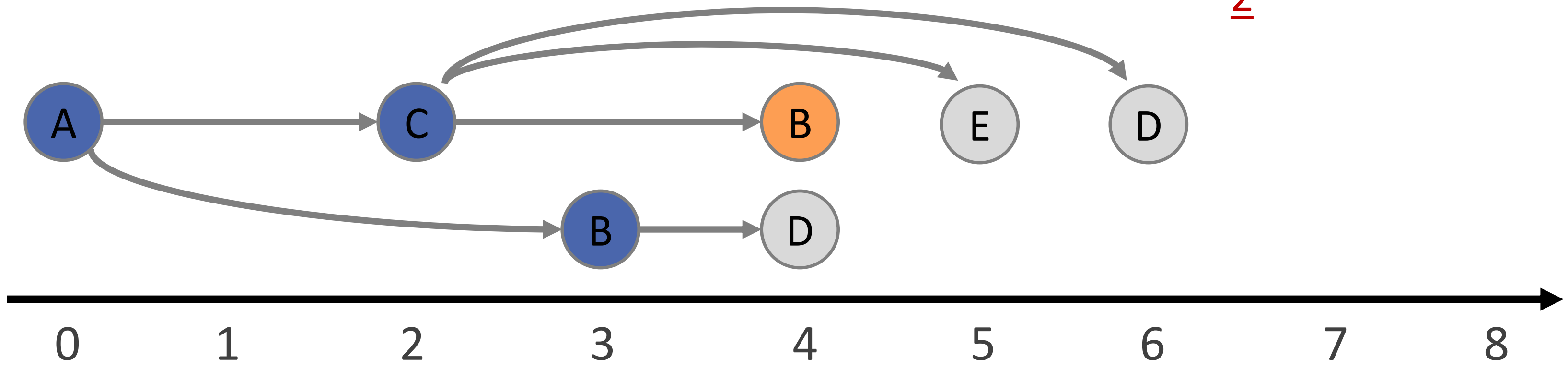
Example: Dijkstra's algorithm

Finds shortest path tree on a graph with weighted edges

Input graph



Task graph



To be processed

First to visit vertex

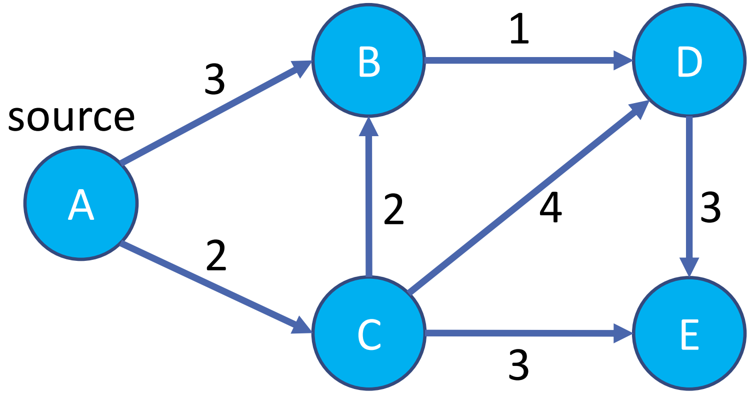
Vertex already visited

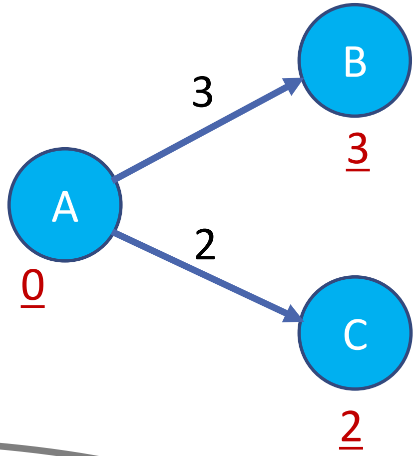
Order = Distance from source node

Example: Dijkstra's algorithm

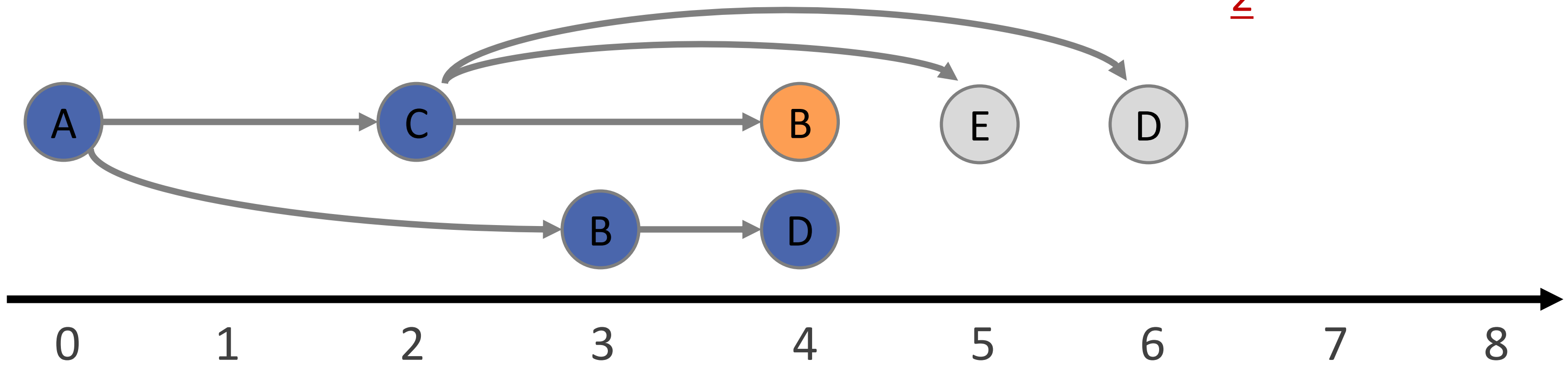
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



To be processed

First to visit vertex

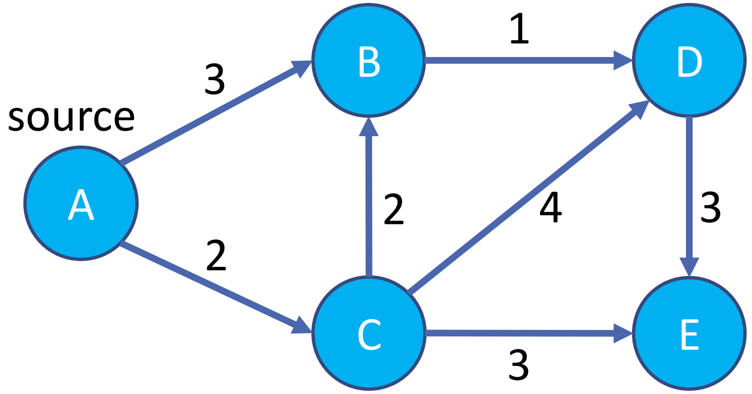
Vertex already visited

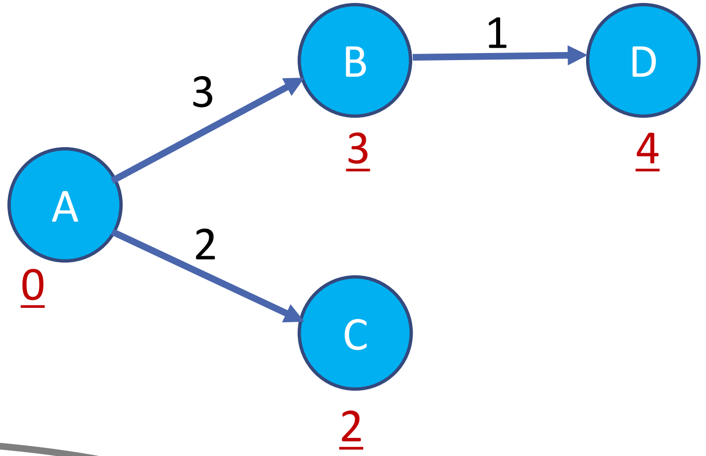
Order = Distance from source node

Example: Dijkstra's algorithm

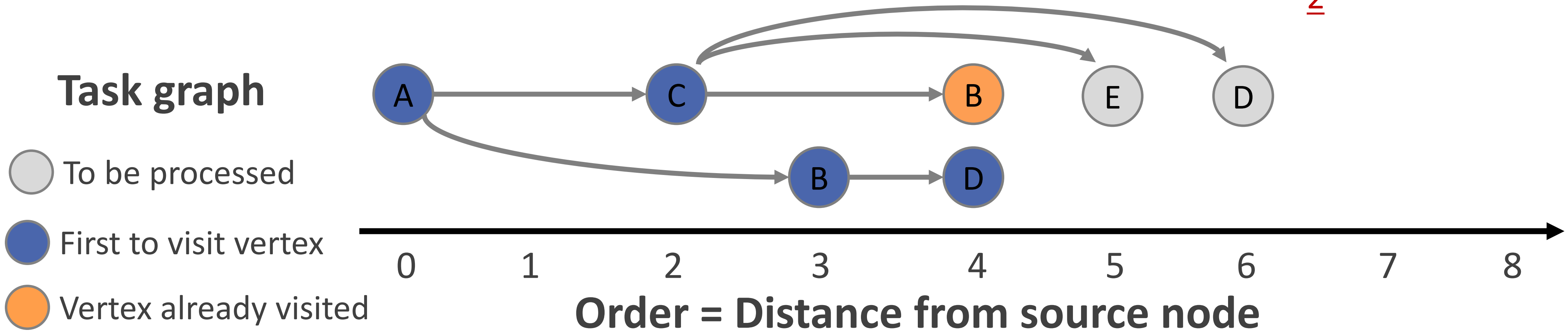
Finds shortest path tree on a graph with weighted edges

Input graph





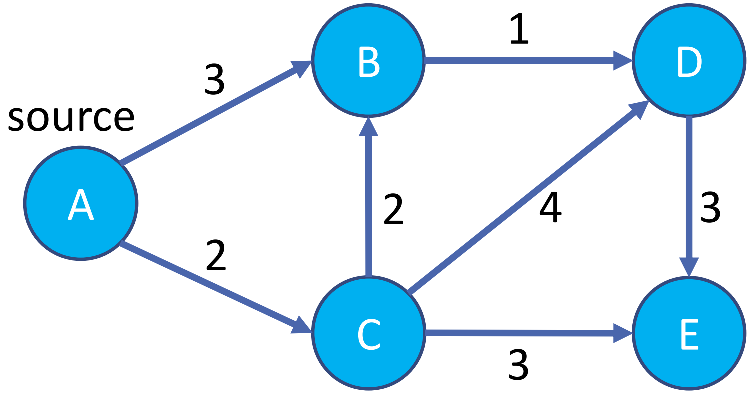
Task graph

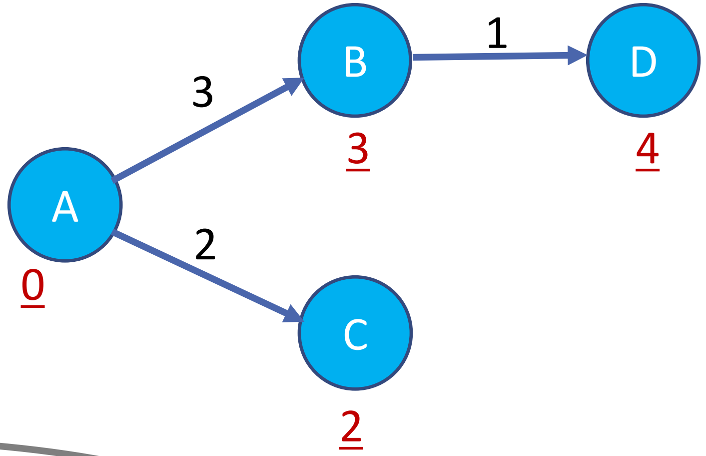


Example: Dijkstra's algorithm

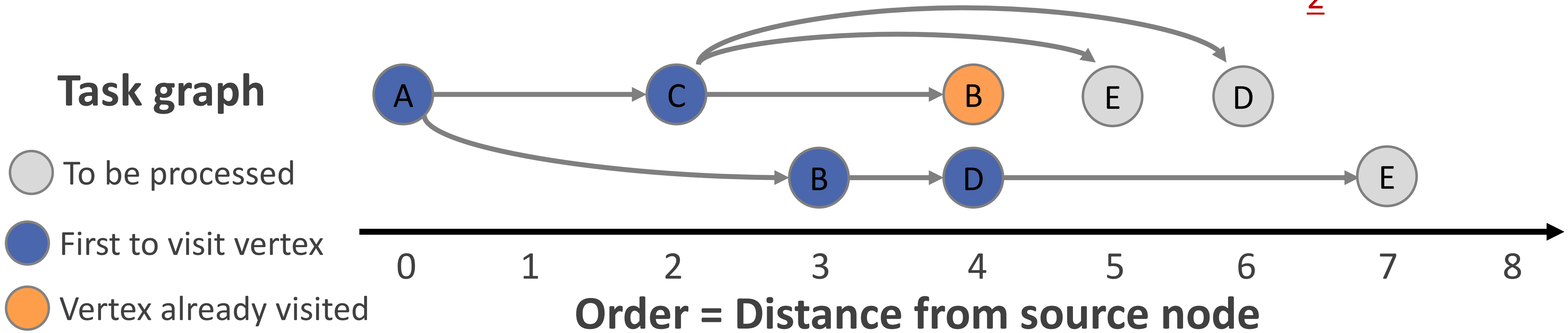
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph

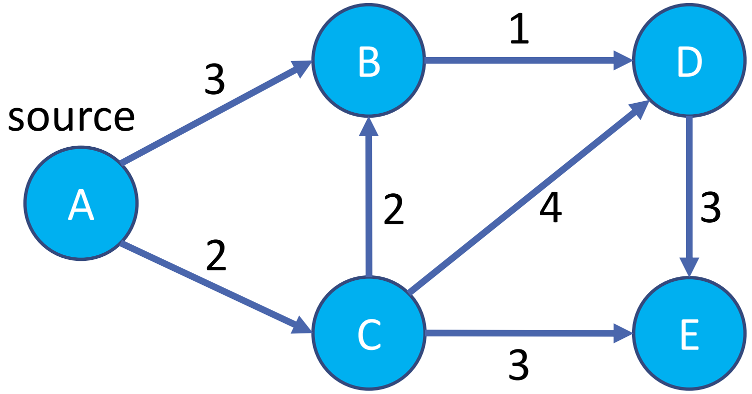


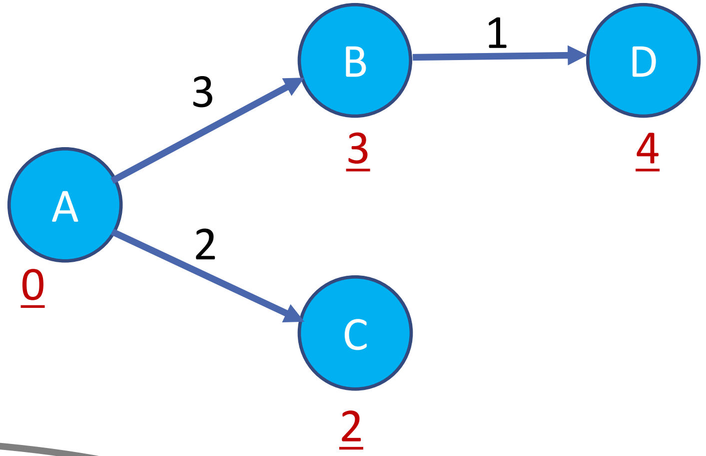
Order = Distance from source node

Example: Dijkstra's algorithm

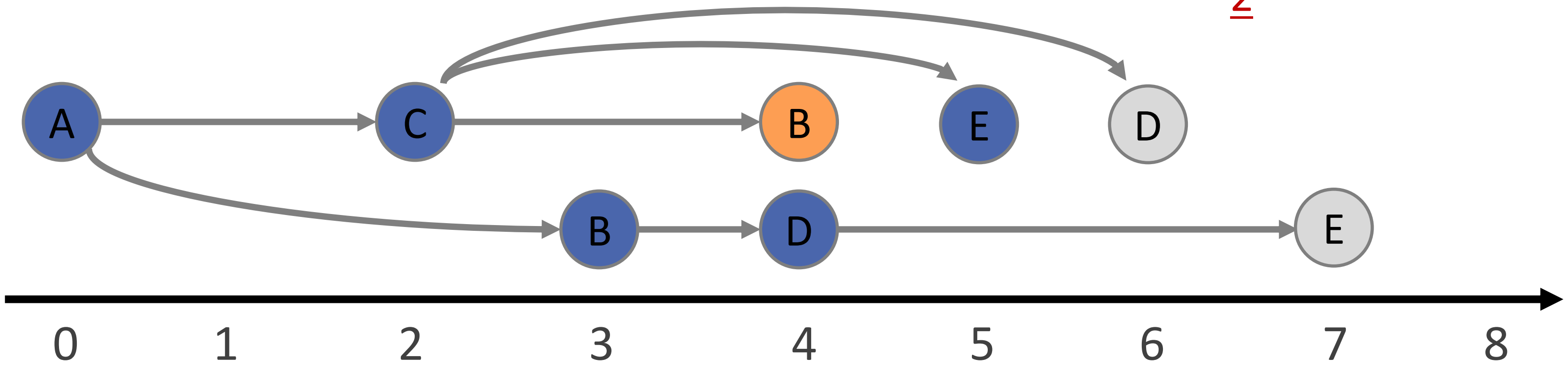
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



Order = Distance from source node

To be processed

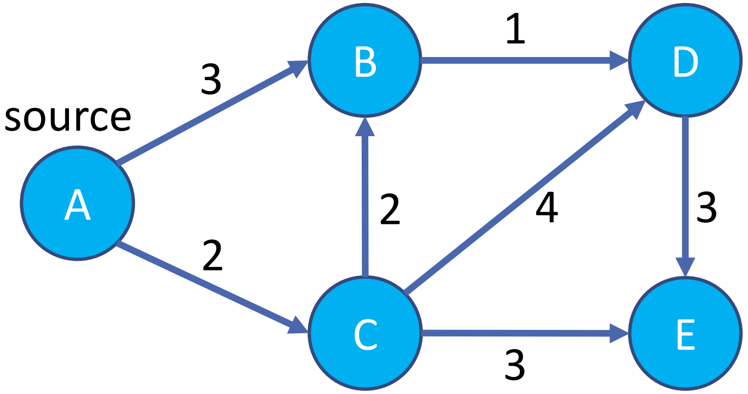
First to visit vertex

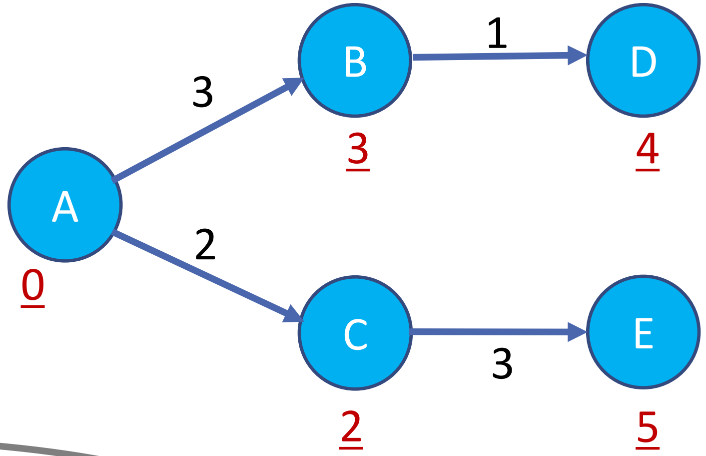
Vertex already visited

Example: Dijkstra’s algorithm

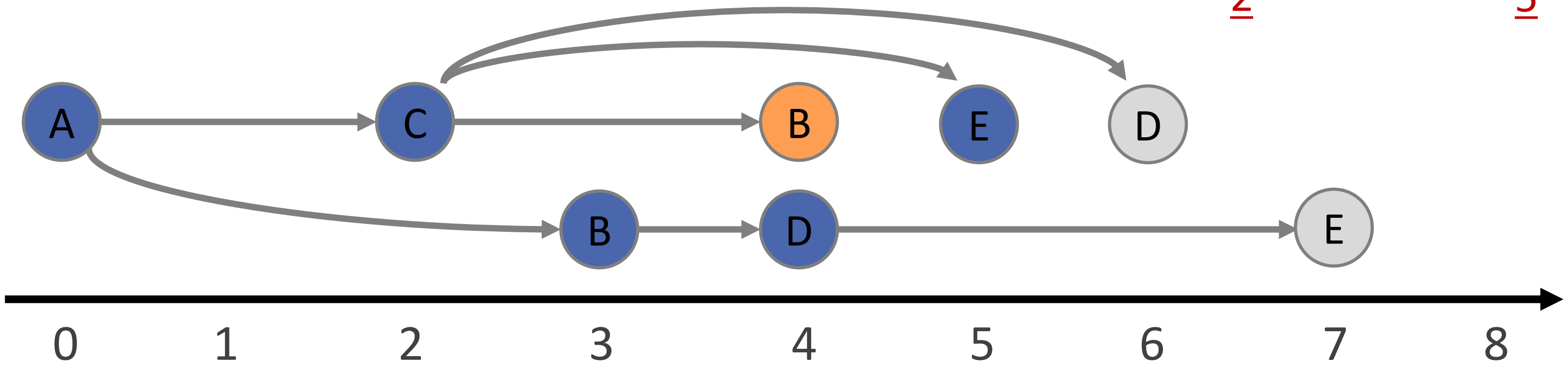
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



To be processed

First to visit vertex

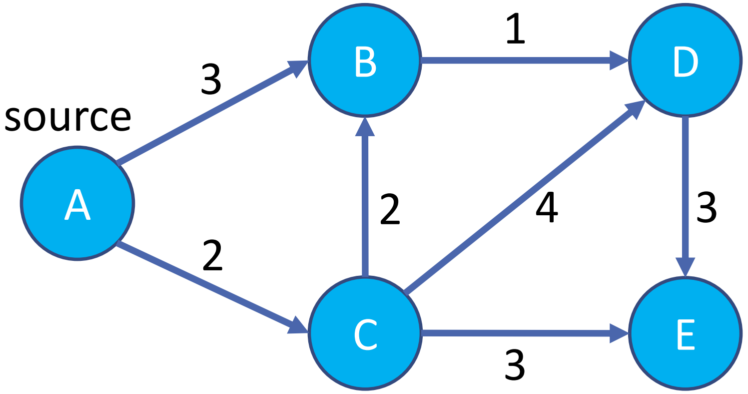
Vertex already visited

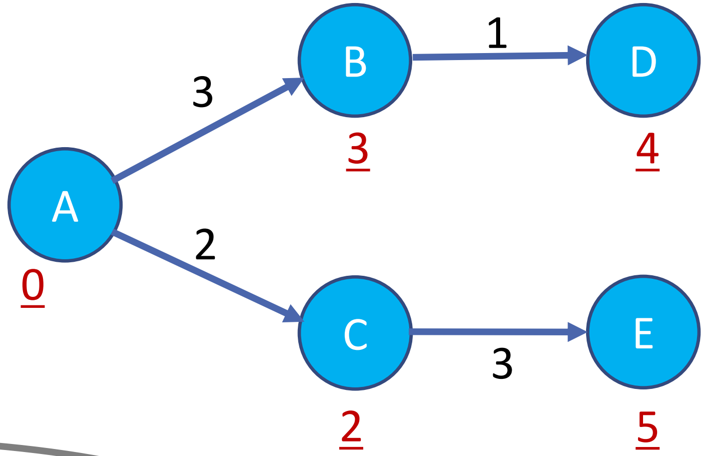
Order = Distance from source node

Example: Dijkstra’s algorithm

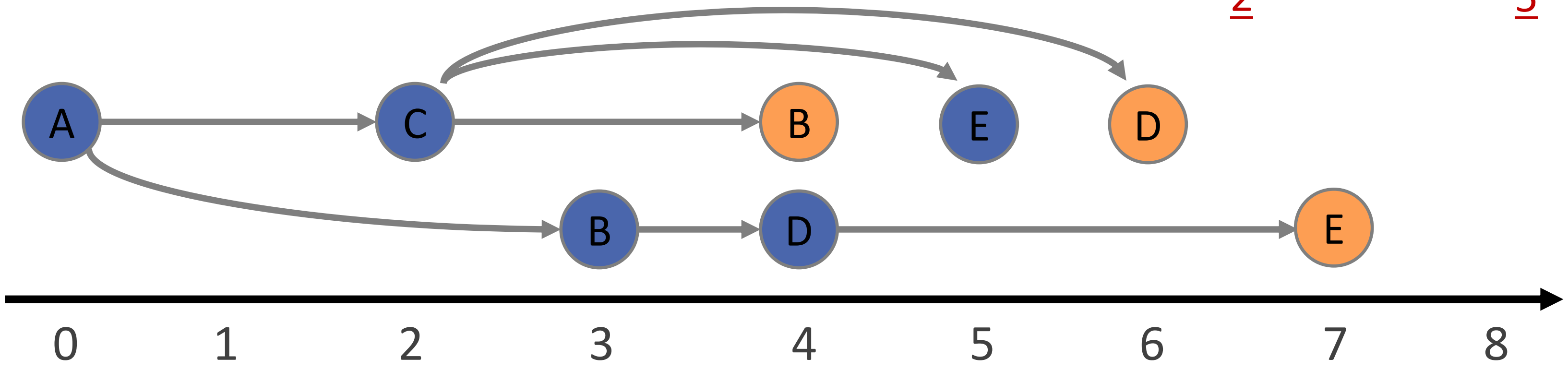
Finds shortest path tree on a graph with weighted edges

Input graph





Task graph



To be processed

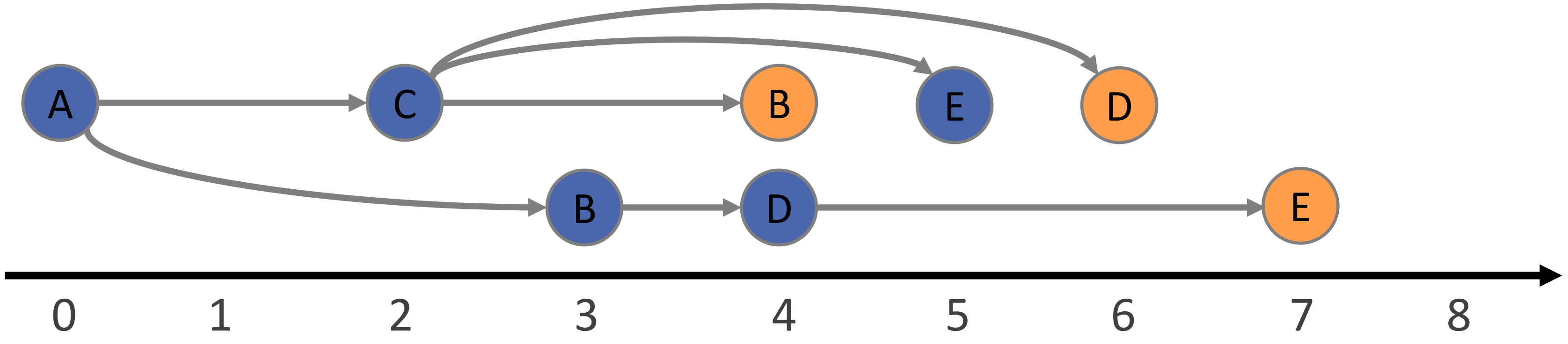
First to visit vertex

Vertex already visited

Order = Distance from source node

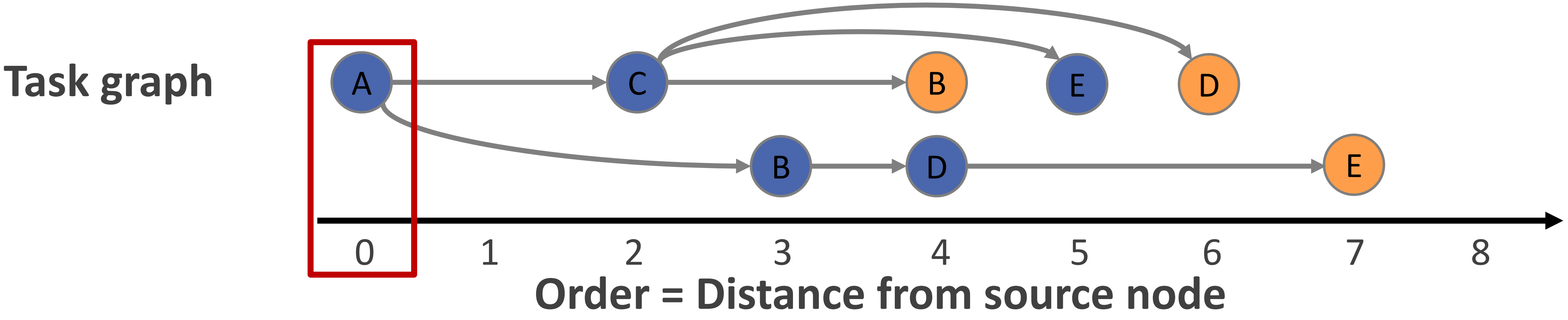
Parallelism in Dijkstra's algorithm?

Task graph



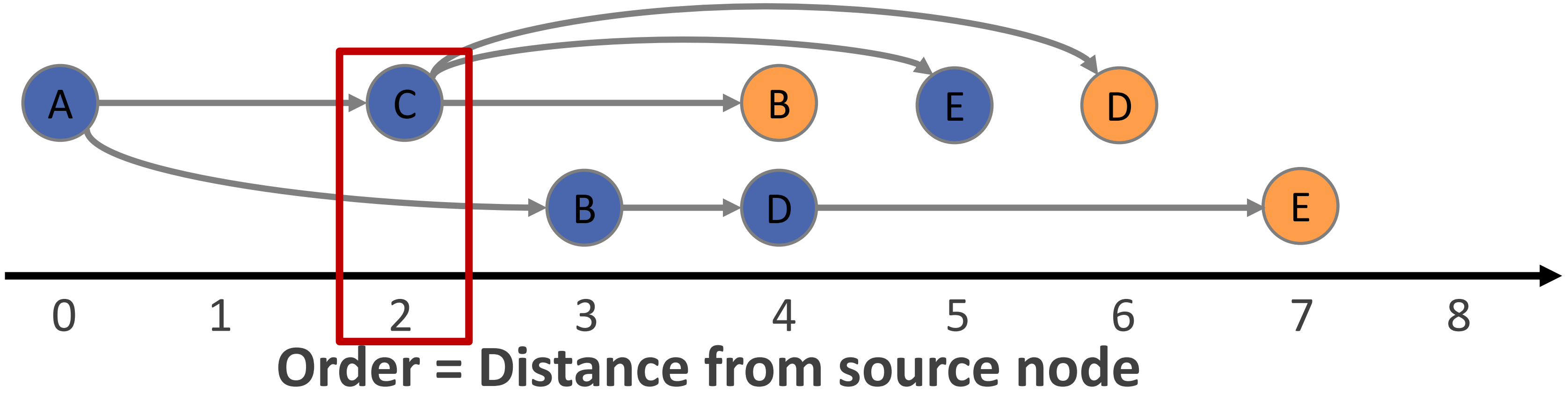
Order = Distance from source node

Parallelism in Dijkstra's algorithm?



