Global Task Dependencies in PGAS Applications

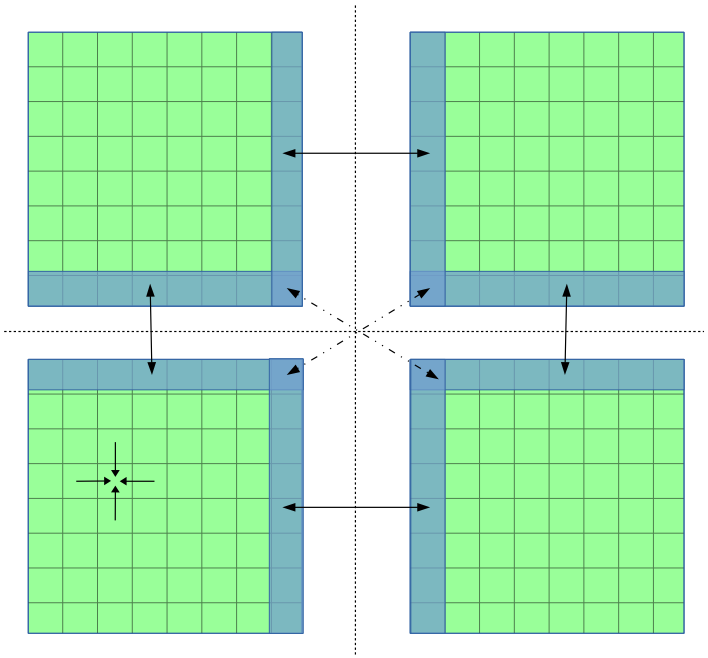
Joseph Schuchart

April 6, 2017

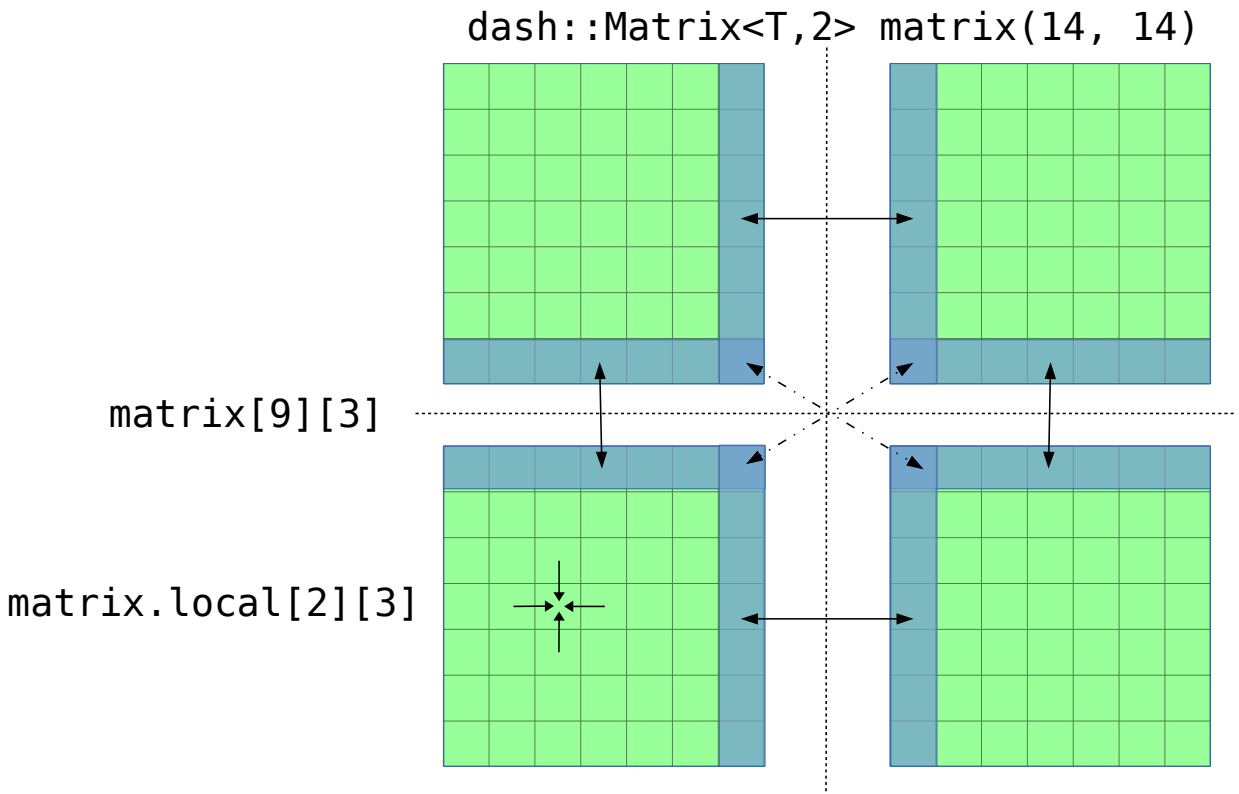
Outline

Domain Decomposition

- Coarse distribution across processes
- Taskification of block computation
- Inner blocks depend on local neighboring blocks
- Boundary blocks depend on Halo blocks from other processes



Domain Decomposition with DASH

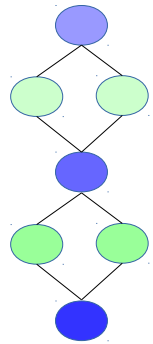


Task-based Programming

- Expose fine-grained concurrency
- Task creation and synchronization (direct and indirect)
- Emerging task-based programming models:
  - OpenMP 4.0
  - OmpSs
  - Chapel
  - HPX
  - Sequoia
  - UPC
  - ...

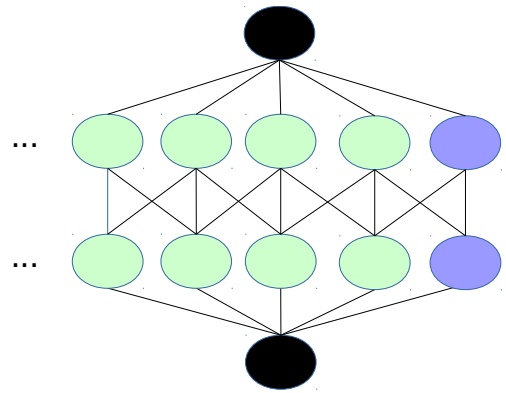
OpenMP Tasking Example

```
int do_work()
{
  int x = 1;

  #pragma omp parallel shared(x)
  #pragma omp single
  {
  for (int i = 0; i < 10; i++)
  {
    #pragma omp task depend(out: x)
    { x = 2*i; }      // T1: dependency on T2 and T3
    #pragma omp task depend(in: x)
    { print(x); }     // T2: dependency on T1, parallel with T3
    #pragma omp task depend(in: x)
    { compute(x); }   // T3: dependency on T1, parallel with T2
  }
  }
  #pragma omp taskwait
}
```



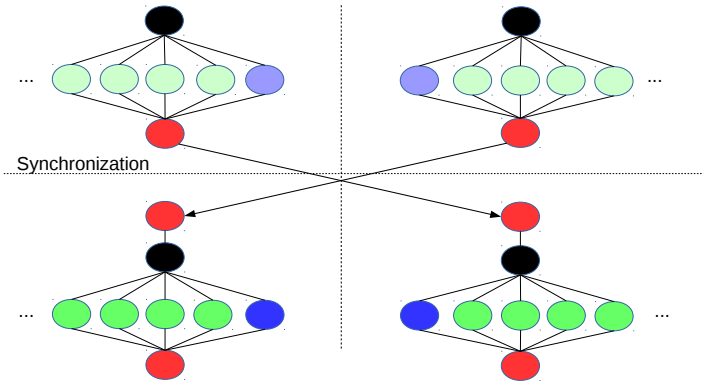
Local Task Dependencies

```
double d[2][N];

for (int t = 0; t < TIMESTEPS; t++) {

  int out = t%2;

#pragma omp task \
        depend(in:  d[out][N-2]) \
        depend(out: d[out][N-1])
  { compute_boundary(d[out]) }

  for (int i = 1; i < N-1; i++) {
#pragma omp task \
          depend(in: d[out][i-1], d[out][i+1]) \
          depend(out: d[out][i])
    { compute_cell(out, i); }
  }
}
```



