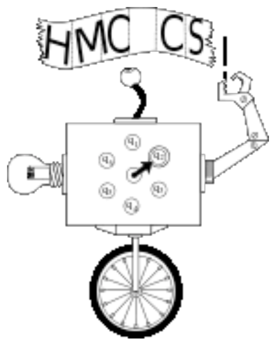


CS 181E: Fundamentals of Parallel Programming

Instructor: Vivek Sarkar
Co-Instructor: Ran Libeskind-Hadas

http://www.cs.hmc.edu/courses/2012/fall/cs181e/

Recap of Lecture 1

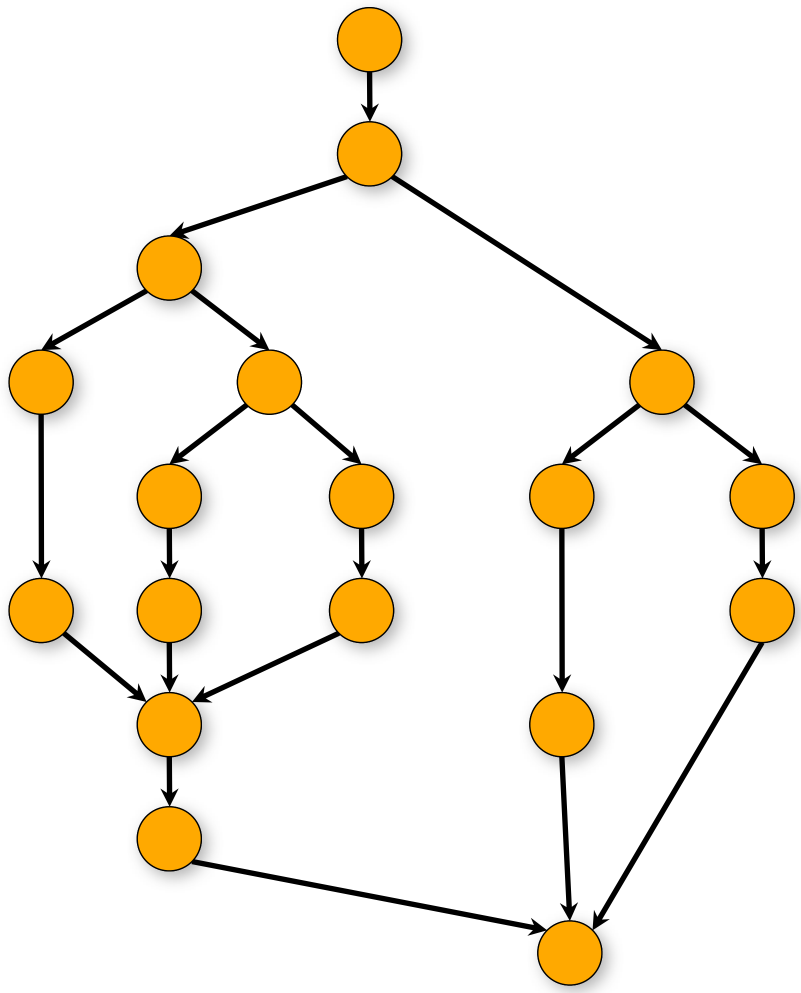
- Introduction
- Async-Finish Parallel Programming
- Computation Graphs
- Abstract Performance Metrics
- Parallel Array Sum

Worksheet #1 Solution: Insert finish to get correct Two-way Parallel Array Sum program

```
// Start of Task T0 (main program)
sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
finish {
  async { // Task T1 computes sum of upper half of array
    for(int i=X.length/2; i < X.length; i++) sum2 += X[i];
  }
  // T0 computes sum of lower half of array
  for(int i=0; i < X.length/2; i++) sum1 += X[i];
}
// Task T0 waits for Task T1 (join)
return sum1 + sum2;
```

Worksheet #2 solution: what is the critical path length and ideal speedup of this graph?