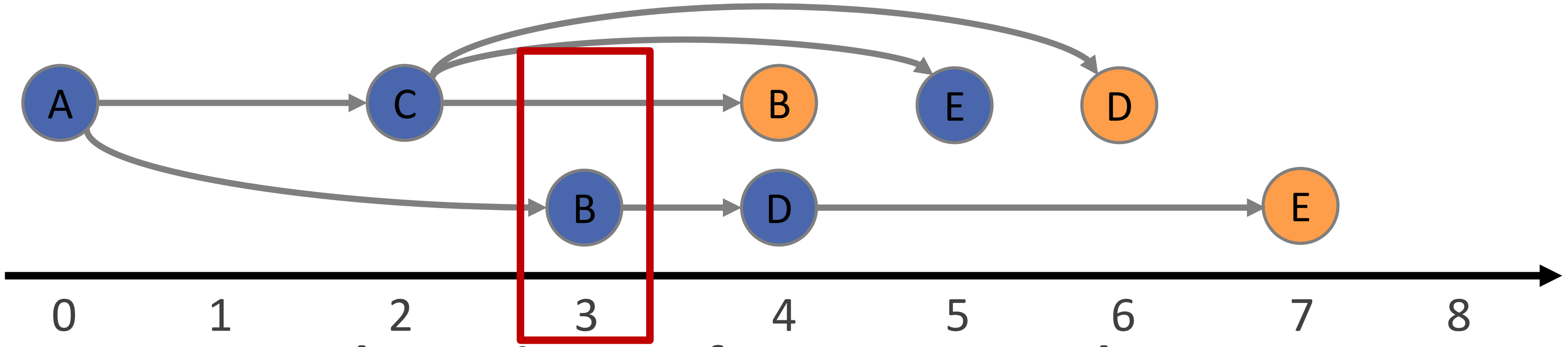
Parallelism in Dijkstra's algorithm?

Task graph



Parallelism in Dijkstra's algorithm?

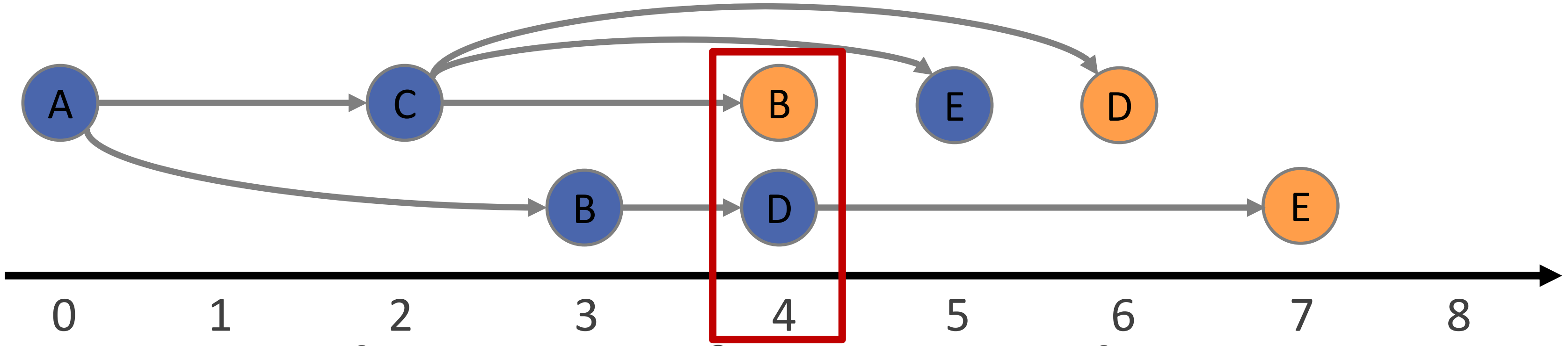
Task graph



Order = Distance from source node

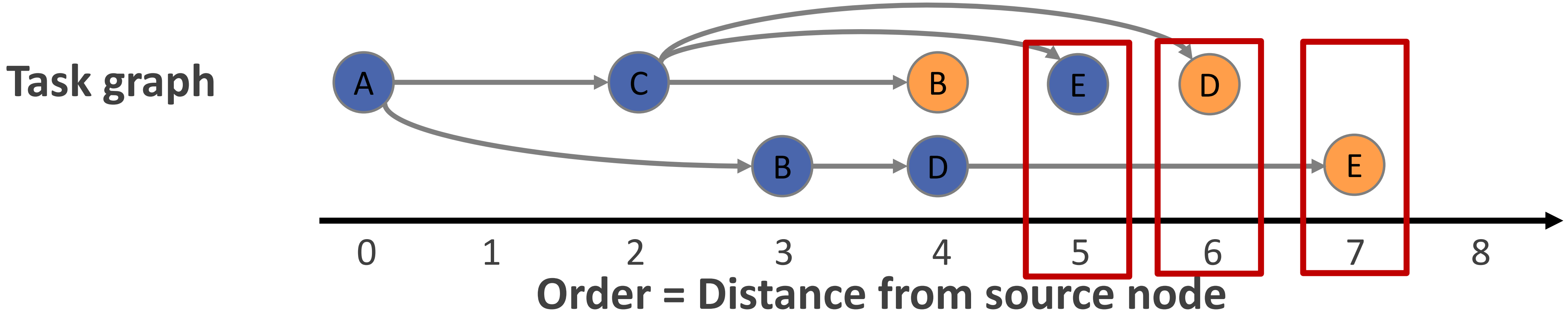
Parallelism in Dijkstra's algorithm?

Task graph



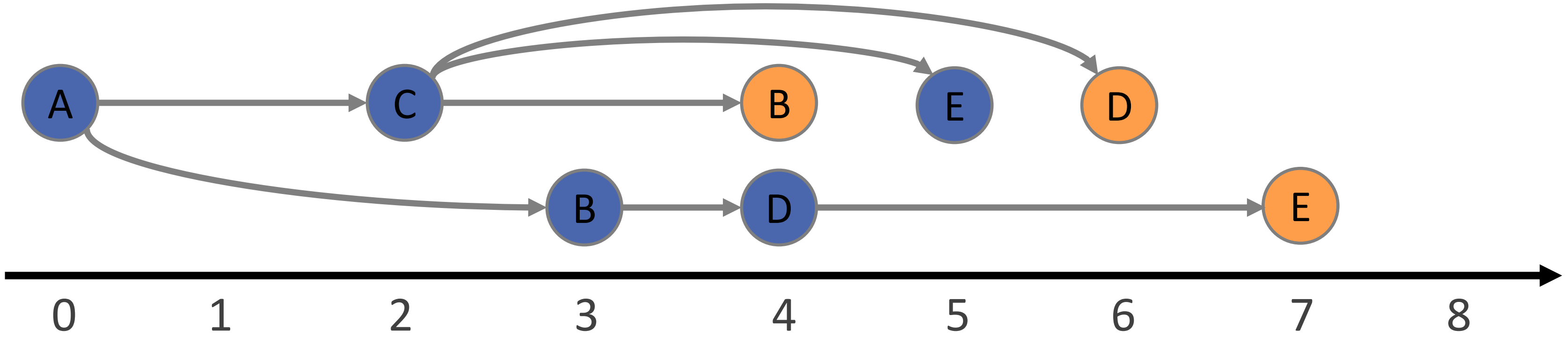
Order = Distance from source node

Parallelism in Dijkstra's algorithm?



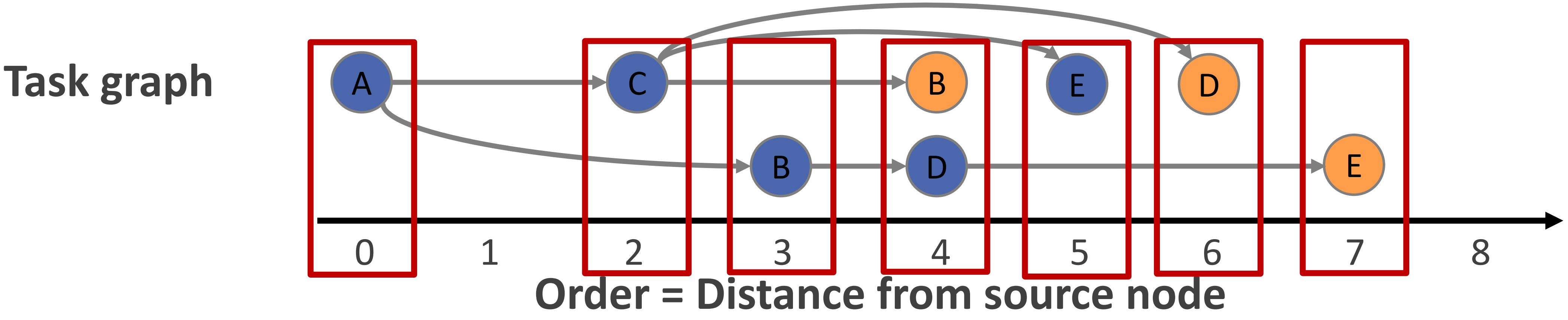
Parallelism in Dijkstra's algorithm?

Task graph



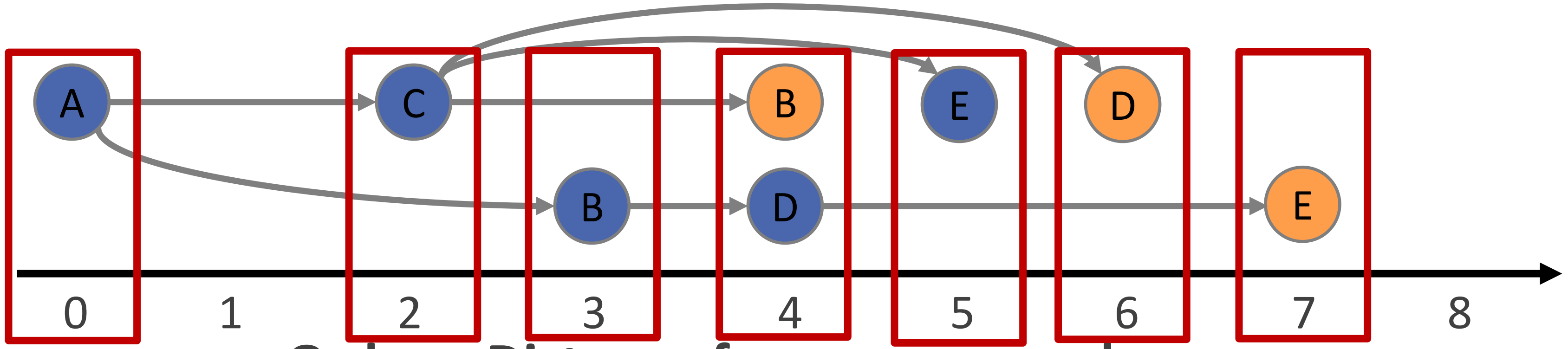
Order = Distance from source node

Parallelism in Dijkstra's algorithm?



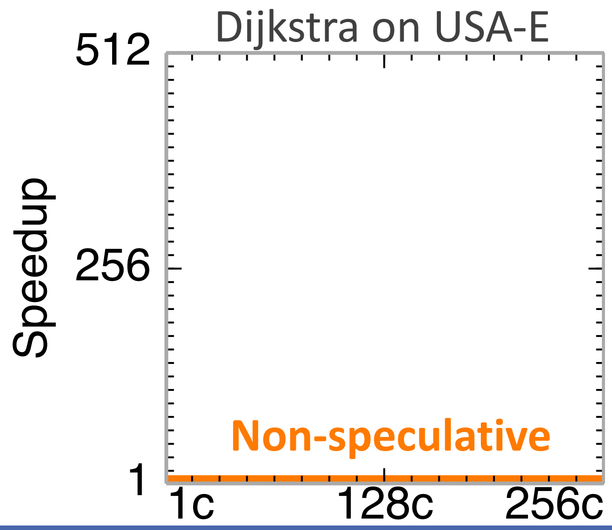
Parallelism in Dijkstra's algorithm?

Task graph



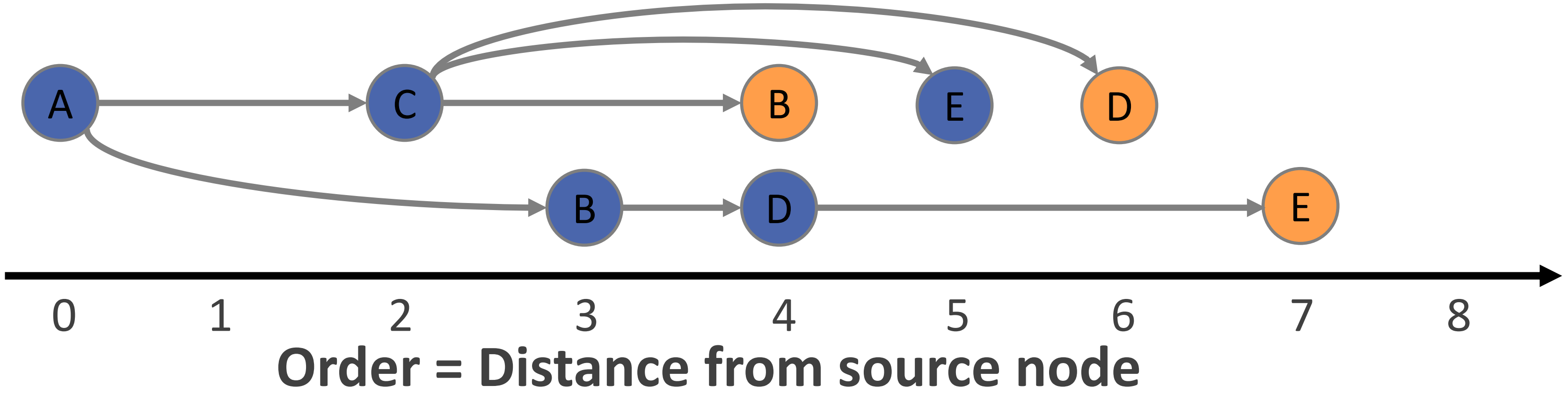
Order = Distance from source node

Dijkstra performance

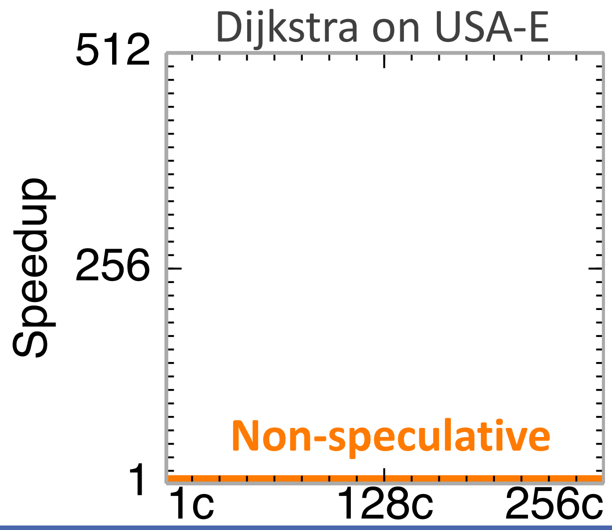


Parallelism in Dijkstra's algorithm?

Task graph

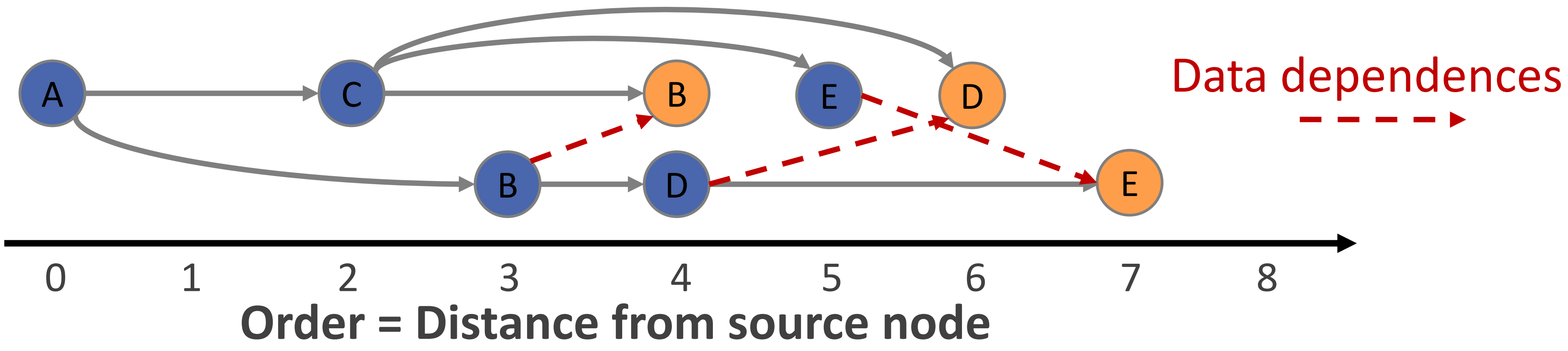


Dijkstra performance

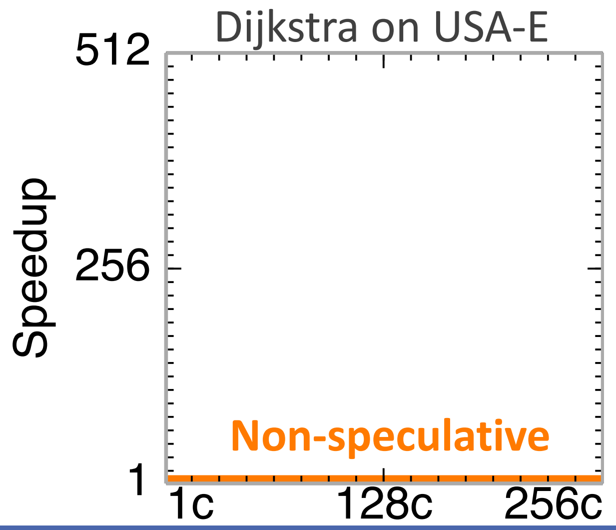


Parallelism in Dijkstra's algorithm?

Task graph

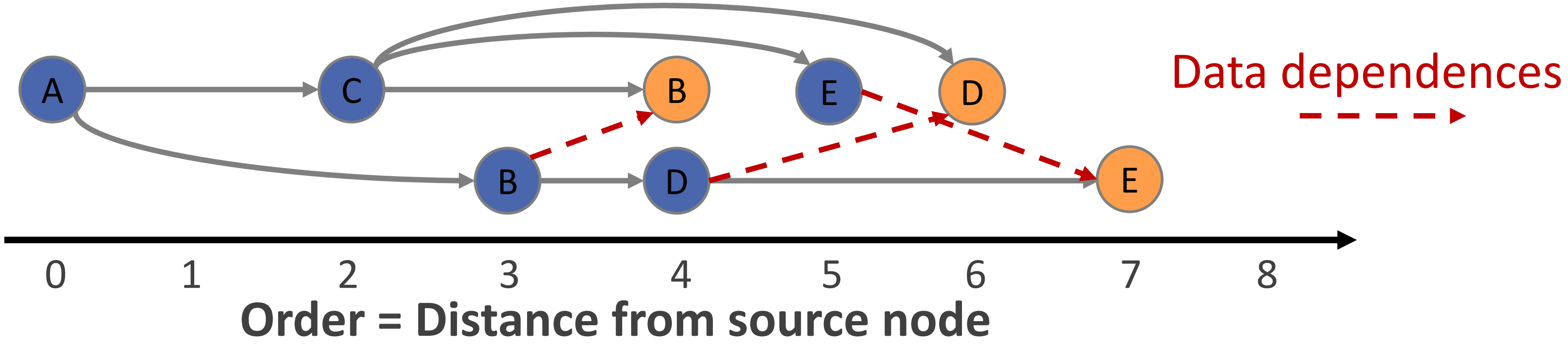


Dijkstra performance

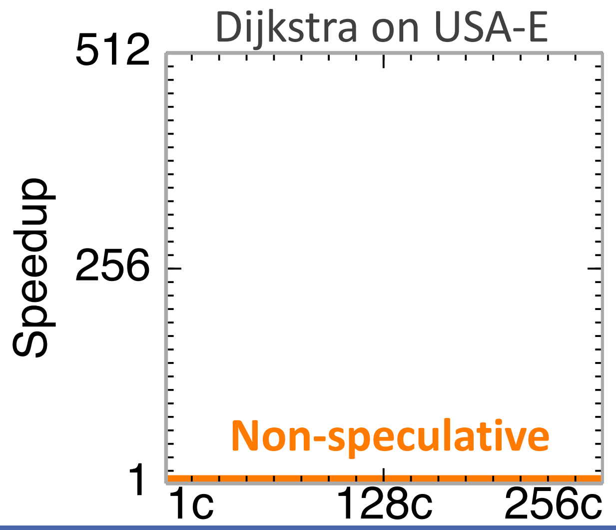


Parallelism in Dijkstra's algorithm?

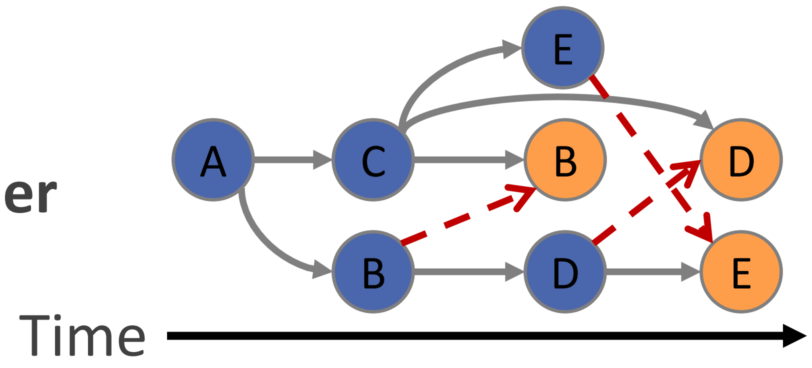
Task graph



Dijkstra performance



Valid out-of-order schedule



Dijkstra as a Swarm program [MICRO'15]

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors) {
      Timestamp nDist = dist + weight(v, n);
      swarm::enqueue(dijkstraTask, nDist, n);
    }
  }
}
```

Dijkstra as a Swarm program [MICRO'15]

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors) {
      Timestamp nDist = dist + weight(v, n);
      swarm::enqueue(dijkstraTask, nDist, n);
    }
  }
}
```

Dijkstra as a Swarm program [MICRO’15]

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors) {
      Timestamp nDist = dist + weight(v, n);
      swarm::enqueue(dijkstraTask, nDist, n);
    }
  }
}
```

Dijkstra as a Swarm program [MICRO'15]

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors) {
      Timestamp nDist = dist + weight(v, n);
      swarm::enqueue(dijkstraTask, nDist, n);
    }
  }
}
```

Function Pointer **Timestamp** **Arguments**

Dijkstra as a Swarm program [MICRO'15]

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors) {
      Timestamp nDist = dist + weight(v, n);
      swarm::enqueue(dijkstraTask, nDist, n);
    }
  }
}
swarm::enqueue(dijkstraTask, 0, sourceVertex);
swarm::run();
```

Function Pointer **Timestamp** **Arguments**

Dijkstra as a Swarm program [MICRO'15]

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors) {
      Timestamp nDist = dist + weight(v, n);
      swarm::enqueue(dijkstraTask, nDist, n);
    }
  }
}
swarm::enqueue(dijkstraTask, 0, sourceVertex);
swarm::run();
```

Function Pointer **Timestamp** **Arguments**

Implicit Parallelism

No explicit synchronization

Conveys new work to hardware as soon as possible

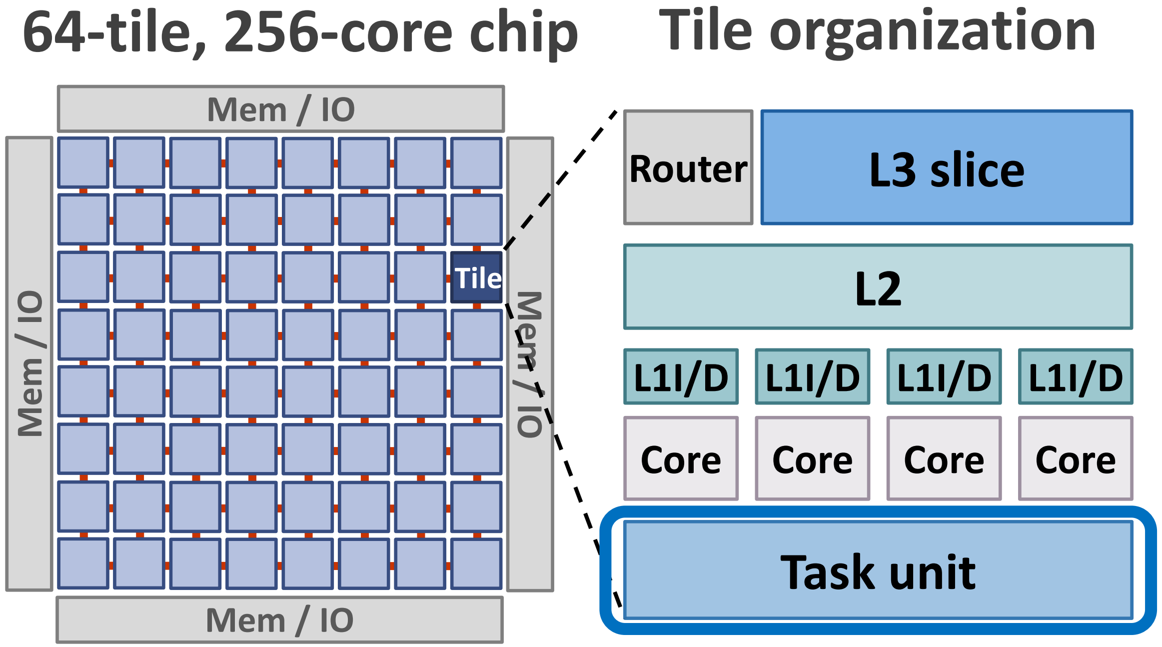
Swarm microarchitecture [MICRO'15]

Swarm executes all tasks speculatively and out of order

Large hardware task queues

Scalable ordered speculation

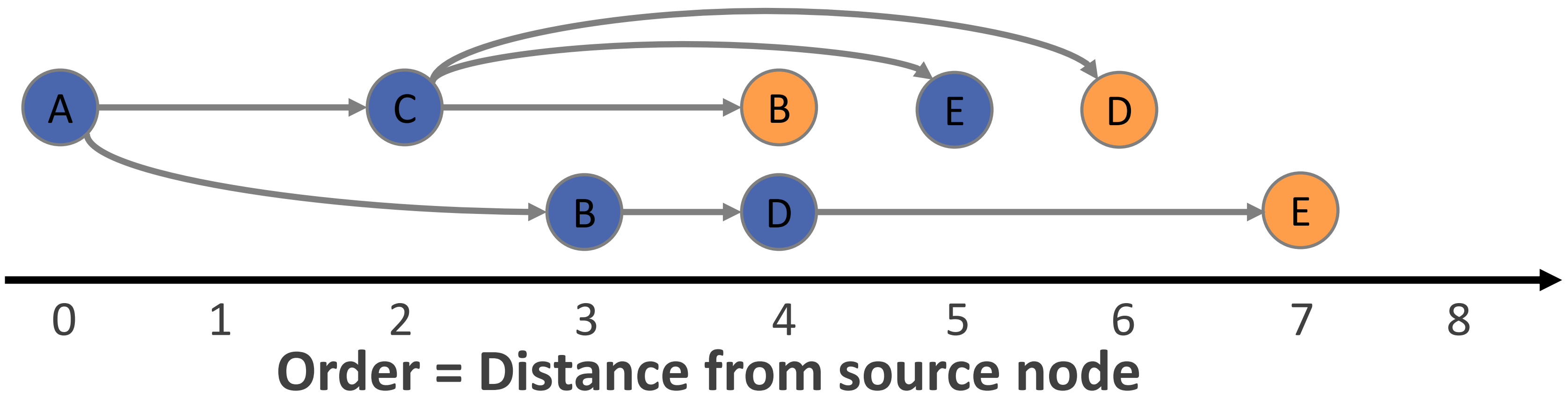
Scalable ordered commits



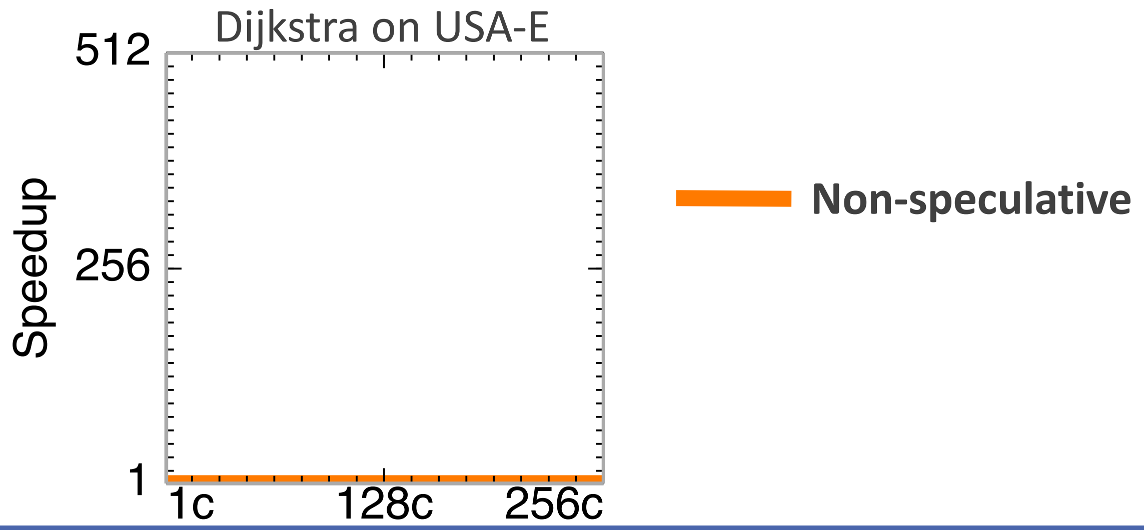
Efficiently supports thousands of tiny speculative tasks

