COMP 322: Fundamentals of Parallel Programming

Lecture 4: Parallel Speedup, Efficiency, Amdahl’s Law

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

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

- Recap of parallel complexity for ArraySum1
- Speedup, Efficiency, Amdahl’s Law
- Use of Abstract Performance Metrics
Lower and Upper Bounds for Greedy Schedulers (Recap)

\[
\max(\text{WORK}(G)/P, \text{CPL}(G)) \leq T_p \leq \text{WORK}(G)/P + \text{CPL}(G)
\]

where

- \( G = \) computation graph
- \( \text{WORK}(G) = \) sum of time(\(N\)), for all nodes \(N\) in \(G\)
- \( \text{CPL}(G) = \) length of a longest directed path in \(CG\) \(G\), when adding up the execution times of all nodes in the path
- The above bounds are for greedy schedulers and an idealized model of \(P\) parallel processors
- There may be cases when the lower and upper bounds are not achievable
Cases when Lower and Upper Bounds approach each other

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

Case 1: There's lots of parallelism, \( WORK(G)/CPL(G) \gg P \)

\[\Rightarrow WORK(G)/P \gg CPL(G)\]

\[\Rightarrow WORK(G)/P \leq T_p \leq WORK(G)/P + CPL(G)\]

\[\Rightarrow T_p \approx WORK(G)/P\]

Case 2: There's little parallelism, \( WORK(G)/CPL(G) \ll P \)

\[\Rightarrow WORK(G)/P \ll CPL(G)\]

\[\Rightarrow CPL(G) \leq T_p \leq CPL(G) + WORK(G)/P\]

\[\Rightarrow T_p \approx CPL(G)\]
ArraySum1: Computing the sum of an array in parallel (Recap)

1. for ( int stride = 1; stride < X.length ; stride *= 2 ) {
2.   // size = number of additions 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; }

// Divide x by y, and round up to next largest int

static int ceilDiv(int x, int y) { return (x+y-1) / y; }
Reduction Tree Schema for computing Array Sum in parallel

- Define $N = \text{X.length}$
- $\text{WORK} = N - 1 = O(N)$
- Critical path length (number of stages), $\text{CPL} = O(\log(N))$
ArraySum1 pre-pass when $P < \text{array length}$

1. // Start of pre-pass: compute $P$ partial sums in parallel

2. finish for(int $j = 0; j < P; j++$) // Create $P$ tasks

3. async {

4. // Compute sum of $A[j], A[j+P], \ldots$ in task (processor) $j$

5. // Any other decomposition into $P$ partial sums is fine too

6. for(int $i = j; i < A.length; i += P$) $X[j] += A[i]$;

7. } // finish-for-async

8. // End of pre-pass: now $X[0..P-1]$ has $P$ partial sums of array $A$

9. // Use ArraySum1 algorithm (slide 5) to obtain total sum

**Complexity analysis**

- Parallel time for pre-pass in lines 1-7 = $O(N/P)$, where $N = A.length$
- Parallel time for ArraySum1 algorithm = $O(\log P)$
- Total parallel time, $T(N,P) = O(N/P + \log P)$
ArraySum: Ideal Parallel Time as function of $P$

- Total parallel time, $T(N,P) = \frac{N}{P} + \log_2(\min(P,N))$, depends on
  - Input size, $N$
  - Number of processors, $P$

![Graph showing parallel time $T(N,P)$ vs. $P$ for different $N$ values](image)
Goals for Today’s Lecture

- Recap of parallel complexity for ArraySum1
- Speedup, Efficiency, Amdahl’s Law
- Use of Abstract Performance Metrics
Speedup Definitions

• Speedup(N,P) = T(N,1)/T(N,P)
  —Factor by which the use of P processors speeds up execution time relative to 1 processor, for input size N
  —For ideal executions without overhead, 1 <= Speedup(P) <= P

• Strong scaling
  —Goal is linear speedup for a given input size
    - When Speedup(N,P) = k*P, for some constant k, 0 < k < 1
  —In practice, we may also see
    - Speedup(P) < 1 (slowdown)
    - Speedup(P) > P (super-linear speedup)

• Weak scaling
  —Increase problem size to use processors more efficiently
  —Define Weak-Speedup(N(P),P) = T(N(P),1)/T(N(P),P), where input size N(P) increases with P
ArraySum: Speedup as function of P

- Speedup(N,P) = T(N,1)/T(N,P) = N/(N/P + log\(_2\)(min(P,N))
- Asymptotically, Speedup(N,P) → N/log\(_2\)N, as P → infinity
Efficiency Metrics

