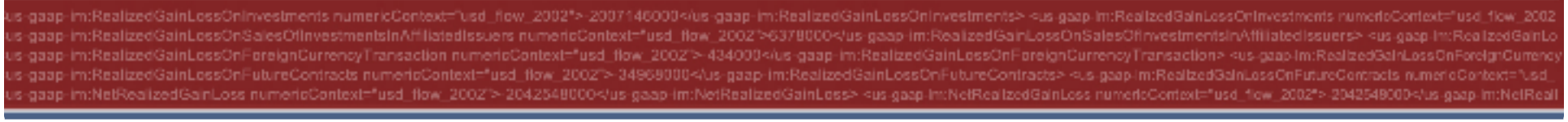
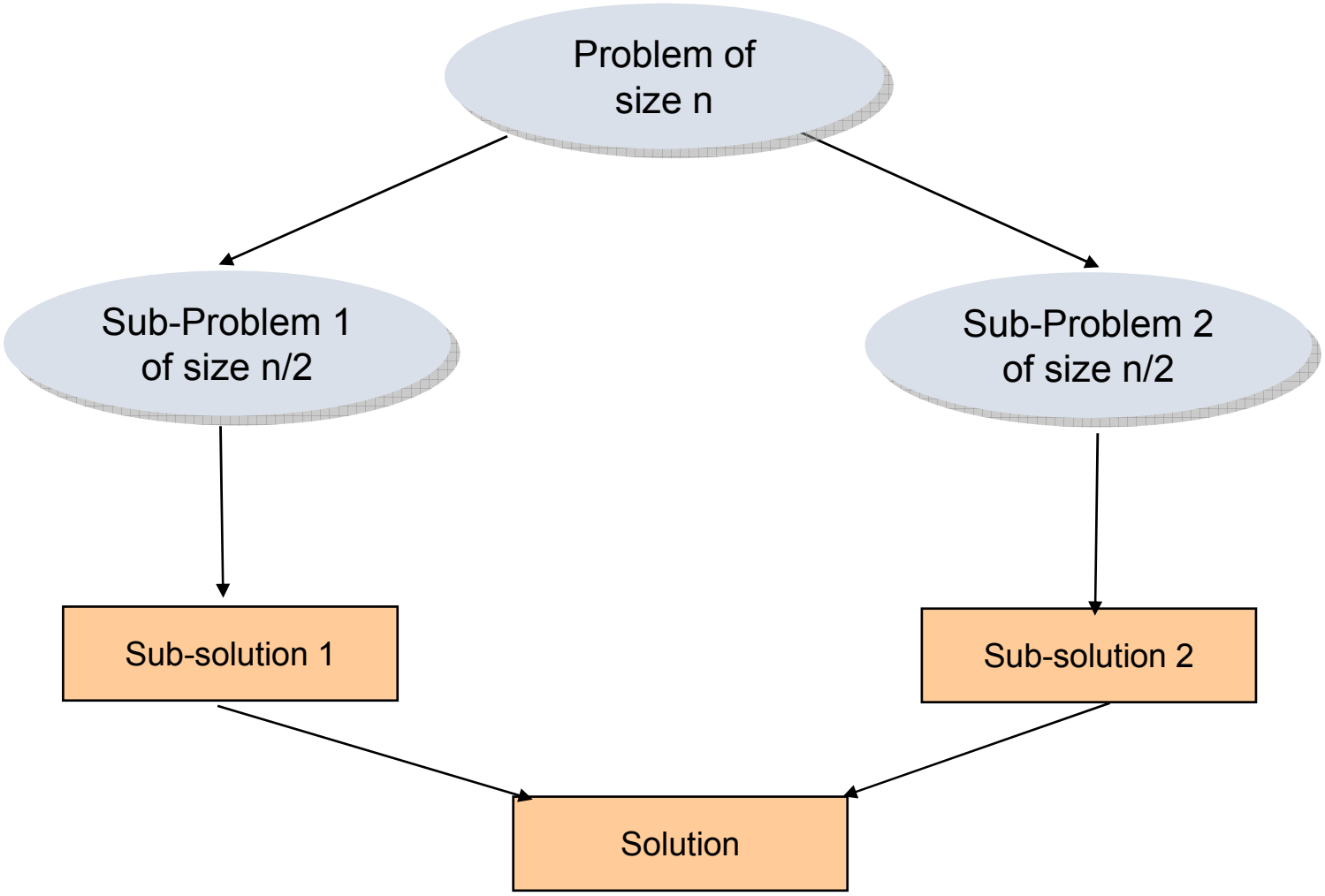
# Sorting


