COMP 322: Fundamentals of Parallel Programming

Lecture 4: Abstract Performance Metrics (contd), Parallel Efficiency, Amdahl’s Law, Weak Scaling

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

https://wiki.rice.edu/confluence/display/PARPROG/COMP322
Announcements

- **Coursera access**
  - You should only access the course site via rice.coursera.org and Shibboleth

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

- **Week 1 lecture quiz will be posted by Tuesday**

- **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
Fundamentals of Parallel Programming
Vivek Sarkar

Use this link

Login via Shibboleth
You can login via your school credentials to this class.

Not this one
Solution to Worksheet #3: Strong Scaling for Array Sum

• Assume \( T(S, P) \sim \frac{\text{WORK}(G, S)}{P} + \frac{\text{CPL}(G, S)}{P} = \frac{(S-1)}{P} + \log_2(S) \) for a parallel array sum computation with input size \( S \) on \( P \) processors

• Strong scaling
  — Assume \( S = 1024 \implies \log_2(S) = 10 \)
  — Compute Speedup\((P)\) for \( S=1024 \) on 10, 100, 1000 processors
    - \( T(P) = \frac{1023}{P} + 10 \)
    - Speedup(10) = \( \frac{T(1)}{T(10)} \approx 9.2 \)
    - Speedup(100) = \( \frac{T(1)}{T(100)} \approx 51.1 \)
    - Speedup(1000) = \( \frac{T(1)}{T(1000)} \approx 102.3 \)
    - Ideal parallelism = \( \frac{T(1)}{T(\infty)} = \frac{1033}{10} = 103.3 \)
  — Why is it worse than linear?
    - The critical path limits speedup as \( P \) increases (speedup is limited by ideal parallelism)
Plot of Speedup(P) as a function of P

Data points:
- Speedup as a function of number of processors, P
- Ideal parallelism

Axes:
- X-axis: 1.E+00 to 1.E+06
- Y-axis: 0 to 120
Plot of parallel time, $T(P)$, as a function of $P$

Parallel time as a function of number of processors, $P$

Critical path length
Outline of Today’s Lecture

• Abstract Performance Metrics (contd)
• Parallel Efficiency, Amdahl's Law
• Weak Scaling

• Acknowledgments
  — COMP 322 Module 1 handout, Sections 3.3, 3.4
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, \texttt{perf.doWork(N)}, within a step to indicate abstraction execution of \textit{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 \texttt{WORK(G), CPL(G), Ideal Speedup = \texttt{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.
**Big-O notation --- where should doWork() calls be placed?**

- **Answer:** It depends. For ArraySum, we counted each add operator as 1 unit. In HW1 (Quicksort), we asked you to count each call to combine() as 1 unit. 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**
  
  $\text{Cost-X}(n) < \text{factor} \times f(n)$, for sufficiently large $n$, for some constant factor

- **Examples:**
  
  - $\text{Cost-A}(n) = 2n^3 + n^2 + 1$ Cost-A is $O(n^3)$
  - $\text{Cost-B}(n) = 3n^2 + 10$ Cost-B is $O(n^2)$
  - $\text{Cost-C}(n) = 2^n$ Cost-C is $O(2^n)$
Some well-known “Complexity Classes"

<table>
<thead>
<tr>
<th>Complexity Class</th>
<th>Description</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>$O(n \cdot \log n)$</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>
So, where should doWork() 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 ignores differences within a constant factor, you can always use a unit cost as a stand-in for a constant number of operations
Another example: String Search
(count of all occurrences)

• Inputs
  — text: a long string with N characters to search in
  — pattern: a short string of M characters to search for

• Output
  — count of all occurrences of pattern in text

• Example
  — text: “abacadabracabraacadababacadabracabraacadabracadabra”
  — pattern: aca
  — number of occurrences: 6

• Applications
  — Word processing, virus scans, information retrieval, computational biology, web search engines, ...

• Variations
  — Existence of an occurrence, index of any occurrence, indices of all occurrences
Brute Force Sequential Algorithm for String Search

1. public static int search(char[] pattern, char[] text) {
2.     int M = pattern.length; int N = text.length; int count = 0;
3.     for (int i = 0; i <= N - M; i++) {
4.         int j; // search for pattern starting at text[i]
5.         for (j = 0; j < M; j++) {
6.             // Count each char comparison as 1 unit of work
7.             perf.doWork(1); // Assume that all else takes zero time!
8.             if (text[i+j] != pattern[j]) break;
9.         } // for (j = ... )
10.        if (j == M) count = count+1; // found at offset i
11.    }
12.    return count;
13. }