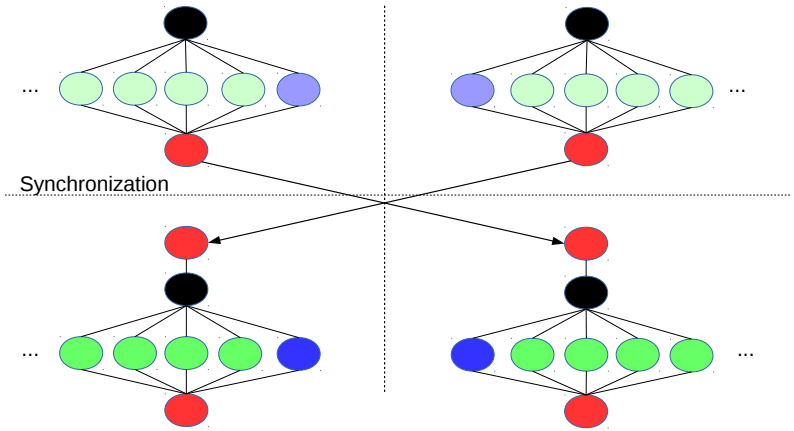
Local Task Dependencies

```
double d[2][N];

for (int t = 0; t < TIMESTEPS; t++) {

  int out = t%2;

#pragma omp task \
        depend(in:  d[out][N-2]) \
        depend(out: d[out][N-1])
  { compute_boundary(d[out]) }

  for (int i = 1; i < N-1; i++) {
#pragma omp task \
          depend(in: d[out][i-1], d[out][i+1]) \
          depend(out: d[out][i])
    { compute\_cell(out, i); }
  }

  #pragma omp taskwait
  exchange_boundaries();
}
```

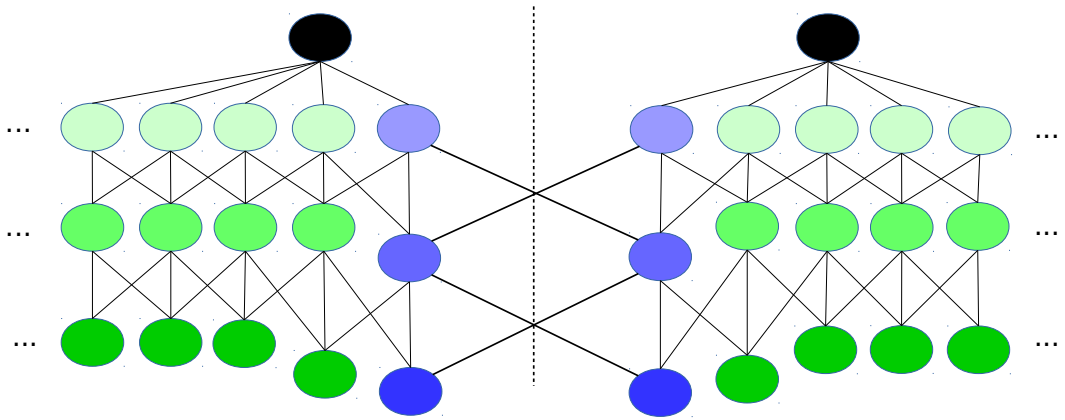


MPI + OpenMP

- Synchronization through messages
- Blocked waiting for communication (imbalances)
- Complex to further taskify
- No messages in PGAS applications



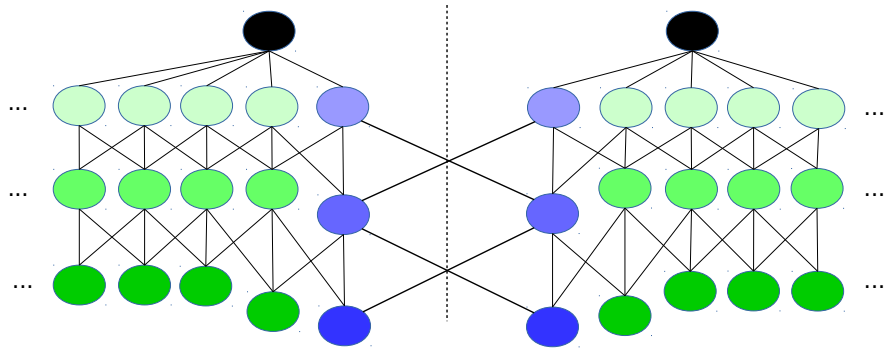
Global Task Dependencies



Global Task Dependencies

Goals:

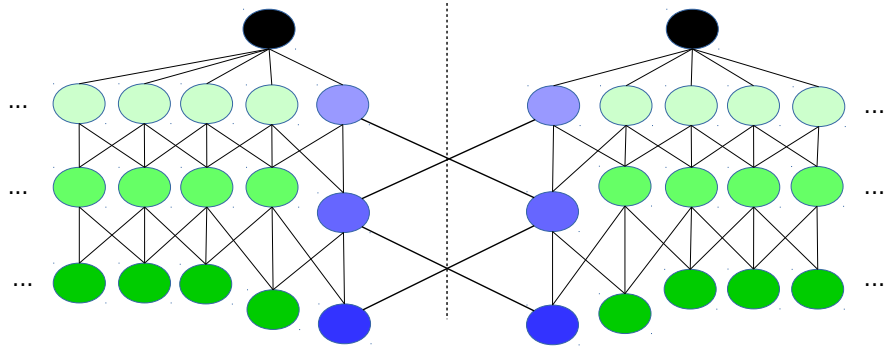
- Reduce required synchronization between processes
- Implicit task synchronization through data dependencies
- Run tasks as soon as all input data is available
  - Increase exposed concurrency
  - Hide communication costs



Global Task Dependencies

Goals:

- Reduce required synchronization between processes
- Implicit task synchronization through data dependencies
- Run tasks as soon as all input data is available
  - Increase exposed concurrency
  - Hide communication costs
- Increase programmer productivity
- Integrated C++ framework: PGAS+Tasking



Global Data Dependencies: Example

Process 0

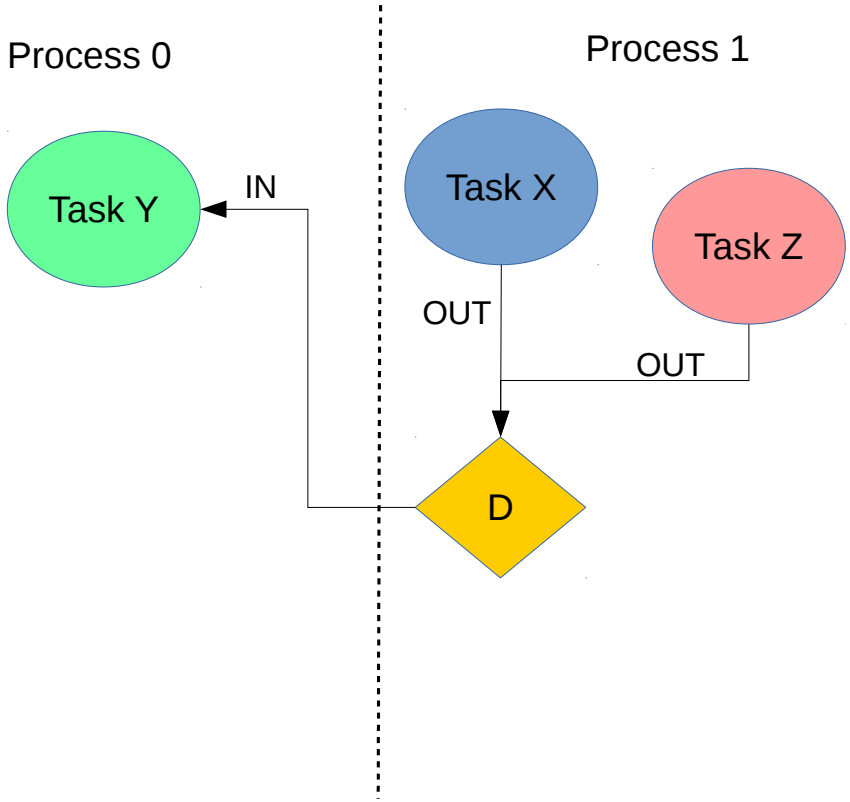
```
dash::Array arr(2);

for (int i=0; i < NITER; ++i) {
  dash::Task(
    in(arr[1]),
    [](){
      print(arr[1]);
    });
}
```

