SCHEDULING
MACRO-DATAFLOW PROGRAMS
ON
TASK-PARALLEL RUNTIME SYSTEMS

SAĞNAK TAŞIRLAR

Thesis

Our thesis is that advances in task parallel runtime systems can enable a macro-dataflow programming model, like Concurrent Collections (CnC), to deliver productivity and performance on modern multicore processors.

Approach

- Macro-dataflow for expressiveness
  - Determinism
  - Race/deadlock freedom
  - Higher level abstraction

- Task parallel runtimes for performance
  - Portable scalability
  - Contemporary consensus

Motivation

- Parallelism not accessible to those who need it most
  - Imposed serial thinking
  - Parallelism for the masses, not just computer scientists

- Parallel programming models of today:
  - Hide machine details but expose parallelism details
  - Constrain expressiveness

Contributions

- Scheduling CnC on Habanero Java ★
- Evaluation of scheduling performance for CnC ★
- Introduction of Data Driven Futures (DDF) construct
- Implementation of DDF construct
- Implementation and evaluation of data driven runtime with DDFs

★ Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, Sağnak Taşırlar, "The CnC Programming Model ", submitted for publication to the Journal of Supercomputing, 2010

Outline

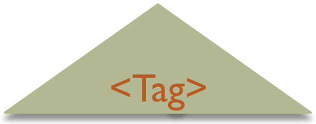
- Background
- CnC Scheduling
- Data Driven Futures
- Results
- Wrap up

Dynamic Task Parallelism

- Properties
  - Over exposure of parallelism
  - Scales up/down with # of cores
  - Scheduling maps sets of tasks to threads at runtime
- Habanero Java (HJ) employs:
  - Finish/async parallelism
    - Feeds child tasks through lexical scope
  - Work sharing/stealing runtime scheduling

CnC concepts

- Step
  - Computation abstraction
  - Side effect free
  - Functional w.r.t. input
  - Special step: Environment
- Item
  - Dynamic single assignment
  - Value not storage
- Tag
  - Data tag to index items
  - Control tag to index steps

Collection	Graphical Notation	Textual Notation
Step	 (SomeStep)	(SomeStep)
Item	 [SomeItem]	[SomeItem]
Tag	 <Tag>	<Tag>

Concurrent Collections model

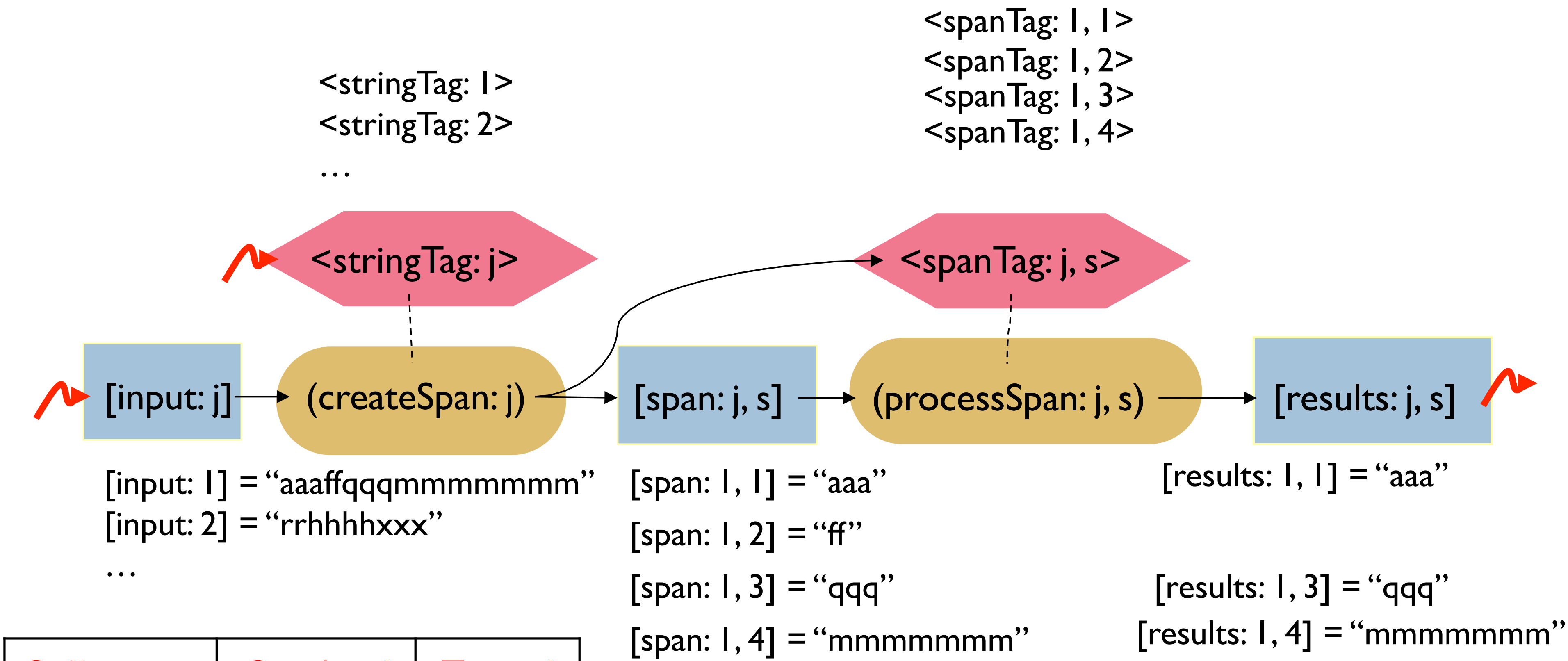
- Can be classified as:
  - Declarative
  - Deterministic
  - Dynamic single assignment
  - Macro-dataflow
  - Coordination language
- Goal: consider only <u>semantic</u> ordering constraints
  - Inherent in the application not the implementation
  - Will be described by the CnC graph

Example Program Specification

- Break up an input string
  - Sequences of repeated single characters
- Filter allowing only
  - Sequences of odd length

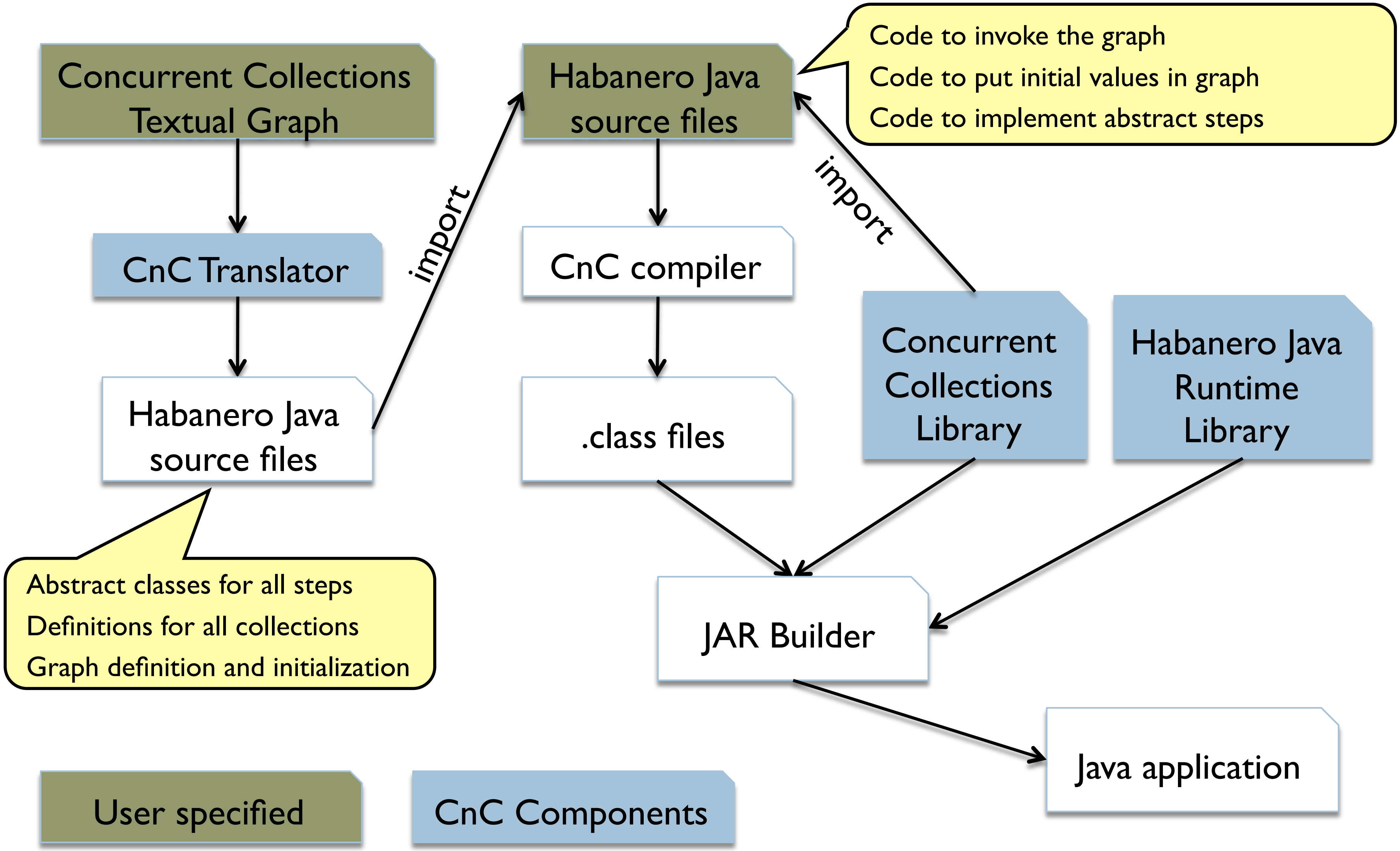
Input string	Sequences of repeated characters	Filtered sequences
“aaaffqqqmmmmmmm”	“aaa”	“aaa”
	“ff”	
	“qqq”	“qqq”
	“mmmmmmm”	“mmmmmmm”

CnC Implementation of Example Program



Collection	Graphical	Textual
Step	( )	(…)
Item	[ ]	[…]
Tag	<>	< … >

CnC-Habanero Java build model



Code to invoke the graph
Code to put initial values in graph
Code to implement abstract steps

Abstract classes for all steps
Definitions for all collections
Graph definition and initialization

Outline

- ☐ Background
- ☐ CnC Scheduling
- ☐ Data Driven Futures
- ☐ Results
- ☐ Wrap up

CnC Scheduling Challenges

- Control & data dependences are first level constructs
    - Task parallel frameworks have them coupled
- Step instances have multiple predecessors
    - Need to wait for all predecessors
    - Layered readiness concepts
        - Control dependence satisfied
        - Data dependence satisfied
        - Schedulable / Ready

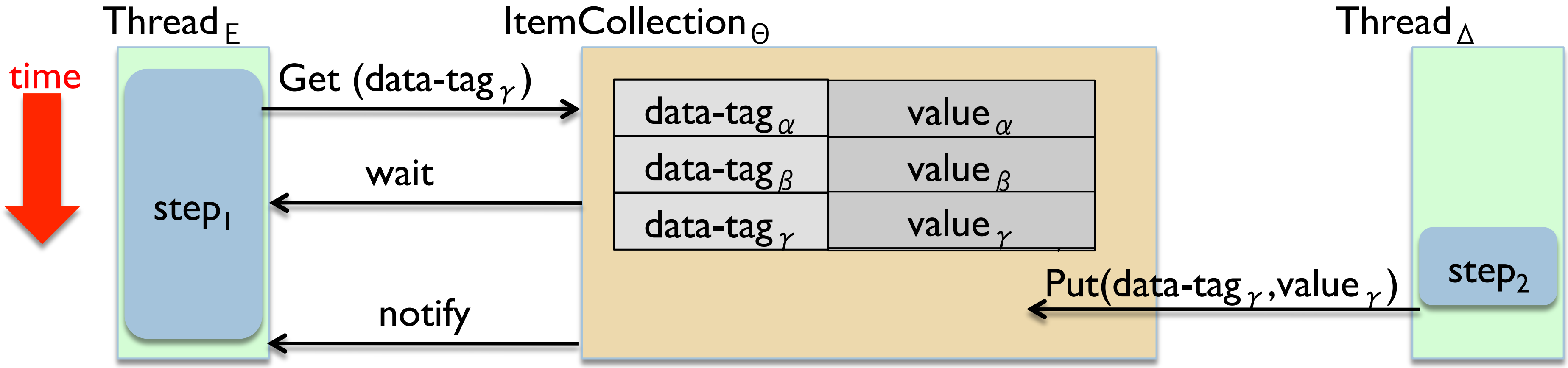
Eager scheduling

- Assume control dependence satisfaction is readiness
  - Conforms to task parallel runtime assumption

