COMP 322: Fundamentals of Parallel Programming

Lecture 8: Dataflow Programming with Futures and Data-Driven Futures

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

Acknowledgments

- "Parallel Programming with Microsoft .Net : Futures"
  - http://programming4.us/enterprise/3004.aspx
- "A Wavefront Parallelisation of CTMC Solution using MTBDDs", Yi Zhang, David Parker, Marta Kwiatkowska
  - http://www.prismmodelchecker.org/papers/pds05.pdf
  - www.gridpp.ac.uk/gridpp14/mesc.ppt
- Data-Driven Tasks and their Implementation. Sagnak Tasirlar, Vivek Sarkar. Proceedings of the International Conference on Parallel Processing (ICPP) 2011, September 2011.

Goals for Today's Lecture

- <u>Recap of Future Tasks from Lecture 7</u>
- **Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)**

HJ Futures: Tasks with Return Values (Recap)

`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

Extending Async Tasks with Return Values

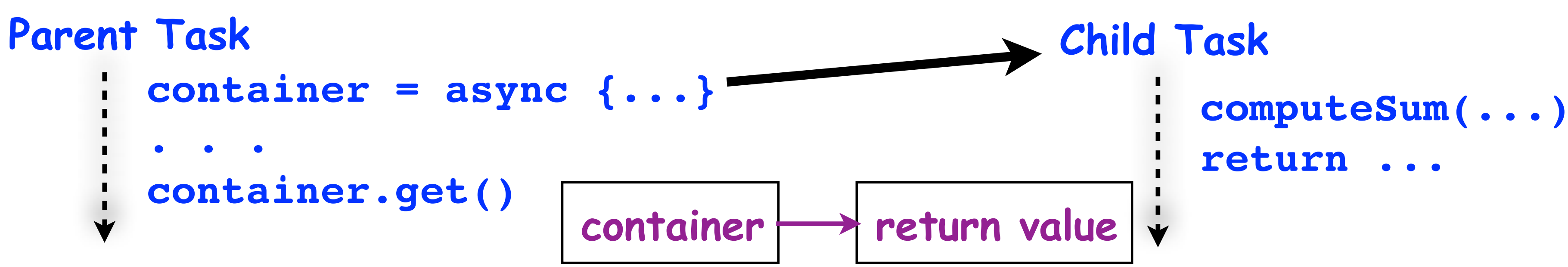
- Example Scenario in PseudoCode

```
// Parent task creates child async task
future<int> container = async<int> { return computeSum(...); };
. . .
// Later, parent examines the return value
int sum = container.get();
```

- Two key issues to be addressed:

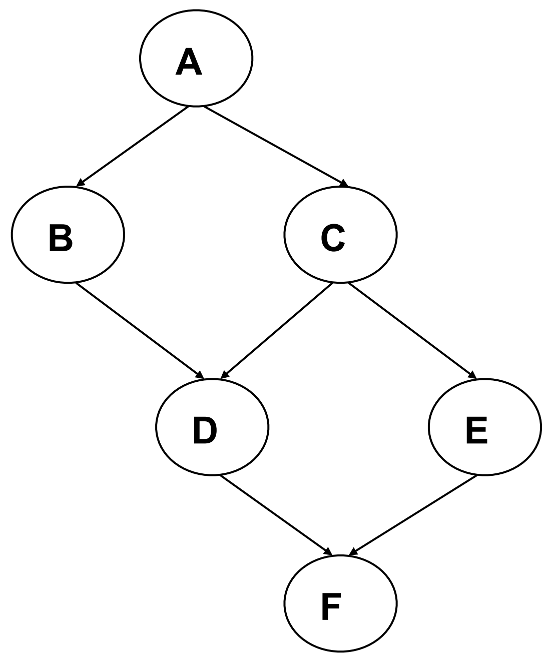
  1) Distinction between container and value in container
     - Types future<T> vs T

  2) Synchronization to avoid race condition in container accesses
     - get() operation blocks until value becomes available

