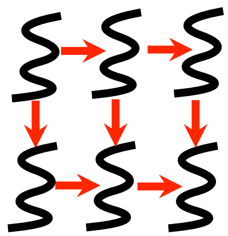
DATA-DRIVEN TASKS AND THEIR IMPLEMENTATION

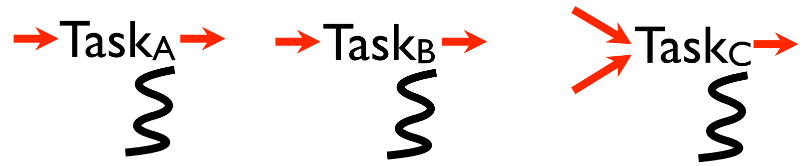
SAĞNAK TAŞIRLAR, VIVEK SARKAR
DEPARTMENT OF COMPUTER SCIENCE. RICE UNIVERSITY

Fork/Join graphs constraint ||-ism

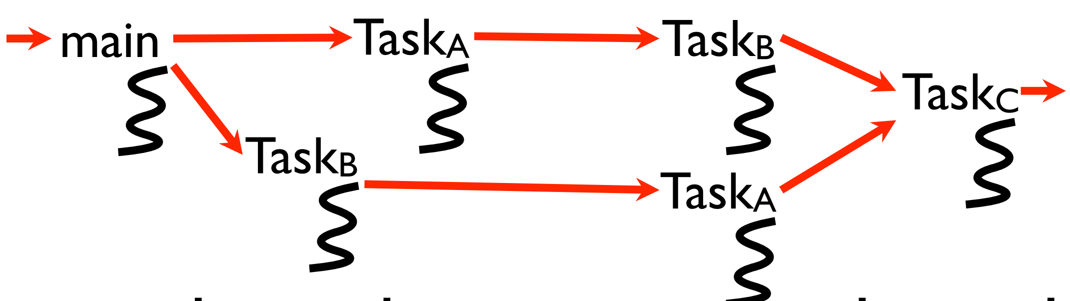
- Fork/Join models restrict task graphs to be series-parallel
  - Can not describe  without hampering ||-ism
- Fork/Join models constrain control and data dependences
  - Tasks can only be created after all data dependences satisfied
  - Necessitates ordering task creation to conform to that restriction
- <u>May hamper performance</u>

Macro-dataflow for intuitive ||-ism

- Kernel based programming



- Build a task graph of kernel instantiations

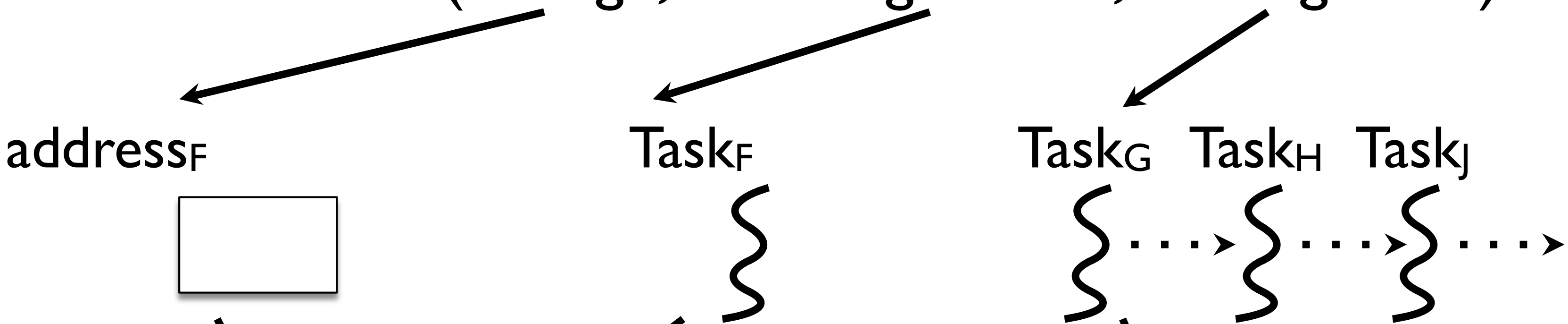


- Restrict dependences to true dependences
  - race-freedom, determinism

- <u>Provides productivity</u>

Futures [Baker & Hewitt 1977]

- `future` = (*storage*, *resolvingProcess*, *waitingTasks*)




```
Future F = {stmt1;…; return v;}

task g = {stmt; F.get();…;}
```

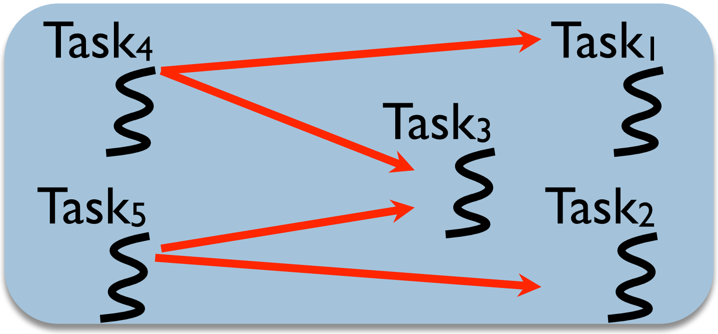
Data-Driven Futures (DDFs) & Data-Driven Tasks (DDTs)

`DataDrivenFuture` = (*storage*, *waitingTasks*)

- Creation
  - Create an empty Data-Driven Future (DDF) object
- Resolution (`put`) (*resolvingProcess*)
  - Resolve what value a DDF is referring to
- Data-Driven Tasks (DDTs) (`async await(...)`)
  - A task provides a consumer list of DDFs on declaration
  - A task can only read DDFs that it is registered to
- Difference from futures:
  - Creation of container (DDF) and computation (DDT) are separate events

