Data-Driven Tasks as an Execution Model for Concurrent Collections

Sağnak Taşırlar

Motivation for Data-Driven Tasks

- CnC provides:
  - macro-dataflow abstractions
  - implicit parallelism across kernel (step) instantiations
  - item collections to capture data dependences between step instances
- Task-based runtimes need extension to support CnC
  - Blocking (Coarse-Fine)
  - Delayed `async`
  - Data-Driven Rollback & Replay

Motivation for Data-Driven Tasks

- Extend task-parallel models with Data-Driven Tasks (DDTs) !
- Data-Driven Tasks:
  - specifies its input constraints in an `await` clause containing a list of Data-Driven Futures (DDFs) produced by other tasks
  - creation of DDTs and production of DDFs are unrelated events
  - DDFs can be garbage-collected like other data structures

  - Direct support for CnC semantics ("assembly language" for CnC)
  - Brings benefits of CnC semantics to task-parallel programmers

Mapping CnC to Task-Parallelism

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

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

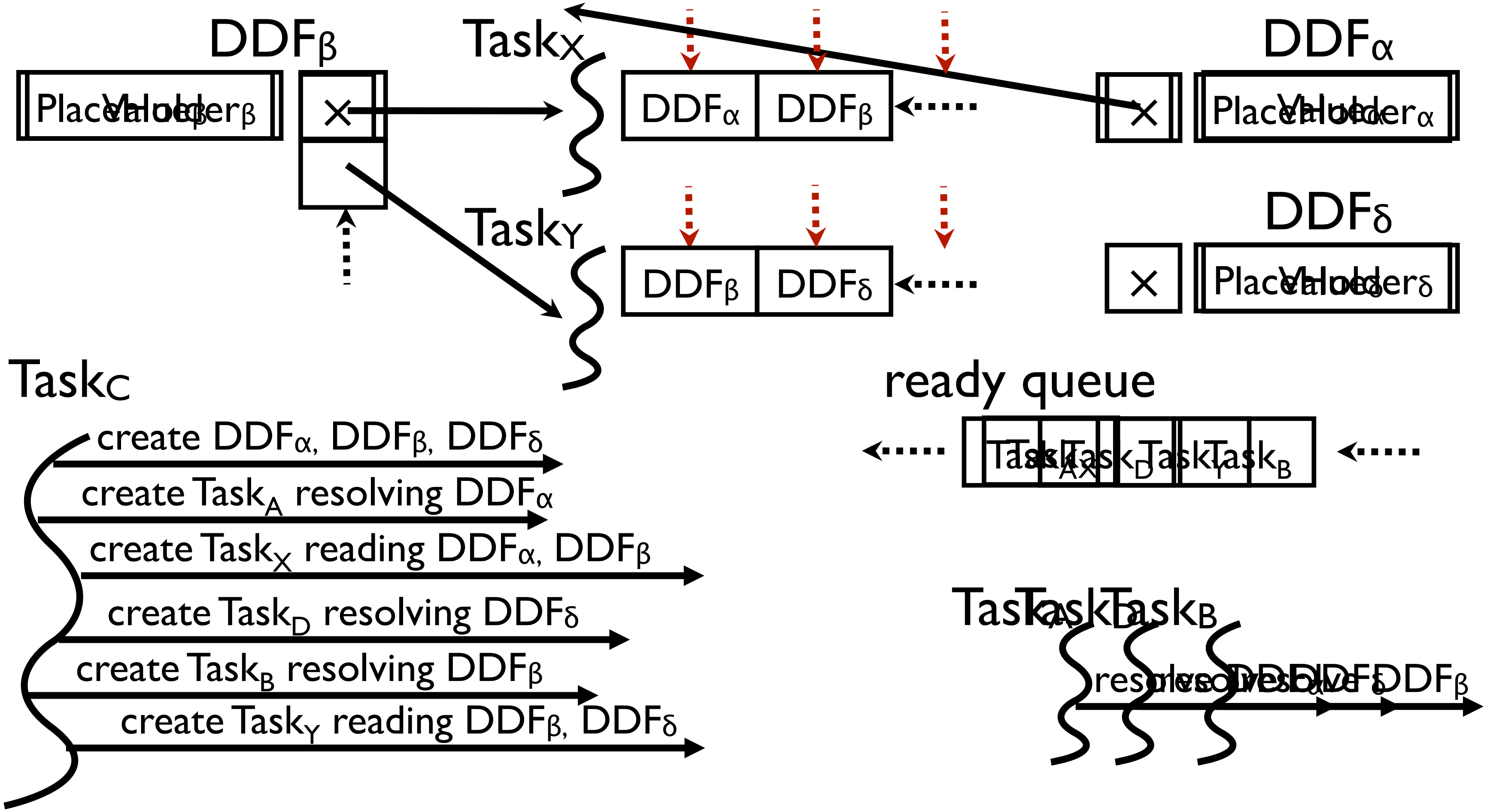
- Task parallel synchronization construct
  - Acts as a reference to single assignment value
- Creation
  - Create a dummy reference object
- Resolution (`put`)
  - Resolve what value a DDF is referring to
- Data-Driven Tasks (DDTs) (`async await`)
  - A task provides a consume list of DDFs on declaration
  - A task can only read DDFs that it is registered to

DDF/DDT Code Sample


```
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 );
}
```

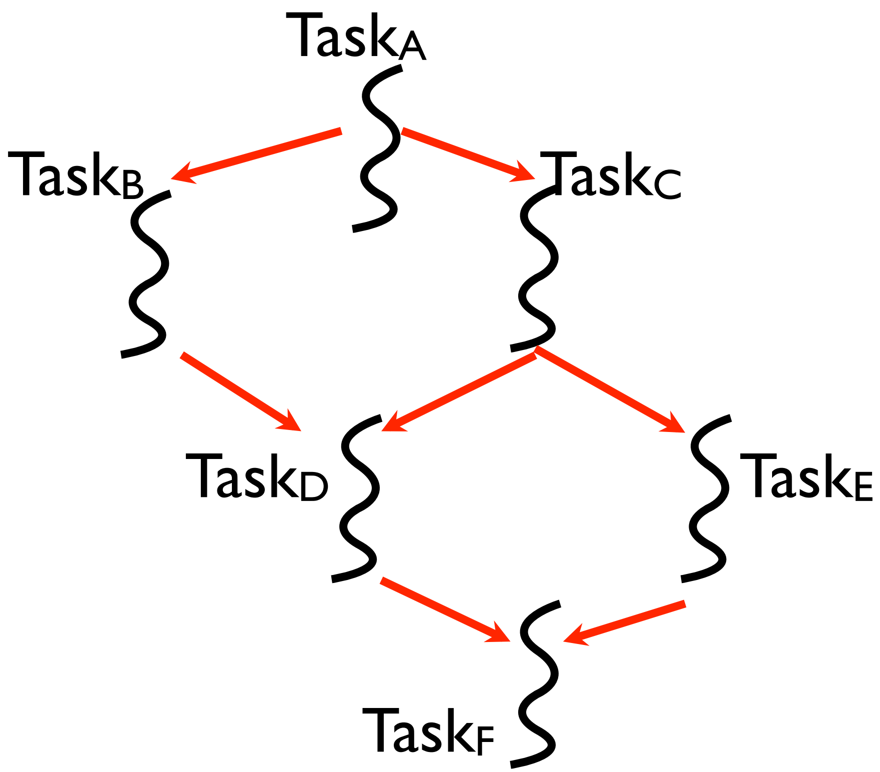
Data Driven Scheduling

- Steps register self to items wrapped into DDFs



Benefits of DDFs

- Non-series-parallel task dependency graphs support



- Single assignment value lifetime restriction
  - Not global lifetime
  - Creator:
    - feeds consumers
    - gives access to producer
  - Lifetime restricted to
    - Creator lifetime
    - Resolver lifetime
    - Consumers lifetimes

Compiling CnC to DDF

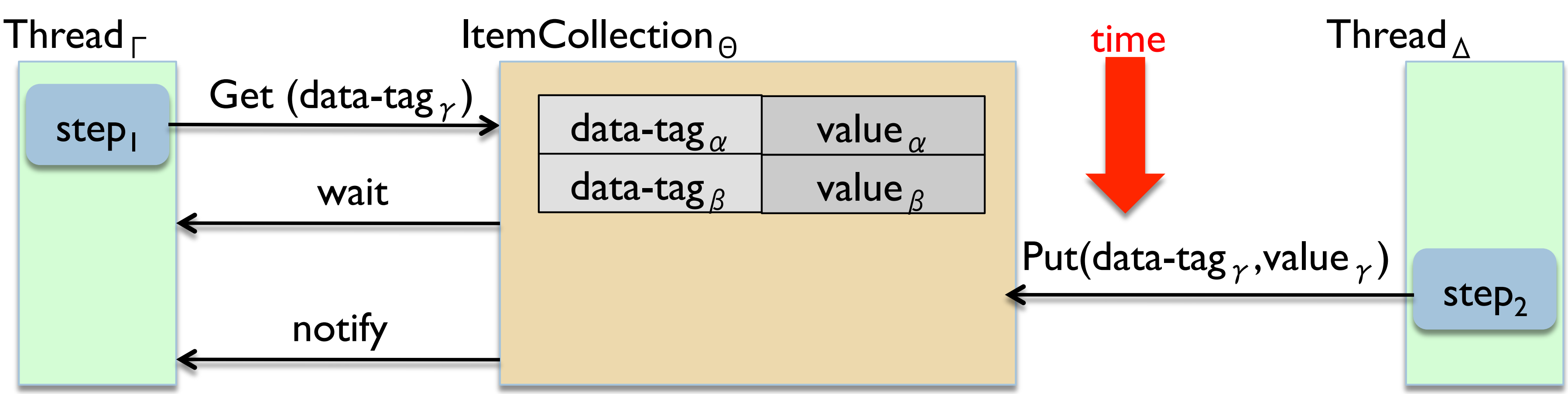
- Given which item instances a step instance reads
  - Currently the user provides a function that returns a list
  - May generate that list automatically by tag functions
- Every step instance can be described as a DDT
  - Habanero-Java supports DDFs and DDTs
- Item Collections are collections of DDFs
  - Tabular nature obsoletes the memory benefits for now

Preliminary Experimental Results

- DDT/DDF results obtained at task-parallel level
  - using individual DDFs
  - without allocating item collections or CnC
- Compared DDTs with four other CnC schedulers
  - Fine/Coarse Grain Blocking
  - Delayed `async`
  - Data-Driven Rollback & Replay

