COMP 322: Fundamentals of Parallel Programming

Lecture 3: Computation Graphs, Abstract Performance Metrics, Array Reductions

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Acknowledgments for Today’s Lecture

• Cilk lectures, http://supertech.csail.mit.edu/cilk/
Goals for Today’s Lecture

• Lower and upper bounds for abstract parallel execution time
• Parallel Array sum and Complexity Analysis
• Abstract execution metrics in HJ
Computation Graphs for HJ Programs (Recap)

- A Computation Graph (CG) captures the dynamic execution of an HJ program, for a specific input.
- CG nodes are “steps” in the program’s execution.
  - A step is a sequential subcomputation without any async, begin-finish and end-finish operations.
- CG edges represent ordering constraints.
  - “Continue” edges define sequencing of steps within a task.
  - “Spawn” edges connect parent tasks to child async tasks.
  - “Join” edges connect the end of each async task to its IEF’s end-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).
Lower Bounds on Execution Time

- Let $T_p$ = execution time of computation graph on $P$ processors
  - Assume an idealized machine where node $N$ takes $\text{TIME}(N)$ regardless of which processor it executes on, and that there is no overhead for creating parallel tasks

- Observations
  - $T_1 = \text{WORK}(G)$
  - $T_\infty = \text{CPL}(G)$

- Lower bounds
  - 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 for Greedy Scheduling

Theorem [Graham ’66]. Any “greedy scheduler” achieves
\[ T_p \leq \frac{\text{WORK}(G)}{P} + \text{CPL}(G) \]

• A greedy scheduler 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.
Upper Bound on Execution Time: Greedy-Scheduling Theorem

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 ≥ \( P \) nodes are ready at that time, or incomplete otherwise

\# complete time steps \( \leq \frac{\text{WORK}(G)}{P} \), since each complete step performs \( P \) work.

\# incomplete time steps \( \leq \text{CPL}(G) \), since each incomplete step reduces the span of the unexecuted dag by 1.
Optimality of Greedy Schedulers

Combine lower and upper bounds to get

$$\max(WORK(G)/P, CPL(G)) \leq T_p \leq WORK(G)/P + CPL(G)$$

**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, $WORK(G)/CPL(G) \gg P$
- Or there’s little parallelism, $WORK(G)/CPL(G) \ll P$
Goals for Today’s Lecture

• Lower and upper bounds for abstract parallel execution time
• Parallel Array sum and Complexity Analysis
• Abstract execution metrics in HJ
int sum = 0;
for (int i=0 ; i < X.length ; i++ )
    sum += X[i];

• The original computation graph is sequential
• We studied a 2-task parallel program for this problem
• How can we expose more parallelism?
Reduction Tree Schema for computing Array Sum in parallel

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)
Parallel Program that satisfies dependences in Reduction Tree schema (for X.length = 8)

```java
finish { // STAGE 1: stride = 1, size = 4 parallel additions
    async X[0]+=X[1]; async X[2]+=X[3];
}

finish { // STAGE 2: stride = 2, size = 2 parallel additions
    async X[0]+=X[2]; async X[4]+=X[6];
}

finish { // STAGE 3: stride = 4, size = 1 parallel addition
    async X[0]+=X[4];
}
```
Computation Graph for ArraySum1

STAGE 1
- $X[0] += X[1]$ (Stmt 1)

STAGE 2
- $X[0] += X[2]$ (Stmt 3)

STAGE 3
- $X[0] += X[4]$ (Stmt 5)

End-Finish (main)
Generalization to arbitrary sized arrays (ArraySum1)

```java
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
    // Compute size = number of additions to be performed in stride
    int size = ceilDiv(X.length, 2*stride);

    finish for(int i = 0; i < size; i++)
    async {
        if ( (2*i+1)*stride < X.length )
            X[2*i*stride] += X[(2*i+1)*stride];
    } // finish-for-async
} // for

// Divide x by y, round up to next largest int, and return result
static int ceilDiv(int x, int y) { return (x+y-1) / y; }
```
Complexity Analysis of ArraySum1

