COMP 322: Fundamentals of Parallel Programming

Lecture 9: Parallel Random Access Machine (PRAM) Computation Model

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

Announcements

- Next week's lectures on Feb 9th & 11th (Wed & Fri) will be given by John Mellor-Crummey
- Homework 3 is due by 5pm on Monday, Feb 7th
  - This is a programming assignment with abstract performance metrics
  - To prepare for HW3, please make sure that you can compile and run the programs from Lab 2 on your own, using the –perf option. In case of problems, please send email to comp322-staff @ mailman.rice.edu
- We have requested 24-hour access to Ryon building and Ryon 102 lab for all students enrolled in COMP 322
- Preferred naming convention for homework folders in clear is hw_?? e.g. hw_3
  - Please try and use this convention in the future

Acknowledgments for Today's Lecture

- Michael J. Quinn. Parallel computing (2nd ed.): theory and practice. McGraw-Hill, Inc., New York, NY, USA, 1994. ISBN 0-07-051294-9
- "Introduction to Parallel Computing", 2nd Edition, Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar, Addison-Wesley, 2003
- COMP 322 Lecture 9 handout

Introduction

- Rich set of theoretical results obtained for sequential algorithms by using a simplified abstraction of hardware, the Random Access Machine (RAM)
  - Implementation of sequential RAM algorithms usually work as advertised i.e., execution times on real machines usually follow trends predicted by big-O complexity analysis
- The PRAM model (pronounced “P RAM”) also led to a rich set of parallel algorithms by using a simplified abstraction of parallel hardware
  - As we will see, there is a much larger gap between parallel PRAM algorithms and real parallel programs compared to the gap between sequential RAM algorithms and real sequential programs