Blocking CnC Schedulers

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



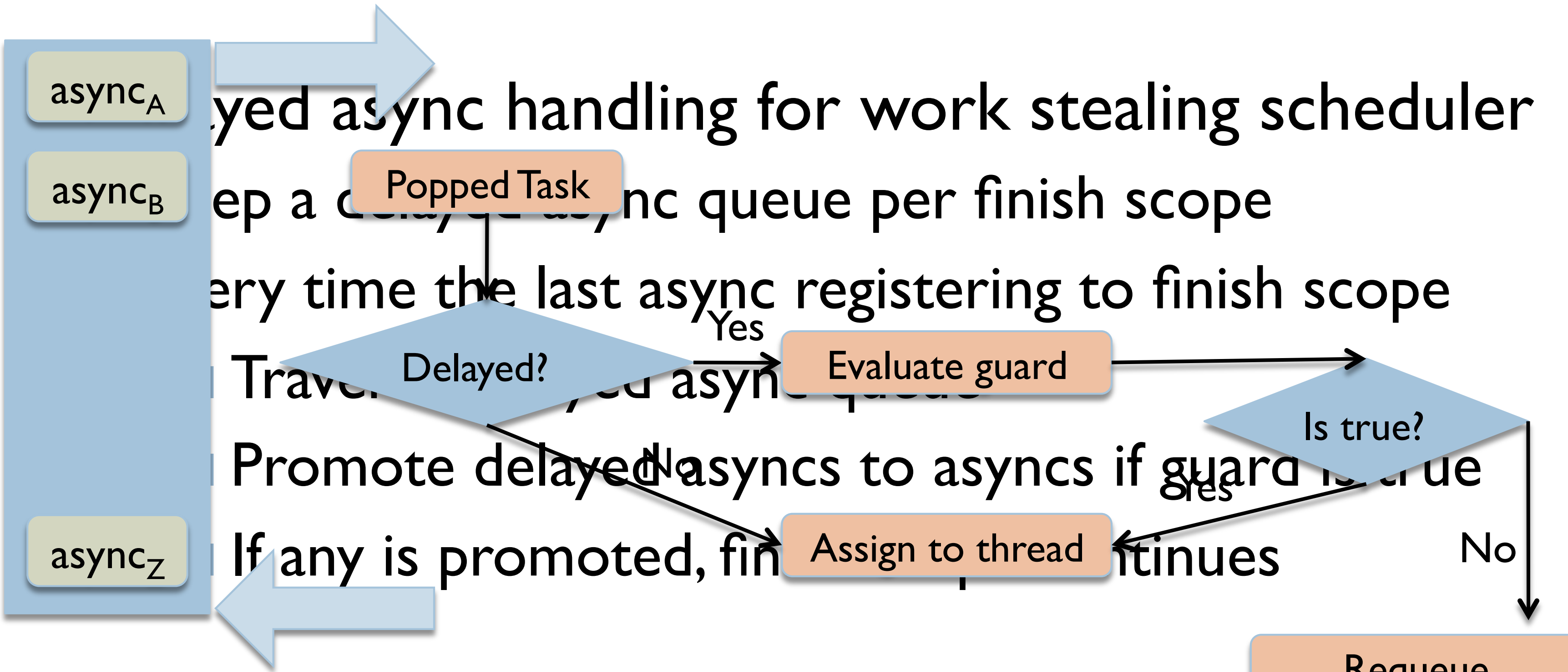
Delayed Async Scheduling

- Every CnC step is a guarded execution
  - Guard condition is the availability 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;
    }
}
```

Delayed Asyncs

- Guarded execution construct for HJ
  - Promote to async when guard evaluates to true
- ...yed async handling for work stealing scheduler
  - ...ep a d... ...nc queue per finish scope
  - ...ery time the last async registering to finish scope
    - Trav... ...ed async queue
    - Promote delayed asyncs to asyncs if guard ... true
    - If any is promoted, fin... ...tinues

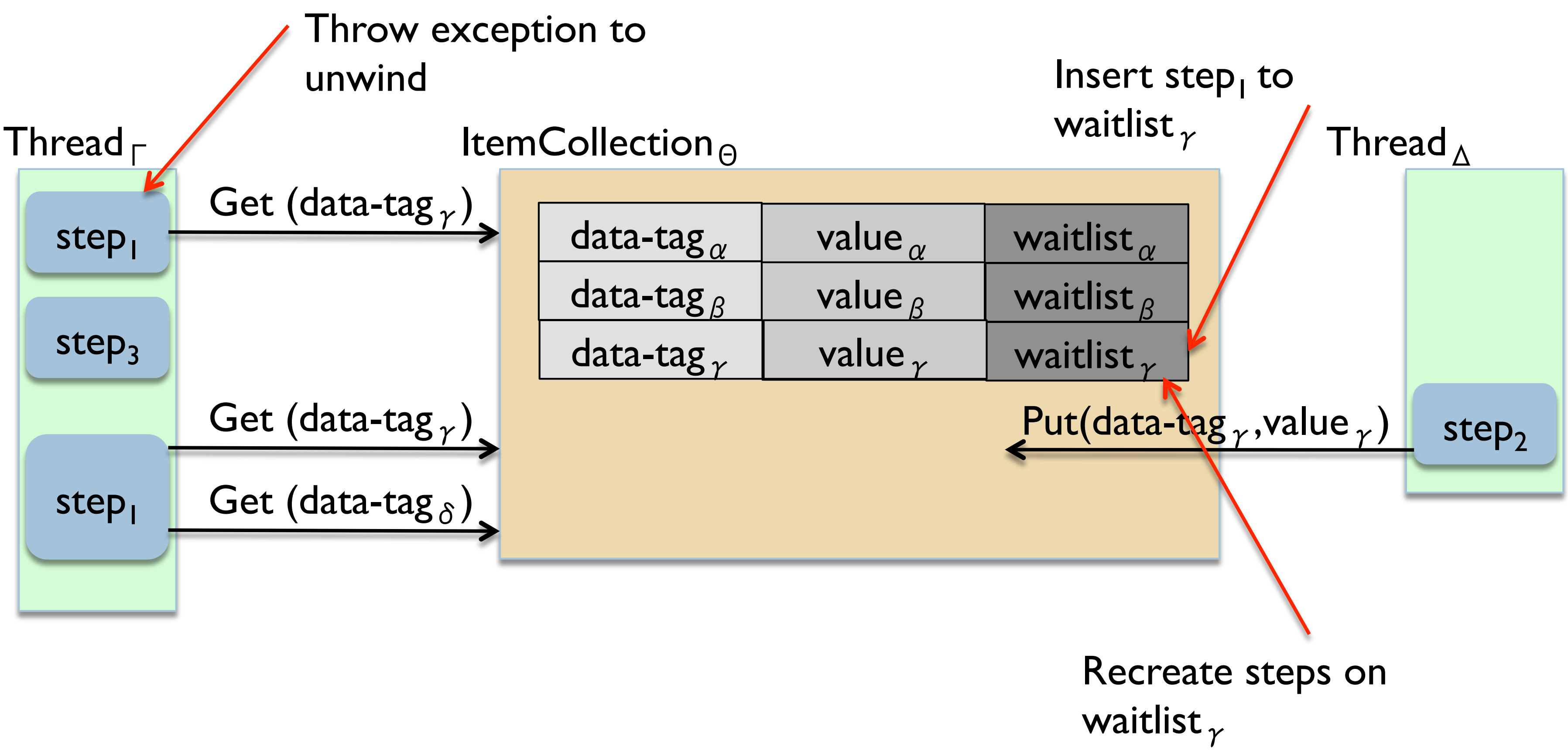


Work Sharing Ready Task Queue

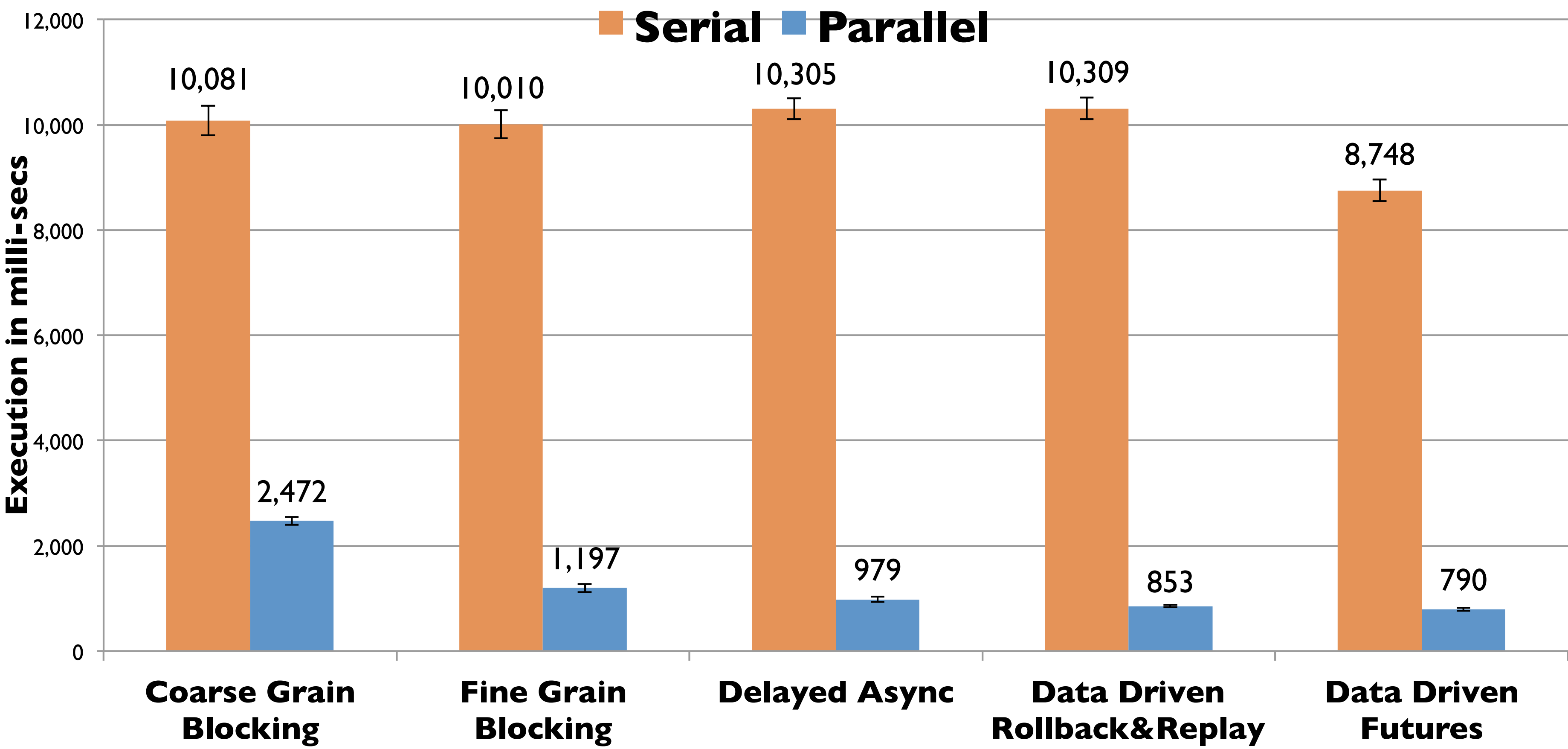
Data-Driven Rollback & Replay

- 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 and continue with another ready task

Data Driven Rollback & Replay

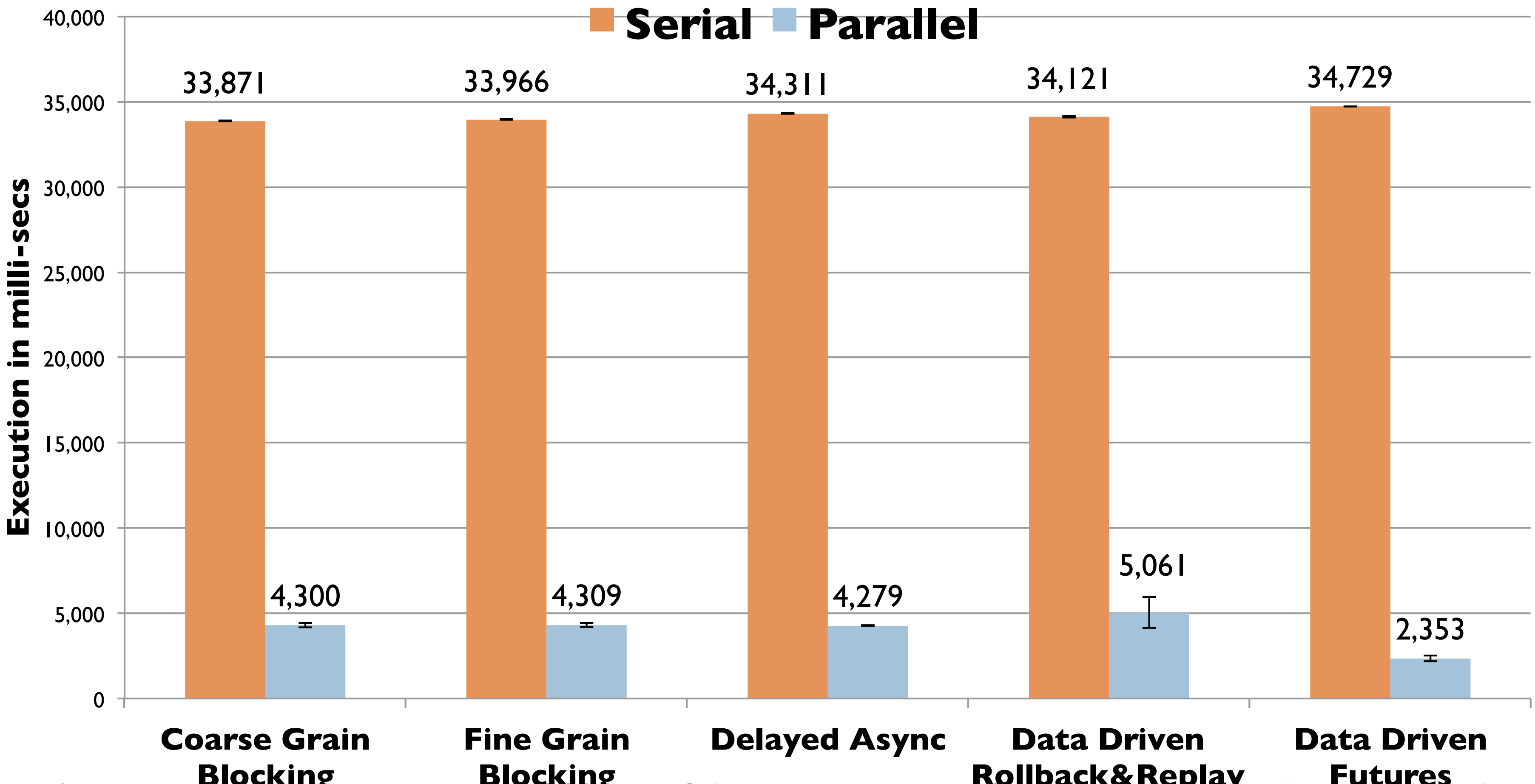


Cholesky decomposition



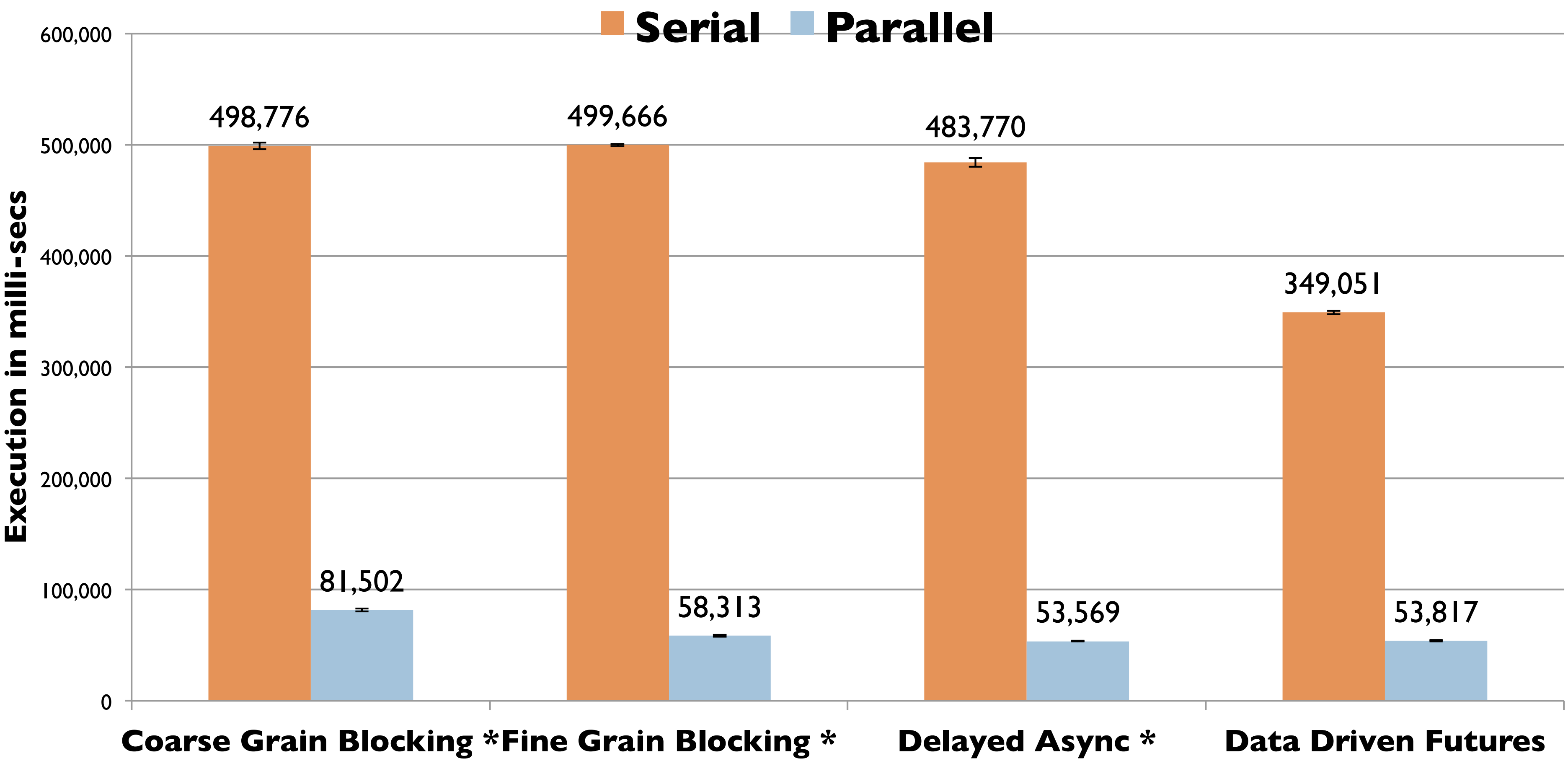
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Black-Scholes formula



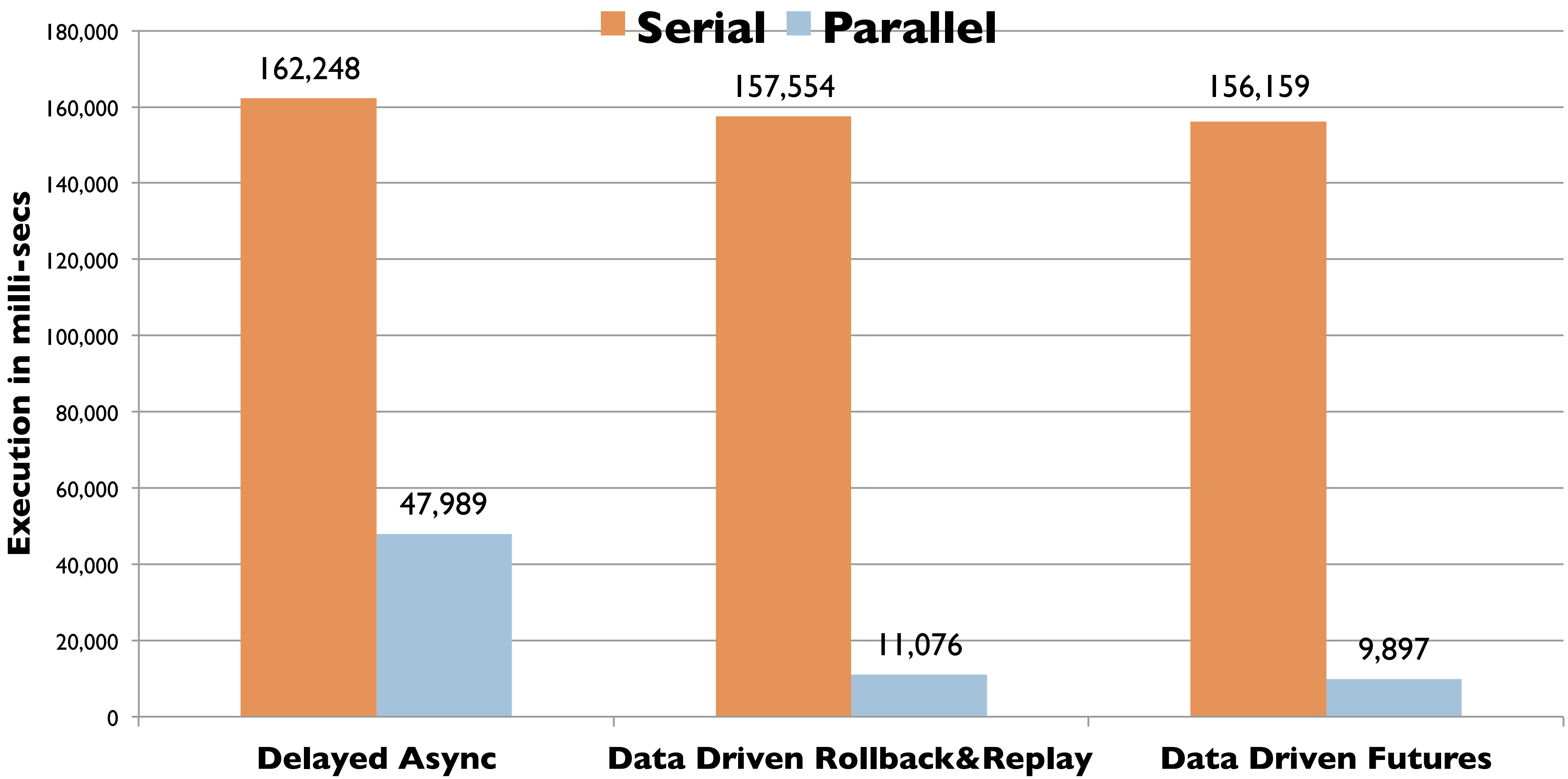