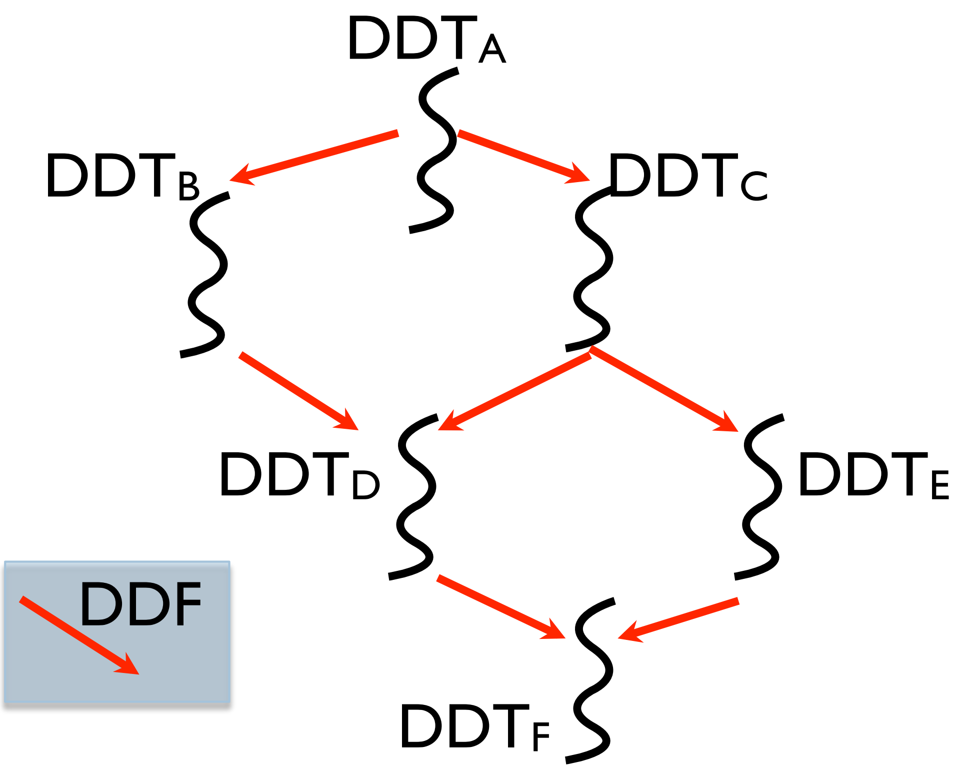
DDF/DDT Code Sample


```
DataDrivenFuture left = new DataDrivenFuture ();
DataDrivenFuture right = new DataDrivenFuture();
finish {
    async await ( left ) useLeftChild(left); // Task1
    async await ( right ) useRightChild(right); // Task2
    async await ( left, right ) useBothChildren( left, right ); // Task3
    async left.put(leftChildCreator()); // Task4
    async right.put(rightChildCreator()); // Task5
}
```



DDTs provide

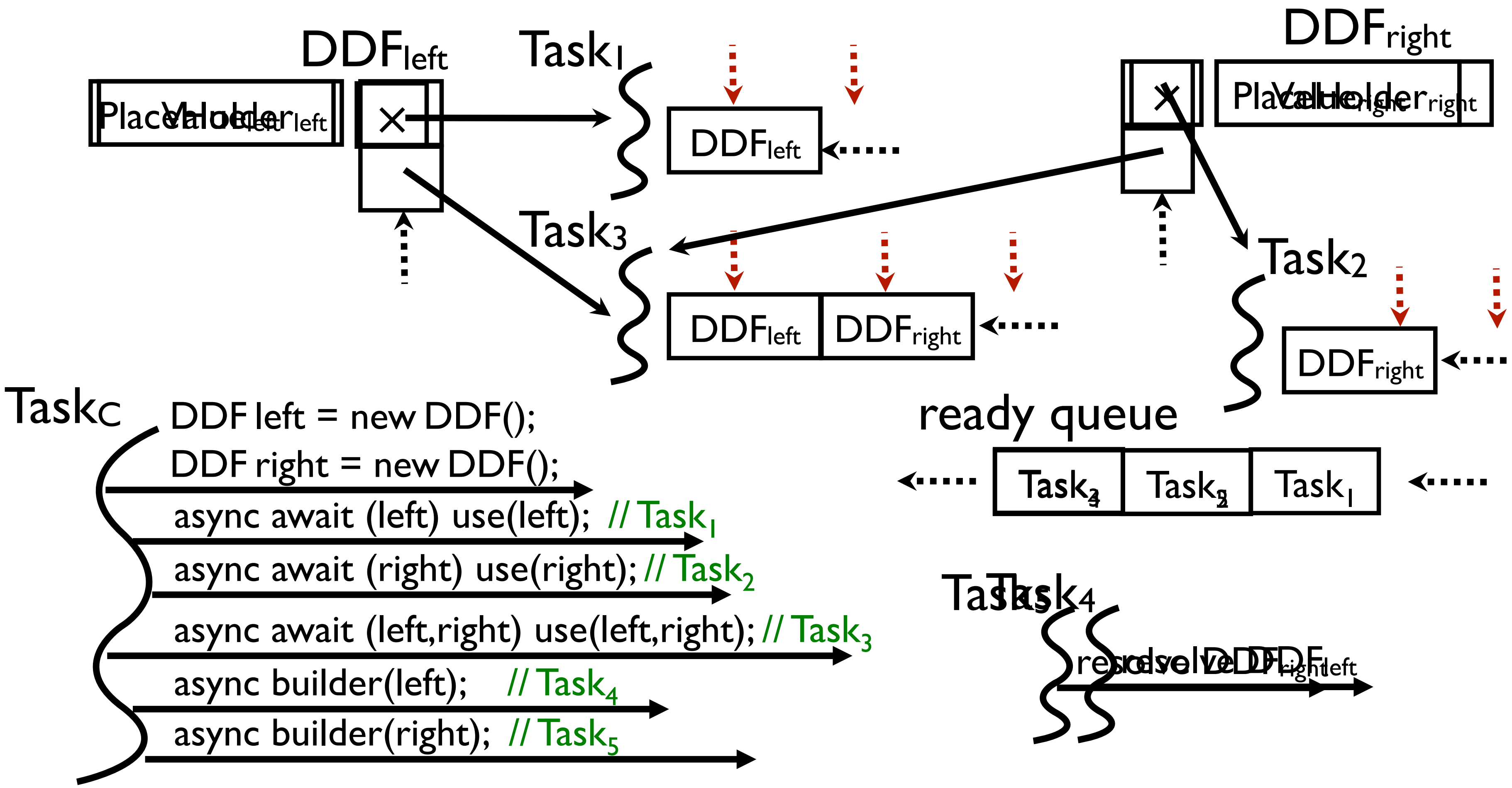
- Non-series-parallel task dependence graph support
  - Less restricted parallelism
  - Better scheduling opportunities



