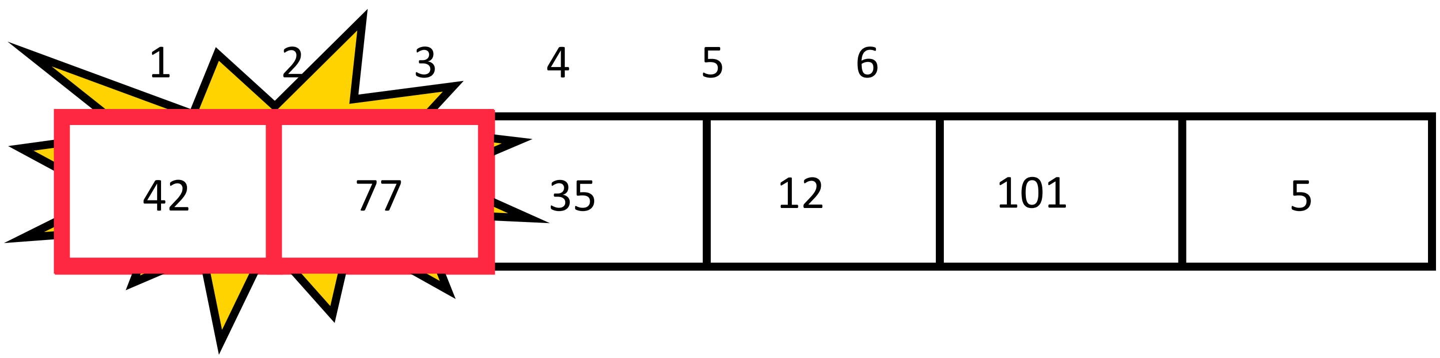
Parallel Sort
and
More Open MP

Chapter 5.6

Bryan Mills, PhD

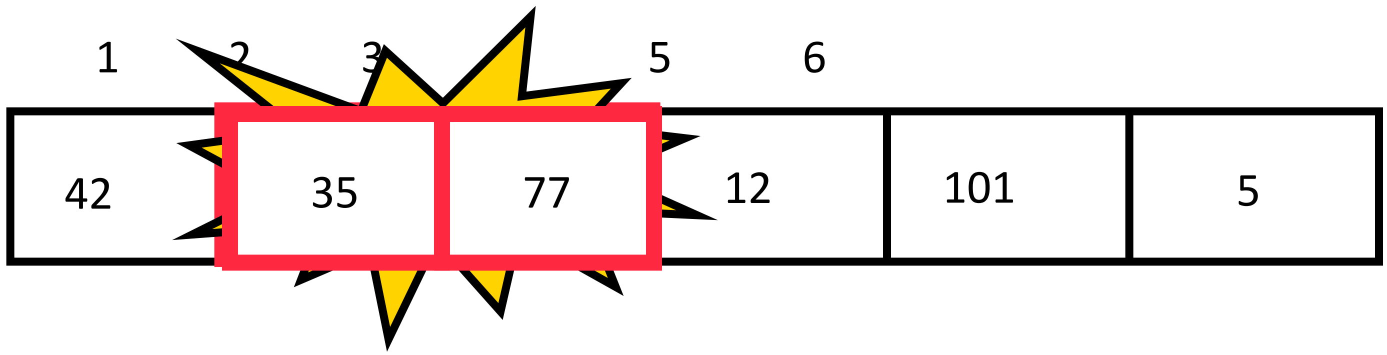
"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **"Bubble" the largest value to the end using pair-wise comparisons and swapping**



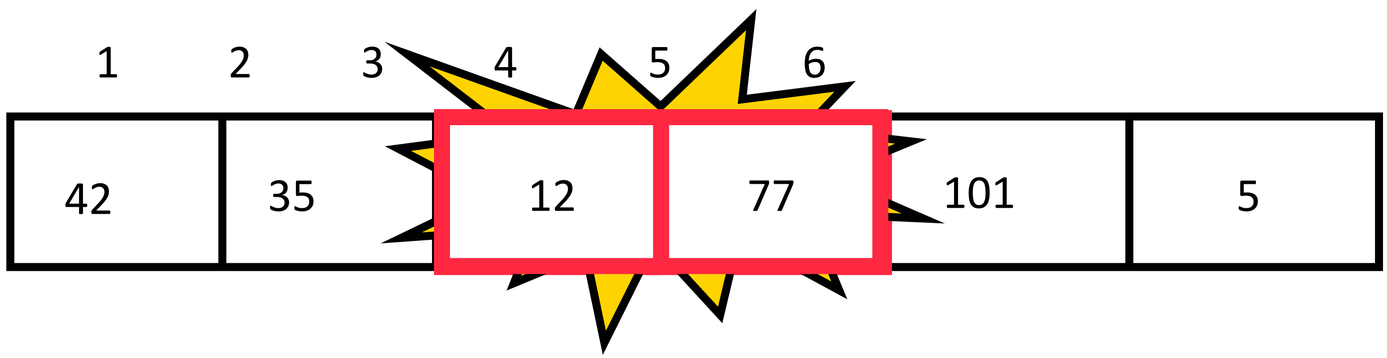
"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **"Bubble" the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

