CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar

Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/

Recap of Lecture 2

- Parallel Array Sum (contd)
- Speedup, Efficiency, Amdahl's Law
- Understanding Data and Control Flow between an Async Task and its Parent
- Data Races and Determinism

Worksheet #3 solution: Complexity analysis of k-way Parallel Array Sum algorithm

- Consider a k-way parallel array-sum algorithm, where 1 <= k <= n
  - Compute k partial sums in parallel, each of size n/k
  - Sequentially combine the k partial sums into a single sum
- Total number of additions, WORK = k (n/k -1) + k = O(n)
- What is the critical path length?
  - CPL = O(n/k + k)
  - Stage 1 takes O(n/k) time and Stage 2 takes O(k) time
- What value of k gives the smallest value of CPL?
  - Optimal value of k = sqrt(n)

Outline of Today's Lecture

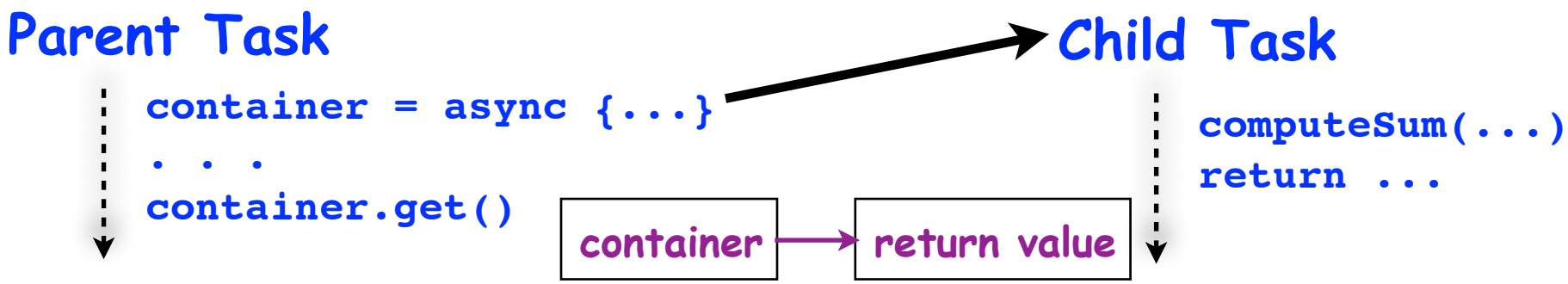
- <u>Futures --- Tasks with Return Values</u>
- Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)
- Finish Accumulators

Extending Async Tasks with Return Values

- Example Scenario in PseudoCode

```
1. // Parent task creates child async task
2. container = async<int>  { return computeSum(X, low, mid); };
3. . . .
4. // Later, parent examines the return value
5. int sum = container.get();
```

- Two key issues to be addressed:
  1) Distinction between container and value in container
  2) Synchronization to avoid race condition in container accesses



HJ Futures: Tasks with Return Values

```
async<T> { <Stmt-Block> }
```

- Creates a new child task that executes Stmt-Block, which must terminate with a return statement returning a value of type T
- Async expression returns a reference to a container of type future<T>
- Values of type future<T> can only be assigned to final variables

```
Expr.get()
```

- Evaluates Expr, and blocks if Expr's value is unavailable
- Expr must be of type future<T>
- Return value from Expr.get() will then be T
- Unlike finish which waits for all tasks in the finish scope, a get() operation only waits for the specified async expression

Example: Two-way Parallel Array Sum using Future Tasks

```
// Parent Task T1 (main program)
// Compute sum1 (lower half) and sum2 (upper half) in parallel
final future<int> sum1 = async<int> { // Future Task T2
  int sum = 0;
  for(int i=0 ; i < X.length/2 ; i++) sum += X[i];
  return sum;
}; //NOTE: semicolon needed to terminate assignment to sum1
final future<int> sum2 = async<int> { // Future Task T3
  int sum = 0;
  for(int i=X.length/2 ; i < X.length ; i++) sum += X[i];
  return sum;
}; //NOTE: semicolon needed to terminate assignment to sum2
//Task T1 waits for Tasks T2 and T3 to complete
int total = sum1.get() + sum2.get();
```

Why are these semicolons needed?