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on Xeon with input size 1,000,000 and with tile size 62,500

Rician Denoising



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

Heart Wall Tracking

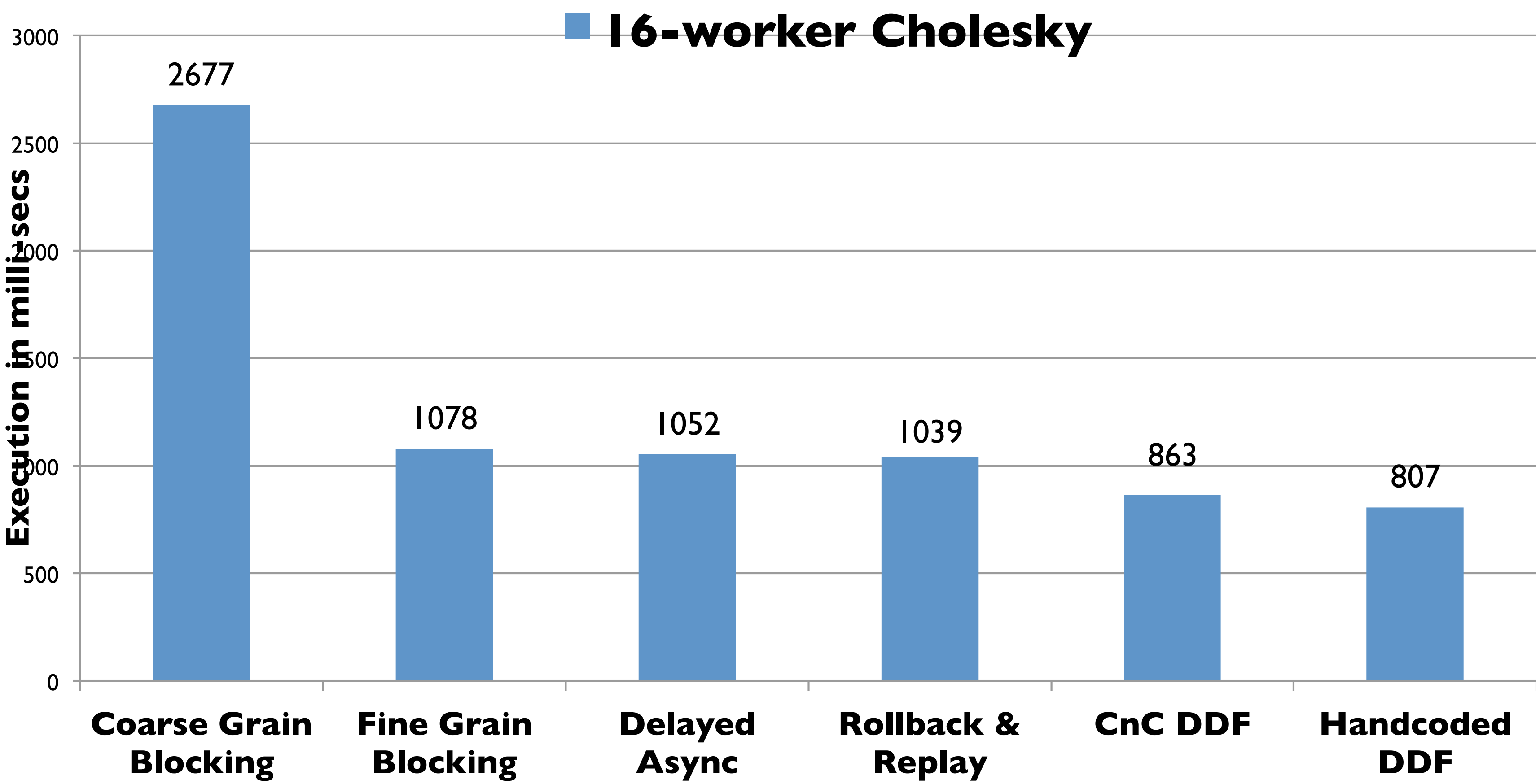


Minimum execution times of 13 runs of single threaded and 16-threaded executions for Heart Wall Tracking CnC application with C steps on Xeon with 104 frames

Status of the HJ-CnC-DDF impl'n

- User has to implement a `getAwaitsList()`
  - returns a list of DDFs referring to the items to be read
- If `getAwaitsList()` is correctly implemented
  - No safety checks as of now
- CnC runtime generates and executes DDTs
- Item Collections implicitly (un)wraps items as DDFs

Cholesky decomposition



Average execution times of 30 runs of 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

Future Work

- Automatic `getAwaitsList()` creation
- Non-tabular (decentralized) Item Collections
- Push DDF creation to the innermost possible scope
- Environment as a DDT to avoid waiting whole graph

Feedback and clarifications

- Thanks for your attention

Backup slides

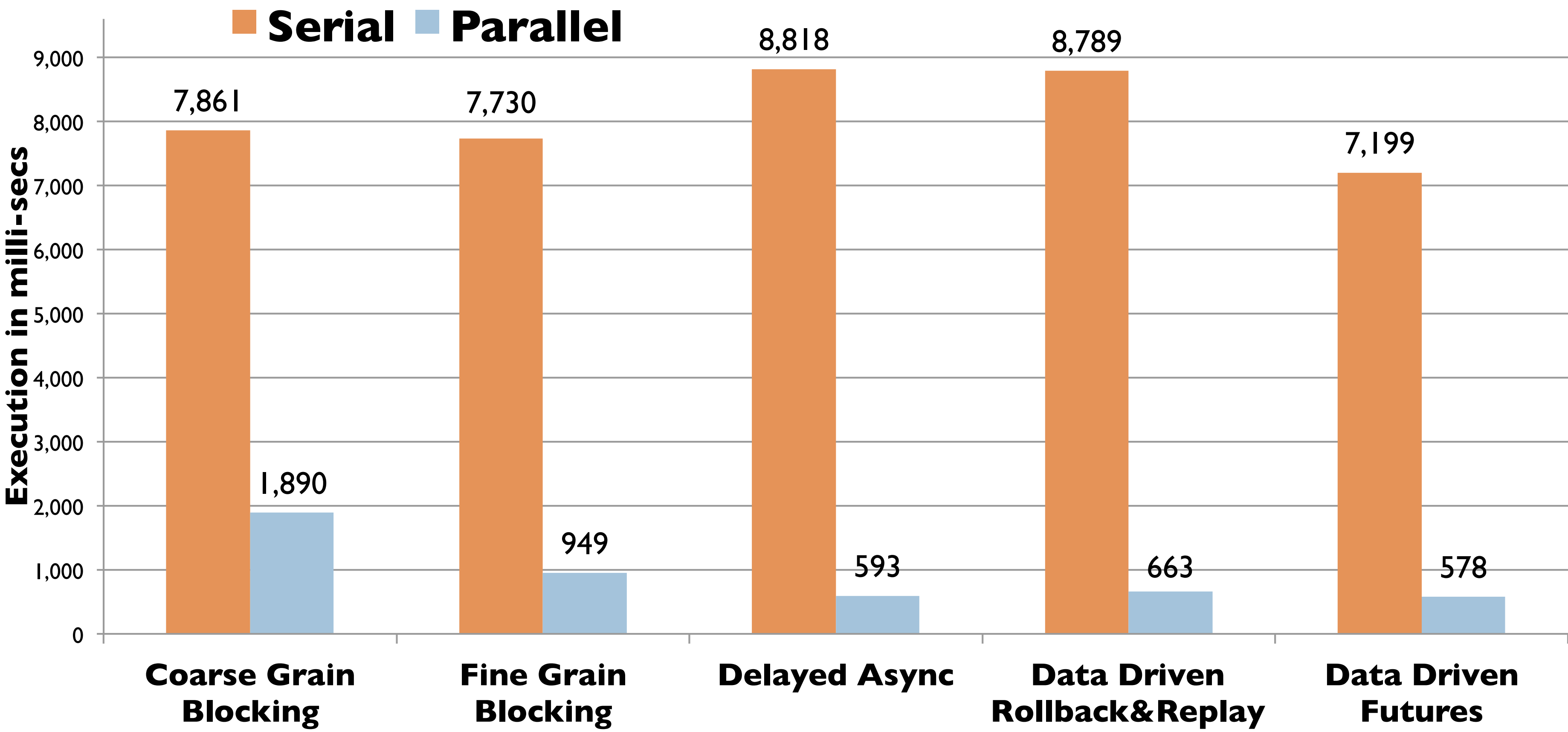
Hand-coded Cholesky DDF