us-gaap:RealizedGainLossOnInvestments numericContext="usd_flow_2002"> 2007146000</us-gaap:RealizedGainLossOnInvestments> <us-gaap:RealizedGainLossOnInvestments numericContext="usd_flow_2002
us-gaap:RealizedGainLossOnSalesOfInvestmentsInAffiliatedIssuers numericContext="usd_flow_2002">6378000</us-gaap:RealizedGainLossOnSalesOfInvestmentsInAffiliatedIssuers> <us-gaap:RealizedGainLo
us-gaap:RealizedGainLossOnForeignCurrencyTransaction numericContext="usd_flow_2002"> 434000</us-gaap:RealizedGainLossOnForeignCurrencyTransaction> <us-gaap:RealizedGainLossOnForeignCurrency
us-gaap:RealizedGainLossOnFutureContracts numericContext="usd_flow_2002"> 34969000</us-gaap:RealizedGainLossOnFutureContracts> <us-gaap:RealizedGainLossOnFutureContracts numericContext="usd_
us-gaap:NetRealizedGainLoss numericContext="usd_flow_2002"> 2042548000</us-gaap:NetRealizedGainLoss> <us-gaap:NetRealizedGainLoss numericContext="usd_flow_2002"> 2042548000</us-gaap:NetReali

# Overview

- Divide and Conquer
- Bubble Sort
- Naïve Parallel Sort
- Merging
- Parallel Mergesort
- Parallel Quicksort

# Divide and Conquer



# Parallel Divide and Conquer

- Execute the sub tasks in parallel
- Get sub solution to each sub task
- Combine sub solutions into the solution
  - *The combination step sometime can be parallel, for example. Merging in Mergesort*

# Overview

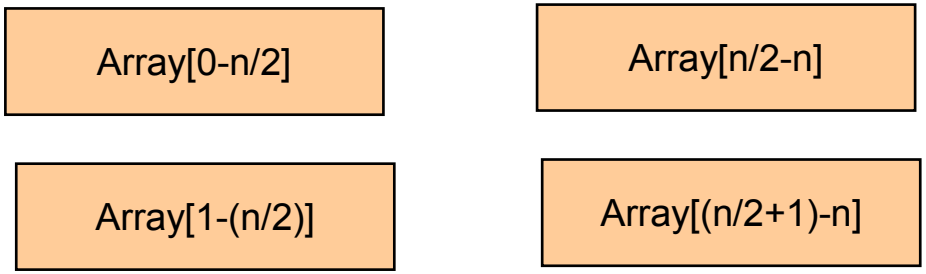
- Divide and Conquer
- ***Bubble Sort***
- Naïve Parallel Sort
- Merging
- Parallel Mergesort
- Parallel Quicksort

# Bubble Sort

- Bubble sort is a *very very* inefficient sorting algorithm. But for some reason, it shows up in almost all algorithm textbooks.
- It swaps two adjacent items if the two items are not ordered until there are no items that can be swapped.
- The complexity of bubble sort is sqr(n) as opposed to n*log(n) of most other well-known sorting algorithms such as mergesort or quicksort

# Parallel Bubble Sort

- Divide the array into n sub-arrays so that each sub-array is assigned to a thread. Each thread executes a bubble sort
- Re-divide the array into n sub-arrays by shifting one item away and assign them into the thread pool



