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

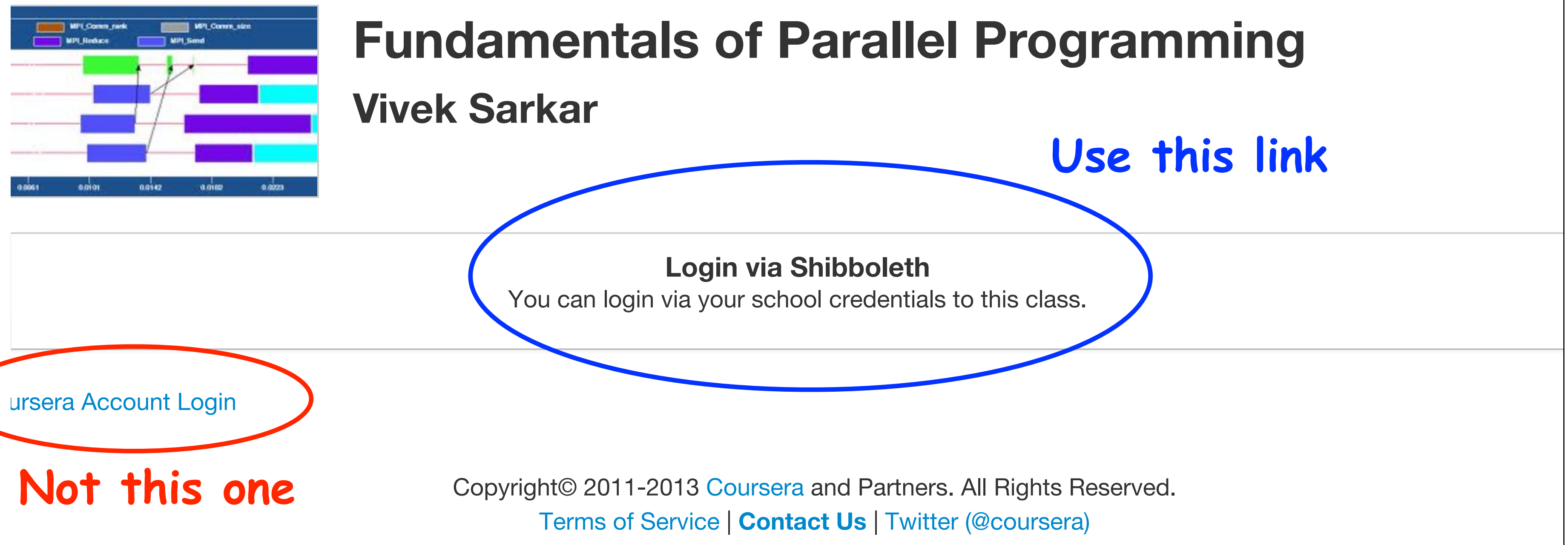
https://wiki.rice.edu/confluence/display/PARPROG/COMP322

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

Coursera web site

(https://rice.coursera.org/parallelprog-001)