Dijkstra's algorithm has speculative parallelism

Task graph

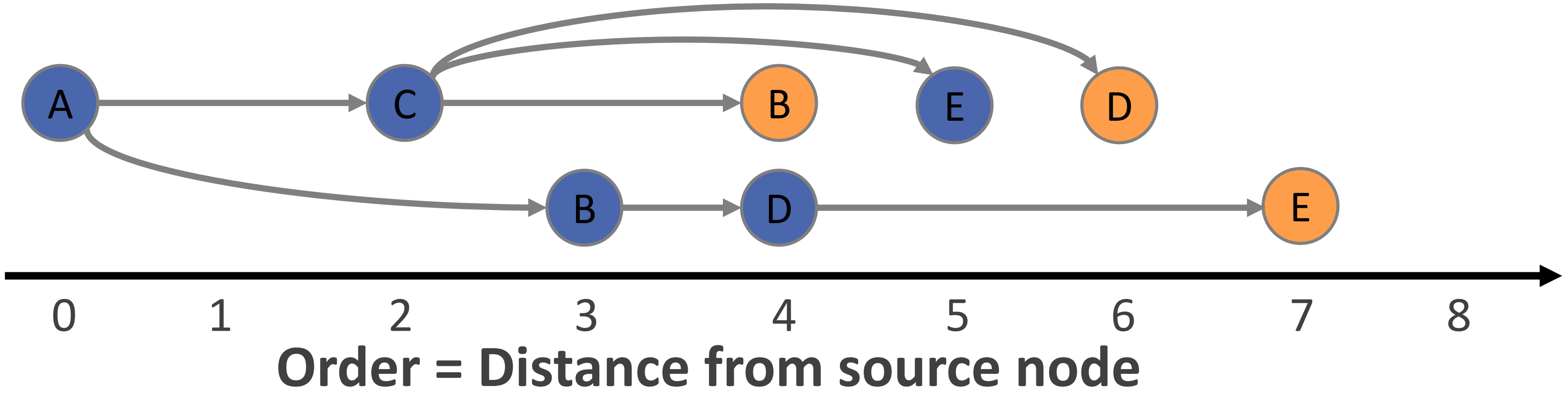


Dijkstra performance

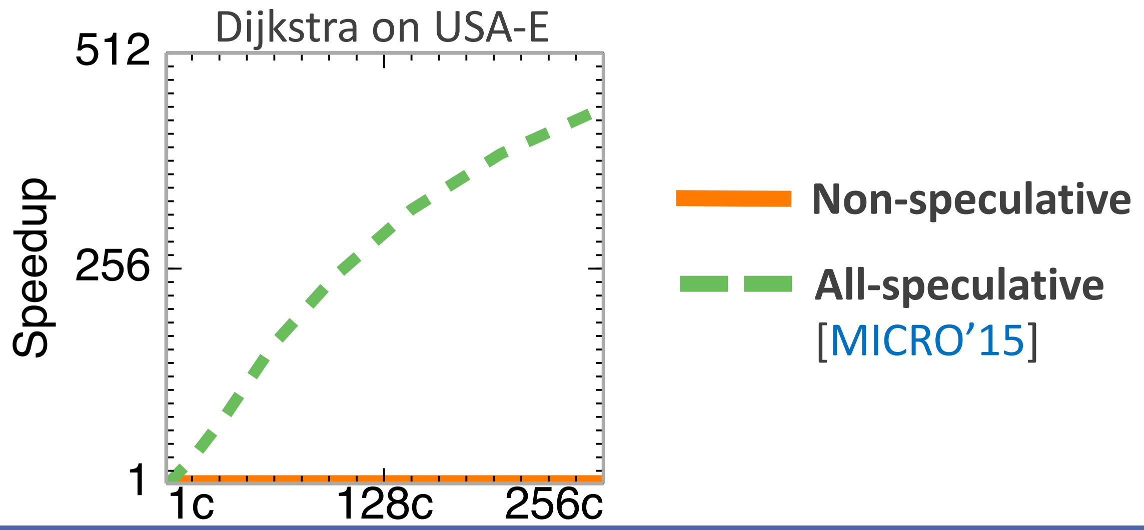


Dijkstra's algorithm has speculative parallelism

Task graph

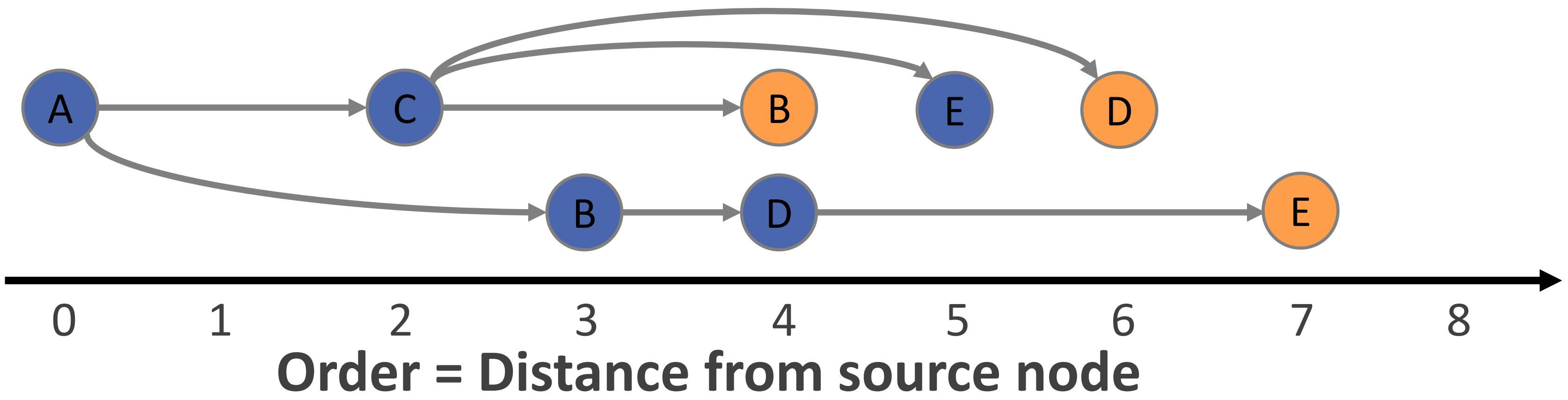


Dijkstra performance

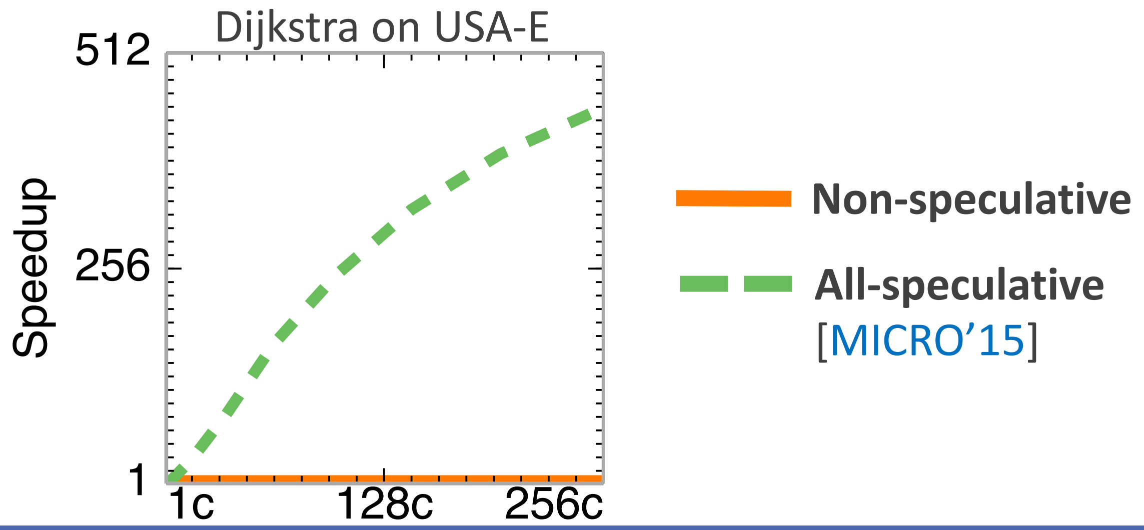


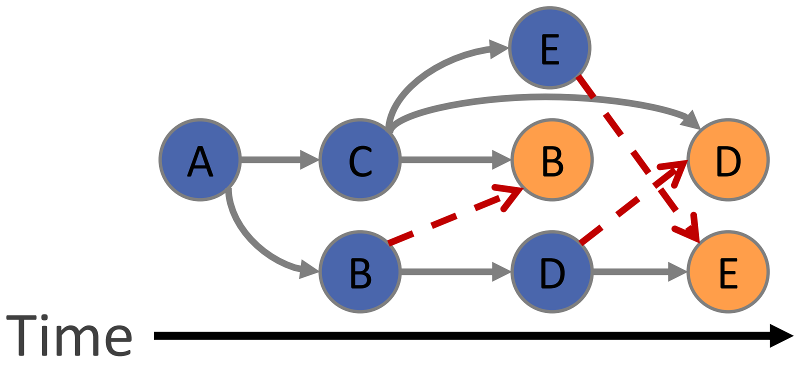
Dijkstra's algorithm has speculative parallelism

Task graph



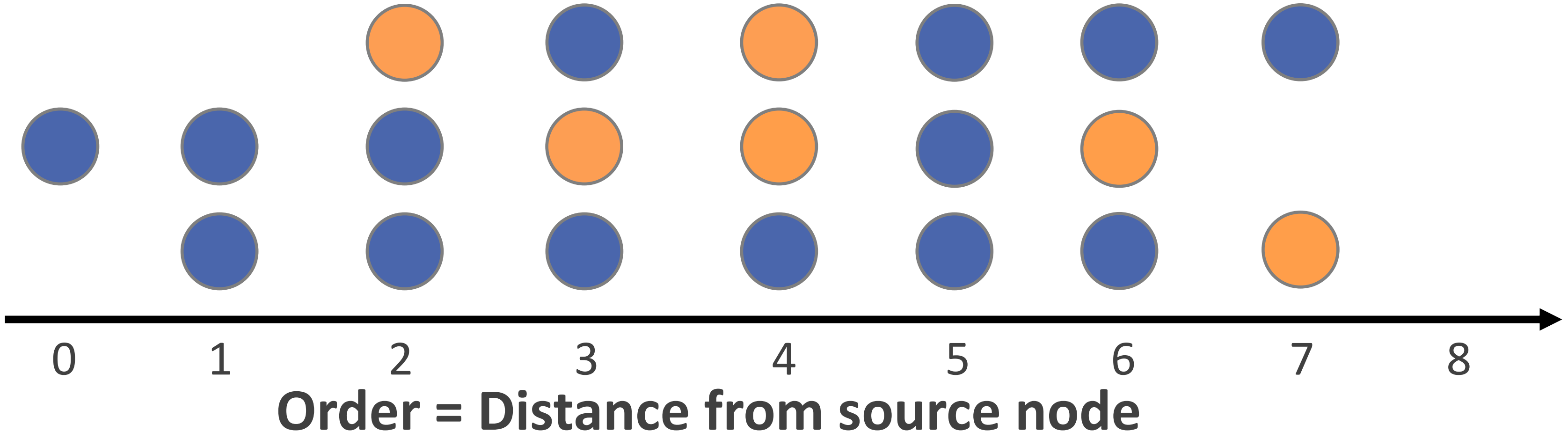
Dijkstra performance



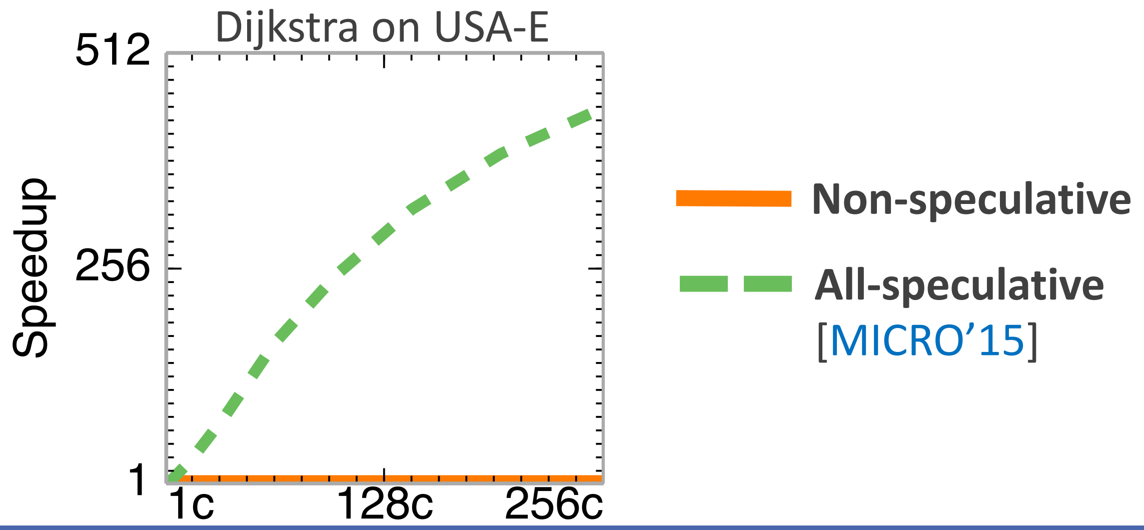


Dijkstra’s algorithm has speculative parallelism

Task graph

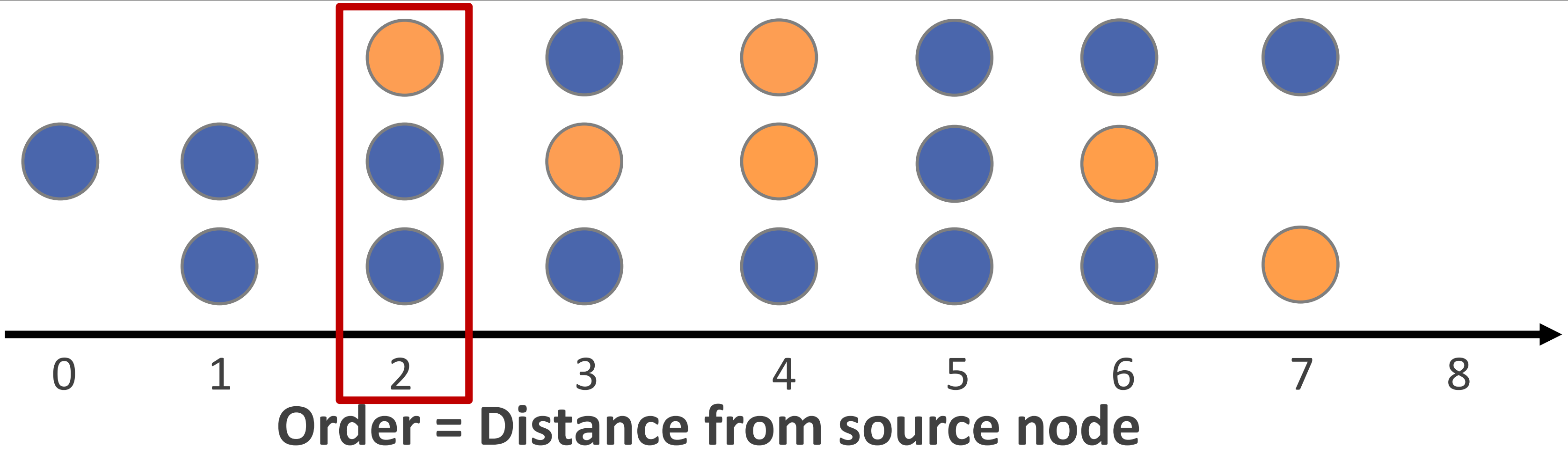


Dijkstra performance

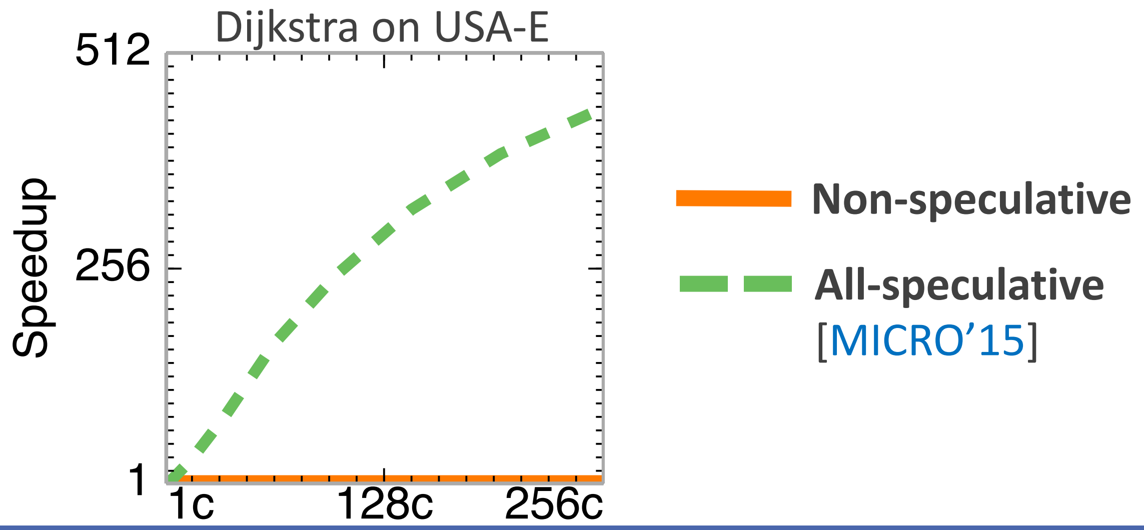


Dijkstra's algorithm has speculative parallelism

Task graph

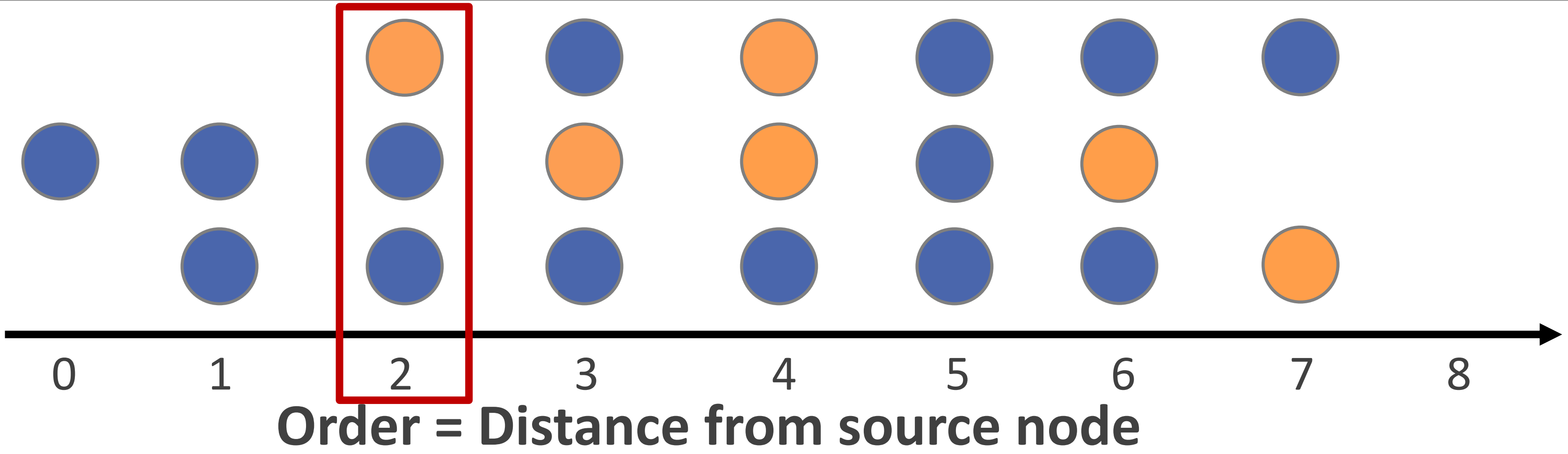


Dijkstra performance



Dijkstra's algorithm has speculative parallelism

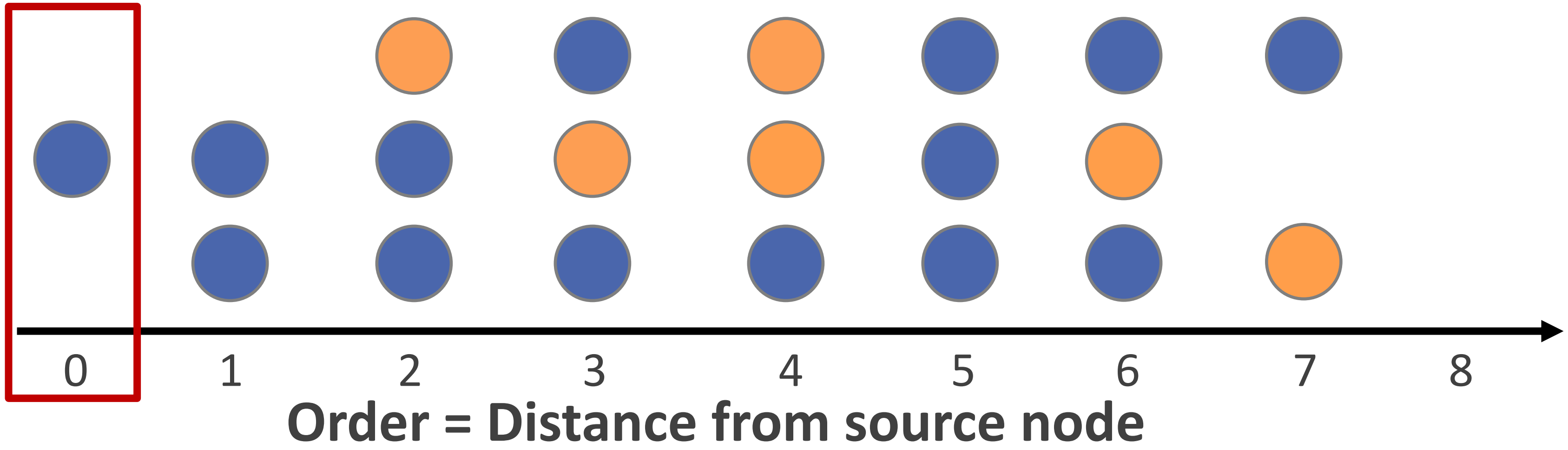
Task graph



Dijkstra performance

Dijkstra’s algorithm has speculative parallelism

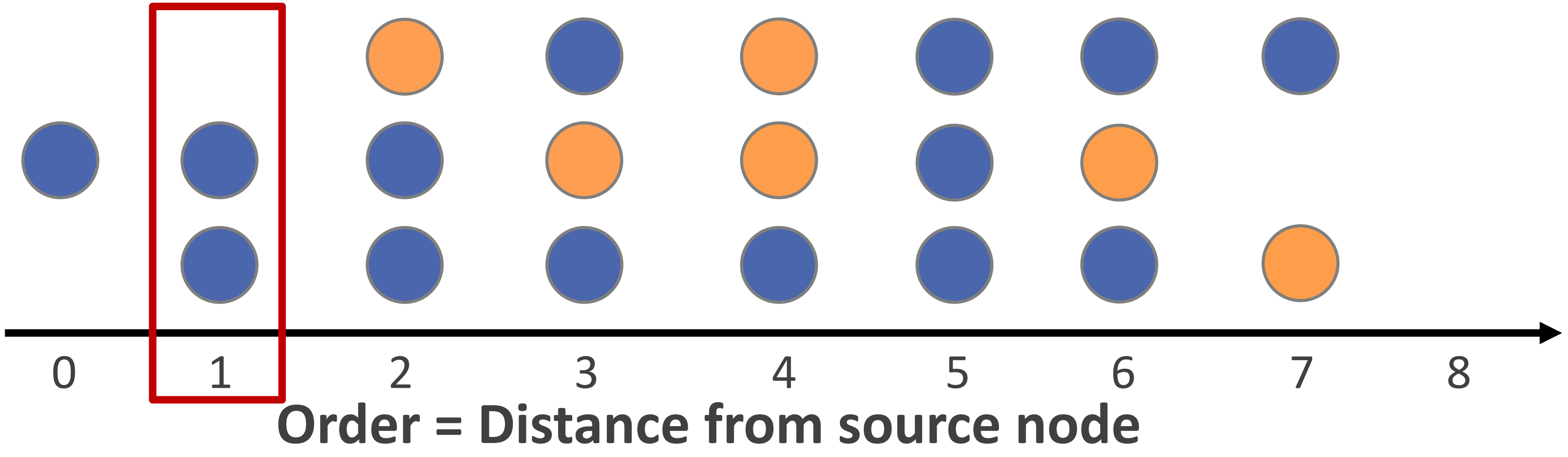
Task graph



Dijkstra performance

Dijkstra's algorithm has speculative parallelism

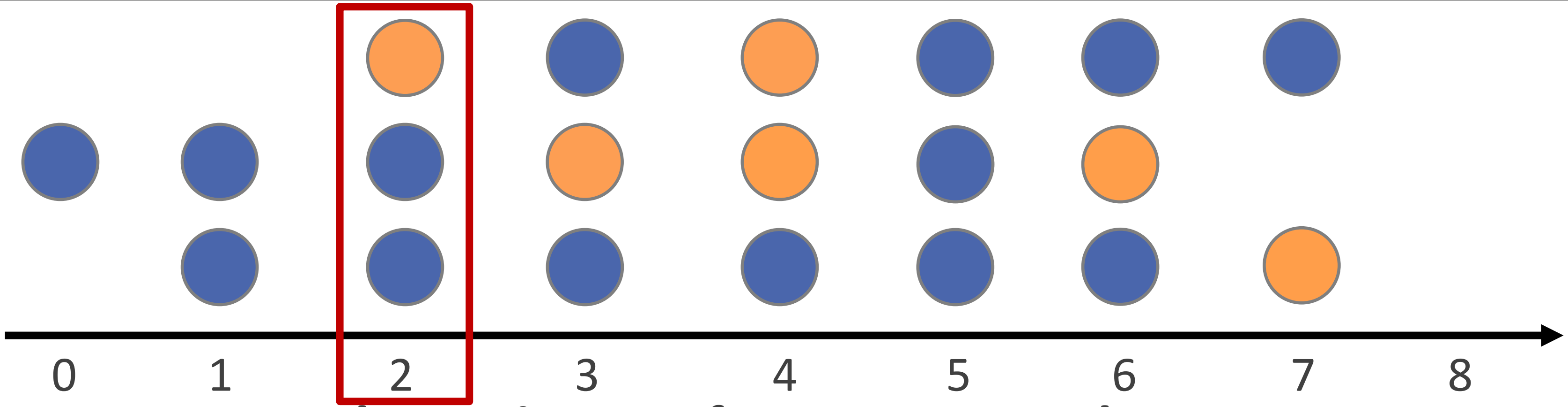
Task graph



Dijkstra performance

Dijkstra’s algorithm has speculative parallelism

Task graph

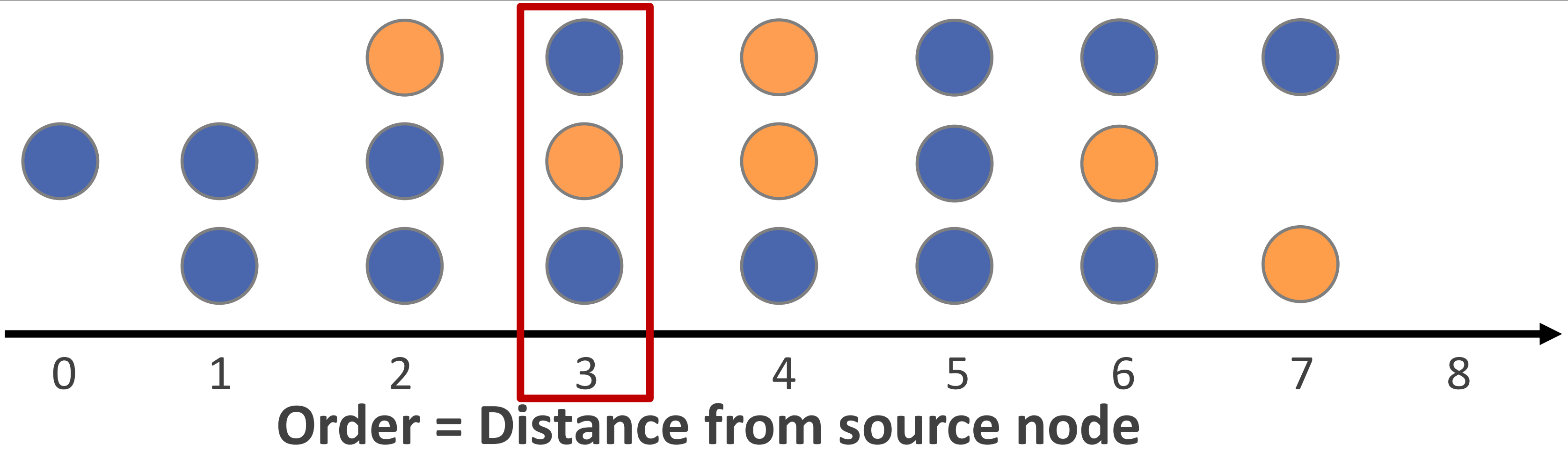


Order = Distance from source node

Dijkstra performance

Dijkstra's algorithm has speculative parallelism

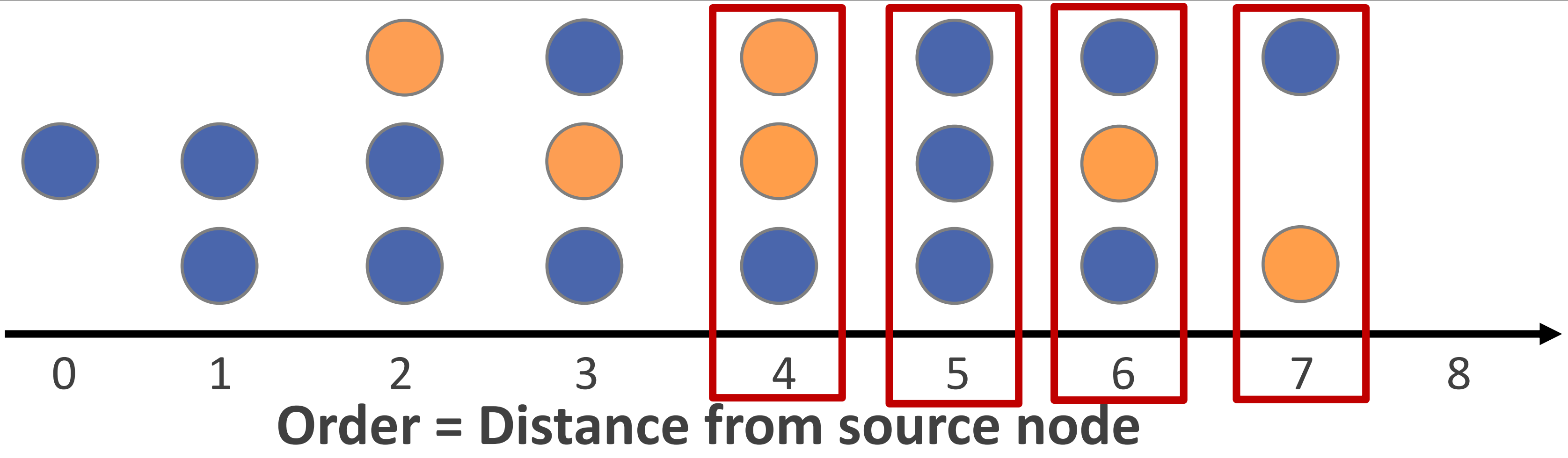
Task graph



Dijkstra performance

Dijkstra's algorithm has speculative parallelism

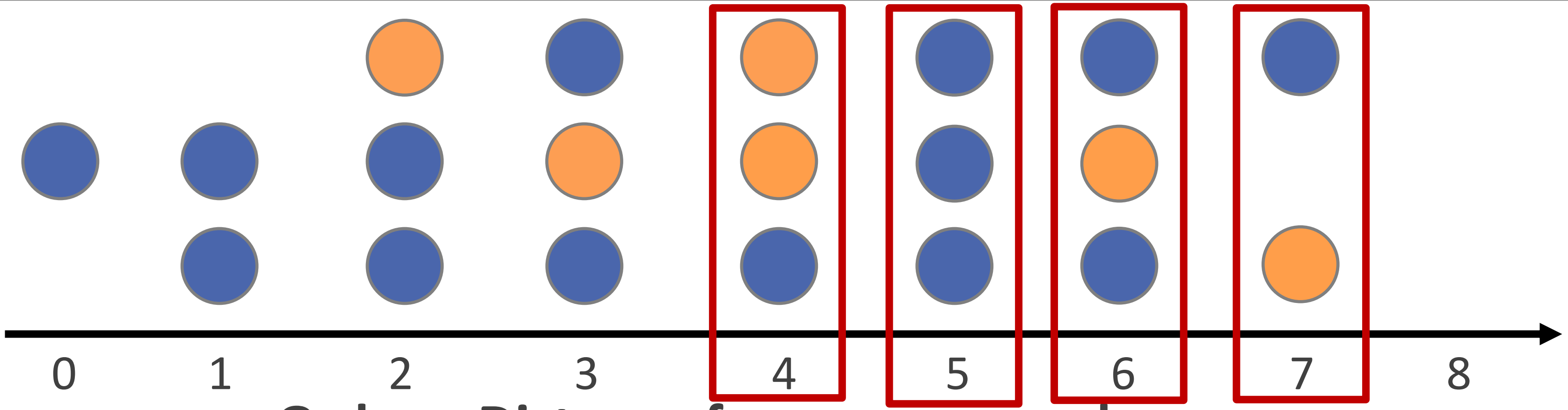
Task graph



Dijkstra performance

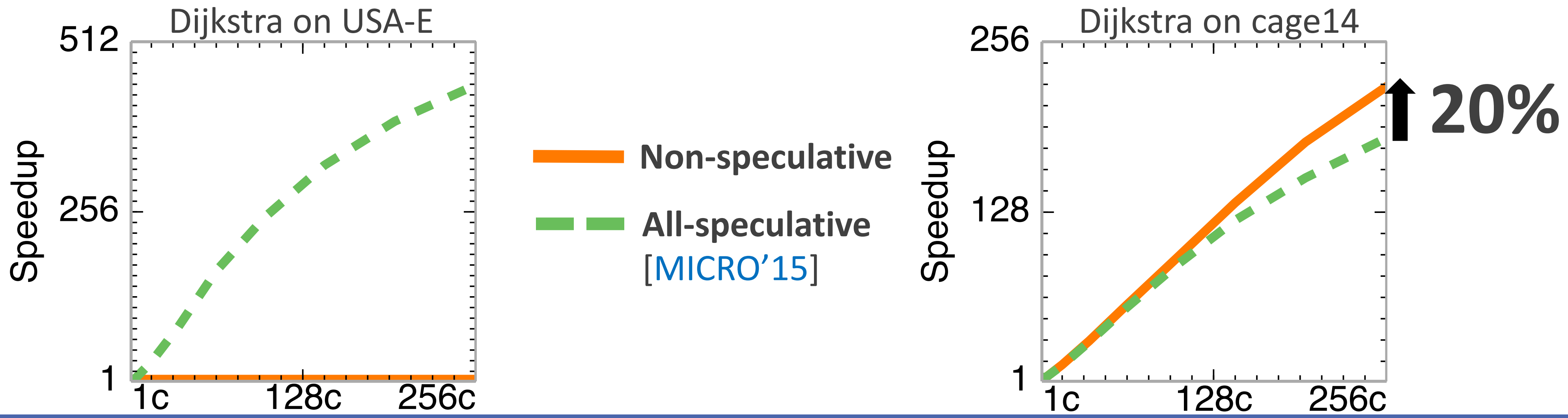
Dijkstra’s algorithm has speculative parallelism