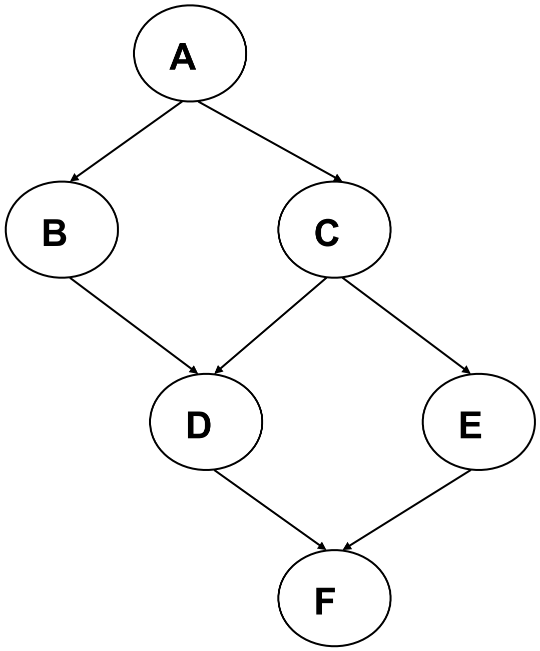


Future Tasks can generate more general Computation Graphs than regular Async Tasks

Can you write a finish-async HJ program that generates the following Computation Graph?



Using Future Tasks to generate previous Computation Graph



Computation Graph

```
// NOTE: return statement is optional
// when return type is void
final future<void> A = async<void>
                  { . . . };
final future<void> B = async<void>
                  { A.get(); . . . };
final future<void> C = async<void>
                  { A.get(); . . . };
final future<void> D = async<void>
                  { B.get(); C.get(); . . . };
final future<void> E = async<void>
                  { C.get(); . . . };
final future<void> F = async<void>
                  { D.get(); E.get(); . . . }
```

Goals for Today's Lecture

- Recap of Future Tasks from Lecture 7
- Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)

Dataflow Computing

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

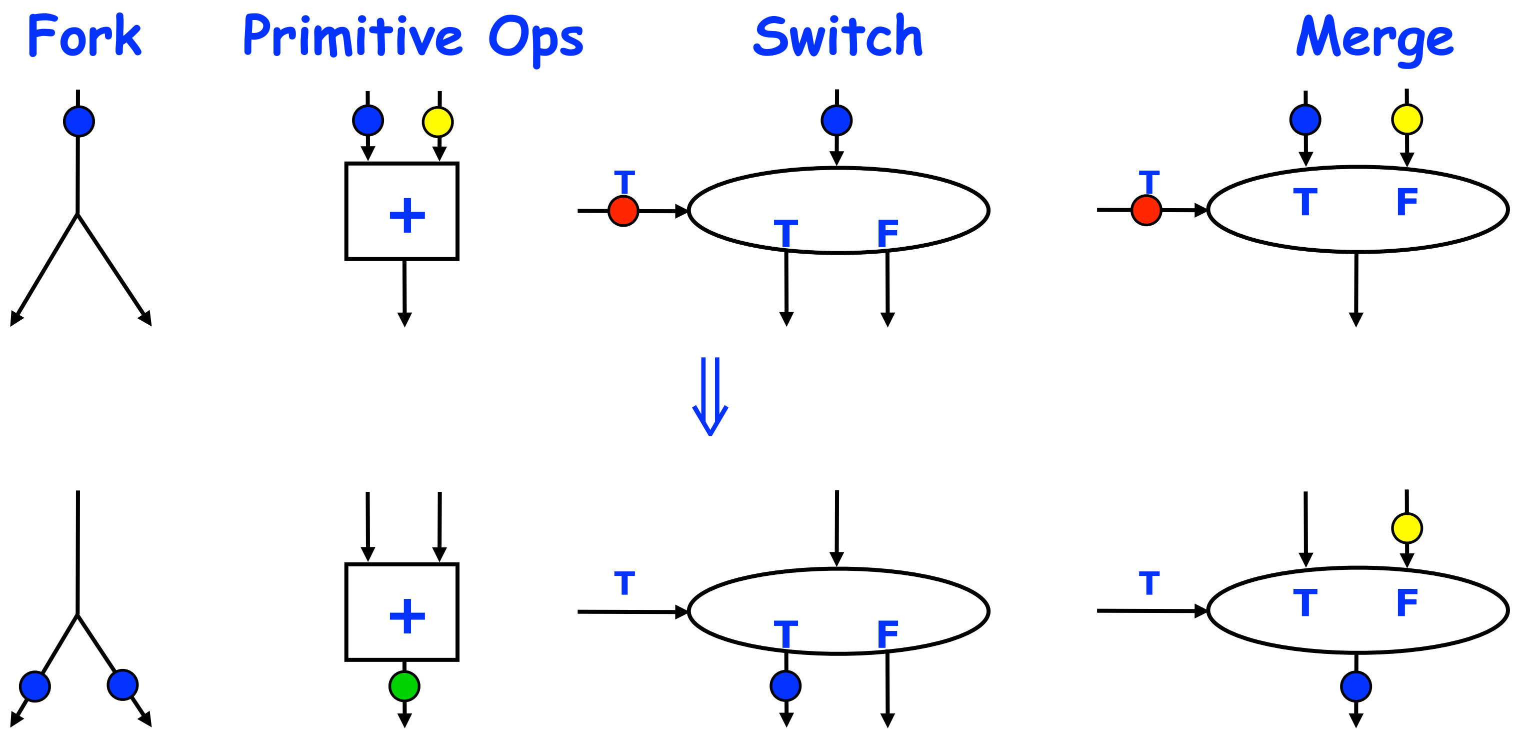
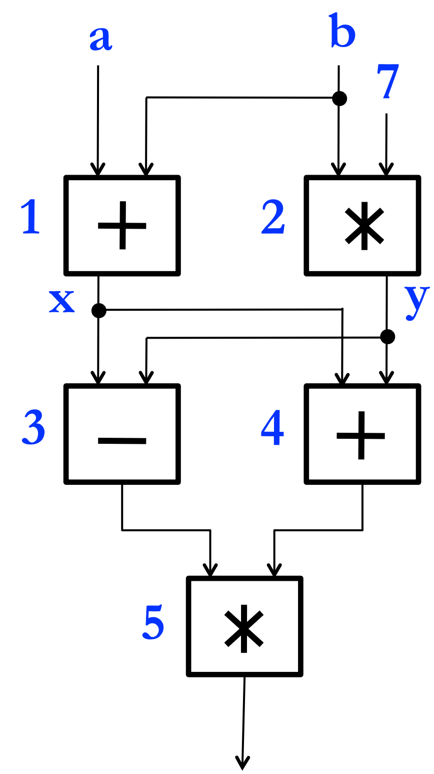