- Assume time(N) = 1 for all nodes in this graph



WORK(G) = 18

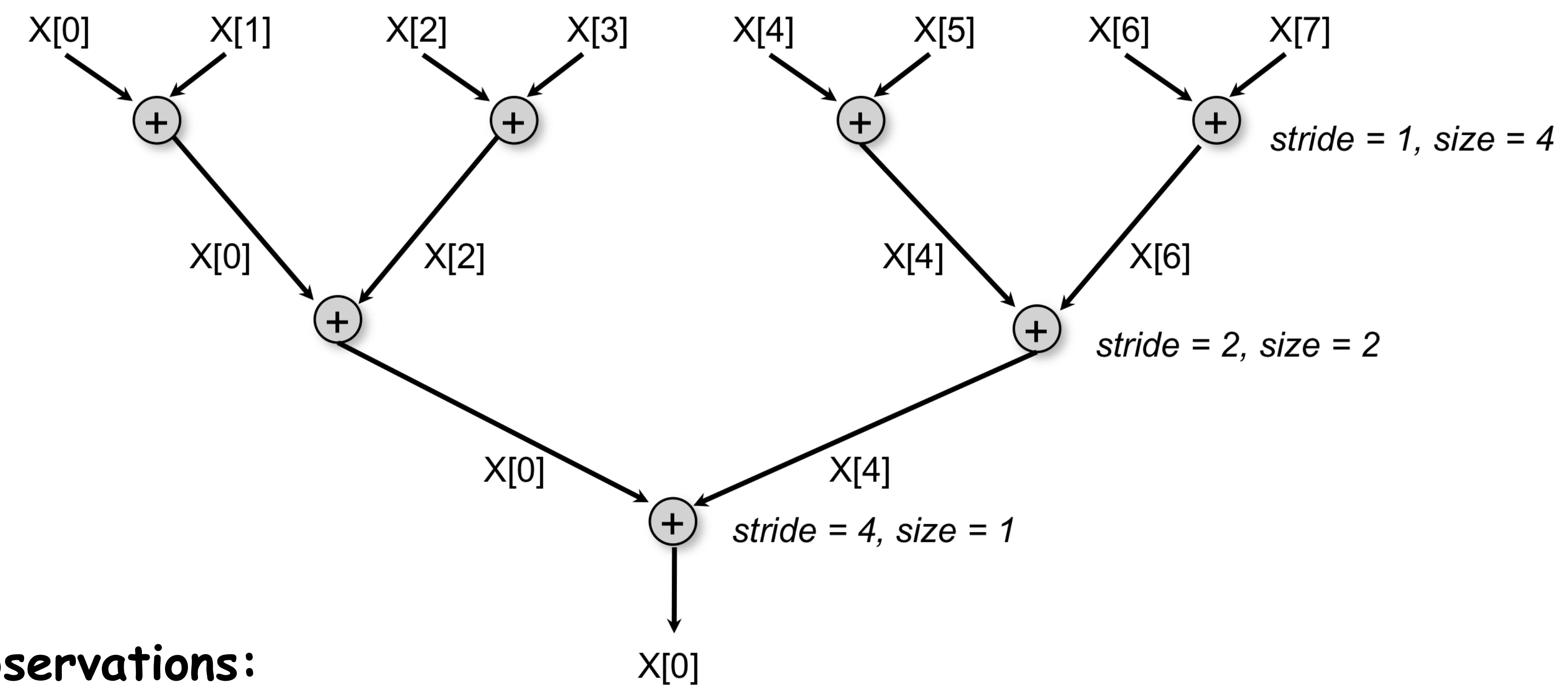
CPL(G) = 9

Ideal Speedup = 2

Outline of Today's Lecture

- <u>Parallel Array Sum (contd)</u>
- Speedup, Efficiency, Amdahl's Law
- Understanding Data and Control Flow between an Async Task and its Parent
- Data Races and Determinism