- Wait till data dependences satisfaction for safety
  - Block on data prematurely tried to be read
  - Discard task reading prematurely, replay when data arrive

Blocking Eager CnC Schedulers

- Use Java wait/notify for premature data access
- Blocking granularity
  - Instance level vs Collection level
- Blocked task blocks whole thread
  - Deadlock possibility
  - Need to create more threads as threads block



time

Thread$_E$

step$_1$

Get (data-tag$_\gamma$)

wait

notify

ItemCollection$_\Theta$

data-tag$_\alpha$	value$_\alpha$
data-tag$_\beta$	value$_\beta$
data-tag$_\gamma$	value$_\gamma$

Put(data-tag$_\gamma$,value$_\gamma$)

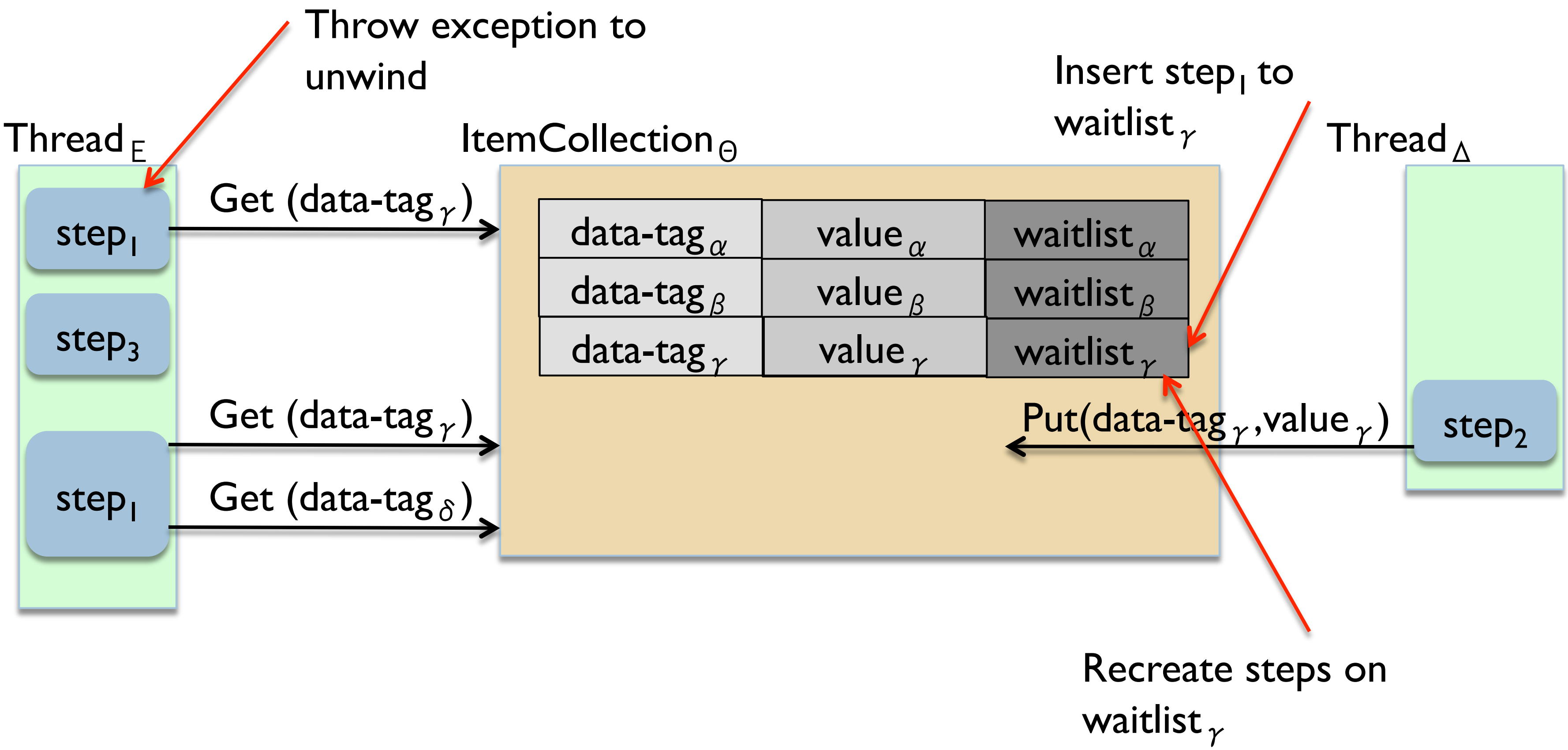
Thread$_\Delta$

step$_2$

Data Driven Rollback & Replay

- Alternative eager scheduling
- Blocking scheduler suffers from
  - Expensive recovery from premature read
    - Blocks whole thread
    - Creates new thread
    - Switch context to the new thread on every failure
- Inform item instance on failed task and discard task
  - Throw an exception to unwind failed task
  - Catch by runtime and continue with another ready task
  - Recreate task when needed item arrives

Data Driven Rollback & Replay



Throw exception to unwind

Thread$_E$

step$_1$

step$_3$

step$_1$

Get (data-tag$_\gamma$)

Get (data-tag$_\gamma$)

Get (data-tag$_\delta$)

ItemCollection$_\Theta$

data-tag$_\alpha$	value$_\alpha$	waitlist$_\alpha$
data-tag$_\beta$	value$_\beta$	waitlist$_\beta$
data-tag$_\gamma$	value$_\gamma$	waitlist$_\gamma$

Put(data-tag$_\gamma$,value$_\gamma$)

Insert step$_1$ to waitlist$_\gamma$

Thread$_\Delta$

step$_2$

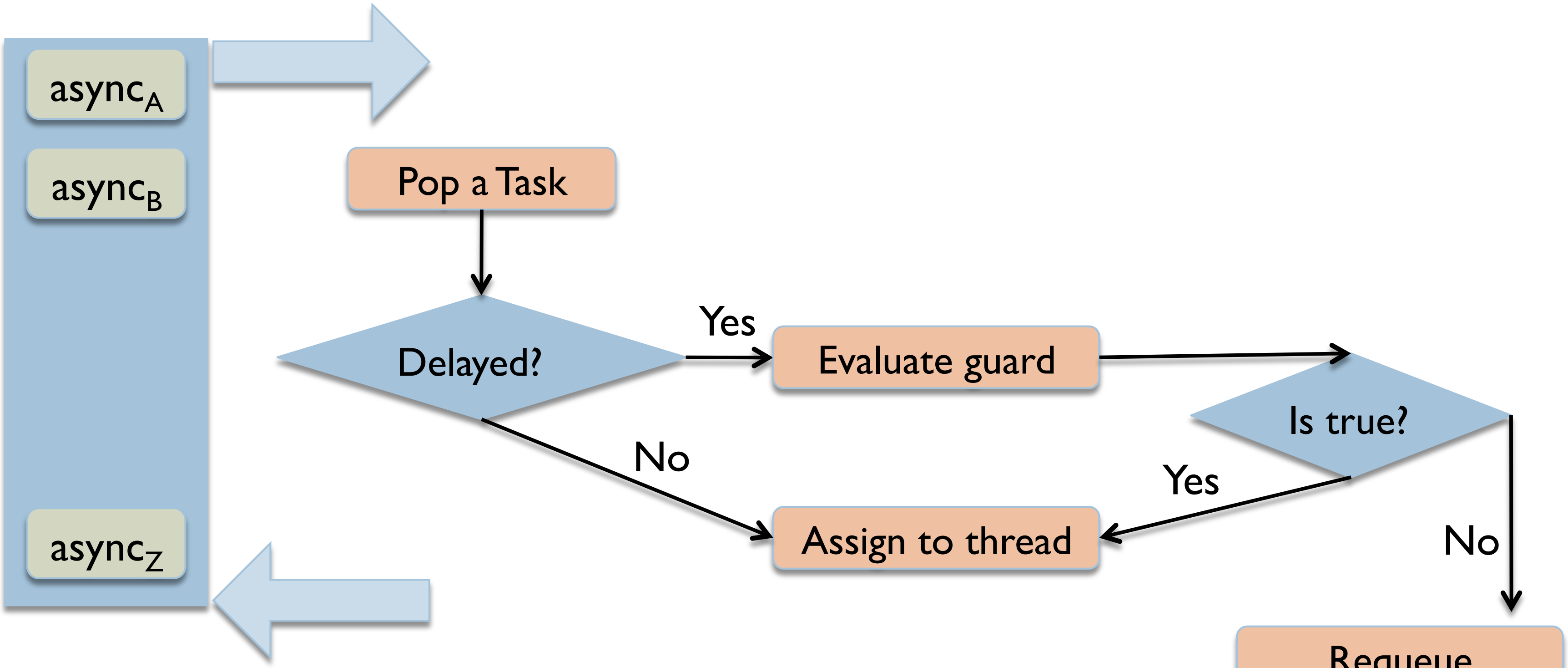
Recreate steps on waitlist$_\gamma$

Data Driven Scheduling

- Do not create tasks until data dependences satisfied
  - No failure, no recovery
  - Restrict computation frontier to <u>ready</u> tasks
- Evaluation of data readiness condition
  - Busy waiting on data (delayed async scheduling)
  - Dataflow like readiness (data driven scheduling)
    - Register tasks on data
    - Data notifies consumer tasks when created

Delayed Asyncs

- Guarded execution construct for HJ
  - Promote to async when guard evaluates to true



async_A

async_B

async_Z

Pop a Task

Delayed?

Yes

Evaluate guard

No

Is true?

Yes

Assign to thread

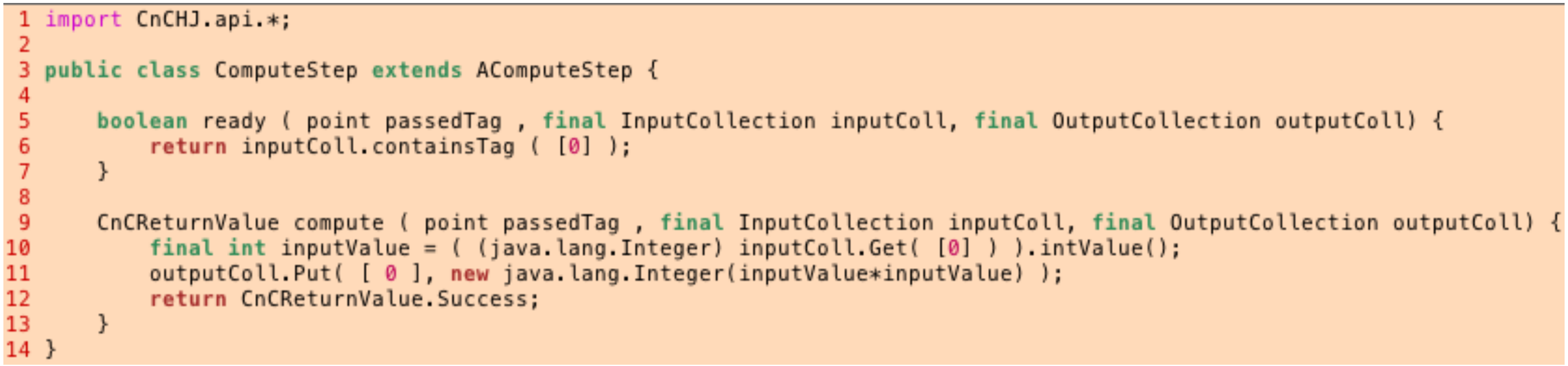
No

Requeue

Work Sharing Ready Task Queue

Delayed Async Scheduling

- Every CnC step is a guarded execution
  - Guard condition is the availability of items to consume
  - Task still created eagerly when provided control
  - Promotes to <u>ready</u> when data provided



```java
import CnCHJ.api.*;

public class ComputeStep extends AComputeStep {

    boolean ready ( point passedTag , final InputCollection inputColl, final OutputCollection outputColl) {
        return inputColl.containsTag ( [0] );
    }

    CnCReturnValue compute ( point passedTag , final InputCollection inputColl, final OutputCollection outputColl) {
        final int inputValue = ( (java.lang.Integer) inputColl.Get( [0] ) ).intValue();
        outputColl.Put( [ 0 ], new java.lang.Integer(inputValue*inputValue) );
        return CnCReturnValue.Success;
    }
}
```