Task graph



Order = Distance from source node

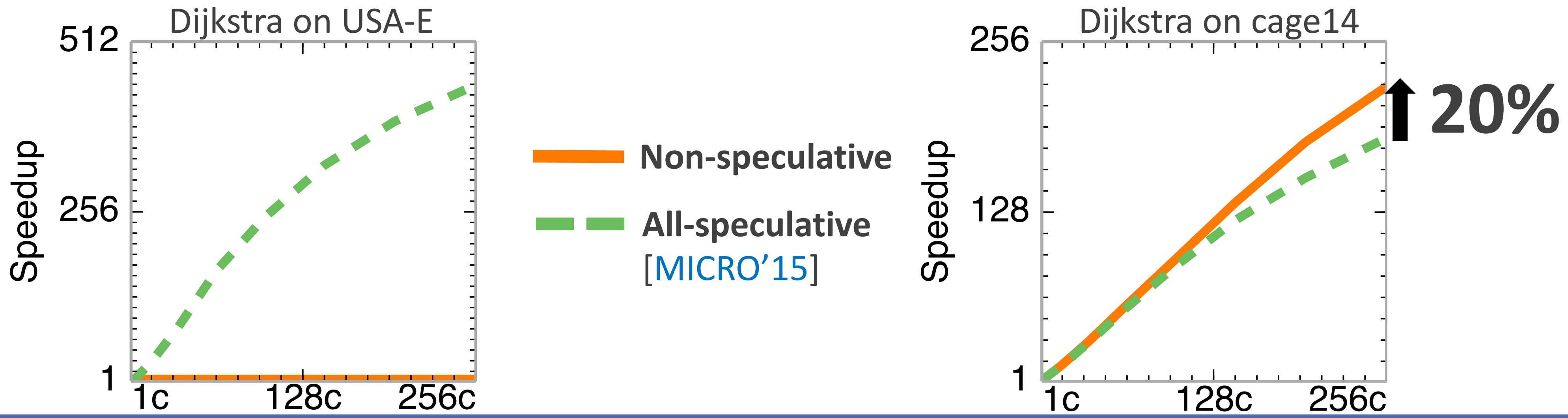
Dijkstra performance



Dijkstra's algorithm has speculative parallelism

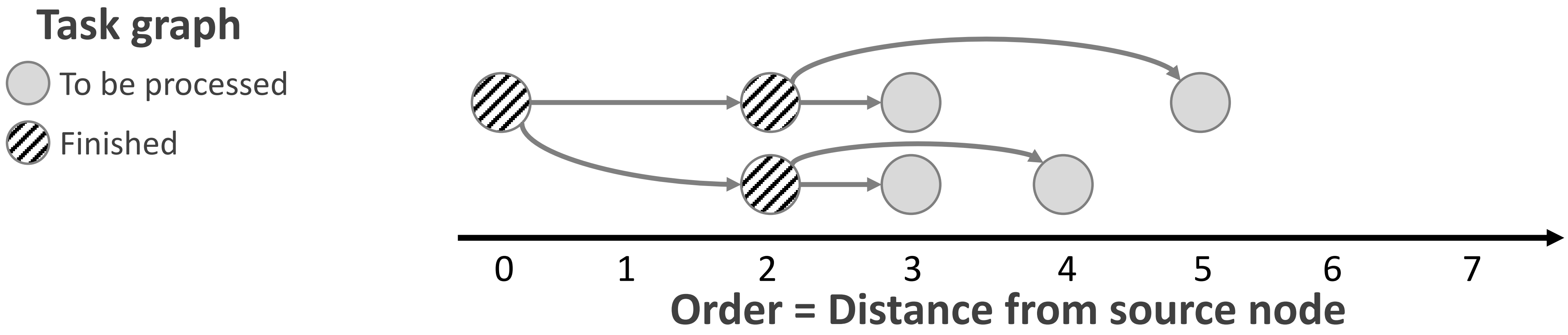
All-or-nothing speculation unduly burdens programmers

Dijkstra performance

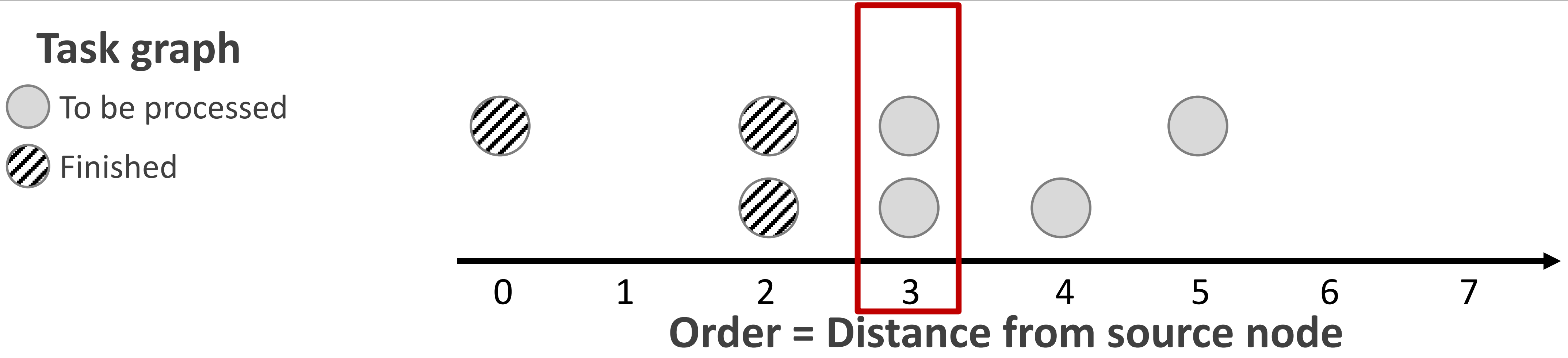


Dijkstra’s algorithm needs a hybrid strategy

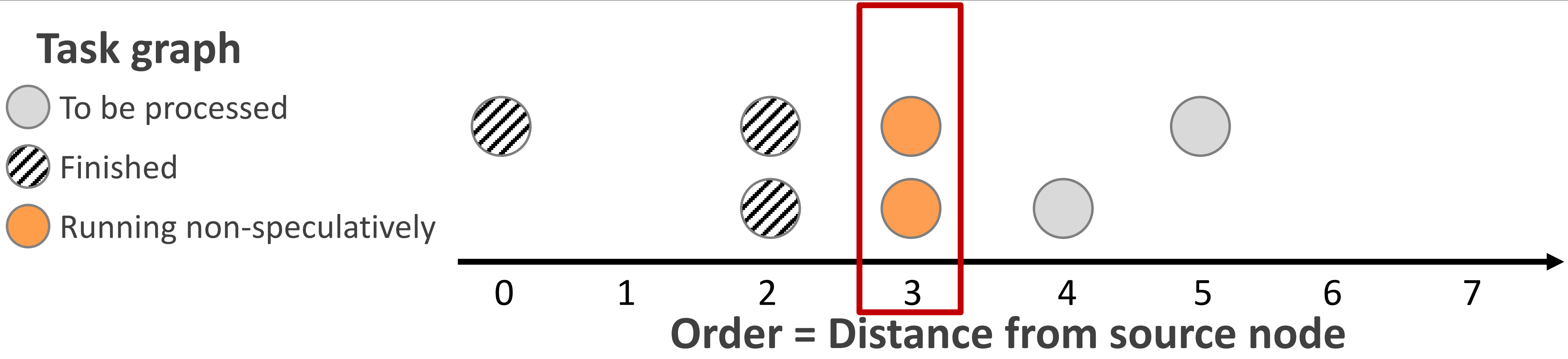
Dijkstra’s algorithm needs a hybrid strategy



Dijkstra’s algorithm needs a hybrid strategy

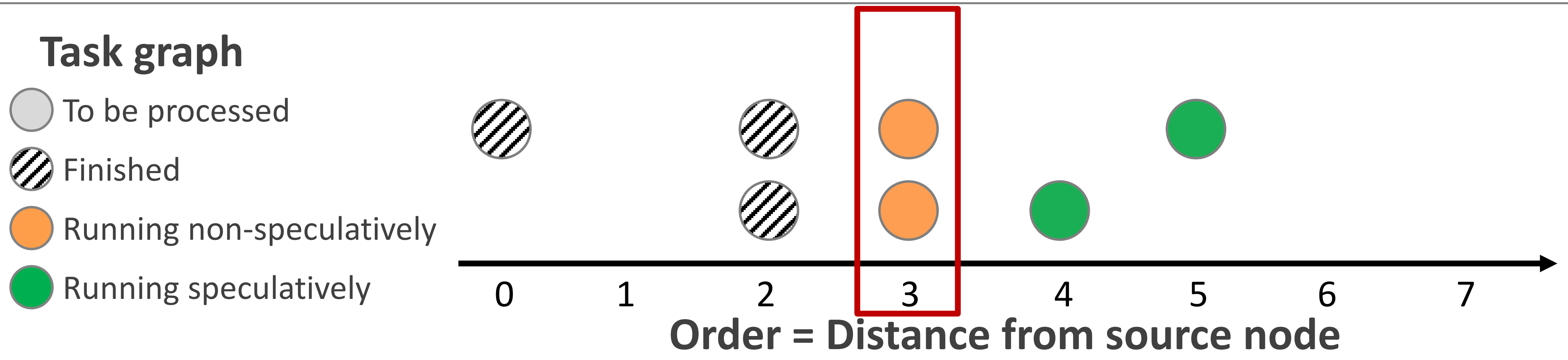


Dijkstra’s algorithm needs a hybrid strategy



Run tasks **non-speculatively** when possible

Dijkstra’s algorithm needs a hybrid strategy



Run tasks **non-speculatively** when possible

Keep cores busy with **speculative** ordered parallelism

Dijkstra’s algorithm needs a hybrid strategy

Run tasks **non-speculatively** when possible

Keep cores busy with **speculative** ordered parallelism

Dijkstra’s algorithm needs a hybrid strategy

Run tasks **non-speculatively** when possible

Keep cores busy with **speculative** ordered parallelism

Dijkstra's algorithm needs a hybrid strategy

Run tasks **non-speculatively** when possible

Keep cores busy with **speculative** ordered parallelism

Dijkstra’s algorithm needs a hybrid strategy

Run tasks **non-speculatively** when possible

Keep cores busy with **speculative** ordered parallelism

Dijkstra’s algorithm needs a hybrid strategy

Run tasks **non-speculatively** when possible

Keep cores busy with **speculative** ordered parallelism

Each task must be runnable in either mode

Dijkstra's algorithm needs a hybrid strategy

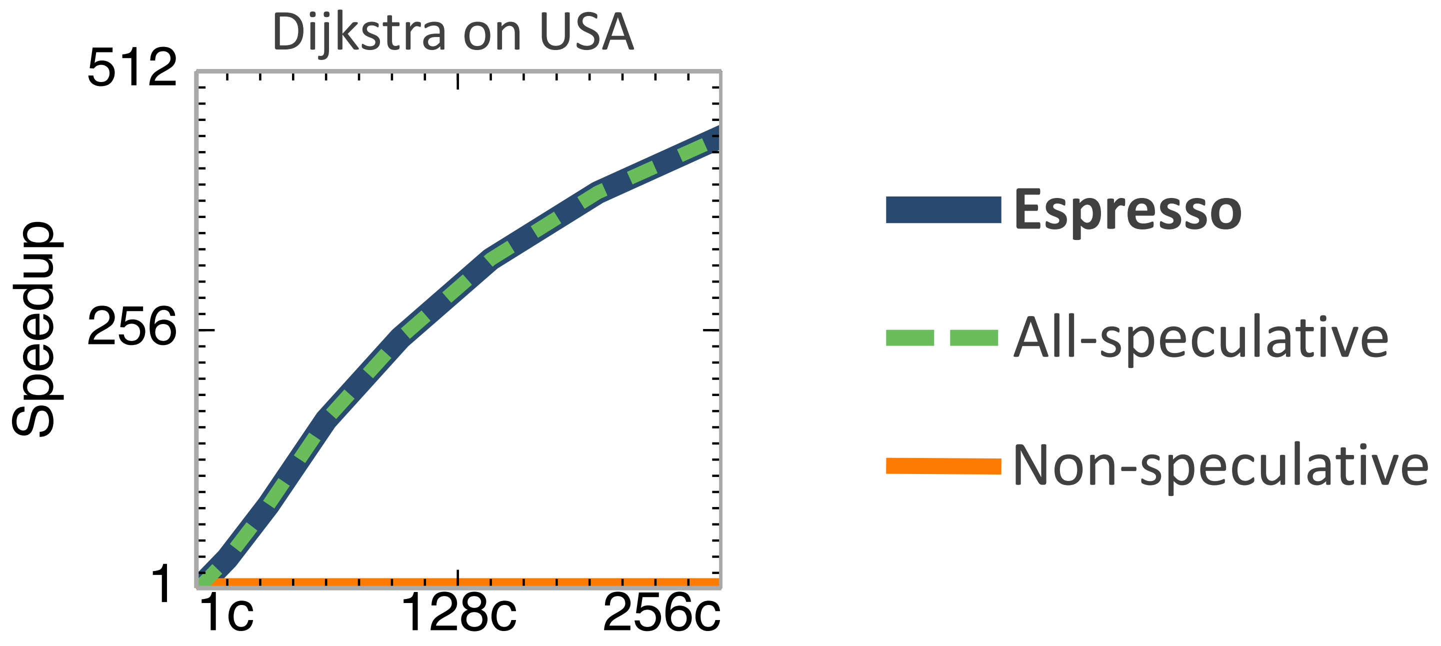
Run tasks **non-speculatively** when possible

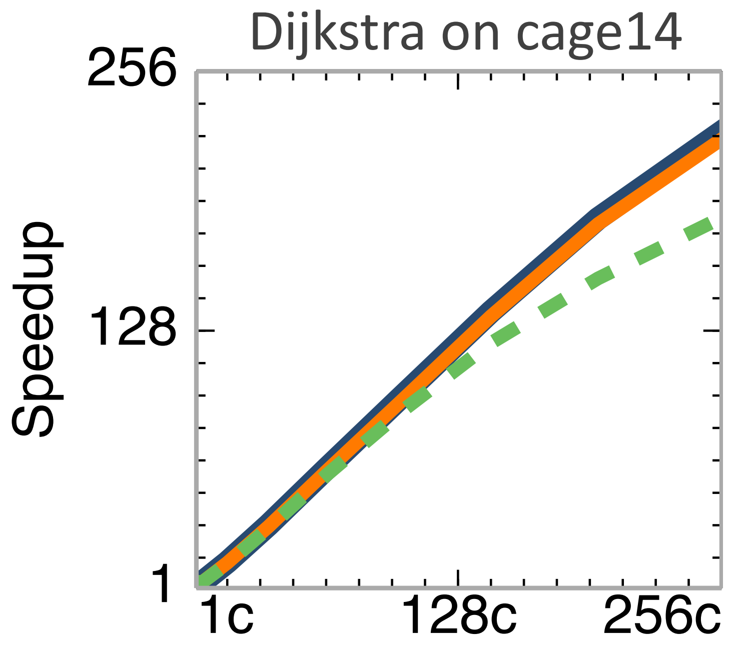
Keep cores busy with **speculative** ordered parallelism

Each task must be runnable in either mode

Tasks in both modes must coordinate on shared data

Espresso reaps the benefits of non-speculative and speculative parallelism

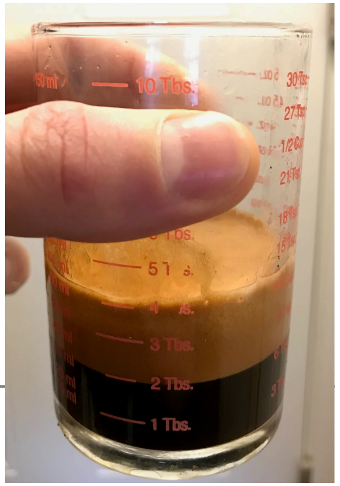




Espresso avoids pathologies and scales best

Espresso

COORDINATING SPECULATIVE AND NON-SPECULATIVE PARALLELISM



Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
        dijkstraTask,
        dist + weight(v, n),
        n->id,
        n);
  }
}
```

Function pointer → dijkstraTask,

Arguments → n);

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
        dijkstraTask,
        dist + weight(v, n),
        n->id,
        n);
  }
}
```

Function pointer

Timestamp

Arguments

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
        dijkstraTask,
        dist + weight(v, n),
        n->id,
        n);
  }
}
```

Function pointer → dijkstraTask

Arguments → n);

	Non-Spec.	Spec.
Timestamp	barrier	ordered commits

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
        dijkstraTask,
        dist + weight(v, n),
        n->id,
        n);
  }
}
```

Function pointer → dijkstraTask

Arguments → n

	Non-Spec.	Spec.
Timestamp	barrier	ordered commits
Locale		

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
        dijkstraTask,
        dist + weight(v, n),
        n->id,
        n);
  }
}
```

Function pointer → dijkstraTask

Arguments → n);

	Non-Spec.	Spec.
Timestamp	barrier	ordered commits
Locale	mutex	reduce conflicts

Espresso execution model

Programs consist of tasks that run *speculatively* or *non-speculatively*

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create(
        dijkstraTask,
        dist + weight(v, n),
        n->id,
        n);
  }
```

Function pointer

Arguments

	Non-Spec.	Spec.
Timestamp	barrier	ordered commits
Locale	mutex	reduce conflicts

Tasks in either mode can coordinate access to shared data

Espresso task dispatch

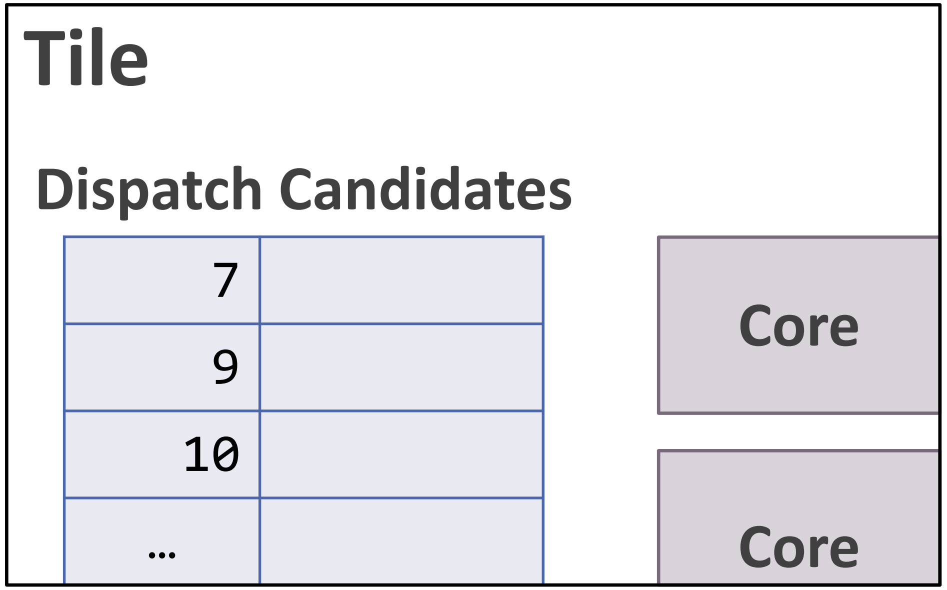
Espresso supports three task *types* that control speculation

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create< type  >(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```

Espresso task dispatch

Espresso supports three task *types* that control speculation

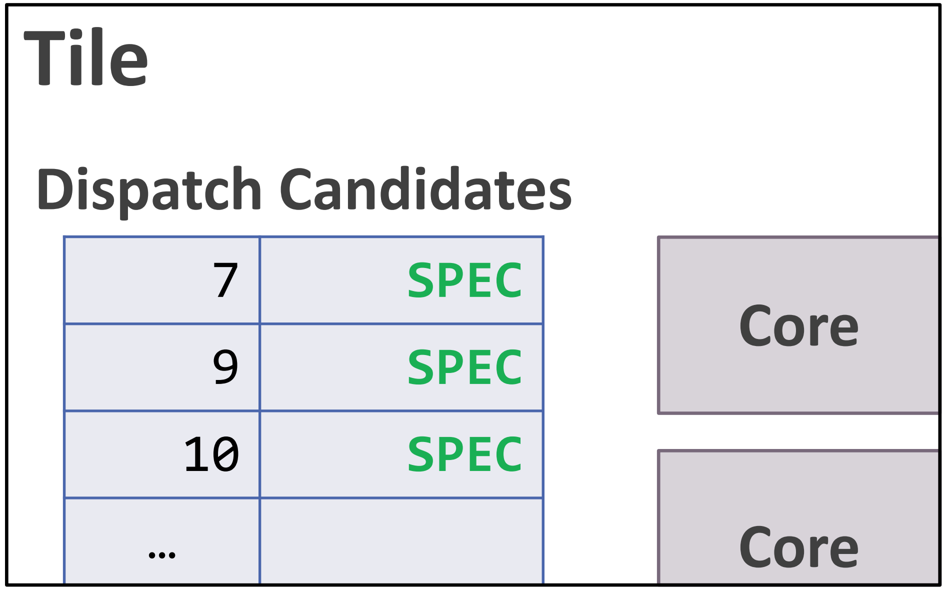
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create< type  >(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

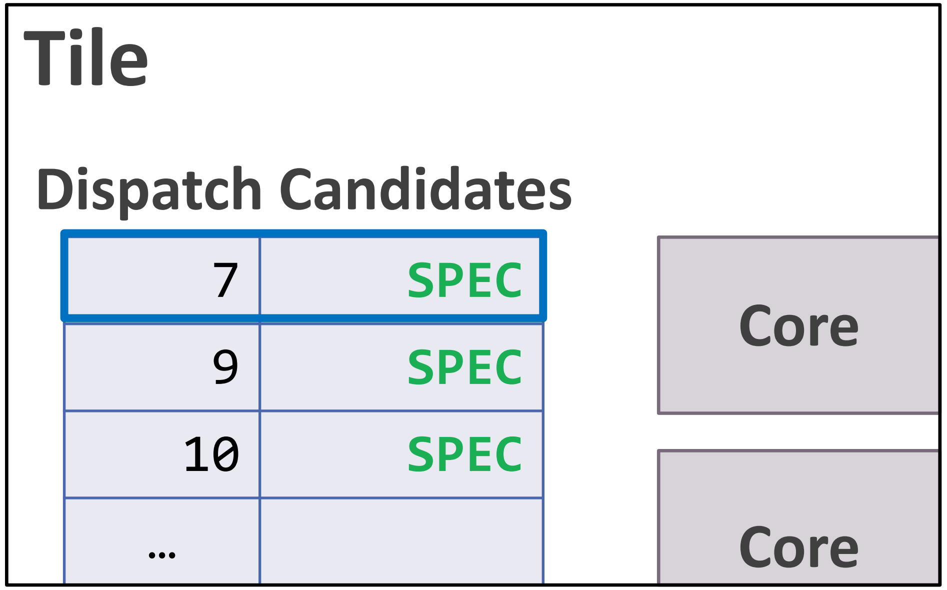
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create< SPEC >(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

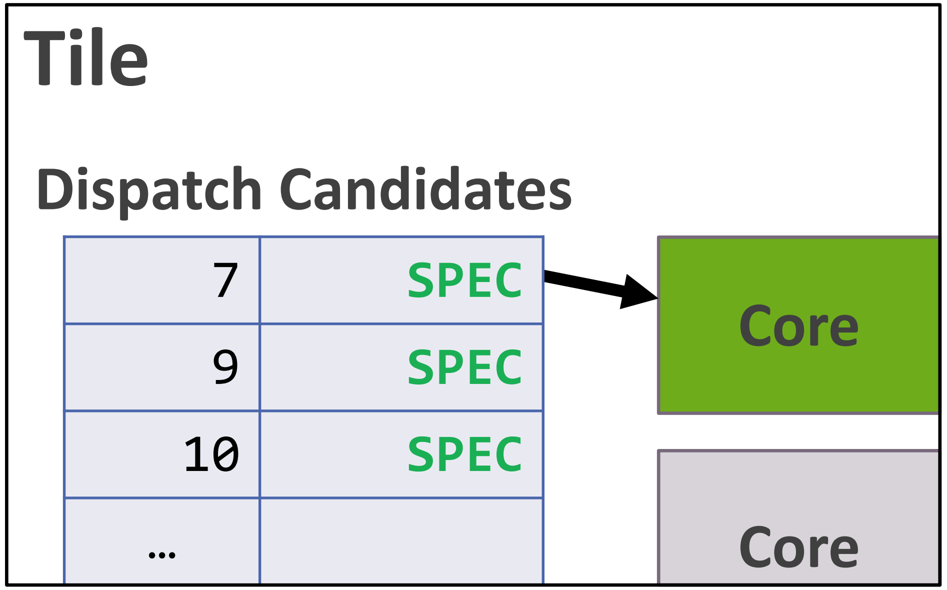
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create< SPEC >(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

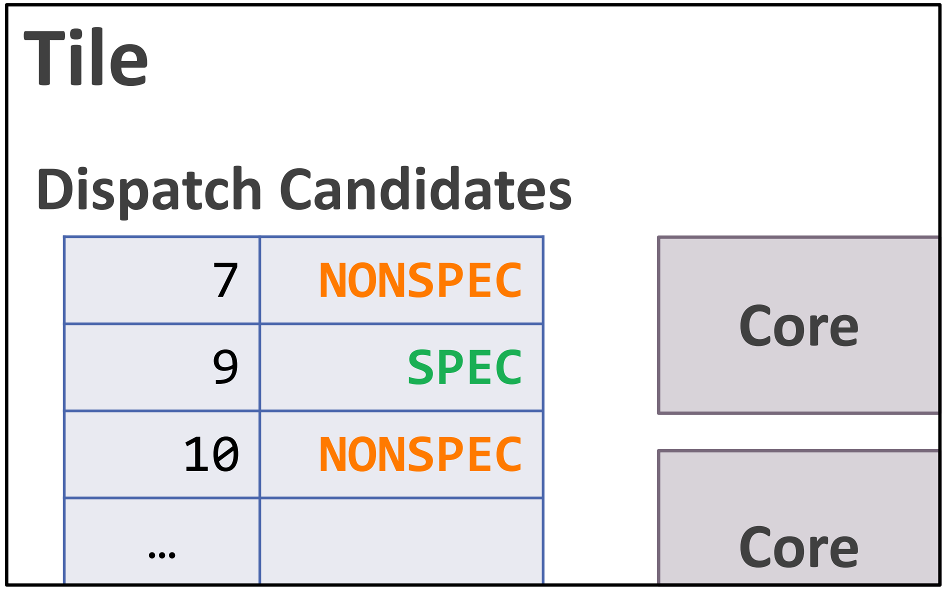
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create< SPEC >(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

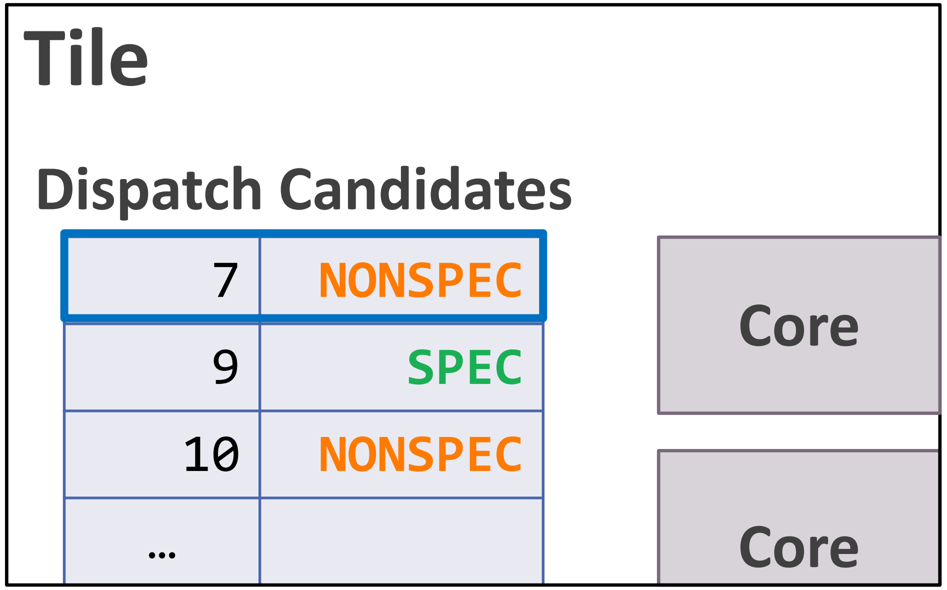
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<NONSPEC>(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

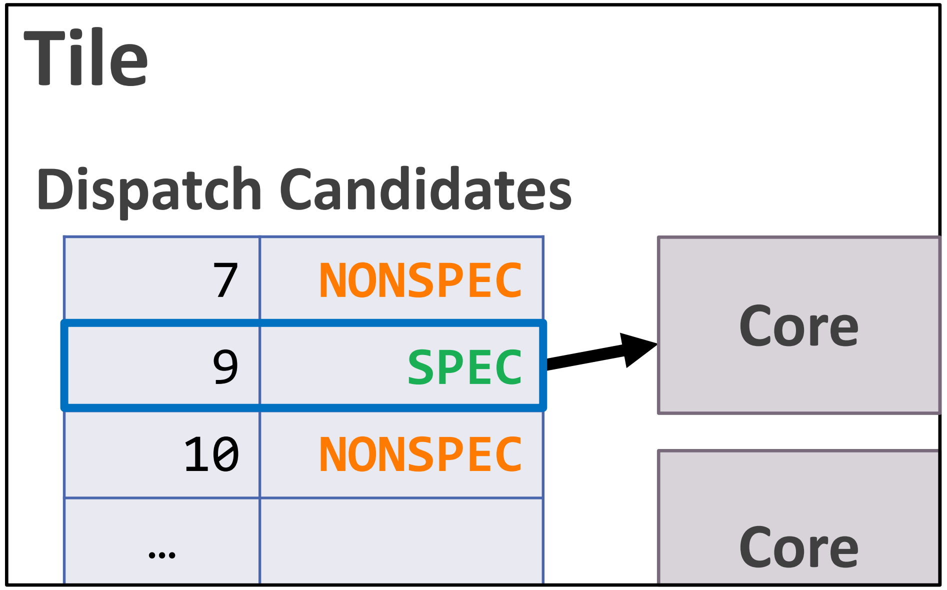
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<NONSPEC>(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

