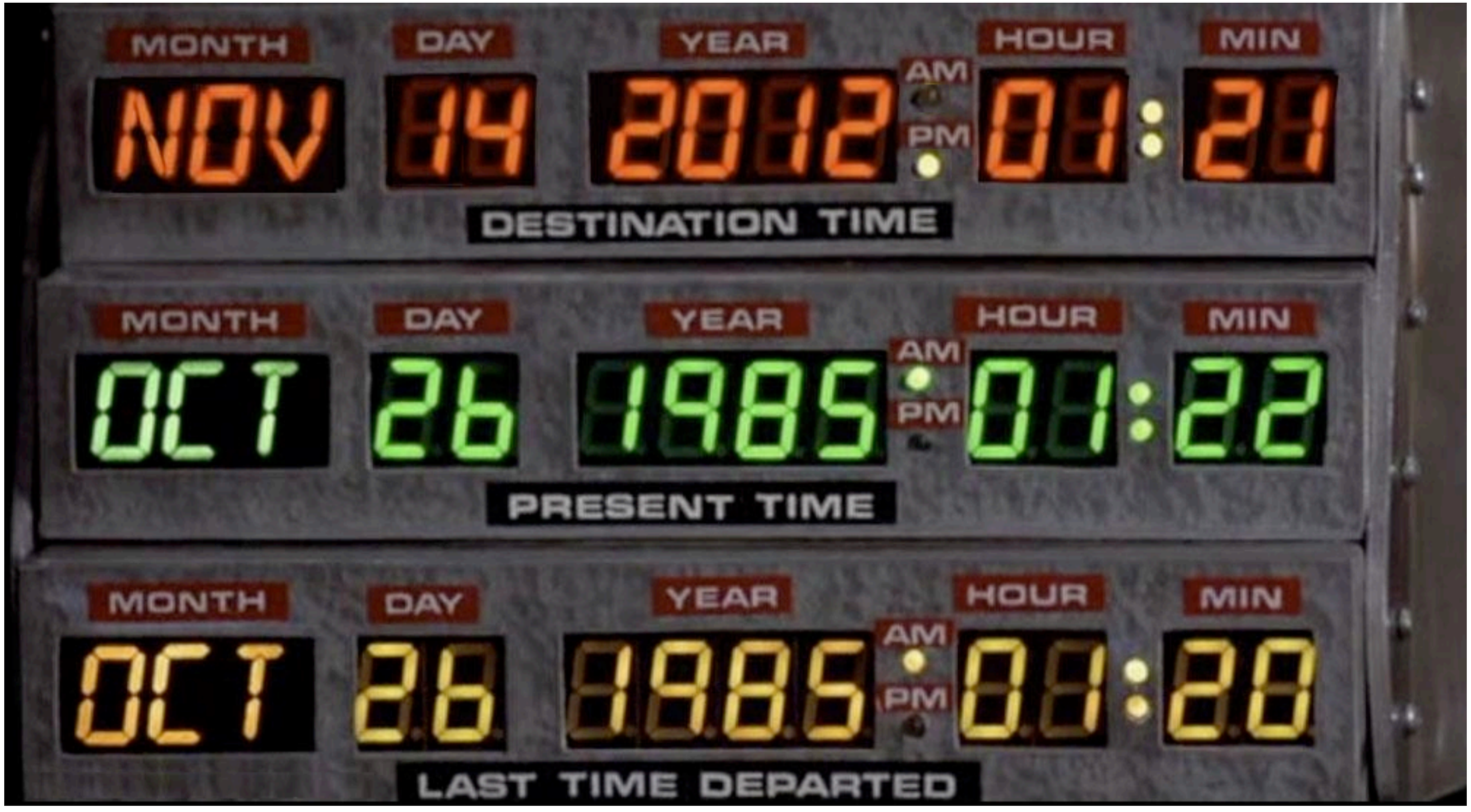


Chapel: Back to the Future

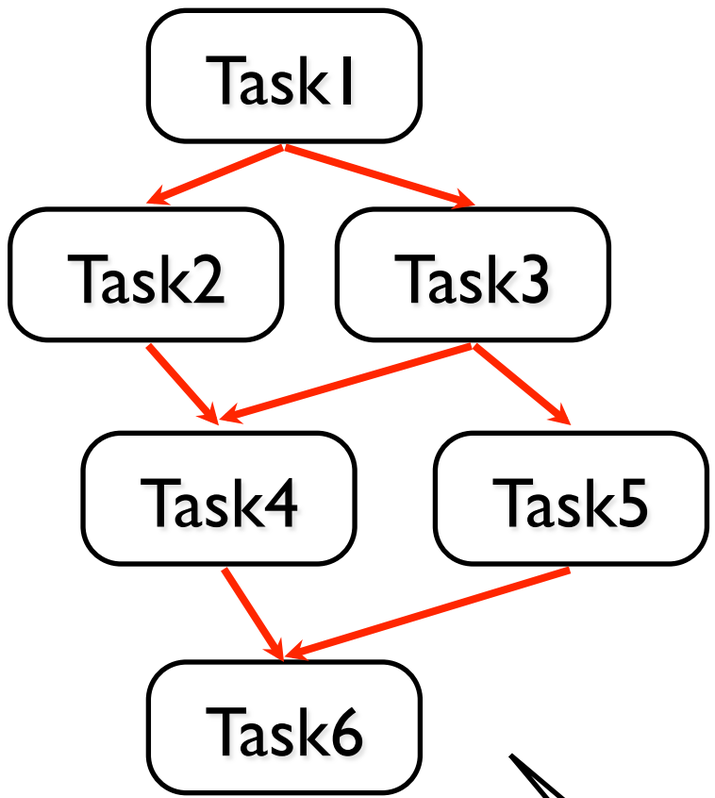
Sağnak Taşırlar (sagnak@rice.edu)
Rice University

Acknowledgments

- Other members of Habanero Multicore Software Research group at Rice for their contributions to this project
  - Vincent Cave, Philippe Charles, Shams Imam, Jisheng Zhao, Vivek Sarkar
- Chapel experts for their suggestions
  - Brad Chamberlain, Michael Ferguson
- This material is based upon work supported by the Department of Defense and the National Science Foundation under a supplement to Grant No. 0964520. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense or the National Science Foundation (NSF)."

RICE

Motivation: how to express the inherent parallelism in your program without additional synchronization?