Future Task Declarations and Uses

- Variable of type future<T> is a reference to a future object
  - Container for return value of T from future task
  - The reference to the container is also known as a "handle"
- Two operations that can be performed on variable V1 of type future<T1> (assume that type T2 is a subtype of type T1):
  - Assignment: V1 can be assigned value of type future<T2>
  - Blocking read: V1.get() waits until the future task referred to by V1 has completed, and then propagates the return value
- Future task body must start with a type declaration, async<T1>, where T1 is the type of the task's return value
- Future task body must consist of a statement block enclosed in { } braces, terminating with a return statement

Comparison of Future Task and Regular Async Versions of Two-Way Array Sum

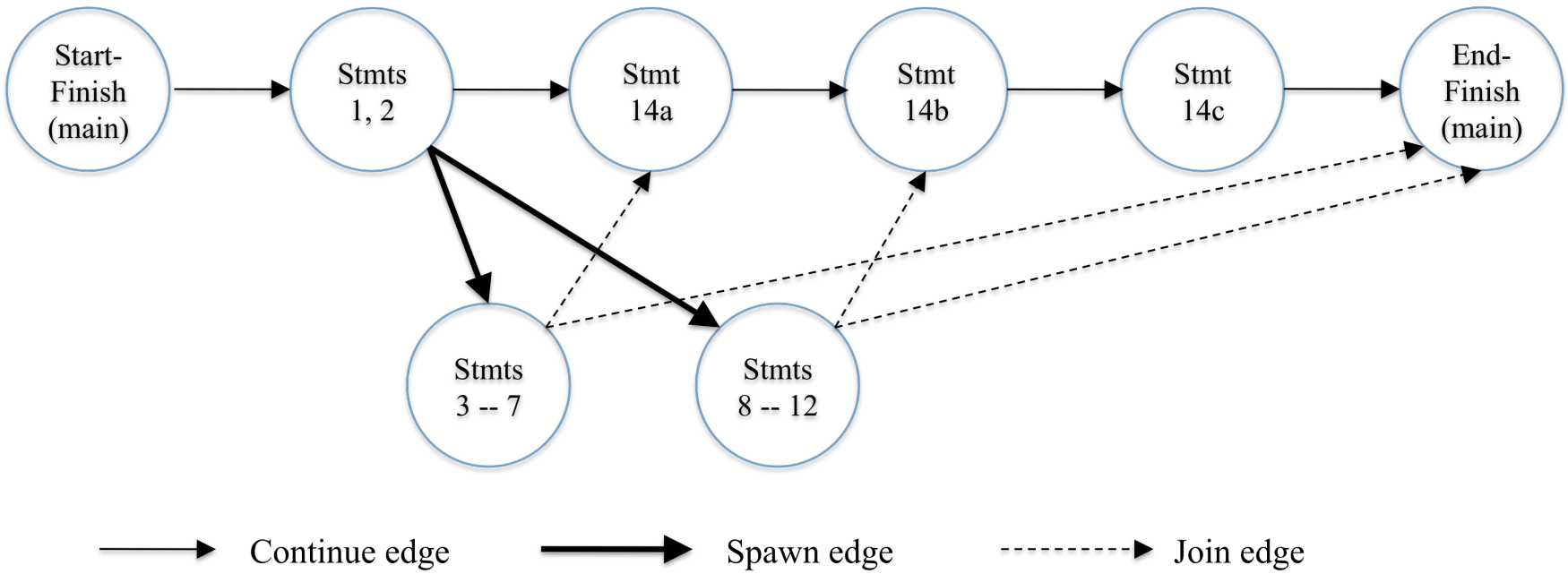
- Future task version initializes two references to future objects, sum1 and sum2, and both are declared as final
- No finish construct needed in this example
  - Instead parent task waits for child tasks by performing sum1.get() and sum2.get()
- Guaranteed absence of race conditions in Future Task example
  - No race on sum because it is a local variable in tasks T2 and T3
  - No race on future variables, sum1 and sum2, because of blocking-read semantics

Computation Graph Extensions for Future Tasks

- Since a get() is a blocking operation, it must occur on boundaries of CG nodes/steps
  - May require splitting a statement into sub-statements e.g.,
    - 14: `int sum = sum1.get() + sum2.get();`