- Define $n = X$.length
- Assume that each addition takes 1 unit of time
  - Ignore all other computations since they are related to the addition by some constant
- Total number of additions, $WORK = n - 1 = O(n)$
- Critical path length (number of stages), $CPL = \text{ceiling}(\log_2(n)) = O(\log(n))$
- Ideal parallelism = $\frac{WORK}{CPL} = \frac{O(n)}{O(\log(n))}$
- Consider an execution on $p$ processors
  - Compute partial sums for batches of $n/p$ elements on each processor
  - Use ArraySum1 program to reduce $p$ partial sums to one total sum
  - $CPL$ for this version is $O(n/p + \log(p))$
  - Parallelism for this version is $\frac{O(n)}{O(n/p + \log(p))}$
  - Algorithm is optimal for $p = n / \log(n)$, or fewer, processors - why?
Generalized Array Reductions

- ArraySum1 can easily be adapted to reduce any associative function \( f \)
  \[
  f(x, y) \text{ is said to be associative if } f(a, f(b, c)) = f(f(a, b), c) \text{ for any inputs } a, b, \text{ and } c
  \]

- Sequential reduction of \( X \), an array of objects of type \( T \):
  \[
  T \text{ result} = X[0];
  \]
  \[
  \text{for(int i=1 ; i < X.length ; i++ ) result} = f(\text{result}, X[i]);
  \]

- Generalized reductions have many interesting applications in practice, as you will see when we learn about Google’s Map Reduce framework

- Execution time of \( f() \) could be much larger than an integer add, and justify the use of an async
Generalized Reduction of WordCount

<table>
<thead>
<tr>
<th>Word</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abc&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;def&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;ghi&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;jkl&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;ghi&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;jkl&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;def&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;jkl&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;ghi&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;jkl&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;abc&quot;</td>
<td>4</td>
</tr>
<tr>
<td>&quot;ghi&quot;</td>
<td>1</td>
</tr>
<tr>
<td>&quot;jkl&quot;</td>
<td>2</td>
</tr>
</tbody>
</table>

Legend:
- X[0]: "abc" 3, "def" 6, "ghi" 1
- X[1]: "abc" 4, "ghi" 1
- X[2]: "ghi" 1
- X[3]: "jkl" 2
- X[4]: "mno" 2
- X[5]: "abc" 4, "ghi" 3, "jkl" 1, "mno" 1
- X[6]: "abc" 5, "jkl" 3, "mno" 5
- X[7]: "abc" 1, "def" 5, "mno" 4

Stride:
- X[0]: stride = 1
- X[1]: stride = 2
- X[2]: stride = 4
Extension of ArraySum1 to reduce an arbitrary associative function, f

for ( int stride = 1; stride < X.length ; stride *= 2 ) {

    // Compute size = number of additions to be performed in stride
    int size=ceilDiv(X.length,2*stride);

    finish for(int i = 0; i < size; i++)
        async {
            if ( (2*i+1)*stride < X.length )
                X[2*i*stride] = f(X[2*i*stride], X[(2*i+1)*stride]);
        } // finish-for-async

} // for

// Divide x by y, round up to next largest int, and return result
static int ceilDiv(int x, int y) { return (x+y-1) / x; }
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 \texttt{perf.addLocalOps()} 
  — Programmer inserts calls of the form, \texttt{perf.addLocalOps(N)}, within a step to indicate abstraction execution of N application-specific abstract operations
  - e.g., floating-point ops, 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 \texttt{-perf=true} runtime option)
  — If an HJ program is executed with this option, abstract metrics are printed at end of program execution with
    \texttt{WORK(G), CPL(G), Ideal Speedup = WORK(G)/ CPL(G)}
Homework 1 Reminder

• Written assignment, due today

• Submit a softcopy of your solution in Word, PDF, or plain text format
  — Try and use turn-in script for submission, if possible
  — Otherwise, email your homework to comp322-staff at mailman.rice.edu

• See course web site for penalties for late submissions
  — Send me email if you have an extenuating circumstance for delay