Process 1

```
dash::Array arr(2);

for (int i=0; i < NITER; ++i) {
  dash::Task(
    out(arr[1]),
    [](){
      arr.local[0] = 2*i;
    });
}
```

Global Data Dependencies: Example

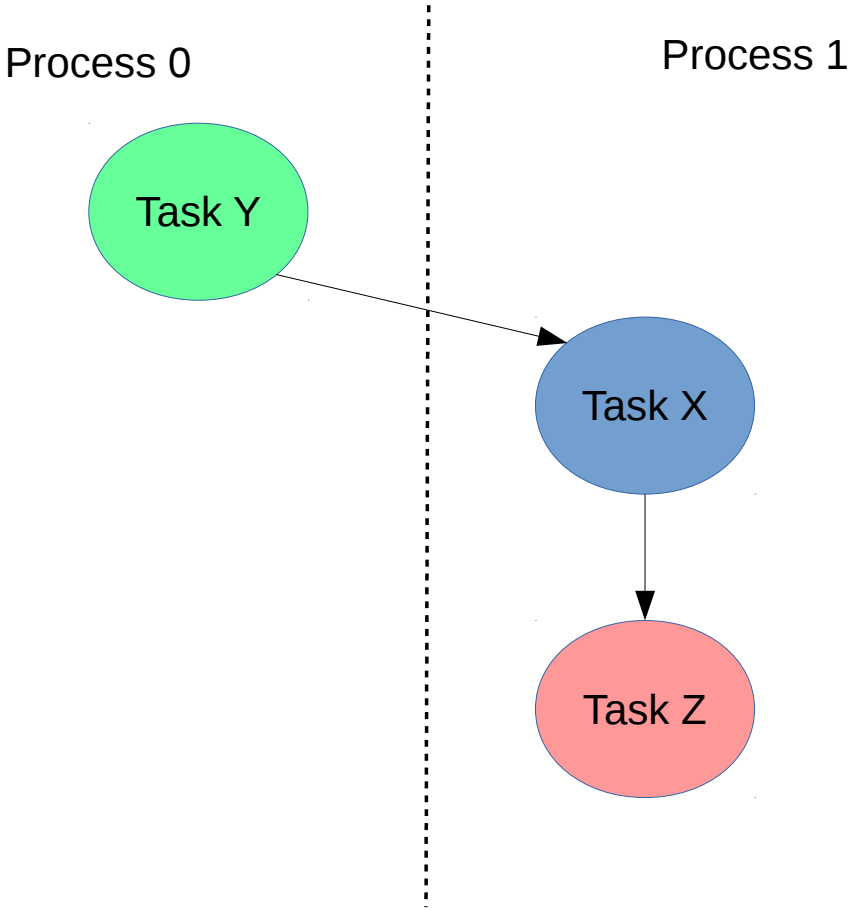
```
print(D);
```



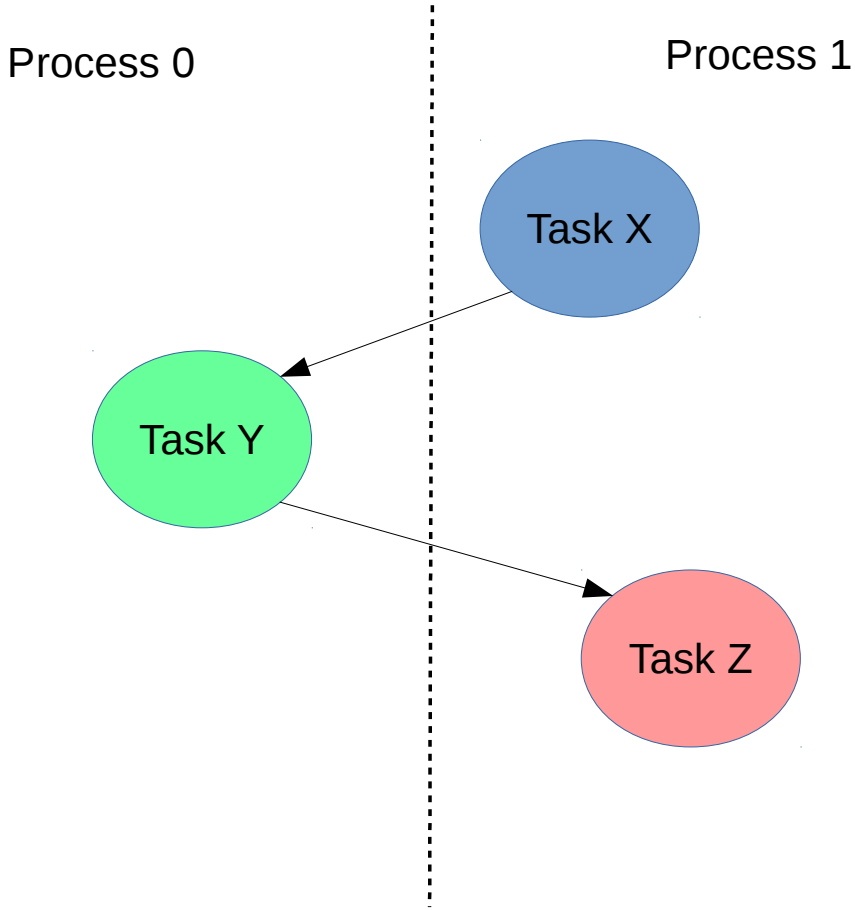

```
D = i++;
```

```
D = i++;
```

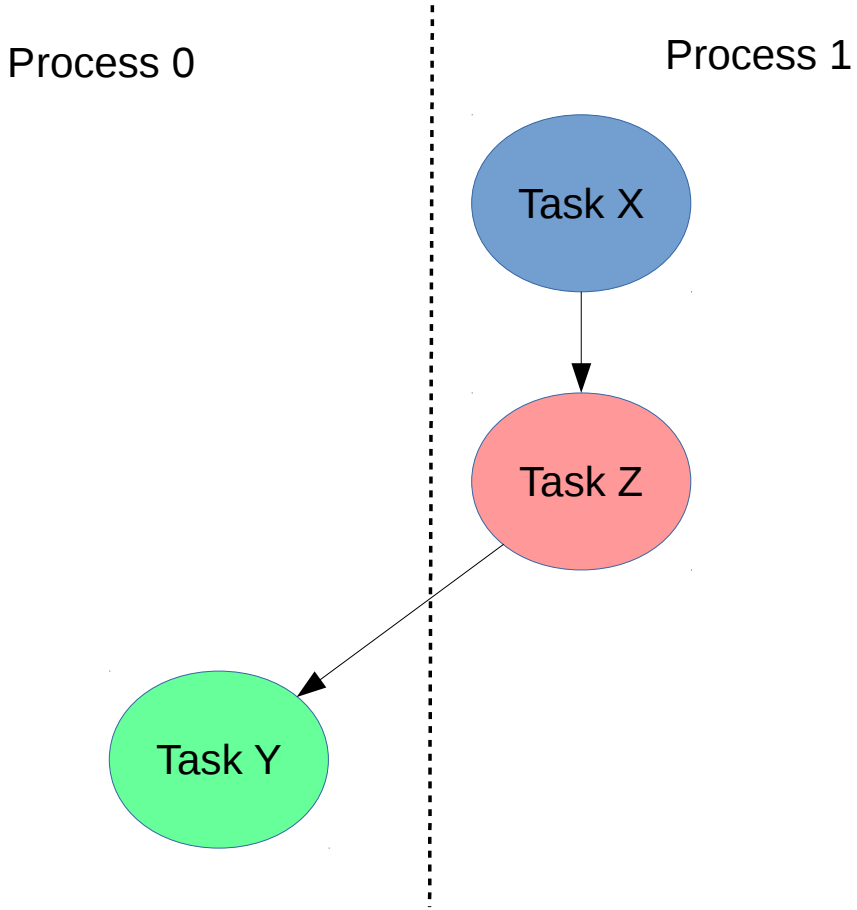
Global Data Dependencies: Example



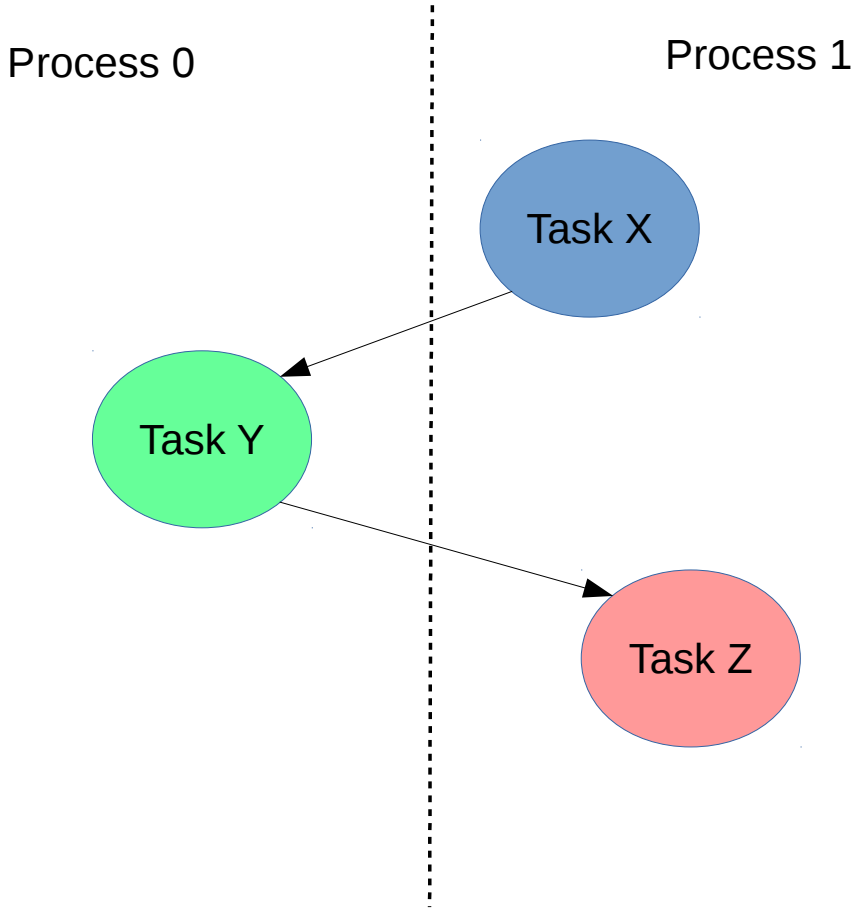
Global Data Dependencies: Example



Global Data Dependencies: Example

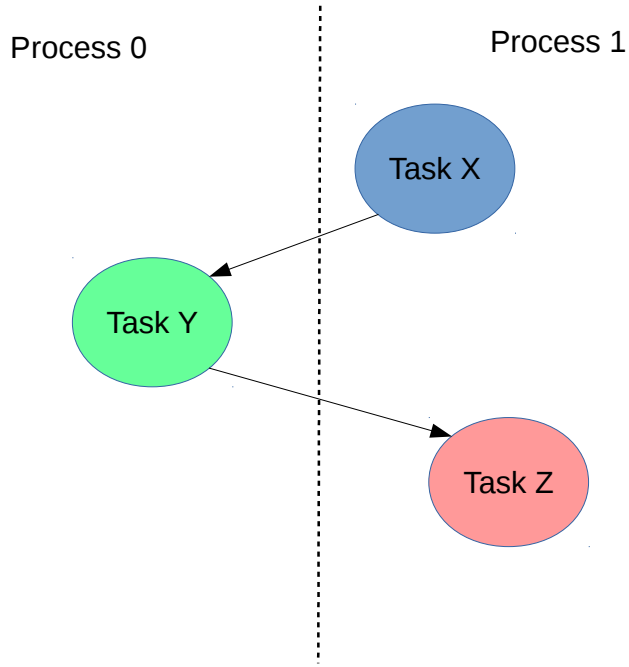


Global Data Dependencies: Example



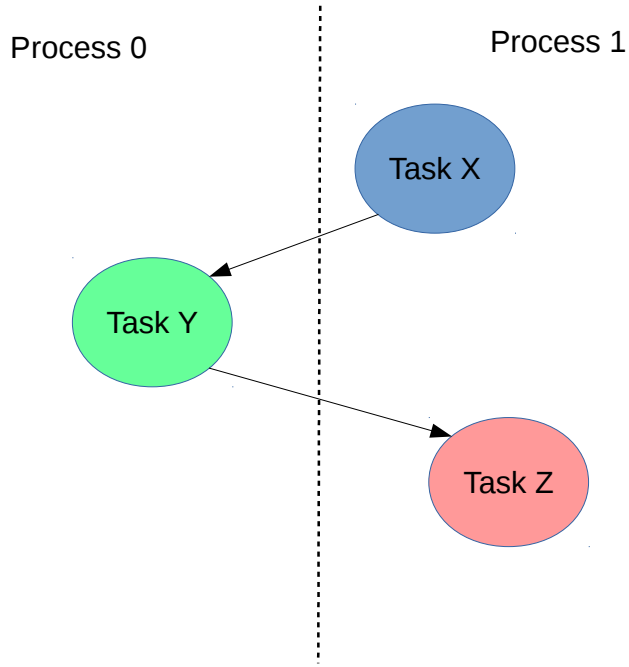
Distributed Task Scheduler

- Seperate scheduler per process
- Communication through active messages
- Remote dependency request and release
- Dependencies repeat over iterations
  ⇝ WAR dependencies



Distributed Task Scheduler

- Seperate scheduler per process
- Communication through active messages
- Remote dependency request and release