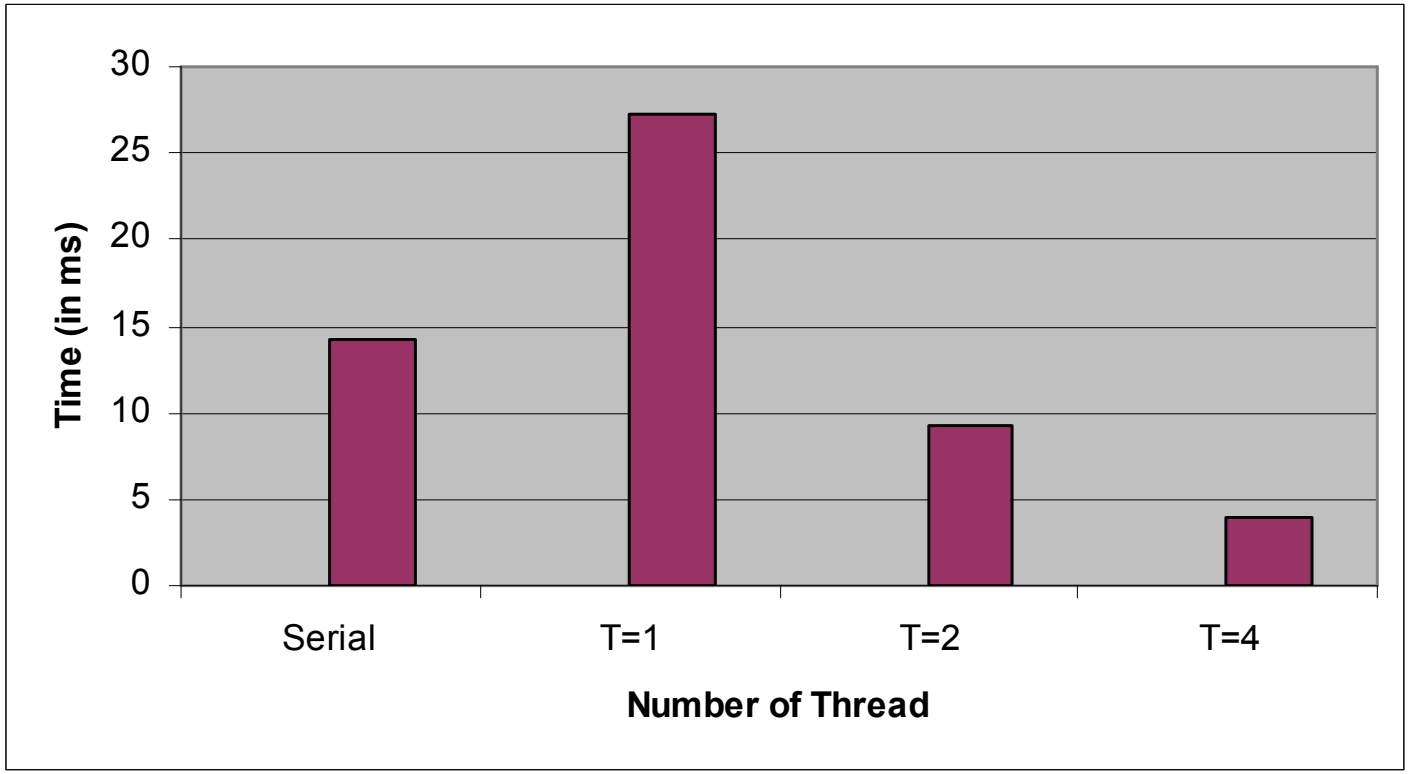

- The algorithm stops when there is no swap going on, also called odd-even sort.

# Example 1: Parallel Bubble Sort

```
while (true) {
        for (int i = 0; i < numOfThreads; i++) {
                int endPos = (i+1)*len ;
                if (endPos > MAX_LENGTH) endPos = MAX_LENGTH;
                threadPool[i] = new BubbleSort(i*len +1, endPos);
                threadPool[i].start();
        }

        for (int i = 0; i < numOfThreads; i++)  { threadPool[i].join(); }

        for (int i = 0; i < numOfThreads; i++) {
                int endPos = (i+1)*len ;
                if (endPos > MAX_LENGTH) endPos = MAX_LENGTH;
                threadPool[i] = new BubbleSort(i*len, endPos);
                threadPool[i].start();
        }

        for (int i = 0; i < numOfThreads; i++) {threadPool[i].join();}

        boolean allDone = true;
        for (int i = 0; i < numOfThreads; i++) {
                if (threadPool[i].done == false) { allDone = false;break;}
        }
        if (allDone) break;
}
```

# Parallel Bubble Sort Performance

- Configuration: 4-core, 2.66Ghz, 4G RAM with Windows Vista 64-bit. The to be sorted array has 24K elements with “String” type, and each String is of length between 1-10.
- The reason T=1 is worse than Serial is there are some “join” operations in T=1 in order to make the implementation universal.
- If we just compare Serial/T=2/T=4, we got pretty reasonable speedup.