Fundamentals of Parallel Programming

Vivek Sarkar

Use this link

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

ursera Account Login

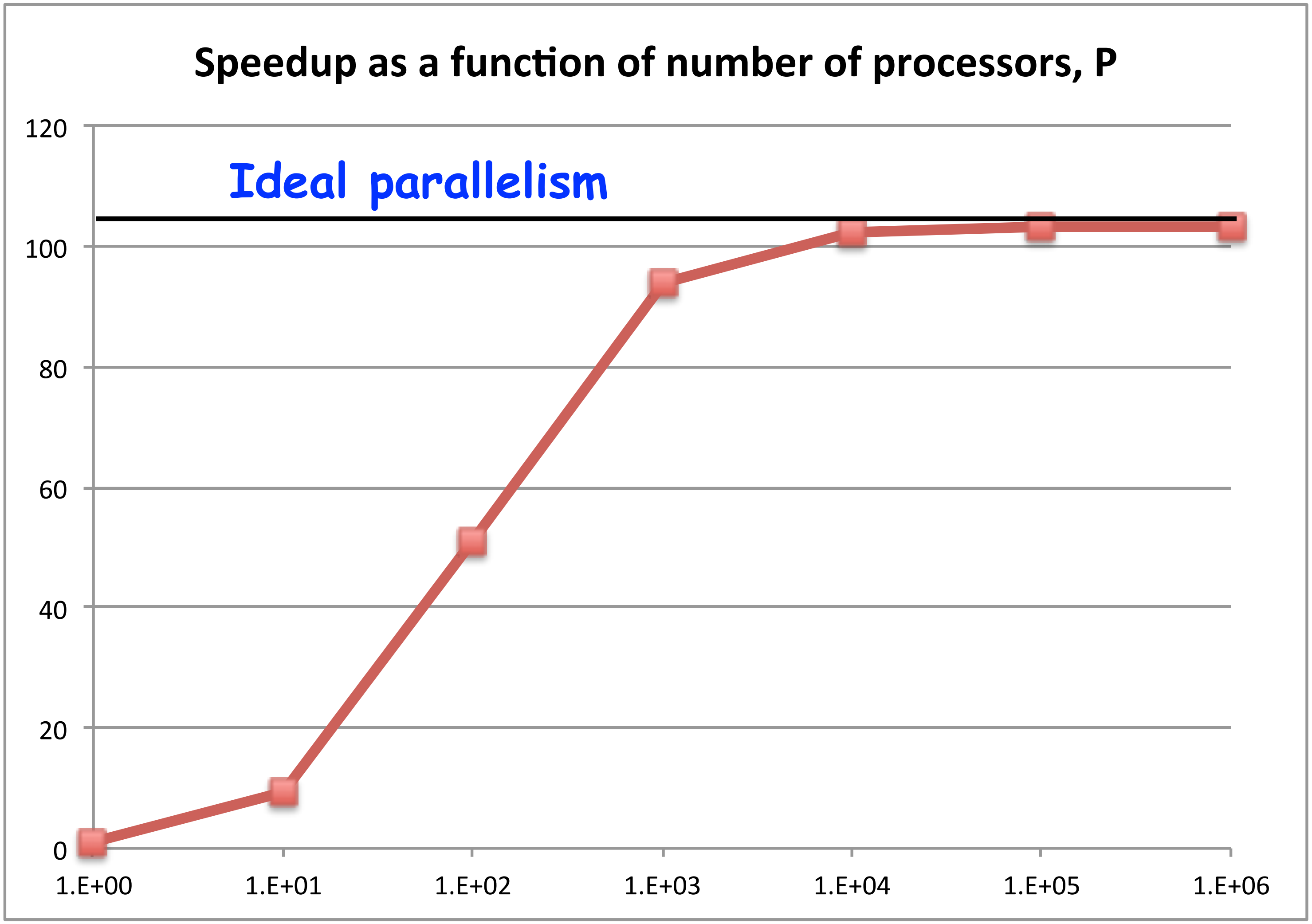
Not this one

Copyright© 2011-2013 Coursera and Partners. All Rights Reserved.
Terms of Service | **Contact Us** | Twitter (@coursera)