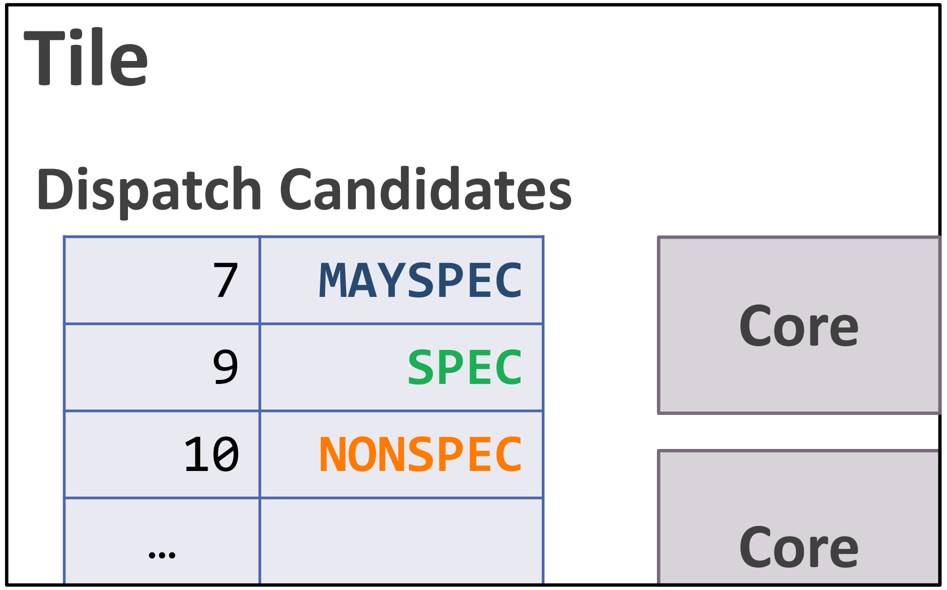
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<NONSPEC>(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

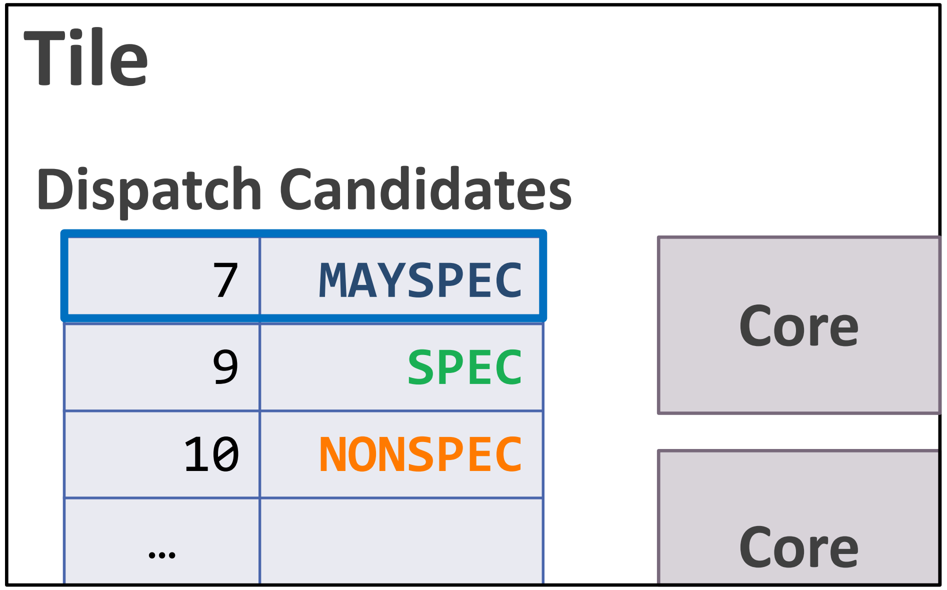
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<MAYSPEC>(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

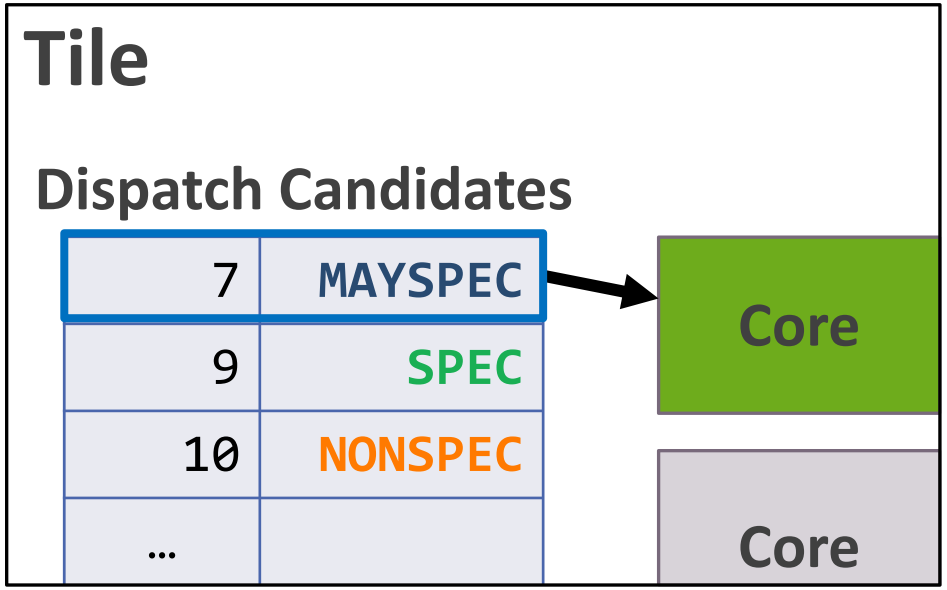
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<MAYSPEC>(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

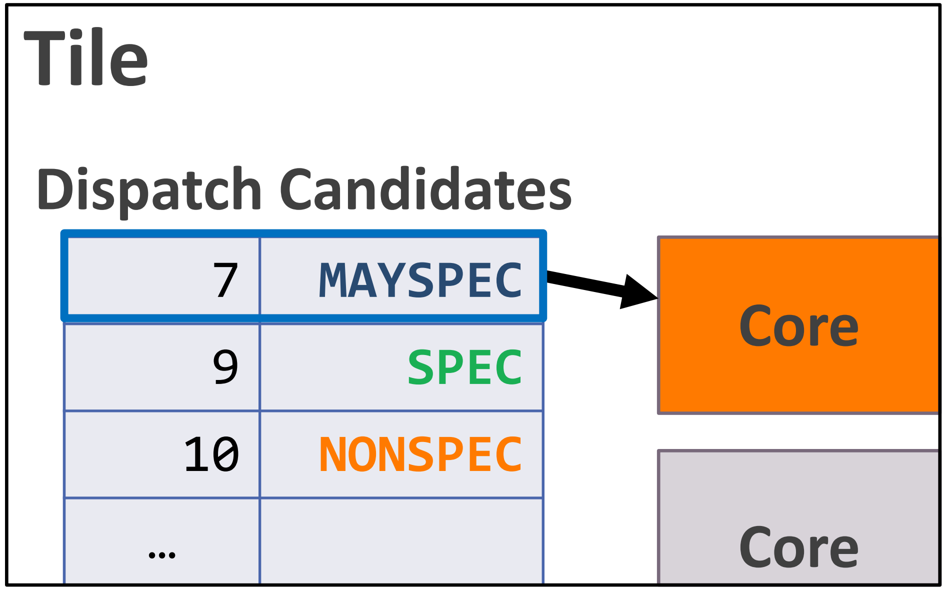
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<MAYSPEC>(
           dijkstraTask,
           dist + weight(v, n),
           n->id,
           n);
  }
}
```



Espresso task dispatch

Espresso supports three task *types* that control speculation

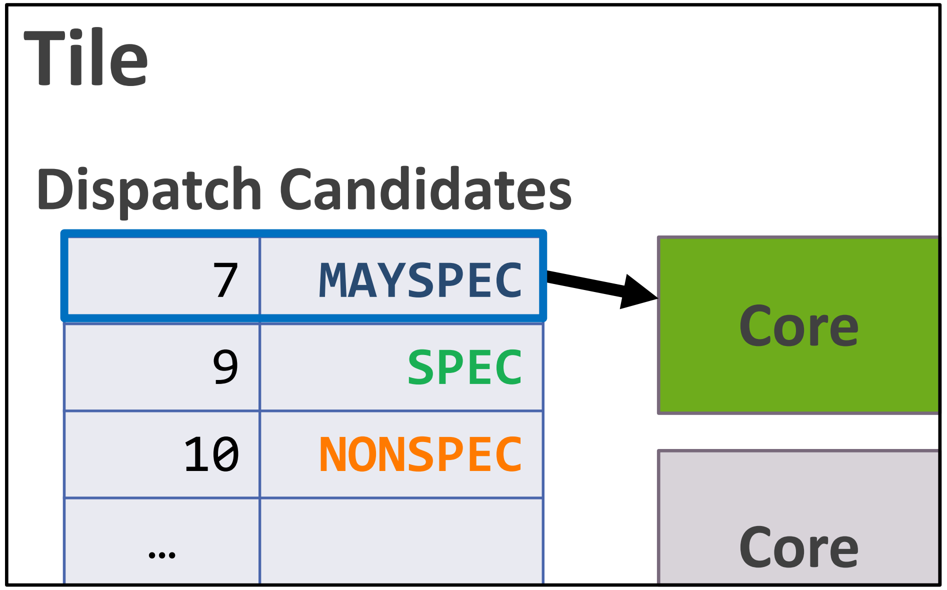
```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<MAYSPEC>(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
}
```



Espresso task dispatch

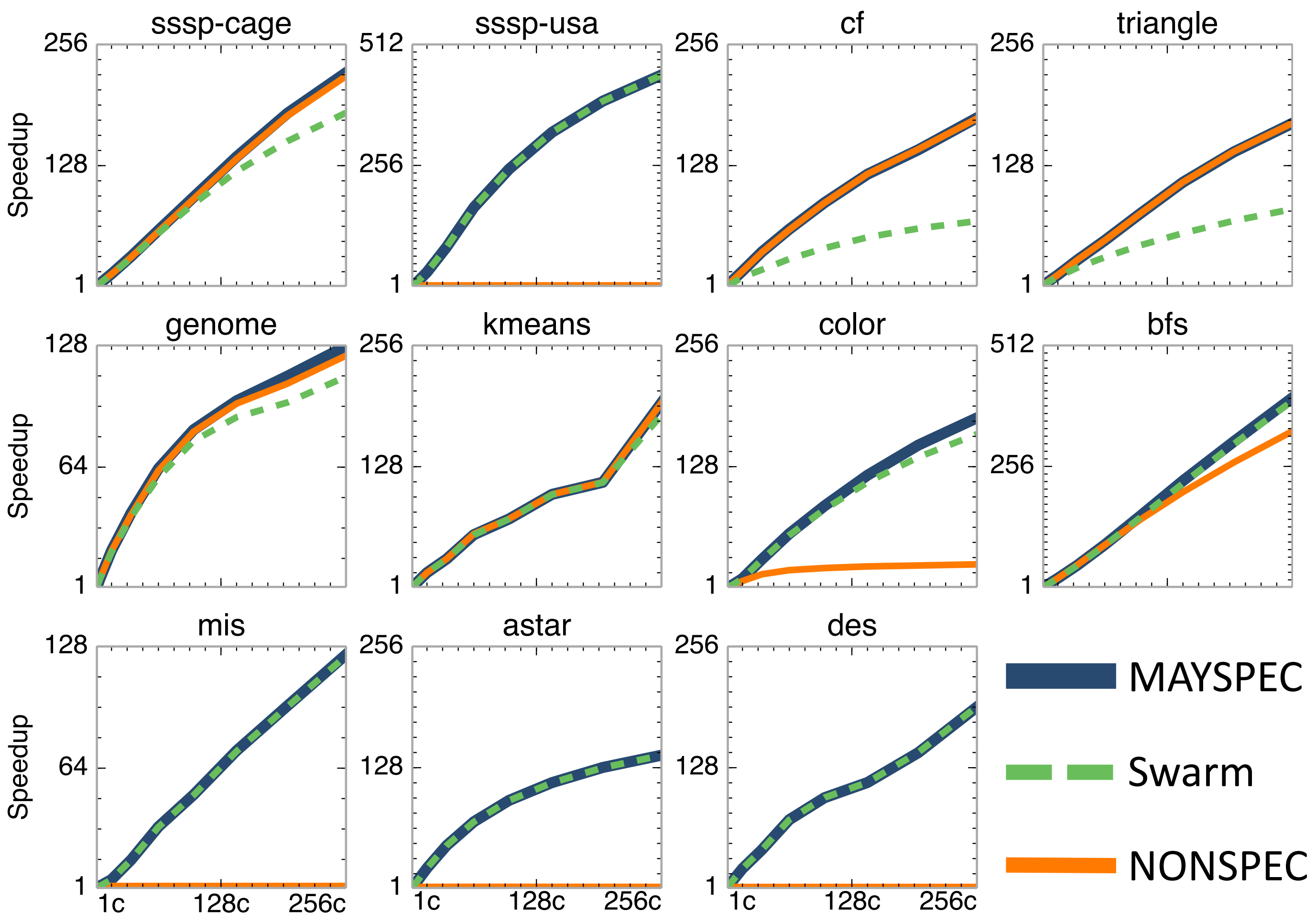
Espresso supports three task *types* that control speculation

