PARALLEL PERFORMANCE

Performance Considerations

- I parallelized my code and …
- The code slower than the sequential version, or
- I don't get enough speedup

- What should I do?

Performance Considerations

- Algorithmic
- Fine-grained parallelism
- True contention
- False contention
- Other Bottlenecks

Algorithmic Bottlenecks

- Parallel algorithms might sometimes be completely different from their sequential counterparts

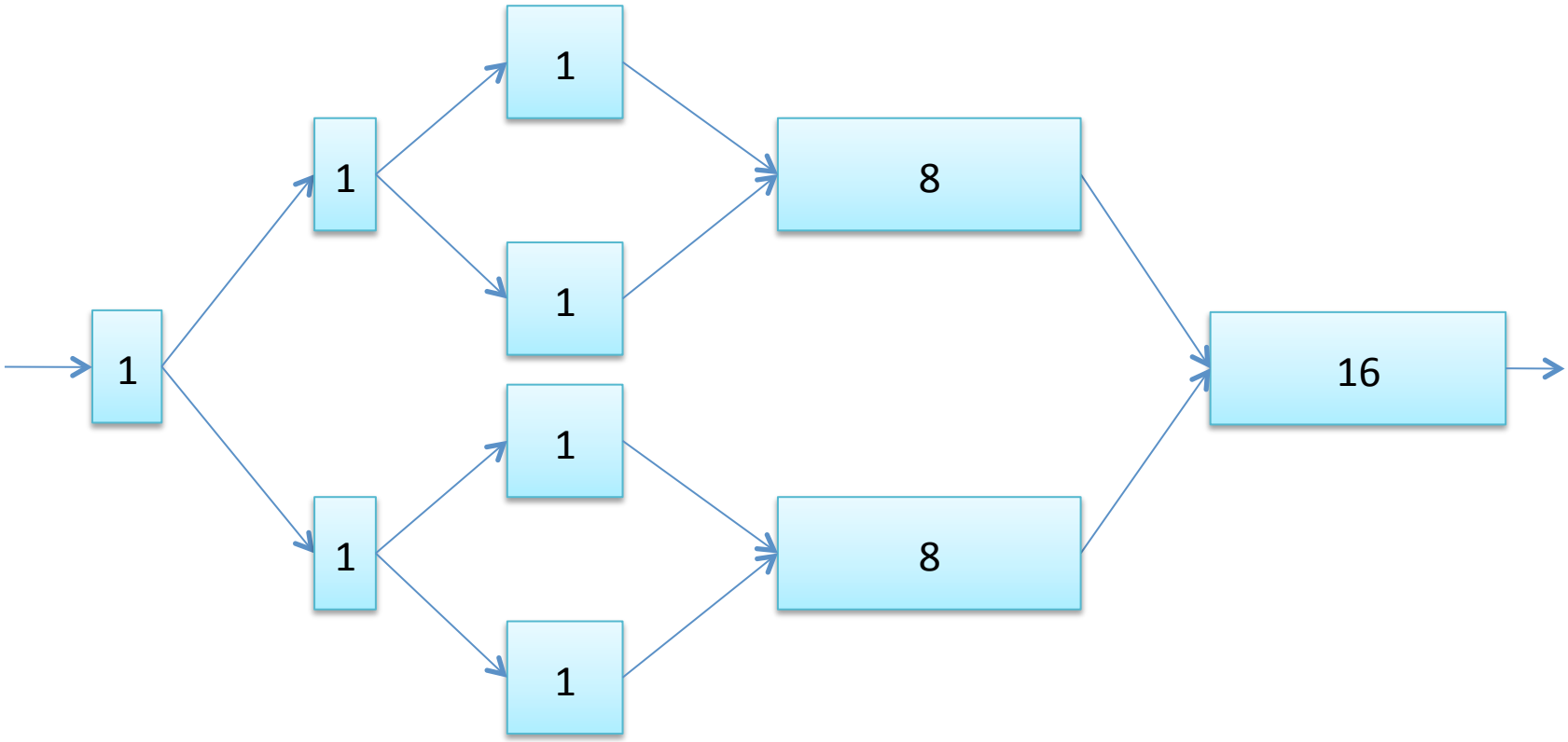
- Might need different design and implementation

Algorithmic Bottlenecks

- Parallel algorithms might sometimes be completely different from their sequential counterparts
- Might need different design and implementation
- Example: MergeSort (once again ☹)

Recall from Previous Lecture

- Merge was the bottleneck



Most Efficient Sequential Merge is not Parallelizable

```
Merge(int* a, int* b, int* result ){

    while( <end condition> ) {
       if(*a <= *b){
          *result++ = *a++;
       }
       else {
          *result++ = *b++;
       }
    }

}
```