```
        DataDrivenFuture [][][] outLkji = null;

        for ( int numIters = 0; numIters < 30; ++numIters ) {
            final DataDrivenFuture [][][] lkji = new DataDrivenFuture [numTiles][][];

            for( int i = 0 ; i < numTiles ; ++i ) {
                lkji[i] = new DataDrivenFuture [i+1][numTiles+1];
                for( int j = 0 ; j <= i ; ++j ) {
                    for( int k = 0 ; k <= numTiles ; ++k ) {
                        lkji[i][j][k] = new DataDrivenFuture();
                    }
                }
            }

            int A_i, A_j, T_i, T_j;
            for( int i = 0 ; i < numTiles ; ++i ) {
                for( int j = 0 ; j <= i ; ++j ) {
                    // Allocate memory for the tiles.
                    double [][] temp = new double[ tileSize ][ tileSize ];
                    // Split the matrix into tiles and write it into the item space at time 0.
                    // The tiles are indexed by tile indices (which are tag values).
                    for( A_i = i * b, T_i = 0 ; T_i < tileSize ; ++A_i, ++T_i ) {
                        for( A_j = j * b, T_j = 0 ; T_j < tileSize ; ++A_j, ++T_j ) {
                            temp[ T_i ][ T_j ] = A[ A_i ][ A_j ];
                        }
                    }
                    //lkji[i][j][0] = new DataDrivenFuture ((java.lang.Object)temp);
                    lkji[i][j][0].put((java.lang.Object)temp);
                }
            }

            long begin = java.lang.System.currentTimeMillis();
            finish {
                for ( int k = 0; k < numTiles; ++k ) {
                    final DataDrivenFuture pivot_kkk = lkji[k][k][k];
                    final DataDrivenFuture pivot_kkk1 = lkji[k][k][k+1] ;

                    async await (pivot_kkk ) {
                        s1_obj.compute([k], tileSize, pivot_kkk, pivot_kkk1);
                    }

                    for( int j = k + 1 ; j < numTiles ; ++j) {
                        async await ( lkji[j][k][k], pivot_kkk1) {
                            s2_obj.compute([k,j], tileSize , lkji[j][k][k], pivot_kkk1, lkji[j][k][k+1]);
                        }