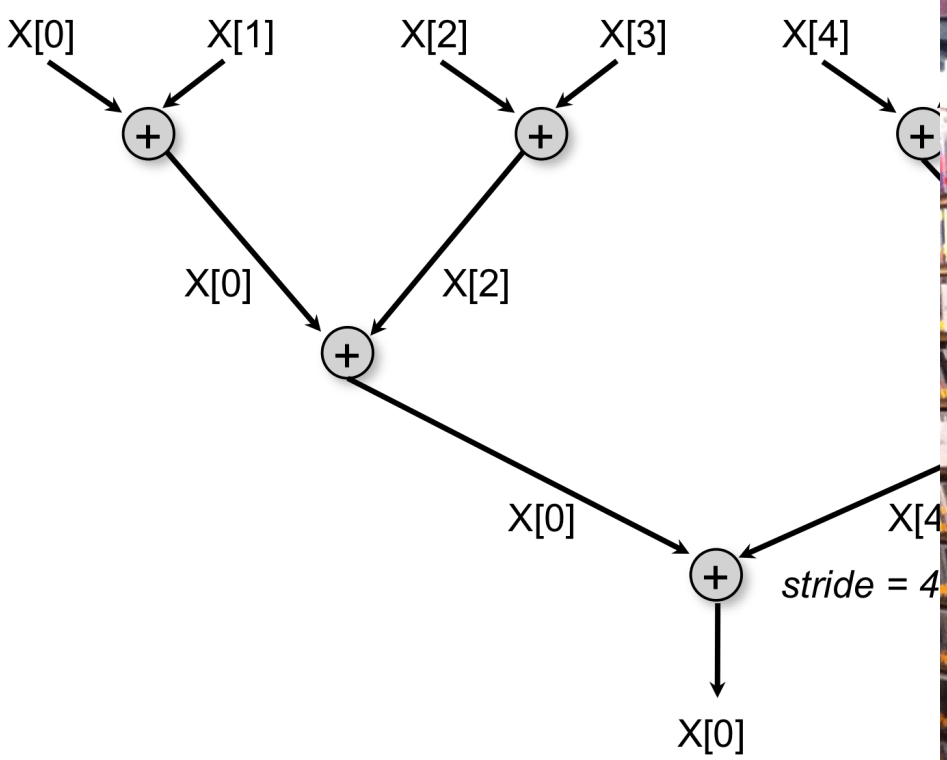
    can be split into three sub-statements
    - 14a `int temp1 = sum1.get();`
    - 14b `int temp2 = sum2.get();`
    - 14c `int sum = temp1 + temp2;`
- Spawn edge connects parent task to child future task, as before
- Join edge connects end of future task to Immediately Enclosing Finish (IEF), as before
- Additional join edges are inserted from end of future task to each get() operation on future object

Computation Graph for Two-way Parallel Array Sum using Future Tasks



NOTE: Generation of computation graphs and data race detection in current HJ implementation do not support futures as yet

Reduction Tree Schem
(Recap



Questions:

- **How can we implement this schema u**
- **Can we avoid overwriting elements of**

Array Sum using Future Tasks (ArraySum2)

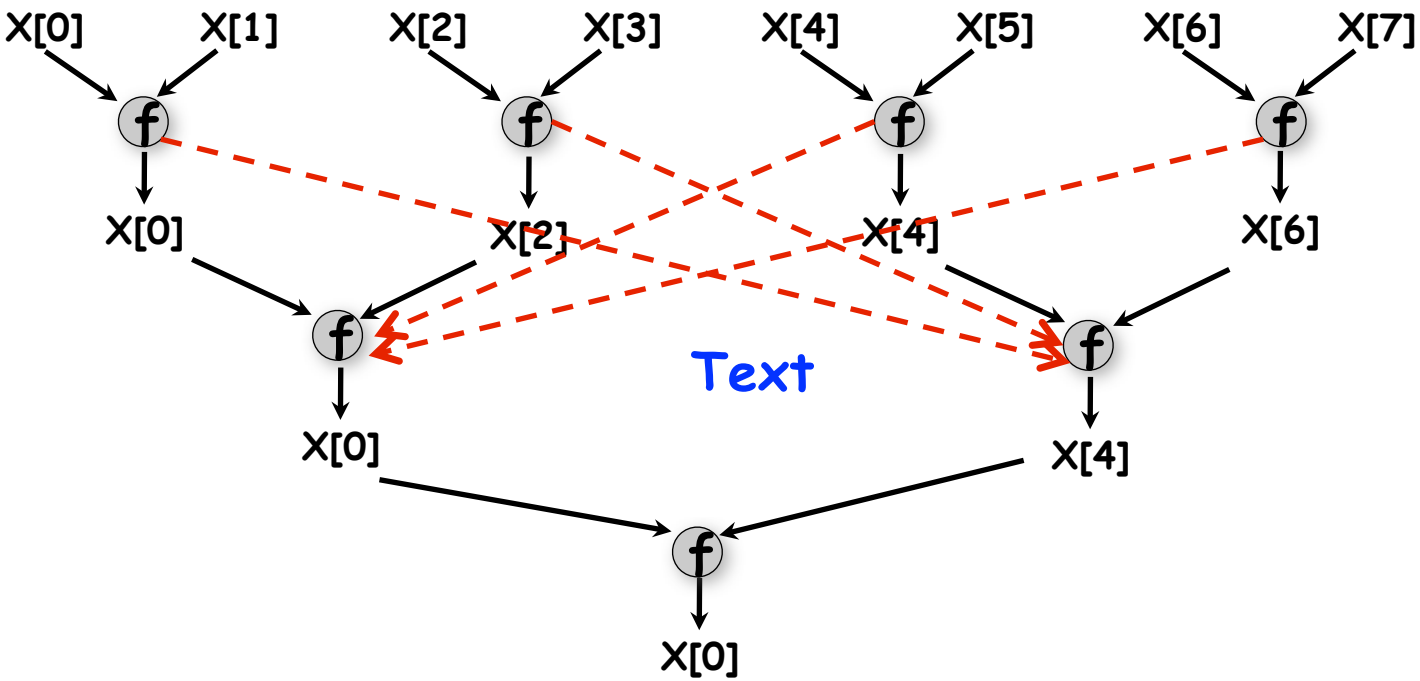
Recursive divide-and-conquer pattern

```
1.  static int computeSum(int[] X, int lo, int hi) {
2.    if ( lo > hi ) return 0;
3.    else if ( lo == hi ) return X[lo];
4.    else {
5.      int mid = (lo+hi)/2;
        final future<int> sum1 =
6.            async<int> { return computeSum(X, lo, mid); };
7.      final future<int> sum2 =
8.           async<int> { return computeSum(X, mid+1, hi); };
9.     // Parent now waits for the container values
10.     return sum1.get() + sum2.get();
11.    }
12.  } // computeSum
13. int sum = computeSum(X, 0, X.length-1); // main program
```