Outline

- ☐ Background
- ☐ CnC Scheduling
- ☐ Data Driven Futures
- ☐ Results
- ☐ Wrap up

Data Driven Futures (DDFs)

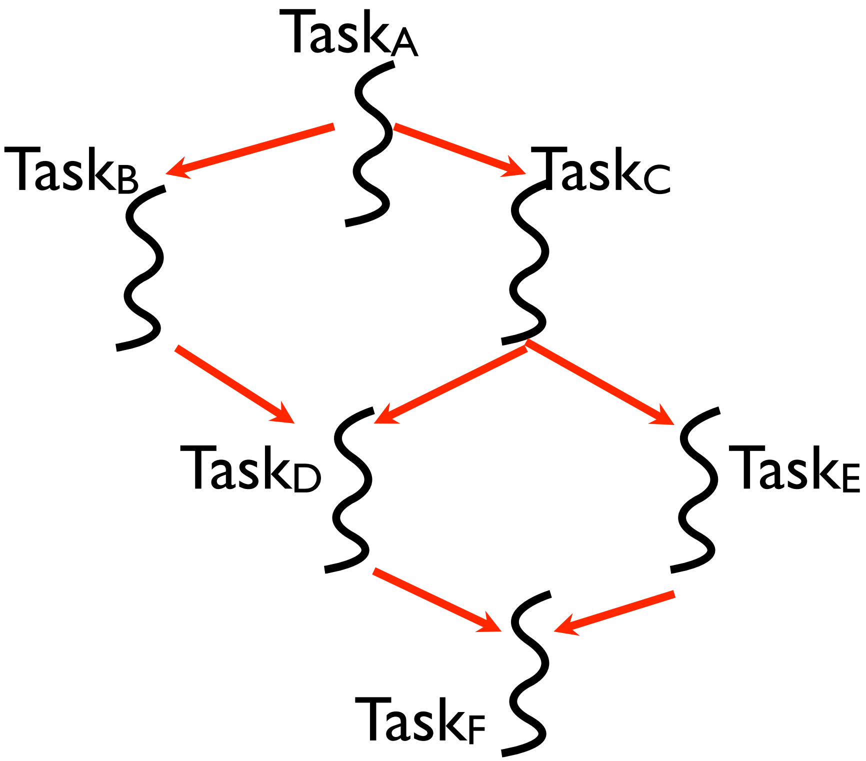
- Task parallel synchronization construct
  - Acts as a reference to single assignment value
- 🟦 Creation
  - Create a dangling reference object
- 🟩 Resolution (Put)
  - Resolve what value a DDF is referring to
- 🟥 Registration (Await)
  - A task provides a consume list of DDFs on declaration
  - A task can only read DDFs that it is registered to

Data Driven Futures (DDFs)

```
DataDrivenFuture leftChild = new DataDrivenFuture();
DataDrivenFuture rightChild = new DataDrivenFuture();
finish {
    async leftChild.put(leftChildCreator());
    async rightChild.put(rightChildCreator());
    async await ( leftChild ) useLeftChild(leftChild);
    async await ( rightChild ) useRightChild(rightChild);
    async await ( leftChild, rightChild )
                    useBothChildren( leftChild, rightChild );
}
```

Contributions of DDFs

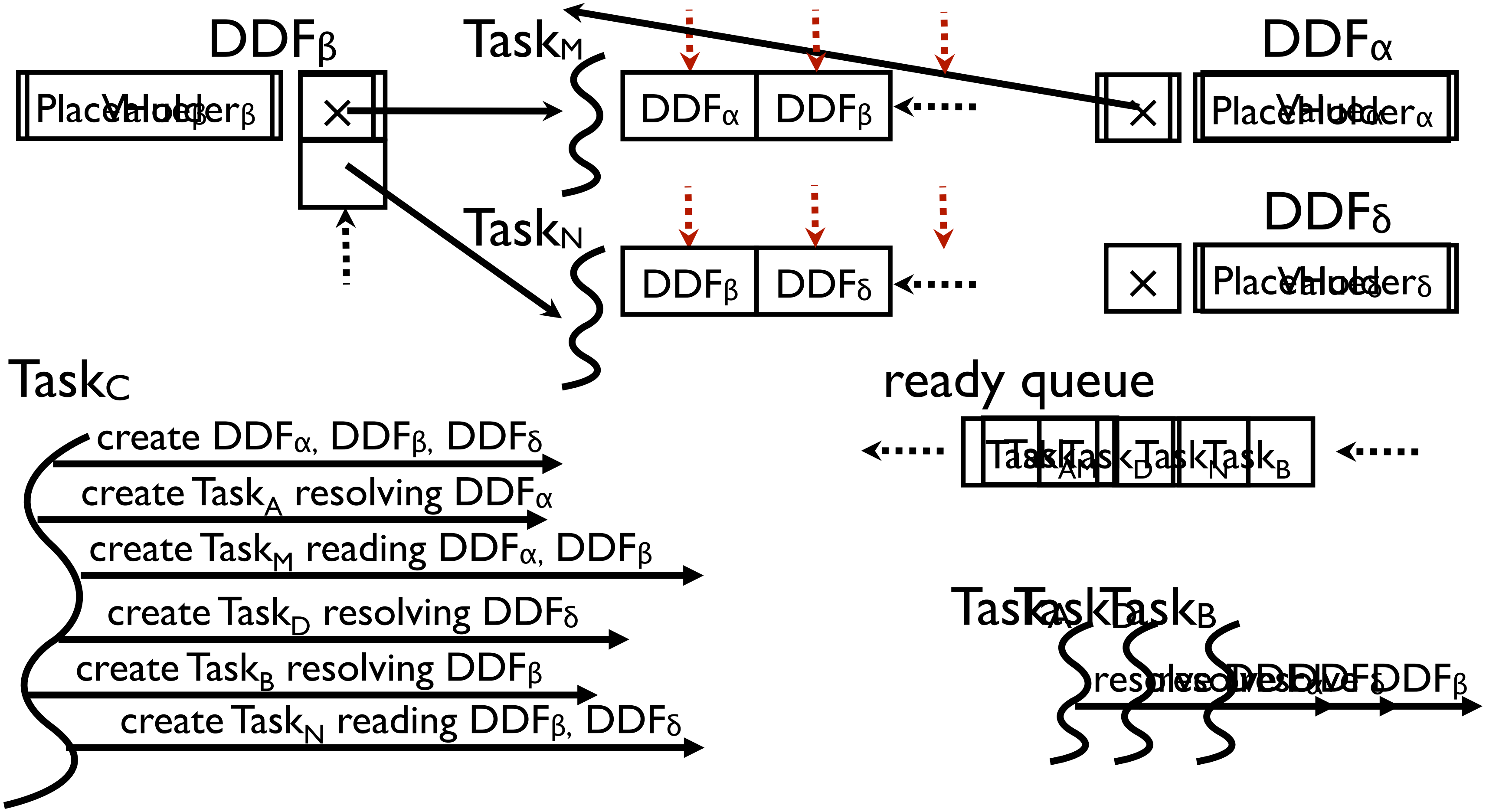
- Non-series-parallel task dependency graphs support



- Memory footprint reduction
  - Exposes only ready parts of the execution frontier
  - Not global lifetime
  - Creator:
    - feeds consumers
    - gives access to producer
  - Lifetime restricted to
    - Creator lifetime
    - Resolver lifetime
    - Consumers lifetimes
  - Can be garbage collected on a managed runtime

Data Driven Scheduling

- Steps register self to items wrapped into DDFs



create DDFα, DDFβ, DDFδ

create $Task_A$ resolving DDFα

create $Task_M$ reading DDFα, DDFβ

create $Task_D$ resolving DDFδ

create $Task_B$ resolving DDFβ

create $Task_N$ reading DDFβ, DDFδ

Outline

- Background
- CnC Scheduling
- Data Driven Futures
- Results
- Wrap up

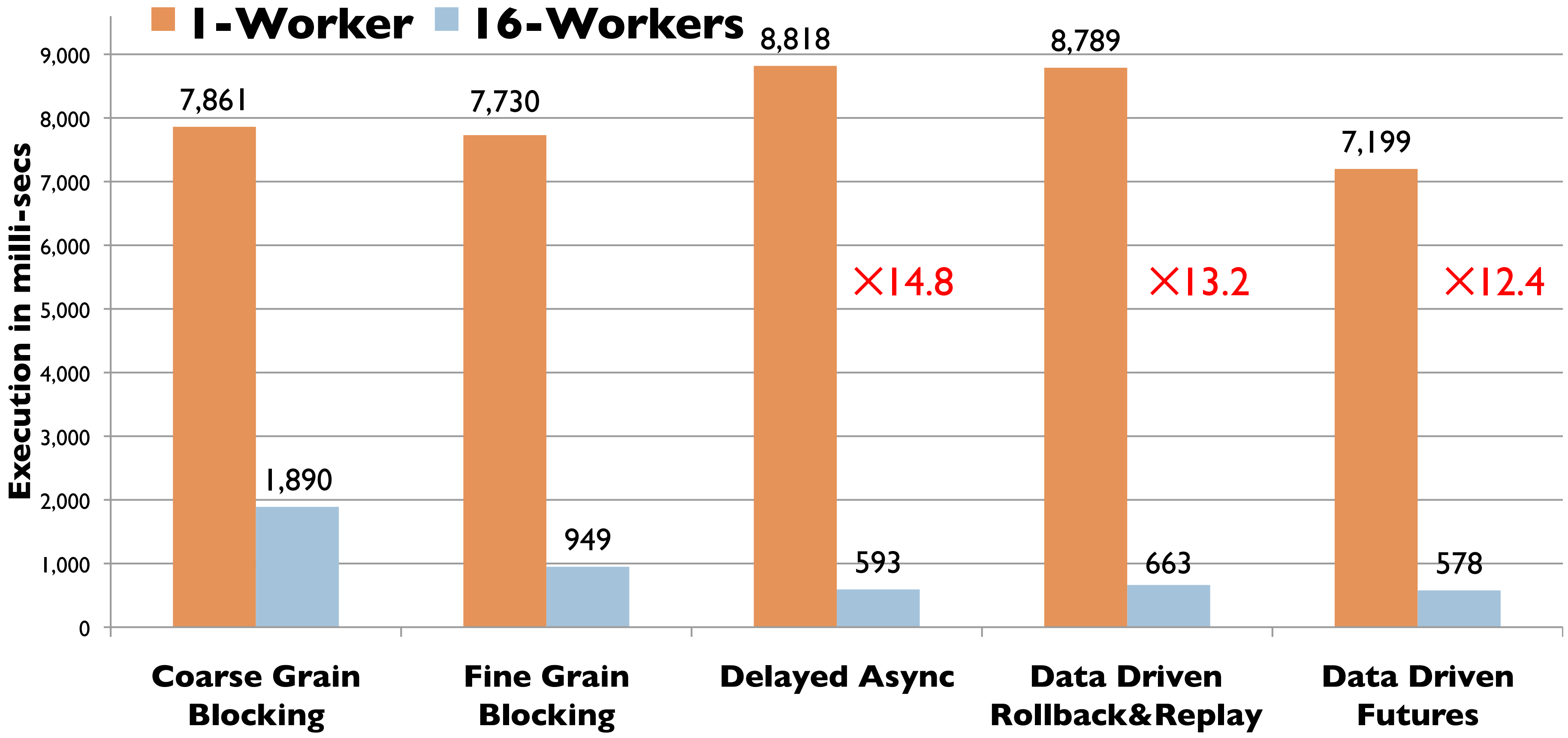
Performance Evaluation Legend

- Coarse Grain Blocking
  - Eager blocking scheduling on item collections for CnC-HJ
- Fine Grain Blocking
  - Eager blocking scheduling on item instances for CnC-HJ
- Delayed Async
  - Data Driven scheduling via HJ delayed asyncs for CnC-HJ
- Data Driven Rollback & Replay
  - Eager scheduling with replay and notifications for CnC-HJ
- Data Driven Futures
  - Hand coded CnC application equivalent on HJ with DDFs