                        for( int i = k + 1 ; i < j ; ++i) {
                            async await ( lkji[j][i][k], lkji[i][k][k+1], lkji[j][k][k+1]) {
                                s3_2_obj.compute( [k,j,i], tileSize , lkji[j][i][k], lkji[i][k][k+1], lkji[j][k][k+1], lkji[j][i][k+1]);
                            }
                        }
                        async await ( lkji[j][j][k], lkji[j][k][k+1]) {
                            s3_obj.compute( [k,j,j], tileSize , lkji[j][j][k], lkji[j][k][k+1], lkji[j][j][k+1]);
                        }
                    }
                }
            }
```

Cholesky decomposition

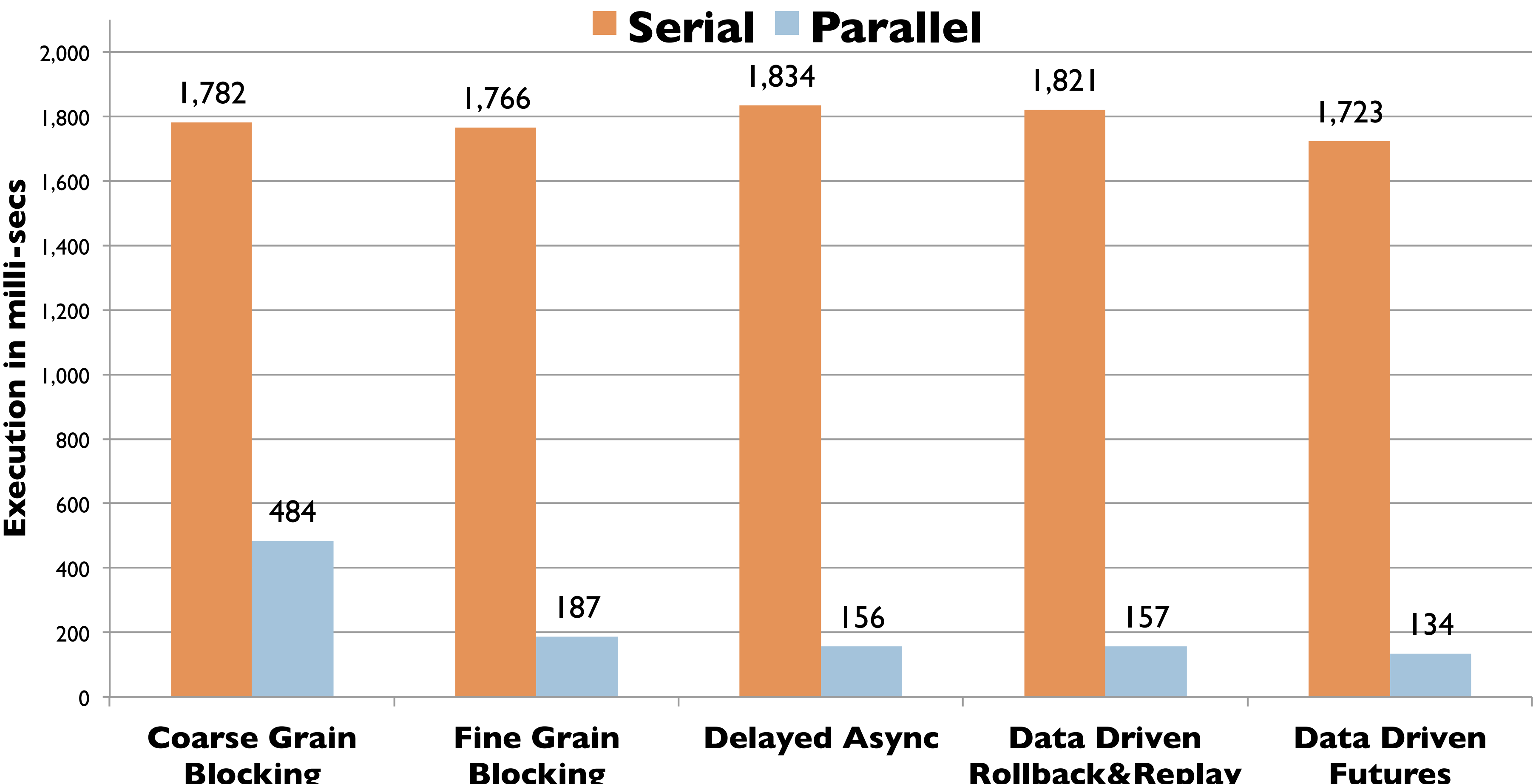
- Dense linear algebra kernel
- Three inherent kernels
  - Need to be pipelined for best performance
  - Loop parallelism within some kernels
  - Data parallelism within some kernels
- CnC was shown to beat optimized libraries

Cholesky decomposition



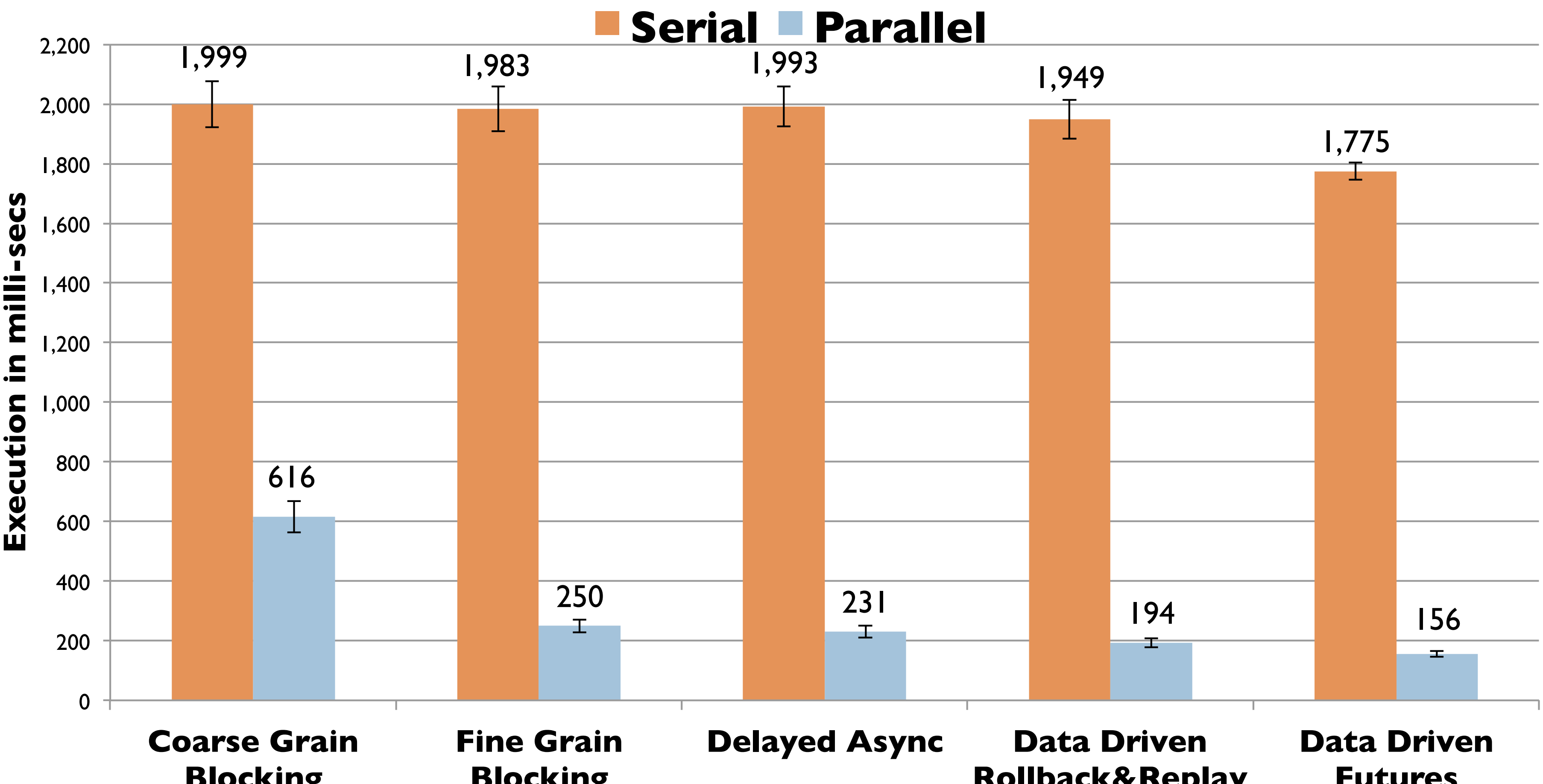