Random Access Machine (RAM) model for sequential algorithms

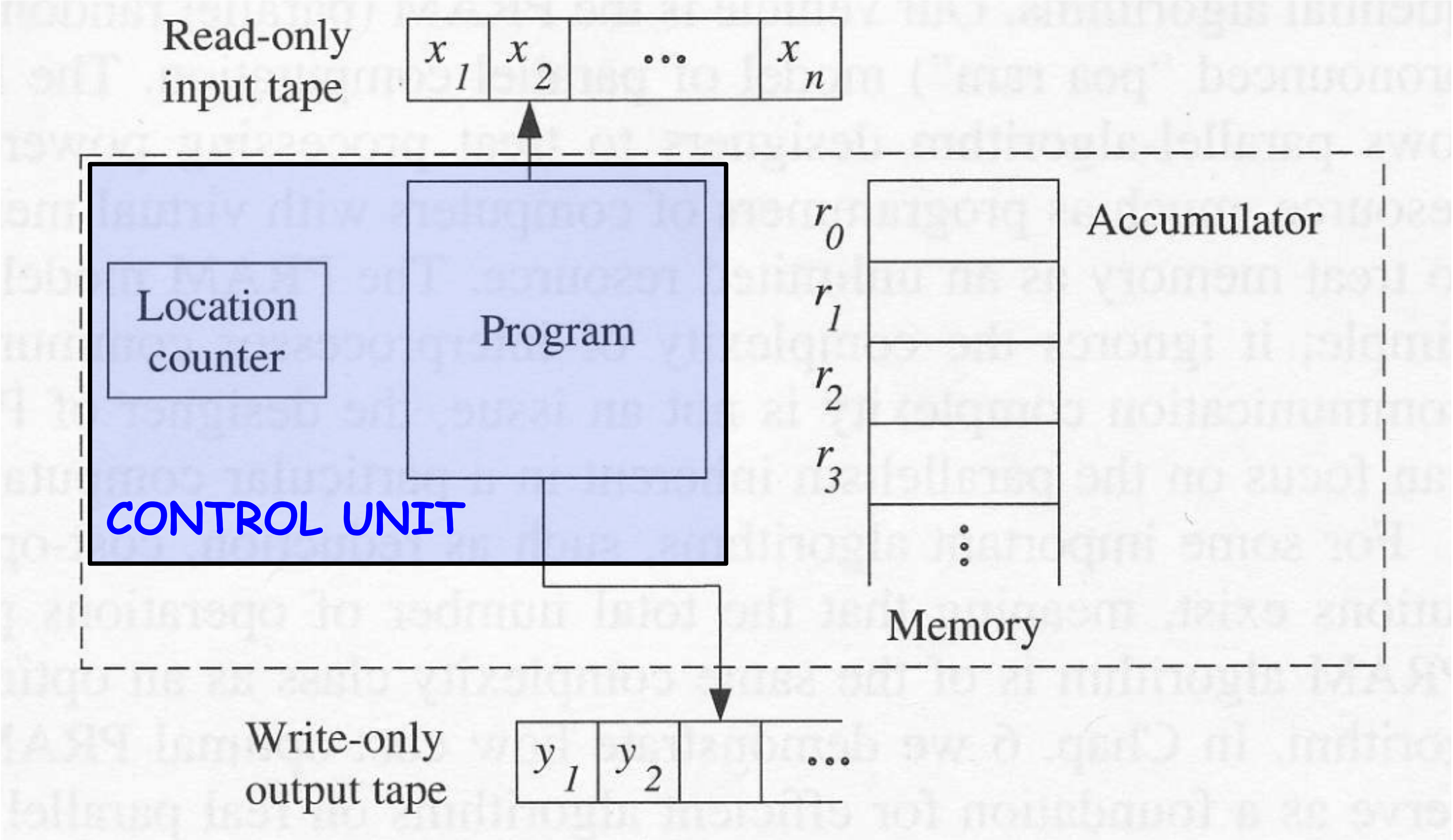


Figure source: Figure 2-1 in [Quinn 1994]

Key Features of sequential RAM model

- Input of size *n* read from *input tape*
- Output written to *output tape*
- *Control unit* consisting of *Program* + *Location counter*
  - Program cannot be modified
- Randomly accessible *memory* of unbounded size
- Time complexity = number of constant-time statements/ instructions executed by program
- Space complexity = maximum number of constant-sized memory locations used during program execution ("high water mark")
- Big-O analysis used to model time and space complexity