- Single assignment (SA)
  - Race-freedom on DDF accesses
  - Determinism if all shared data is expressed as DDFs
- SA-value lifetime restriction
  - Smaller than graph lifetime
  - DDF creator:
    - Provides DDF reference to producers and consumers
  - DDF lifetime depends on
    - Creator lifetime
    - Resolver lifetime
    - Consumers' lifetimes

Data-Driven Scheduling

- Steps register self to items wrapped into DDFs



Mapping Macro-Dataflow to Task-Parallelism

- Control & data dependences as first level constructs
  - Task-parallel frameworks have them coupled e.g., OpenMP, Cilk
- Kernel instantiations may have multiple predecessors
  - Need to wait for all
  - Staged readiness concepts
    - Created (control dependence satisfied)
    - Data dependences satisfied
    - Schedulable / Ready
- DDTs provide a natural implementation for Macro-Dataflow
  - Every kernel instantiation is a DDT
  - Data dependences between DDTs are expressed through DDFs
  - Provides race freedom

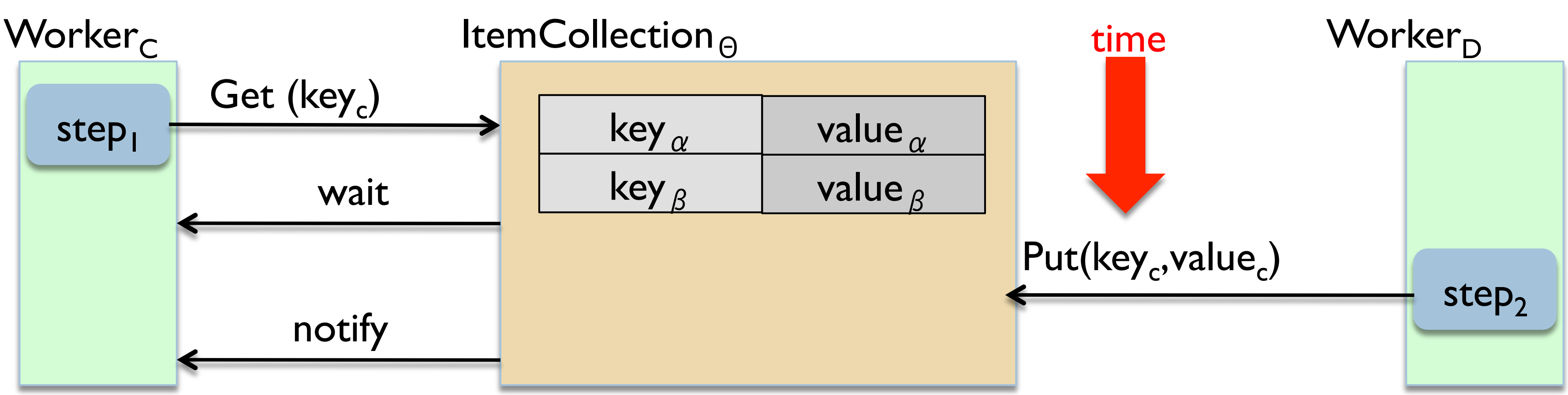
Experimental Results

- Compared DDT implementation with four macro-data schedulers from past work
  - that used Concurrent Collections (CnC)
  - CnC uses global data collections to synchronize tasks

- DDT/DDF results obtained at task-parallel level
  - without allocating global data collections
  - CnC can be automatically translated to DDFs (ongoing work)