Figure 1: Example instruction sequence and its dataflow graph

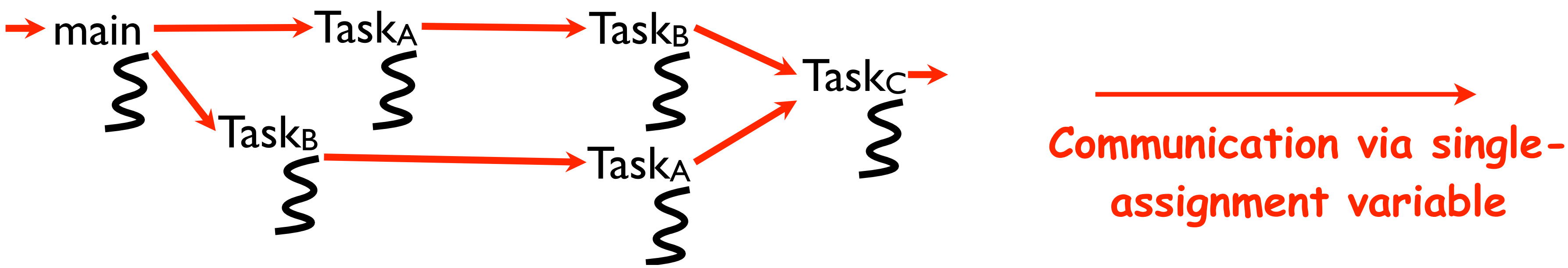
```
x = a + b;
y = b * 7;
z = (x-y) * (x+y);
```