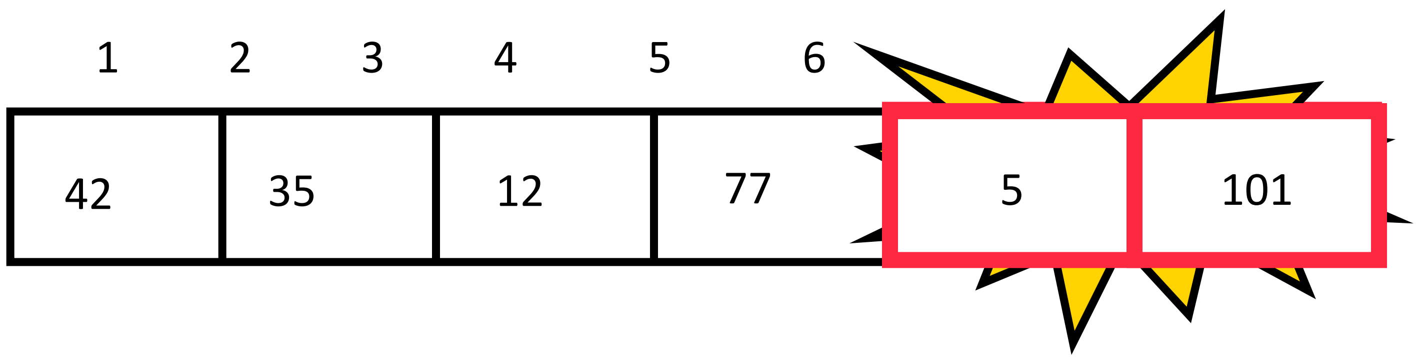
- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **"Bubble" the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
42	35	12	77	101	5

No need to swap

"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**



"Bubbling Up" the Largest Element

- **Traverse a collection of elements**
  - **Move from the front to the end**
  - **“Bubble” the largest value to the end using pair-wise comparisons and swapping**

1	2	3	4	5	6
42	35	12	77	5	101

Largest value correctly placed

Reducing the Number of Comparisons

1	2	3	4	5	6
77	42	35	12	101	5

1	2	3	4	5	6
42	35	12	77	5	101

1	2	3	4	5	6
35	12	42	5	77	101

1	2	3	4	5	6
12	35	5	42	77	101

1	2	3	4	5	6
12	5	35	42	77	101

Bubble Sort

```
for (list_length = n; list_length >= 2; list_length--) {
  for (i = 0; i < list_length; i++) {
    if (a[i] > a[i+1]) {
      tmp = a[i];
      a[i] = a[i+1];
      a[i+1] = tmp;
    }
  }
}
```

Serial Odd-Even Transposition Sort

```
for (phase = 0; phase < n; phase++)
   if (phase % 2 == 0)
      for (i = 1; i < n; i += 2)
         if (a[i-1] > a[i]) Swap(&a[i-1],&a[i]);
   else
      for (i = 1; i < n-1; i += 2)
         if (a[i] > a[i+1]) Swap(&a[i], &a[i+1]);
```

Serial Odd-Even Transposition Sort

Phase	Subscript in Array 0		1		2		3
0	9	$\leftrightarrow$	7		8	$\leftrightarrow$	6
	7		9		6		8
1	7		9	$\leftrightarrow$	6		8
	7		6		9		8
2	7	$\leftrightarrow$	6		9	$\leftrightarrow$	8
	6		7		8		9
3	6		7	$\leftrightarrow$	8		9
	6		7		8		9

First OpenMP Odd-Even Sort

```
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp parallel for num_threads(thread_count) \
            default(none) shared(a, n) private(i, tmp)
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Second OpenMP Odd-Even Sort

```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i−1] > a[i]) {
               tmp = a[i−1];
               a[i−1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n−1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

Odd-even sort with two parallel for directives and two for directives. (Times are in seconds.)

thread_count	1	2	3	4
Two `parallel` **for** directives	0.770	0.453	0.358	0.305
Two **for** directives	0.732	0.376	0.294	0.239