```
void dijkstraTask(Timestamp dist, Vertex* v) {
  if (v->distance == UNSET) {
    v->distance = dist;
    for (Vertex* n : v->neighbors)
      espresso::create<MAYSPEC>(
          dijkstraTask,
          dist + weight(v, n),
          n->id,
          n);
  }
```



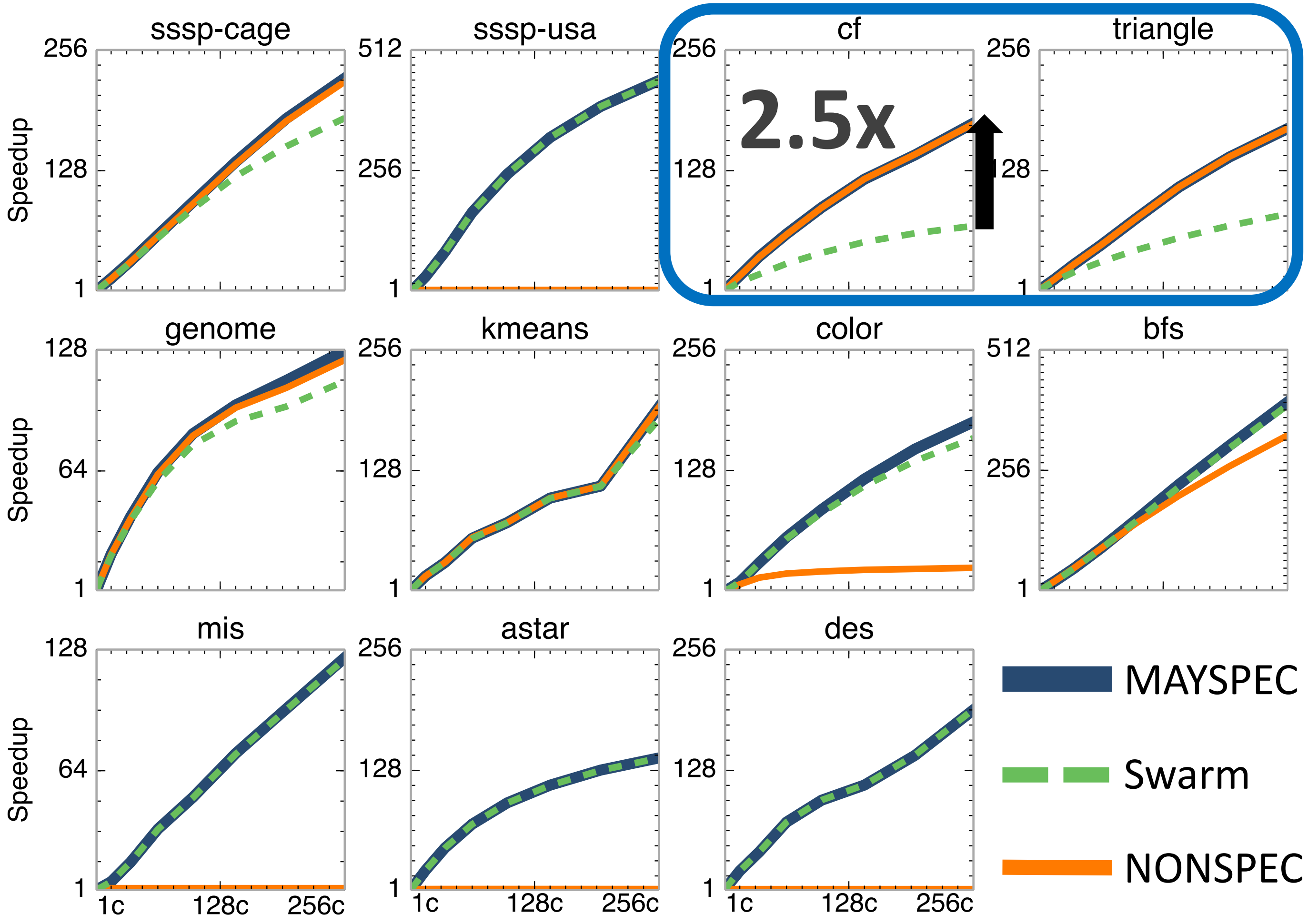
MAYSPEC lets the system decide whether to speculate

Espresso improves efficiency and programmability



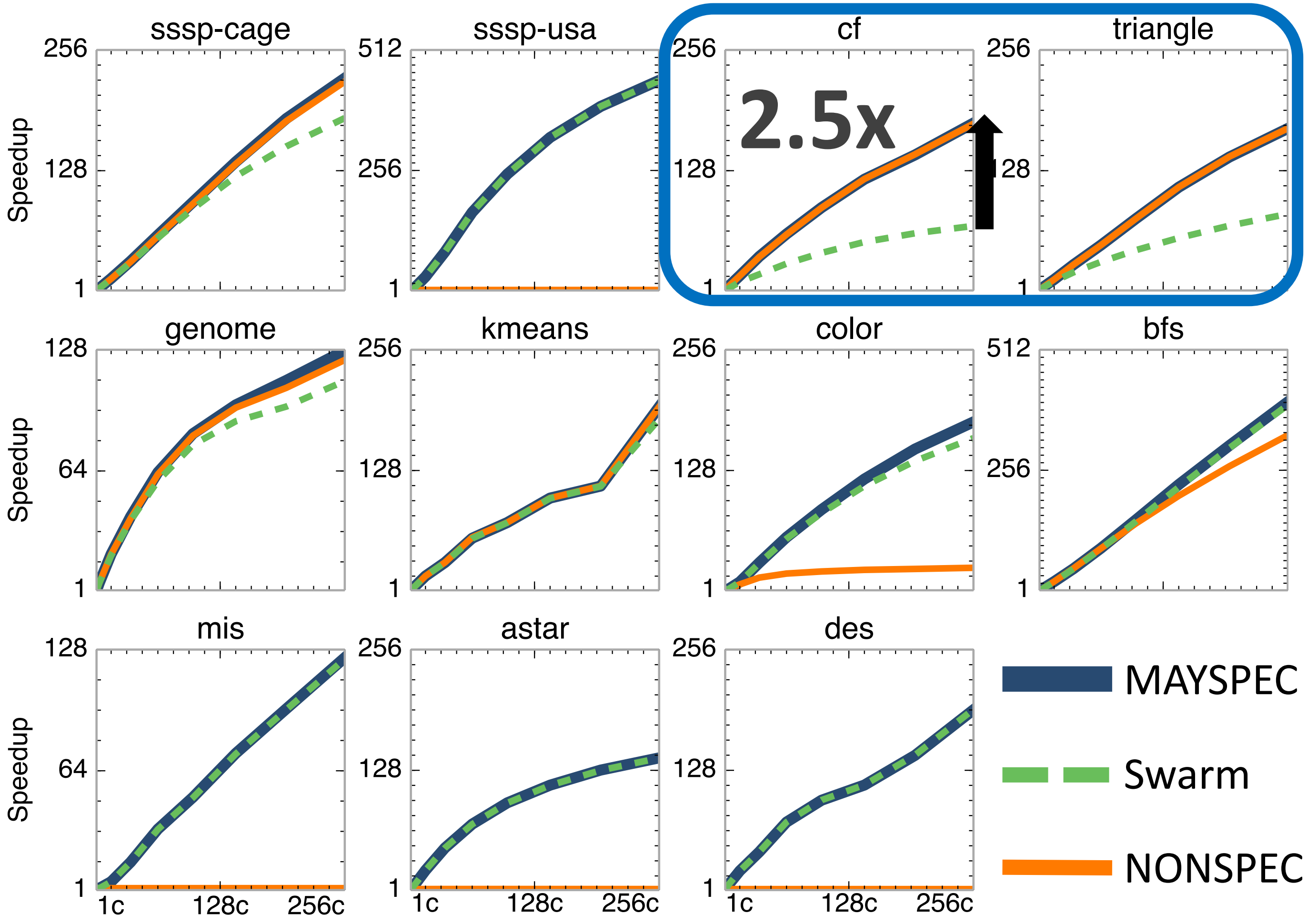
MAYSPEC allows programmers to exploit the best of speculative and non-speculative parallelism

Espresso improves efficiency and programmability



MAYSPEC allows programmers to exploit the best of speculative and non-speculative parallelism

Espresso improves efficiency and programmability



MAYSPEC allows programmers to exploit the best of speculative and non-speculative parallelism

MAYSPEC: 198x
Swarm: 162x
22%
NONSPEC: 29x gmean
6.9x

Please see the paper for more details!

Microarchitectural details

Interactions between speculative and non-speculative tasks:

- How are conflicts detected and resolved?
- How do timestamps-as-barriers affect the ordered commit protocol?

Espresso exception model

Additional results analysis

Capsules

ENABLING SOFTWARE-MANAGED SPECULATION WITH ORDERED PARALLELISM

Some actions should bypass HW speculation

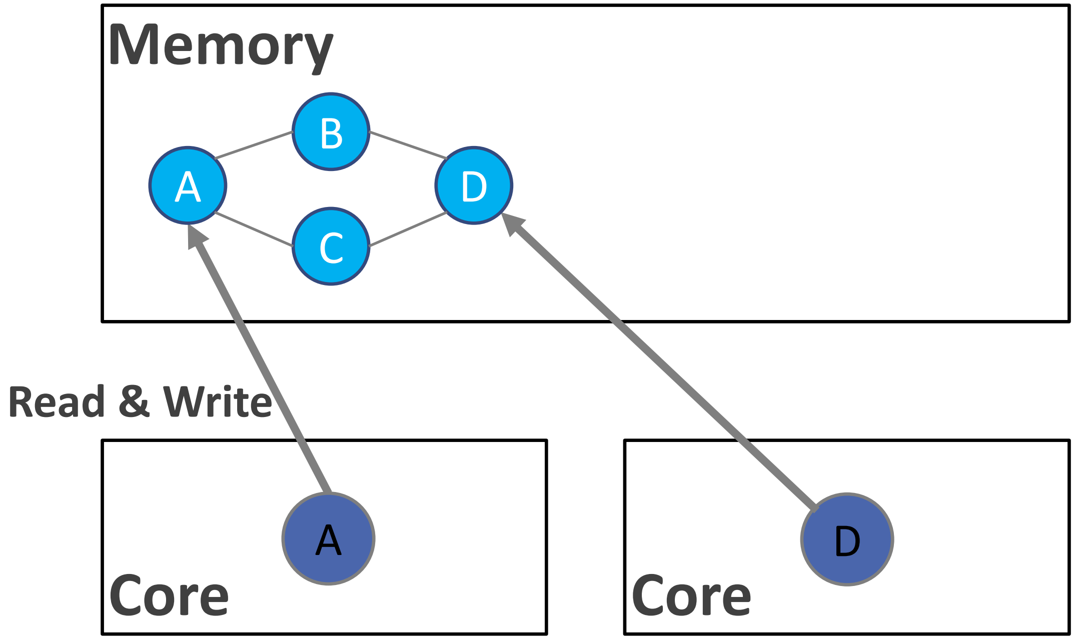
Discrete event simulation (DES) needs speculation to scale

DES also allocates memory within tasks

Some actions should bypass HW speculation

Discrete event simulation (DES) needs speculation to scale

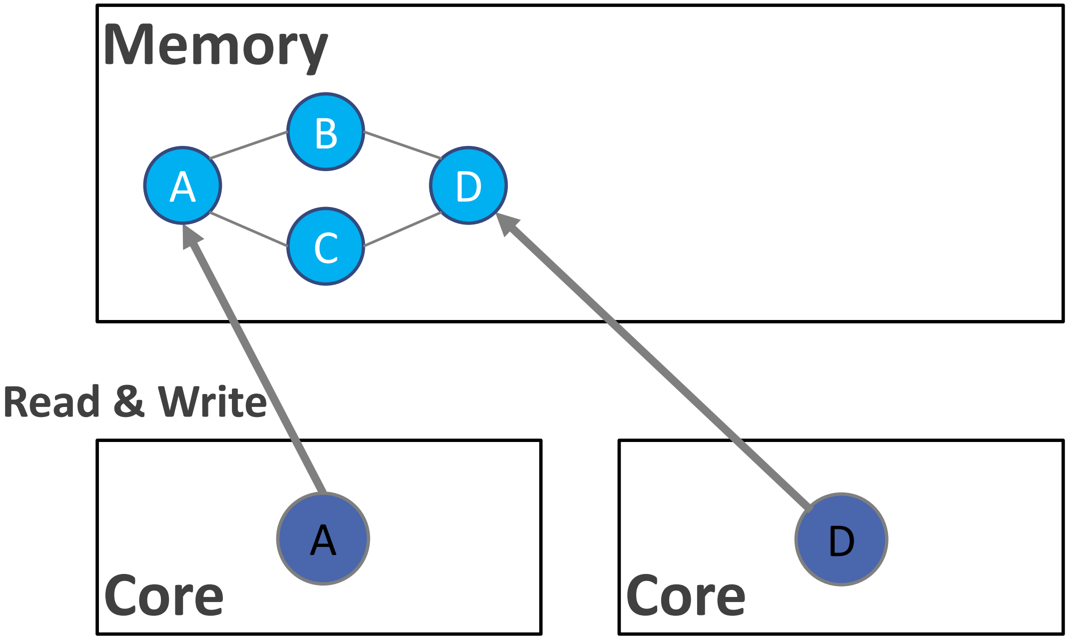
DES also allocates memory within tasks

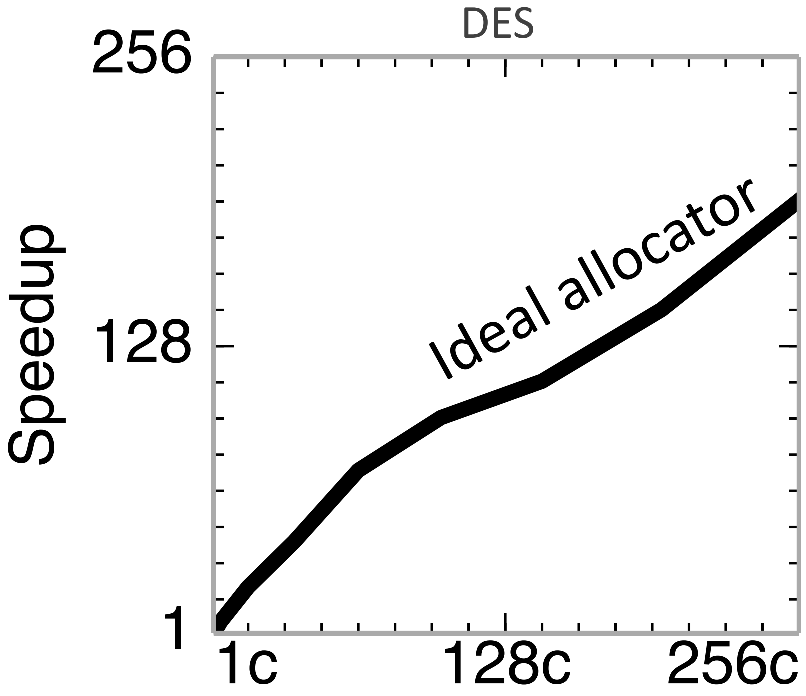


Some actions should bypass HW speculation

Discrete event simulation (DES) needs speculation to scale

DES also allocates memory within tasks

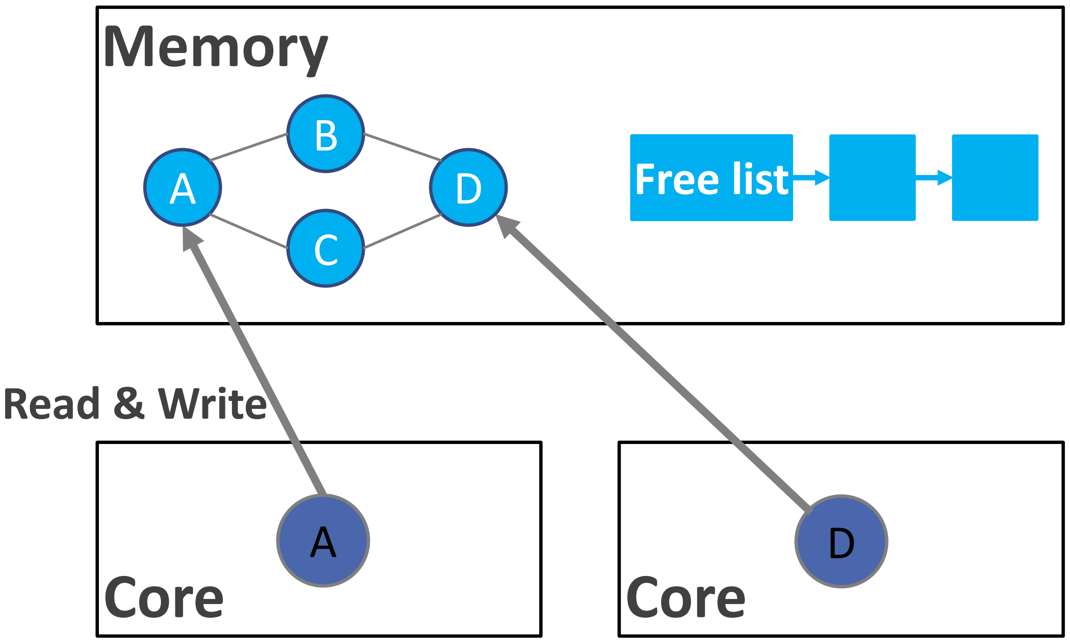


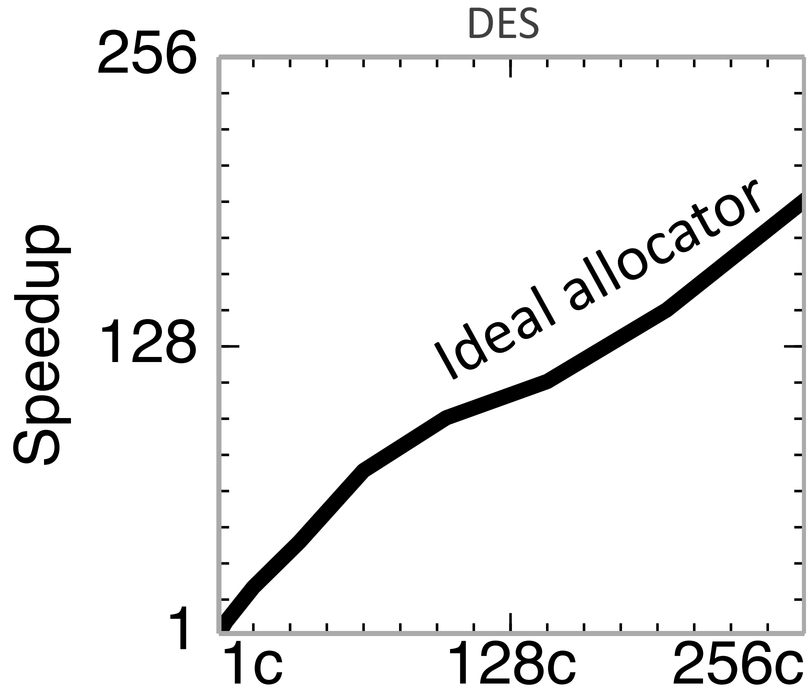


Some actions should bypass HW speculation

Discrete event simulation (DES) needs speculation to scale

DES also allocates memory within tasks

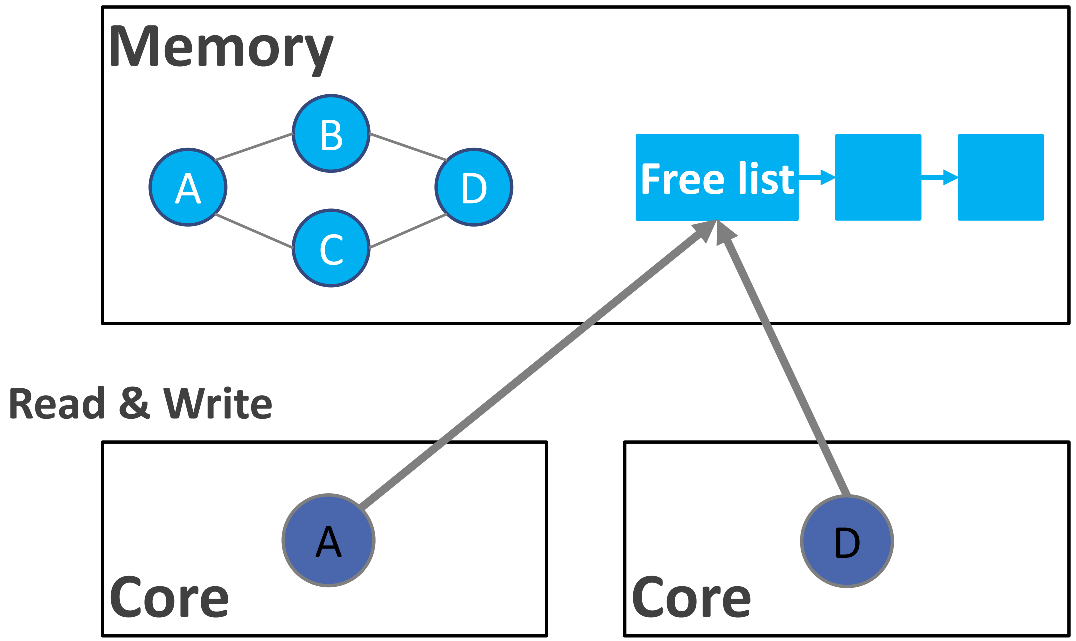


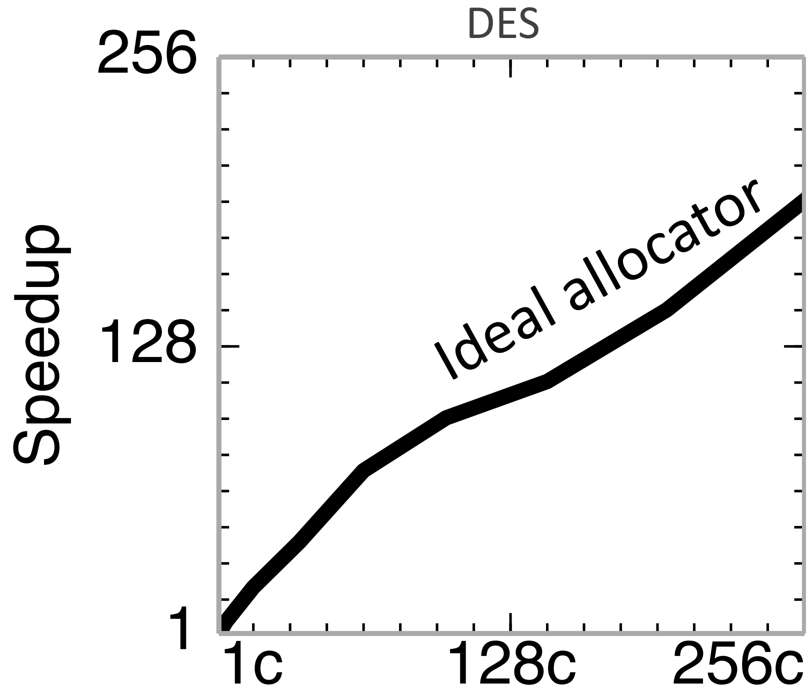


Some actions should bypass HW speculation

Discrete event simulation (DES) needs speculation to scale

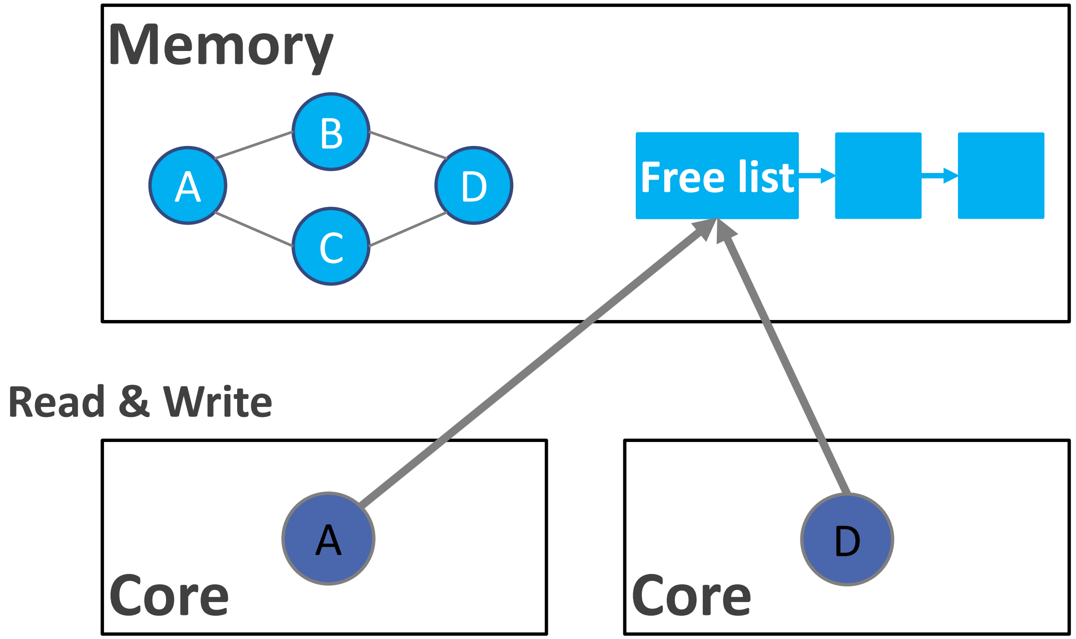
DES also allocates memory within tasks

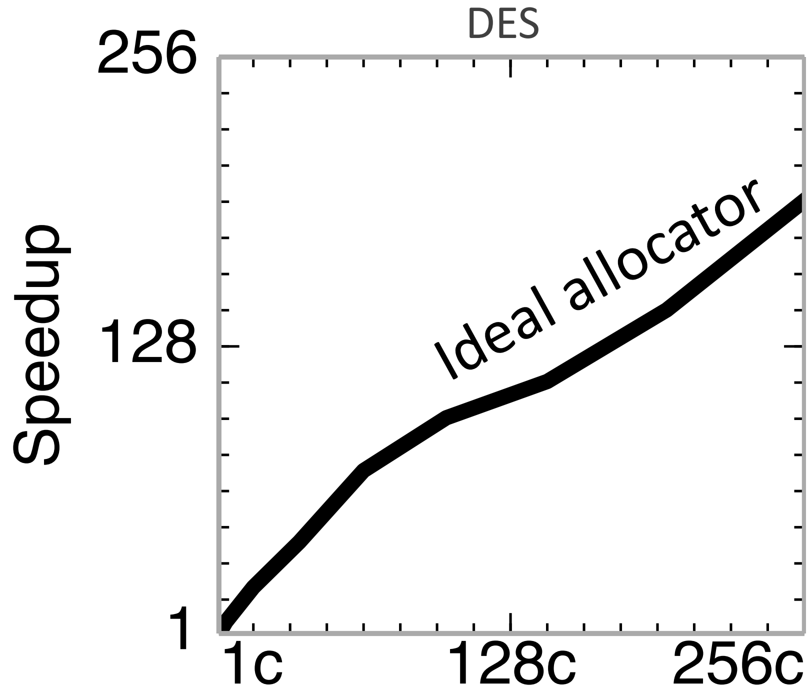




Some actions should bypass HW speculation

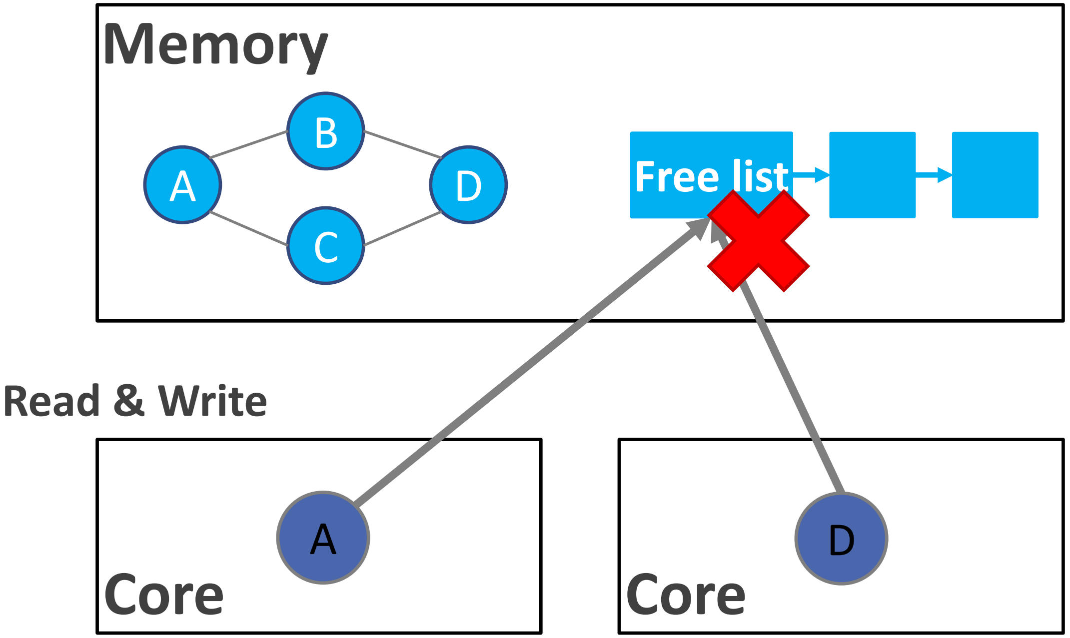
Dependences on allocator metadata cause aborts among otherwise independent tasks

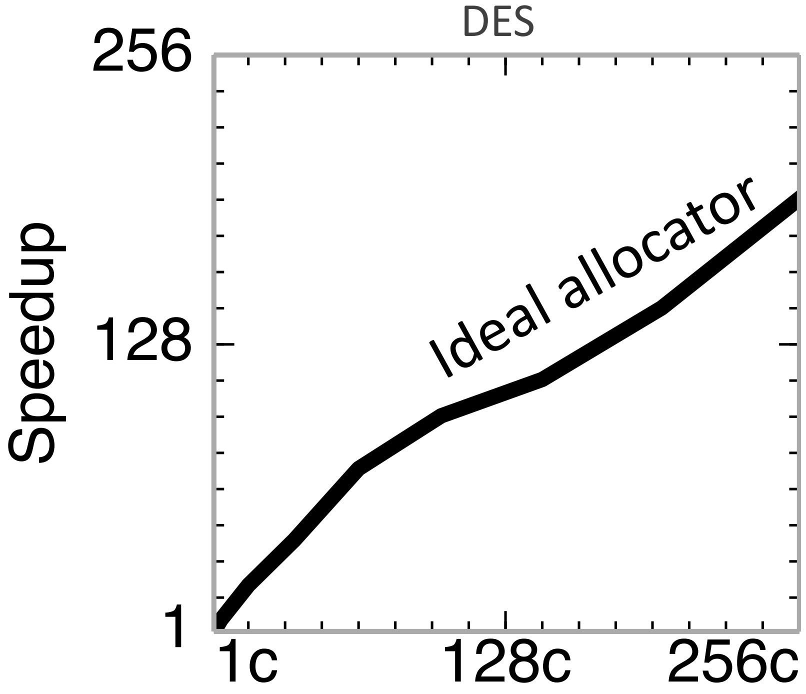




Some actions should bypass HW speculation

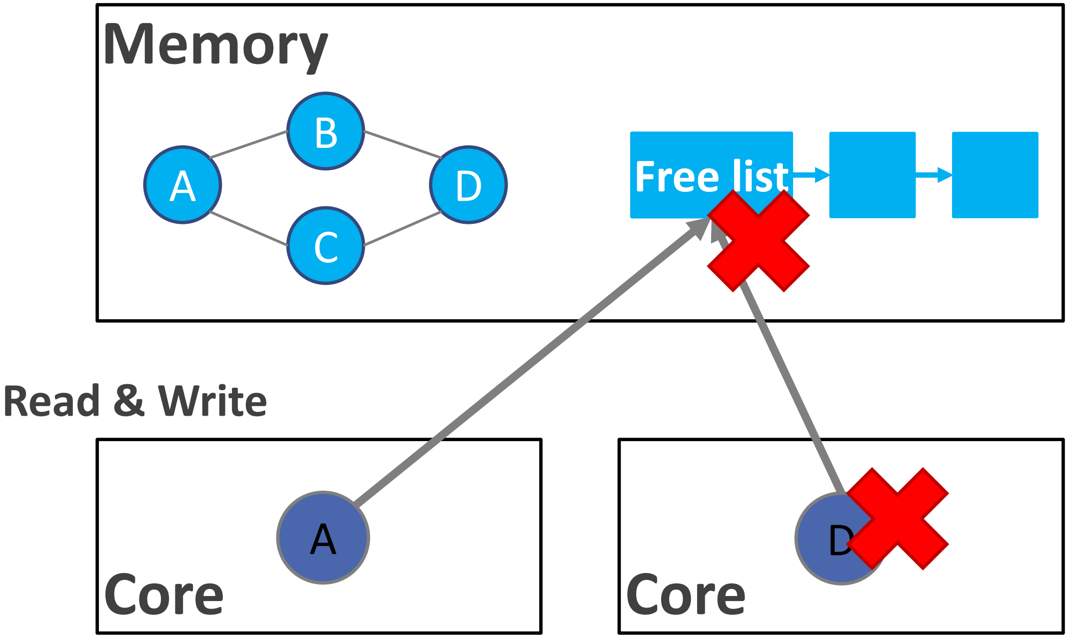
Dependences on allocator metadata cause aborts among otherwise independent tasks

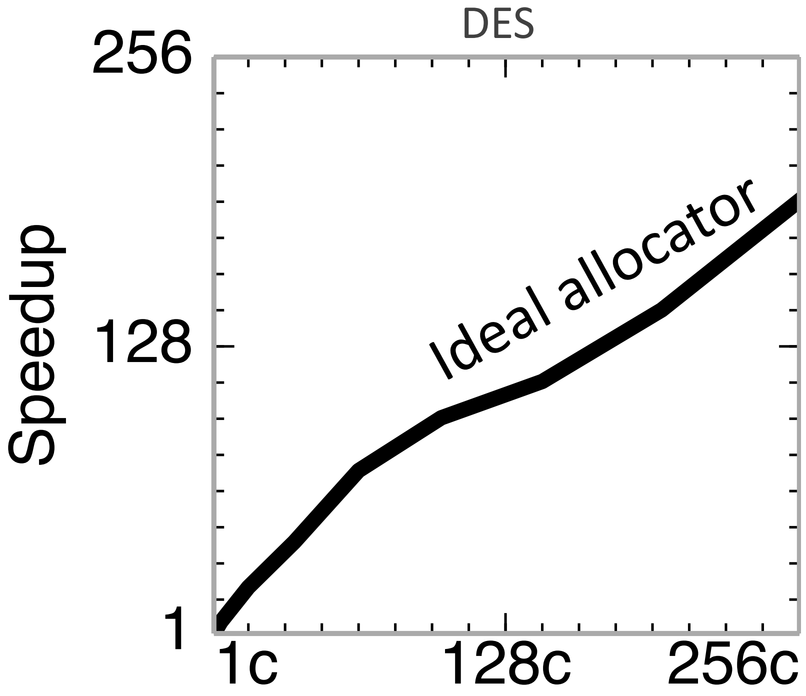




Some actions should bypass HW speculation

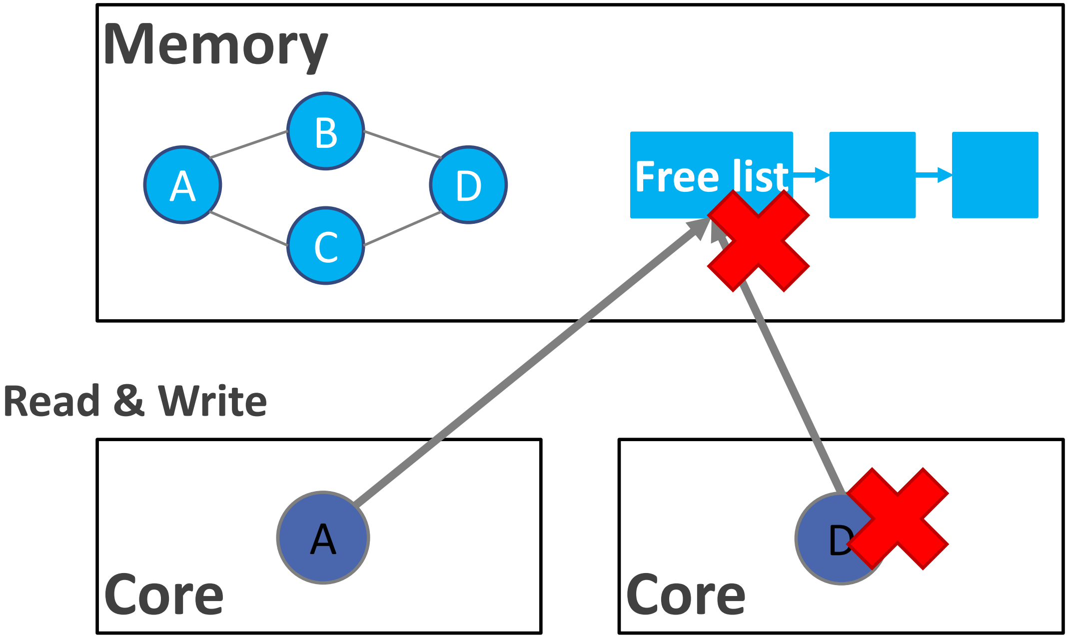
Dependences on allocator metadata cause aborts among otherwise independent tasks

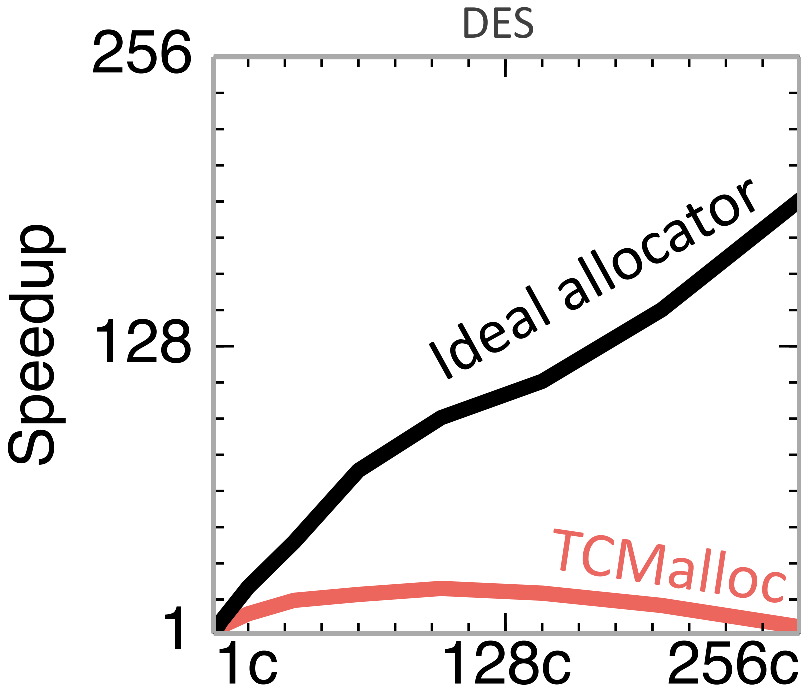