Blocking Schedulers

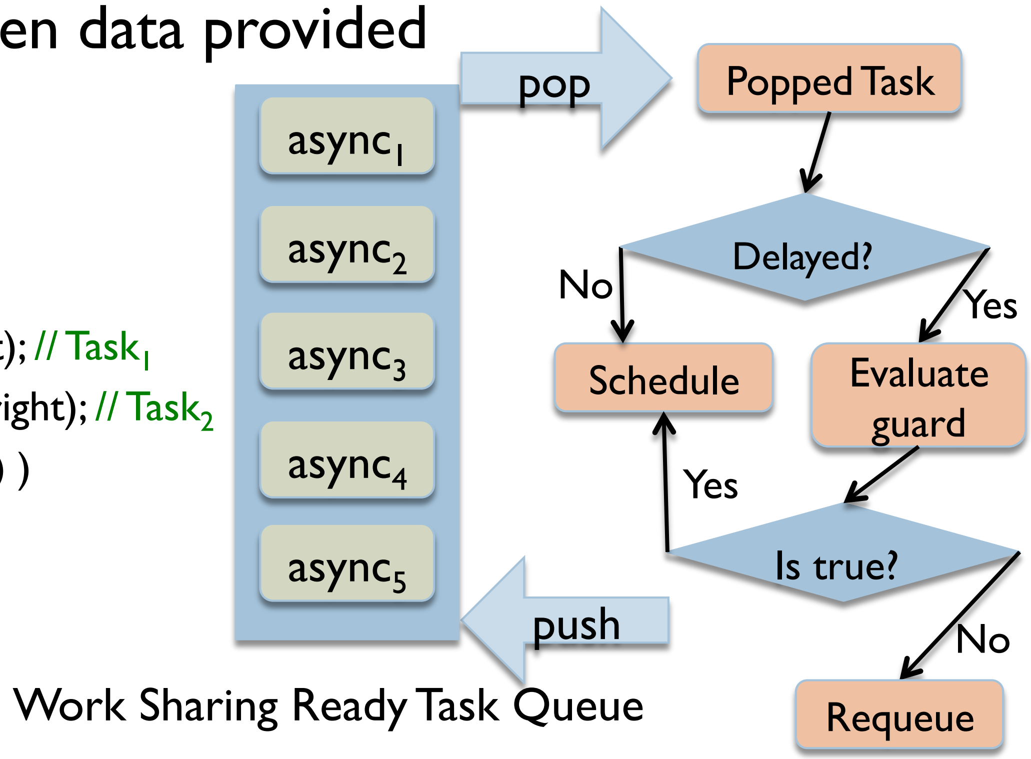
- Use Java wait/notify for premature data access
- Blocking granularity
  - Instance level vs Collection level (fine-grain vs. coarse-grain)
- A blocked task blocks an entire worker thread
  - Need to create more worker threads to avoid deadlock