Parallel Merge Sort

- **How can we do this?**

Merge Sort Example

99	6	86	15	58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

99	6

86	15

58	35

86	4	0

Merge Sort Example

99	6	86	15	58	35	86	4	0

99	6	86	15

58	35	86	4	0

99	6

86	15

58	35

86	4	0

99

6

86

15

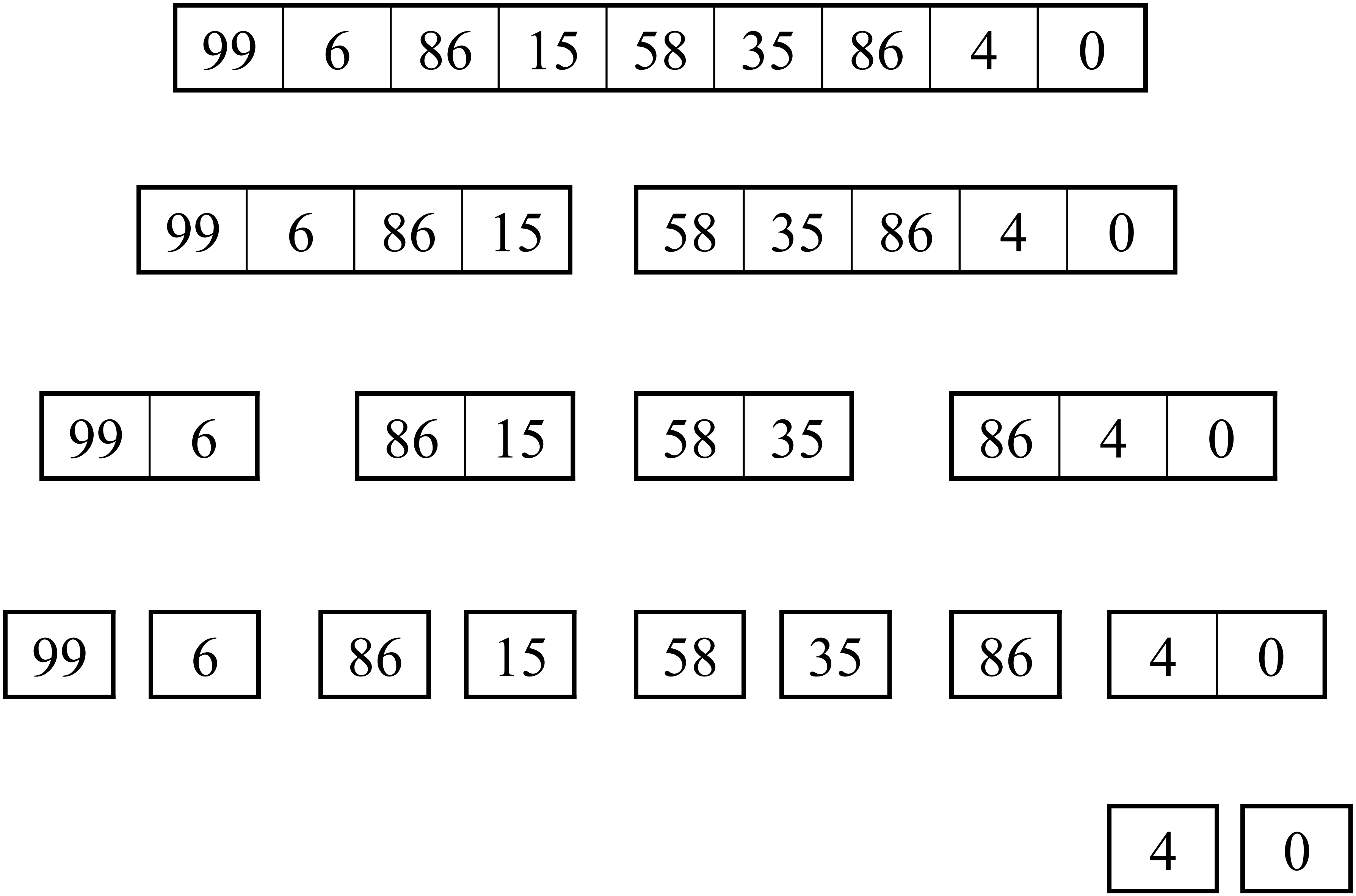
58

35

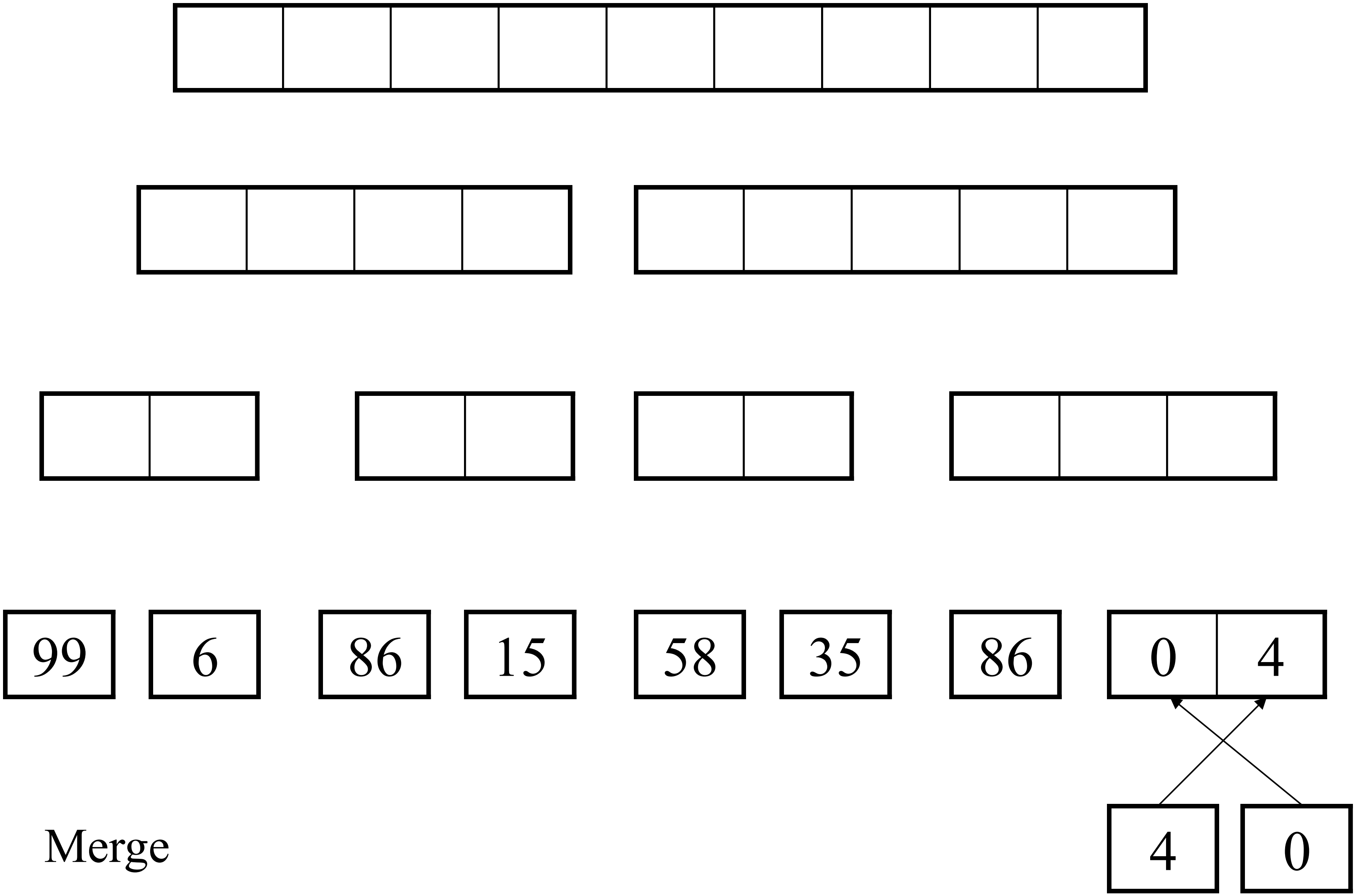
86

4	0

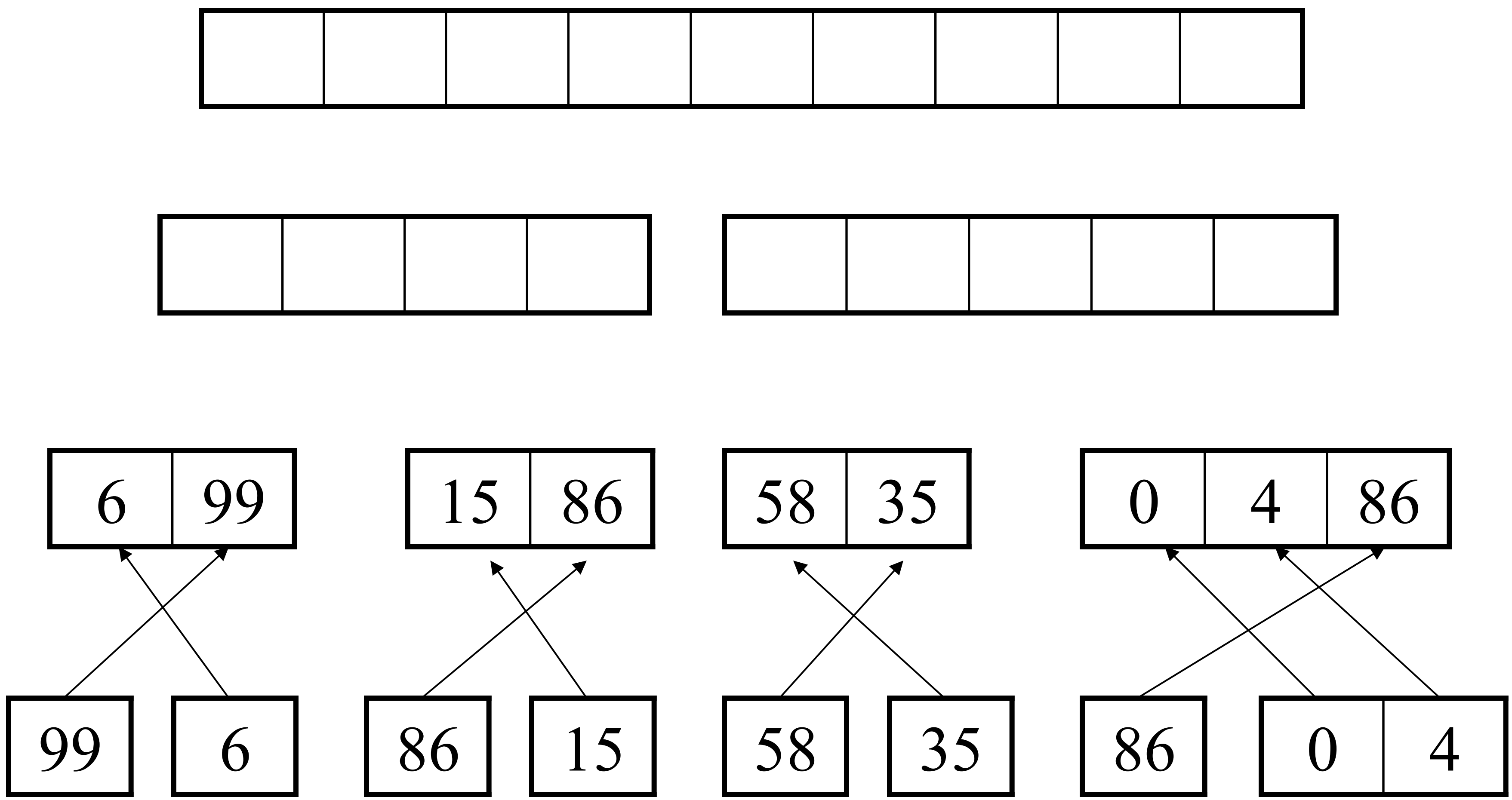