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky decomposition



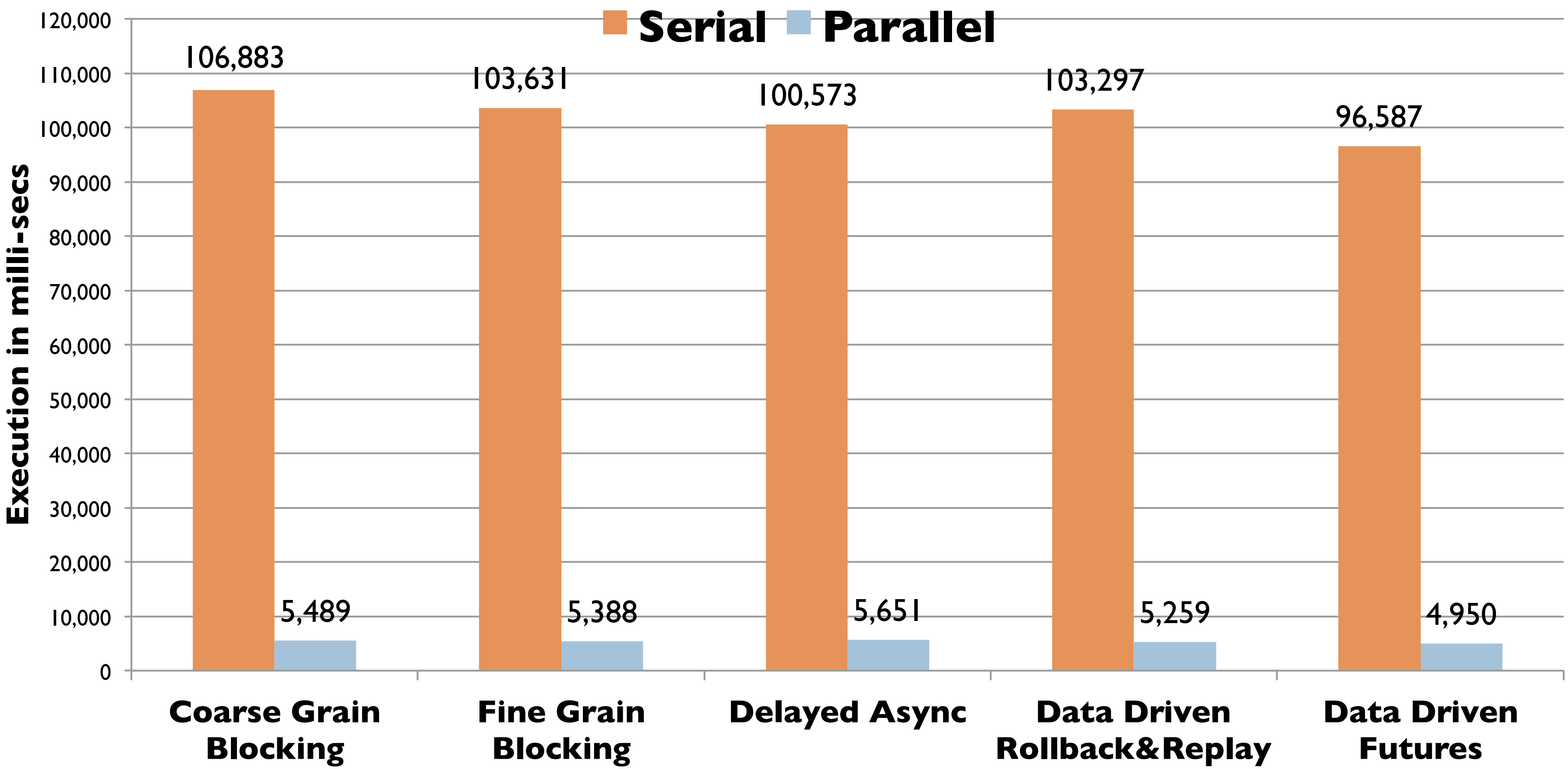
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java and Intel MKL steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky decomposition



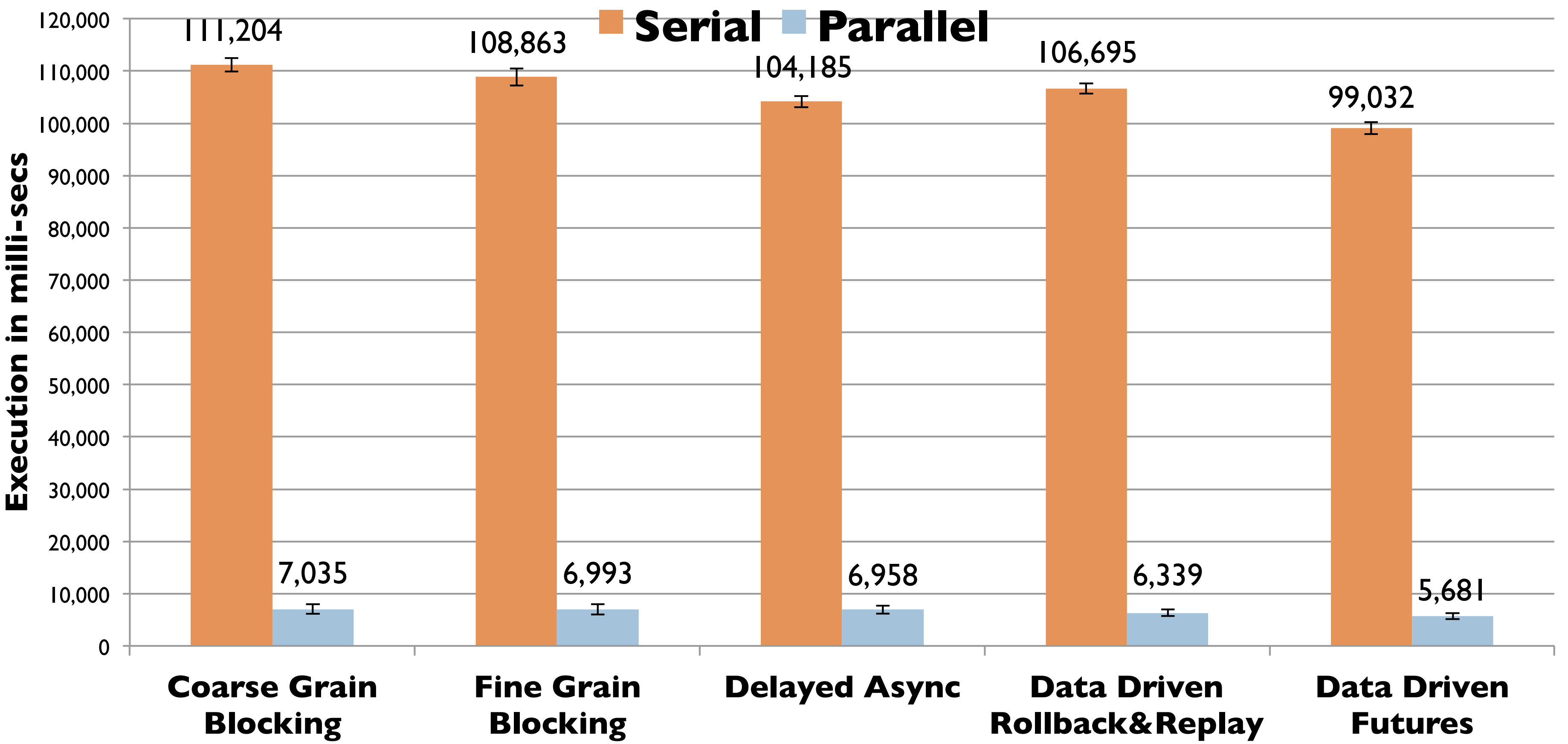
Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero Java and Intel MKL steps on Xeon with input matrix size 2000 × 2000 and with tile size 125 × 125

Cholesky decomposition



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

Cholesky decomposition

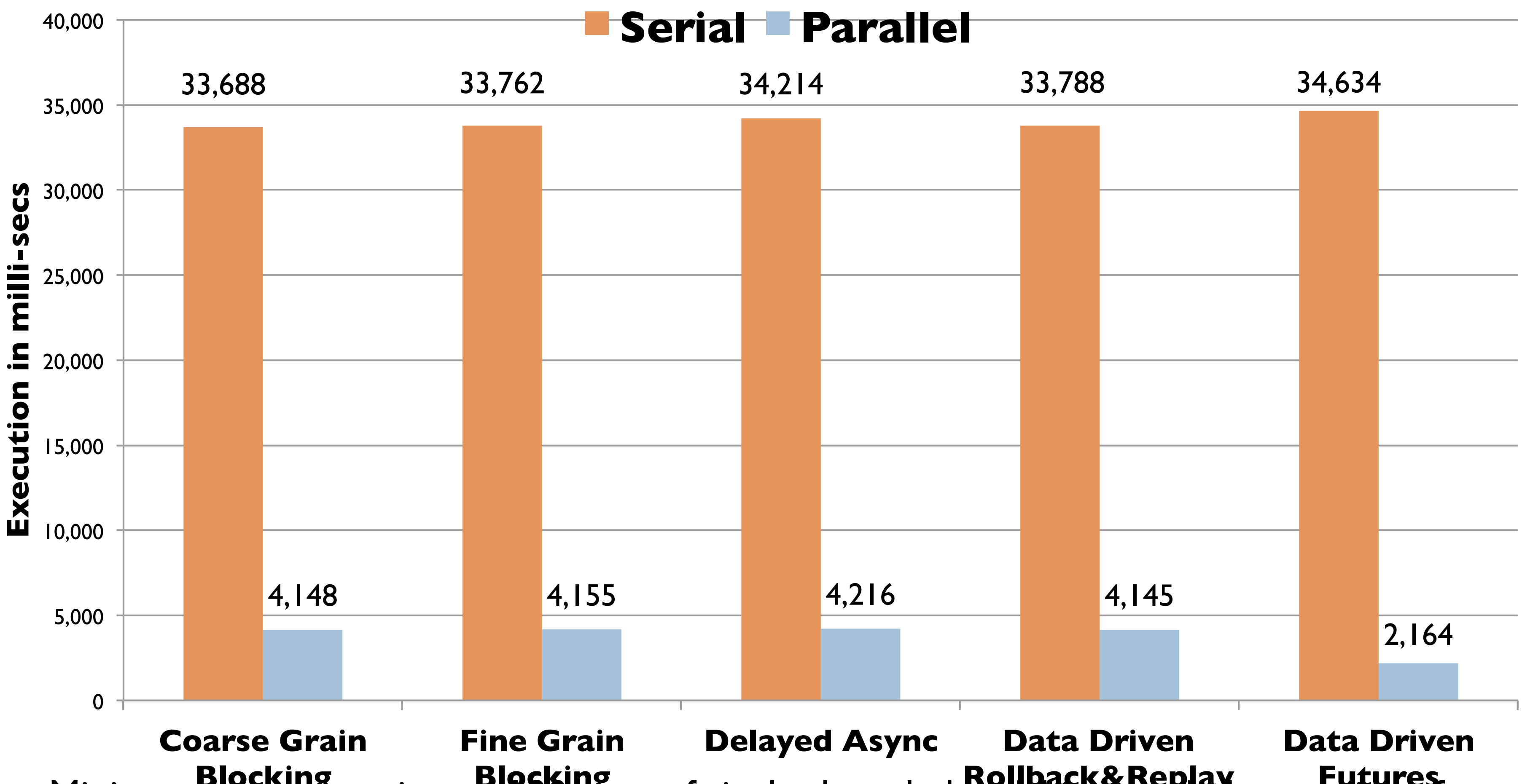


Average execution times and 90% confidence interval of 30 runs of single threaded and 16-threaded executions for blocked Cholesky decomposition CnC application with Habanero-Java steps on Niagara with input matrix size 2000 × 2000 and with tile size 125 × 125

Black-Scholes formula

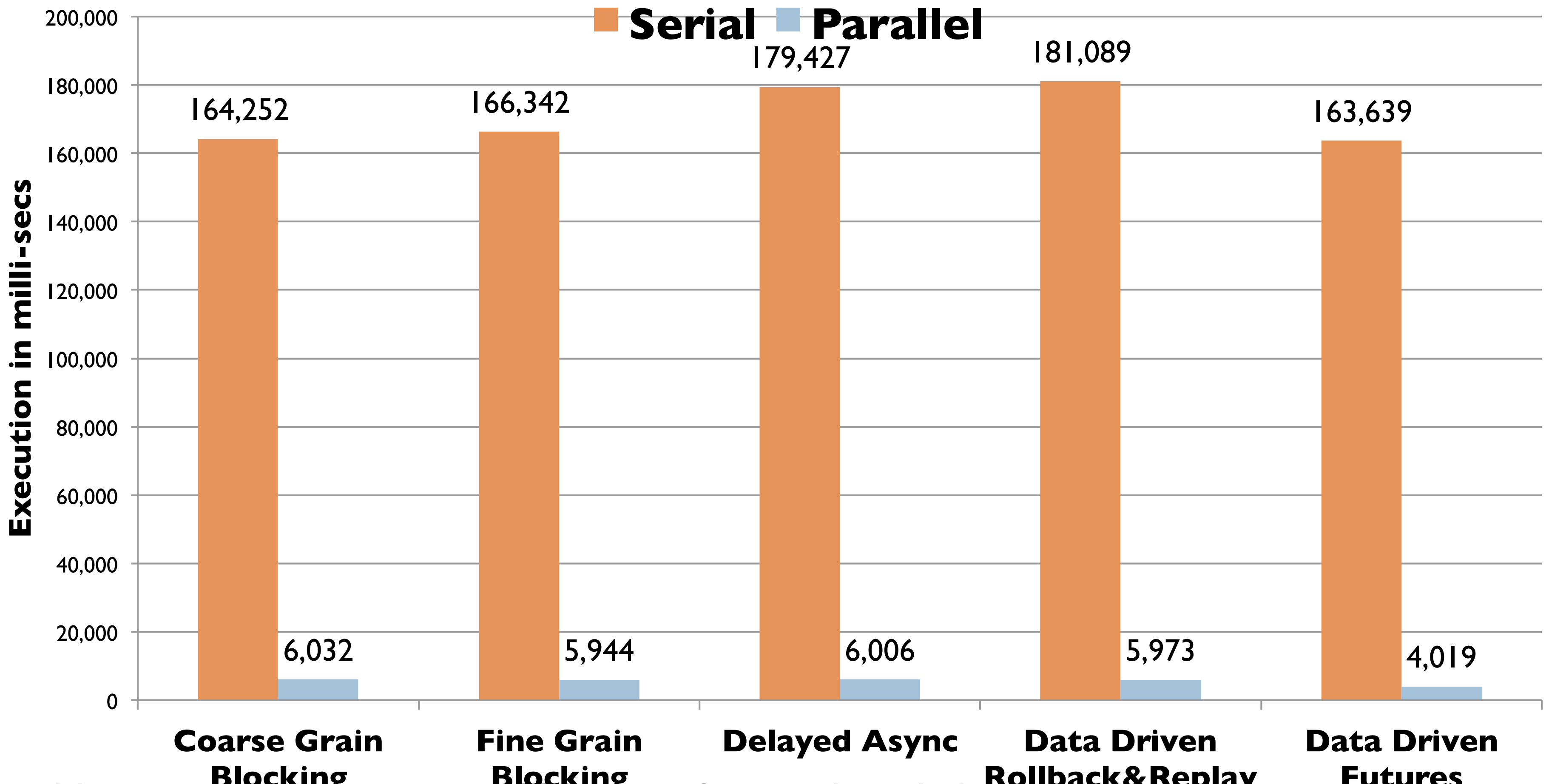
- Only one step
  - The Black-Scholes formula
- Embarrassingly parallel
- Good indicator of scheduling overhead