- Dependencies repeat over iterations
  ⇝ WAR dependencies

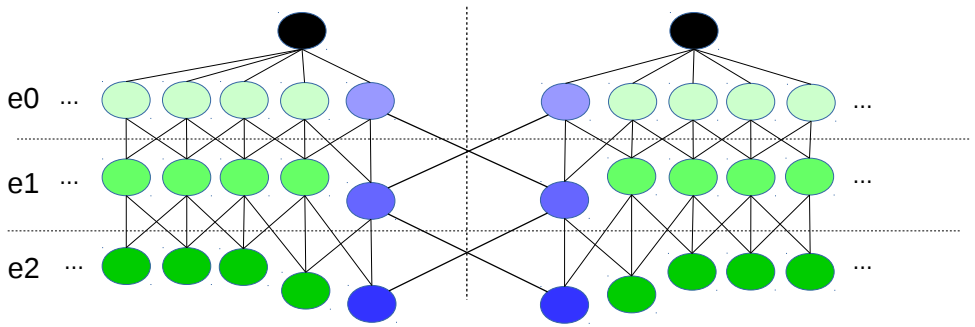


How to ensure correct task ordering?

Global Task Ordering (I)

(Working) Solution: Introduce dependency epochs

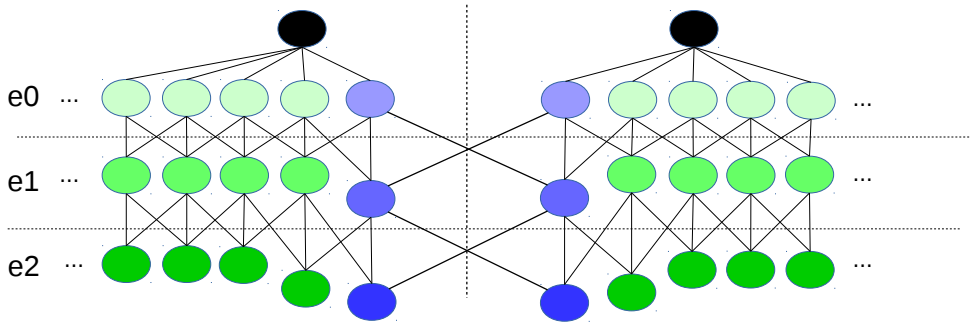
- *Lamport's clock* for dependencies
- User specifies epoch a dependency refers to
- Three-step dependency handling
  1. Collect all local and remote dependencies until global synchronization
  2. Match remote to local dependencies
  3. Continuoulsy release local and remote tasks



Global Task Ordering (I)

(Working) Solution: Introduce dependency epochs

- *Lamport's clock* for dependencies
- User specifies epoch a dependency refers to
- Three-step dependency handling
  1. Collect all local and remote dependencies until global synchronization
  2. Match remote to local dependencies
  3. Continuoulsy release local and remote tasks
- Tasks in the first epoch may run immediately