Cholesky Decomposition Introduction

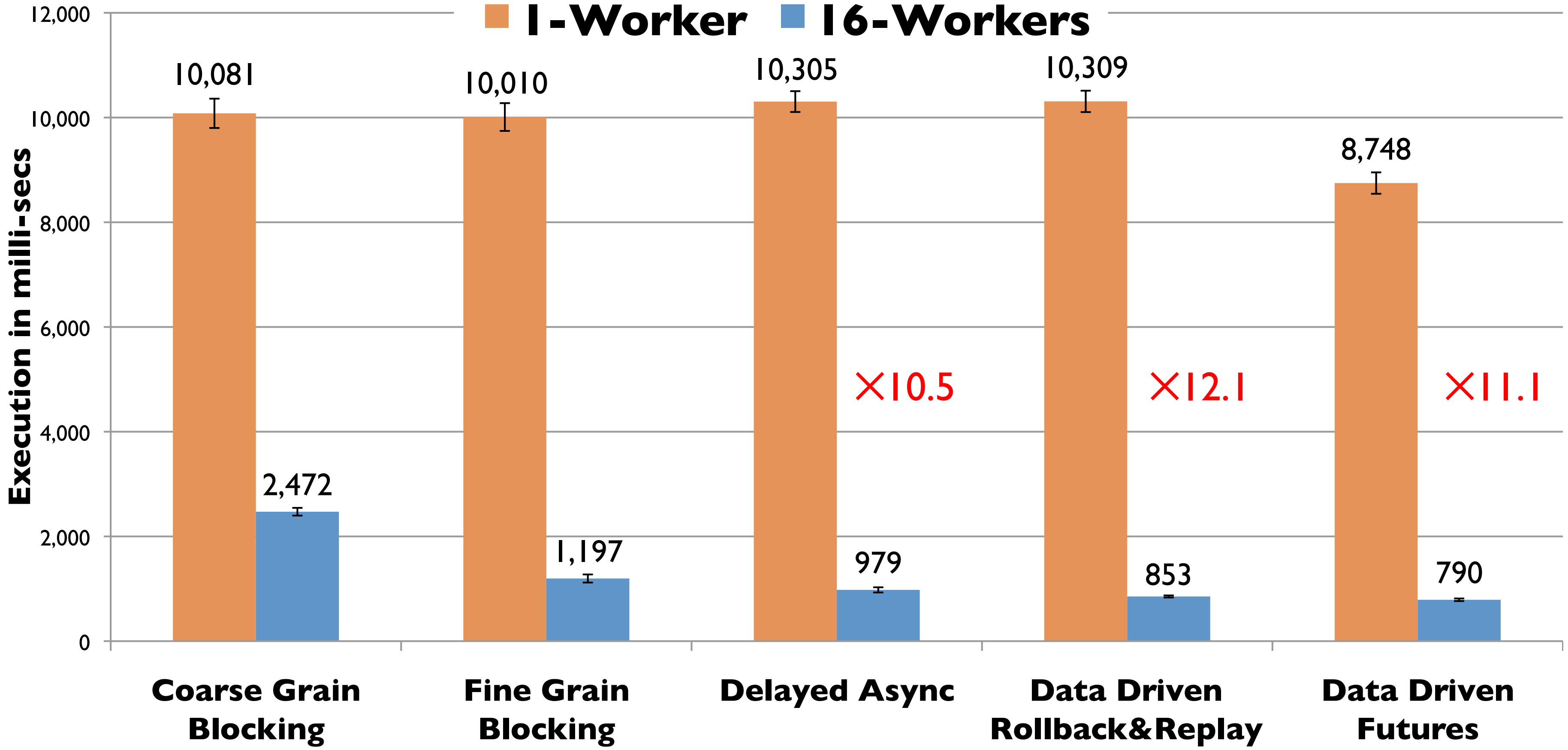
- Dense linear algebra kernel
- Three inherent kernels
  - Need to be pipelined for best performance
  - Loop parallelism within some kernels
  - Data parallelism within some kernels
- CnC shown to beat optimized libraries, like IntelMKL

Cholesky Decomposition on Xeon



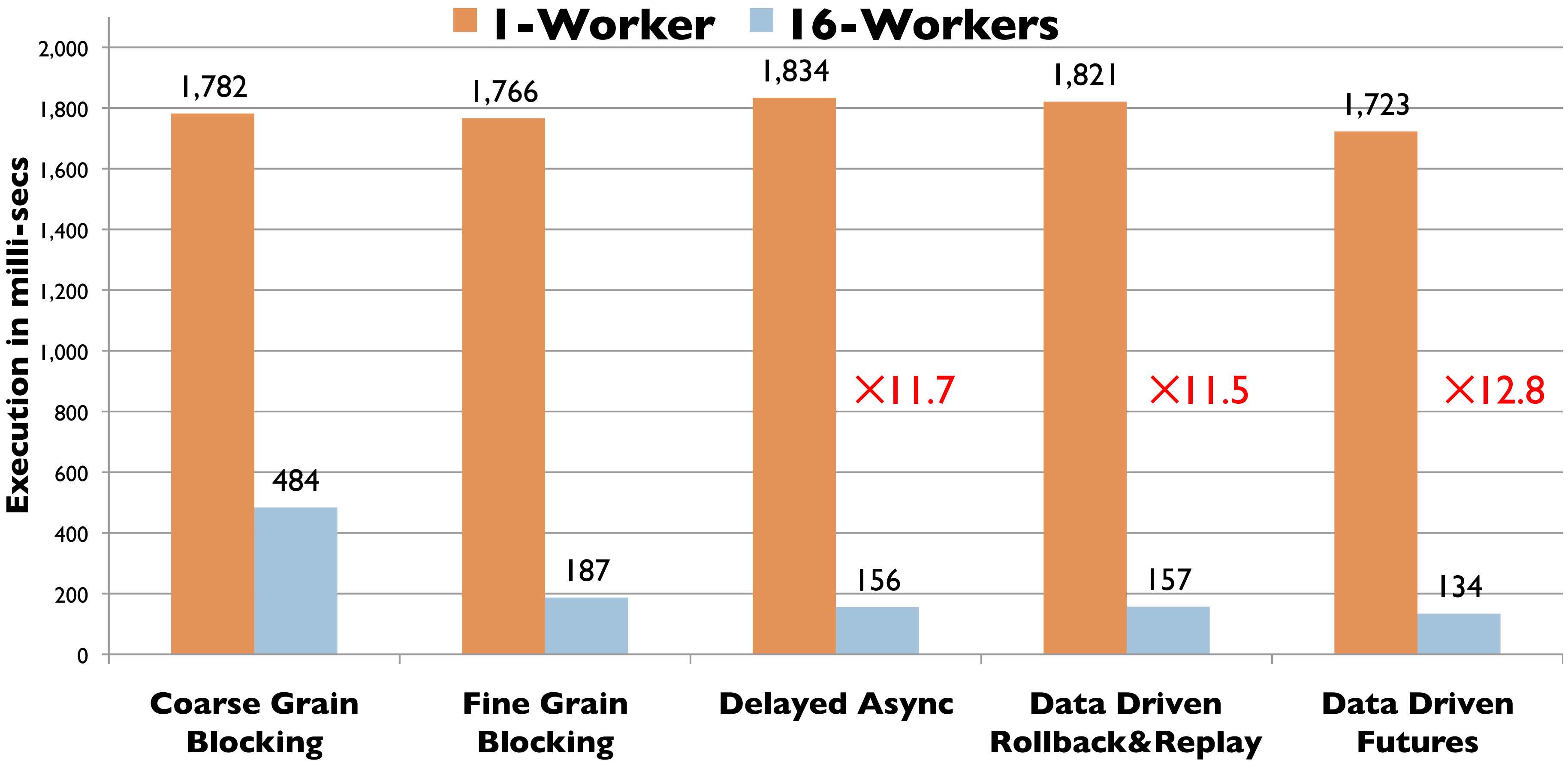
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky Decomposition on Xeon



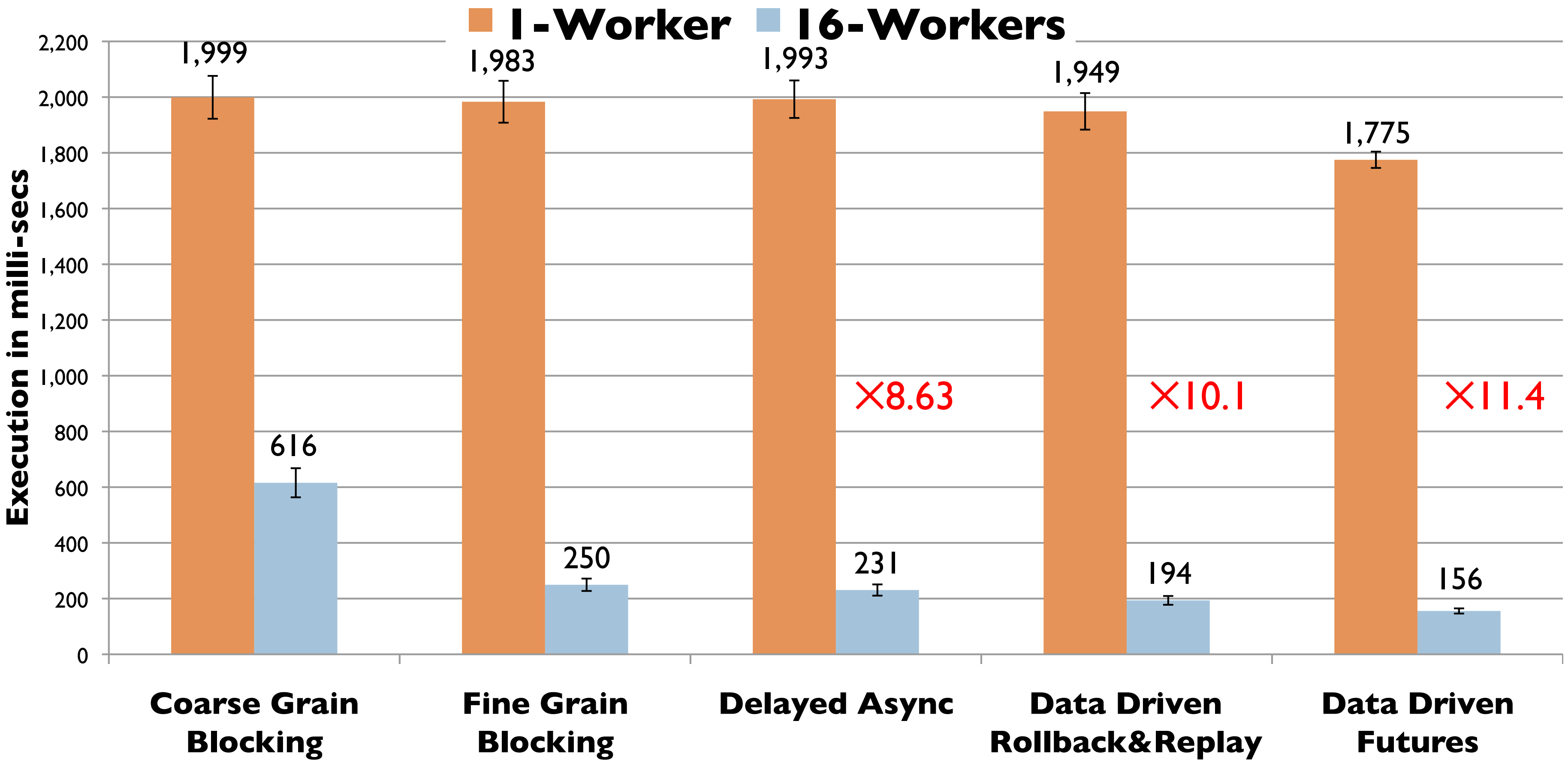
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky Decomposition with MKL on Xeon