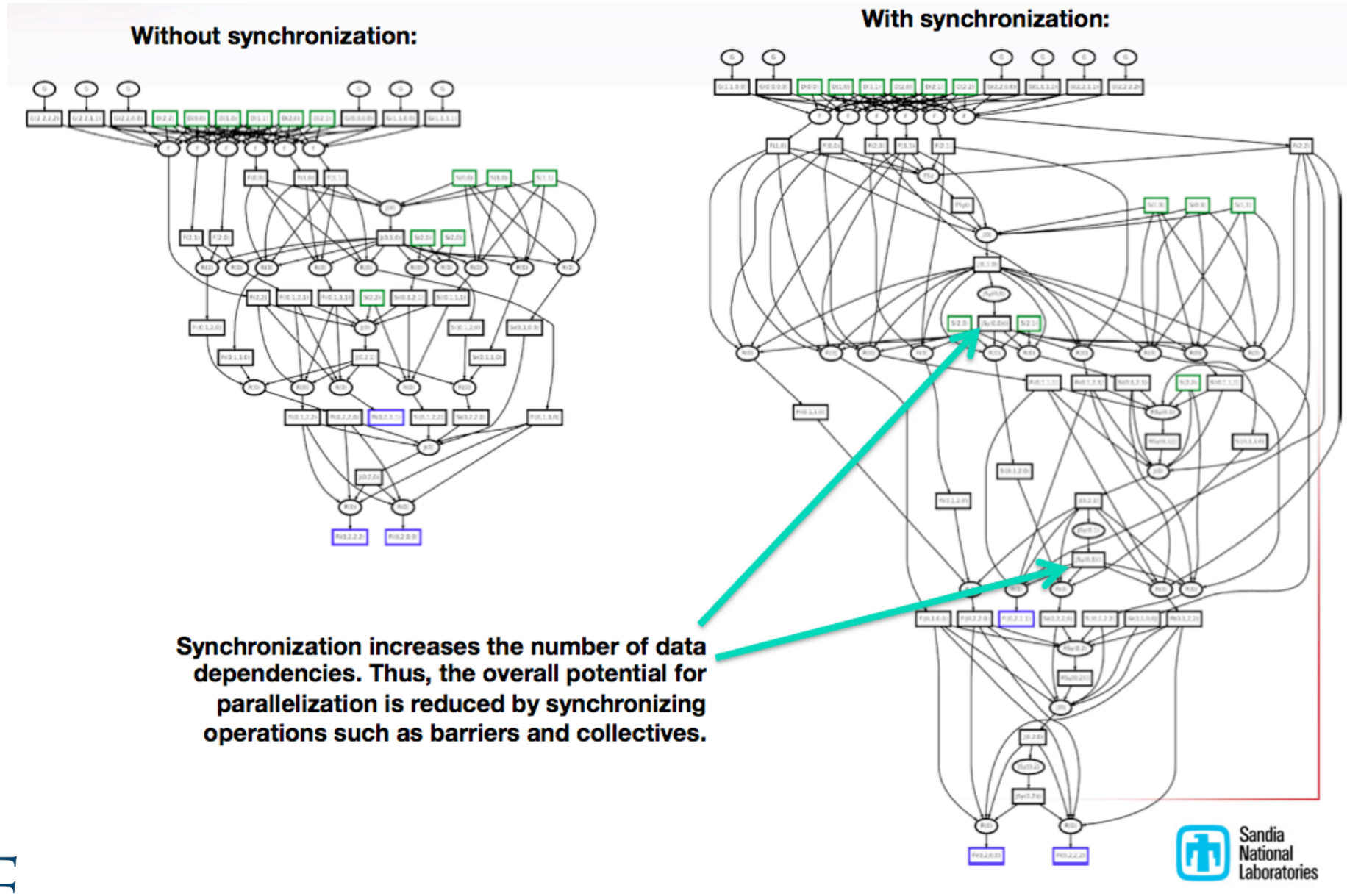
- `begin/sync`
- `spawn/sync`
- `finish/async`
- `forall`

are not sufficient in general

Use boxes & arrows

RICE

Data Dependencies in Hartree-Fock Theory application without & with additional synchronization



Source: “Pushing back the point of diminishing returns for parallel performance”, Curtis Janssen, Workshop on Exascale Programming Challenges, July 2011, Marina del Rey, CA

Proposal for Futures in Chapel: extend begin-tasks with return values

Example with syntax supported by current prototype

```
// Parent task creates child task (begin-expression)
var container = begin:int {...; computeSum(X, low, mid);};
. . .
// Later, parent examines the return value from the last statement
var sum = container.get();
```

Three key features of futures:

1. Ability for a task (begin-expression) to return a value
   - Type of return value is declared (or inferred) for begin task
2. Distinction between container and value in container
   - Type of value in container is T
   - Type of container and begin-expression is Future[T]
3. Synchronization to avoid race condition in container accesses
   - get() operation blocks until value becomes available

RICE

Important Semantic Properties of Futures

1. Any dependence graph can be expressed using futures without adding additional synchronization
2. No data race possible on return value of a future
3. If all futures are stored in immutable (const) variables, then no deadlock cycle can be created with get() calls