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

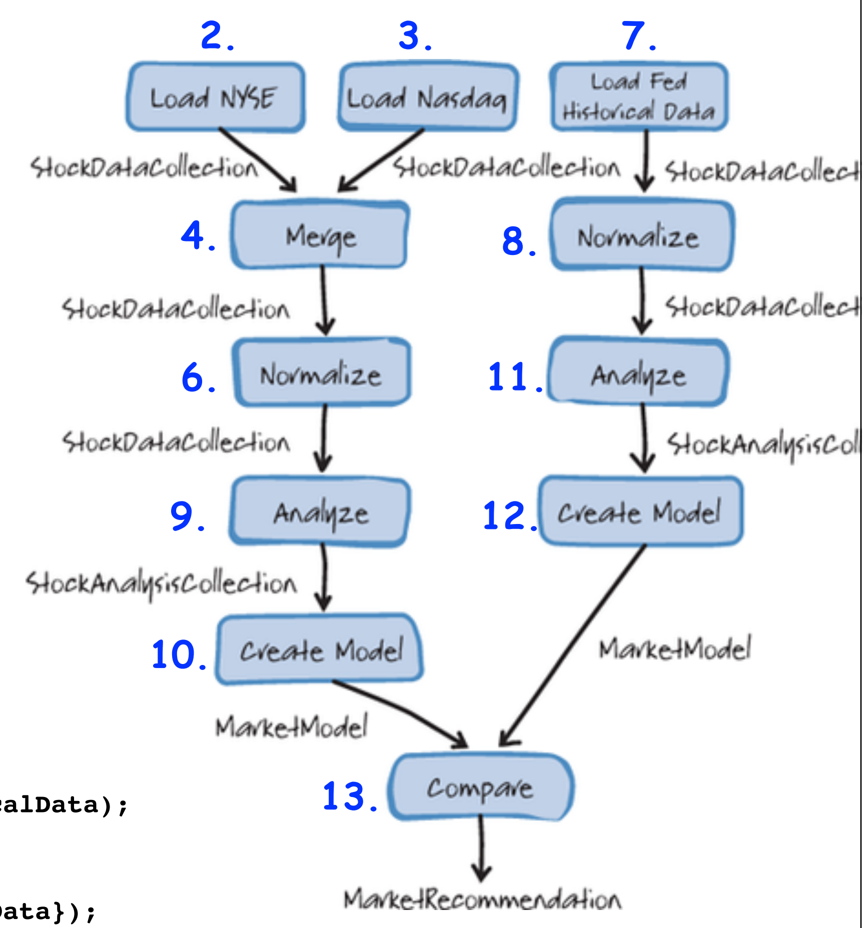
Productivity Benefits of Macro-Dataflow Programming



- "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();
```

- Allocate an instance of a data-driven-future object (container)

```
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) ;
```

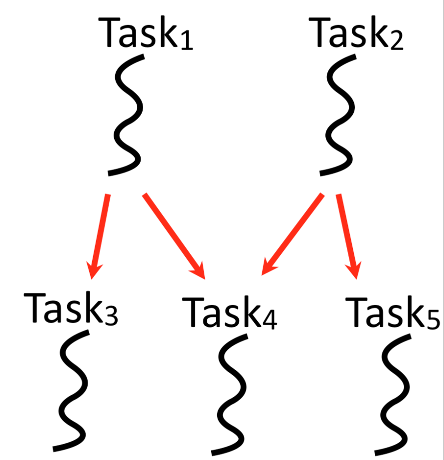
- Store object V in ddfA, thereby making ddfA available
- Single-assignment rule: at most one put is permitted on a given DDF