Extension of ArraySum2 to reduce an arbitrary associative function, f

```
static int reduce(int[] X, int lo, int hi) {
  if ( lo > hi ) return identity();
  else if ( lo == hi ) return X[lo];
  else {
    int mid = (lo+hi)/2;
    final future<int> sum1 =
      async<int> {return computeSum(X, lo, mid);};
    final future<int> sum2 =
      async<int> {return computeSum(X, mid+1, hi);};
    return f(sum1.get(), sum2.get());
  }
} // computeSum
int retVal = reduce(X, 0, X.length-1); // main program
```

Extra dependences in ArraySum1 program (for-finish-for-async)



- - - → Extra dependence edges due to finish-async stages (not present in ArraySum2 version with futures)

- Which of ArraySum1 or ArraySum2 will perform better if the time taken by the reduction operator depends on its inputs e.g., as in WordCount ?

Why must Future References be declared as final?

```
static future<int> f1=null;

static future<int> f2=null;



void main(String[] args) {

  f1 = async<int> {return a1();};

  f2 = async<int> {return a2();};
```

```
int a1() { // Task T1
 while (f2 == null); // spin loop
  return f2.get(); //T1 waits for T2
}


int a2() { // Task T2
  while (f1 == null); // spin loop
  return f1.get(); //T2 waits for T1
}
```

cyclic wait condition

- Above situation cannot arise in HJ because f1 and f2 must be final
- Final declaration ensures that variable (handle) cannot be modified after initialization
- WARNING: such spin loops are an example of bad parallel programming practice in application code (they should only be used by expert systems programmers, and even then sparingly)
  - Their semantics depends on the memory model!

Future Tasks with void Return Type

- Key difference between regular async's and future tasks is that future tasks have a future<T> return value
- We can get an intermediate capability by setting T=void as shown
- Can be useful if a task needs to synchronize on a specific task (instead of finish), but doesn't need a future object to communicate a return value

```
sum1 = 0; sum2 = 0; // Task T1
// Assume that sum1 & sum2 are fields
final future<void> a1 = async<void> {
  for (int i=0; i < X.length/2; i++)
      sum1 += X[i]; // Task T2
};
final future<void> a2 = async<void> {
  for (int i=X.length/2; i < X.length; i++)
      sum2 += X[i]; // Task T3
```

