Lecture 12: Scheduling

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu
Computation Graphs

Structured Parallelism (Finish/async)

Futures and Future Tasks

Promises and Data-Driven Tasks
Computation Graphs

- Structured parallelism (finish/async):
  Create structured graphs (similar to what structured programming can create)
  No high-level data representation: have to share data
  Fast implementation, easy to synchronize large # of tasks

- Futures and future tasks:
  Easy to construct unstructured, arbitrary graphs
  Elegant, functional high-level data representation: futures
  Functional, “push” model: “where is the data going to, create futures for those”
  Large overhead when handling large # of tasks

- Promises and data-driven tasks:
  Easy to construct unstructured, arbitrary graphs with unknown task-promise association
  Data-driven, “pull” model: “what data does this DDT depend on, create promises for those”
  Can have a faster implementation than futures
  Large overhead when handling large # of tasks
Ordering Constraints and Transitive Edges in a Computation Graph

• The primary purpose of a computation graph is to determine if an ordering constraint exists between two steps (nodes).
  —Observation: Node A must be performed before node B if there is a path of directed edges from A and B.

• An edge, X → Y, in a computation graph is said to be transitive if there exists a path of directed edges from X to Y that does not include the X → Y edge.
  —Observation: Adding or removing a transitive edge does not change the ordering constraints in a computation graph.
Ideal Parallelism (Recap)

• Define *ideal parallelism* of Computation G Graph as the ratio, \( \text{WORK}(G)/\text{CPL}(G) \)

• Ideal Parallelism only depends on the computation graph, and is the speedup that you can obtain with an unbounded number of processors

**Example:**

\[
\begin{align*}
\text{WORK}(G) &= 26 \\
\text{CPL}(G) &= 11 \\
\text{Ideal Parallelism} &= \frac{\text{WORK}(G)}{\text{CPL}(G)} = \frac{26}{11} \approx 2.36
\end{align*}
\]
What is the critical path length of this parallel computation?

1. `finish (())` → `{`  // F1
2. `async (())` → `A`;  // Boil water & pasta (10)
3. `finish (())` → `{`  // F2
4. `async (())` → `B1`;  // Chop veggies (5)
5. `async (())` → `B2`;  // Brown meat (10)
6. `});`  // F2
7. `B3;`  // Make pasta sauce (5)
8. `});`  // F1
Scheduling of a Computation Graph on a fixed number of processors

Node label = time(N), for all nodes N in the graph

NOTE: this schedule achieved a completion time of 11. Can we do better?
Scheduling of a Computation Graph on a fixed number of processors

• Assume that node N takes \( \text{TIME}(N) \) regardless of which processor it executes on, and that there is no overhead for creating parallel tasks.

• A schedule specifies the following for each node:
  - \( \text{START}(N) \) = start time
  - \( \text{PROC}(N) \) = index of processor in range \( 1 \ldots P \)

  such that
  - \( \text{START}(i) + \text{TIME}(i) \leq \text{START}(j) \), for all CG edges from \( i \) to \( j \) (Precedence constraint)
  - A node occupies consecutive time slots in a processor (Non-preemption constraint)
  - All nodes assigned to the same processor occupy distinct time slots (Resource constraint)
Greedy Schedule

• A greedy schedule is one that never forces a processor to be idle when one or more nodes are ready for execution

• A node is **ready** for execution if all its predecessors have been executed

• Observations
  – \( T_1 = \text{WORK}(G) \), for all greedy schedules
  – \( T_\infty = \text{CPL}(G) \), for all greedy schedules

• \( T_P(S) \) = execution time of schedule \( S \) for computation graph \( G \) on \( P \) processors
Lower Bounds on Execution Time of Schedules

- Let $T_p$ = execution time of a schedule for computation graph $G$ on $P$ processors
  - $T_p$ can be different for different schedules, for same values of $G$ and $P$

- Lower bounds for all greedy schedules
  - Capacity bound: $T_p \geq \text{WORK}(G)/P$
  - Critical path bound: $T_p \geq \text{CPL}(G)$

- Putting them together
  - $T_p \geq \max(\text{WORK}(G)/P, \text{CPL}(G))$
Upper Bound on Execution Time of Greedy Schedules

Theorem [Graham ’66]. Any greedy scheduler achieves

$$T_P \leq \frac{\text{WORK}(G)}{P} + \text{CPL}(G)$$

Proof sketch:
Define a time step to be complete if $P$ processors are scheduled at that time, or incomplete otherwise

# complete time steps $\leq \frac{\text{WORK}(G)}{P}$

# incomplete time steps $\leq \text{CPL}(G)$
Combine lower and upper bounds to get
\[ \max(\text{WORK}(G)/P, \text{CPL}(G)) \leq T_P \leq \text{WORK}(G)/P + \text{CPL}(G) \]

**Corollary:** Any greedy scheduler achieves execution time \( T_P \) that is within a factor of 2 of the optimal time (since \( \max(a,b) \) and \( (a+b) \) are within a factor of 2 of each other, for any \( a \geq 0, b \geq 0 \) ).

**Corollary 2:** Lower and upper bounds approach the same value whenever:
- There’s lots of parallelism, \( \text{WORK}(G)/\text{CPL}(G) \gg P \)
- Or there’s little parallelism, \( \text{WORK}(G)/\text{CPL}(G) \ll P \)
Abstract Performance Metrics

- **Basic Idea**
  - Count operations of interest, as in big-O analysis, to evaluate parallel algorithms
  - Abstraction ignores many overheads that occur on real systems
- **Calls to doWork()**
  - Programmer inserts calls of the form, `doWork(N)` within a task (async, future task or data-driven task) to indicate abstract execution of N application-specific abstract operation
    - e.g., in lab 4, we included one call to `doWork(1)` for each double addition, and ignore the cost of everything else
  - Abstract metrics are enabled by calling `HjSystemProperty.abstractMetrics.set(true)` at start of program execution
  - If an HJ program is executed with this option, abstract metrics can be printed at end of program execution with calls to `abstractMetrics().totalWork()`, `abstractMetrics().criticalPathLength()`, and `abstractMetrics().idealParallelism()`
Abstract Performance Metrics

- Pay attention where you put doWork() calls

- What does this mean?

```javascript
var bottom = future(() => ...);
var top = future(() => ...)
doWork(1);
return bottom.get() + top.get();
```

- Correct:

```javascript
var bottom = future(() => ...);
var top = future(() => ...);

var bottomVal = bottom.get();
var topVal = top.get();
doWork(1);
return bottomVal + topVal;
```