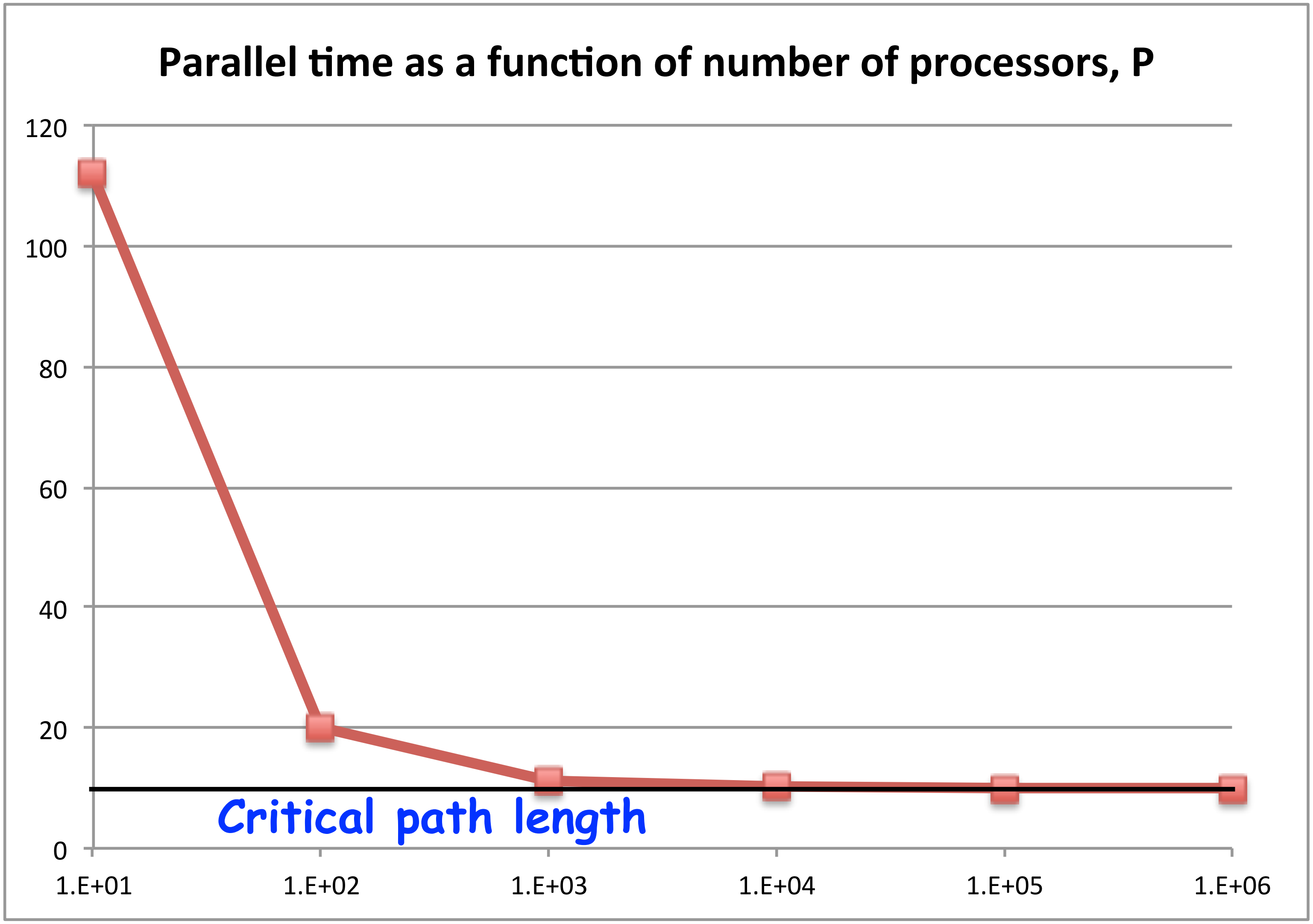
Solution to Worksheet #3: Strong Scaling for Array Sum

- Assume T(S,P) ~ WORK(G,S)/P + CPL(G,S) = (S-1)/P + log2(S) for a parallel array sum computation with input size S on P processors
- Strong scaling
  - Assume S = 1024 ==> log2(S) = 10
  - Compute Speedup(P) for S=1024 on 10, 100, 1000 processors
    - T(P) = 1023/P + 10
    - Speedup(10) = T(1)/T(10) ~ 9.2
    - Speedup(100) = T(1)/T(100) ~ 51.1
    - Speedup(1000) = T(1)/T(1000) ~ 102.3
    - Ideal parallelism = $T(1)/T(_\infty)$ = 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



Plot of parallel time, T(P), as a function of P



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
        - https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

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

```
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
  // Compute size = number of adds 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 ) {
        perf.doWork(1);
        X[2*i*stride] += X[(2*i+1)*stride];
      }
    } // finish-for-async
} // for

```

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
  - Cost-X(n) < factor * $f(n)$, for sufficiently large n, for some constant factor
- Examples:
  - Cost-A(n) = $2*n^3 + n^2 + 1$ Cost-A is $O(n^3)$
  - Cost-B(n) = $3*n^2 + 10$ Cost-B is $O(n^2)$
  - Cost-C(n) = 2^n Cost-C is $O(2^n)$

Some well-known “Complexity Classes"

$O(1)$	constant-time	(head, tail)
$O(\log n)$	logarithmic	(binary search)
$O(n)$	linear	(vector multiplication)
$O(n * \log n)$	"n logn"	(sorting)
$O(n^2)$	quadratic	(matrix addition)
$O(n^3)$	cubic	(matrix multiplication)
$n^{O(1)}$	polynomial	(…many! …)
$2^{O(n)}$	exponential	(guess password)

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: “abacadabrabracabracadababacadabrabracabracadabrabrabr”
  - 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

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

What is the complexity of this algorithm?