# Overview

- Divide and Conquer
- Bubble Sort
- ***Naïve Parallel Sort***
- Merging
- Parallel Mergesort
- Parallel Quicksort

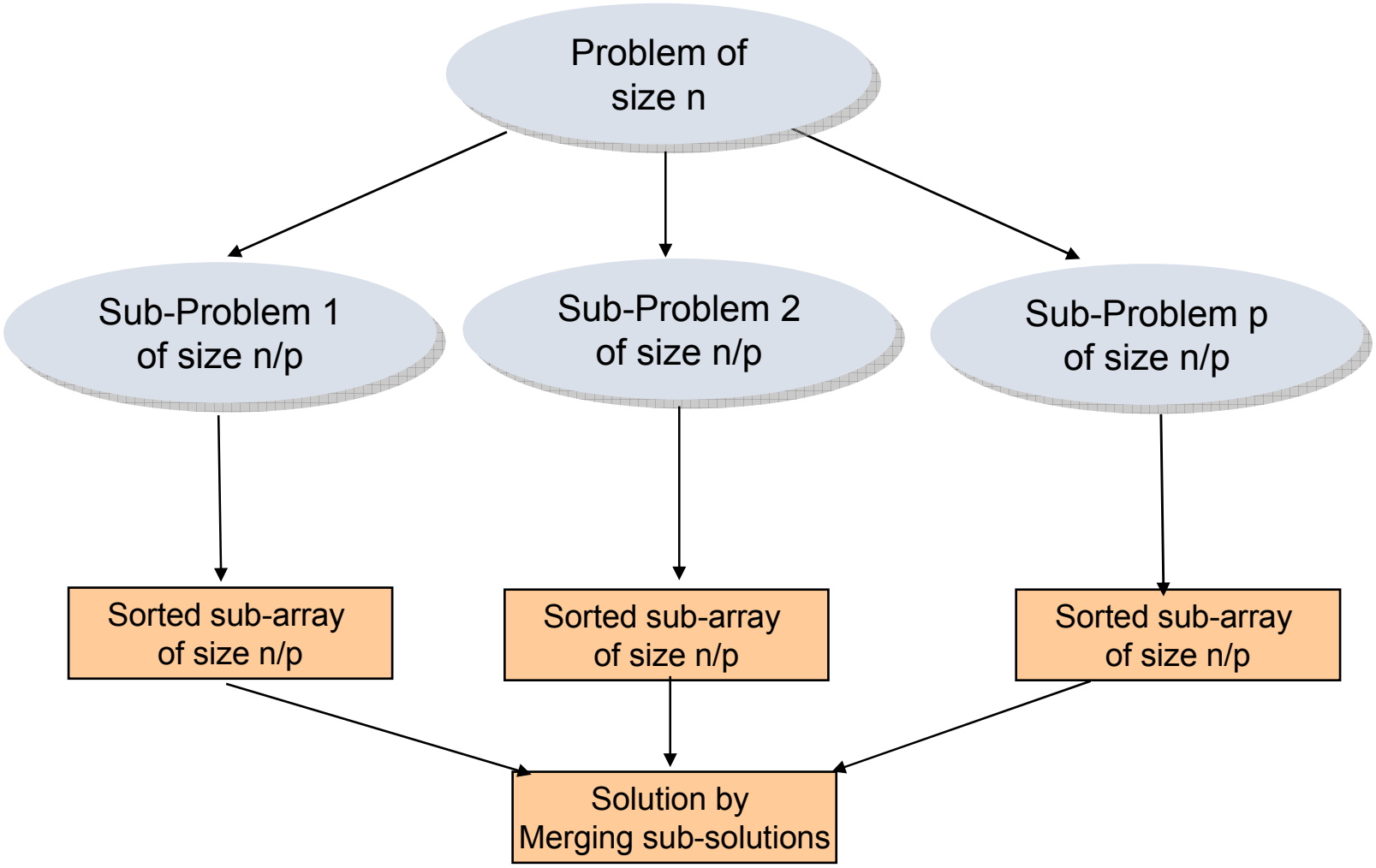
# Take Advantage of Existing Sort Lib

- In `java.util.Arrays`, there are a series of "`sort()`" methods
  - *For arrays with primitive types, such as* `int, long, float, double`*, it performs "quicksort"*
    - *Quicksort has average performance of n*log(n), worst case at sqr(n). Quicksort is a very interesting algorithm that its average performance is way better than the worse performance. Even today a lot of research has been conducted in different variations.*
  - *For other array types, it performs "mergesort", an n*log(n) algorithm but sometimes not as fast as quicksort mainly because of its array copying operations*
- We can divide the array into several sub-arrays and assign sub-arrays to several threads to perform "sort" in parallel among sub-arrays
- After all sub-arrays are sorted, a "merging" operation is performed to merge all sorted sub-arrays

# Naïve Parallel Sorting

- We can divide the array into several sub-arrays and assign different sub-arrays to different threads to perform serial “sort” (in existing library) in parallel among sub-arrays
- After all sub-arrays are sorted, a “merging” operation is performed to merge all sorted sub-arrays