```
ddfA.get()
```

- Return value 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

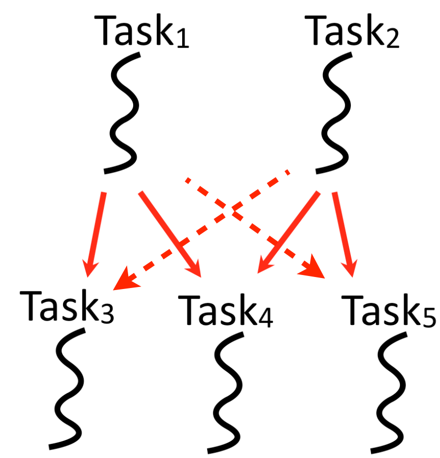
```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await(left) leftReader(left); // Task3
  async await(right) rightReader(right); // Task5
  async await(left,right)
        bothReader(left,right); // Task4
  async left.put(leftWriter()); // Task1
  async right.put(rightWriter());// Task2
}
```



- **await** clauses capture data flow relationships

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

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

- Case 2: 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.

"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!

Wavefront techniques: use case for DDFs and DDTs

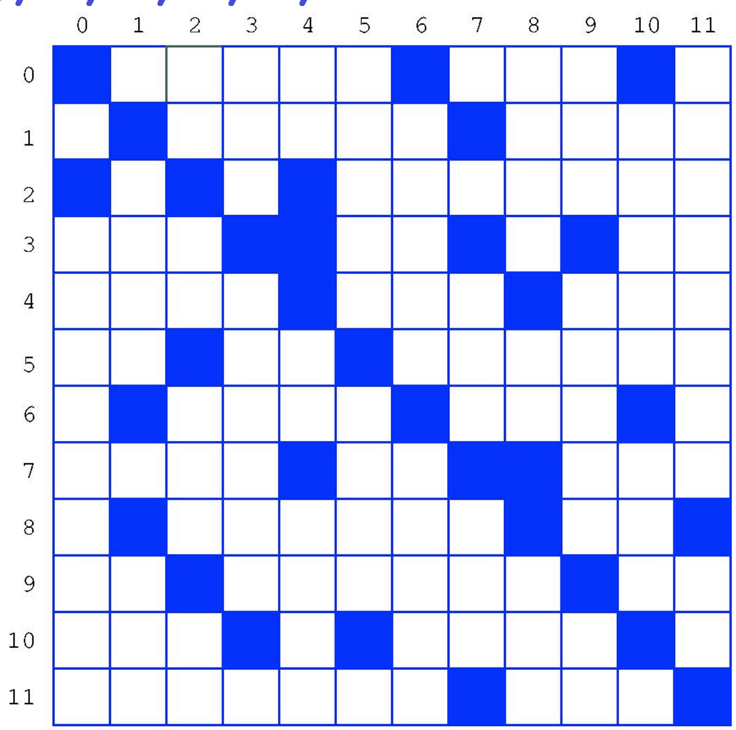
- An approach to parallel programming, e.g. Joubert et al '98
  - Divide a computation into many tasks
  - Form a schedule for these tasks

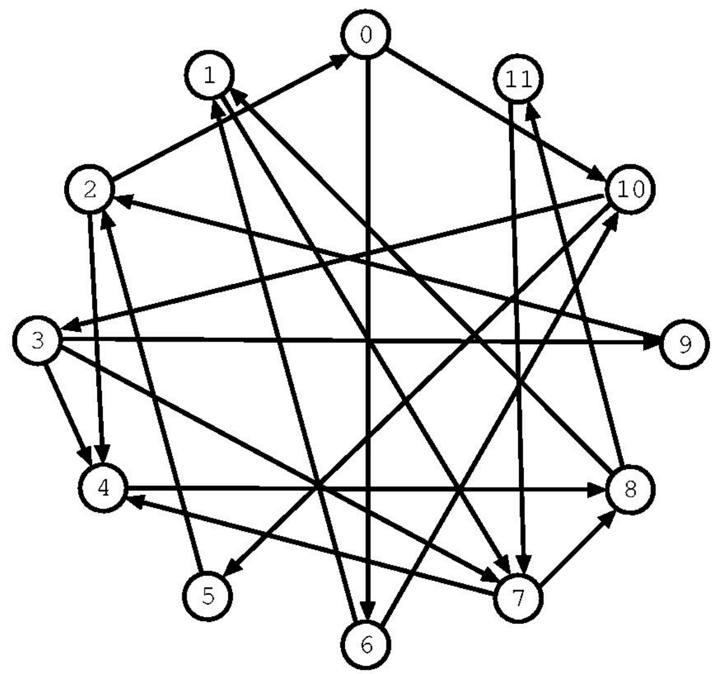
- A schedule contains several wavefronts
  - Each wavefront comprises tasks that are algorithmically independent of each other
  - i.e. correctness is not affected by the order of execution

- The execution is carried out from one wavefront to another
  - Tasks assigned according to the dependency structure
  - Each wavefront contains tasks that can be executed in parallel

Adjacency graph for Gauss Seidel Algorithm

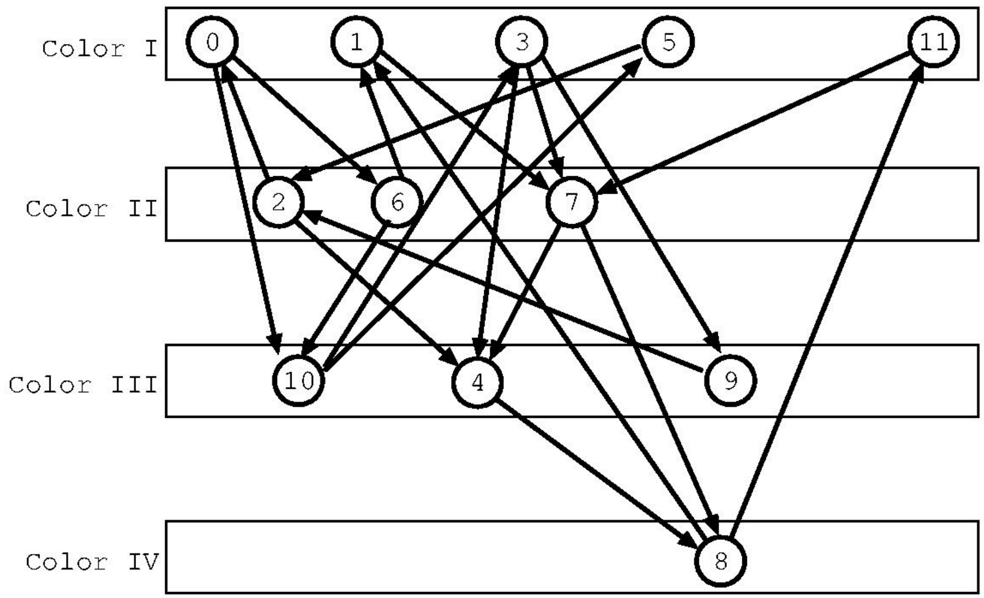
- i-->j edge in adjacency graph if (i,j) entry in matrix has non-zero entry (adjacency graph is not a computation dependence graph)
- Algorithm can compute two rows in parallel if they are not connected by an edge in the adjacency graph i.e., if they form an independent set e.g., 0, 1, 3, 5, 11





Generating a Wavefront Schedule

- By coloring the adjacency graph



- Can generate a schedule to let the computation perform from one color (stage) to another
- Schedule can be implemented using DDFs and DDTs

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(); };
```