RICE

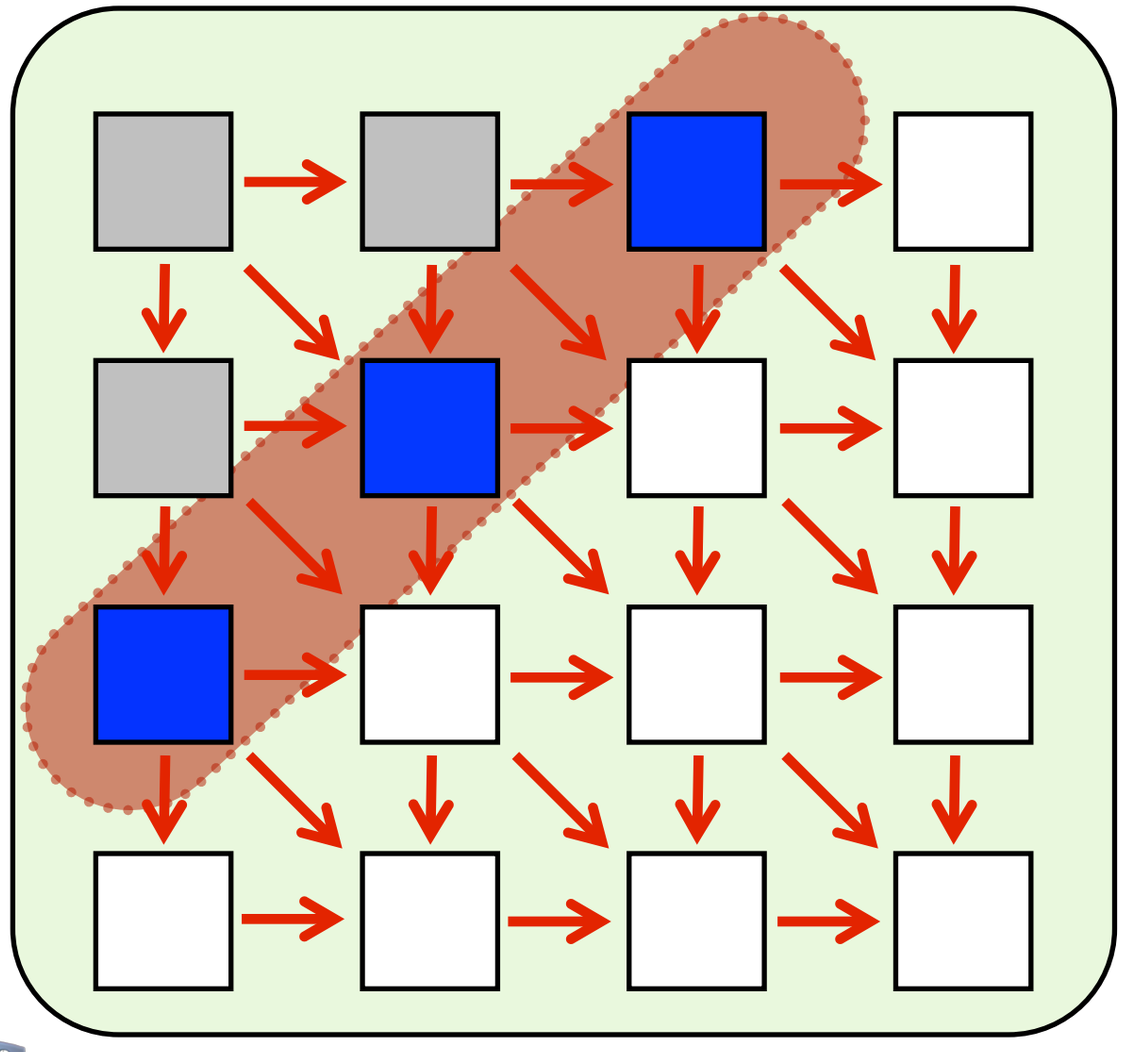
Futures can be used to support team-join operations

```
const team = begin : bool { // Team of tasks A & B
                    sync { begin A(); begin B(); }
                        true; // Signals end of team execution
                }
. . .
team.await(); // Performs a join on team
```