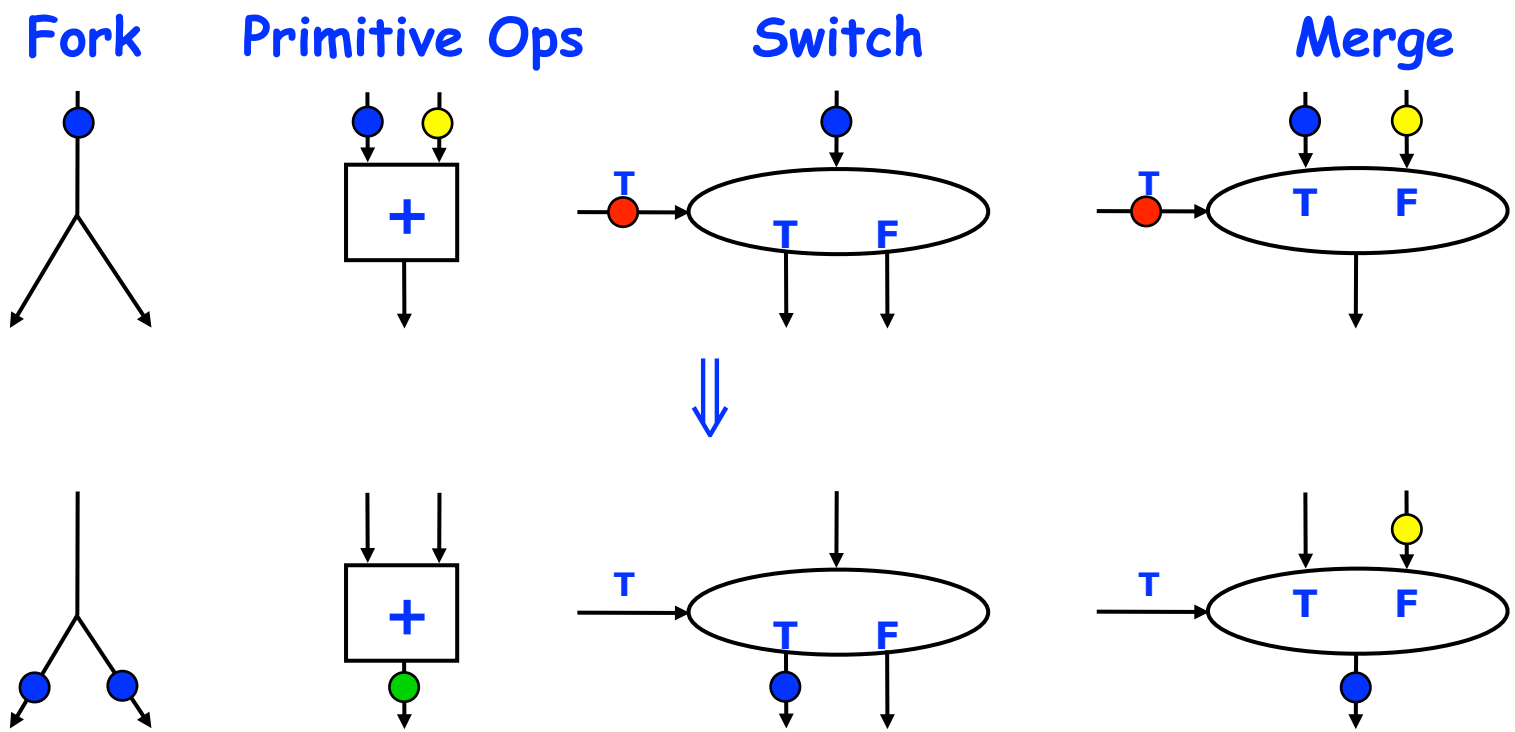
Which construct is more general: futures or async-finish? Let's do worksheet #4 to figure it out!

Outline of Today's Lecture

- Futures --- Tasks with Return Values
- <u>Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)</u>
- Finish Accumulators

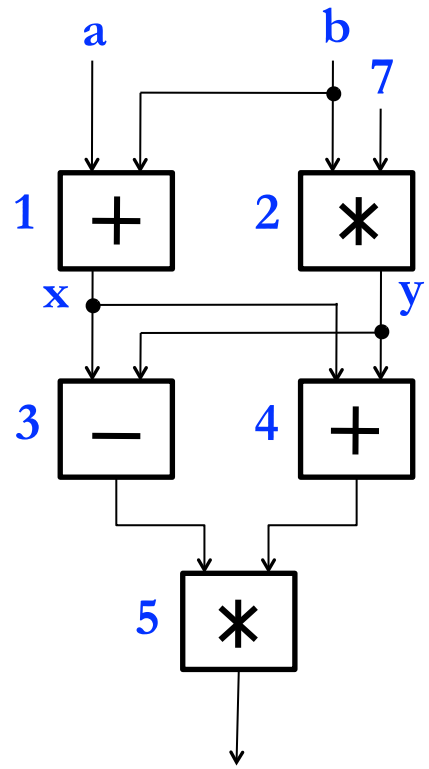
Dataflow Computing

- **Original idea: replace machine instructions by a small set of dataflow operators**



Example instruction sequence and its dataflow graph

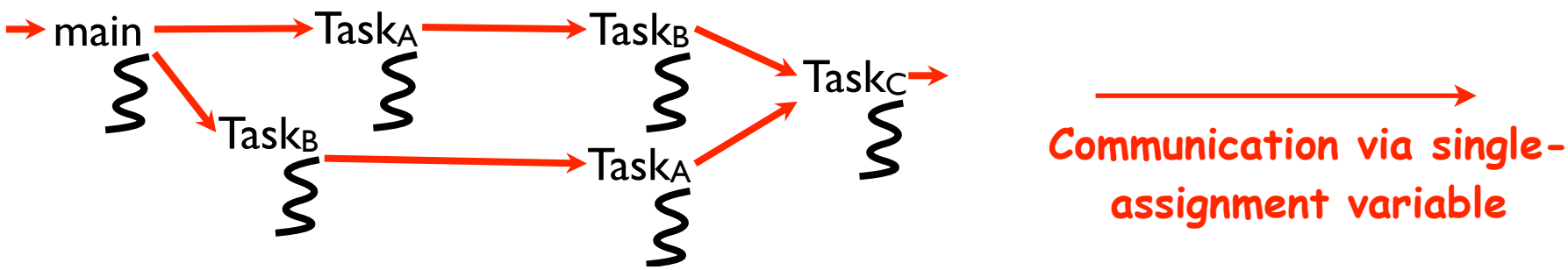
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



An operator executes when all its input values are present; copies of the result value are distributed to the destination operators.