Some actions should bypass HW speculation

Dependences on allocator metadata cause aborts among otherwise independent tasks





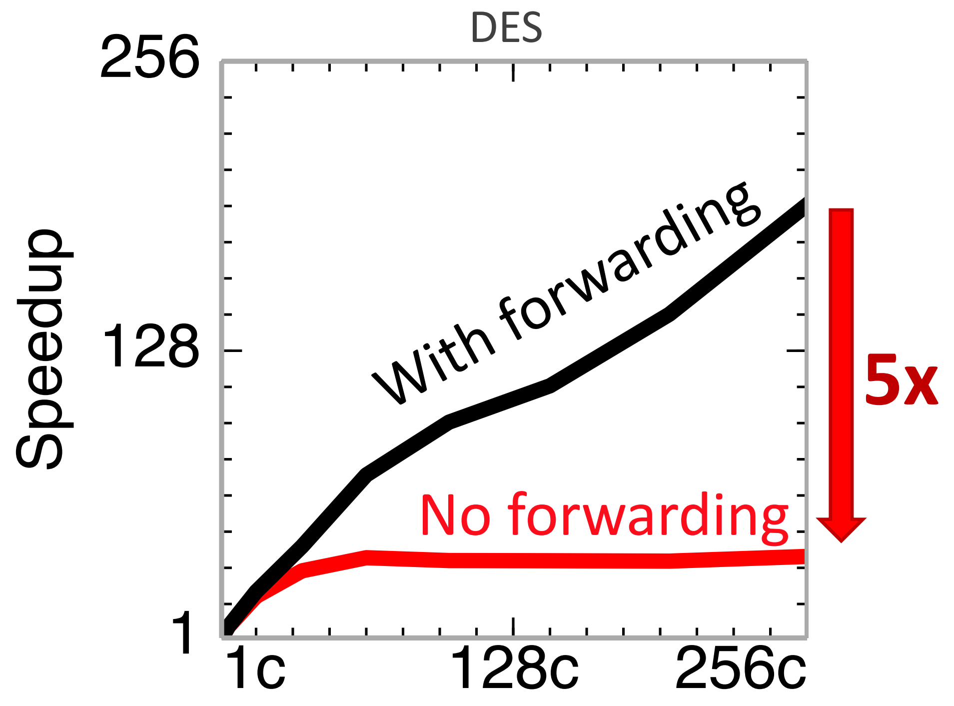
Disable hardware speculation [Moravan, ASPLOS'06]? **Speculative data forwarding** creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Disable hardware speculation [Moravan, ASPLOS'06]? **Speculative data forwarding** creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

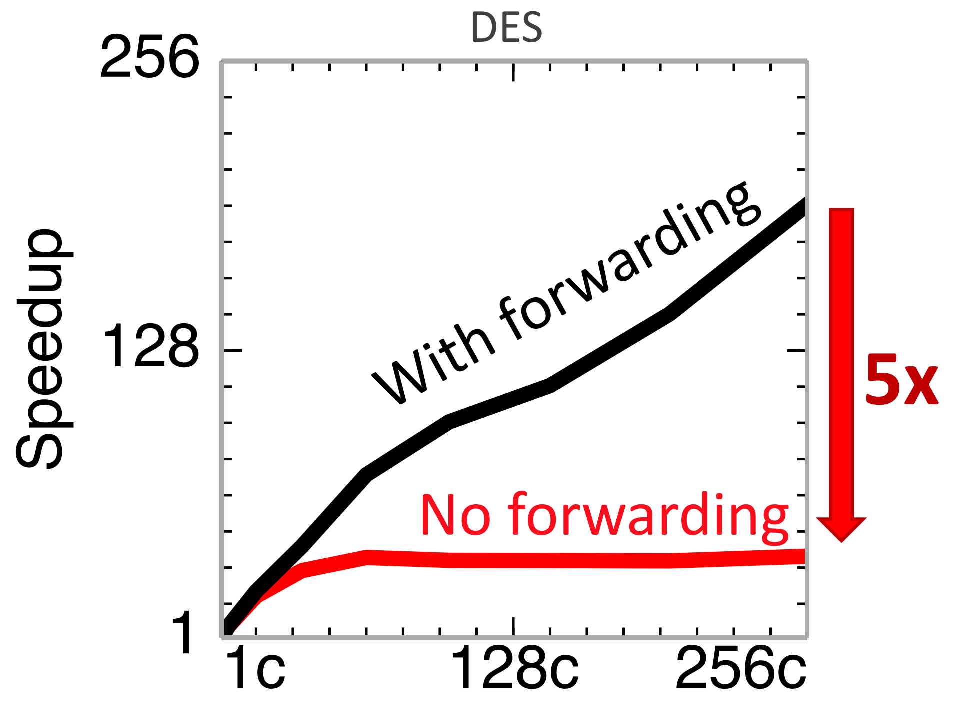
Critical for ordered parallelism



Disable hardware speculation [Moravan, ASPLOS'06]? **Speculative data forwarding** creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism

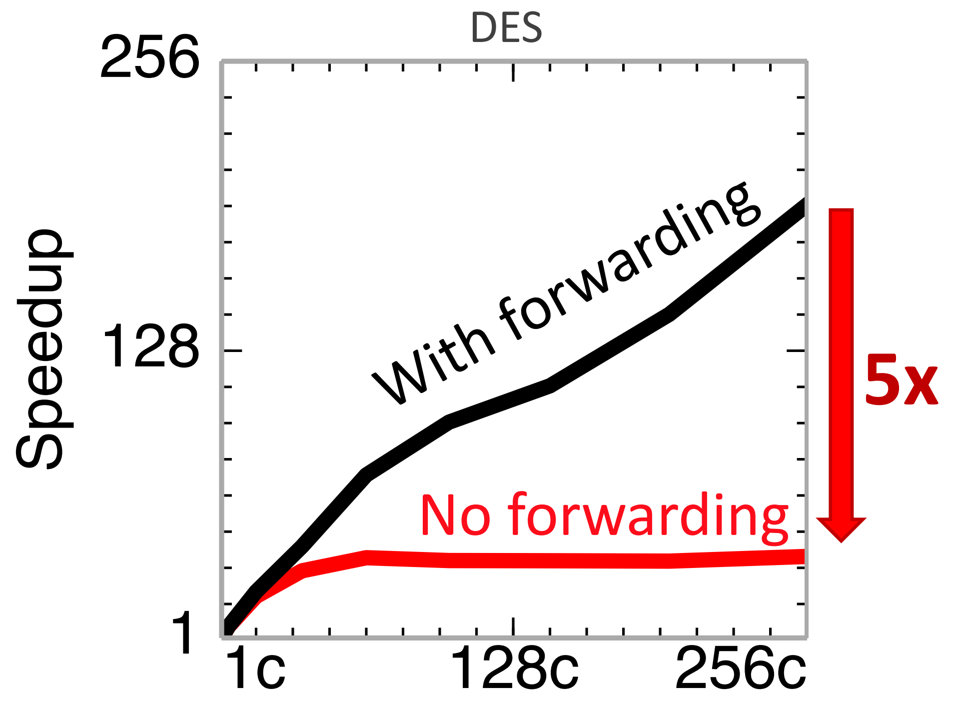


Can cause tasks to lose integrity 😈

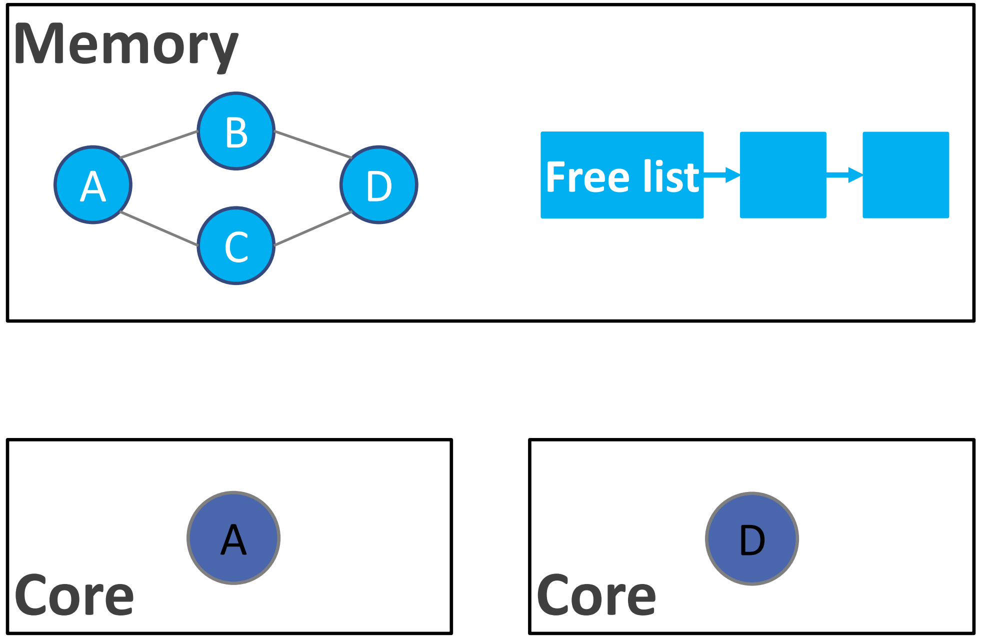
Disable hardware speculation [Moravan, ASPLOS'06]? **Speculative data forwarding** creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



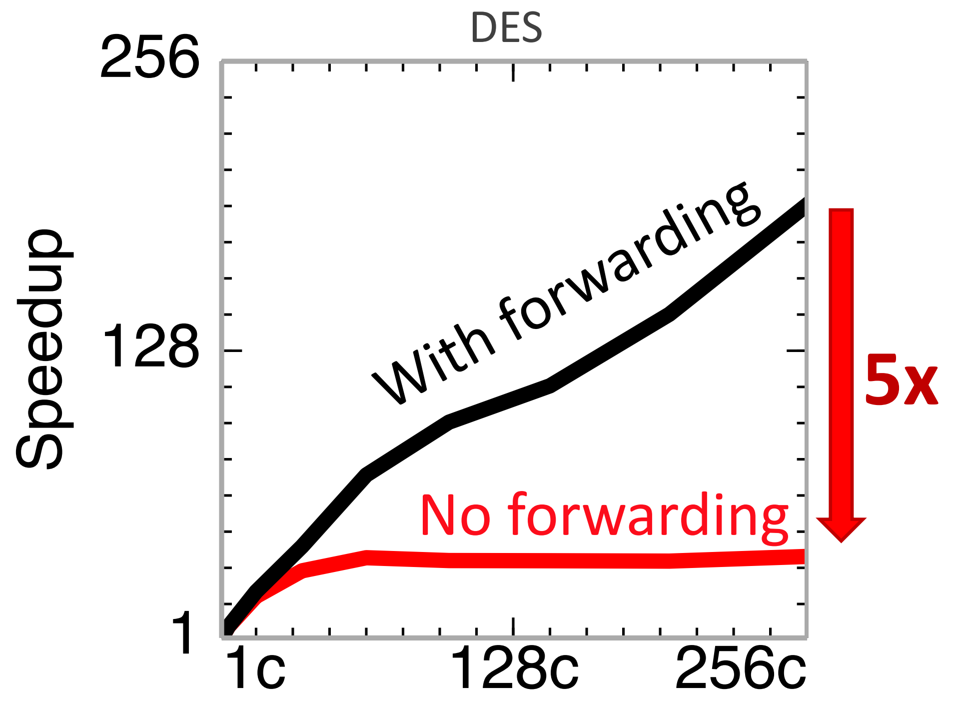
Can cause tasks to lose integrity 👿



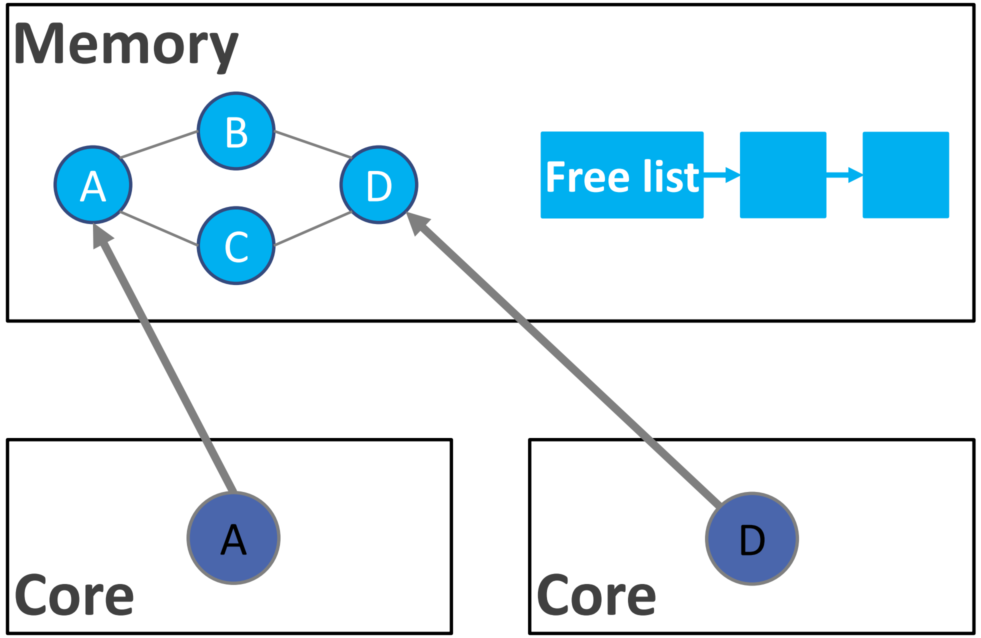
Disable hardware speculation [Moravan, ASPLOS'06]? Speculative data forwarding creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



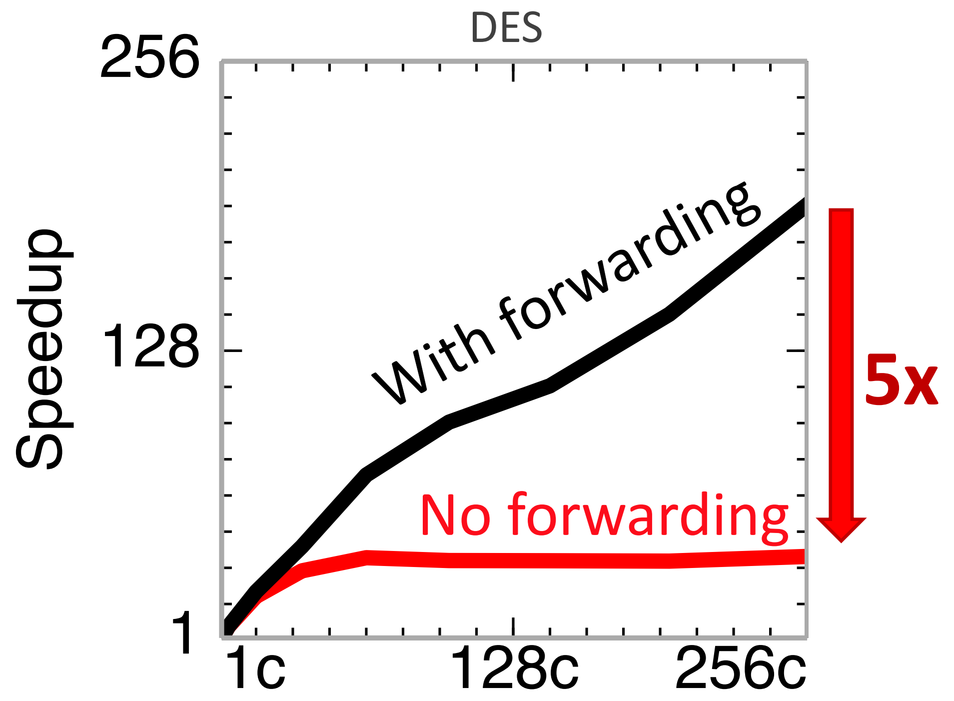
Can cause tasks to lose integrity 👿



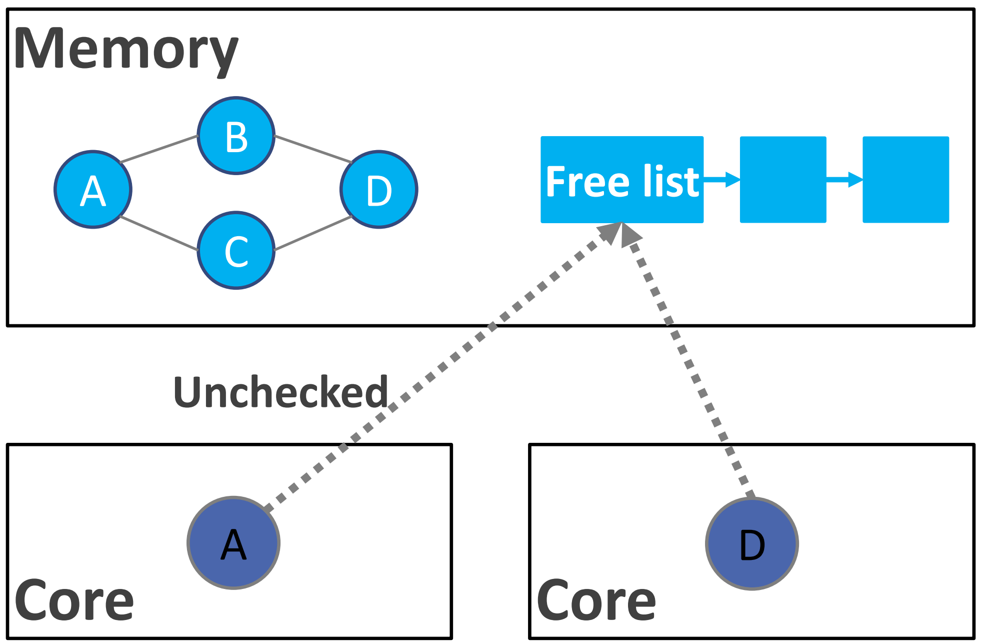
Disable hardware speculation [Moravan, ASPLOS'06]? **Speculative data forwarding** creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



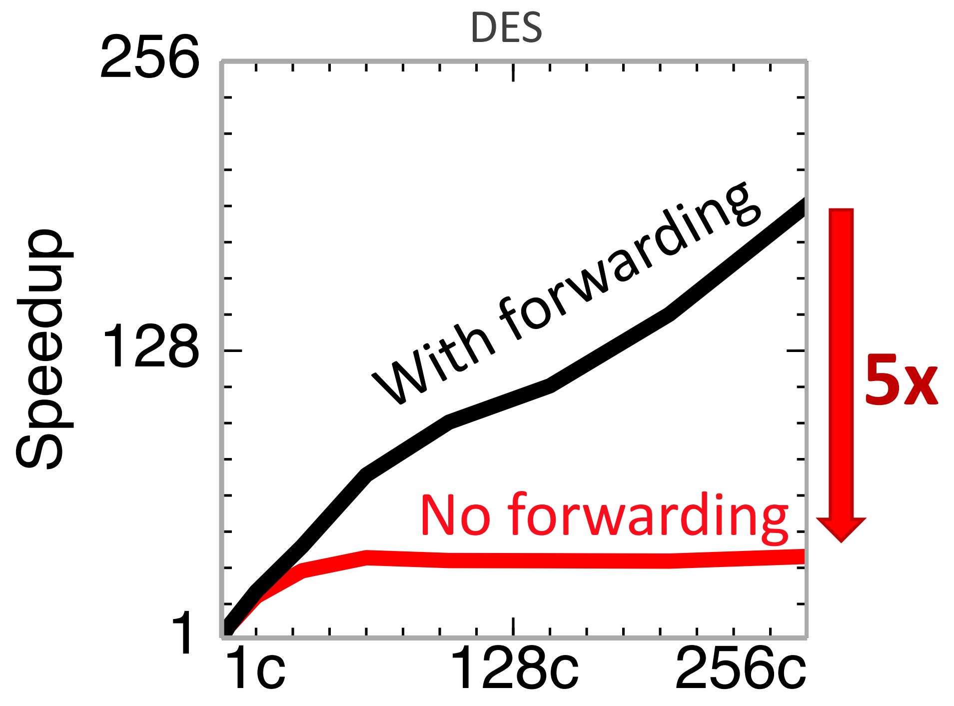
Can cause tasks to lose integrity 👿



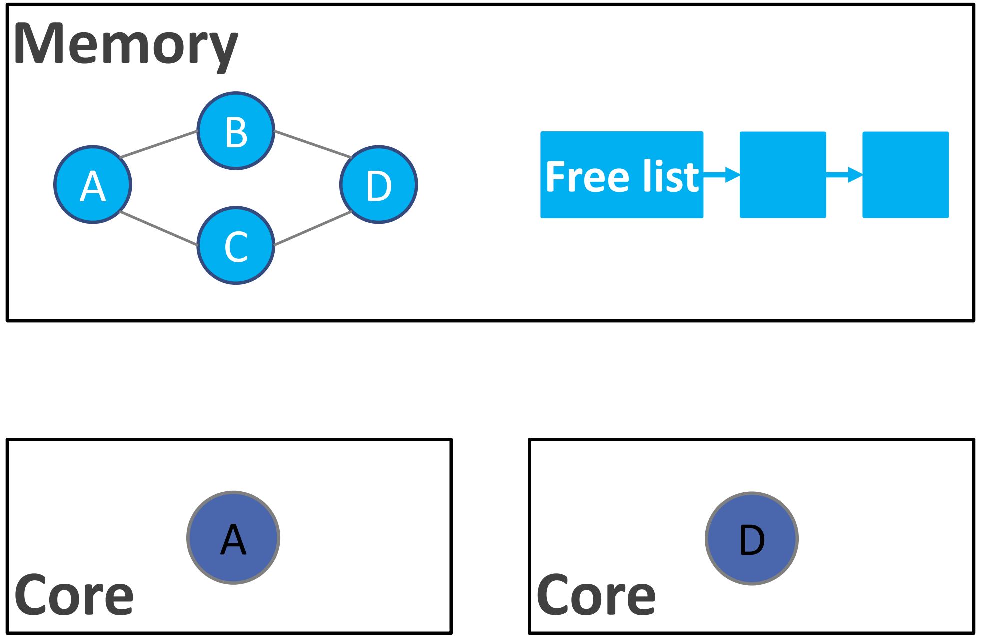
Disable hardware speculation [Moravan, ASPLOS'06]? Speculative data forwarding creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



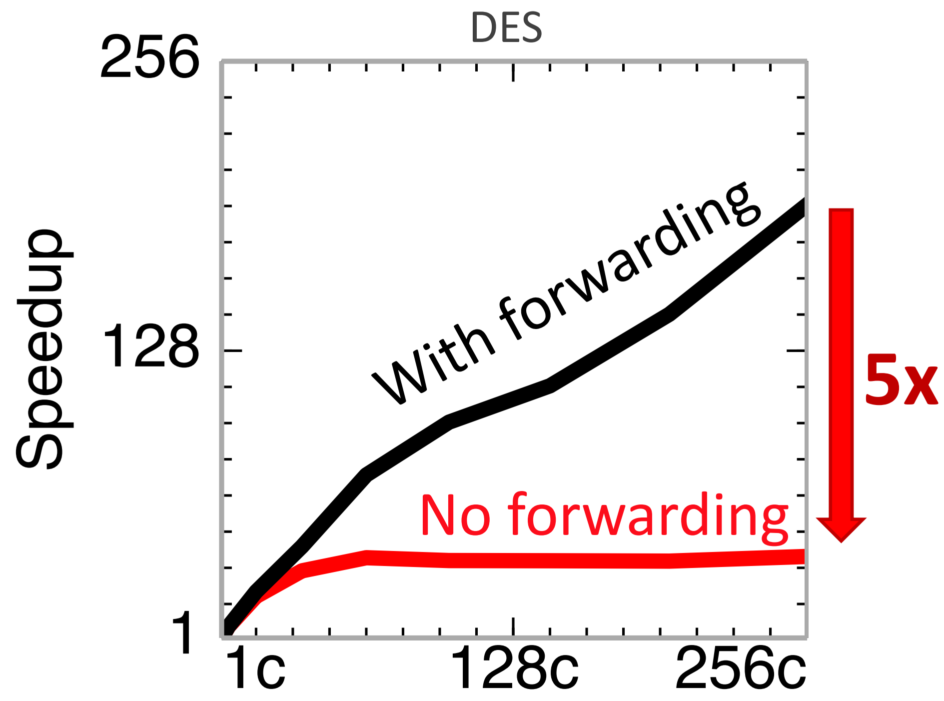
Can cause tasks to lose integrity 👿



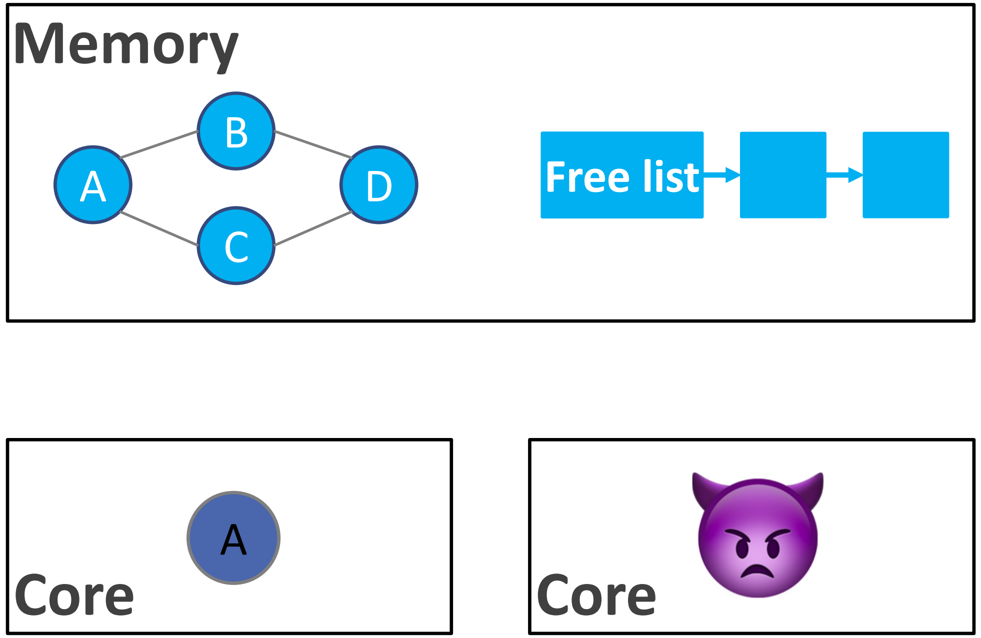
Disable hardware speculation [Moravan, ASPLOS'06]?
Speculative data forwarding creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



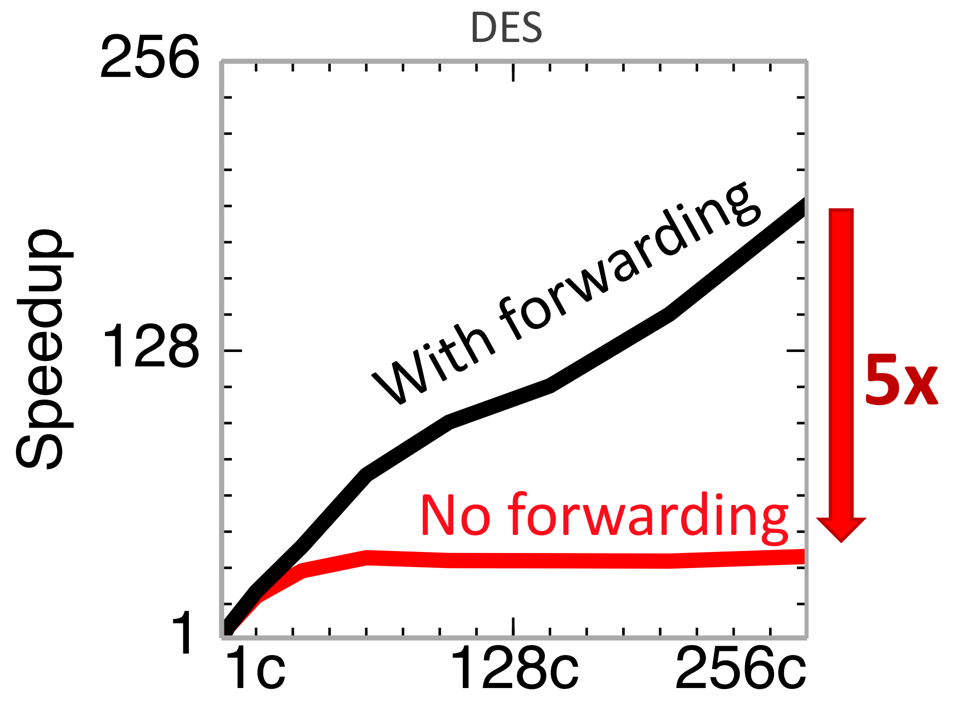
Can cause tasks to lose integrity 😈



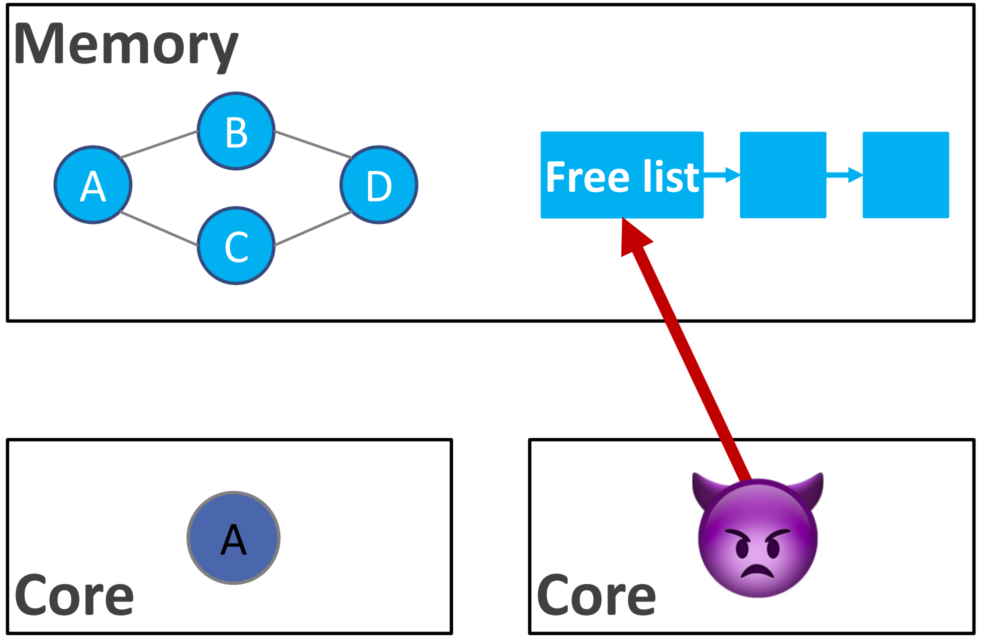
Disable hardware speculation [Moravan, ASPLOS'06]? **Speculative data forwarding** creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



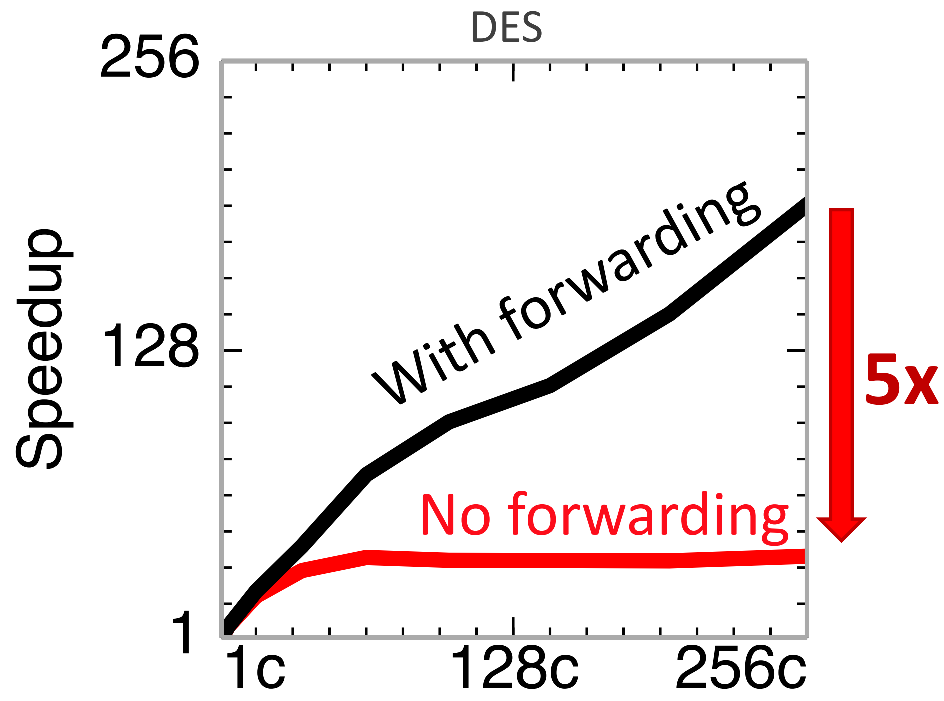
Can cause tasks to lose integrity 😈



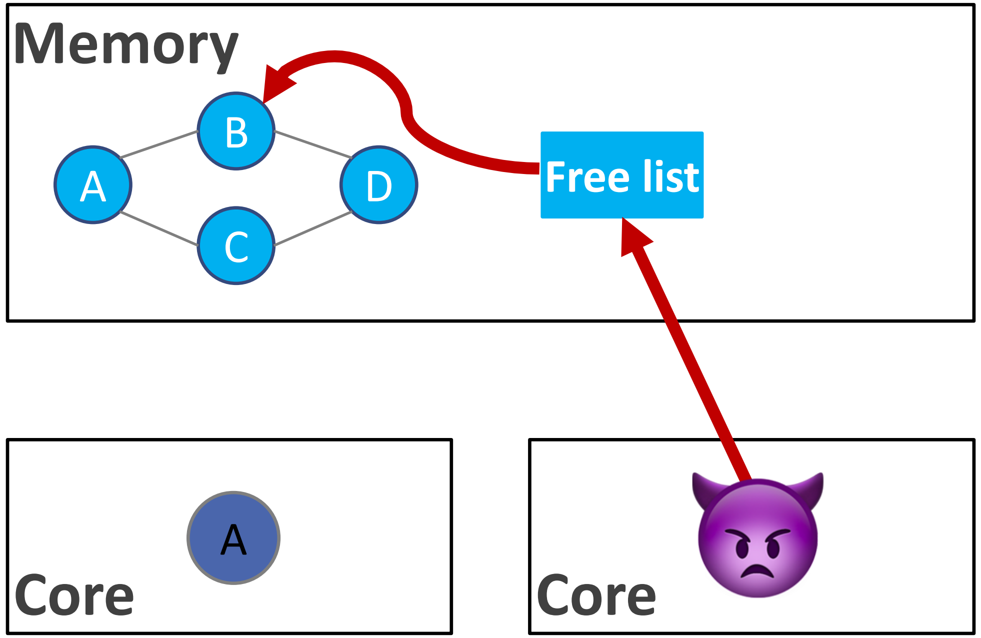
Disable hardware speculation [Moravan, ASPLOS'06]?
Speculative data forwarding creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



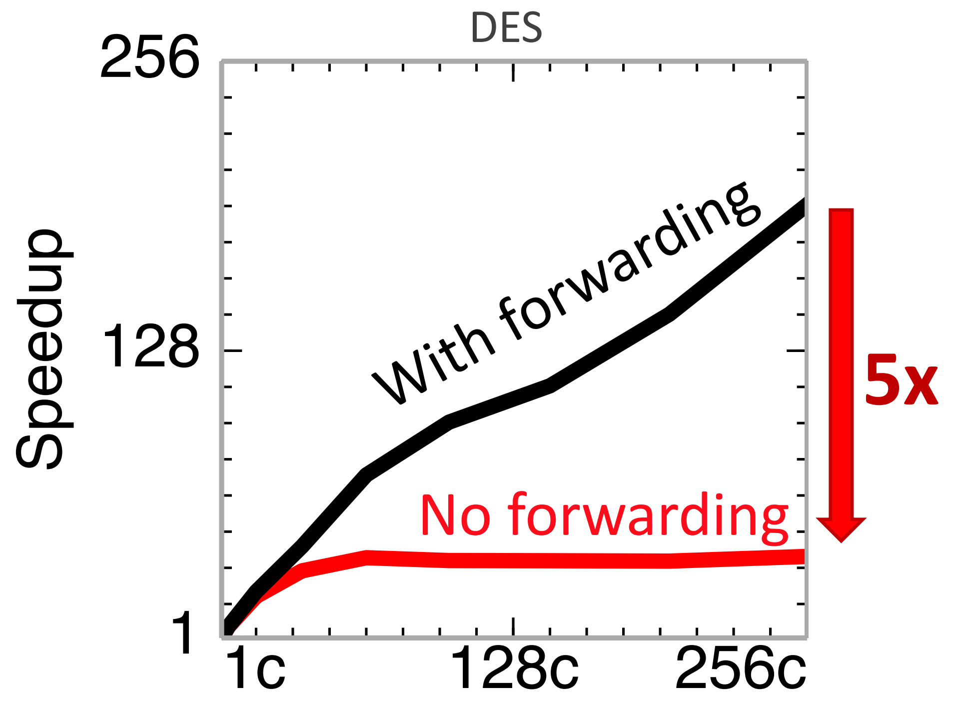
Can cause tasks to lose integrity 😈



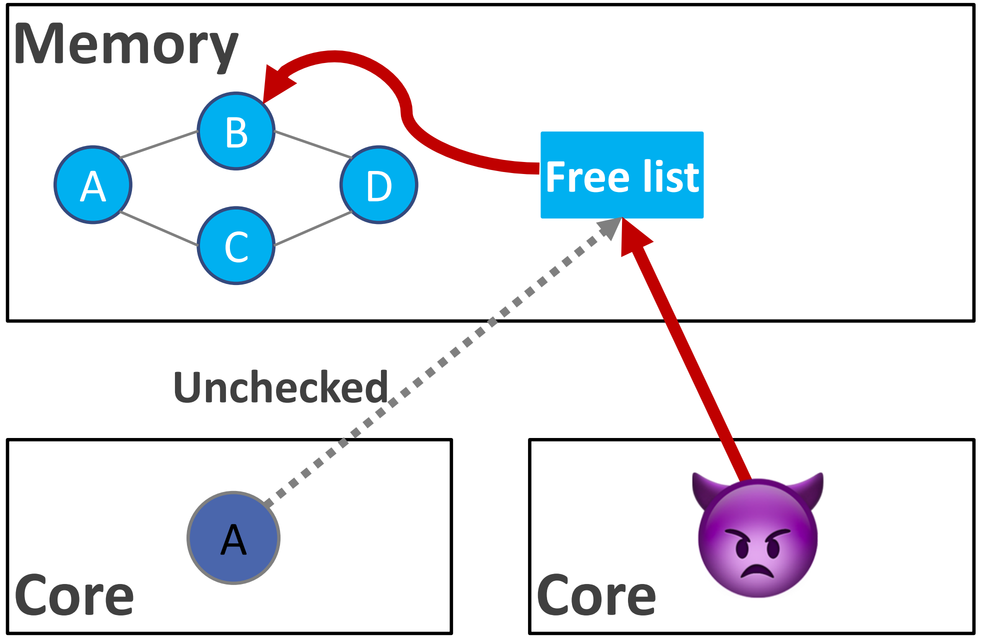
Disable hardware speculation [Moravan, ASPLOS'06]? **Speculative data forwarding** creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



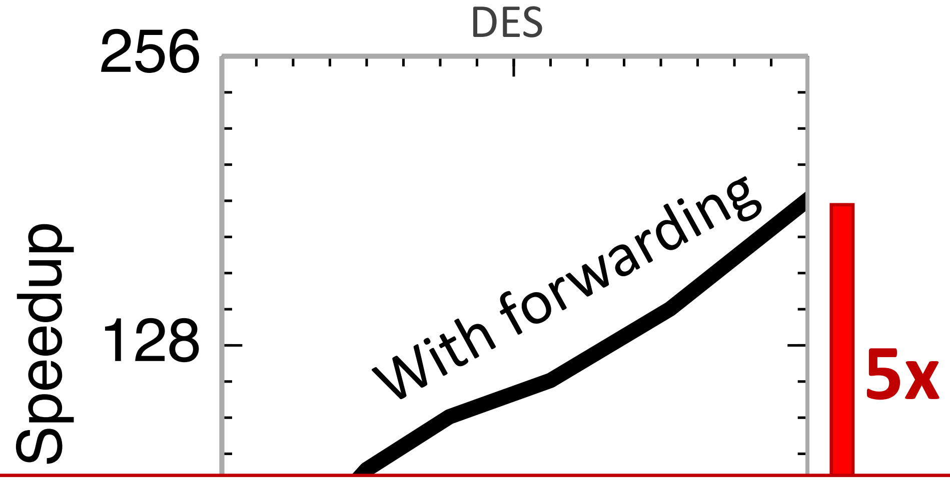
Can cause tasks to lose integrity 😈



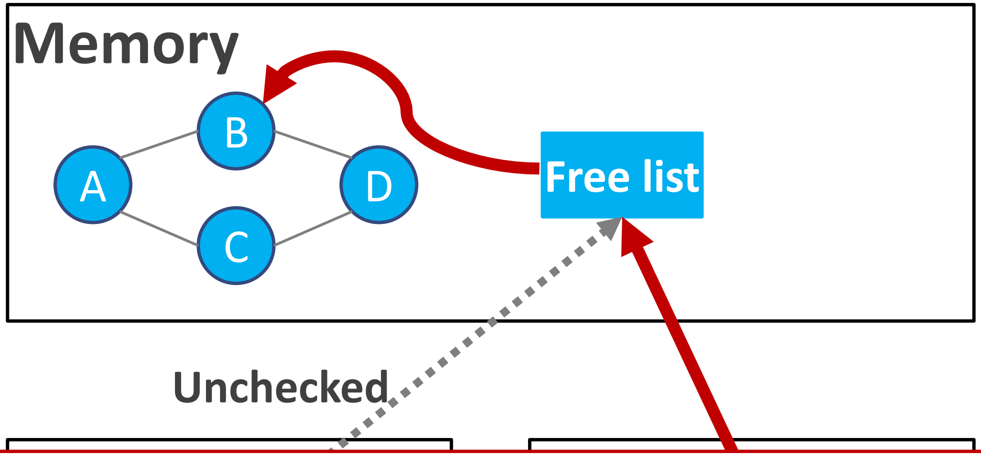
Disable hardware speculation [Moravan, ASPLOS'06]? Speculative data forwarding creates challenges