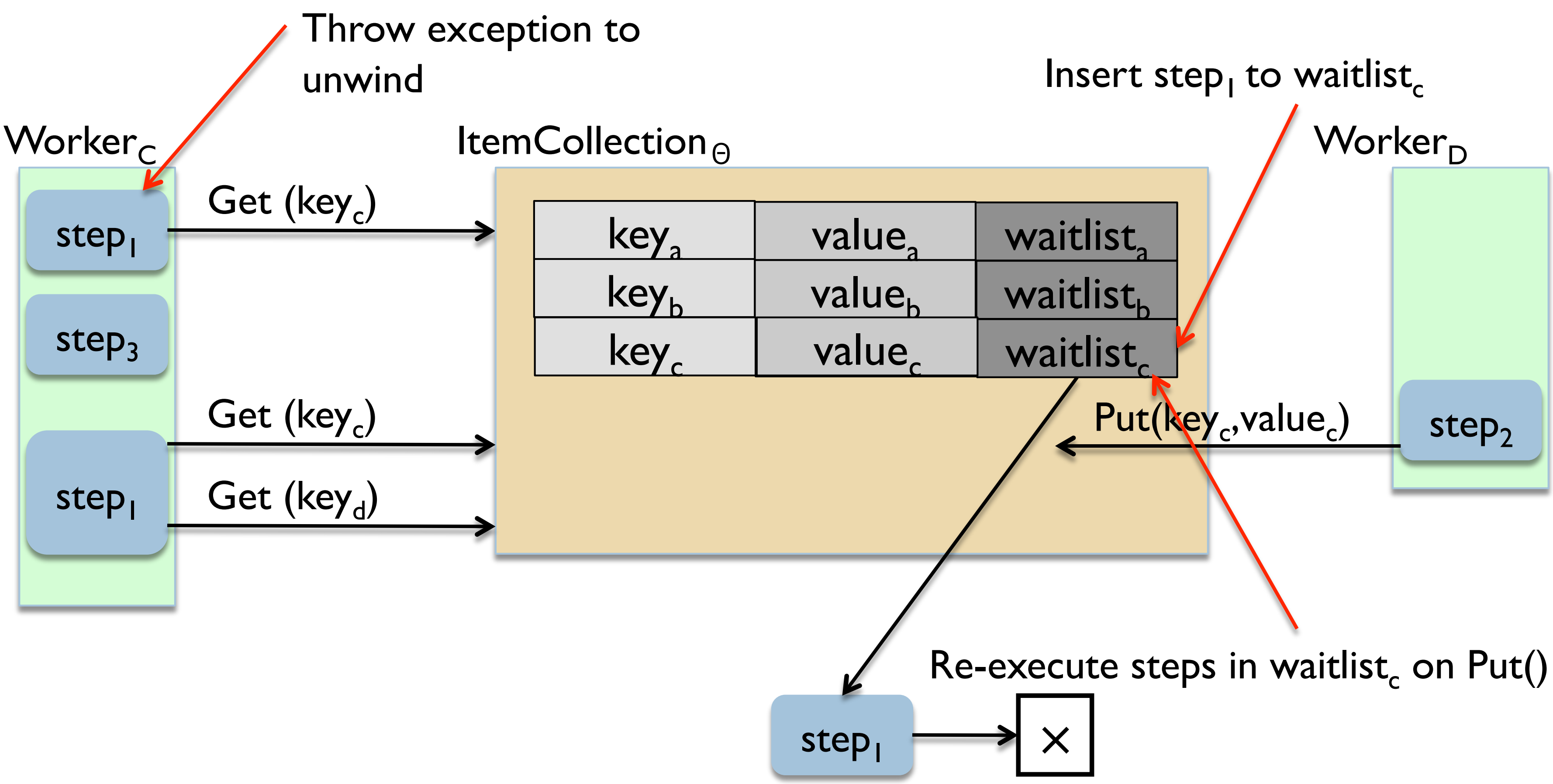
Delayed `async` Scheduling

- Every kernel instantiation is a guarded execution
  - Guard condition is the availability of input data
  - Task can be created eagerly before input data is available
  - Promoted to <u>ready</u> when data provided

```
Value left = new Value ();
Value right = new Value ();
finish {
  async when ( left.isReady() ) useLeftChild(left); // Task1
  async when ( right.isReady()) useRightChild(right); // Task2
  async when ( right.isReady() && left.isReady() )
    useBothChildren( left, right ); // Task3
  async left.put(leftChildCreator()); // Task4
  async right.put(rightChildCreator()); // Task5
}
```