Reduction Tree Schema for computing Array Sum in parallel



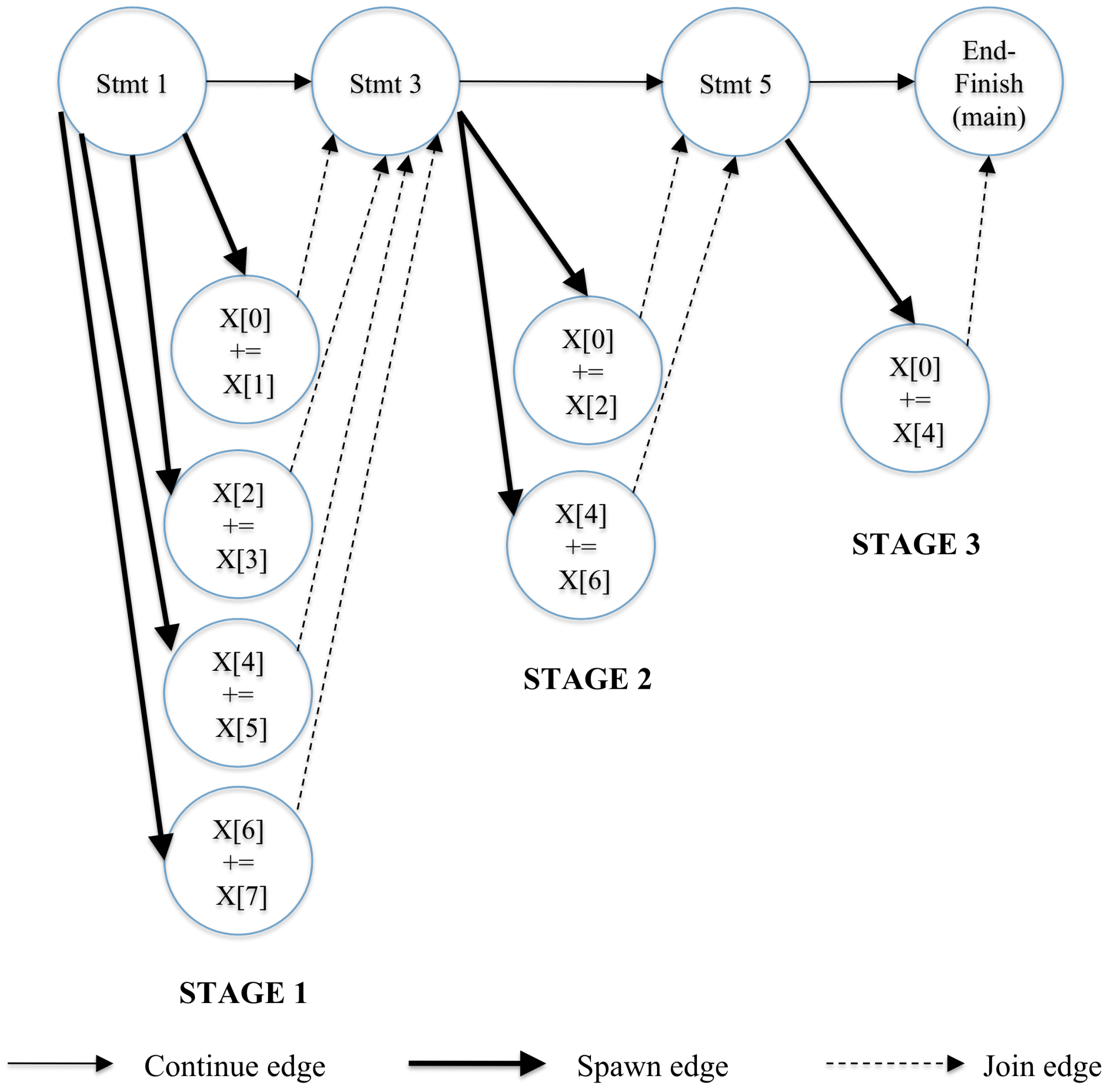
Observations:

- 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)

```
finish { // STAGE 1: stride = 1, size = 4 parallel
   additions

  async X[0]+=X[1]; async X[2]+=X[3];

  async X[4]+=X[5]; async X[6]+=X[7];

}

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



Generalization to arbitrary sized arrays (ArraySum1)

```
for ( int stride = 1; stride < X.length ; stride *= 2 ) {
  // Compute size = number of additions to be performed
  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
static int ceilDiv(int x, int y) { return (x+y-1) / y; }
```

Recap of Big-O notation

- We 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 $O(2^n)$
- 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
  - Idea behind HJ's abstract performance metrics

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)

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 = O(log(n))
- Ideal parallelism = WORK/CPL = O(n) / O(log(n))
- Execution time on P processors
  - Use upper bound from Lecture 1 to get
  - T_P = O(WORK/P + CPL) = O(n/P + log(n))
  - Speedup on P processors = T_1/T_P = O(n) / O(n/P + log(n))
  - Algorithm is optimal for P = n / log(n), or fewer, processors – why?

Refining the complexity analysis with a pre-pass for sequential partial sums