Speculative tasks can access data written by earlier, uncommitted tasks

Critical for ordered parallelism



Can cause tasks to lose integrity 👿



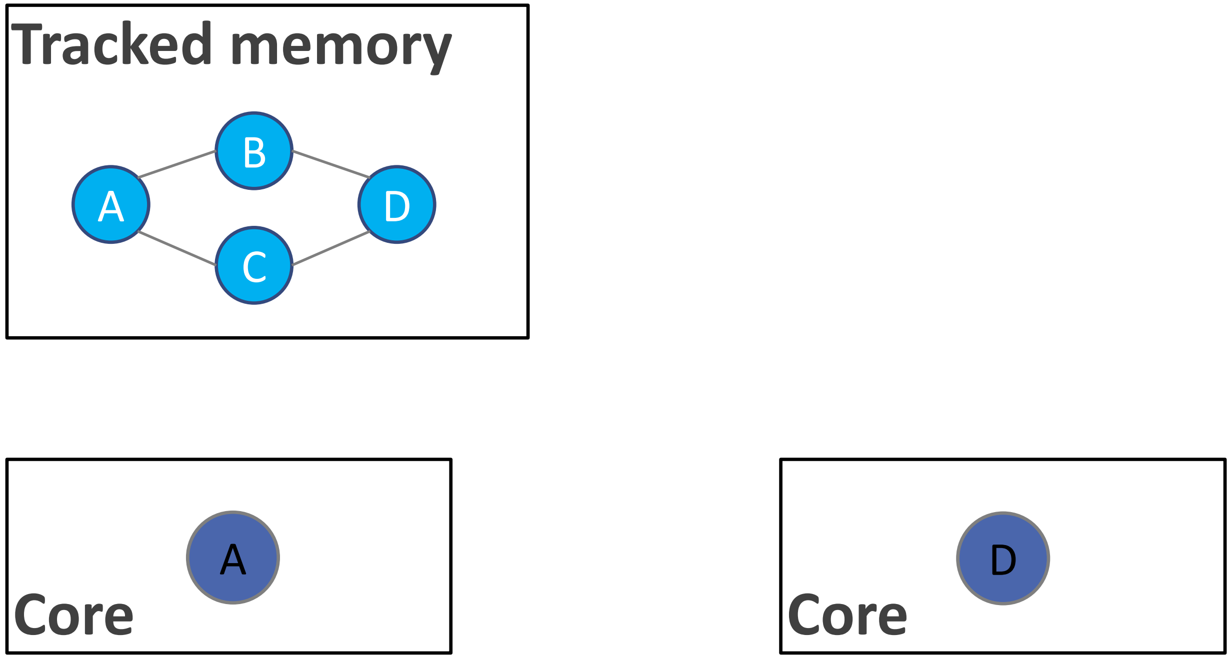
Simply disabling hardware speculation is unsafe with speculative forwarding

Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity

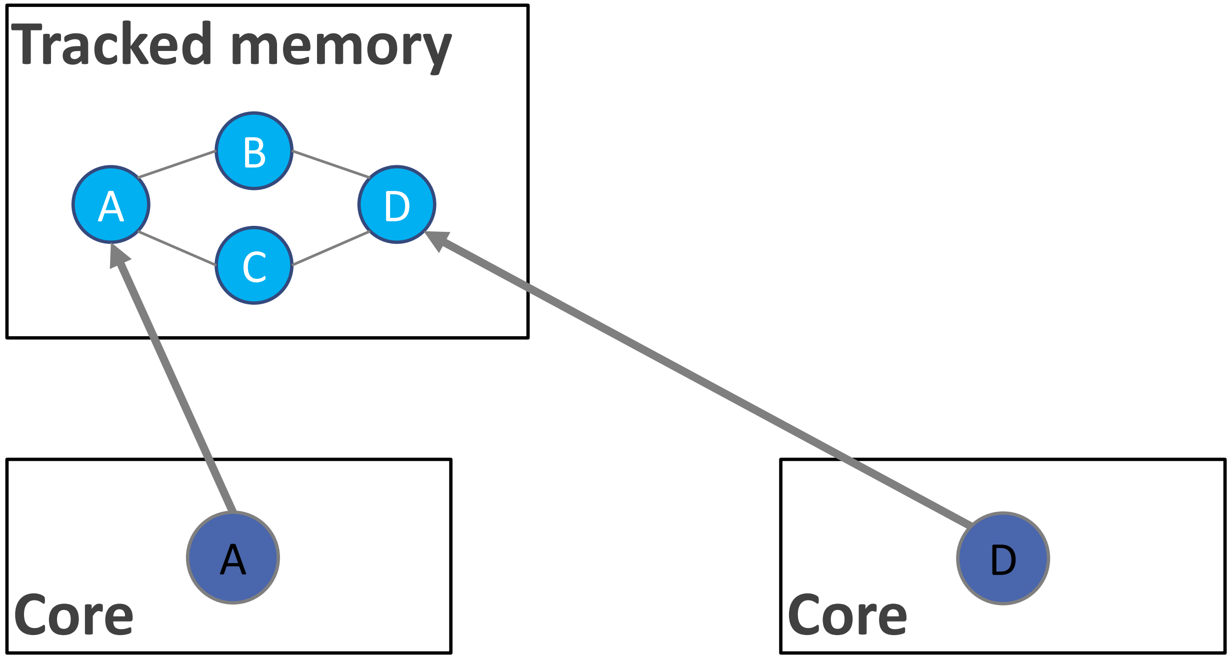
Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity



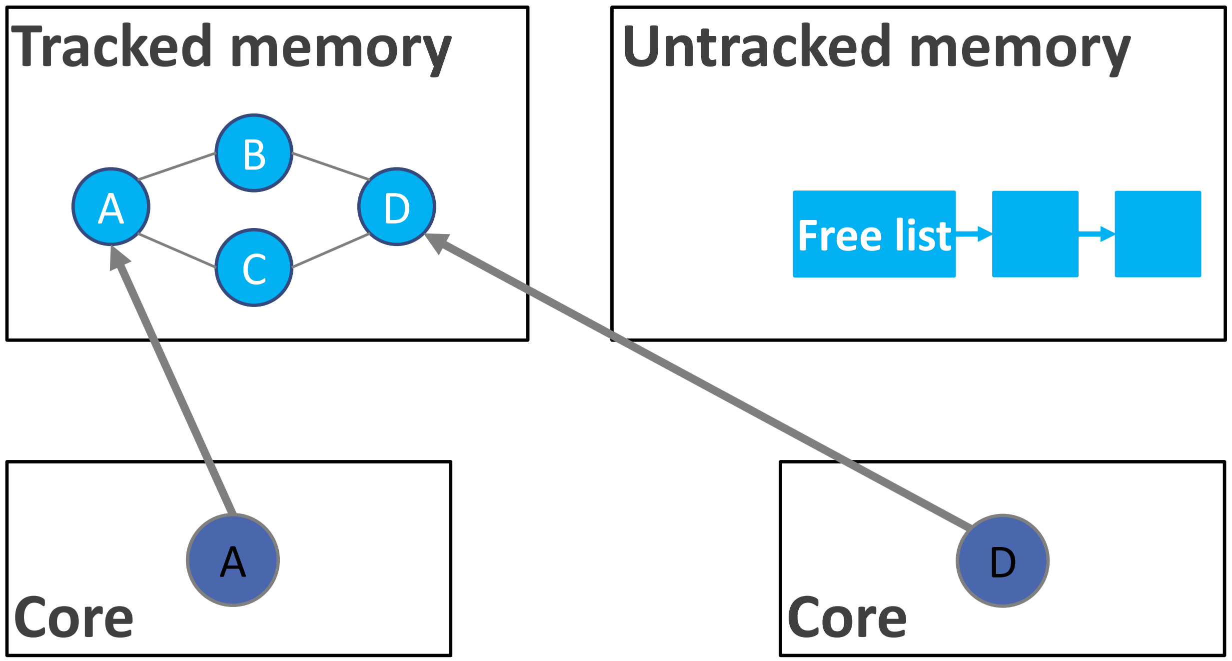
Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity



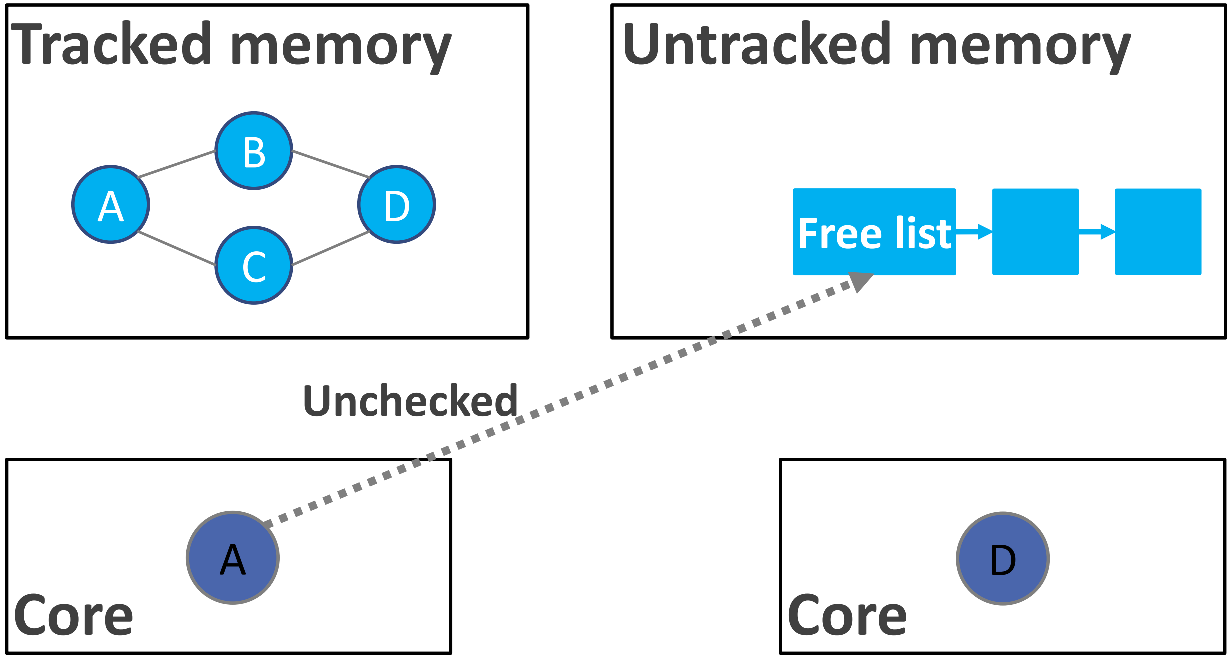
Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity



Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity

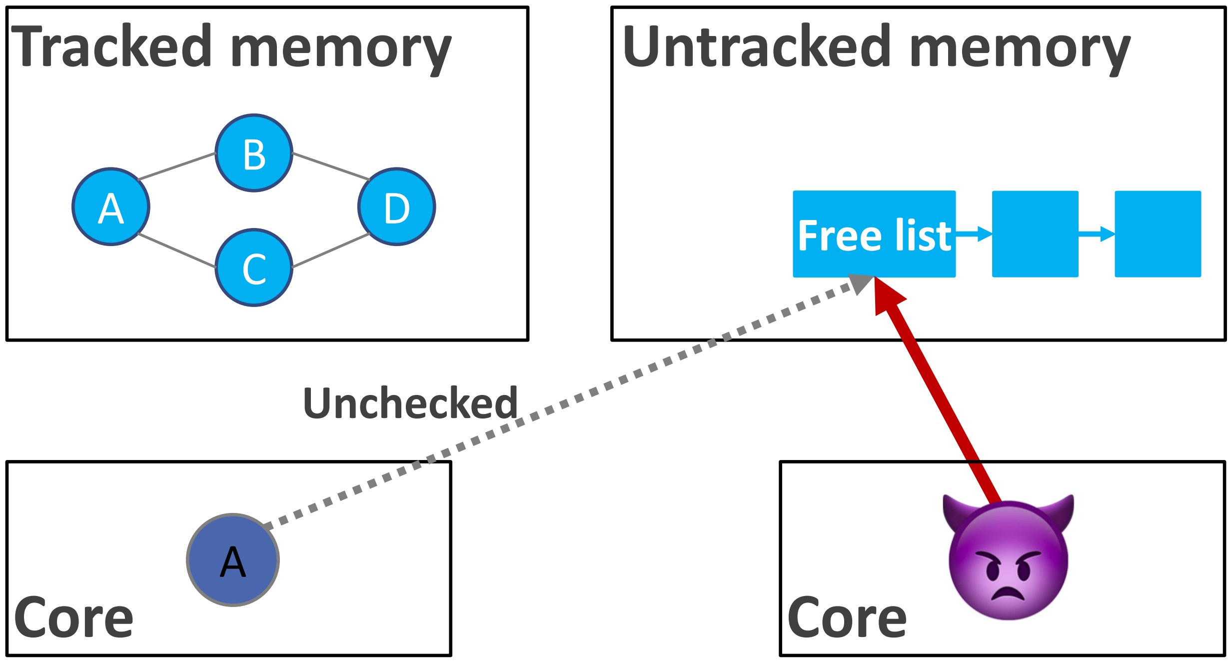


Unversioned, no conflict checks

Only accessible by
- non-speculative tasks
- speculative capsules

Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity

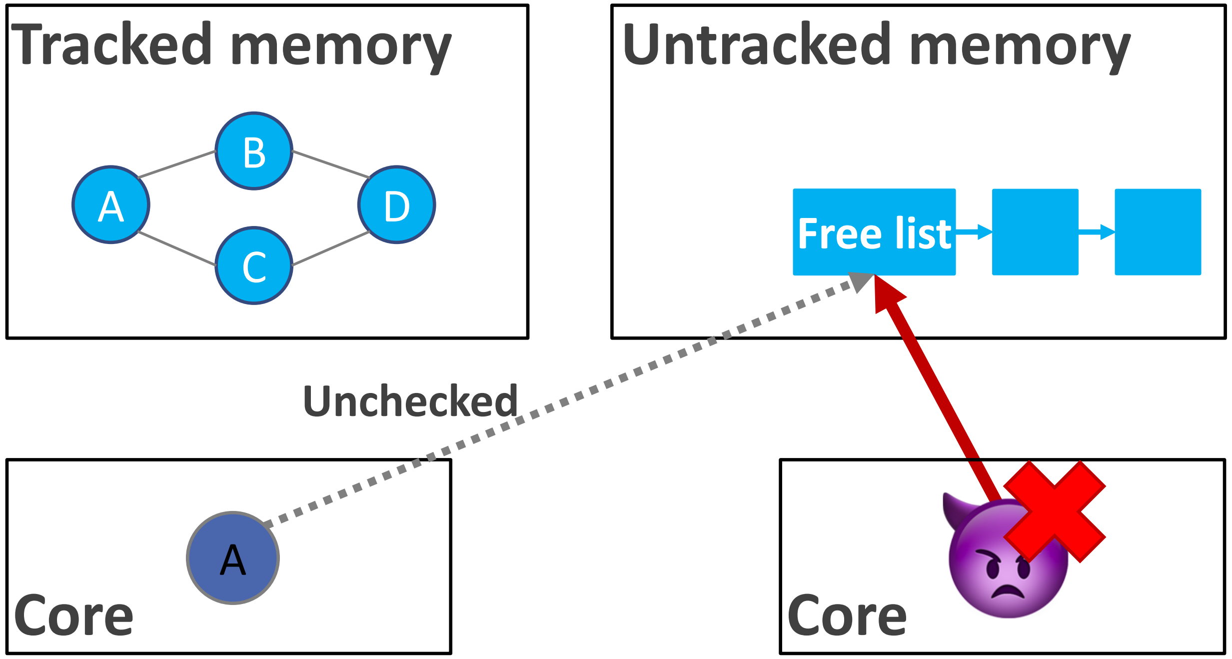


Unversioned, no conflict checks

Only accessible by
- non-speculative tasks
- speculative capsules

Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity

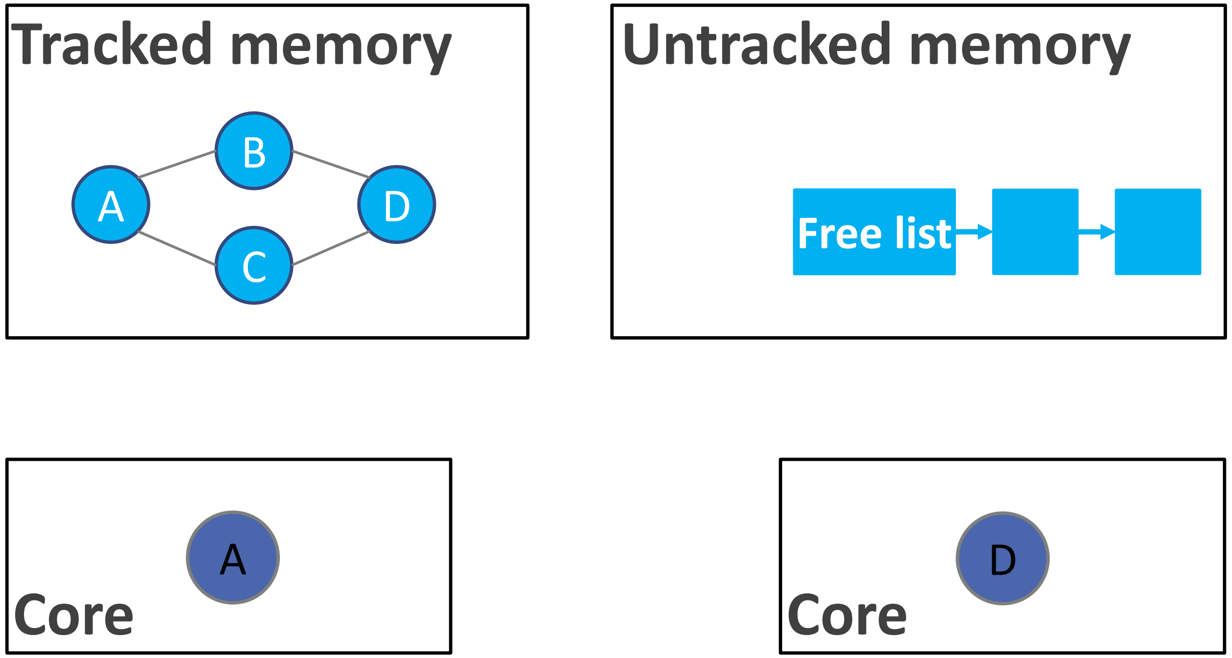


Unversioned, no conflict checks

Only accessible by
- non-speculative tasks
- speculative capsules

Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity



Unversioned, no conflict checks

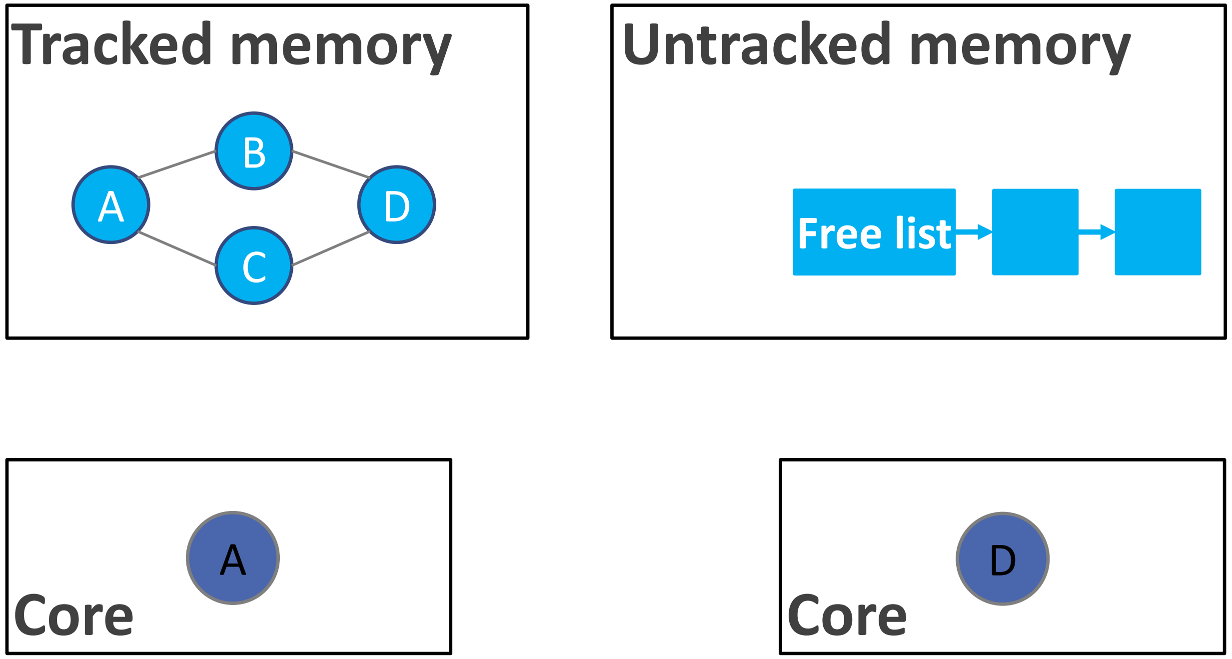
Only accessible by

- non-speculative tasks
- speculative capsules

Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity

Vectored call interface: guarantees control-flow integrity in a capsule



Unversioned, no conflict checks

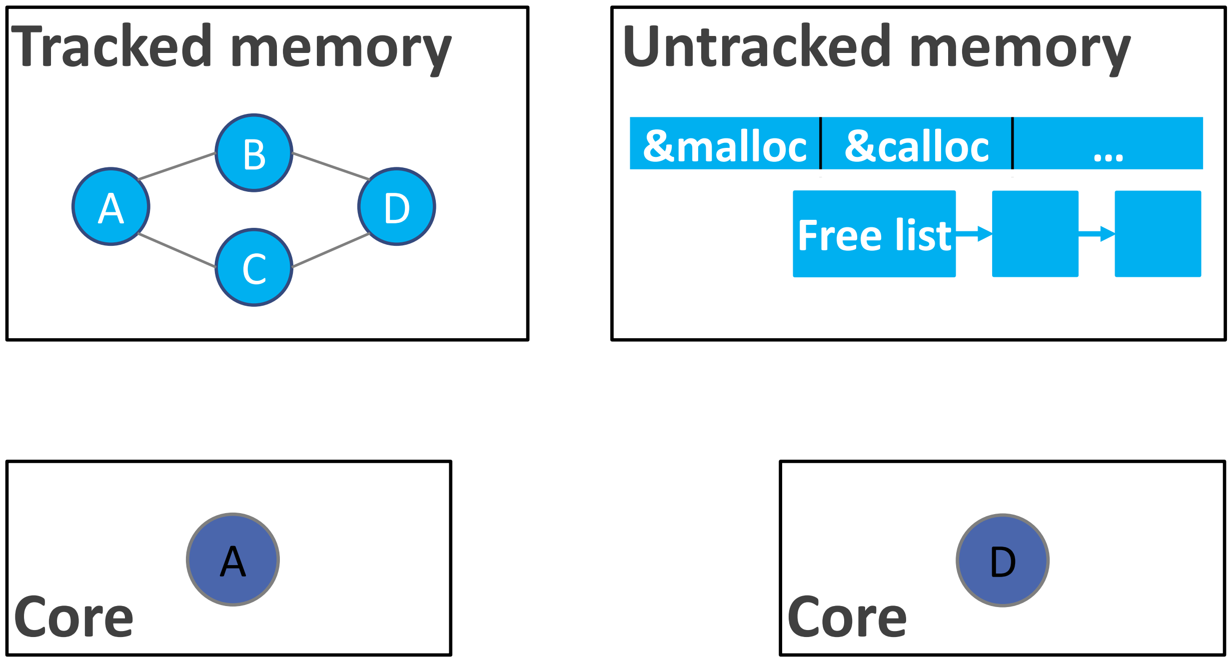
Only accessible by

- non-speculative tasks
- speculative capsules

Capsules ensure safety through OS-like protections

Untracked memory: protected from tasks that lose integrity

Vectored call interface: guarantees control-flow integrity in a capsule



Unversioned, no conflict checks

Only accessible by

- non-speculative tasks
- speculative capsules

Holds the capsule call vector

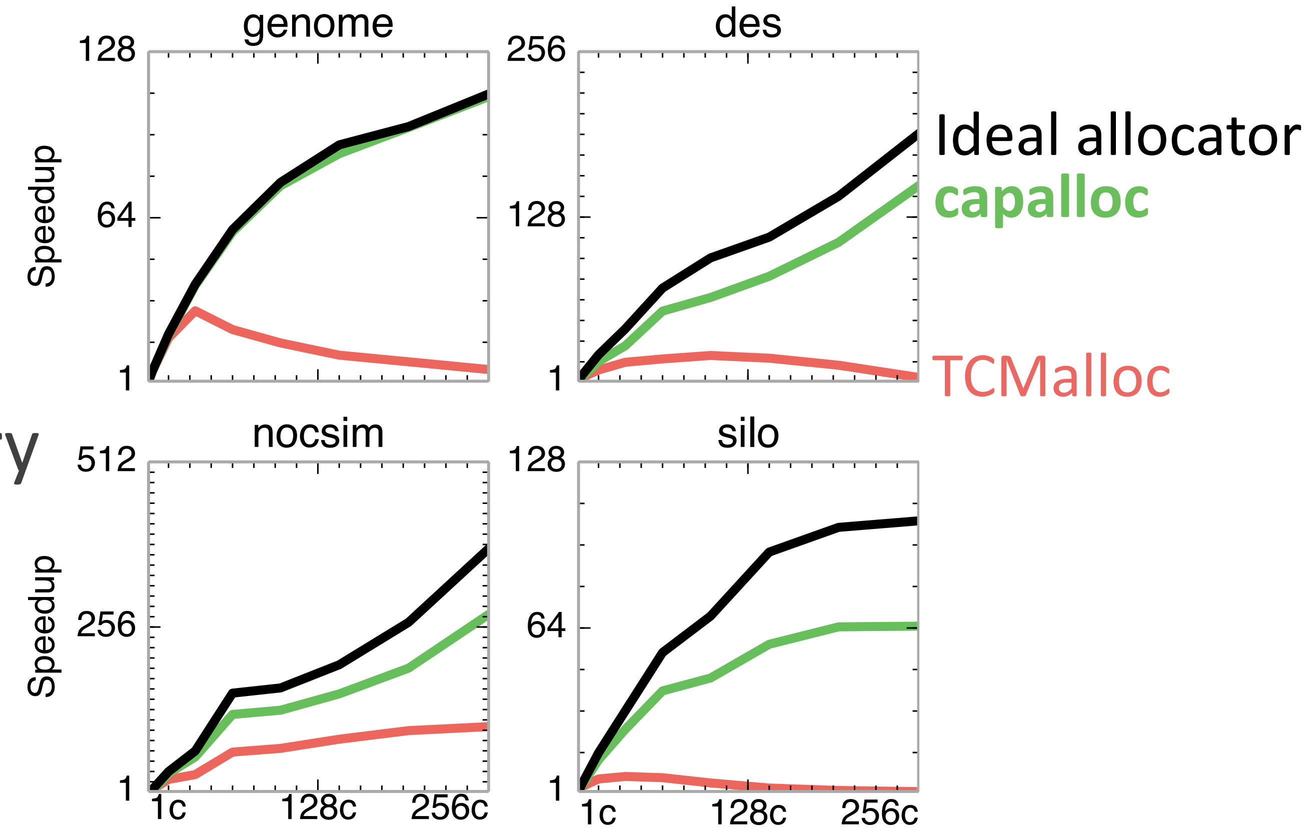
Capsules enable important system services

Capsule-based **allocator**

`malloc`, etc. are capsule functions

metadata resides in untracked memory

Only gmean 30% slower than ideal



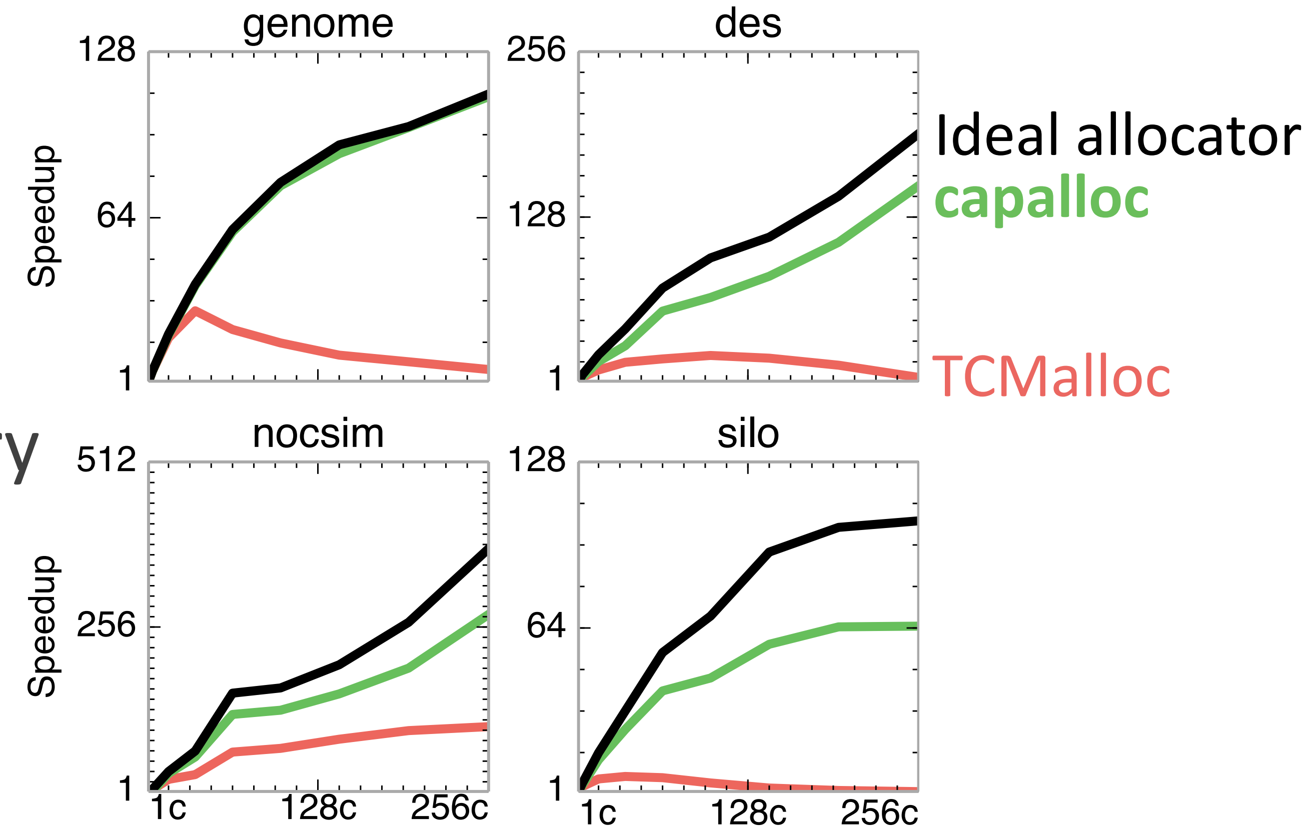
Capsules enable important system services

Capsule-based **allocator**

`malloc`, etc. are capsule functions

metadata resides in untracked memory

Only gmean 30% slower than ideal



capalloc retains the scalability of an ideal allocator

Conclusion

Speculative systems should support non-speculative execution to improve efficiency, ease programmability, and enable new capabilities

Espresso: an execution model for speculative and non-speculative tasks

- Provides shared synchronization mechanisms to all tasks
- Lets the system adaptively run tasks speculatively or non-speculatively

Capsules: speculative tasks safely invoke software-managed speculation

- Enable important speculation-friendly services like scalable memory allocation