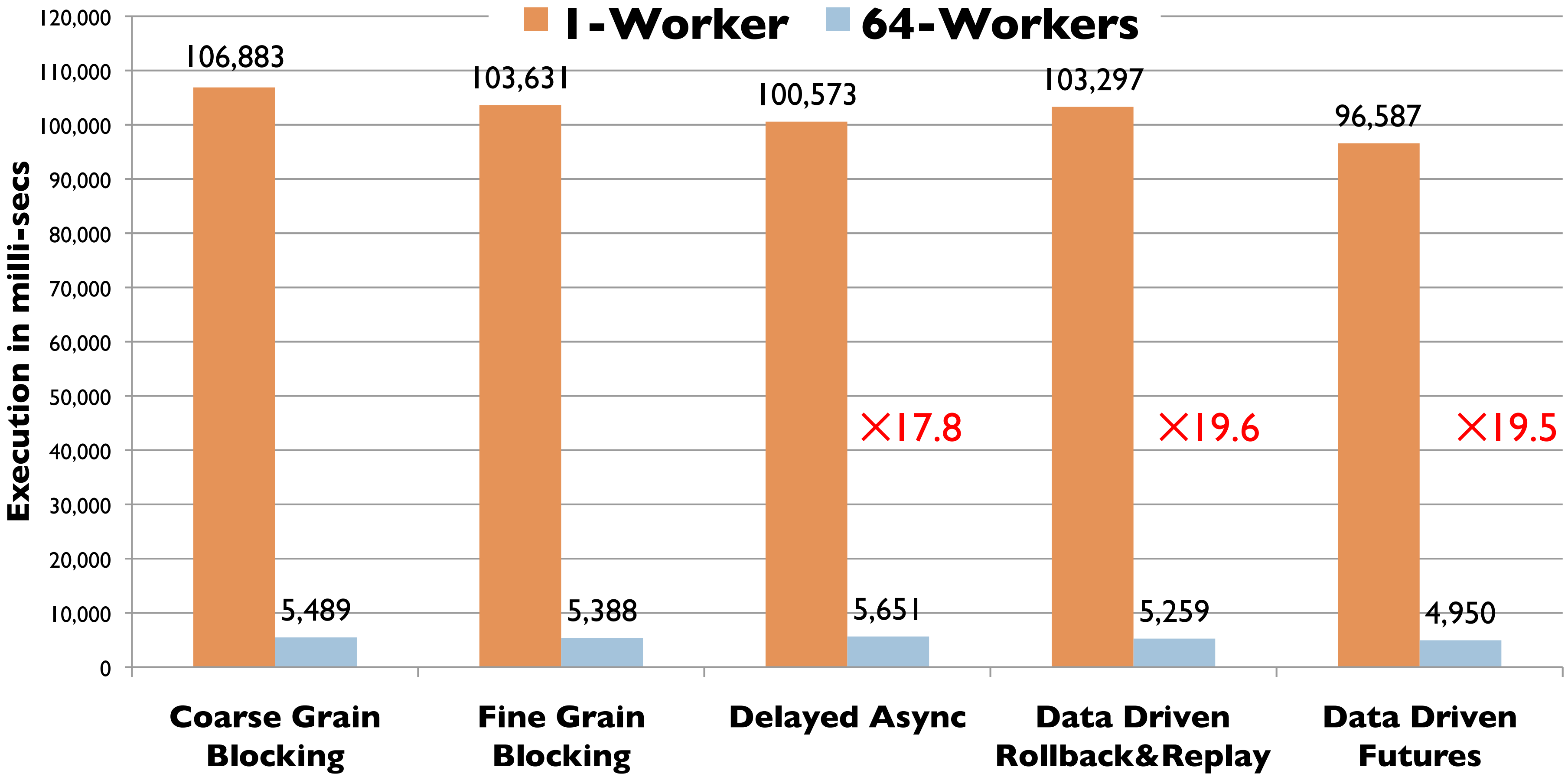
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java and Intel MKL steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky Decomposition with MKL on Xeon



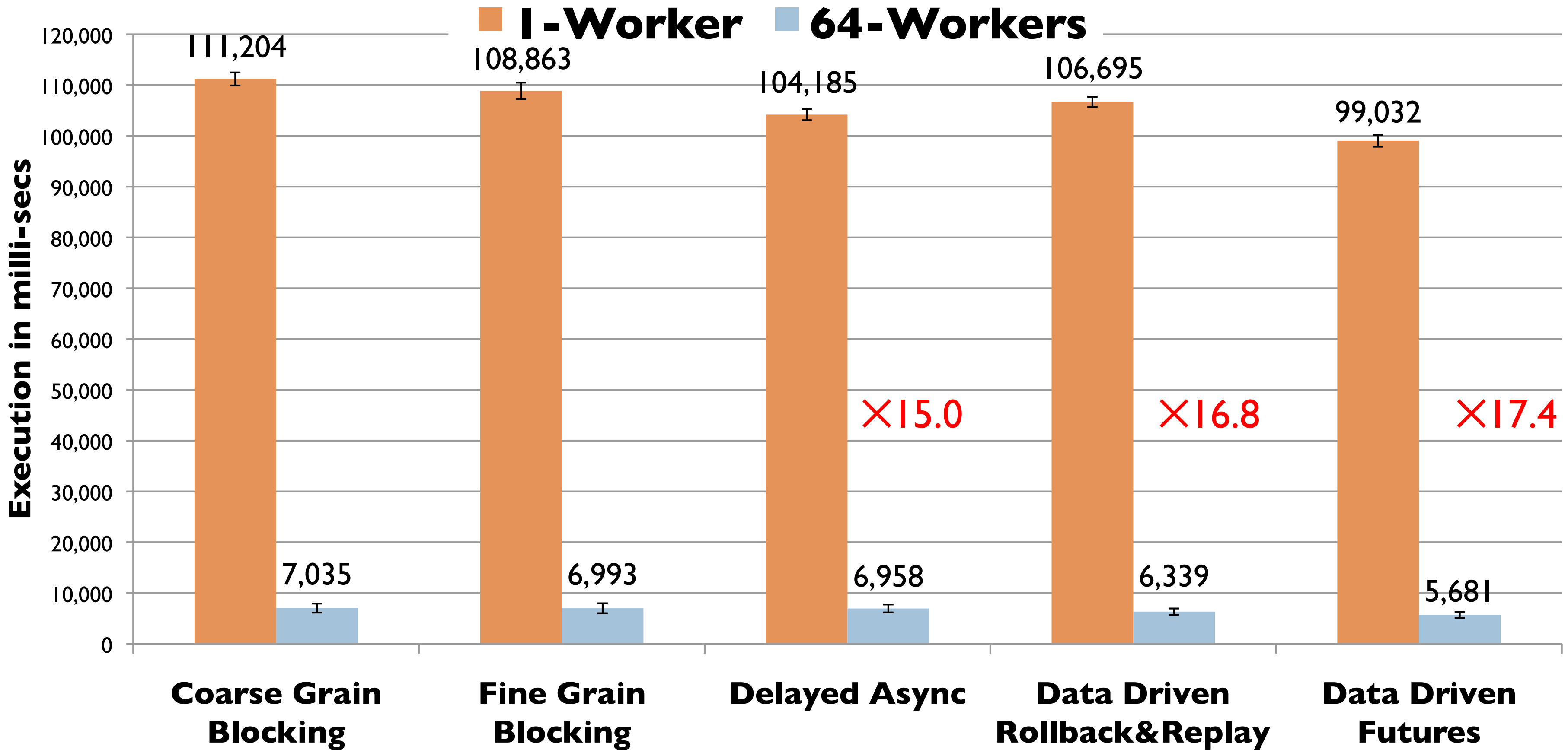
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero Java and Intel MKL steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky Decomposition on UltraSPARC T2



Minimum execution times of 30 runs of single threaded and 64-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on UltraSPARC T2 with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky Decomposition on UltraSPARC T2

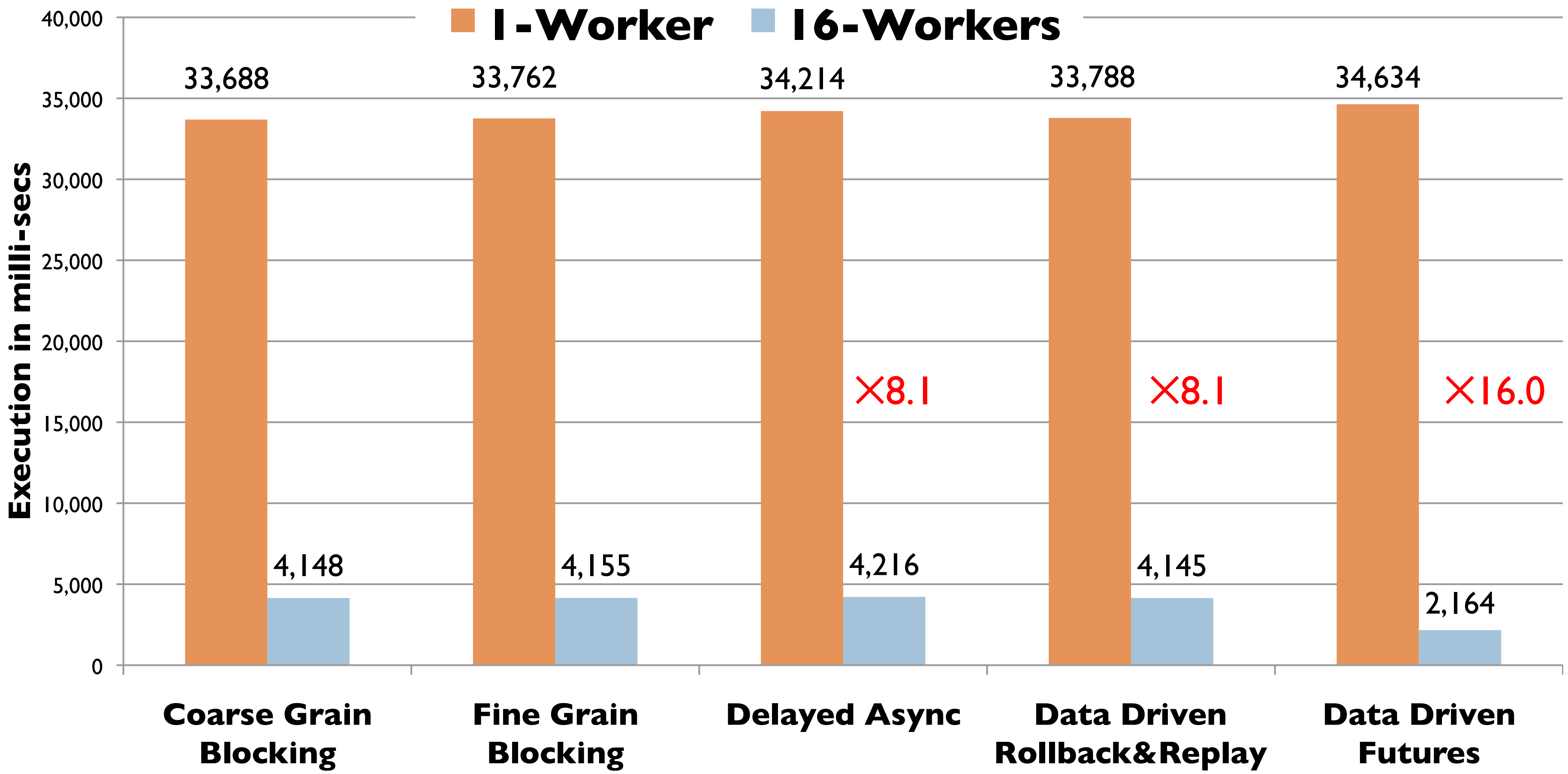


Average execution times and 90% confidence interval of 30 runs of single threaded and 64-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on UltraSPARC T2 with input matrix size 2000 × 2000 and with tile size 125 × 125

