Trade-offs in Parallel Programming

Vivek Sarkar
Department of Computer Science
Rice University

Quicksort with Parallel Tasks (Recap from Lecture 34 and Lab 14)

```
public static ArrayList<Integer> quickSort(ArrayList<Integer> a) {
     if (a.isEmpty()) return new ArrayList<Integer>();
     ArrayList<Integer> left = new ArrayList<Integer>();
     ArrayList<Integer> mid = new ArrayList<Integer>();
     ArrayList<Integer> right = new ArrayList<Integer>();
     int pivot = a.get(a.size()/2); // Use midpoint element as pivot
     for (Integer i : a)
         if ( i < a.get(0) ) left.add(i); // Use element 0 as pivot
         else if ( i > a.get(0)) right.add(i);
         else mid.add(i)
     // Now, left, mid, right contain the three partitions of
     // array a with respect to pivot
     // Continue on next slide ...
```

Quicksort with Parallel Tasks (Recap from Lecture 34 and Lab 14)

```
  FutureTask<ArrayList<Integer>> left_t = // Closure for recursive call
    new FutureTask<ArrayList<Integer>>(
      new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(left); } } );
  FutureTask<ArrayList<Integer>> right_t = // Closure for recursive call
    new FutureTask<ArrayList<Integer>>(
      new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(right); } } );
  // Execute each closure in a parallel thread
  new Thread(left_t).start(); new Thread(right_t).start();
  // Wait for result of FutureTask's left_t and right_t
  ArrayList<Integer> left_s = left_t.get(); // Sorted version of left
  ArrayList<Integer> right_s = right_t.get(); // Sorted version of right
  return left_s.addAll(mid).addAll(right_s);
} // quickSort
```

What were your experiences with this example in Lab 14?

- How much does the sequential execution time increase due to addition of closures?
  - 3% - 5% may be typical e.g., 4.3 seconds to 4.5 seconds for an array with 2,000,000 elements
- What happens if you run the parallel version on a large array (e.g., 2,000,000 elements)?
  - java.lang.OutOfMemoryError is typical
  - Why does the parallel version need more memory than the sequential version?
- What happens if you only use two threads at the outermost level?
  - Some reduction in execution time is typical e.g., 4.5 seconds to 3 seconds
  - Why is it not reduced by a factor of 2 on a 2-core machine?
- Other issues? e.g., variations in execution times due to JIT compilation

Why does sequential execution time increase with use of closures?

```
FutureTask<ArrayList<Integer>> left_t = // Closure for recursive call
    new FutureTask<ArrayList<Integer>>(
      new Callable<ArrayList<Integer>>() {
        public ArrayList<Integer> call() { return quickSort(left); } } );
. . .
ArrayList<Integer> left_s = left_t.get(); // Sorted version of left
```

- Extra overhead in allocating Callable and FutureTask objects
- Extra overhead in get() operation on FutureTask
- Impact of overhead depends on task granularity i.e., amount of work being done inside FutureTask
- Impact is not signficant (on average) for quickSort() method

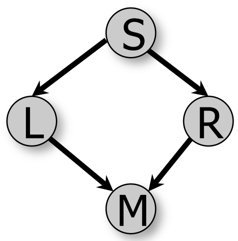
Why does the fully parallel version run out of memory?

```
// Execute each closure in a parallel thread
new Thread(left_t).start(); new Thread(right_t).start();
```

- Each new thread allocates space for a thread stack (typically, 256KB – 512KB by default)
- How many threads (approximately) are created when sorting an array with 2,000,000 elements?
- Also, when can space for intermediate arrays and closures be reclaimed (garbage collected) in sequential vs. parallel versions?