What is the complexity of this algorithm?
Parallel Algorithm for String Search

• Consider a parallel algorithm in which each iteration is spawned as a separate async task
  — Some modifications will be needed to ensure that there are no “data races” on count in line 10
    - For example, replace count by an array indexed by iteration i, and set each element to 0 or 1 depending on whether or not an occurrence was found. Sum up the array elements at the end.
  — Other parallel algorithms are possible too

• For the above algorithm
  — \( \text{WORK} = O(M \times N) \)
  — \( \text{CPL} = O(M) \)
  — Abstract execution time can be approximated by its upper bound,
    - \( T(M, N, P) = \frac{M \times N}{P} + M \)
  — Ignores time for Array Sum, etc. since only character comparison is counted as work
Outline of Today’s Lecture

• Abstract Performance Metrics (contd)

• Parallel Efficiency, Amdahl's Law

• Weak Scaling

• Acknowledgments
  — COMP 322 Module 1 handout, Sections 3.3, 3.4
How many processors should we use?

- **Efficiency**\( (P) = \frac{\text{Speedup}(P)}{P} = \frac{T_1}{(P \times T_P)} \)
  - Processor efficiency --- figure of merit that indicates how well a parallel program uses available processors
  - For ideal executions without overhead, \( \frac{1}{P} \leq \text{Efficiency}(P) \leq 1 \)

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

- **How many processors to use?**
  - Common goal: choose number of processors, \( P \) for a given input size, \( S \), so that efficiency is at least 0.5
ArraySum: Speedup as function of array size, $S$, and number of processors, $P$

- \[ \text{Speedup}(S,P) = \frac{T(S,1)}{T(S,P)} = \frac{S}{S/P + \log_2(S)} \]
- Asymptotically, \[ \text{Speedup}(S,P) \rightarrow \frac{S}{\log_2 S}, \text{ as } P \rightarrow \infty \]

**How many processors should we use?**
Time for worksheet #3!
Amdahl’s Law [1967]

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

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

- This upper bound on speedup simplistically assumes that work in program can be divided into sequential and parallel portions
  
  \[ \text{Sequential portion of WORK} = q \]
  
  - also denoted as \( f_S \) (fraction of sequential work)

  \[ \text{Parallel portion of WORK} = 1-q \]
  
  - 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
Outline of Today’s Lecture

• Abstract Performance Metrics (contd)
• Parallel Efficiency, Amdahl's Law
• Weak Scaling

• Acknowledgments
  — COMP 322 Module 1 handout, Sections 3.3, 3.4
Strong Scaling and Speedup (Recap)

- Define 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 <= Speedup(P) <= P
  - Linear speedup
    - When Speedup(P) = k*P, for some constant k, 0 < k < 1
- Referred to as “strong scaling” because input size is fixed
Weak Scaling

- Consider a computation graph, $CG$, in which all node execution times are parameterized by input size $S$
  - $TIME(N,S) =$ time to execute node $N$ with input size $S$
  - $WORK(G,S) =$ sum of $TIME(N,S)$ for all nodes $N$
  - $CPL(G,S) =$ critical path length for $G$, assuming node $N$ takes $TIME(N,S)$

- Let $T(S,P) =$ time to execute $CG$ with input size $S$ on $P$ processors

- Weak scaling
  - Allow input size $S$ to increase with number of processors i.e., make $S$ a function of $P$
  - Define Weak-Speedup($S(P),P$) = $T(S(P),1)/T(S(P),P)$, where input size $S(P)$ increases with $P$
    - Note that $T(S(P),1)$ is a hypothetical projection of running a larger problem size, $S(P)$, on 1 processor
Weak Scaling for Array Sum

- Recall that $T(S,P) = \frac{(S-1)}{P} + \log_2(S)$ for a parallel array sum computation
- For weak scaling, assume $S(P) = 1024^P$

  $\Rightarrow \text{Weak-Speedup}(S(P),P) = \frac{T(S(P),1)}{T(S(P),P)}$

  $= \frac{(1024^P-1)+\log_2(1024^P))}{((1024^P-1)/P+\log_2(1024^P))} \sim P$
Worksheet #4: how many processors should we use for ArraySum?

Name 1: ___________________          Name 2: ___________________

For ArraySum on $P$ processors and input array size, $S$,

$$\text{Speedup}(S,P) = \frac{T(S,1)}{T(S,P)} = \frac{S}{(S/P + \log_2(S))}$$

- **Question:** For a given $S$, what value of $P$ should we choose to obtain $\text{Efficiency}(P) = 0.5$? Recall that $\text{Efficiency}(P) = 0.5 \implies \text{Speedup}(S,P)/P = 0.5$.

- **Answer (derive value of $P$ as a symbolic function of $S$):**