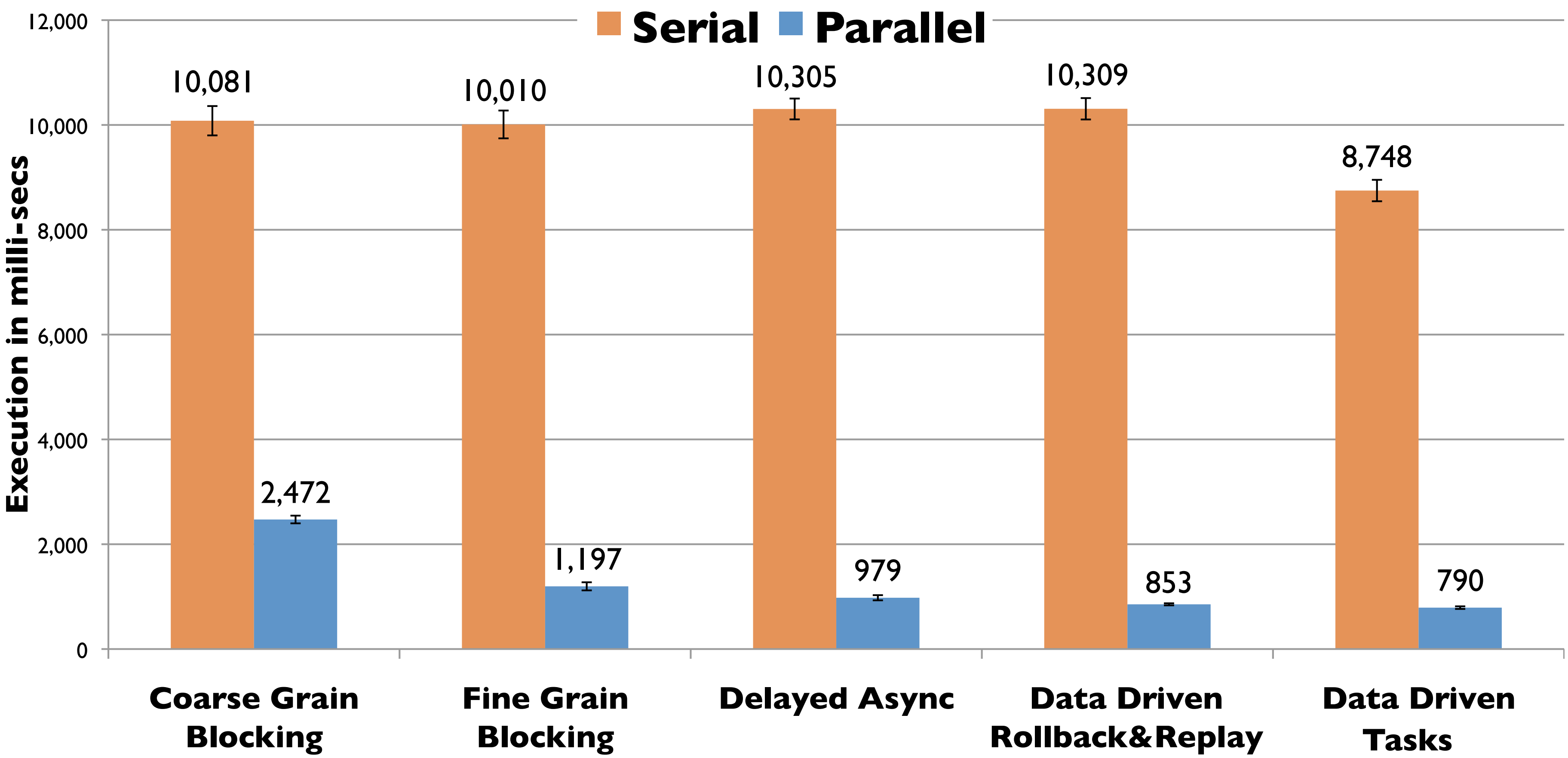
Data Driven Rollback & Replay



Experimental Setup

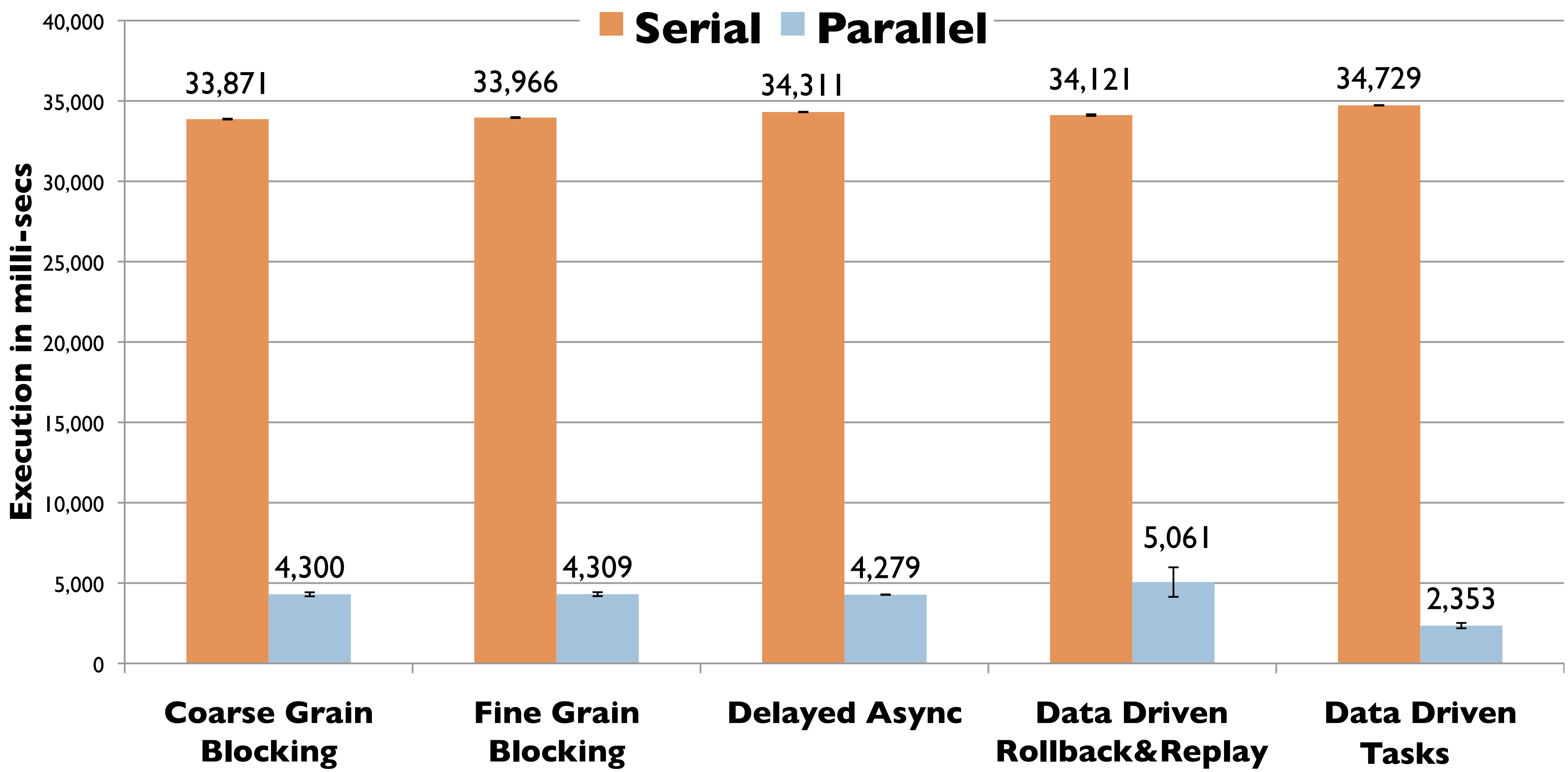
- 4-socket Xeon quad-core Intel E7730 2.4 GHz
  - Shared 3MB L2 cache per pair of cores.
  - Main memory 32 GBs.
  - #worker threads:16
- 8-way SMT 8-core Niagara Sun UltraSPARC T2
  - Shared 4MB L2 cache
  - #worker threads: 64
- 32-bit Sun Hotspot JDK 1.6 JVM
  - GCC 4.1.2 for JNI
- 30 runs for statistical soundness
- Read 'Serial' as single-threaded execution of || code