Merge Sort Example



Merge Sort Example

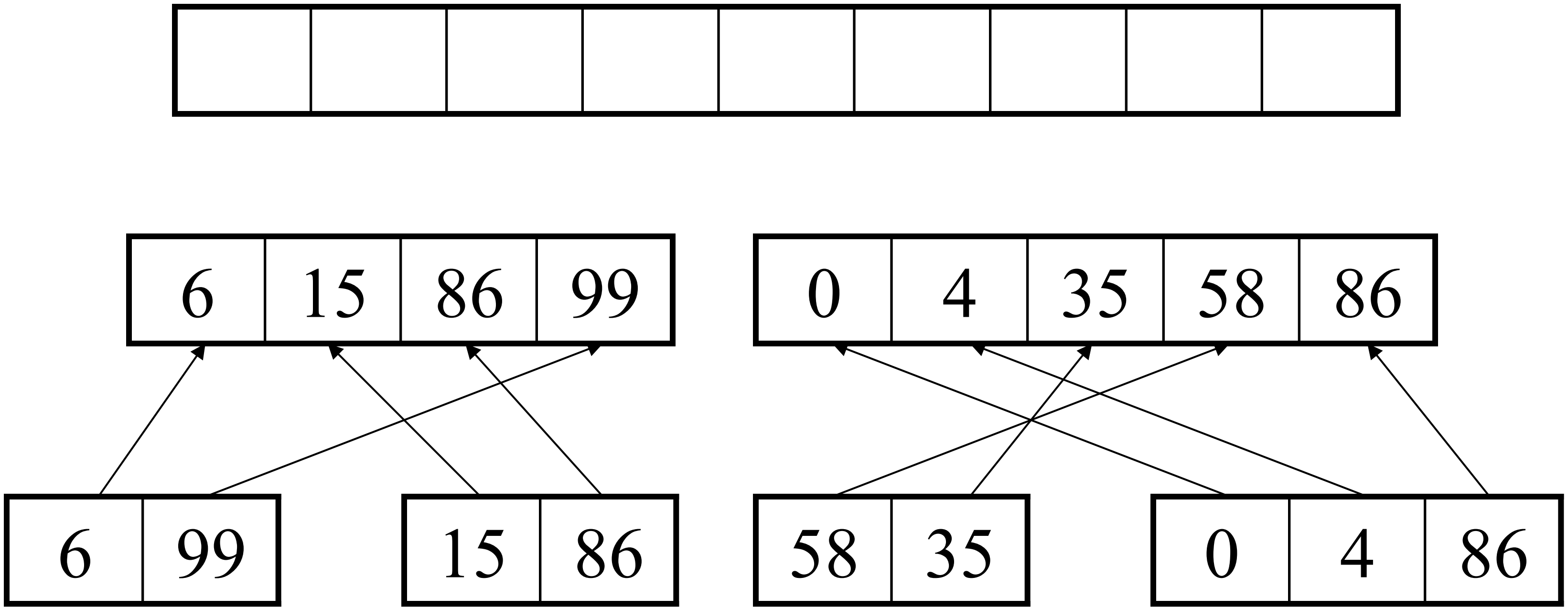


Merge Sort Example



Merge

Merge Sort Example



Merge

Merge Sort Example

0	4	6	15	35	58	86	86	99

6	15	86	99

0	4	35	58	86

Merge

Merge Sort Example

0	4	6	15	35	58	86	86	99

```
void MergeSort(int *A, int n) {
    int mid,i, *L, *R;
    if(n < 2) return;
    mid = n/2;  // find the mid index.

    L = (int*)malloc(mid*sizeof(int));
    R = (int*)malloc((n - mid)*sizeof(int));

    for(i = 0;i<mid;i++) L[i] = A[i];
    for(i = mid;i<n;i++) R[i-mid] = A[i];

    MergeSort(L,mid);  // sorting the left subarray
    MergeSort(R,n-mid);  // sorting the right subarray
    Merge(A,L,mid,R,n-mid);
}
```

This is looking good, lets make execute in parallel!

```
void Merge(int *A,int *L,int leftCount,int *R,int rightCount) {
   int i,j,k;
    // i - to mark the index of left subarray (L)
    // j - to mark the index of right subarray (R)
    // k - to mark the index of merged subarray (A)
    i = 0; j = 0; k =0;

    while(i<leftCount && j< rightCount) {
        if(L[i]  < R[j]) A[k++] = L[i++];
        else A[k++] = R[j++];
    }
    while(i < leftCount) A[k++] = L[i++];
    while(j < rightCount) A[k++] = R[j++];
}
```

Merge is the problem? How do we merge in parallel?