```
// Start of pre-pass: compute P partial sums in parallel
finish for(int j = 0; j < P; j++) // Create P tasks
   async {
     // Compute sum of A[j],A[j+P],... in task (processor) j
     // Any other decomposition into P partial sums is fine too
     for(int i = j; i < A.length; i += P) X[j] += A[i];
   } // finish-for-async
// End of pre-pass: now X[0..P-1] has P partial sums of array A
// Use ArraySum1 algorithm (slide 5) to obtain total sum
```

Complexity analysis

- Parallel time on P processors for lin
- Parallel time for adding P partial su
- Total parallel time, T(N,P) = O(N/P
  - Better estimate than O(N/P + log

That was fun! Let's do another complexity analysis in worksheet #1!

Generalized Array Reductions

- ArraySum1 can easily be adapted to reduce any associative function f
  - f(x,y) is said to be associative if f(a,f(b,c)) = f(f(a,b),c) for any inputs a, b, and c
- Sequential reduction of X, an array of objects of type T:

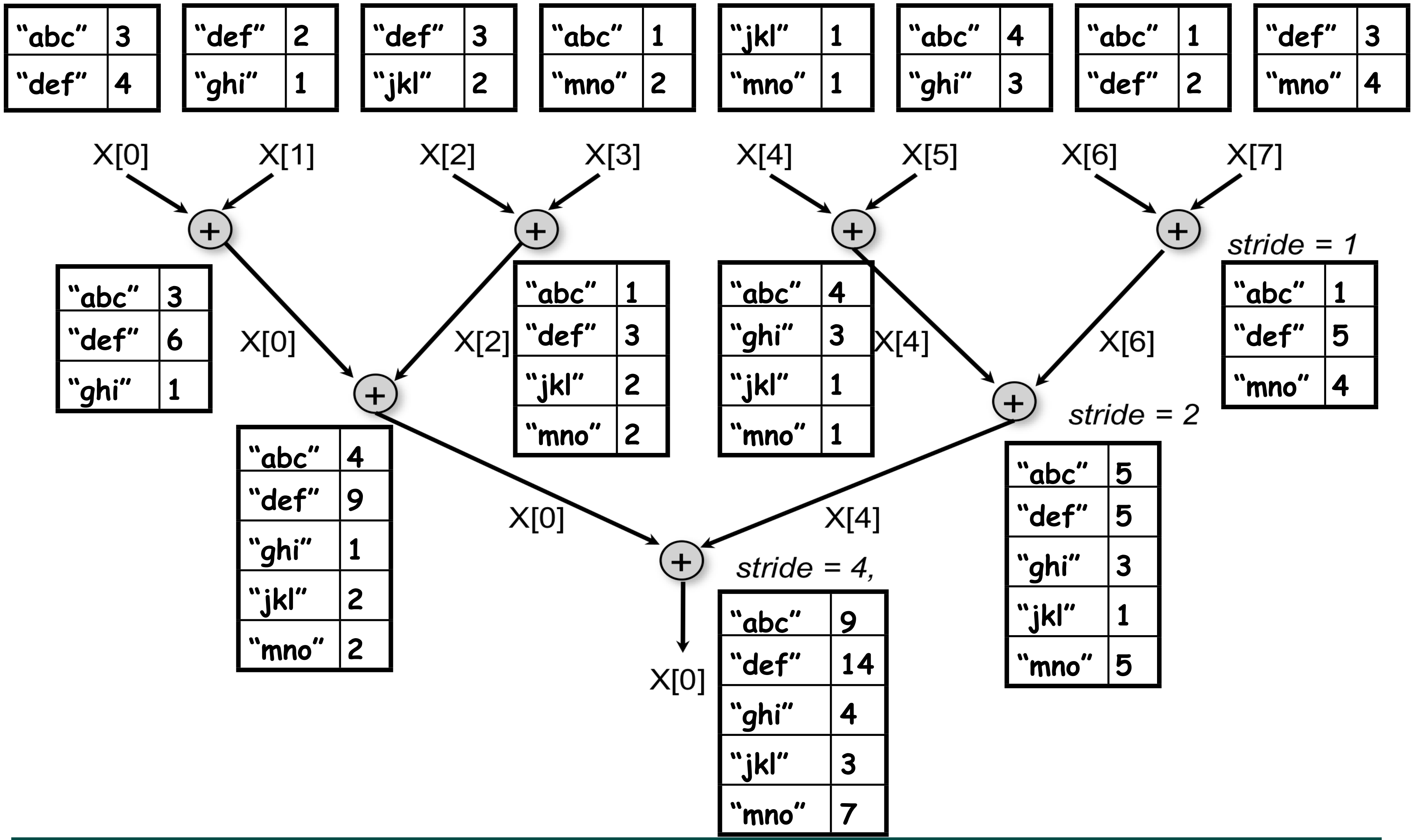
```
T result=X[0];

for(int i=1 ; i < X.length ; i++ ) result=f(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