# Naïve Parallel Sorting

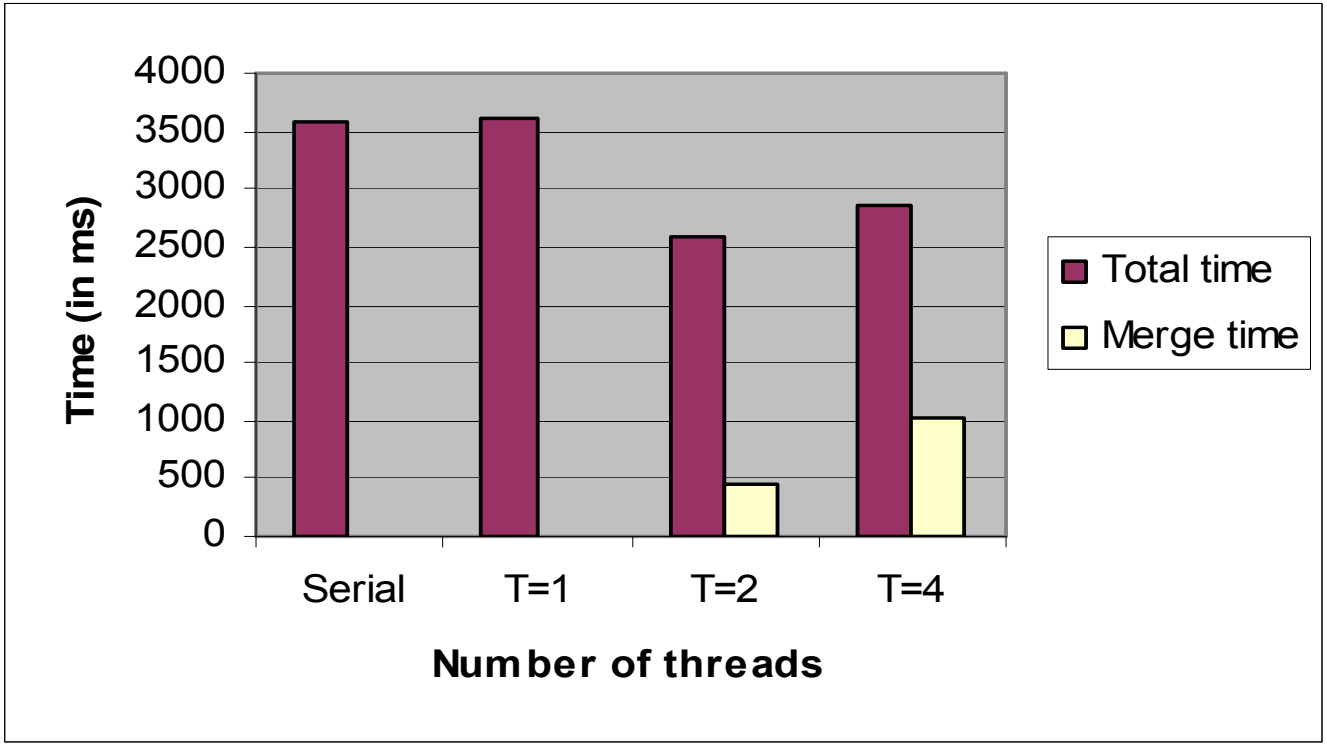


# Example 2: Naïve Parallel Sort

```
for (int i = 0; i < numOfThreads; i++) {
                int endPos = (i+1)*len;
                if (endPos > MAX_LENGTH) endPos = MAX_LENGTH;
                threadPool[i] = new MergeSort1(i*len, endPos); // java.util.Arrays.sort()
                threadPool[i].start();
        }

        for (int i = 0; i < numOfThreads; i++) {threadPool[i].join();}
        for (int i = 0; i < (numOfThreads-1); i++) {
                String[] tgt = merge(stringArray, 0, (i+1)*len, stringArray, (i+1)*len, (i+2)*len);
                System.arraycopy(tgt, 0, stringArray, 0, tgt.length);
        }

}
```