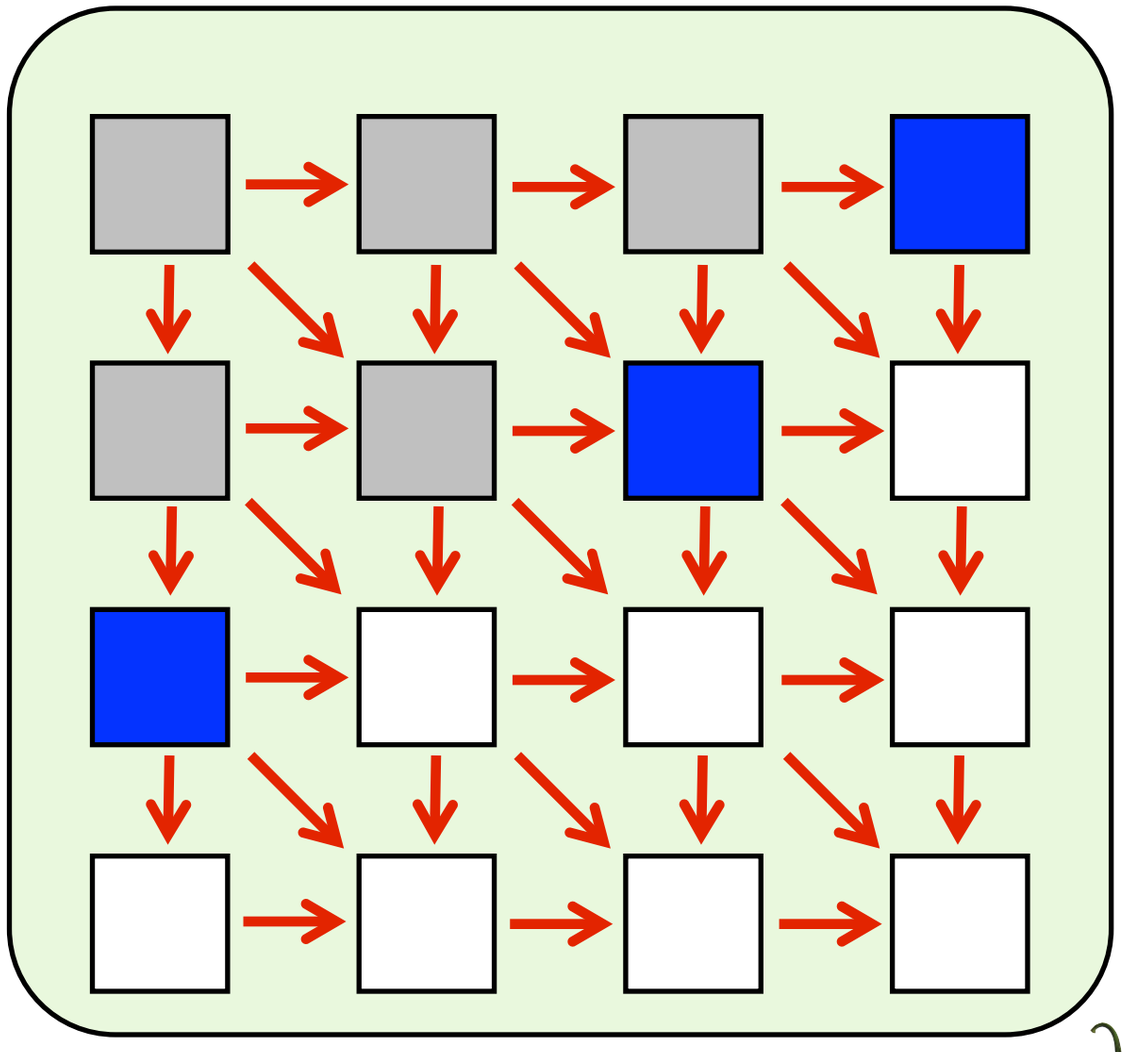
RICE

Smith-Waterman LSA Benchmark

Synchronous wavefront (using forall)



Asynchronous wavefront (using futures)



RICE

Preliminary Results

	Single Tile	64 tiles	256 tiles
Chapel version w/ forall loop (16 qthreads)	10.362	3.620	2.635
Chapel version w/ futures (16 qthreads)	10.410	3.198	1.678

In seconds
Collected on a 16-core Xeon

RICE

Current Status and Future Work

Current Status

- Prototype implementation of futures in Chapel completed with compiler and runtime extensions
- Patch available on request; in process of being submitted to Chapel open source project