Parallel Algorithm for String Search

- Consider a parallel algorithm in which each i 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
  - WORK = O(M*N)
  - CPL = O(M)
  - Abstract execution time can be approximated by its upper bound,
    - T(M,N,P) = M*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
    - https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

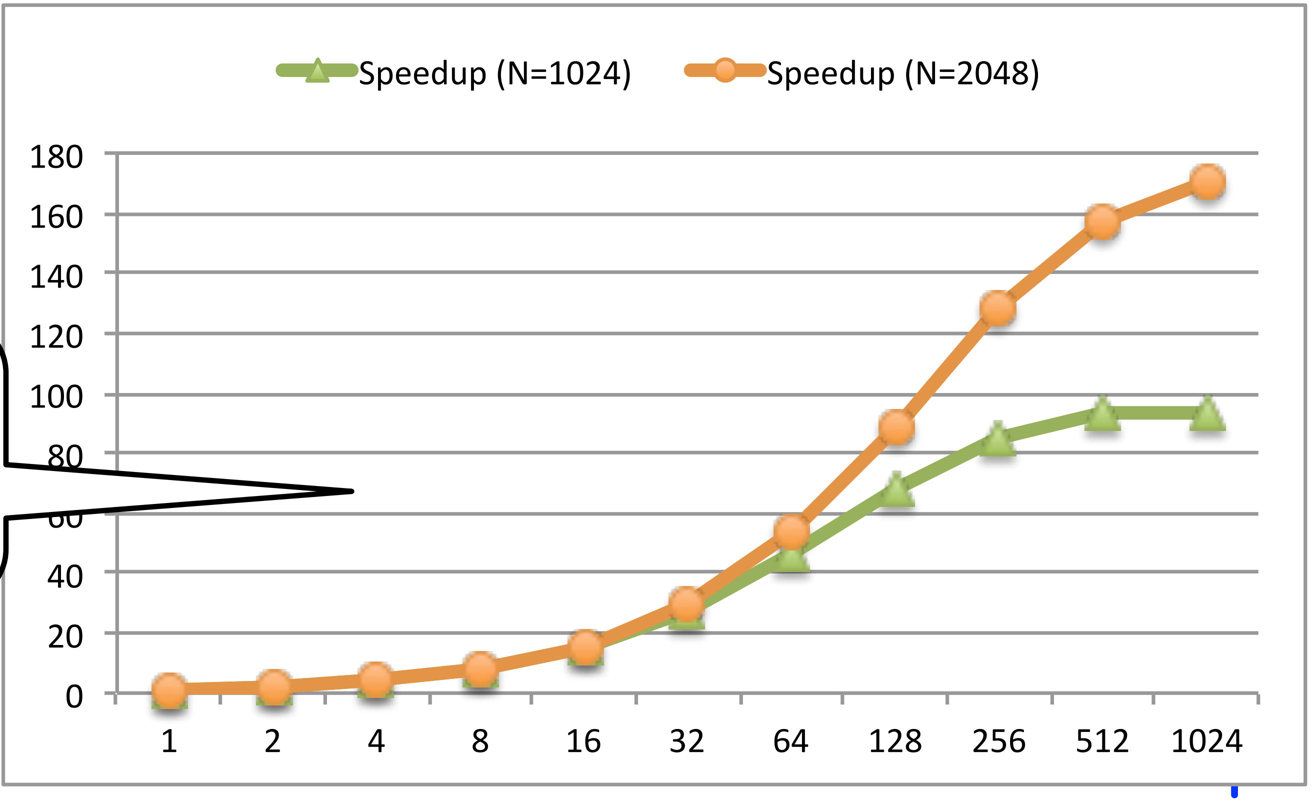
How many processors should we use?

