Announcements

- Coursera forum on HJ Environment and Setup Issues
  - Please post your issues, and also respond to postings by other students when you can help

- Instructor’s office hours are during 2pm - 3pm on MWF
  - Please stop by if you have problems with any of the following
    - Accessing the Module 1 handout
    - Using the turnin script
    - You did not receive any email sent to comp322-all

- Homework 1 has been posted
  - Contains written and programming components
  - Due by 5pm on Wednesday, Jan 23rd
  - Must be submitted using “turnin” script introduced in Lab 1
    - In case of problems, email a zip file to comp322-staff at mailman.rice.edu before the deadline
  - See course web site for penalties for late submissions
Complexity Measures for Computation Graphs (Recap)

Define

- **TIME(N)** = execution time of node N
- **WORK(G)** = sum of **TIME(N)**, for all nodes N in CG G
  - **WORK(G)** is the total work to be performed in G
- **CPL(G)** = length of a longest path in CG G, when adding up execution times of all nodes in the path
  - Such paths are called critical paths
  - **CPL(G)** is the length of these paths (critical path length)
  - **CPL(G)** is also the smallest possible execution time for the computation graph
Ideal Parallelism (Recap)

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

Ideal Parallelism is independent of the number of processors that the program executes on, and only depends on the computation graph.
Solution to Worksheet #2: what is the critical path length and ideal parallelism of this graph?

\[ CPL(G) = \text{length of a longest path in computation graph } G \]

- time(N) is labeled for all nodes N in the graph

\[
\begin{align*}
\text{WORK}(G) &= 26 \\
\text{CPL}(G) &= 11 \\
\text{Ideal Parallelism} &= \frac{\text{WORK}(G)}{\text{CPL}(G)} \\
&= \frac{26}{11} \approx 2.36
\end{align*}
\]
Scheduling of a Computation Graph on a fixed number of processors: Example

Start time | Proc 1 | Proc 2 | Proc 3
---|---|---|---
0 | A | | |
1 | B | | |
2 | C | N | |
3 | D | N | I
4 | D | N | J
5 | D | N | K
6 | D | Q | L
7 | E | R | M
8 | F | R | O
9 | G | R | P
10 | H | | |
11 | | | |

Diagram of the computation graph with node labels and edge directions.
Scheduling of a Computation Graph on a fixed number of processors, $P$

- 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:
  - $\text{START}(N) = \text{start time}$
  - $\text{PROC}(N) = \text{index of processor in range } 1 \ldots P$

such that:

  - $\text{START}(i) + \text{TIME}(i) \leq \text{START}(j)$, 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)
Lower Bounds on Execution Time of Schedules

- Let $T_p$ = execution time of a schedule for computation graph $G$ on $P$ processors
  - Can be different for different schedules
- 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
  - $T_p \geq \max(\text{WORK}(G)/P, \text{CPL}(G))$
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
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 \textit{complete} if \( \geq P \) nodes are ready at that time, or \textit{incomplete} otherwise

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

\[ \# \text{ incomplete time steps} \leq \text{CPL}(G) \]
What are the best-case and worst-case schedules that we can obtain for this example on 2 processors?

- WORK(G) = 24
- CPL(G) = 12
- For P=2, WORK(G)/P = 12
- Lower bound = max(12,12) = 12
- Upper bound = 12 + 12 = 24
- Best (13) and worst (14) values for $T_2$ are in the range, 12 ... 24
Bounding the performance of Greedy Schedulers

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

**Corollary 1:** 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, \( \frac{\text{WORK}(G)}{\text{CPL}(G)} \gg P \)
- Or there’s little parallelism, \( \frac{\text{WORK}(G)}{\text{CPL}(G)} \ll P \)
Strong Scaling and Speedup

- Define \( \text{Speedup}(P) = \frac{T_1}{T_P} \)
  - Factor by which the use of \( P \) processors speeds up execution time relative to 1 processor, for a fixed input size
  - For ideal executions without overhead, \( 1 \leq \text{Speedup}(P) \leq P \)
  - Linear speedup
    - When \( \text{Speedup}(P) = kP \), for some constant \( k \), \( 0 < k < 1 \)
- Referred to as “strong scaling” because input size is fixed
Reduction Tree Schema for computing Array Sum in parallel