Parallel Merge Algorithm

- Merge two sorted arrays A and B using divide and conquer

Parallel Merge Algorithm

- Merge two sorted arrays A and B

1. Let A be the larger array (else swap A and B)
2. Let n be the size of A
3. Split A into A[0…n/2-1], A[n/2], A[n/2+1…n]
4. Do binary search to find smallest m such that

   B[m] >= A[n/2]
5. Split B into B[0...m-1], B[m,..]
6. return Merge(A[0…n/2-1], B[0…m-1]), A[n/2],

   Merge(A[n/2+1…n], B[m…])

Assignment 1 Extra Credit

- Implement the Parallel Merge algorithm and measure performance improvement with your work stealing queue implementation

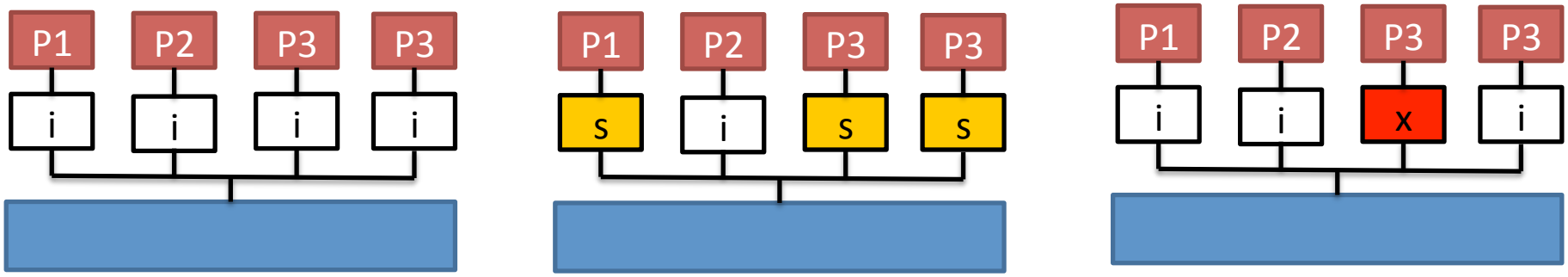
Fine Grained Parallelism

- Overheads of Tasks
  - Each Task uses some memory (not as much resources as threads, though)
  - Work stealing queue operations
- If the work done in each task is small, then the overheads might not justify the improvements in parallelism

False Sharing
Data Locality & Cache Behavior

- Performance of computation depends HUGELY on how well the cache is working
- Too many cache misses, if processors are “fighting” for the same cache lines
- Even if they don’t access the same data

Cache Coherence



- Each cacheline, on each processor, has one of these states:
  - i - invalid : not cached here
  - s - shared : cached, but immutable
  - x - exclusive: cached, and can be read or written
- State transitions require communication between caches (cache coherence protocol)
  - If a processor writes to a line, it removes it from all other caches

Ping-Pong & False Sharing

- Ping-Pong
  - If two processors both keep writing to the same location, cache line has to go back and forth
  - Very inefficient (lots of cache misses)
- False Sharing
  - Two processors writing to two different variables may happen to write to the same cacheline
    - If both variables are allocated on the same cache line
  - Get ping-pong effect as above, and horrible performance

False Sharing Example

```
void WithFalseSharing()
{
    Random rand1 = new Random(), rand2 = new Random();
    int[] results1 = new int[20000000],
          results2 = new int[20000000];
    Parallel.Invoke(
        () => {
            for (int i = 0; i < results1.Length; i++)
                results1[i] = rand1.Next();
        },
        () => {
            for (int i = 0; i < results2.Length; i++)
                results2[i] = rand2.Next();
        });
}
```

False Sharing Example

```
void WithFalseSharing()
{
    Random rand1 = new Random(), rand2 = new Ra
    int[] results1 = new int[20000000],
          results2 = new int[20000000];
    Parallel.Invoke(
        () => {
            for (int i = 0; i < results1.Length
                results1[i] = rand1.Next();
        },
        () => {
            for (int i = 0; i < results2.Length
                results2[i] = rand2.Next();
        });
}
```

rand1, rand2 are allocated at same time => likely on same cache line.

Call to Next() writes to the random object => Ping-Pong Effect

False Sharing, Eliminated?

```
void WithoutFalseSharing()
{
    int[] results1, results2;
    Parallel.Invoke(
        () => {
            Random rand1 = new Random();
            results1 = new int[20000000];
            for (int i = 0; i < results1.Length; i++)
                results1[i] = rand1.Next();
        },
        () => {
            Random rand2 = new Random();
            results2 = new int[20000000];
            for (int i = 0; i < results2.Length; i++)
                results2[i] = rand2.Next();
        });
}
```

rand1, rand2 are allocated by different tasks
=>
Not likely on same cache line.