Global Data Dependencies: Example (II)

Process 0

```
dash::Array arr(2);

// create tasks
for (int i=0; i < NITER; ++i) {
  dash::Task(
    in(arr[1], i),
    [](){
      print(arr[1]);
    });
}

// synchronize and start
// processing tasks
dash::wait();
```

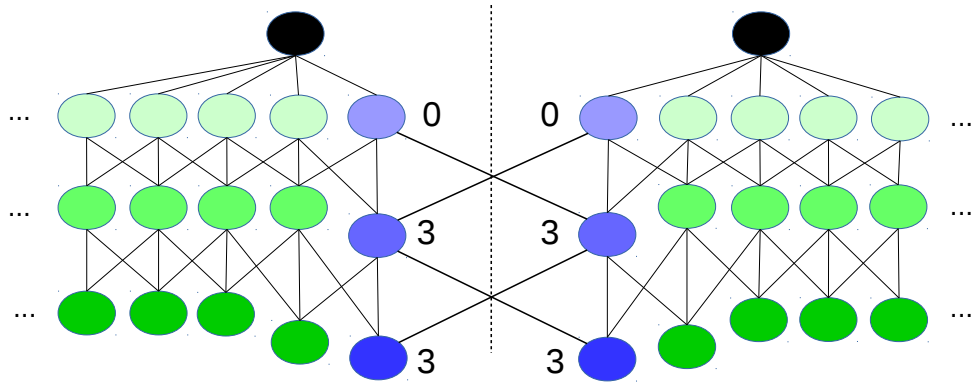
Process 1

```
dash::Array arr(2);

// create tasks
for (int i=0; i < NITER; ++i) {
  dash::Task(
    out(arr[1], i),
    [](){
      arr.local[0] = 2*i;
    });
}

// synchronize and start
// processing tasks
dash::wait();
```

Global Task Ordering (II)

(Alternative) Solution: Specify number of input dependencies

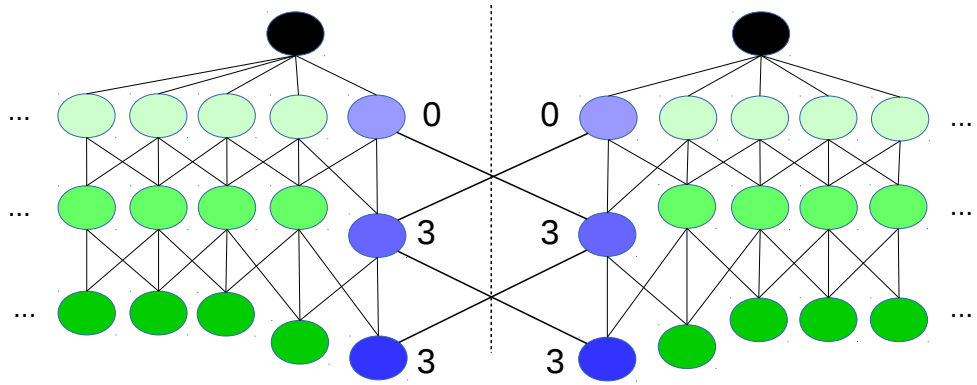
- *Semaphore* for output dependencies
- User specifies the number of input dependencies matching each output dependency
- No global synchronization required
- Continuous execution and release of tasks



Global Task Ordering (II)

(Alternative) Solution: Specify number of input dependencies

- *Semaphore* for output dependencies
- User specifies the number of input dependencies matching each output dependency
- No global synchronization required
- Continuous execution and release of tasks
- **Potentially error-prone?**



Global Data Dependencies: Example (III)

Process 0

```
dash::Array arr(2);

// create tasks and
// execute when possible
for (int i=0; i < NITER; ++i) {
  dash::Task(
    in(arr[1]),
    [](){
      print(arr[1]);
    });
}
```

Process 1

```
dash::Array arr(2);

// create tasks and
// execute when possible
for (int i=0; i < NITER; ++i) {
  dash::Task(
    out(arr[1], 1),
    [](){
      arr.local[0] = 2*i;
    });
}
```

DASH Prototype Implementation

- Thread-safe DASH/DART
- Thread-pool management using `pthreads`
- (Nested) Task creation and execution
- Thread-local run queues w/ task-stealing

DASH Prototype Implementation

- Thread-safe DASH/DART
- Thread-pool management using `pthreads`
- (Nested) Task creation and execution
- Thread-local run queues w/ task-stealing
- Dependencies:
  - Local direct task depdencies
  - Local input/output dependencies (similar to OpenMP)
  - Remote input dependencies (128-bit global Pointer)
- MPI-RMA-based active message queue

DASH Prototype Implementation

- Thread-safe DASH/DART
- Thread-pool management using `pthreads`
- (Nested) Task creation and execution
- Thread-local run queues w/ task-stealing
- Dependencies:
  - Local direct task depdencies
  - Local input/output dependencies (similar to OpenMP)
  - Remote input dependencies (128-bit global Pointer)
- MPI-RMA-based active message queue

- *Stay tuned for benchmark results!*

Conclusion

- PGAS programming model: **one-sided** remote memory access
- New task synchronization strategies necessary

Conclusion

- PGAS programming model: **one-sided** remote memory access
- New task synchronization strategies necessary
- **Proposed solution**: Global data dependencies
  - Concept known from OpenMP
  - Integrated PGAS+Tasking runtime
  - Increase productivity
  - Reduce idle times

Conclusion

- PGAS programming model: **one-sided** remote memory access
- New task synchronization strategies necessary
- **Proposed solution**: Global data dependencies
  - Concept known from OpenMP
  - Integrated PGAS+Tasking runtime
  - Increase productivity
  - Reduce idle times
- Hints on ordering required by user (**How?**)

Future Work

- Improvements to the scheduler
- Alternatives to dependency epochs
- Advanced dependency types
- C++ interface
- Tools support (OMPT? XMPT? DMPT?)

Thank you for your attention!

joseph.schuchart@hlrs.de
github.com/dash-project/

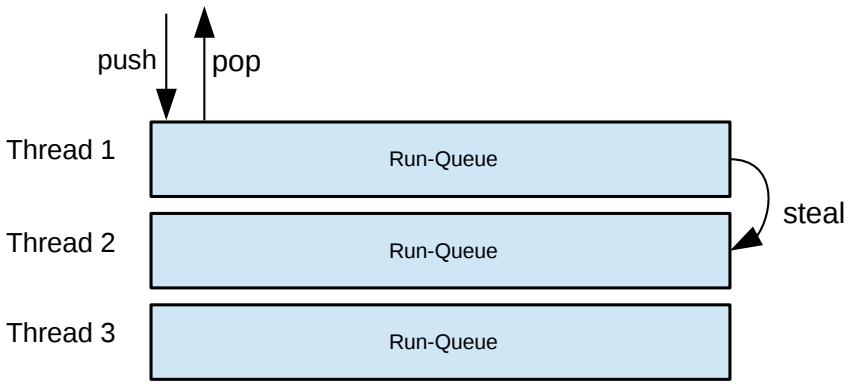
Future Work

- Dynamic control of number of threads
- Yielding the current task
- Task priorities

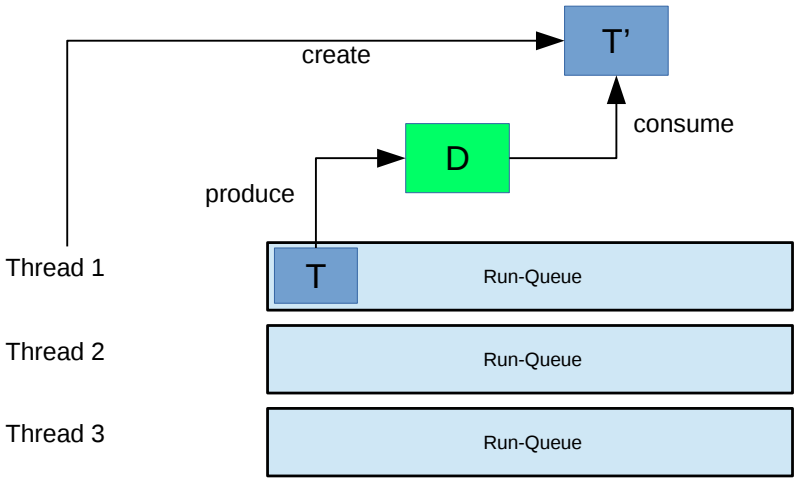
Future Work

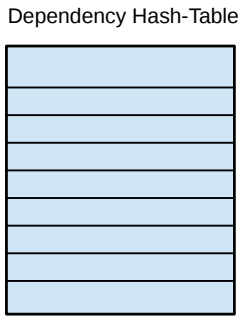
- Dynamic control of number of threads
- Yielding the current task
- Task priorities
- Additional dependency types:
  - Sub-range dependencies (?)
  - Remote output dependencies
  - Copy-in dependency (renaming on user-defined buffer)
  - Dependency on any asynchronous event