Cholesky decomposition



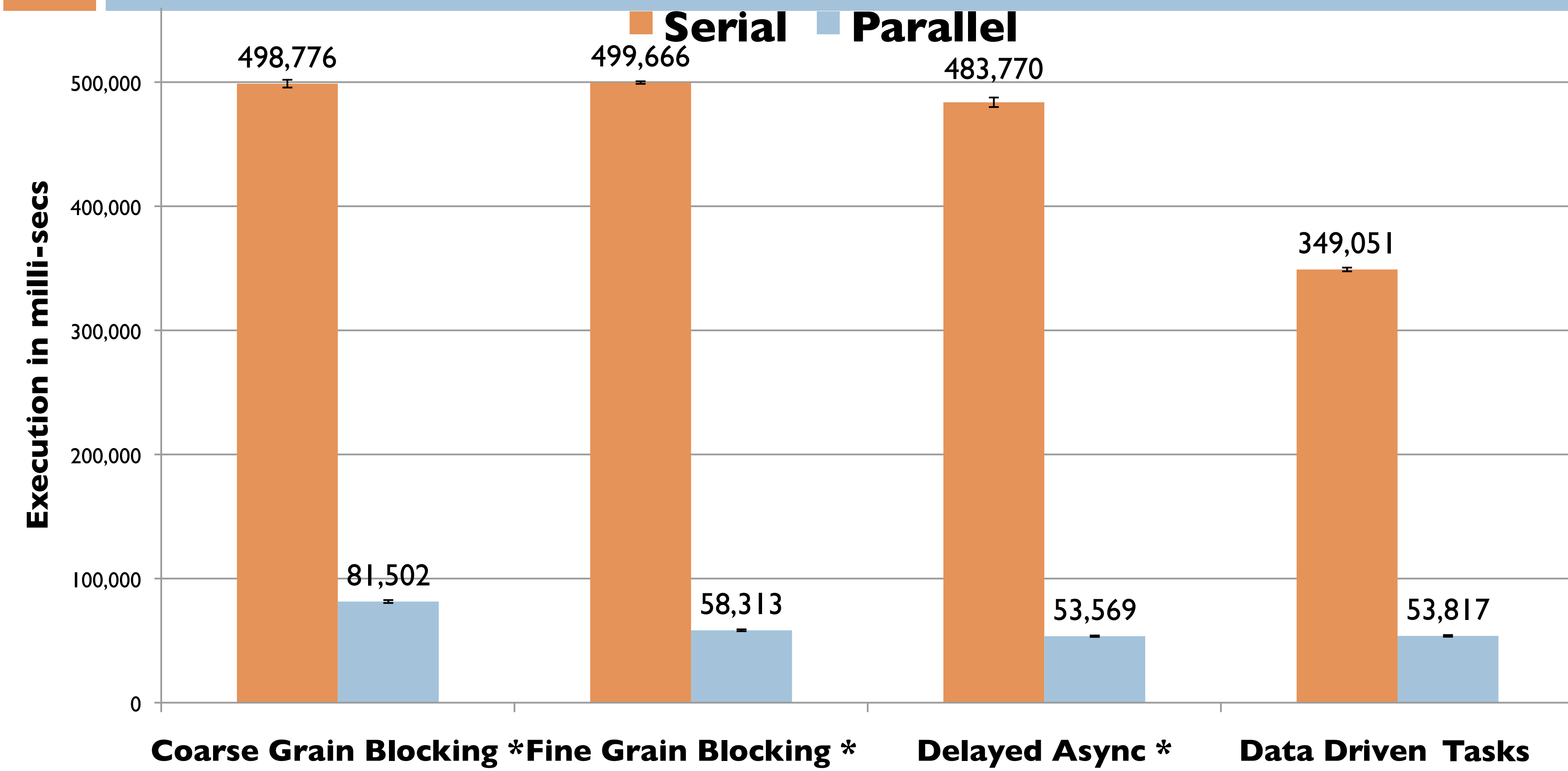
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on 16-core Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Black-Scholes formula (PARSEC)



Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on 16-core Xeon with input size 1,000,000 and with tile size 62,500

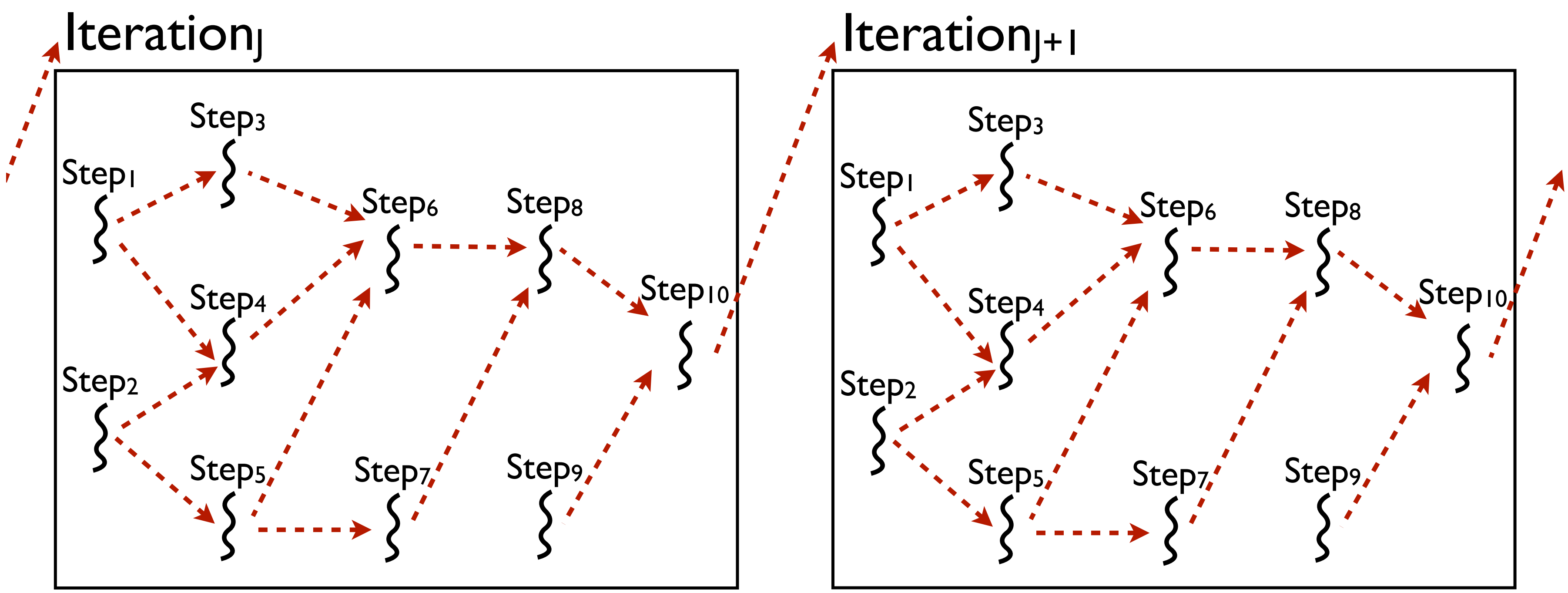
Rician Denoising (Medical Imaging)