Black-Scholes formula



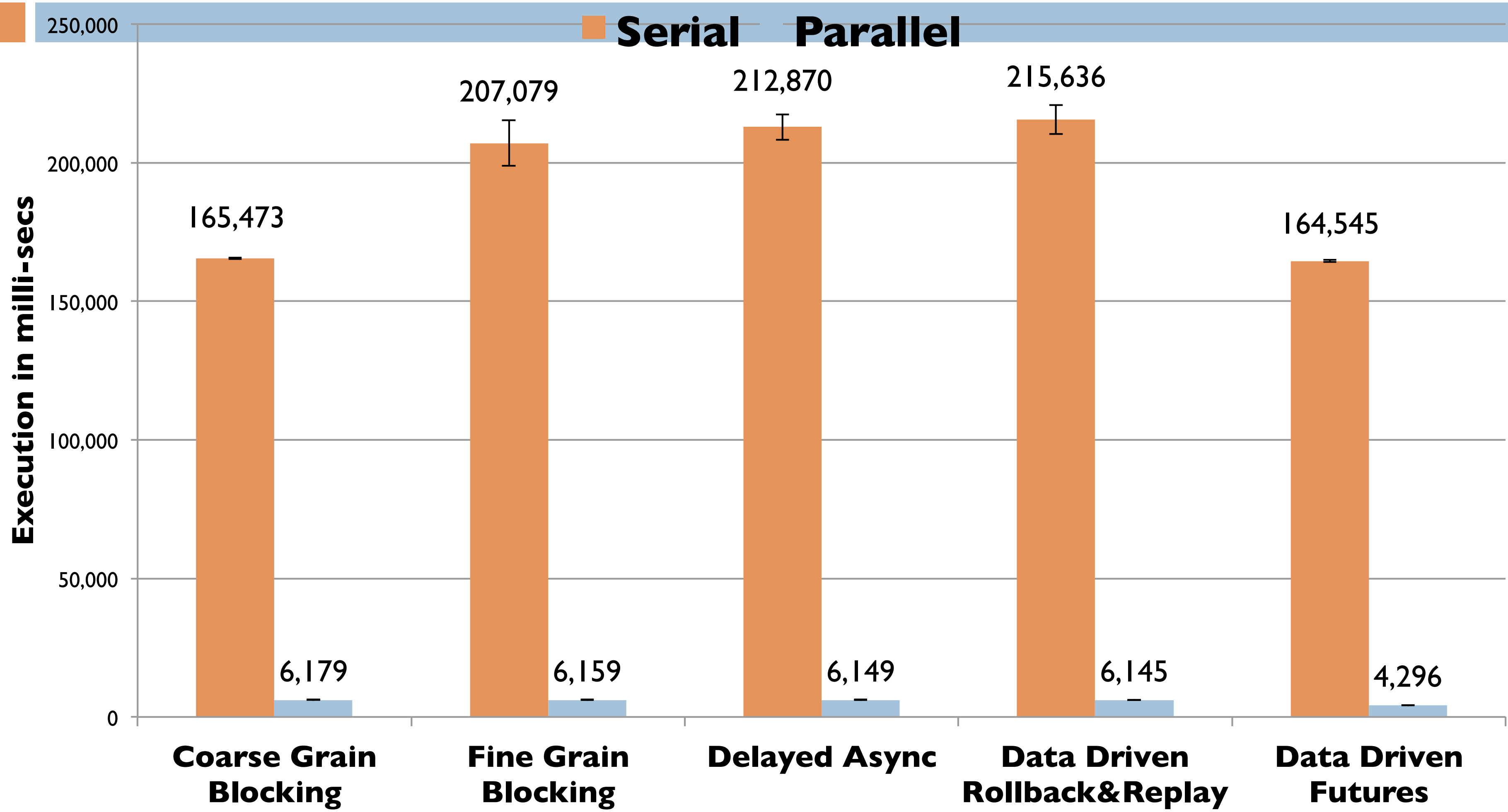
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on Xeon with input size 1,000,000 and with tile size 62,500

Black-Scholes formula



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

Black-Scholes formula

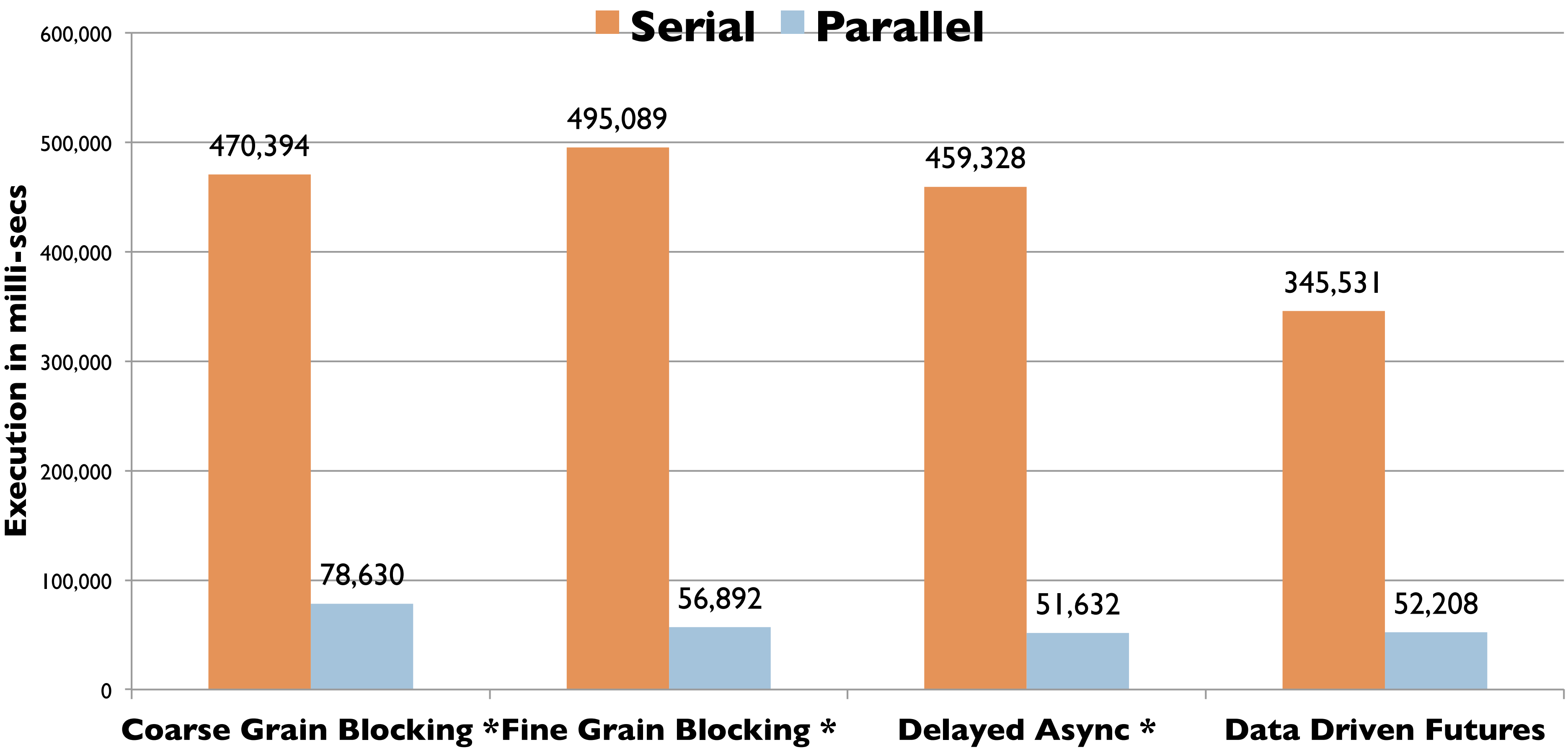


Average execution times and 90% confidence interval of 30 runs of single threaded and 64-threaded executions for blocked Black-Scholes CnC application with Habanero-Java steps on Niagara with input size 1,000,000 and with tile size 15,625

Rician Denoising

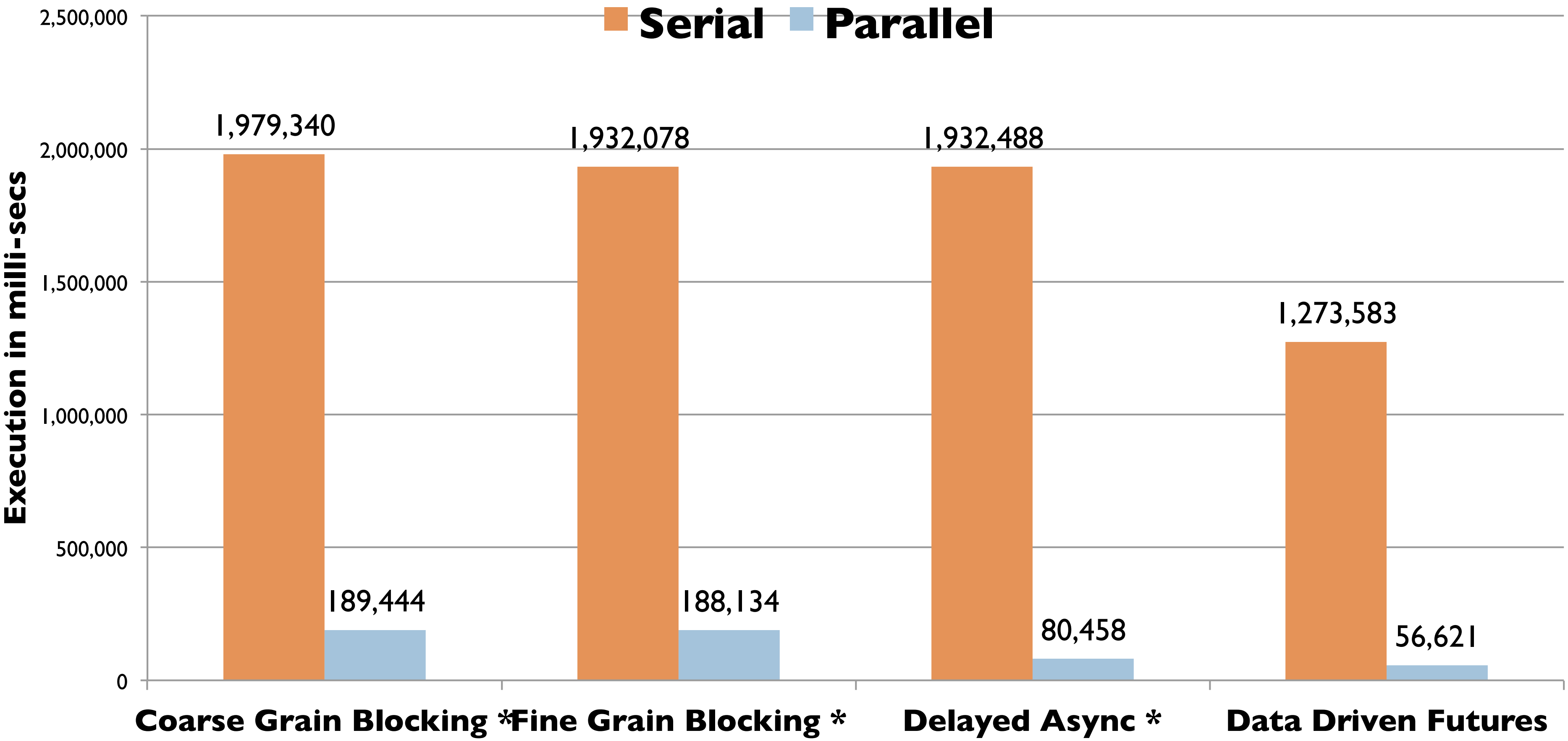
- Image processing algorithm
  - More than 4 kernels
    - Mostly stencil computations
  - Non trivial dependency graph
  - Fixed point algorithm
- Enormous data size
  - CnC schedulers needed explicit memory management
  - DDFs took advantage of garbage collection

Rician Denoising



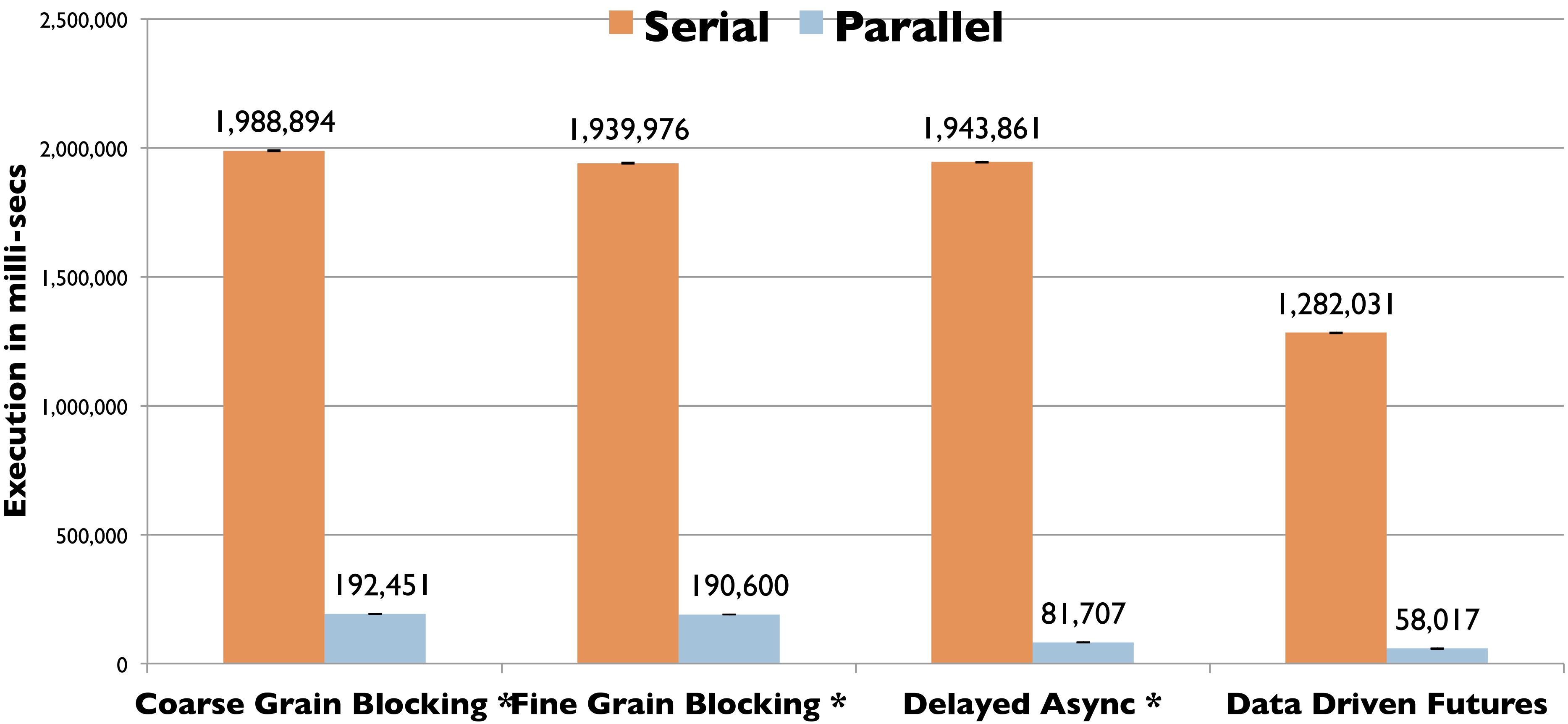