t =

99	6	86	15	58	35	86	4	0

a =

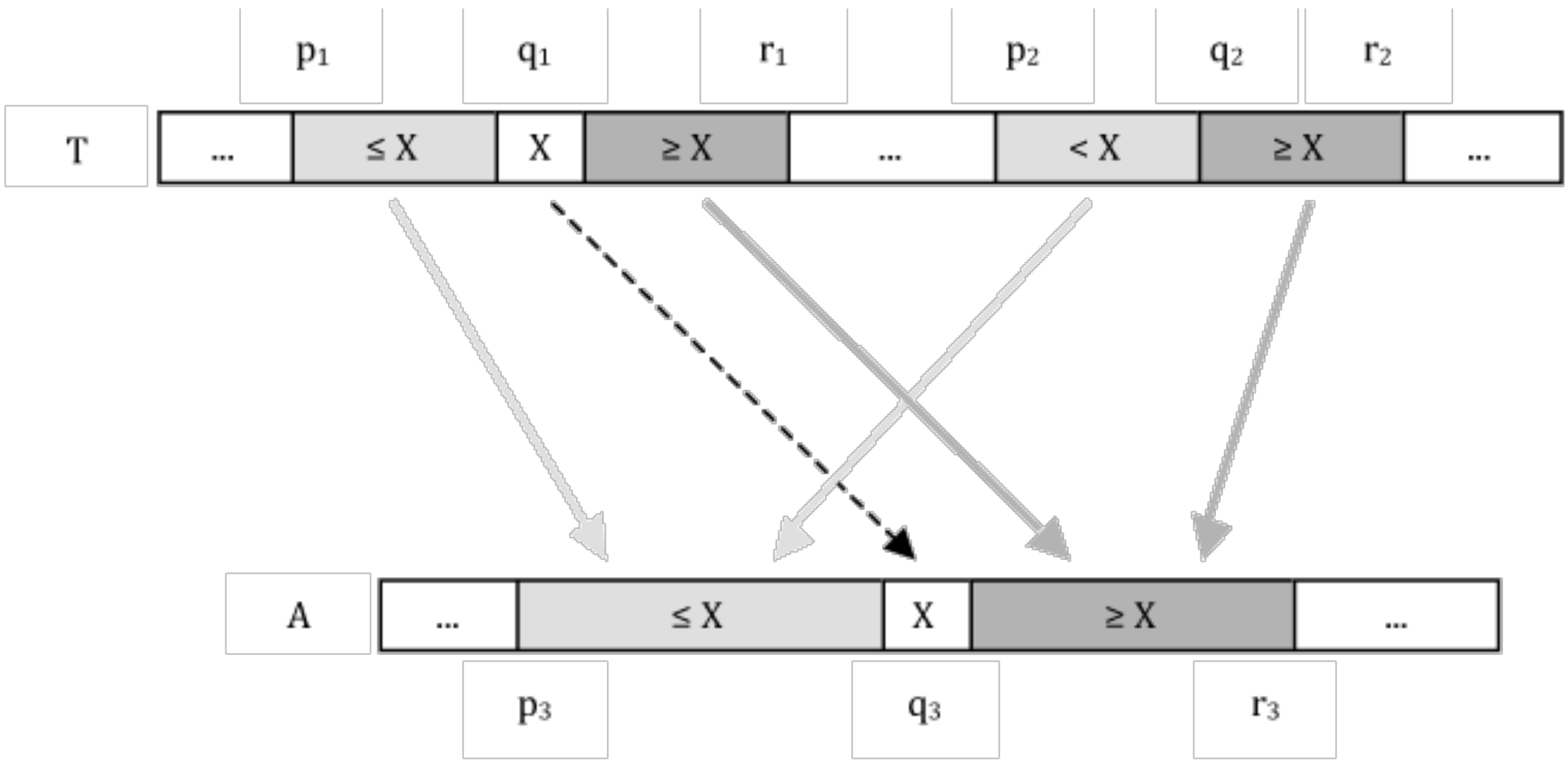
0	4	6	15	35	58	86	86	99

```
void MergeSort(int t, int p, int r, int *a) {
    int mid, i, n;
    n = r - p;
    if(n < 2) return;
    mid = n/2;  // find the mid index.

    MergeSort(t, p, mid, a);  // sorting the left subarray
    MergeSort(t, mid+1, r, a);  // sorting the right subarray
    Merge(t, p, mid, mid+1, r, a, p);
}
```

Note issues with inplace, ignore for now!

Parallel Merge



Merge two ranges of array T[p1 .. r1] and T[p2 .. r2] into destination array A starting at index p3.

```
Void MergePar(int t, int p1, int r1, int p2, int r2,
              int* a, int p3 )
{
    int length1 = r1 - p1 + 1;
    int length2 = r2 - p2 + 1;
    if ( length1 < length2 )
    {
        exchange(      p1,       p2 );
        exchange(      r1,       r2 );
        exchange( length1, length2 );
    }
    if ( length1 == 0 ) return;
    int q1 = ( p1 + r1 ) / 2;
    int q2 = BinarySearch( t[ q1 ], t, p2, r2 );
    int q3 = p3 + ( q1 - p1 ) + ( q2 - p2 );
    a[ q3 ] = t[ q1 ];
    MergePar( t, p1,     q1 - 1, p2, q2 - 1, a, p3     );
    MergePar( t, q1 + 1, r1,     q2, r2,     a, q3 + 1 );
}
```

How to Parallelize?
Sections

- **Sections will be executed in parallel**
- **Can combine parallel and section like we did with for**
- **If more threads than sections then idle threads will exist**
- **If less threads than sections then some sections will execute in serial**

Sections

```
#define N 1000
main (){
  int i;float a[N], b[N], c[N];
  for (i=0; i < N; i++) a[i] = b[i] = … ;

# pragma omp parallel shared(a,b,c) private(i)
  {
   . . .
# pragma omp sections
  {
#   pragma omp section
    {
      for (i=0; i < N/2; i++) c[i] = a[i] + b[i];
    }
#   pragma omp section
    {
      for (i=N/2; i < N; i++) c[i] = a[i] + b[i];
    }
  } /* end of sections */
. . .
 } /* end of parallel */
}
```