Parallel Random Access Machine (PRAM) model

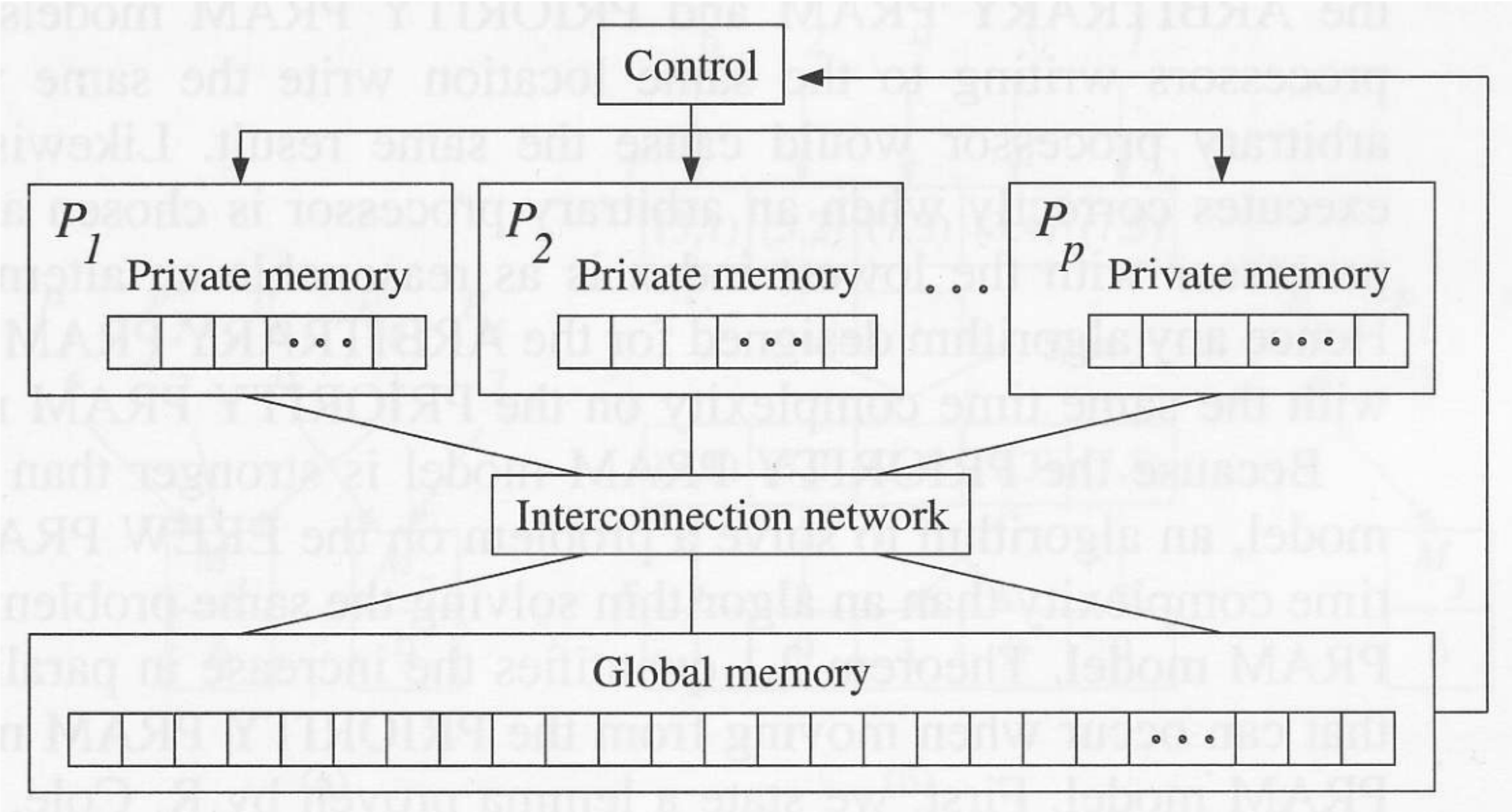


Figure source: Figure 2-2 in [Quinn 1994]