Black-Scholes formula

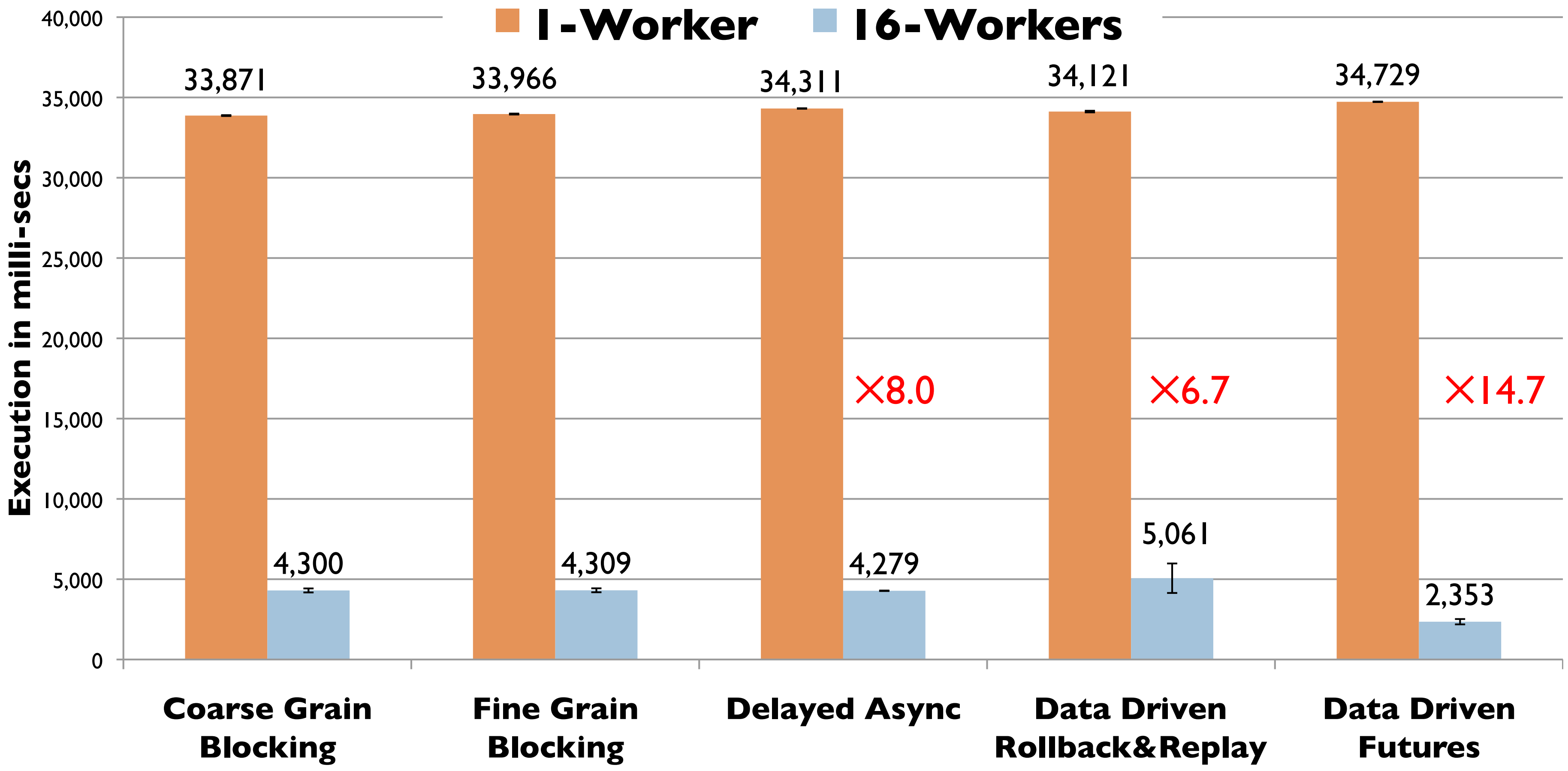
- Only one step
  - The Black-Scholes formula
- Embarrassingly parallel
- Good indicator of scheduling overhead

Black-Scholes on Xeon



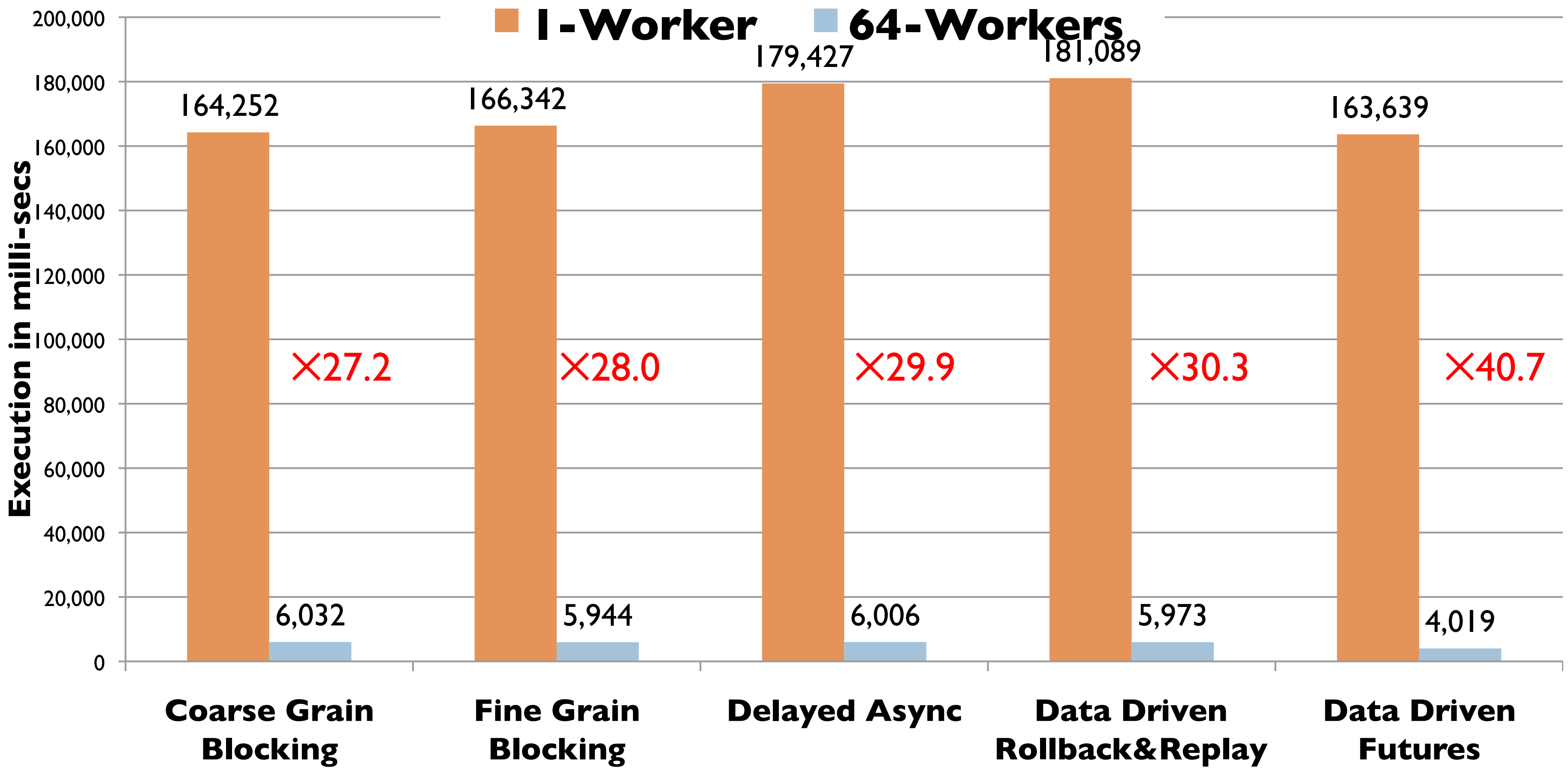
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on Xeon with input size 1,000,000 and with tile size 62,500

Black-Scholes on Xeon



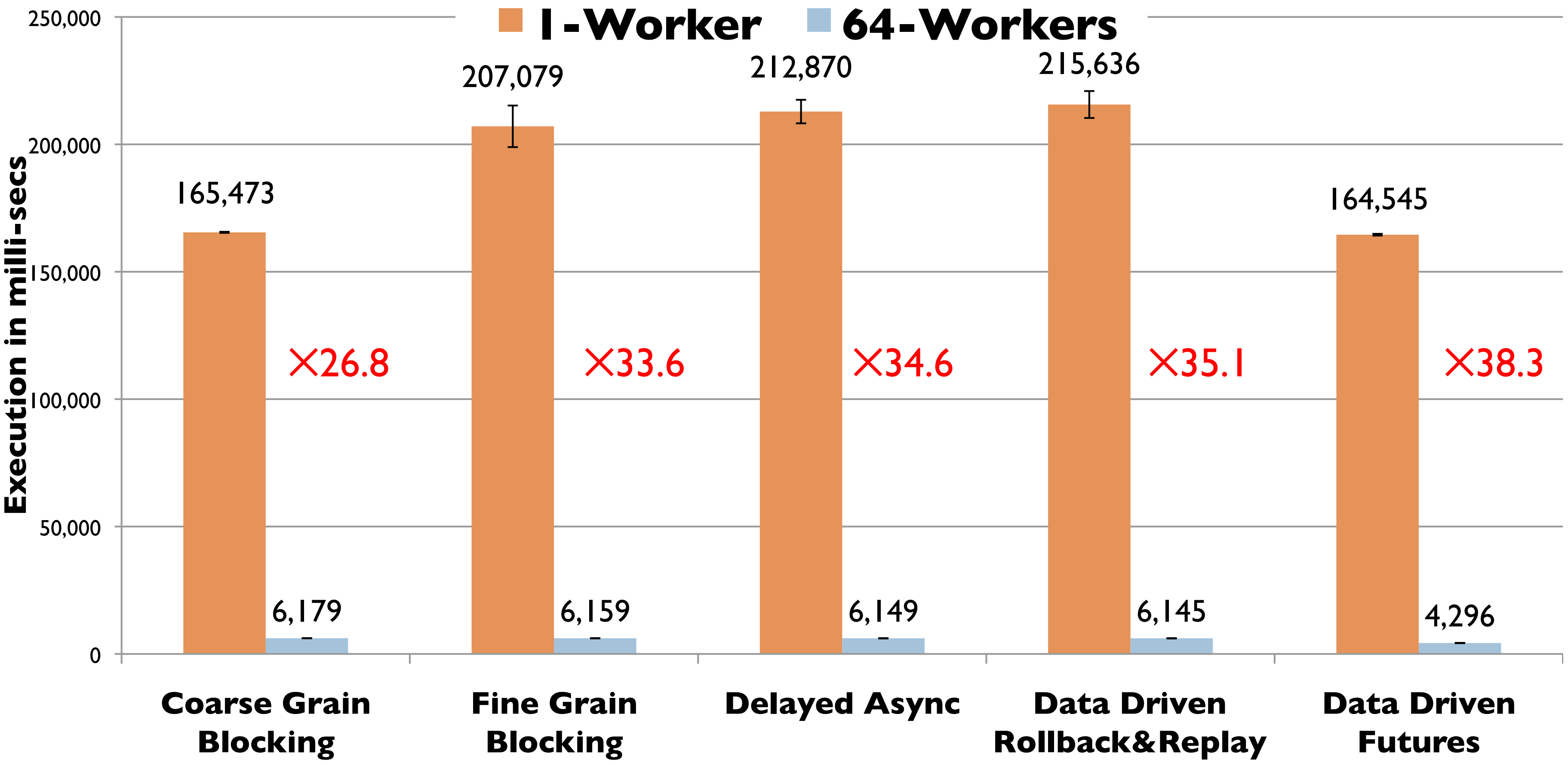
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on Xeon with input size 1,000,000 and with tile size 62,500

Black-Scholes on UltraSPARC T2



Minimum execution times of 30 runs of single threaded and 64-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on UltraSPARC T2 with input size 1,000,000 and with tile size 15,625

Black-Scholes on UltraSPARC T2

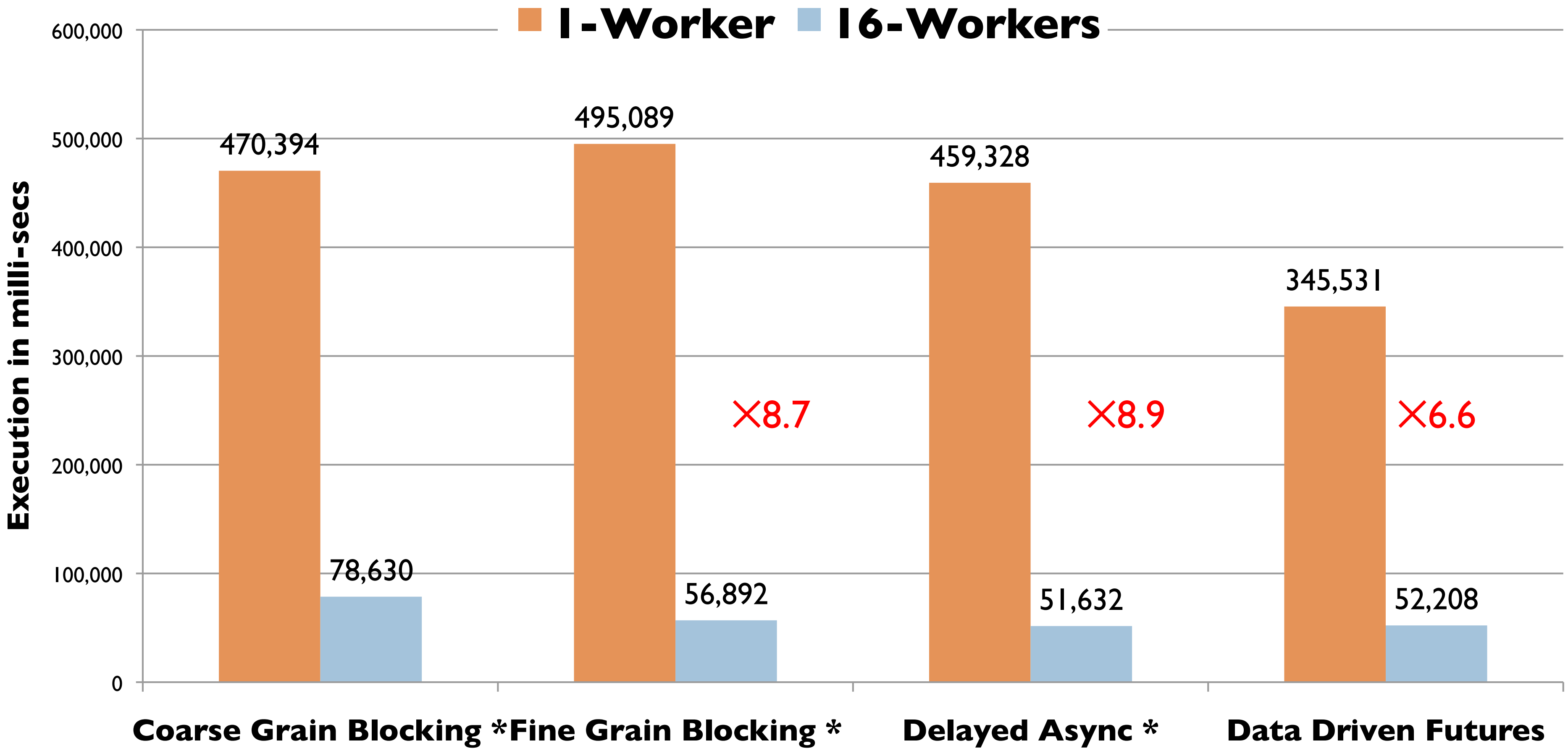


Average execution times and 90% confidence interval of 30 runs of single threaded and 64-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on UltraSPARC T2 with input size 1,000,000 and with tile size 15,625

