Lecture 11: Scheduling

Mack Joyner
mjoyner@rice.edu

http://comp322.rice.edu
Homework 2: GitHub Contributors

GitHub Username: mrjoyner
Password/Token: *******************************
Organization: trabian

Load contributors

<table>
<thead>
<tr>
<th>Login</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
GitHub Contributors

GitHub Username: mrjoyner
Password/Token: *******************************
Organization: trabian

Load contributors

Login | Contributions
GUI Events with Java Swing

• Swing enables you to build a GUI in Java and respond to user events
• Containers (e.g. JFrame)
• Components
  — JButton
  — JLabel
  — JTextField
• Users interact with the GUI and trigger actions (events)
• ActionListeners are setup for a component to respond to the event
GitHub Contributors

<table>
<thead>
<tr>
<th>GitHub Username</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>mrjoyner</td>
<td></td>
</tr>
<tr>
<td>dhh</td>
<td>3393</td>
</tr>
<tr>
<td>jeremy</td>
<td>3352</td>
</tr>
<tr>
<td>mpdehaan</td>
<td>3321</td>
</tr>
<tr>
<td>josevalim</td>
<td>2501</td>
</tr>
<tr>
<td>tenderlove</td>
<td>2268</td>
</tr>
<tr>
<td>billdawson</td>
<td>1823</td>
</tr>
<tr>
<td>fxn</td>
<td>1600</td>
</tr>
<tr>
<td>marshall</td>
<td>1498</td>
</tr>
<tr>
<td>vishalduggal</td>
<td>1494</td>
</tr>
<tr>
<td>tehkinwatt</td>
<td>1422</td>
</tr>
</tbody>
</table>
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, $X \rightarrow Y$, 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 $X \rightarrow Y$ edge.
  - Observation: Adding or removing a transitive edge does not change the ordering constraints in a computation graph.
Observations:
- Any node with out-degree > 1 must be an async (must have an outgoing spawn edge)
- Any node with in-degree > 1 must be an end-finish (must have an incoming join edge)
- Adding or removing transitive edges does not impact ordering constraints

1. A();
2. finish { // F1
3. async D();
4. B();
5. E();
6. finish { // F2
7. async H();
8. F();
9. } // F2
10. G();
11. } // F1
12. C();
Ideal Parallelism (Recap)

- Define **ideal parallelism** of Computation G Graph as the ratio, WORK(G)/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:
WORK(G) = 26
CPL(G) = 11
Ideal Parallelism = WORK(G)/CPL(G) = 26/11 ~ 2.36
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

Step A

Step B1

Step B2

Step B3
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 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) = \text{start time}
  — \text{PROC}(N) = \text{index of processor in range } 1 \ldots P

such that
  — \text{START}(i) + \text{TIME}(i) \leq \text{START}(j), \text{ for all CG edges from } i \text{ to } j \ (\text{Precedence constraint})
  — A node occupies consecutive time slots in a processor \ (\text{Non-preemption constraint})
  — All nodes assigned to the same processor occupy distinct time slots \ (\text{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 \textit{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 $T_P = \text{execution time of a schedule for computation graph } G \text{ on } P \text{ processors}$

- $T_P$ can be different for different schedules, for same values of $G$ and $P$

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

- Putting them together
  - $T_P \geq \max(WORK(G)/P, CPL(G))$
Theorem [Graham ‘66]. Any greedy scheduler achieves

\[ T_P \leq \text{WORK}(G)/P + \text{CPL}(G) \]

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

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

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

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, \( \text{WORK}(G)/\text{CPL}(G) \ll P \)