- Efficiency(P) = Speedup(P)/ P = $T_1/(P * T_P)$
  - 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
  - $S_{1/2}$ = 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 $S_{1/2}$ 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

- Speedup(S,P) = T(S,1)/T(S,P) = $S/(S/P + \log_2(S))$
- Asymptotically, Speedup(S,P) --> $S/\log_2 S$, as P --> infinity

Speedup(S,P)



How many processors should we use?
Time for worksheet #3!

Amdahl's Law [1967]

- If $q \le 1$ is the fraction of 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) $\le 1/q$.
- Observation follows directly from critical path length lower bound on parallel execution time
  - CPL >= q * T(S,1)
  - T(S,P) >= q * T(S,1)
  - Speedup(S,P) = T(S,1)/T(S,P) <= 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

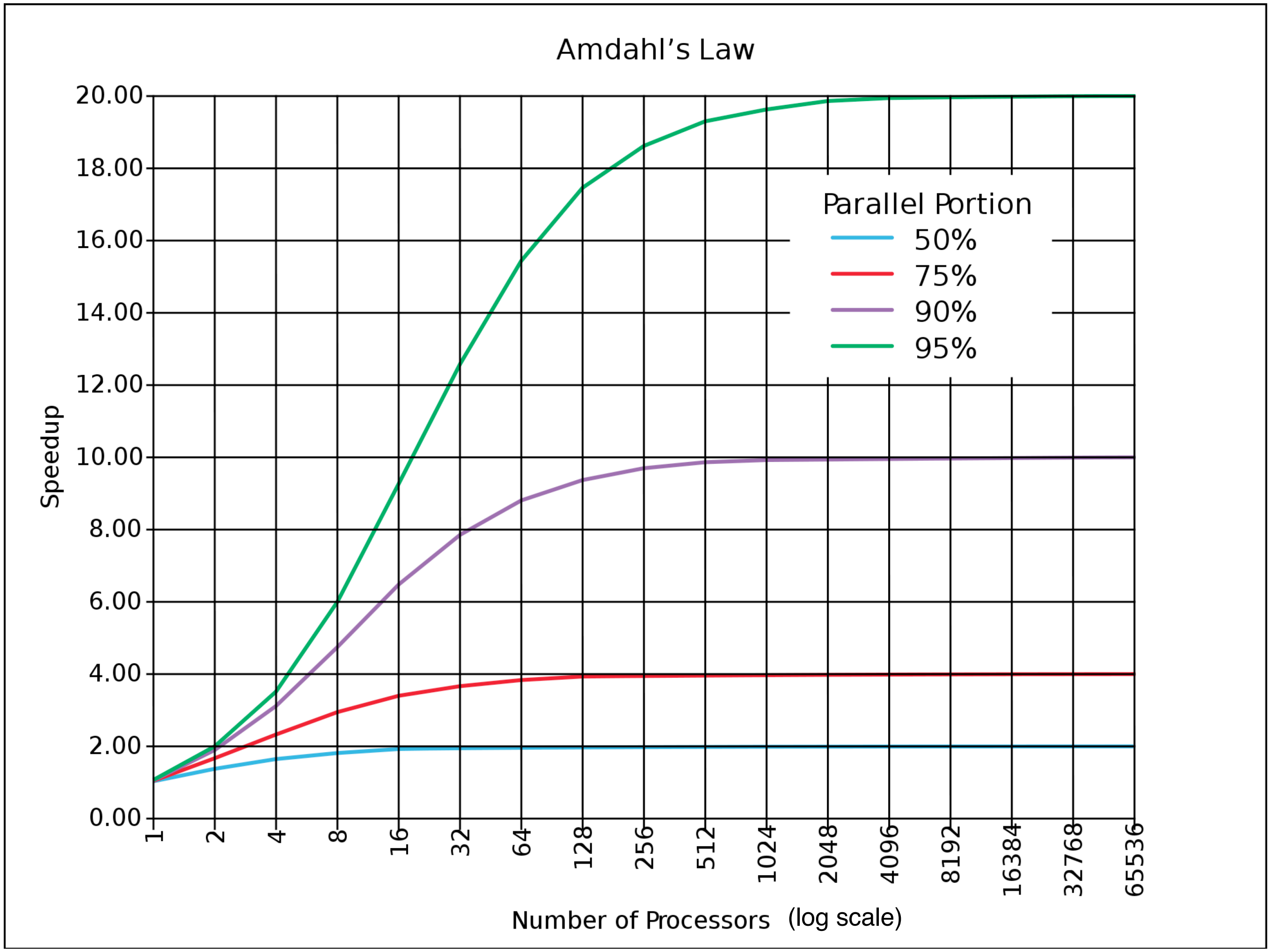


Figure source: http://en.wikipedia.org/wiki/Amdahl's law

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
    - https://svn.rice.edu/r/comp322/course/module1-2013-01-06.pdf