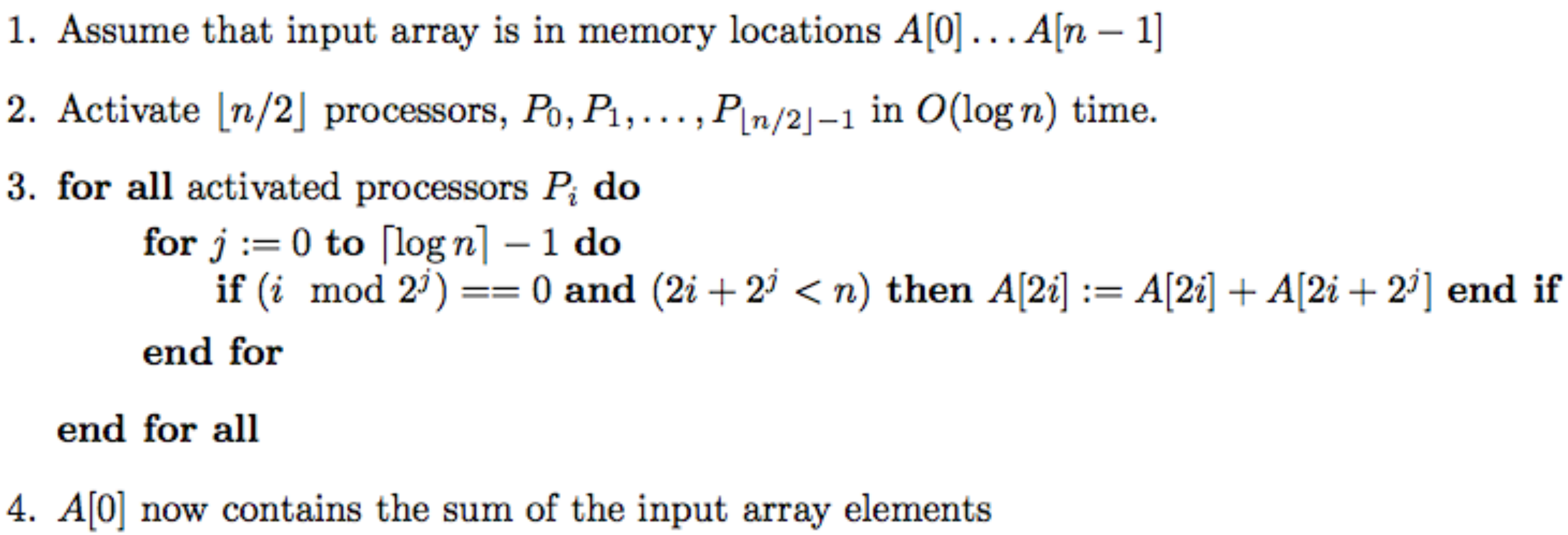
Key Features of PRAM model

- Input placed in Global memory at start of program execution
- Output placed in Global memory at end of program execution
- Unbounded number of processors, each with Private memory
  - Processors need to be explicitly *activated*
- *Single control unit* for all processors
  - all processors execute the same program statement issued by the control unit at the same time in lock step.
  - each processor has a distinct *index* which can be used as an operand in the statements that it executes
- Time complexity = number of constant-time statements executed by control unit
- Space complexity = maximum number of constant-sized memory locations used across Global memory and Private memories during program execution
- Big-O analysis used to model time and space complexity

PRAM program execution

- Each constant-time statement issued by the Control unit is called a *step*
- Each active processor executes a step as follows
  a) Copy constant number of locations from global memory to private memory
  b) Perform computation by executing a constant number of RAM instructions on private memory
  c) Conditionally copy a constant number of locations from private memory to global memory
- Synchronous execution:
  - *Implicit finish after each step.* No active processor will start next step until all active processors have completed previous step.
  - *Implicit finish* after read/compute/write portions within a step (items a), b), c) above)
- Example of a constant-time statement
  - if A[i] != 0 then B[i] := 1/A[i];
  - Each active processor P_i will access distinct global memory locations (A[i], B [i]) due to use of processor index *i*

PRAM algorithm for Array Sum



```
1. Assume that input array is in memory locations $A[0] \ldots A[n-1]$

2. Activate $\lfloor n/2 \rfloor$ processors, $P_0, P_1, \ldots, P_{\lfloor n/2 \rfloor - 1}$ in $O(\log n)$ time.

3. for all activated processors $P_i$ do
        for $j := 0$ to $\lceil \log n \rceil - 1$ do
            if $(i \mod 2^j) == 0$ and $(2i + 2^j < n)$ then $A[2i] := A[2i] + A[2i + 2^j]$ end if
        end for

   end for all

4. $A[0]$ now contains the sum of the input array elements
```

- Consider translating step 3 to HJ by using an outer parallel forall and inner sequential for loop
  - Would it be a correct translation of the PRAM algorithm?