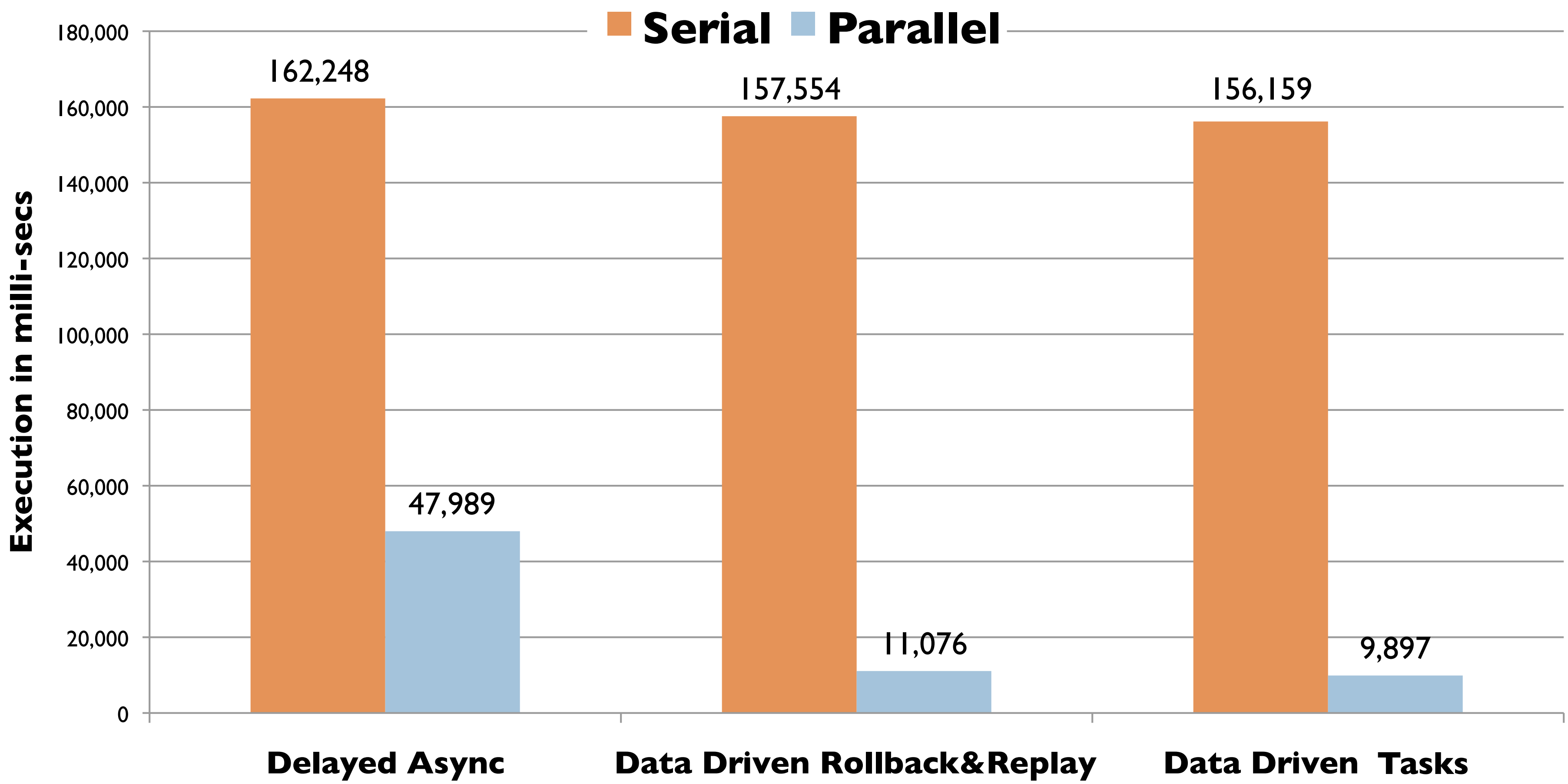
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on Xeon with input image size 2937 × 3872 and with tile size 267 × 484

* Explicit memory management required for non-DDT schedules to avoid out-of-memory exception

Heart Wall Tracking Dependence Graph



Heart Wall Tracking (Rodinia)



Minimum execution times of 13 runs of single threaded and 16-threaded executions for Heart Wall Tracking CnC application with C steps on Xeon with 104 frames

Related Work

- Futures
  - Can build arbitrary task graphs
  - `get()/force()` is usually a blocking operation
  - `future` task creation is bound to container at creation time
- Dataflow
  - Typically blocks on one datum (Ivar) at a time, unlike async await (…)
- Nabbit (Cilk library)
  - Can build arbitrary task graphs, more explicit than DDTs
  - No garbage collection and unwinding of task graph
- Concurrent Collections (CnC)
  - Globalized data collections and general tags (keys) makes memory management challenging
  - DDTs can be used to obtain more efficient implementations of CnC

Conclusions

- Data-Driven Futures and Data-Driven Tasks
  - help build arbitrary task graphs and extend task-parallel frameworks
  - introduce the more-intuitive macro-dataflow to programmers on task-parallel frameworks
  - support Data-Driven scheduling that outperforms alternative schedulers in both execution time and memory requirements
  - help to implement blocking in tasks without blocking workers

Future Work

- Compile Concurrent Collections down to DDTs
- Compiler optimizations to move DDF allocations to further reduce lifetimes
- Hierarchical DDTs for granularity optimizations
- Work-stealing support for DDTs
- Use DDTs to implement all blocking synchronizations without blocking worker, i.e. replace each waiting continuation as a DDT
- Locality aware scheduling with DDTs

For a hands-on trial, visit http://habanero.rice.edu/hj

http://habanero.rice.edu/cnc