# Naïve Parallel Sort Performance

- Configuration: 4-core, 2.66Ghz, 4G RAM with Windows Vista 64-bit. The to be sorted array has 2.4M elements with “String” type, and each String is of length between 1-10.
- When number of threads increases to 4, no performance gain. Merging takes substantial amount of time



# Advantage of Naïve Parallel Sort

- It's simple, no need to understand the internals of serial sorting algorithms
- It can work with any existing serial sorting algorithms
- Reasonable performance when number of processors is small
  - $O(n*log(n/p)/p + n) = O(n*log(n)/p – nlog(p)/p + n) = O(n*log(n)/p)$

# Overview

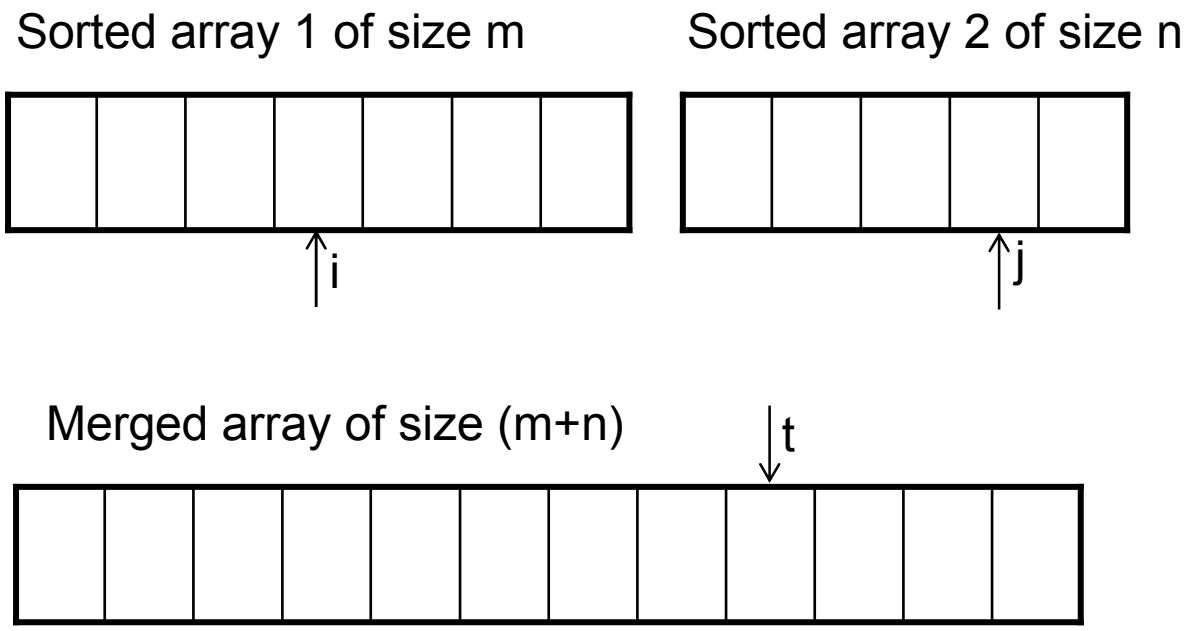
- Divide and Conquer
- Bubble Sort
- Naïve Parallel Sort
- ***Merging***
- Parallel Mergesort
- Parallel Quicksort

