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
  — www.gridpp.ac.uk/gridpp14/mesc.ppt

Goals for Today’s Lecture

• Recap of Future Tasks from Lecture 7

• Dataflow Computing, Data-Driven Futures (DDFs) and Data-Driven Tasks (DDTs)
HJ Futures: Tasks with Return Values (Recap)

```java
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

```java
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**
  1. // Parent task creates child async task
  2. future<int> container = async<int> { return computeSum(...); };
  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
     - Types future<T> vs T
  2) Synchronization to avoid race condition in container accesses
     - get() operation blocks until value becomes available

Parent Task

```
container = async {...}
...
container.get()
```

Child Task

```
computeSum(...)
return ...
```

container \rightarrow return value
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

1. // NOTE: return statement is optional
2. // when return type is void
3. final future<void> A = async<void>
4.     { . . . };
5. final future<void> B = async<void>
6.     { A.get(); . . . };
7. final future<void> C = async<void>
8.     { A.get(); . . . };
9. final future<void> D = async<void>
10.    { B.get(); C.get(); . . . };
11. final future<void> E = async<void>
12.    { C.get(); . . . };
13. final future<void> F = async<void>
14.    { 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
An operator executes when all its input values are present; copies of the result value are distributed to the destination operators.
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

```java
1. public MarketRecommendation DoAnalysisSequential() {
2.     StockDataCollection nyseData = LoadNyseData();
3.     StockDataCollection nasdaqData = LoadNasdaqData();
4.     StockDataCollection mergedMarketData =
5.         MergeMarketData(new[]{nyseData, nasdaqData});
6.     StockDataCollection normalizedMarketData =
7.         NormalizeData(mergedMarketData);
8.     StockDataCollection fedHistoricalData =
9.         LoadFedHistoricalData();
10.    StockDataCollection normalizedHistoricalData =
11.       NormalizeData(fedHistoricalData);
12.    StockAnalysisCollection analyzedStockData =
13.       AnalyzeData(normalizedMarketData);
14.    MarketModel modeledMarketData = RunModel(analyzedStockData);
15.    StockAnalysisCollection analyzedHistoricalData =
16.       AnalyzeData(normalizedHistoricalData);
17.    MarketModel modeledHistoricalData = RunModel(analyzedHistoricalData);
18.    MarketRecommendation recommendation =
19.        CompareModels(new[]{modeledMarketData, modeledHistoricalData});
20.    return recommendation;
21.}
```

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

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
Finish-async version of the same example has more dependences

1. // Assume that left and right are fields in this object
2. finish {
3.   \texttt{async left = put(leftWriter()); } \quad \textit{Task1}
4.   \texttt{async right = put(rightWriter()); } \quad \textit{Task2}
5. }
6. finish {
7.   \texttt{async leftReader(left); } \quad \textit{Task3}
8.   \texttt{async rightReader(right); } \quad \textit{Task5}
9.   \texttt{async bothReader(left, right); } \quad \textit{Task4}
10.}
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

```java
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

- \( i \rightarrow j \) 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, whereas 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, whereas 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
  ```java
  final future<int> f = async<int> { return g(); };
  ...
  ... = f.get();
  ```

• DDF version
  ```java
  DataDrivenFuture f = new DataDrivenFuture();
  async { f.put(g()) };
  ...
  finish async await (f) { ... = f.get(); }
  ```
Implementing DDFs/DDTs using Future tasks

- **DDF version**

  ```java
  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**

  ```java
  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>)

1. finish {
2.   DataDrivenFuture ddfA = new DataDrivenFuture();
3.   DataDrivenFuture ddfB = new DataDrivenFuture();
4.   DataDrivenFuture ddfC = new DataDrivenFuture();
5.   DataDrivenFuture ddfD = new DataDrivenFuture();
6.   DataDrivenFuture ddfE = new DataDrivenFuture();
7.   async { ... ; ddfA.put(""); } // Task A
8.   async await(ddfA) { ... ; ddfB.put(""); } // Task B
9.   async await(ddfA) { ... ; ddfC.put(""); } // Task C
10.  async await(ddfB, ddfC) { ... ; ddfD.put(""); } // Task D
11.  async await(ddfC) { ... ; ddfE.put(""); } // Task E
12.  async await(ddfD, ddfE) { ... } // Task F
13. } // 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