(Local) Task Dependencies

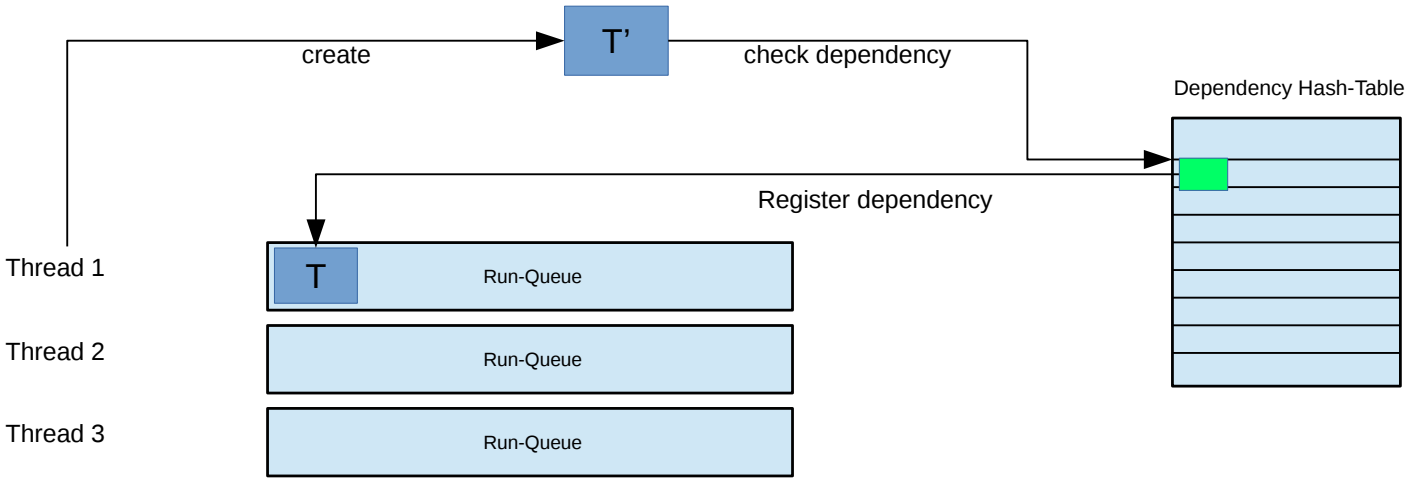


(Local) Task Dependencies

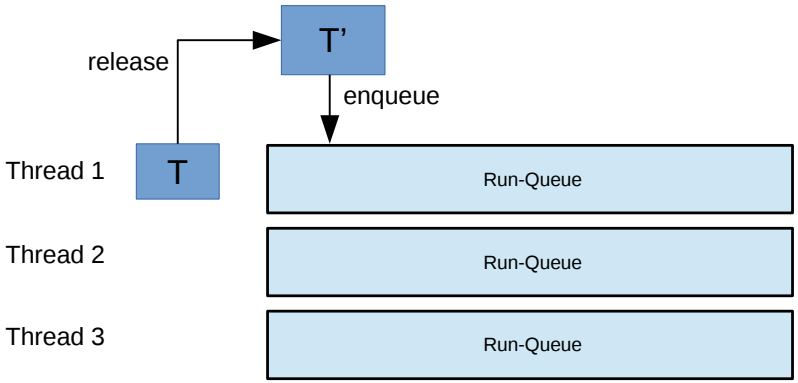


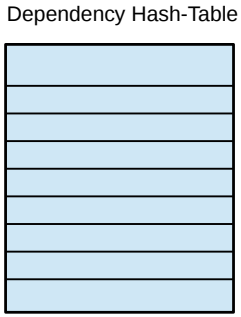


(Local) Task Dependencies

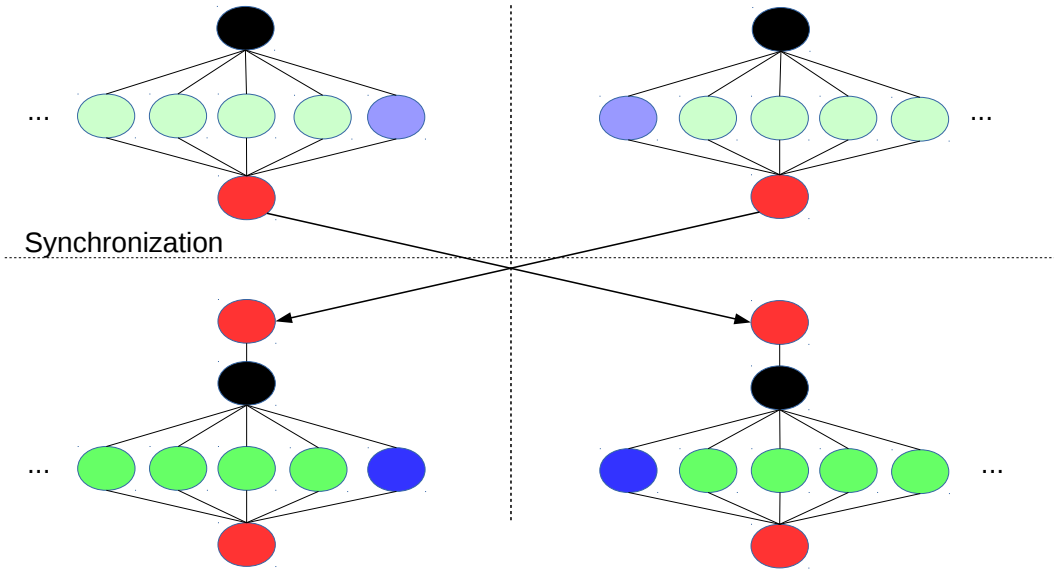


(Local) Task Dependencies