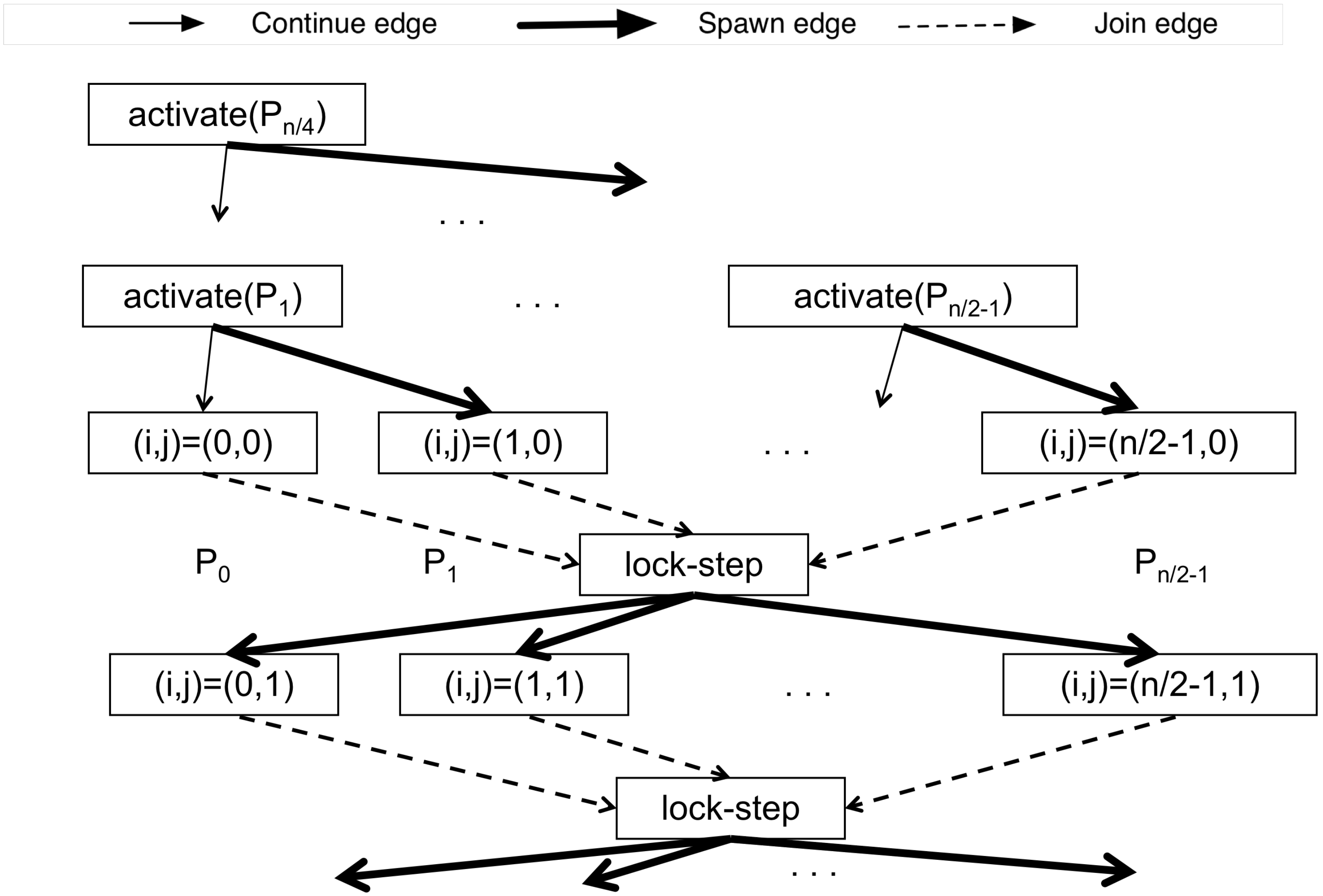
Direct translation of PRAM Array sum algorithm to HJ task-parallel program

```
forall (point[i] : [0:n/2-1]) {
  for (point[j] : [0:ceilLog2(n)-1]) {
    int exp2j = 1<<j;
    if (i % exp2j == 0 && 2*i+exp2j < n)
      A[2*i] = A[2*i] + A[2*i+exp2j]
  } // for
} // forall
static int ceilLog2(int n) { // returns 0 if n <= 0
  int r=0; while (n > 1) { r++; n = n >> 1; } return r;
}
```

Is there a data race in this program?

If so, why was the PRAM algorithm correct?

PRAM model has implicit finish after each step



Correct translation of PRAM Array sum algorithm to HJ task-parallel program

```
1. for (point[j] : [0:ceilLog2(n)-1]) {
2.   forall (point[i] : [0:n/2-1]) {
3.     int exp2j = 1<<j;
4.     if (i % exp2j == 0 && 2*i+exp2j < n)
5.       A[2*i] = A[2*i] + A[2*i+exp2j]
6.   } // for
7. } // forall
```

- Moving the forall loop inside the for loop inserts implicit finish after each step (lines 3, 4, 5)
- Think of a PRAM program as sequential at the outer level, while executing each step as a forall loop across all processors

Conflicting Memory Operations in PRAM model

- No conflict possible between global memory operations in different steps – why?
- No conflicts possible between read and write global memory operations in same step – why?
- No conflicts possible on read/write operations on private memory – why?
- This only leaves the possibility of conflicting write operations on global memory in the same step

Variants of PRAM model

- EREW (Exclusive Read Exclusive Write): No read or write conflicts are permitted on global memory in same step.
- CREW (Concurrent Read Exclusive Write): Multiple active processors may read from the same location in the same step, but only one active processor may write to a given location in one step. Default assumption in the PRAM model.
- CRCW (Concurrent Read Concurrent Write): Multiple active processors may read from or write to the same location in the same step. Different policies for conflicting writes:
  - Common CRCW rule: All conflicting writes must write the same common value. Deterministic output.
  - Arbitrary CRCW rule: If multiple processors write to the same global location in the same step, one of the values is arbitrarily chosen. Nondeterministic data race among atomic writes.
  - Priority CRCW rule: If multiple processors write to the same global location in the same step, then the value provided by the processor with the lowest index is chosen as the winner. Deterministic output.