Minimum execution times of 30 runs of single threaded and 16-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on Xeon with input image size 2937 × 3872 and with tile size 267 × 484

Rician Denoising



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

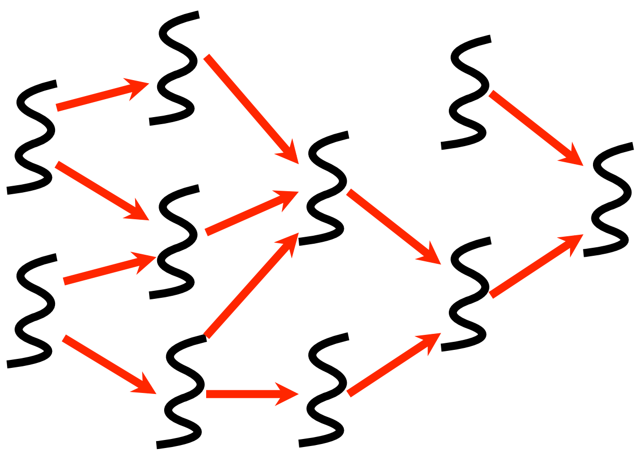
Rician Denoising



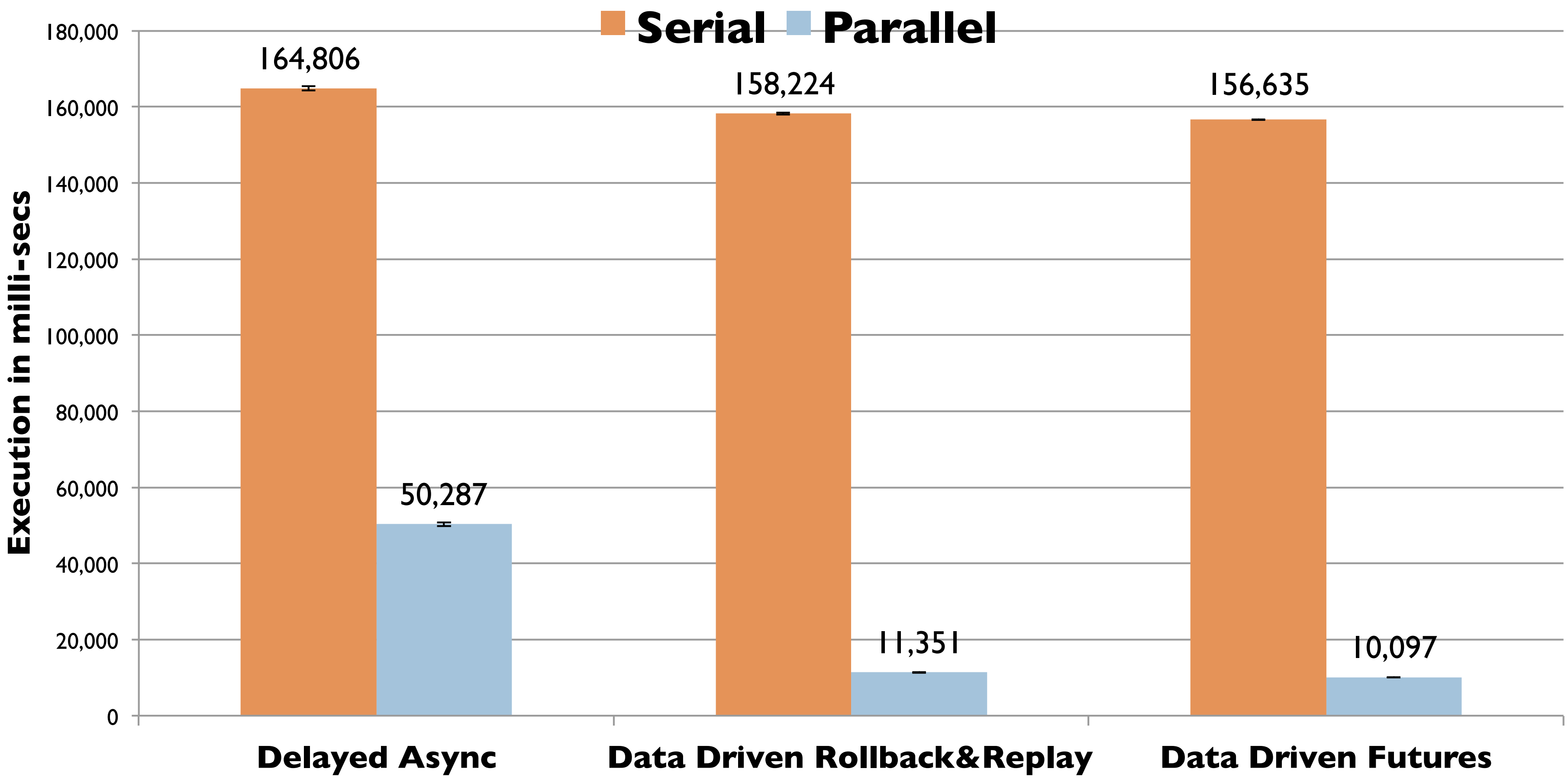
Average execution times and 90% confidence interval of 30 runs of single threaded and 64-threaded executions for blocked Rician Denoising CnC application with Habanero-Java steps on Niagara with input image size 2937 × 3872 and with tile size 267 × 484

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



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

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

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

- 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.) 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
- Committee
  - Zoran Budimlić, Keith D. Cooper, Vivek Sarkar, Lin Zhong

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