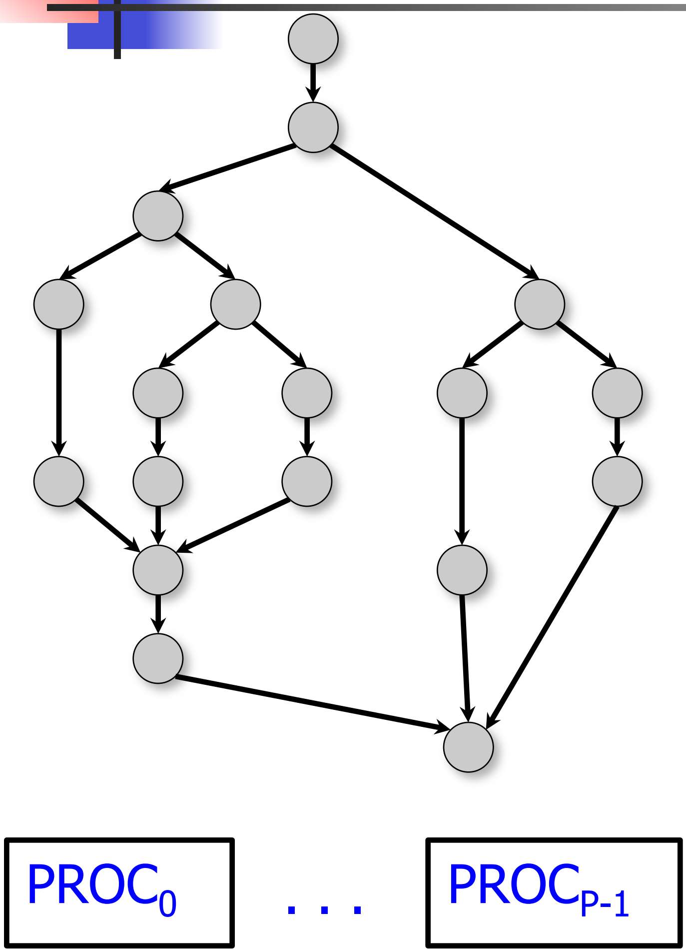
Why does the 2-thread version not speed up execution time by 2x on 2 cores?

- Impact of overhead
  - Parallel version does more work (executes more instructions in total) than sequential version due to creation of closures and threads
- Impact of serialization
  - Top-level quickSort() has four parts
    - S: Start program and split array
    - L: Recursively sort left subarray
    - R: Recursively sort right subarray
    - M: Merge subarrays and end program



  - What would be the "ideal" speedup if all four parts took the same time?

Computation Graph Abstraction



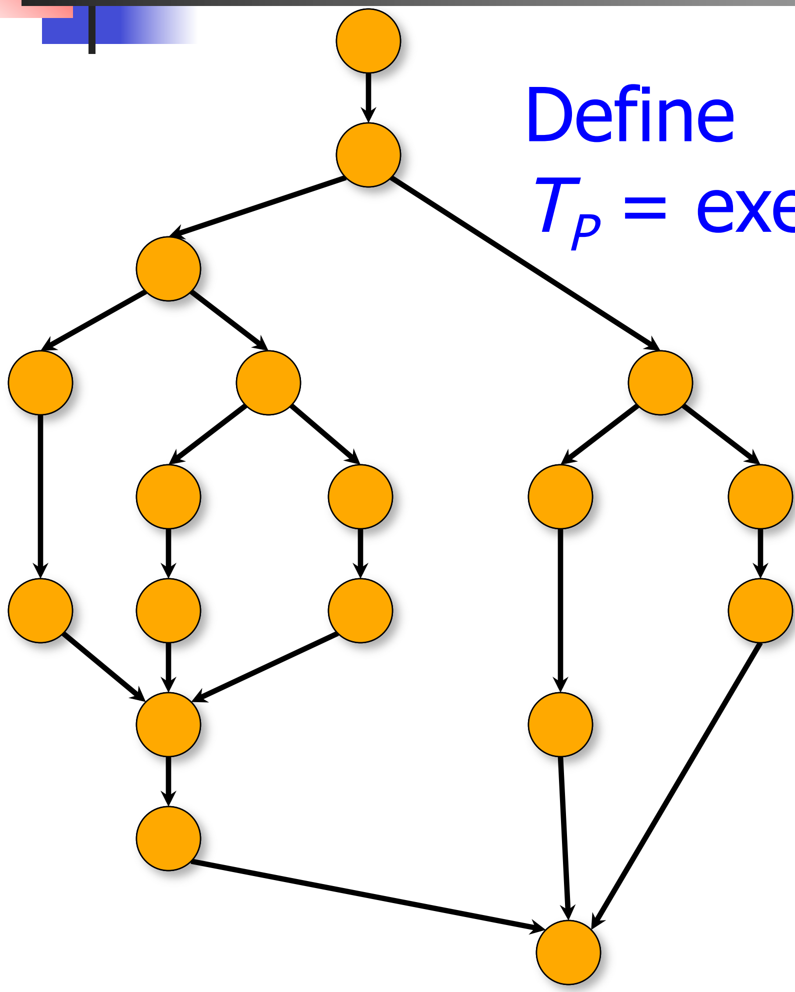
Computation graph abstraction:

- *Node = arbitrary sequential computation*
- *Edge = dependence (successor node can only execute after predecessor node has completed)*

Processor abstraction:

- *P identical processors*
- *Each processor executes one node at a time*

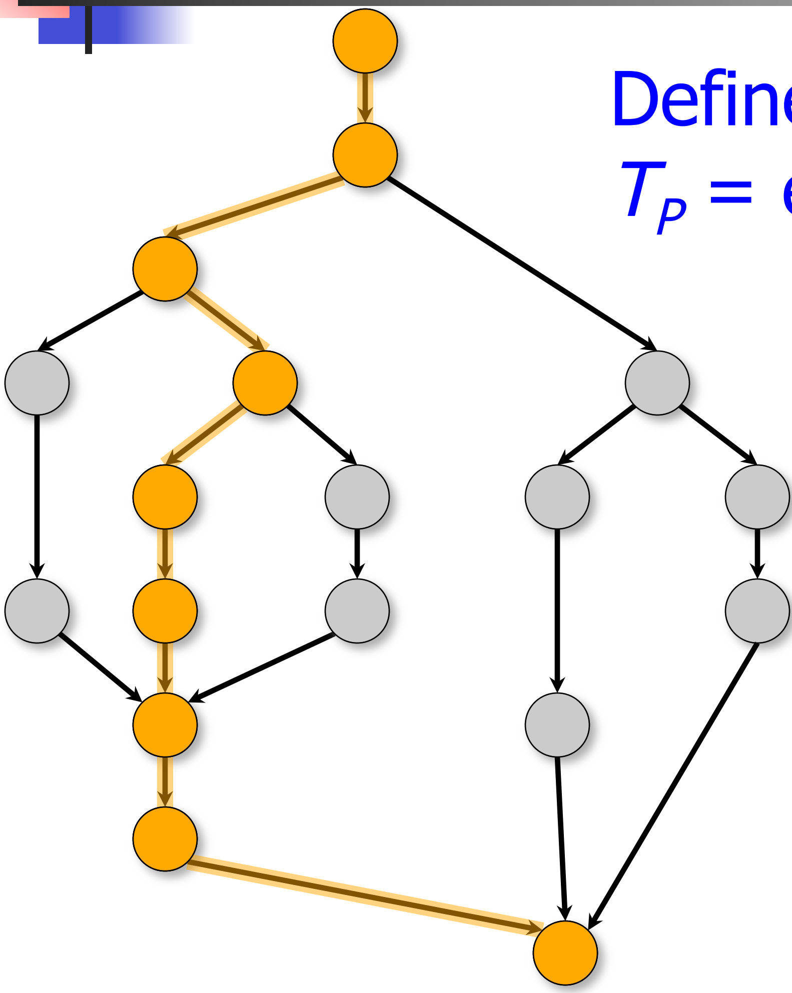
Parallel Execution Time



Define
T_P = execution time on P processors

Therefore,
T_1 = *total work*

Computational Depth



Define

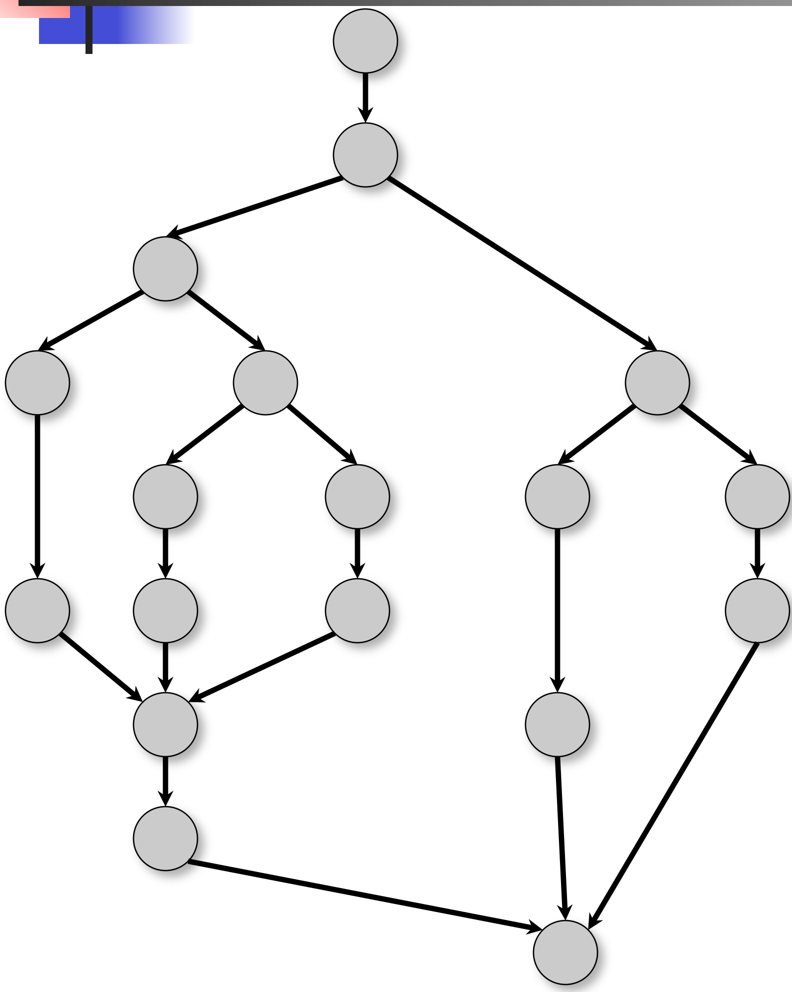