Assume input array size = S, and each add takes 1 unit of time:

- WORK(G) = S - 1
- CPL(G) = \log_2(S)
- Assume \( T_p = \frac{\text{WORK}(G)}{P} + \text{CPL}(G) = \frac{(S - 1)}{P} + \log_2(S) \)
  - Within a factor of 2 of any schedule’s execution time

What is the speedup for this parallel algorithm? Time for worksheet #3!
Algorithm based on updates to array

Observations:

- This algorithm overwrites $X$ (make a copy if $X$ is needed later)
- $stride =$ distance between array subscript inputs for each addition
- $size =$ number of additions that can be executed in parallel in each level (stage)
Async-Finish Parallel Program for Array Sum
(for X.length = 8)

1. `finish` {  //STAGE 1: stride = 1, size = 4 parallel additions
2.   `async` X[0] += X[1]; `async` X[2] += X[3];
4. }
5. `finish` {  //STAGE 2: stride = 2, size = 2 parallel additions
7. }
8. `finish` {  //STAGE 3: stride = 4, size = 1 parallel additions
9.   `async` X[0] += X[4];
10. }
11. // Final sum is now in X[0]
Generalization to arbitrary sized arrays (ArraySum1)

1. for ( int stride = 1; stride < X.length; stride *= 2 ) {
2.   // Compute size = number of adds to be performed in stride
3.   int size=ceilDiv(X.length,2*stride);
4.   finish for(int i = 0; i < size; i++)
5.     async {
6.       if ( (2*i+1)*stride < X.length )
7.         X[2*i*stride] += X[(2*i+1)*stride];
8.     } // finish-for-async
9. } // for
10.
11.// Divide x by y, and round up to next largest int
12.static int ceilDiv(int x, int y) { return (x+y-1) / y; }
Computation Graph for ArraySum1

Stmnt 1

X[0] += X[1]


Stmnt 3

X[0] += X[2]

Stmnt 5


End-Finish (main)

X[0] += X[4]

STAGE 1

STAGE 2

STAGE 3

Continue edge

Spawn edge

Join edge
HJ Abstract Performance Metrics

• Basic Idea
  — Count operations of interest, as in big-O analysis
  — Abstraction ignores overheads that occur on real systems

• Calls to perf.doWork()
  — Programmer inserts calls of the form, perf.doWork(N), within a step to indicate abstraction execution of N application-specific abstract operations
    - e.g., adds, compares, stencil ops, data structure ops
  — Multiple calls add to the execution time of the step

• Enabled by selecting “Show Abstract Execution Metrics” in DrHJ compiler options (or -perf=true runtime option)
  — If an HJ program is executed with this option, abstract metrics are printed at end of program execution with WORK(G), CPL(G), Ideal Speedup = WORK(G) / CPL(G)
Inserting call to perf.doWork() in ArraySum1

1. for ( int stride = 1; stride < X.length ; stride *= 2 ) { 
2.   // Compute size = number of adds to be performed in stride 
3.   int size=ceilDiv(X.length,2*stride);
4.   finish for(int i = 0; i < size; i++)
5.     async {
6.       if ( (2*i+1)*stride < X.length ) {
7.         perf.doWork(1);
8.         X[2*i*stride] += X[(2*i+1)*stride];
9.       }
10.     } // finish-for-async
11. } // for
12. 
Worksheet #3: Strong Scaling for Array Sum

Name 1: ___________________          Name 2: ___________________

• Assume $T(S,P) \sim \text{WORK}(G,S)/P + \text{CPL}(G,S) = (S-1)/P + \log_2(S)$ for a parallel array sum computation

• Strong scaling
  — Assume $S = 1024 \Rightarrow \log_2(S) = 10$
  — Compute Speedup($P$) for 10, 100, 1000 processors
    - $T(P) = 1023/P + 10$
    - Speedup(10) = $T(1)/T(10) =$
    - Speedup(100) = $T(1)/T(100) =$
    - Speedup(1000) = $T(1)/T(1000) =$
  — Why is it worse than linear?
Outline of Today’s Lecture

• Computation Graphs (contd)
• Parallel Speedup, Strong Scaling
• Abstract Performance Metrics

• Acknowledgments
  — COMP 322 Module 1 handout, Sections 3.1, 3.2, 3.3