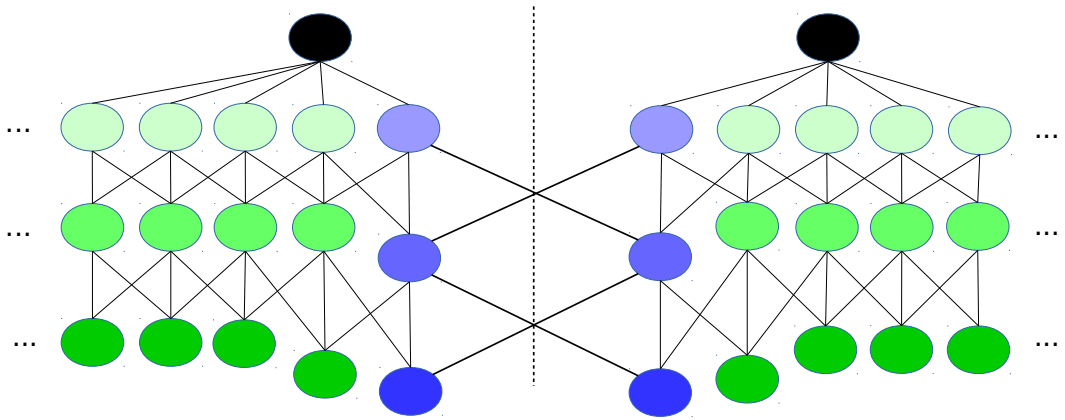




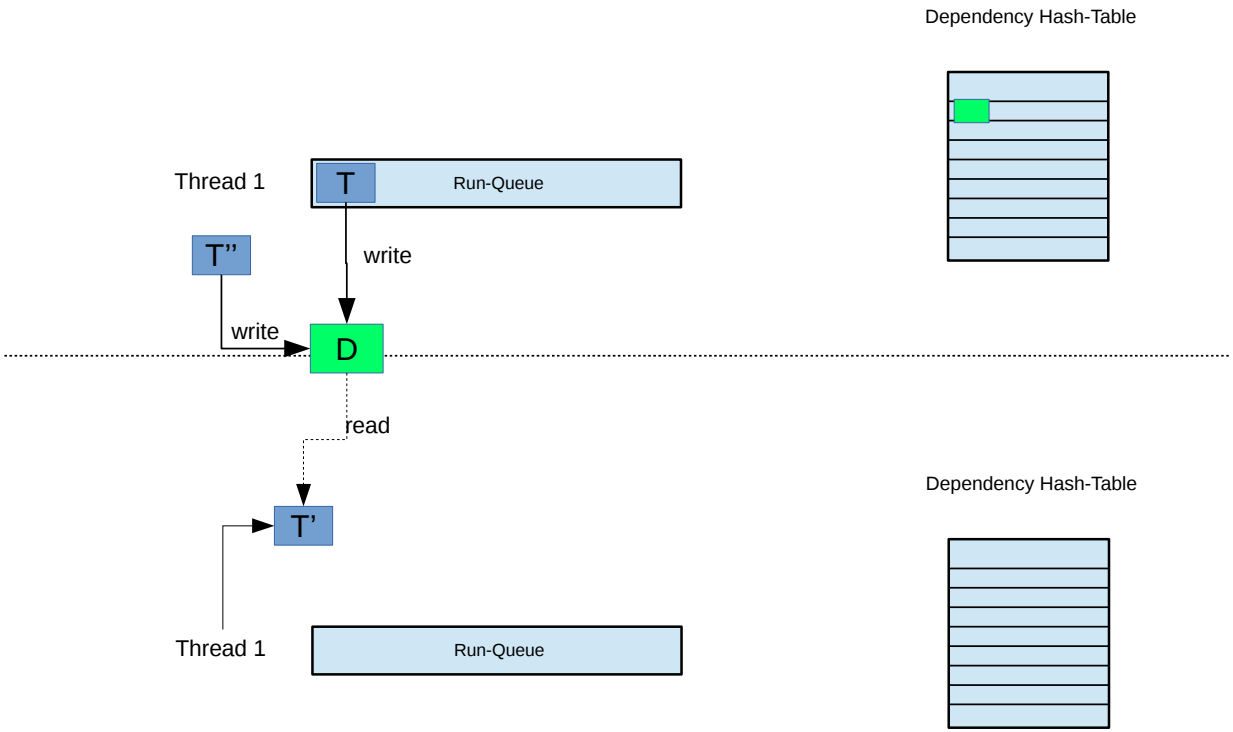
Global Task Dependencies



Global Task Dependencies

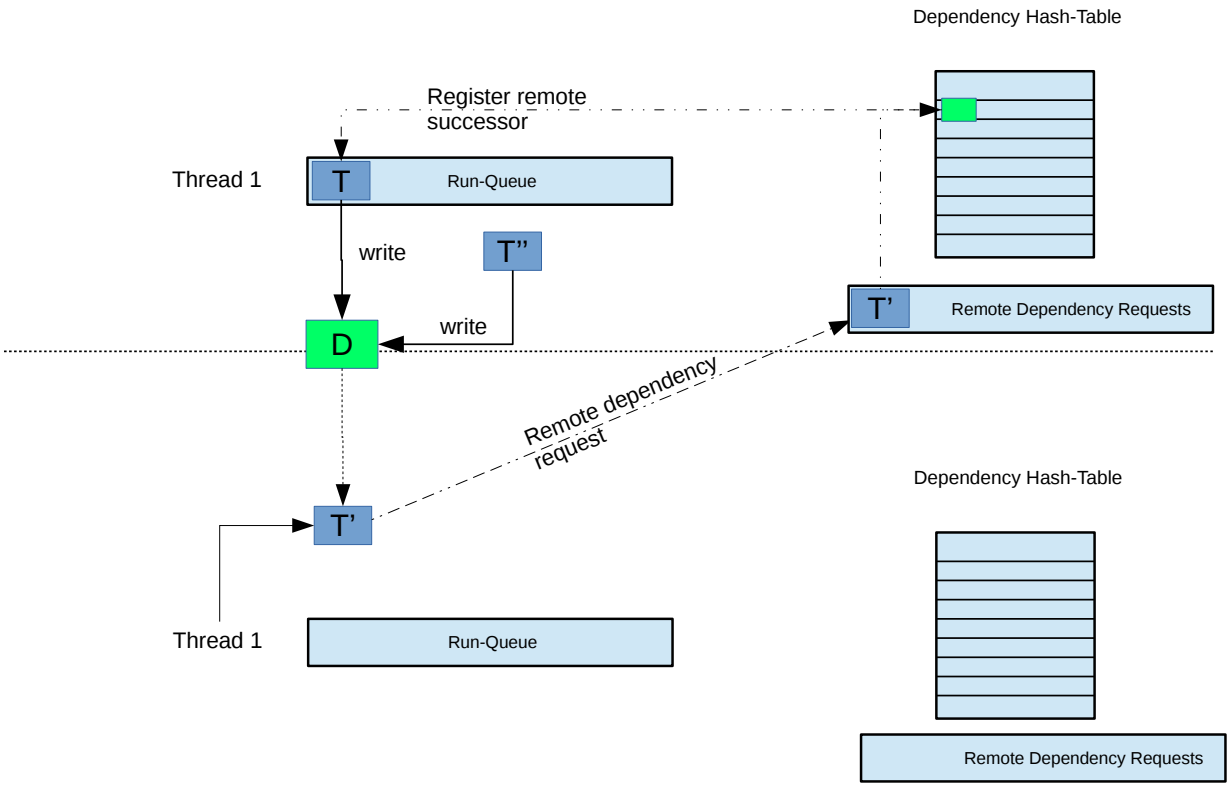


Global Task Dependencies



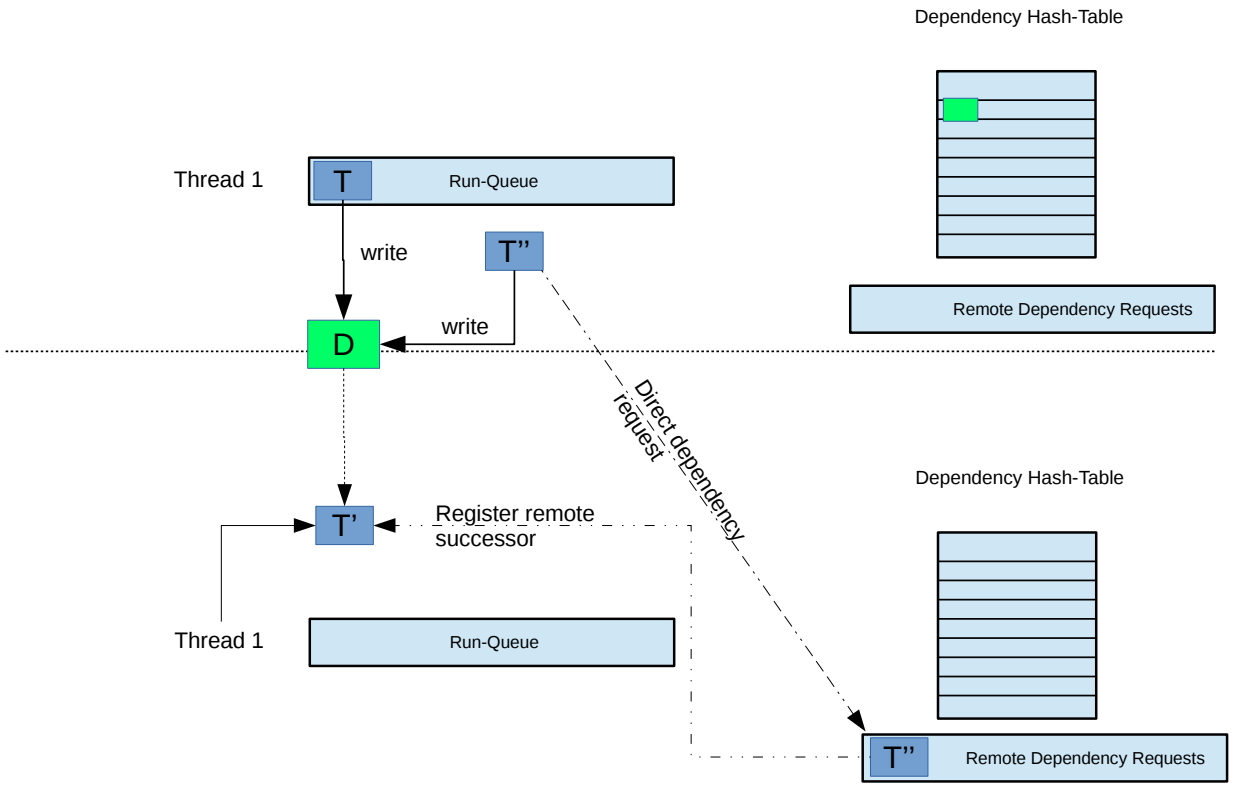
Correct order: $T \rightarrow T' \rightarrow T''$

