COMP 322: Fundamentals of Parallel Programming

Lecture 8: Finish, Async, Computation Graphs

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Homework #1 Hints

- **sorted** operation on streams results in ascending order. To sort in descending order, use `sorted(Comparator.reverseOrder())`.

- **groupingBy**, convert elements of stream into type you want by passing `Collectors.mapping(map-function, downstream-collector)` as an additional argument. For parallel streams use `groupingByConcurrent`.

Create a mapping between customer IDs and their order IDs whose status is "PENDING"
```java
orderRepo.findAll().stream()
    .filter(order -> order.getStatus().equals("PENDING"))
    .collect(Collectors.groupingBy(order -> order.getCustomer().getId(),
                                   Collectors.mapping(Order::getCustomerId,
                                                        Collectors.toSet())));
```

Acknowledgement: Chase Hartsell
Async and Finish Statements for Task Creation and Termination

async S

- Creates a new child task that executes statement S

// T₀ (Parent task)
STMT0;
finish { //Begin finish
  async {
    STMT1; //T₁ (Child task)
  }
  STMT2; //Continue in T₀
}
//End finish (wait for T₁)
STMT3; //Continue in T₀

finish S

- Execute S, but wait until all asyncs in S’s scope have terminated.

// T₀ (Parent task)
STMT0

---

T₁

fork

STMT1

join

T₀

STMT0

---

STMT2

STMT3


Example of a Sequential Program: Computing sum of array elements

Algorithm 1: Sequential ArraySum

Input: Array of numbers, X.
Output: \( sum = \text{sum of elements in array } X \).

\[
\begin{align*}
\text{sum} & \leftarrow 0; \\
\text{for } i & \leftarrow 0 \text{ to } X.\text{length} - 1 \text{ do} \\
& \quad \text{sum} \leftarrow \text{sum} + X[i]; \\
\text{return } \text{sum};
\end{align*}
\]
Parallelization Strategy for 2 cores (Two-way Parallel Array Sum)

Basic idea:
- Decompose problem into two tasks for partial sums
- Combine results to obtain final answer
- Parallel divide-and-conquer pattern
Two-way Parallel Array Sum using async & finish constructs

Algorithm 2: Two-way Parallel ArraySum

Input: Array of numbers, $X$.
Output: $\text{sum} = \text{sum of elements in array } X$.

// Start of Task T1 (main program)
$\text{sum1} \leftarrow 0$; $\text{sum2} \leftarrow 0$;

// Compute sum1 (lower half) and sum2 (upper half) in parallel.
finish {
    async {
        // Task T2
        for $i \leftarrow 0$ to $X\.length/2 - 1$ do
            $\text{sum1} \leftarrow \text{sum1} + X[i]$;
    };

    async {
        // Task T3
        for $i \leftarrow X\.length/2$ to $X\.length - 1$ do
            $\text{sum2} \leftarrow \text{sum2} + X[i]$;
    };
}

// Task T1 waits for Tasks T2 and T3 to complete
// Continuation of Task T1
$\text{sum} \leftarrow \text{sum1} + \text{sum2}$;
return $\text{sum}$;
Two-way Parallel Array Sum using async & finish constructs

Algorithm 2: Two-way Parallel ArraySum

**Input:** Array of numbers, $X$.

**Output:** $sum = \text{sum of elements in array } X$.

// Start of Task T1 (main program)

$sum1 \leftarrow 0; \ \ \ sum2 \leftarrow 0;$

// Compute sum1 (lower half) and sum2 (upper half) in parallel.

finish{

async{

// Task T2

for $i \leftarrow 0$ to $X.length/2 - 1$ do

$sum1 \leftarrow sum1 + X[i];$

};

async{

// Task T3

for $i \leftarrow X.length/2$ to $X.length - 1$ do

$sum2 \leftarrow sum2 + X[i];$

};

...more work...

$sum \leftarrow sum1 + sum2;$

return $sum;$


...more work...

$sum \leftarrow sum1 + sum2;$

return $sum;$
Two-way Parallel Array Sum using futures

// Parent Task T1 (main program)
// Compute sum1 (lower half) & sum2 (upper half) in parallel
var sum1 = future(() => { // Future Task T2
    int sum = 0;
    for (int i = 0; i < X.length / 2; i++) sum += X[i];
    return sum;
});
var sum2 = future(() => { // Future Task T3
    int sum = 0;
    for (int i = X.length / 2; i < X.length; i++) sum += X[i];
    return sum;
});
...more work...
int total = sum1.get() + sum2.get();
Computation Graphs

- A Computation Graph (CG) captures the dynamic execution of a parallel program, for a specific input
- CG nodes are “steps” in the program’s execution
  - A step is a sequential subcomputation without any spawned, begin-finish or end-finish operations
- CG edges represent ordering constraints
  - “Continue” edges define sequencing of steps within a task
  - “Spawn” edges connect parent tasks to child spawned tasks
  - “Join” edges connect the end of each spawned task to its IEF’s end-must finish operations
- All computation graphs must be acyclic
  - It is not possible for a node to depend on itself
- Computation graphs are examples of “directed acyclic graphs” (DAGs)
Which statements can potentially be executed in parallel with each other?

1. `finish { // F1`
2. `async { A; }`
3. `finish { // F2`
4. `async { B1; }`
5. `async { B2; }`
6. `} // F2`
7. `B3;`
8. `} // F1`

**Key idea**: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.
Parallelize Tasks

Assume you have 2 washers and 2 dryers. Assume there’s 0 cost to spawn a task.

Place “finish” and “async” blocks around the following tasks:

1. Run load 1 in washer (LW1)
2. Run load 2 in washer (LW2)
3. Run load 1 in dryer (LD1)
4. Run load 2 in dryer (LD2)
Assume you have 2 washers and 2 dryers. Assume there’s 0 cost to spawn a task.

Place “finish” and “spawn” blocks around the following tasks:

1. finish { // F1
2. async { Run load 1 in washer (LW1) }
3. async { Run load 2 in washer (LW2) }
4.} // F1
5. async { Run load 1 in dryer (LD1) }
6. async { Run load 2 in dryer (LD2) }
Assume you have 2 washers and 2 dryers. Assume there’s 0 cost to spawn a task.

Place “finish” and “spawn” blocks around the following tasks:

1. finish { // F1
2.     async { Run load 1 in washer (LW1); Run load 1 in dryer (LD1) }
3.     async { Run load 2 in washer (LW2); Run load 2 in dryer (LD2) }
4. } // F1
Draw Computation Graph for Solution
Key idea: If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.
Key idea: If two statements, X and Y, have no path of directed edges from one to the other, then they can run in parallel with each other.
1. finish { // F1
2. async LW1;
3. async LW2;
4.} // F1
5. async LD1;
6. async LD2;

Key idea: If two statements, X and Y, have no path of directed edges from one to the other, then they can run in parallel with each other.
1. `finish { // F1`
2. `async { LW1; LD1 }`
3. `async { LW2; LD2 }`
4. `} // F1`

**Key idea:** If two statements, X and Y, have *no path of directed edges* from one to the other, then they can run in parallel with each other.
Announcements & Reminders

• IMPORTANT:
  — Watch videos for topics 1.3, 4.5 for next lecture
• HW 1 is due on Wednesday, Feb 1st
• Quiz 2 is due on Monday, Feb 6th
• Module 1 handout is available
• See course web site for syllabus, work assignments, due dates, …
  • http://comp322.rice.edu