Rician Denoising

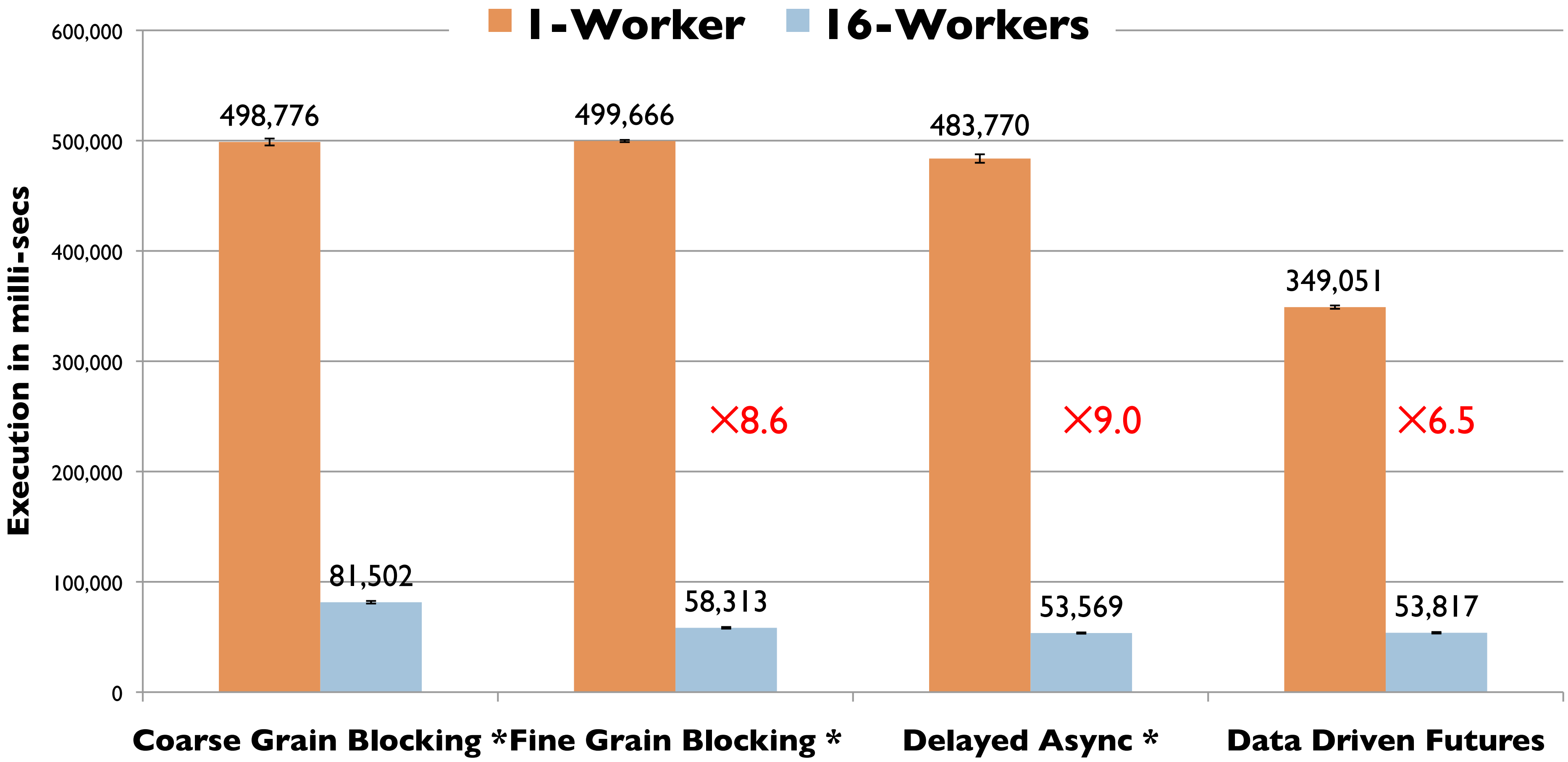
- Image processing algorithm
  - More than 4 kernels
    - Mostly stencil computations
  - Non trivial dependency graph
  - Fixed point algorithm
- Enormous data size
  - CnC schedulers needed explicit memory management
  - DDFs took advantage of garbage collection

Rician Denoising on Xeon



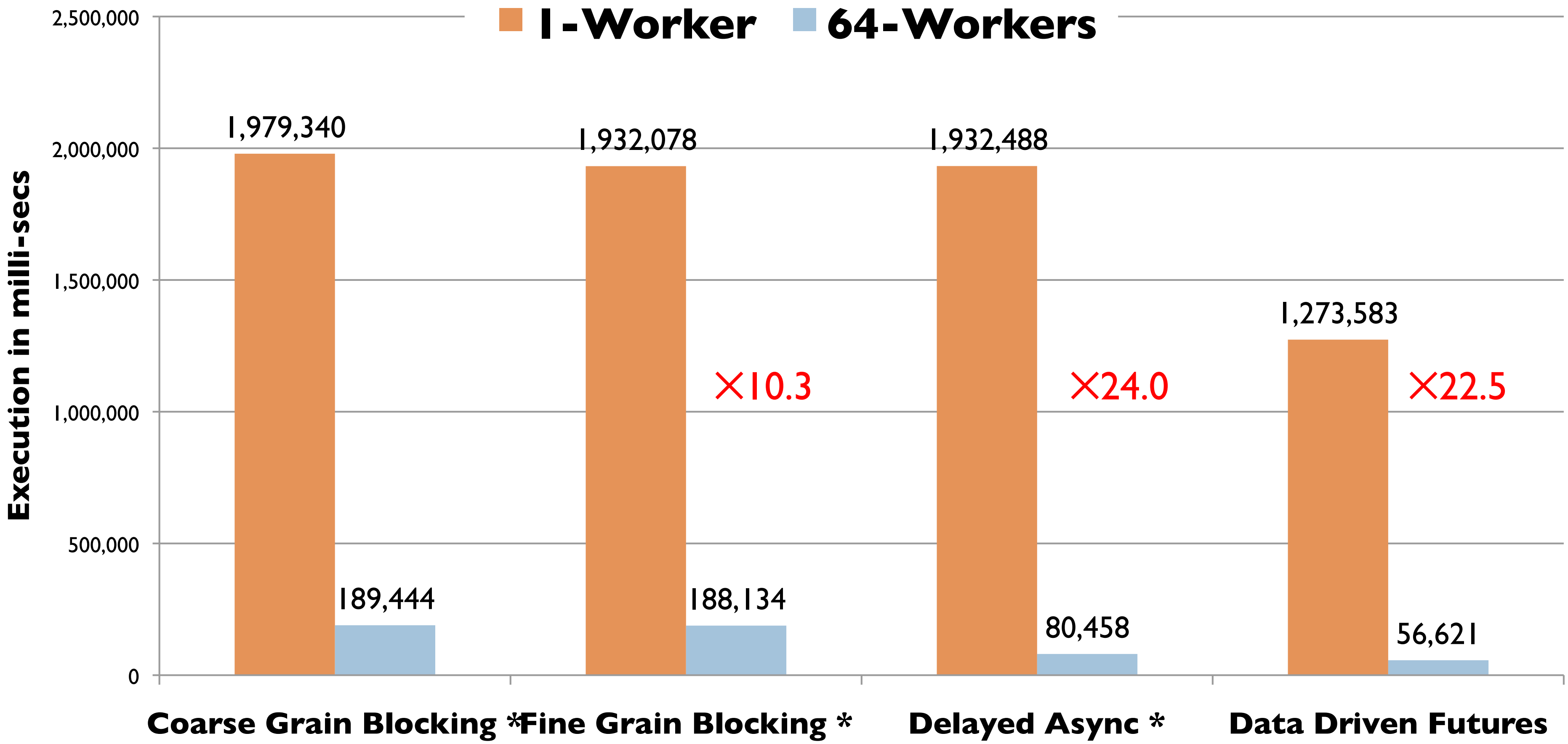
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on Xeon with input image size 2937 × 3872 and with tile size 267 × 484

Rician Denoising on Xeon



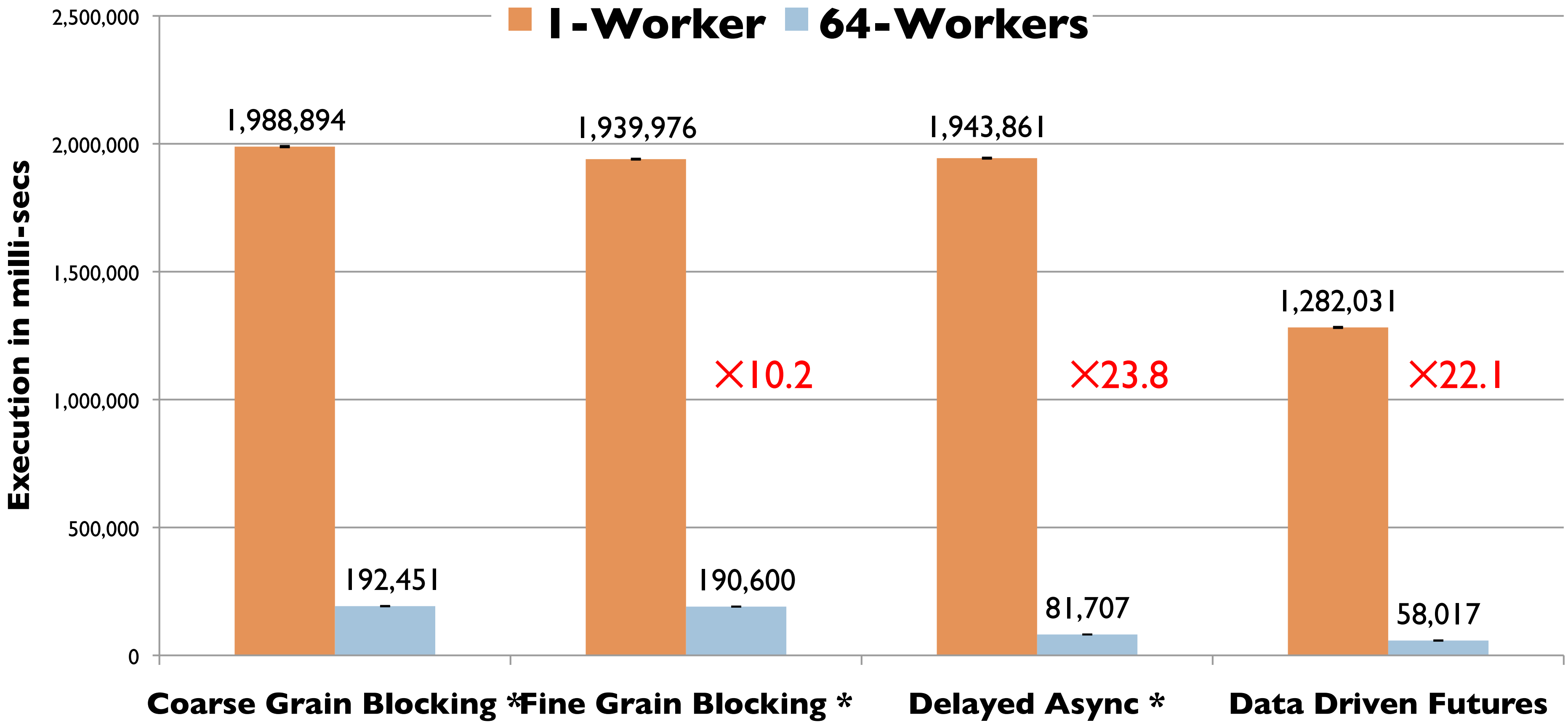
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on Xeon with input image size 2937 × 3872 and with tile size 267 × 484