No separate control flow

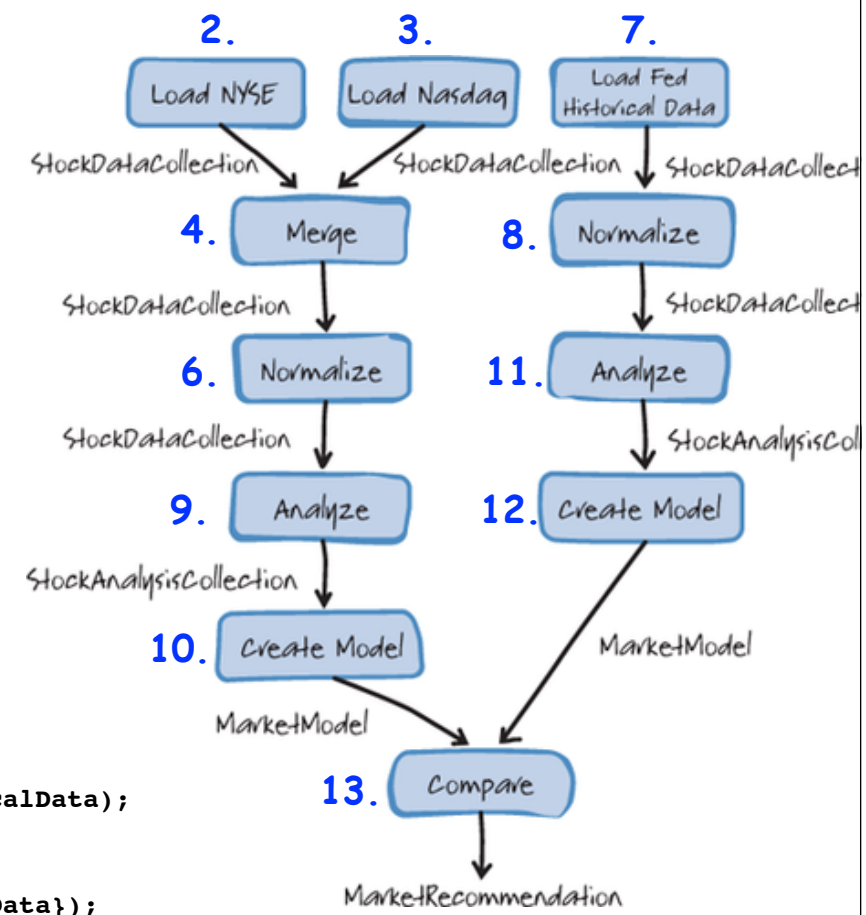
Productivity Benefits of Macro-Dataflow Programming



- "Macro-dataflow" = extension of dataflow model from instruction-level to task-level operations
- General idea: build an arbitrary task graph, but restrict all inter-task communications to single-assignment variables
  - Static dataflow ==> graph fixed when program execution starts
  - Dynamic dataflow ==> graph can grow dynamically
- Semantic guarantees: race-freedom, determinism
  - Deadlocks are possible due to unavailable inputs (but they are deterministic)

"Adatum Dashboard" Example: Sequential Version

```
1.public MarketRecommendation DoAnalysisSequential() {
2.  StockDataCollection nyseData = LoadNyseData();
3.  StockDataCollection nasdaqData = LoadNasdaqData();
4.  StockDataCollection mergedMarketData =
5.    MergeMarketData(new[]{nyseData, nasdaqData});
6.  StockDataCollection normalizedMarketData =
        NormalizeData(mergedMarketData);
7.  StockDataCollection fedHistoricalData =
        LoadFedHistoricalData();
8.  StockDataCollection normalizedHistoricalData =
        NormalizeData(fedHistoricalData);
9.  StockAnalysisCollection analyzedStockData =
        AnalyzeData(normalizedMarketData);
10. MarketModel modeledMarketData = RunModel(analyzedStockData);
11. StockAnalysisCollection analyzedHistoricalData =
        AnalyzeData(normalizedHistoricalData);
12. MarketModel modeledHistoricalData = RunModel(analyzedHistoricalData);
13. MarketRecommendation recommendation =
14.    CompareModels(new[] {modeledMarketData, modeledHistoricalData});
15.  return recommendation;
16.}
```