Strong Scaling and Speedup (Recap)

- Define Speedup(P) = 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

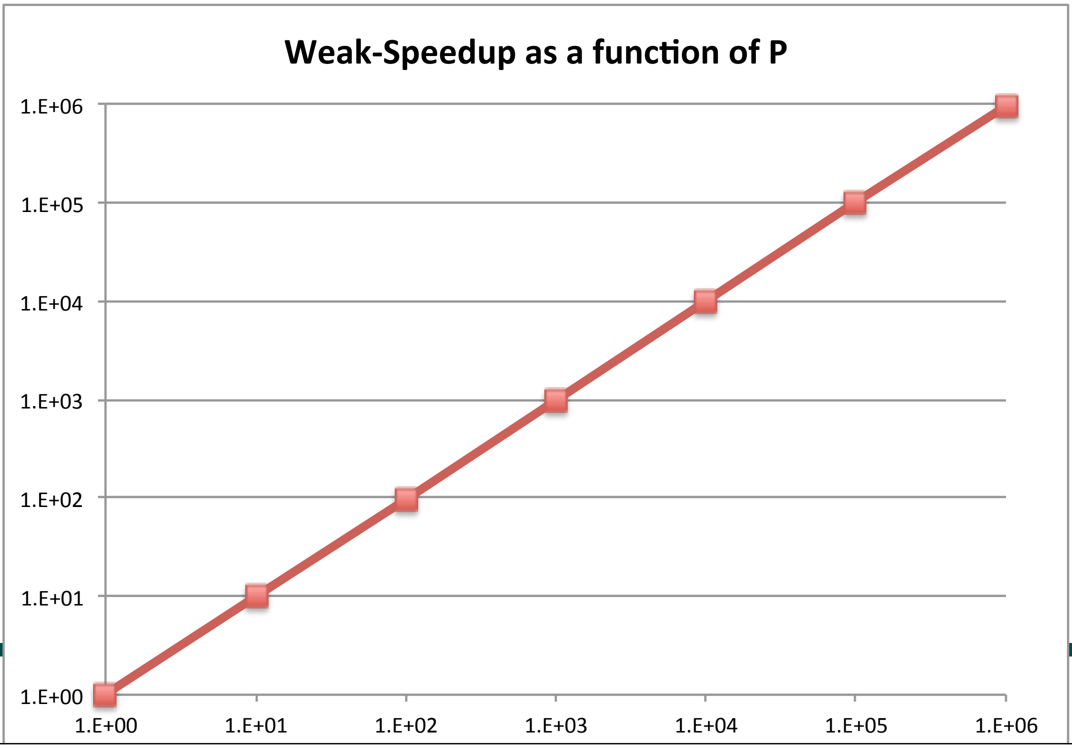
- Consider a computation graph, CG, in which all node execution times are parameterized by input size S
  - TIME(N,S) = time to execute node N with input size S
  - WORK(G,S) = sum of TIME(N,S) for all nodes N
  - CPL(G,S) = critical path length for G, assuming node N takes TIME(N,S)
- Let T(S,P) = time 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) = (S-1)/P + log2(S) for a parallel array sum computation
- For weak scaling, assume S(P) = 1024*P

==> Weak-Speedup(S(P),P) = T(S(P),1)/T(S(P),P)

= ((1024*P-1)+log2(1024*P)) / ((1024*P-1)/P+log2(1024*P)) ~ 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,

Speedup(S,P) = T(S,1)/T(S,P) = $S/(S/P + \log_2(S))$

- Question: For a given S, what value of P should we choose to obtain Efficiency(P) = 0.5? Recall that Efficiency(P) = 0.5 ==> Speedup(S,P)/P = 0.5.
- Answer (derive value of P as a symbolic function of S):