Future Work

- Extend syntax to allow "begin { … }" to be used as a future expression with an implicit return type (disambiguate from begin statement)
- Experiment with other benchmarks
- Experiment with cluster implementation of multi-locale version of futures
- Performance improvements
  - Replace future calls by data-driven tasks (DDTs) as compiler optimization (with runtime support for DDTs in qthreads)

RICE

BACKUP SLIDES START HERE

Syntax Details

`begin : T { Stmt-Block }`

- Create a new child task that executes **`Stmt-Block`**, in which the **last statement** must return a value of type `T`

- Begin-expression returns a reference to a container of type **`future[T]`**, which can be implicit or explicit

- Currently, ": T" is required as an explicit cast of the begin expression to type T. Inferring the type is easy, but "**`begin { Stmt-Block }`**" leads to parsing conflicts.

`Expr.get()`

- Evaluate **`Expr`**, and block if **`Expr`**'s value is unavailable

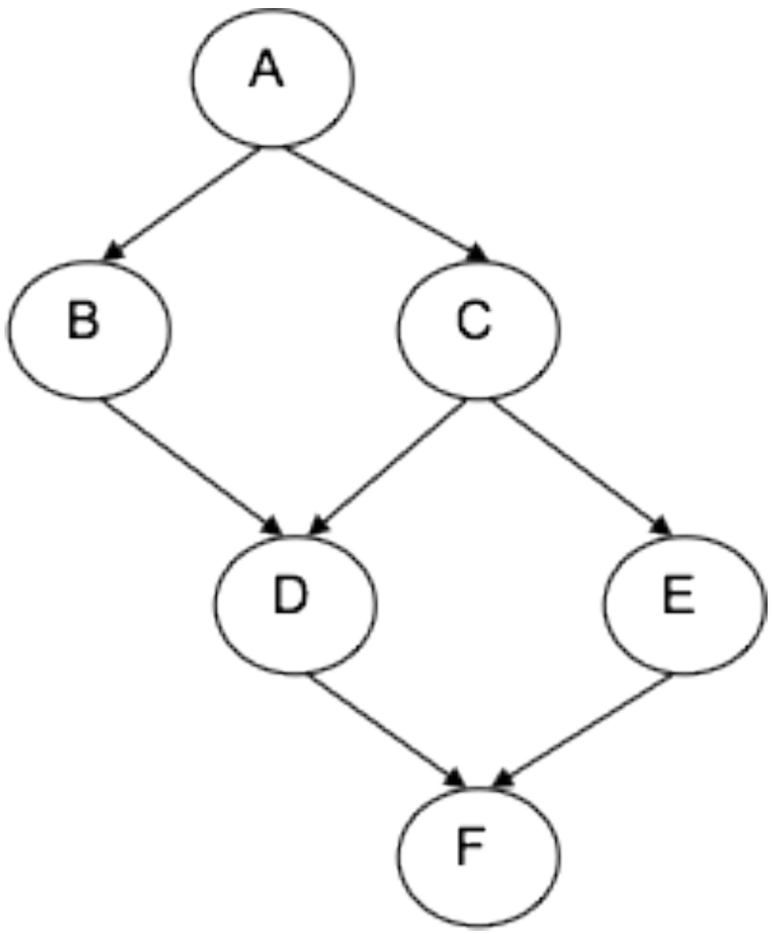
  - **`Expr.await()`** only blocks until value is available but does not return value

- **`Expr`** must be of type **`future[T]`**, and return value from **`Expr.get()`** will then be of type **`T`**

- Unlike sync which waits for all begin-tasks in the sync scope, a **`get()`** operation only waits for the specified begin-expression

RICE

Extension: support void return type in Futures




```
// Example of “dag parallelism”

 const A = begin:void { . . . };

 const B = begin:void { A.await(); . . . };

 const C = begin:void { A.await(); . . . };

 const D = begin:void { B.await();

                        C.await(); . . . };

 const E = begin:void { F.await(); . . . };

 const F = begin:void { D.await();

                        E.await(); . . . };
```

NOTE: above example can be implemented using dummy return value in current prototype

RICE