Source: http://programming4.us/enterprise/3004.aspx

Extending HJ Futures for Macro-Dataflow: Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)

```
ddfA = new DataDrivenFuture<T1>();
```

- Allocate an instance of a data-driven-future object (container)
- Object in container must be of type T1

```
async await(ddfA, ddfB, …) <Stmt>
```

- Create a new data-driven-task to start executing Stmt after all of ddfA, ddfB, … become available (i.e., after task becomes "enabled")

```
ddfA.put(V) ;
```

- Store object V (of type T1) in ddfA, thereby making ddfA available
- Single-assignment rule: at most one put is permitted on a given DDF

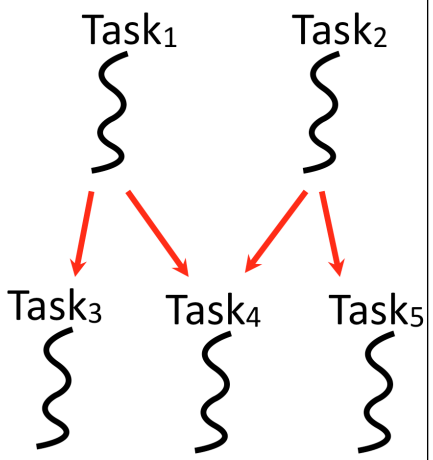
```
ddfA.get()
```

- Return value (of type T1) stored in ddfA
- Can only be performed by async's that contain ddfA in their await clause (hence no blocking is necessary for DDF gets)

Example Habanero Java code fragment with Data-Driven Futures

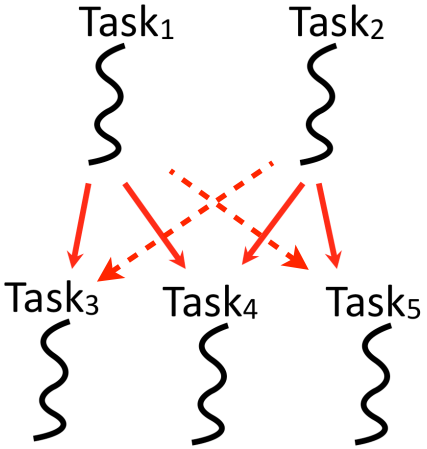
```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.   async await(left) leftReader(left); // Task3
5.   async await(right) rightReader(right); // Task5
6.   async await(left,right)
7.         bothReader(left,right); // Task4
8.   async left.put(leftWriter()); // Task1
9.   async right.put(rightWriter());// Task2
10. }
```

- await clauses capture data flow relationships
- type parameter is optional for DataDrivenFuture
  - if omitted, may require cast operators to be inserted instead
  - (just as with standard Java generics in sequential programs)



Finish-async version of the same example has more dependences

```
// Assume that left and right are fields in this object
finish {
  async left = put(leftWriter()); // Task1
  async right = put(rightWriter()); // Task2
}
finish {
  async leftReader(left); // Task3
  async rightReader(right); // Task5
  async bothReader(left, right); // Task4
}
```



Two Exception (error) cases for DDFs

- <u>Case 1:</u> If two put's are attempted on the same DDF, an exception is thrown because of the violation of the single-assignment rule

- <u>Case 2:</u> If a *get* is attempted by a task on a DDF that was not in the task's await list, then an exception is thrown because DDF's do not support blocking *gets*.

Deadlock example with DDTs

```
1. DataDrivenFuture left = new DataDrivenFuture();
2. DataDrivenFuture right = new DataDrivenFuture();
3. finish {
4.   async await(left) right.put(rightWriter());
5.   async await(right) left.put(leftWriter());
6. }
```

"Adatum Dashboard" Example: Parallel Version using DDTs and DDFs

```
public MarketRecommendation DoAnalysisParallelDDT() {
  async nyseData.put(LoadNyseData());
  async nasdaqData.put(LoadNasdaqData());
  async await(nyseData, nasdaqData)
    mergedMarketData.put(MergeMarketData(new[]{nyseData.get(), nasdaqData.get()}));
  async await(mergedMarketData) normalizedMarketData.put(NormalizeData(mergedMarketData.get()));
  async fedHistoricalData.put(LoadFedHistoricalData());
  async await(fedHistoricalData) normalizedHistoricalData.put(NormalizeData(fedHistoricalData.get()));
  async await(normalizedMarketData) analyzedStockData.put(AnalyzeData(normalizedMarketData.get()));
  async await(analyzedStockData) modeledMarketData.put(RunModel(analyzedStockData.get()));
  async await(normalizedHistoricalData) analyzedHistoricalData.put(AnalyzeData(normalizedHistoricalData.get()));
  async await(analyzedHistoricalData) modeledHistoricalData.put(RunModel(analyzedHistoricalData.get()));
  MarketRecommendation recommendation =
     CompareModels(new[] {modeledMarketData.get(), modeledHistoricalData.get()});
  return recommendation;
}
```

Note that the put, await, and get clauses follow directly from the data flow structure of the program!