- **Efficiency(P) = Speedup(P)/ P = T₁/(P * T_P)**
  - Processor efficiency --- figure of merit that indicates how well a parallel program uses available processors
  - For ideal executions without overhead, 1/P <= Efficiency(P) <= 1

- **Half-performance metric**
  - \(N_{1/2}\) = input size that achieves Efficiency(P) = 0.5 for a given P
  - Figure of merit that indicates how large an input size is needed to obtain efficient parallelism
  - A larger value of \(N_{1/2}\) indicates that the problem is harder to parallelize efficiently
ArraySum: Efficiency as function of P

- Common approach: choose largest number of processors that delivers efficiency above a given limit e.g., 50%
Amdahl’s Law [1967]

- If \( q \leq 1 \) is the fraction of WORK in a parallel program that must be executed sequentially for a given input size \( N \), then the best speedup that can be obtained for that program is \( \text{Speedup}(N,P) \leq 1/q \).

- Observation follows directly from critical path length lower bound on parallel execution time
  - \( \text{CPL} \geq q \times T(N,1) \)
  - \( T(N,P) \geq q \times T(N,1) \)
  - \( \text{Speedup}(N,P) = T(N,1)/T(N,P) \leq 1/q \)

- This upper bound on speedup simplistically assumes that work in program can be divided into sequential and parallel portions
  - Sequential portion of WORK = \( q \)
    - also denoted as \( f_s \) (fraction of sequential work)
  - Parallel portion of WORK = \( 1-q \)
    - also denoted as \( f_p \) (fraction of parallel work)

- Computation graph is more general and takes dependences into account
Illustration of Amdahl’s Law:
Best Case Speedup as function of Parallel Portion

Figure source: http://en.wikipedia.org/wiki/Amdahl’s law
Goals for Today’s Lecture

- Recap of parallel complexity for ArraySum1
- Speedup, Efficiency, Amdahl’s Law
- Use of Abstract Performance Metrics
HJ Abstract Performance Metrics (Recap)

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

• Calls to `perf.addLocalOps()`
  — Programmer inserts calls of the form, `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 `-perf=true` runtime option)
  — If an HJ program is executed with this option, abstract metrics are printed at end of program execution with
    \[ \text{Ideal Speedup} = \frac{\text{WORK}(G)}{\text{CPL}(G)} \]
Where should perf.addLocalOps() calls be placed?

• Answer: It depends. In HW2, we asked you to count each call to combine() as 1 unit, but here’s the general idea ...

• We'll say that a cost function Cost(n) is “order $f(n)$”, or simply “$O(f(n))$” (read “Big-O of $f(n)$”) if
  
  $-Cost-X(n) < \text{factor} \times f(n)$, for sufficiently large $n$, for some constant factor.

• Examples:
  
  $-Cost-A(n) = 2n^3 + n^2 + 1$  
  $Cost-A$ is $O(n^3)$

  $-Cost-B(n) = 3n^2 + 10$  
  $Cost-B$ is $O(n^2)$

  $-Cost-C(n) = 2^n$  
  $Cost-C$ is $O(2^n)$
Famous "Complexity Classes"

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Time Complexity</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>( O(1) )</td>
<td>constant-time</td>
<td>(head, tail)</td>
</tr>
<tr>
<td>( O(\log n) )</td>
<td>logarithmic</td>
<td>(binary search)</td>
</tr>
<tr>
<td>( O(n) )</td>
<td>linear</td>
<td>(vector multiplication)</td>
</tr>
<tr>
<td>( O(n \log n) )</td>
<td>&quot;n logn&quot;</td>
<td>(sorting)</td>
</tr>
<tr>
<td>( O(n^2) )</td>
<td>quadratic</td>
<td>(matrix addition)</td>
</tr>
<tr>
<td>( O(n^3) )</td>
<td>cubic</td>
<td>(matrix multiplication)</td>
</tr>
<tr>
<td>( n^{O(1)} )</td>
<td>polynomial</td>
<td>(...many! ...)</td>
</tr>
<tr>
<td>( 2^{O(n)} )</td>
<td>exponential</td>
<td>(guess password)</td>
</tr>
</tbody>
</table>
Where should perf.addLocalOps() calls be placed?

• Focus on key metric of interest in your algorithm

• Don’t count operations that are incidental to your algorithm
  — They can be important implementation considerations, but may not contribute to understanding your algorithm

• Since big-O analysis does not care about differences within a constant factor, you can just use a unit cost as a stand-in for a constant number of operations