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
  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
static int ceilDiv(int x, int y) { return (x+y-1) / y; }
```

Example of Generalized Reduction: WordCount



"abc"	3
"def"	4

"def"	2
"ghi"	1

"def"	3
"jkl"	2

"abc"	1
"mno"	2

"jkl"	1
"mno"	1

"abc"	4
"ghi"	3

"abc"	1
"def"	2

"def"	3
"mno"	4

"abc"	3
"def"	6
"ghi"	1

"abc"	1
"def"	3
"jkl"	2
"mno"	2

"abc"	4
"ghi"	3
"jkl"	1
"mno"	1

"abc"	1
"def"	5
"mno"	4

"abc"	4
"def"	9
"ghi"	1
"jkl"	2
"mno"	2

"abc"	5
"def"	5
"ghi"	3
"jkl"	1
"mno"	5

"abc"	9
"def"	14
"ghi"	4
"jkl"	3
"mno"	7

Outline of Today's Lecture

- Parallel Array Sum (contd)
- <u>Speedup, Efficiency, Amdahl's Law</u>
- Understanding Data and Control Flow between an Async Task and its Parent
- Data Races and Determinism

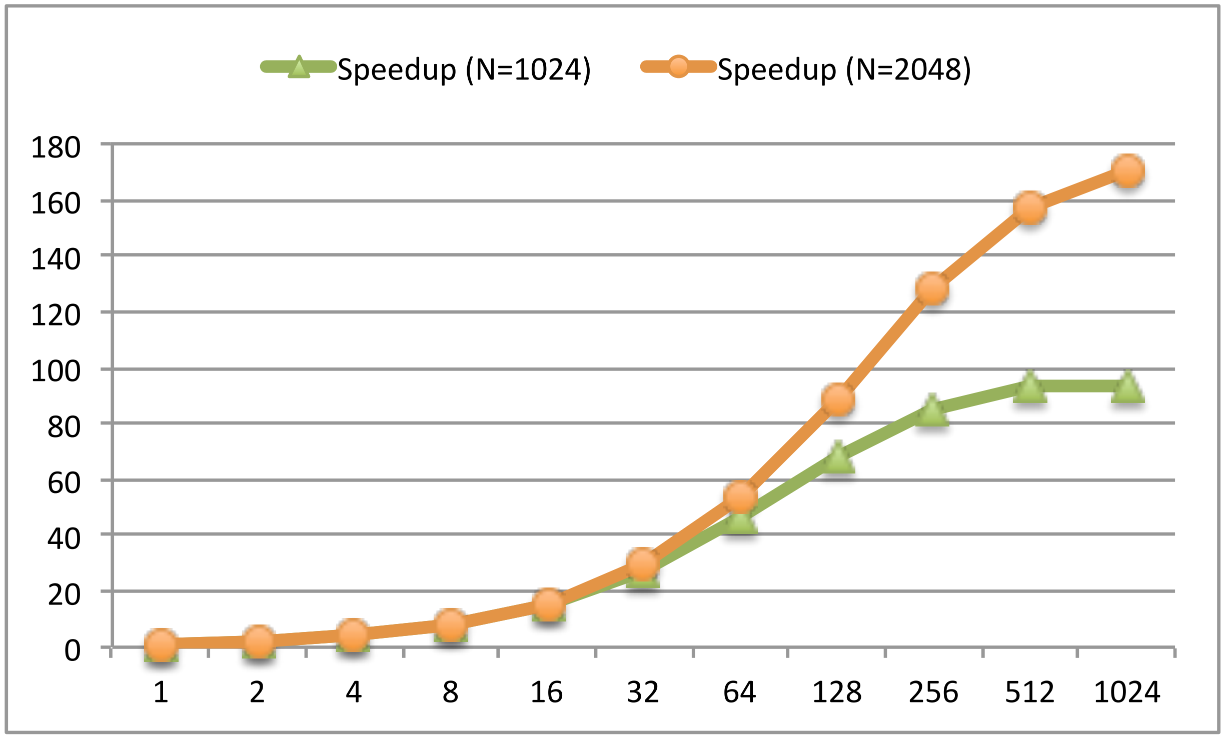
Speedup Definitions

- 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
- Strong scaling
  - Goal is linear speedup for a given input size
    - When Speedup(N,P) = k*P, for some constant k, 0 < k < 1
  - For ideal executions without overhead, 1 <= Speedup(P) <= P
  - 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

Speedup(N,P)



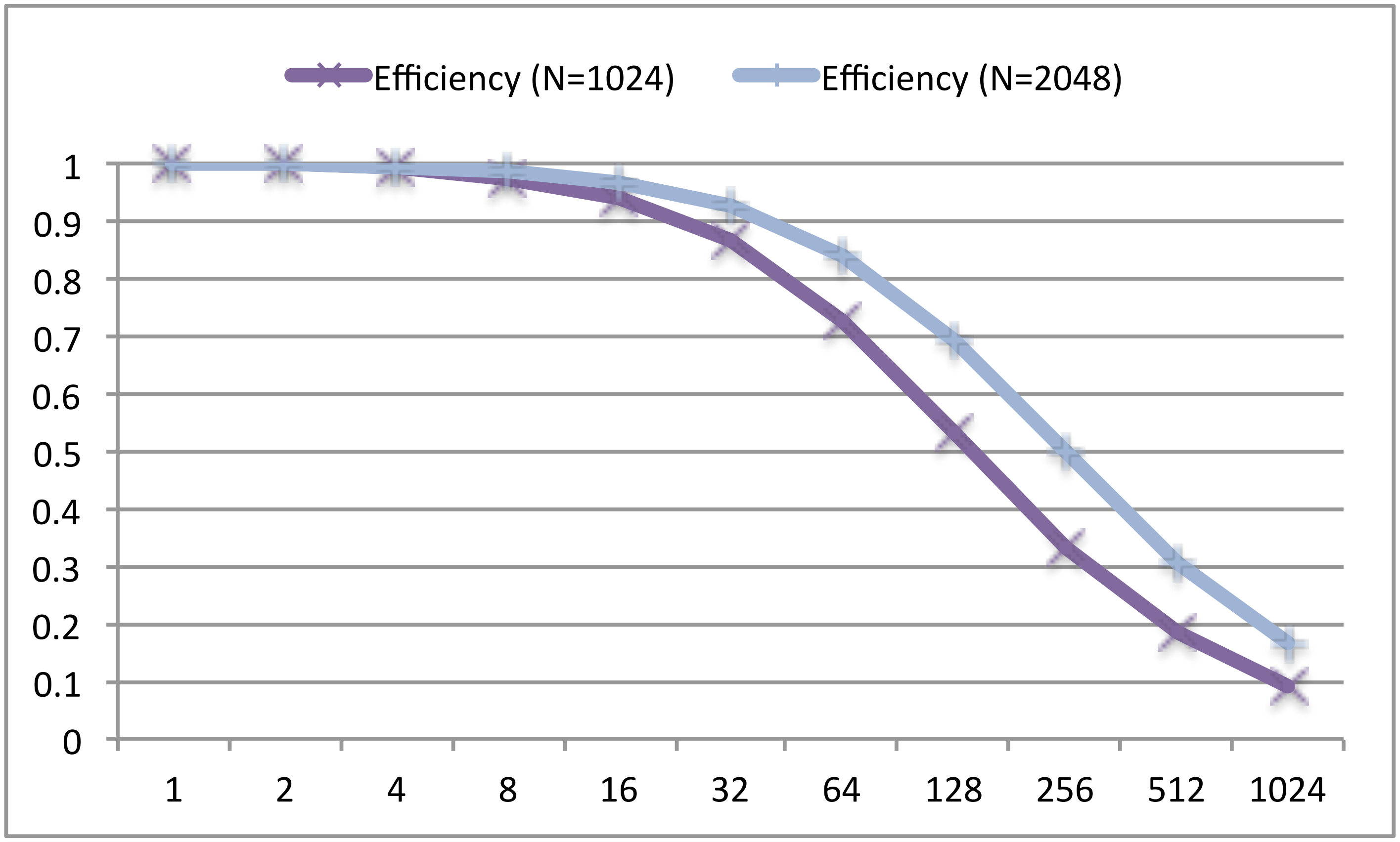
P