Strengths of different PRAM models

- Model B is said to be stronger than model A if a program written for A can run unchanged on model B with the same or smaller execution time and space relative to its execution on model A.
- PRAM variants on previous slide were listed in order of increasing strength, with Priority CRCW being the strongest.
- Bound on impact of models if we try to run a program written for Priority CRCW PRAM on an EREW PRAM
  - A p-processor Priority CRCW PRAM program can be executed on a p-processor EREW PRAM model with an increase in execution time complexity of at most an O(log p) factor

Amdahl's Law [1967]

- If $q \le 1$ is the fraction of WORK in a parallel program that must be executed sequentially, then the best speedup that can be obtained for that program is Speedup $\le 1/q$.
- Observation follows directly from critical path length lower bound on parallel execution time, $t_P \ge CPL(G)$
- If fraction q of WORK is sequential then $CPL(G) \ge qWORK$
- Therefore, Speedup $= t_1/t_P$ must be $\le WORK/(qWORK) = 1/q$

- Sequential portion of WORK = q (also denoted as f_S sometimes)
- Parallel portion of WORK = 1-q (also denoted as f_p sometimes)

Illustration of Amdahl's Law: Best Case Speedup as function of Parallel Portion

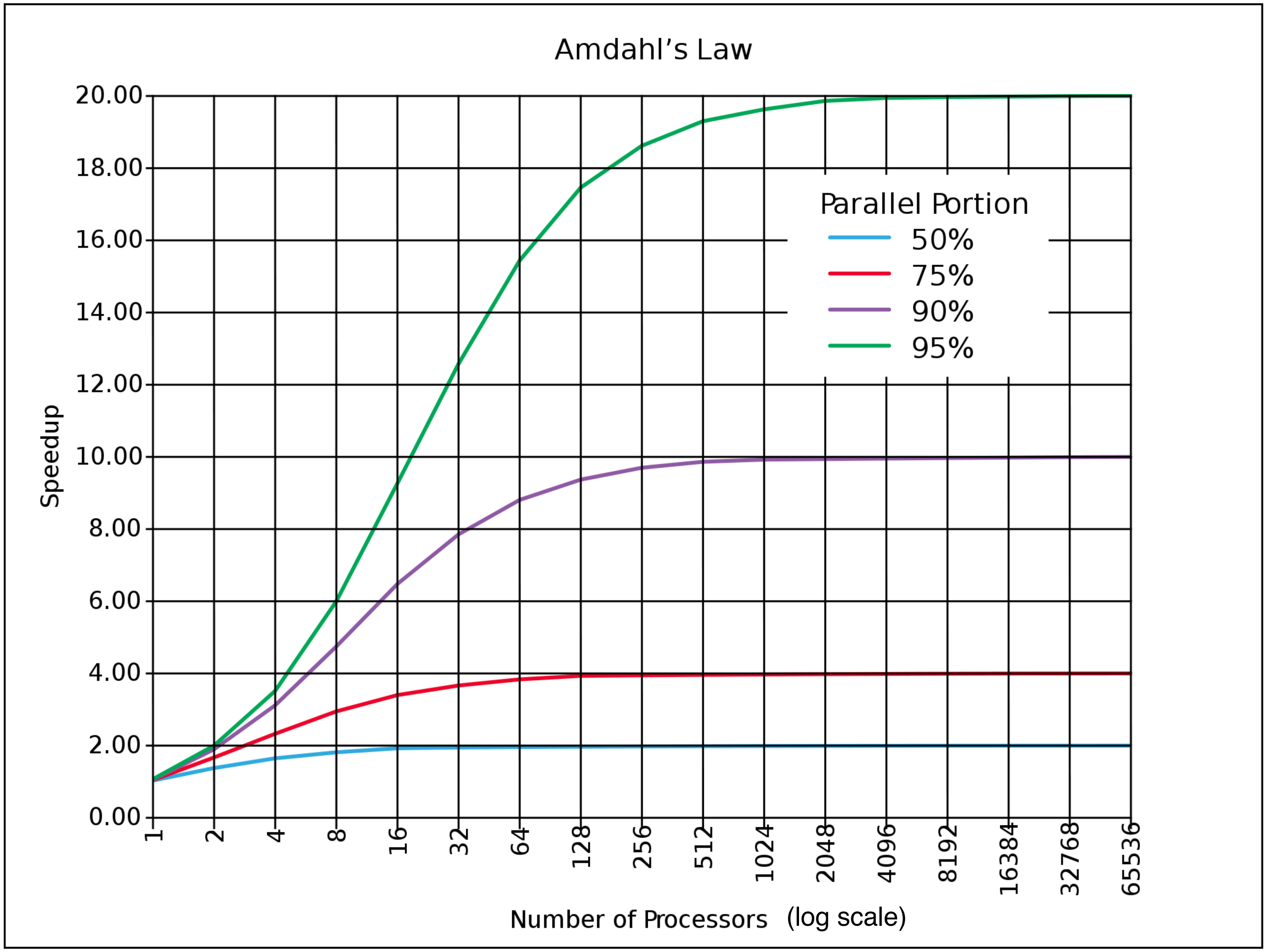
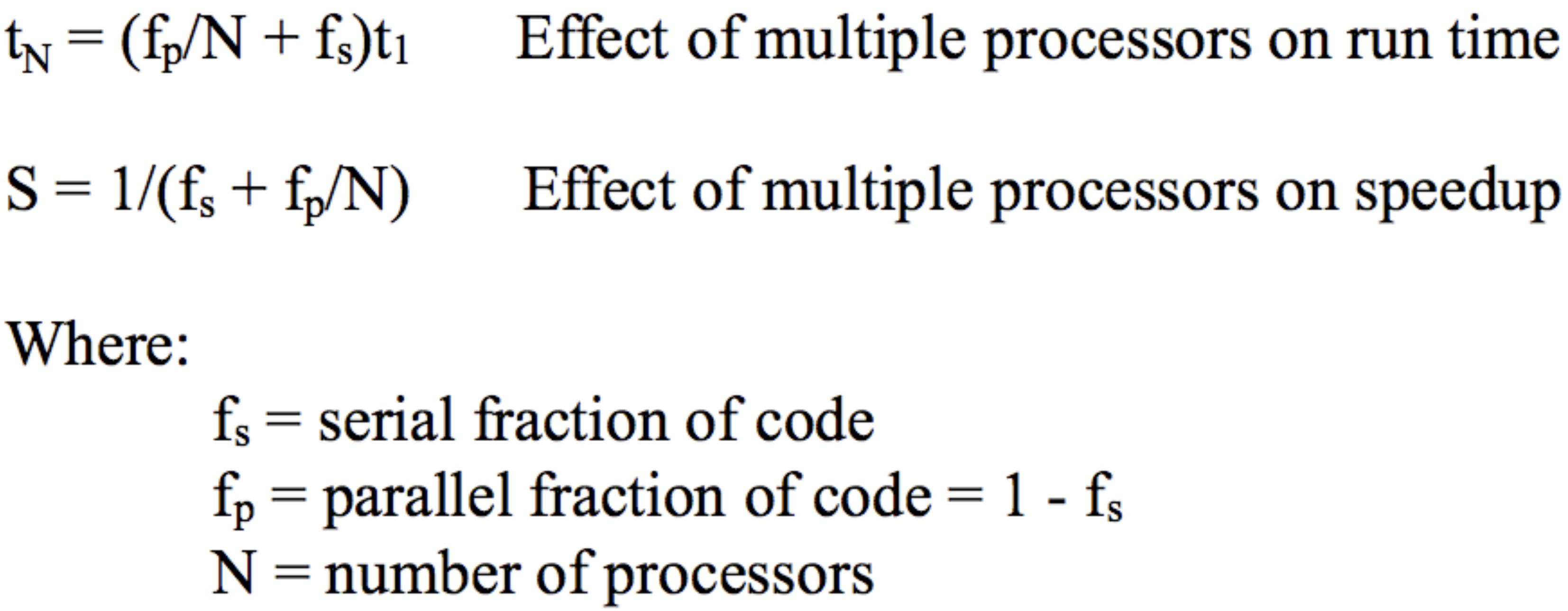