Rician Denoising on UltraSPARC T2



Minimum execution times of 30 runs of single threaded and 64-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on UltraSPARC T2 with input image size 2937 × 3872 and with tile size 267 × 484

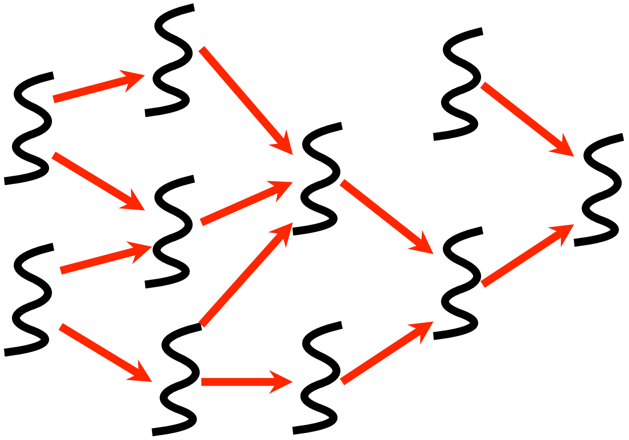
Rician Denoising on UltraSPARC T2



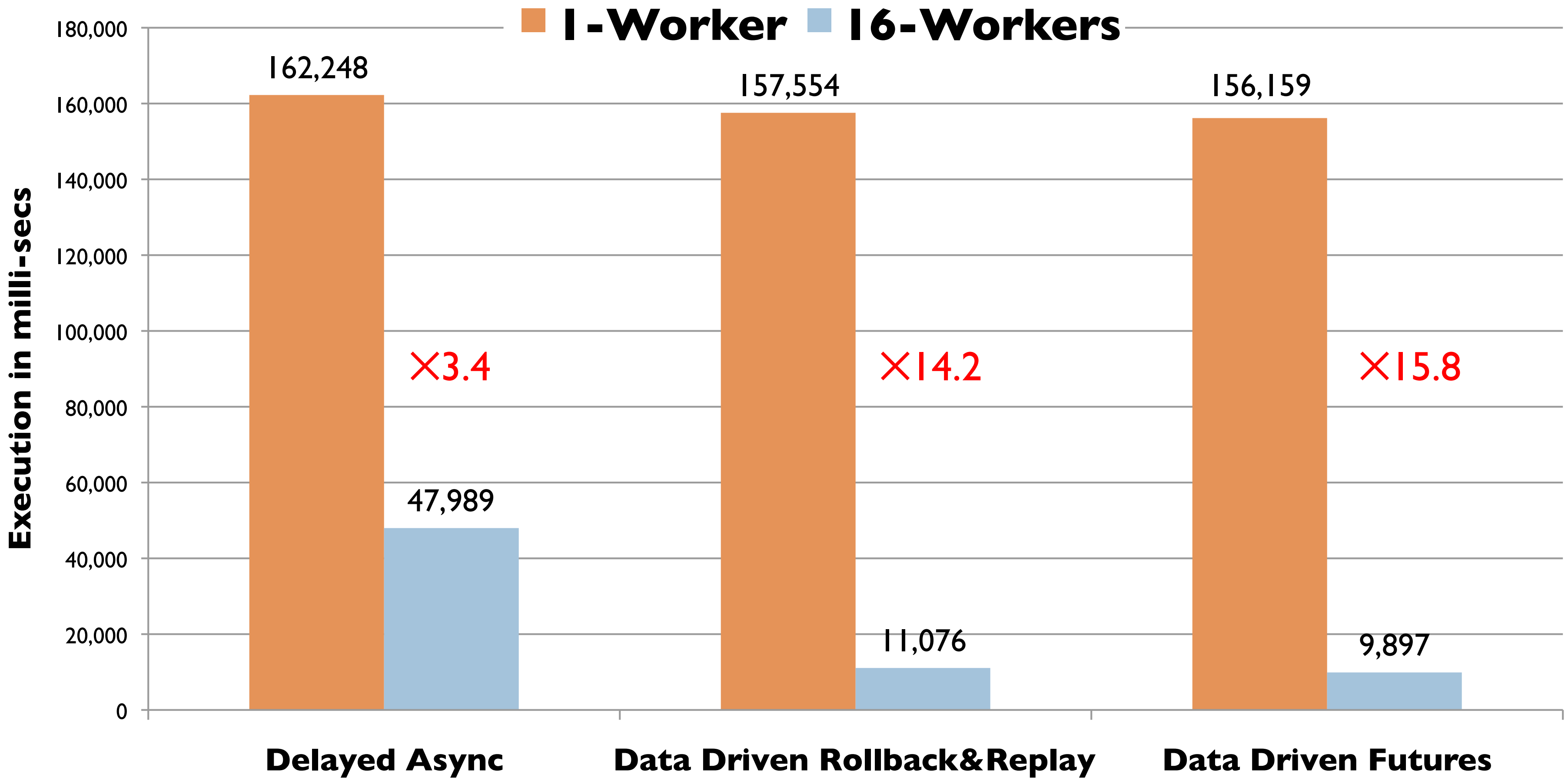
Average execution times and 90% confidence interval of 30 runs of single threaded and 64-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on UltraSPARC T2 with input image size 2937 × 3872 and with tile size 267 × 484

Heart Wall Tracking

- Medical imaging application
  - Nested kernels
    - First level embarrassingly parallel
    - Second level with intricate dependency graph
- Memory management
  - Many failures on eager schedulers
    - Blocking schedulers ran out of memory

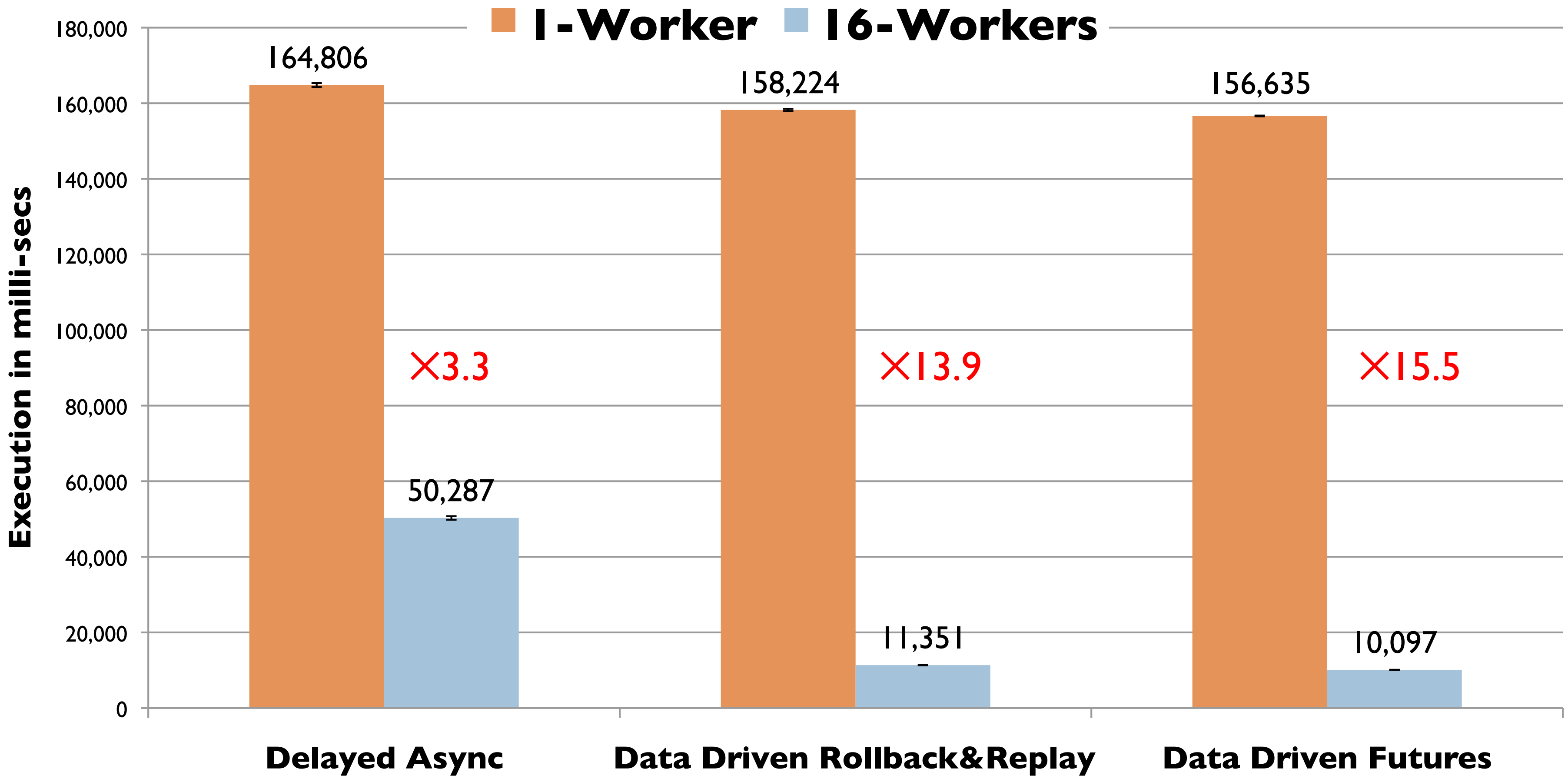


Heart Wall Tracking on Xeon



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

Heart Wall Tracking on Xeon



Average execution times and 90% confidence interval of 13 runs of single threaded and 16-threaded executions for Heart Wall Tracking CnC application with C steps on Xeon with 104 frames

Outline

- Background
- CnC Scheduling
- Data Driven Futures
- Results
- Wrap up

Related work

- Alternative parallel programming models:
  - Either too verbose or constrained parallelism
- Alternative futures, promises
  - Creation and resolution are coupled
  - Either lazy or blocking execution semantics
- Support for unstructured parallelism
  - Nabbit library for Cilk++ allows arbitrary task graphs
    - Immediate successor atomic counter update for notification
    - Does not differentiate between data, control dependences

Conclusions

- Macro-dataflow is a viable parallelism model
  - Provides expressiveness hiding parallelism concerns

- Macro-dataflow can perform competitively
  - Taking advantage of modern task parallel models

Future Work

- Compiling CnC to the Data Driven Runtime
  - Currently hand-ported
  - Need finer grain dependency analysis via tag functions
- Data Driven Future support for Work Stealing
- Compiler support for automatic DDF registration
- Hierarchical DDFs
- Locality aware scheduling support for DDFs

Acknowledgments

- Committee
  - Zoran Budimlić, Keith D. Cooper, Vivek Sarkar, Lin Zhong
- Journal of Supercomputing co-authors
  - Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff P. Lowney, Ryan R. Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach
- Habanero multicore software research project team-members
  - Zoran Budimlić, Vincent Cavé, Philippe Charles, Vivek Sarkar, Alina Simion Sbîrlea, Dragoș Sbîrlea, Jisheng Zhao
- Intel Technology Pathfinding and Innovation Software and Services Group
  - Mark Hampton, Kathleen Knobe, Geoff P. Lowney, Ryan R. Newton, Frank Schlimbach
- Benchmarks
  - Aparna Chandramowlishwaran (Georgia Tech.), Zoran Budimlić(Rice) for Cholesky Decomposition
  - Yu-Ting Chen (UCLA) for Rician Denoising
  - David Peixotto (Rice) for Black-Scholes Formula
  - Alina Simion Sbîrlea (Rice) for Heart Wall Tracking

Feedback and clarifications

- Thanks for your attention