Differences between Futures and DDFs/DDTs

- Consumer blocks on get() for each future that it reads, whereas async-await does not start execution till all DDFs are available
- Producer task can only write to a single future object, where as a DDF task can write to multiple DDF objects
- The choice of which future object to write to is tied to a future task at creation time, where as the choice of output DDF can be deferred to any point with a DDF task
- Future tasks cannot deadlock, but it is possible for a DDF task to never be enabled, if one of its input DDFs never becomes available. This can be viewed as a special case of deadlock.
  - This deadlock case can be resolved by ensuring that each finish construct moves past the end-finish when all enabled async tasks in its scope have terminated, thereby ignoring any remaining non-enabled async tasks.

Implementing Future Tasks using DDFs

- Future version

```
final future<int> f = async<int> { return g(); };
...
... = f.get();
```

- DDF version

```
DataDrivenFuture f = new DataDrivenFuture();
async { f.put(g()) };
 ...
finish async await (f) { ... = f.get(); };
```

Implementing DDFs/DDTs using Future tasks

- DDF version

```
DataDrivenFuture f1 = new DataDrivenFuture();
DataDrivenFuture f2 = new DataDrivenFuture();
async { f1.put(g()) }; async { f2.put(h()) };
// async doesn't start till f1 & f2 are available
async await (f1, f2) { ... = f1.get() + f2.get(); };
```

- Future version

```
final future<int> f1 = async<int> { return g(); };
final future<int> f2 = async<int> { return h(); };
// Async may block at each get() operation
async { ... = f1.get() + f2.get(); };
```

Outline of Today's Lecture

- Futures --- Tasks with Return Values
- Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)
- <u>Finish Accumulators</u>

Finish Accumulators in HJ

- Creation

```
accumulator ac = accumulator.factory.accumulator(operator, type);
```

  - operator can be Operator.SUM, Operator.PROD, Operator.MIN, Operator.MAX or Operator.CUSTOM
  - type can be int.class or double.class for standard operators or any object that implements a "reducible" interface for CUSTOM

- Registration

```
finish (ac1, ac2, ...) { ... }
```

  - Accumulators ac1, ac2, ... are registered with the finish scope

- Accumulation

```
ac.put(data);
```

  - can be performed by any statement in finish scope that registers ac

- Retrieval

```
Number n = ac.get();
```

  - get() is nonblocking because finish provides the necessary synchronization

    Either returns initial value before end-finish or final value after end-finish

  - result from get() will be deterministic if CUSTOM operator is associative and commutative

Example with Multiple Finish Accumulators

```
1.   // T1 allocates accumulator a and b
2.   accumulator a = accumulator.factory.accumulator(SUM, int.class);
3.   accumulator b = accumulator.factory.accumulator(MIN, double.class);
4.   // T1 can invoke put()/get() on a and b any time
5.   a.put(1); // adds 1 to accumulator a
6.   Number v1 = a.get(); // Returns 1
7.   // T1 creates a finish scope registered on a and b
8.   finish (a, b) {
9.     // Any task can invoke put() within the finish
10.    b.put(2.5); // min operation with accumulator b
11.    finish { // Inner finish inherits registrations for a & b
12.      async a.put(2);
13.      b.put(1.5);
14.    }
15.    // Unlikely case: if a task invokes get() within the finish,
16.    // the value returned value is that on entry to the finish
17.    Number v2 = a.get(); // Returns 1
18.  }
19.  // T1 obtains overall sum and min values after end-finish
20.  Number v3 = a.get(); // Returns 1 + 2 = 3
21.  Number v4 = b.get(); // Returns min(2.5,1.5) = 1.5
```

Error Conditions with Finish Accumulators

1. Non-owner task cannot access accumulators outside registered finish

```
// T1 allocates accumulator a
accumulator a = accumulator.factory.accumulator(...);
async { // T2 cannot access a
  a.put(1); Number v1 = a.get();
}
```

2. Non-owner task cannot register accumulators with a finish

```
// T1 allocates accumulator a
accumulator a = accumulator.factory.accumulator(...);
async {
  // T2 cannot register a with finish
  finish (a) { async a.put(1);  }
}
```

Worksheet #4: Computation Graphs for Async-Finish and Future Constructs

1) Can you write an HJ program with <u>async-finish</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

2) Can you write an HJ program with <u>future async-get</u> constructs that generates a Computation Graph with the same ordering constraints as the graph on the right? If so, provide a sketch of the program.

Use the space below for your answers