Efficiency Metrics

- 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
  - $N_{1/2}$ = 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 <u>must be executed sequentially</u> for a given input size N, then the best speedup that can be obtained for that program is Speedup(N,P) $\leq 1/q$.
- Observation follows directly from critical path length lower bound on parallel execution time
  - CPL >= q * T(N,1)
  - T(N,P) >= q * T(N,1)
  - Speedup(N,P) = T(N,1)/T(N,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

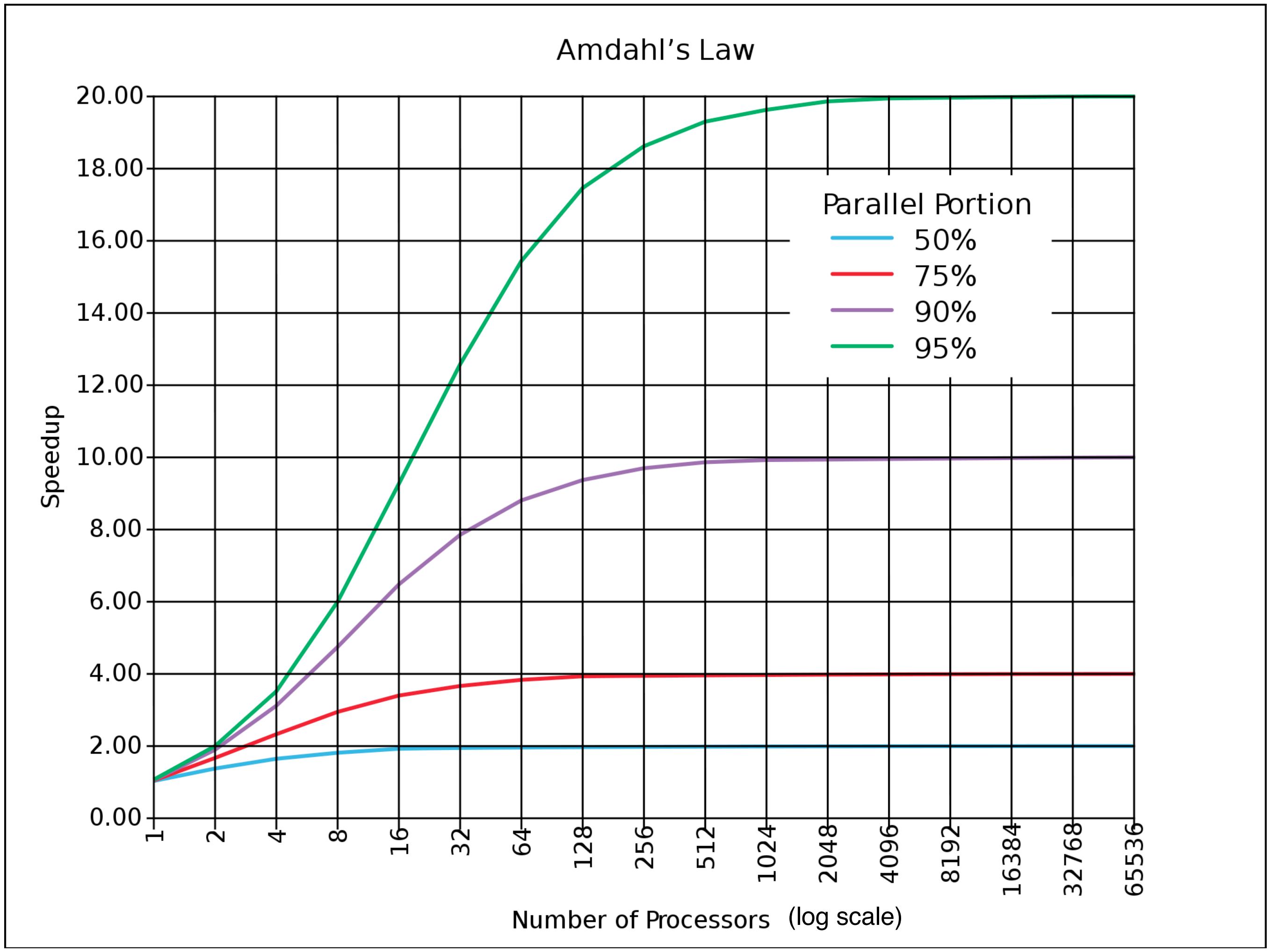


Figure source: http://en.wikipedia.org/wiki/Amdahl's law

Outline of Today's Lecture

- Parallel Array Sum (contd)
- Speedup, Efficiency, Amdahl's Law
- <u>Understanding Data and Control Flow between an Async Task and its Parent</u>
- Data Races and Determinism

How can an Async Task interact with its Parent Task?

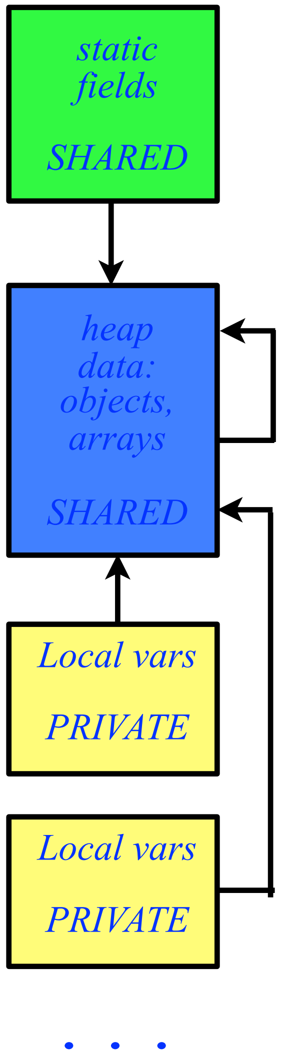
- Data flow
  - Async task can read from static fields, objects, arrays, and local variables written by parent task
    - Same rule as method calls, except that parent's local variables are passed as <u>implicit</u> parameters
  - Async task can write to static fields, objects, arrays (but not parent's local variables) to be read by parent task after end-finish
    - Same rule as method calls, except that method calls also have return values
    - We will learn soon about an extension to asyncs with return values (futures)
- Control flow
  - Async task can execute a return statement (different from method return)
  - Async task can throw an exception
  - NOTE: break/continue cannot cross async boundaries

Shared and Private data in Java's Storage Model

Java's storage model contains three memory regions:

1. Static Data: region of memory reserved for variables that are not allocated or destroyed during a class' lifetime, such as static fields.
   - Static fields can be shared among threads/tasks
2. Heap Data: region of memory for dynamically allocated objects and arrays (created by "new").
   - Heap data can be shared among threads/tasks
3. Stack Data: Each time you call a method, Java allocates a new block of memory called a stack frame to hold its local variables
   - Local variables are private to a given thread/task

All references (pointers) must point to heap data --- no references can point to static or stack data



Data Flow: Use of Static Fields to Communicate Return Value from an Async Tasks (Poor Programming Practice)