T_P = execution time on P processors

If you had an infinite (unbounded) number of cores, then the execution time will be determined by the length of the longest path from start to finish,

T_∞ = computational *depth**

* Also called *critical-path length*

Best-case Lower Bounds on Parallel Execution Time



T_1 = *work*

T_∞ = *depth*

Lower Bounds

- $T_P \geq T_1/P$
- $T_P \geq T_\infty$

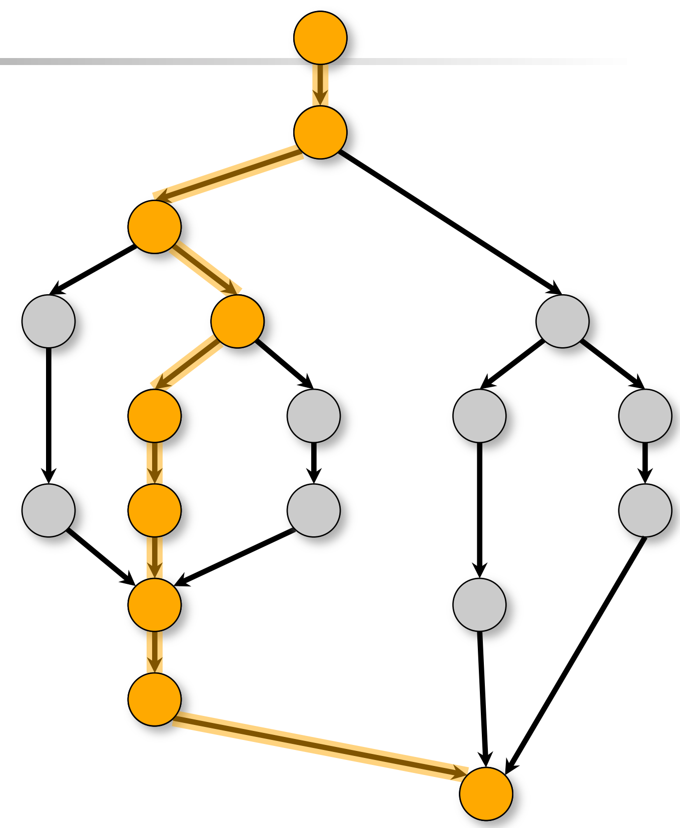
Parallelism ("Ideal Speedup")