How to use in Parallel Merge

```
 . . .
# pragma omp parallel
{
  # pragma omp sections
  {
    # pragma omp section
    MergePar( t, p1,     q1 - 1, p2, q2 - 1, a, p3     );
    # pragma omp section
    MergePar( t, q1 + 1, r1,     q2, r2,     a, q3 + 1 );
  } // END sections block
} // end parallel block

 . . .
```

Note the recursion with parallel sections

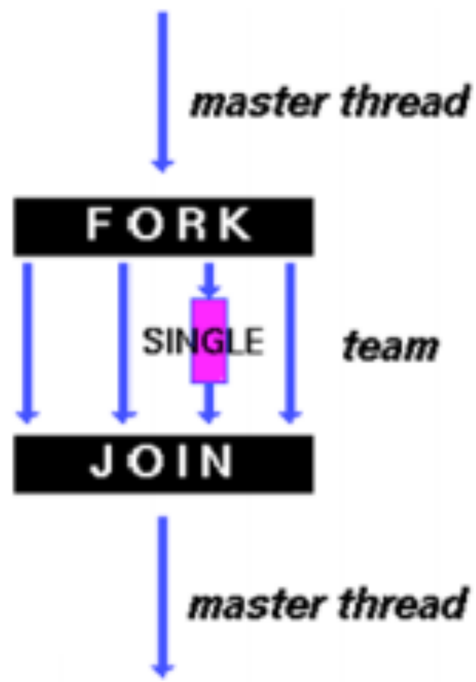
Does this work? How?

We'll spend sometime discussing next week

Single and Master

- **Single indicates that only one thread in team will execute the code.**

  #pragma omp single



- **Master indicates that only the master thread will execute the code.**

  #pragma omp master

Critical with Names

- **Critical ensures that only one thread can execute code block at a time.**

```
# pragma omp critical
  global_result += my_result ;
```

- **You can name critical sections, then critical sections with different names can be executed in parallel**

```
# pragma omp critical(name)
  global_result += my_result ;
```

Barrier

- **When a thread executes a barrier it will wait until all other threads in the team also execute the barrier.**
- **Then threads continue working as usual.**

```
# pragma omp barrier
  … continue on past…
```

Atomic

- **If your critical section is of the form**

  `x += y` **OR** `x = x <operation> y`

- **Compiler can use hardware atomic operations**
- **Think "mini-critical"**

```
#pragma omp parallel for
shared(sum)
for(i = 0; i < n; i++){
   value = f(a[i]);
#  pragma omp atomic
   sum = sum + value;
}
```

We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

Assignment of work using cyclic partitioning.

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
}  /* f */
```

Our definition of function *f*.

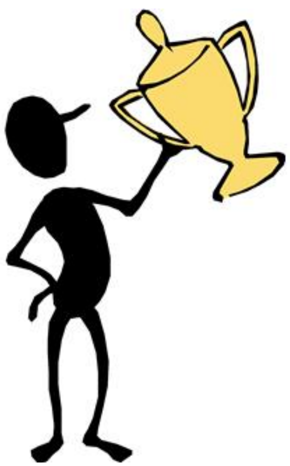
Results

- f(i) calls the sin function *i* times.
- Assume the time to execute f(2i) requires approximately twice as much time as the time to execute f(i).

- n = 10,000
  - one thread
  - run-time = 3.67 seconds.

Results

- n = 10,000
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33
- n = 10,000
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99



The Schedule Clause

- Default schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

- Cyclic schedule:

```
        sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) schedule(static,1)
        for (i = 0; i <= n; i++)
           sum += f(i);
```

schedule (type , chunksize)

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.

The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,1)
```

$$
\begin{aligned}
\text{Thread } 0: &\quad 0,3,6,9 \\
\text{Thread } 1: &\quad 1,4,7,10 \\
\text{Thread } 2: &\quad 2,5,8,11
\end{aligned}
$$

The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,2)
```

$$\begin{aligned} \text{Thread } 0: &\quad 0, 1, 6, 7 \\ \text{Thread } 1: &\quad 2, 3, 8, 9 \\ \text{Thread } 2: &\quad 4, 5, 10, 11 \end{aligned}$$

The Static Schedule Type

twelve iterations, 0, 1, . . . , 11, and three threads

```
schedule(static,4)
```

$$\begin{aligned}\text{Thread } 0: &\quad 0,1,2,3\\ \text{Thread } 1: &\quad 4,5,6,7\\ \text{Thread } 2: &\quad 8,9,10,11\end{aligned}$$

The Dynamic Schedule Type

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

The Guided Schedule Type

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- However, in a guided schedule, as chunks are completed the size of the new chunks decreases.
- If no chunksize is specified, the size of the chunks decreases down to 1.
- If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

The Runtime Schedule Type

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.
- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.

Graveyard