Figure source: http://en.wikipedia.org/wiki/Amdahl's law

Amdahl’s Law: Alternate Formulation

Amdahl’s Law places a strict limit on the speedup that can be realized by using multiple processors. Two equivalent expressions for Amdahl’s Law are given below:



$t_N = (f_p/N + f_s)t_1$ Effect of multiple processors on run time

$S = 1/(f_s + f_p/N)$ Effect of multiple processors on speedup

Where:

f_s = serial fraction of code

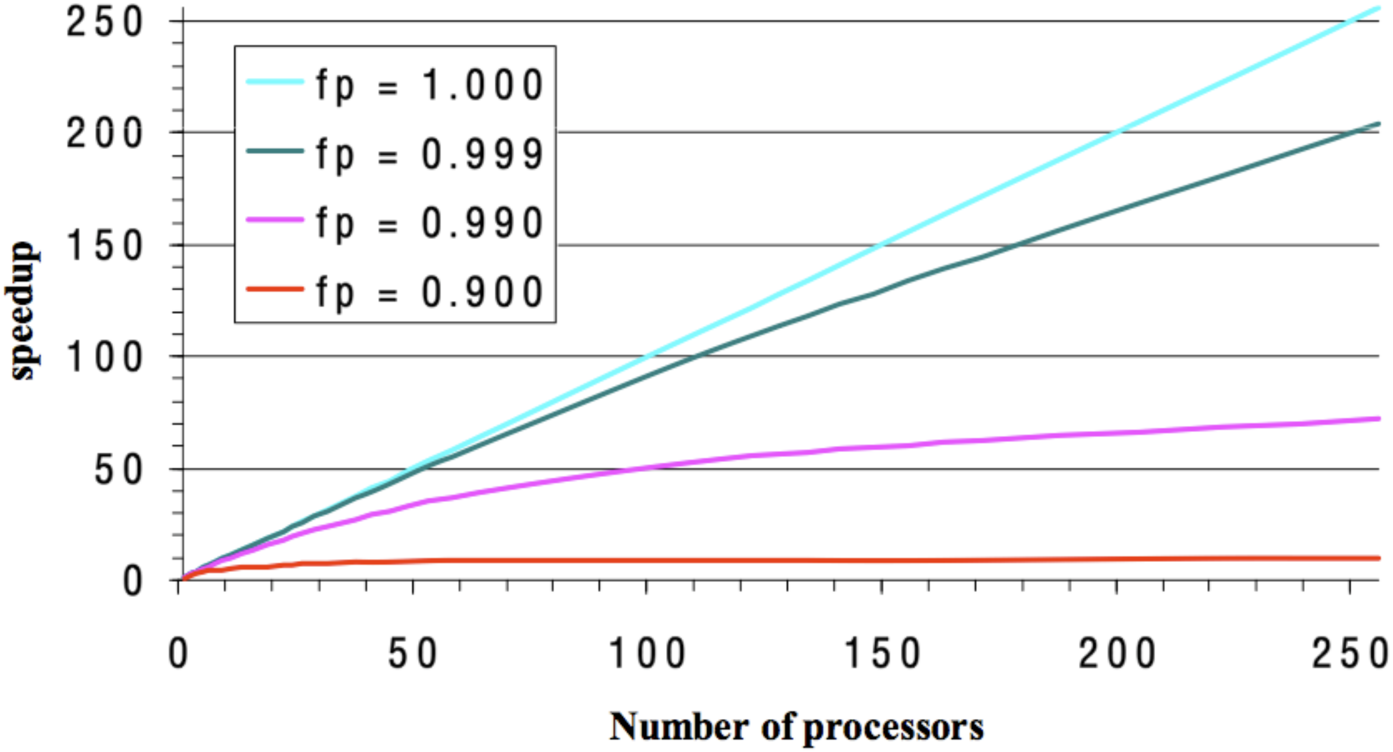
f_p = parallel fraction of code = 1 - f_s

N = number of processors

Source: “Introduction to Parallel Computing”, 2nd Edition, Grama et al, Addison- Wesley, 2003

Another Illustration of Amdahl's Law

It takes only a small fraction of serial content in a code to degrade the parallel performance. It is essential to determine the scaling behavior of your code before doing production runs using large numbers of processors



Source: "Introduction to Parallel Computing", 2nd Edition, Grama et al, Addison- Wesley, 2003