T_P depends on the schedule of computation graph nodes on the processors

➔ Two different schedules can yield different values of T_P for the same P

For convenience, define *parallelism* (or *ideal speedup*) as the ratio T_1/T_∞

Parallelism is independent of P, and only depends on the computation graph

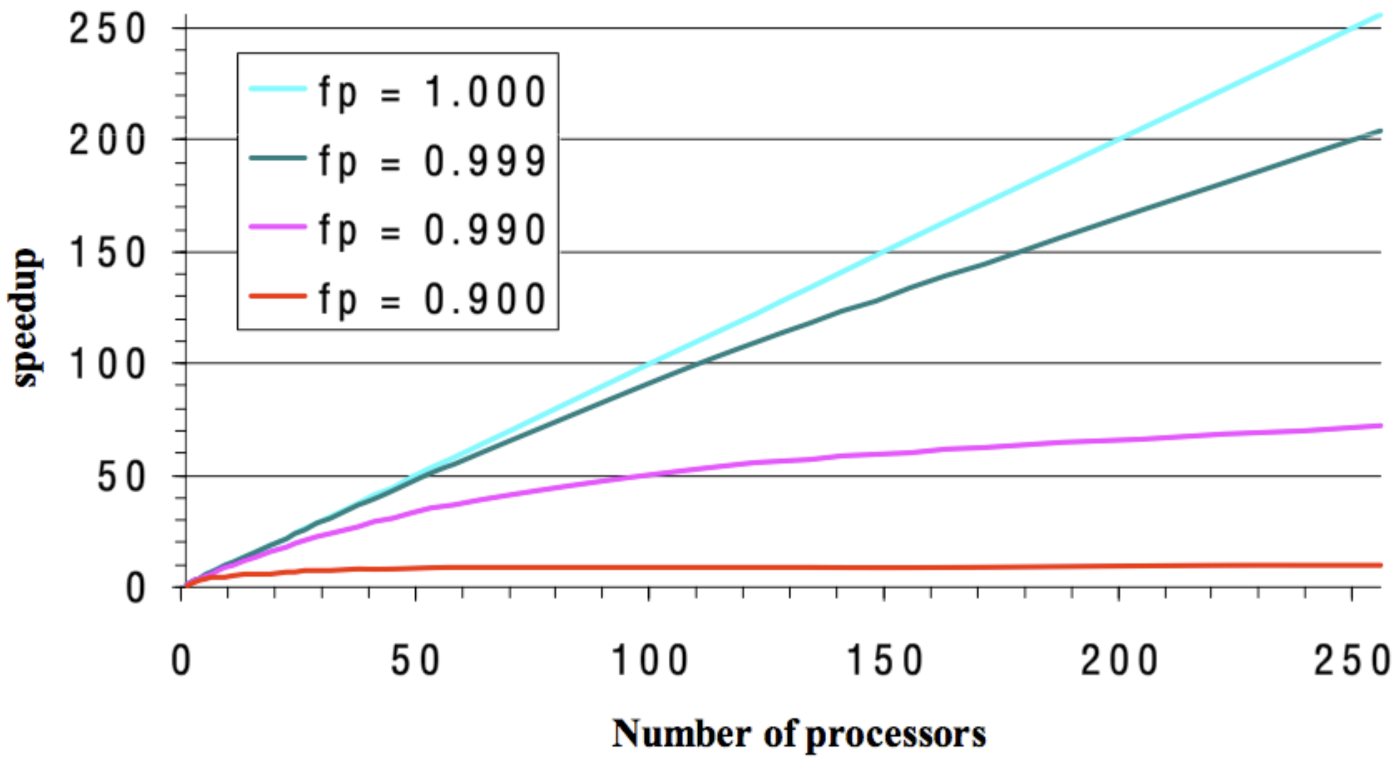


Amdahl's Law

- Consider a program in which f_S is the fraction of work that must be executed sequentially.
- Let T_1 be the total amount of work in the program
- Then, in the best case, the parallel execution time must be at least the sum of
  - $f_S * T_1$ (for the sequential part), and
  - $(1- f_S) * T_1 / P$ (for the parallel part)

Amdahl's Law (contd)

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



Summary of Today's Lecture

- Trade-offs in Parallel Programming
  - Overhead
  - Memory
  - Serialization
- Computation Graph & Critical Path Length
- Lower bounds and Amdahl's Law

- You can learn more about these topics in COMP 322 and COMP 422!