Global Task Dependencies



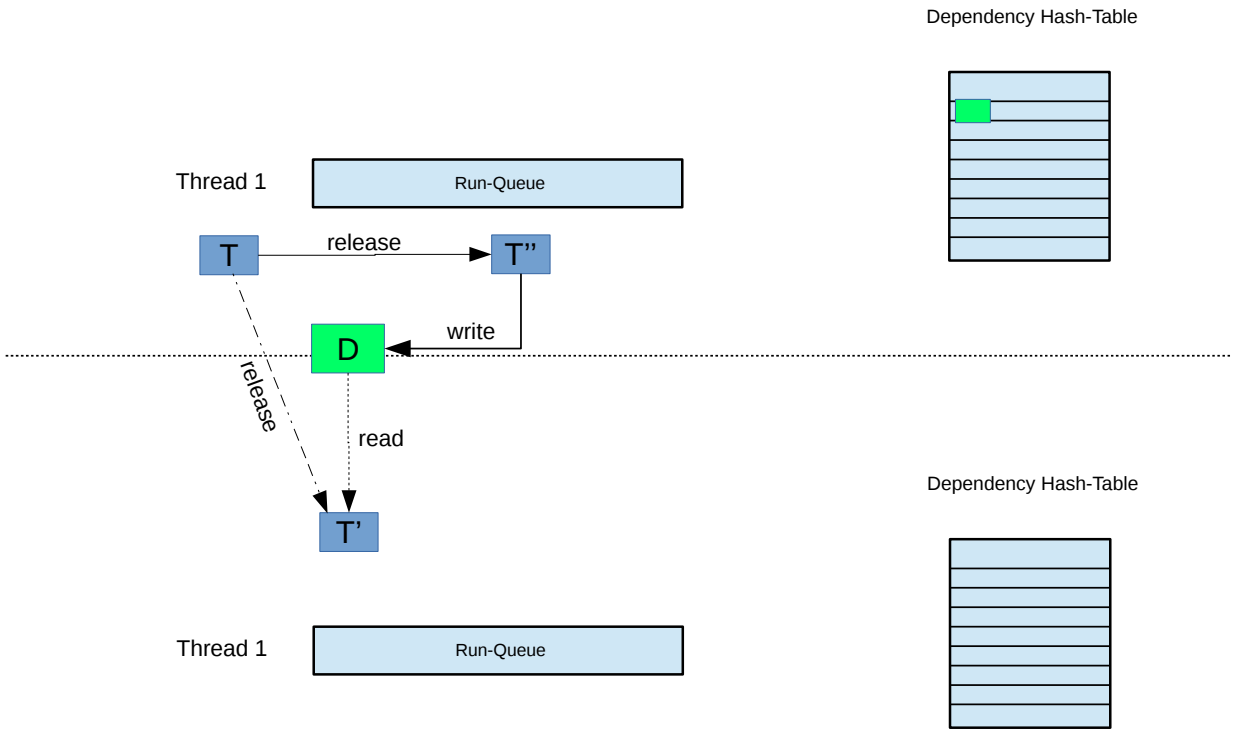
Correct order: $T \rightarrow T' \rightarrow T''$

Global Task Dependencies



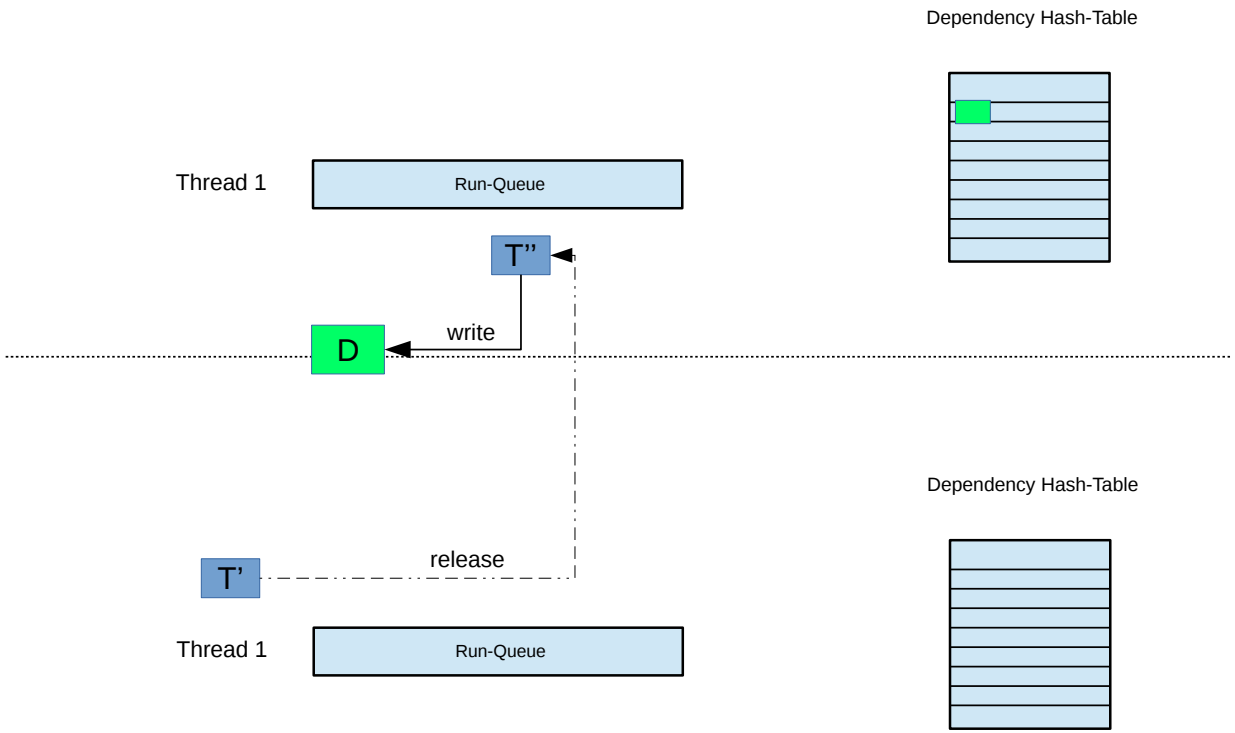
Correct order: $T \rightarrow T' \rightarrow T''$

Global Task Dependencies



Correct order: $T \rightarrow T' \rightarrow T''$

Global Task Dependencies



Correct order: $T \rightarrow T' \rightarrow T''$

DART Tasking API

- Init and tear-down: `dart_tasking_init()` and `dart_tasking_fini()`
- Create tasks:
  - Fire-and-forget: `dart_task_create(fn, data, size, deps, ndeps)`
  - Create a handle: `dart_task_create_handle(fn, data, size, deps, ndeps, *handle)`
- Complete tasks: `dart_task_complete()` and `dart_task_wait(handle)`
- Thread info: `dart_tasking_num_threads()` and `dart_tasking_thread_num()`
- Phase increment: `dart_tasking_phase()`

DASH

- C++11 PGAS framework
- STL-style
  - Container: `Array`, `NArray`, `List`, `UnorderedMap`
  - Algorithms: `copy`, `reduce`, `accumulate`
- One-sided access through `put`/`get`
- Low-level runtime DART implemented in C (MPI-3 RMA)

DASH

- C++11 PGAS framework
- STL-style
  - Container: `Array`, `NArray`, `List`, `UnorderedMap`
  - Algorithms: `copy`, `reduce`, `accumulate`
- One-sided access through `put`/`get`
- Low-level runtime DART implemented in C (MPI-3 RMA)