Data-Driven Tasks as an Execution Model for Concurrent Collections
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
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
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

- Create DDF\(_\alpha\), DDF\(_\beta\), DDF\(_\delta\)
- Create Task\(_A\) resolving DDF\(_\alpha\)
- Create Task\(_X\) reading DDF\(_\alpha\), DDF\(_\beta\)
- Create Task\(_D\) resolving DDF\(_\delta\)
- Create Task\(_B\) resolving DDF\(_\beta\)
- Create Task\(_Y\) reading DDF\(_\beta\), DDF\(_\delta\)
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
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

Blocking CnC Schedulers

1. Get (data-tag_γ)
2. Put(data-tag_γ, value_γ)
3. wait
4. notify

Thread_Γ

ItemCollection_θ

Thread_Δ
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 **ready** 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

Delayed async handling for work stealing scheduler
- Keep a delayed async queue per finish scope
- Every time the last async registering to finish scope
  - Traverse delayed async queue
  - Promote delayed asyncs to asyncs if guard is true
  - If any is promoted, finish scope continues

Work Sharing Ready Task Queue

- async\textsubscript{A}
- async\textsubscript{B}
- async\textsubscript{Z}

Flowchart:
- Popped Task
  - Delayed?
    - No: Requeue
    - Yes: Evaluate guard
      - Is true?
        - Yes: Assign to thread
          - No: Requeue
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

Thread $\Gamma$

- **step1**: Get (data-tag $\gamma$)
- **step3**: Get (data-tag $\gamma$)
- **step1**: Get (data-tag $\delta$)

ItemCollection $\Theta$

- **data-tag $\alpha$**: value $\alpha$
- **data-tag $\beta$**: value $\beta$
- **data-tag $\gamma$**: value $\gamma$

- **waitlist $\alpha$**
- **waitlist $\beta$**
- **waitlist $\gamma$**

Thread $\Delta$

- **step2**: Put(data-tag $\gamma$, value $\gamma$)
- **step3**: Insert step1 to waitlist $\gamma$
- **step1**: Recreate steps on waitlist $\gamma$

Throw exception to unwind
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 $2000 \times 2000$ and with tile size $125 \times 125$. 

Cholesky decomposition
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.
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 $2937 \times 3872$ and with tile size $267 \times 484$. 

Serial

## Coarse Grain Blocking *

<table>
<thead>
<tr>
<th></th>
<th>Execution in milli-secs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse Grain Blocking</td>
<td>498,776 ± 81,502</td>
</tr>
<tr>
<td>Fine Grain Blocking</td>
<td>499,666 ± 58,313</td>
</tr>
<tr>
<td>Delayed Async</td>
<td>483,770 ± 53,569</td>
</tr>
<tr>
<td>Data Driven Futures</td>
<td>349,051 ± 53,817</td>
</tr>
</tbody>
</table>

Parallel
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
User has to implement a `getAwaitsList()` that 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.
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 `getAwaitList()`** 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

```java
DataDrivenFuture[][][] outKji = null;

for (int numIters = 0; numIters < 30; ++numIters) {
    final DataDrivenFuture[][][] kj = new DataDrivenFuture[numTiles][][];
    for (int i = 0; i < numTiles; ++i) {
        k[i] = new DataDrivenFuture[numTiles+1];
        for (int j = 0; j <= i; ++j) {
            for (int k = 0; k <= numTiles; ++k) {
                kj[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) {
                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; A_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];
                    }
                }
                kj[i][j][0] = new DataDrivenFuture((java.lang.Object)temp);
                kj[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 = kj[k][k][k];
            final DataDrivenFuture pivot_kkkl = kj[k][k][k+1];
            async await (pivot_kkk) {
                s1_obj.compute([k], tileSize, pivot_kkk, pivot_kkkl);
            }
            for (int j = k + 1; j < numTiles; ++j) {
                async await (kj[j][k][k], pivot_kkkl) {
                    s2_obj.compute([k,j], tileSize, kj[j][k][k], pivot_kkkl, kj[j][k][k+1]);
                }
                for (int i = k + 1; i <= j; ++i) {
                    async await (kj[j][i][k], kj[j][i][k+1], kj[j][k][k+1]) {
                        s3_obj.compute([k,j,i], tileSize, kj[i][j][k], kj[i][j][k+1], kj[i][j][k][k+1], kj[i][j][k][k+1]);
                    }
                    async await (kj[j][j][k], kj[j][j][k+1]) {
                        s3_obj.compute([k,j,j], tileSize, kj[i][j][k], kj[i][j][k+1], kj[i][j][k][k+1], kj[i][j][k][k+1]);
                    }
                }
            }
        }
    }
    long end = java.lang.System.currentTimeMillis();
    System.out.println("Time: "+(end-begin));
}
```
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
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 $2000 \times 2000$ and with tile size $125 \times 125$
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 $2000 \times 2000$ and with tile size $125 \times 125$
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 $2000 \times 2000$ and with tile size $125 \times 125$. 
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
Only one step
- The Black-Scholes formula
Embarrassingly parallel
Good indicator of scheduling overhead
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.
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
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
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 $2937 \times 3872$ and with tile size $267 \times 484$. 

![Bar chart showing execution times for different blocking strategies.](image-url)
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 $2937 \times 3872$ and with tile size $267 \times 484$. 

Coarse Grain Blocking *Fine Grain Blocking * Delayed Async * Data Driven Futures
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 $2937 \times 3872$ and with tile size $267 \times 484$. 
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