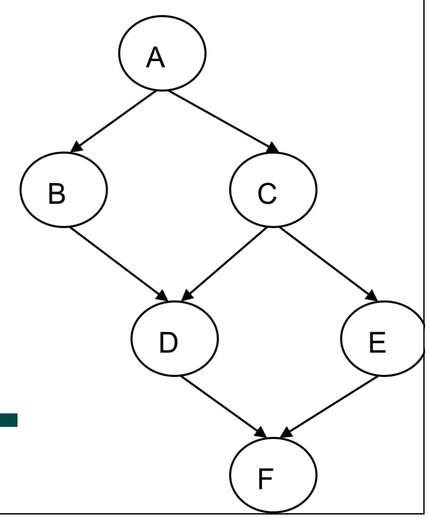
- Future version

```
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(); };
```

Use of DDFs with dummy objects (like future<void>)

```
finish {
  DataDrivenFuture ddfA = new DataDrivenFuture();
  DataDrivenFuture ddfB = new DataDrivenFuture();
  DataDrivenFuture ddfC = new DataDrivenFuture();
  DataDrivenFuture ddfD = new DataDrivenFuture();
  DataDrivenFuture ddfE = new DataDrivenFuture();
  async { ... ; ddfA.put(""); } // Task A
  async await(ddfA) { ... ;  ddfB.put(""); } // Task B
  async await(ddfA) { ... ;  ddfC.put(""); } // Task C
  async await(ddfB,ddfC) { ... ;  ddfD.put(""); } // Task D
  async await(ddfC) { ... ;  ddfE.put(""); } // Task E
  async await(ddfD,ddfE) { ... } // Task F
} // finish
```

- This example uses an empty string as a dummy object



Homework 2 Reminder

- Programming assignment, due Monday, Jan 30th
- Post questions on Piazza (preferred), or send email to comp322-staff at mailman.rice.edu
- You should plan to use turn-in script for HW2 submission
  - Contact teaching staff if you cannot access turn-in by following the instructions for Lab 1
- See course web site for penalties for late submissions