```
static int sum1 = 0, sum2 = 0;
public static void main(String[] argv) { // caller
  int[] X = new int[...];
  ... // Initialize X
  int sum;
  finish { // Async's have same access rules as methods
    async for(int i=X.length/2; i < X.length; i++)
            sum2 += X[i];
    async for(int i=0; i < X.length/2; i++)
            sum1 += X[i];
   }
   sum = sum1 + sum2;
   ....
 }
```

Data Flow: Use of an Object to Communicate Return Values from Async Tasks (Preferred Approach)

```
public class TwoIntegers {int sum1; int sum2;}
. . .
 public static void main(String[] argv) { // caller
 int[] X = new int[...]; ... // Initialize X
 int sum;
 TwoIntegers r = new TwoIntegers();
 finish { // Async's have same access rules as methods
   async for(int i=X.length/2; i < X.length; i++)
           r.sum2 += X[i];
   async for(int i=0; i < X.length/2; i++)
           r.sum1 += X[i];
  }
  sum = r.sum1 + r.sum2;
  ....
}
```

Control Flow: Semantics of HJ return statement

- Java semantics for return
  - Return from enclosing method
- HJ semantics for return statement
  - Return from immediately enclosing async or method

```
void foo() {
  if (...) return; // Returns from method foo()
  async { ... return; ... } // Returns from async
  . . .
}
```

Control Flow: Semantics of HJ break and continue statements

- Java semantics for break/continue
  - Perform appropriate action for innermost enclosing loop (or labeled loop)
  - It's an error to execute a break/continue statement without an enclosing loop
- HJ semantics for break/continue
  - It's also an error to execute a break/continue statement in an async without an enclosing loop in the same async
  - Cryptic error message from HJ compiler
    - "Target of branch statement not found"

```
1. void foo() {
2.   while (...) {
3.     async {
4.       while (...) { ... break; ... } // Okay
5.       break; // Error --- does not relate to while loop in line 2
6.     } } }
```

Examples of Common Errors made by beginner HJ Programmers

```
finish for (int i = 0; i <= N - M
  int j;
  async {
    for (j = 0; j < M; j++) {
      async {
        if (text[i+j] != pattern[j]) break;
      }
      if (j == M) return i;// found at off
  }
}
```

Async cannot modify local variable in parent's scope

No loop enclosing break in async

Return statement in basic async task cannot take a value

Async-Finish Exception Semantics

- Exceptions thrown by multiple async's are accumulated into a "MultipleExceptions" collection at their Immediately Enclosing Finish

```
try {
  finish for (int i = 0; i < size; i++)
    async {
      // Add explicit ArrayIndexOutOfBoundsException with X[-1]
      X[2*i*step] += X[(2*i+1)*step] + X[-1];
    } // finish-for-async
  } // try
catch (Throwable t) {
   if (t instanceof hj.lang.MultipleExceptions)
     ... // Process the collection, t.exceptions
   else // single exception
     ... // Process t
}
```

Outline of Today's Lecture

- Parallel Array Sum (contd)
- Speedup, Efficiency, Amdahl's Law
- Understanding Data and Control Flow between an Async Task and its Parent
- <u>Data Races and Determinism</u>

Example of Incorrect Parallel Program in Homework 0 (Problem 1.2)

```
1.  // Sequential version
2.  for ( p = first; p != null; p = p.next) p.x = p.y + p.z;
3.  for ( p = first; p != null; p = p.next) sum += p.x;
4.
5.  // Incorrect parallel version
6.  for ( p = first; p != null; p = p.next)
7.        async p.x = p.y + p.z;
8.  for ( p = first; p != null; p = p.next)
9.        sum += p.x;
```

Why is the parallel version incorrect?

Data race between write of p.x in line 7 and read of p.x in line 9 !

Formal Definition of Data Races

Formally, a data race occurs on location L in a program execution with computation graph CG if there exist steps (nodes) S1 and S2 in CG such that:

1. S1 does not depend on S2 and S2 does not depend on S1 i.e., there is no path of dependence edges from S1 to S2 or from S2 to S1 in CG, and
2. Both S1 and S2 read or write L, and at least one of the accesses is a write.

Data races are challenging because of

- Nondeterminism: different executions of the parallel program with the same input may result in different outputs.
- Debugging and Testing: it is usually impossible to guarantee that all possible orderings of the accesses to a location will be encountered during program debugging and testing.

Relating Data Races and Determinism

- A parallel program is said to be deterministic with respect to its inputs if it always computes the same answer when given the same inputs.
- Structural Determinism Property
  - If a parallel program is written using the constructs in Module 1 and is guaranteed to be race-free, then it must be deterministic with respect to its inputs. The final computation graph is also guaranteed to be the same for all executions of the program with the same inputs.
- Constructs introduced in Module 1 ("Deterministic Shared-Memory Parallelism") include async, finish, finish accumulators, futures, data-driven tasks (async await), forall, barriers, phasers, and phaser accumulators.
  - The notable exceptions are critical sections, isolated statements, and actors, all of which will be covered in Module 2 ("Nondeterministic Shared-Memory Parallelism")

Worksheet #1: Complexity analysis of k-way Parallel Array Sum algorithm

Your name: ______________________________

- Consider a k-way parallel array-sum algorithm, where 1 <= k <= n
  - Compute k partial sums in parallel, each of size n/k
  - Sequentially combine the k partial sums into a single sum
- Total number of additions, WORK = k (n/k -1) + k = O(n)
- What is the critical path length?
  - CPL =
- What value of k gives the smallest value of CPL?
  - Optimal value of k =