# Merging: The Serial Algorithm

» How do we do serial merging

» Two sorted arrays, merge them into a bigger sorted array

- *O(m+n)*



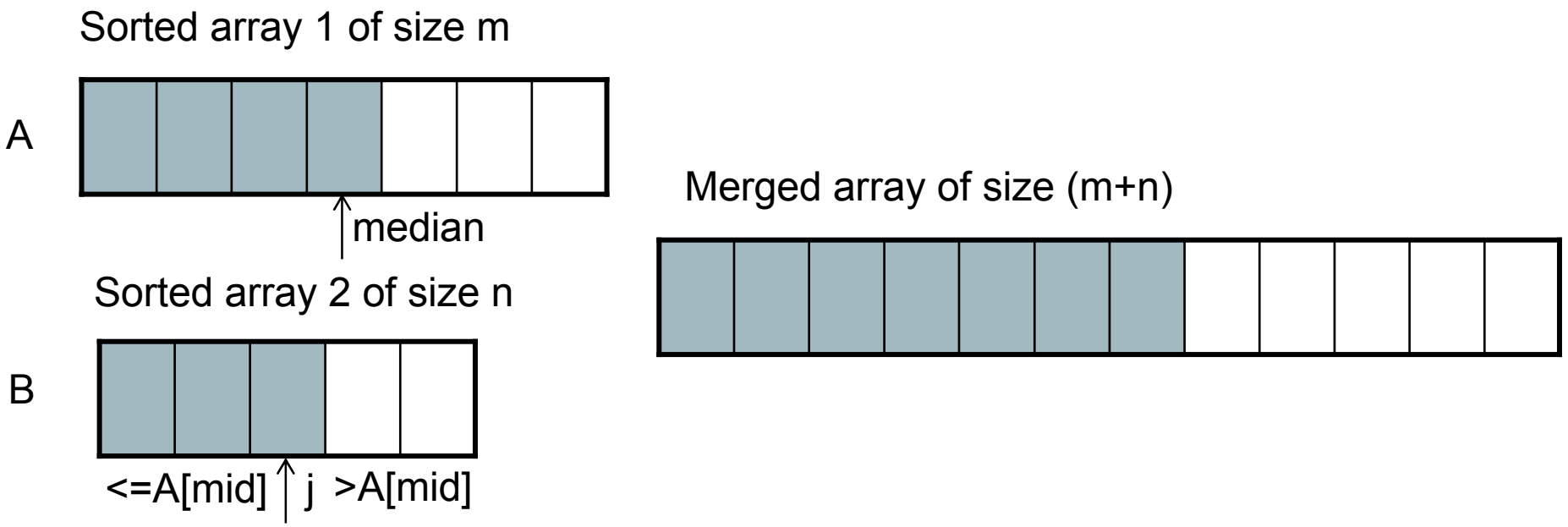
# Merging: Serial Algorithm

```
static String[] merge(String[] src1, int firstStart, int firstEnd, String[] src2, int secondStart, int
    secondEnd) {
»           String[] tgt = new String[secondEnd-secondStart+firstEnd-firstStart];
»           int fi = firstStart;
»           int si = secondStart;
»           for (int i = 0; i < tgt.length; i++) {
»                   if (fi >= firstEnd) {
»                           System.arraycopy(src2, si, tgt, i, (secondEnd-si));
»                           break;
»                   }
»                   if (si >= secondEnd) {
»                           System.arraycopy(src1, fi, tgt, i, (firstEnd-fi));
»                           break;
»                   }
»                   if (src1[fi].compareTo(src2[si]) <= 0) {
»                           tgt[i] = src1[fi]; fi++;
»                   } else {
»                           tgt[i] = src2[si]; si++;
»                   }
»           }
»           return tgt;
»   }
```

# Parallel Merging: The Parallel Algorithm

» How do we do parallel merging? Divide and Conquer

- *Divide the longer array A into two halves using the middle item*
- *Do a binary search on B using the media of A and then divide the shorter array B using the middle item in the longer array*
- *Merge the 1st half of A with the first part of B; and second half of B with second part of B in parallel*



# Example 3: Implementation of Parallel Merging

```
» static void pMerge(String[] tgt, int tStart, String[] src1, int start1, int end1, String[] src2, int start2, int end2)
» {
»       if ((end1-start1) < (end2-start2)) {
»           pMerge(tgt, tStart, src2, start2, end2, src1, start1, end1);
»           return;
»       }
»       if ((end1-start1) < (SERIAL_THRESHOLD+1)) {
»           merge(tgt, tStart, src1, start1, end1, src2, start2, end2);
»           return;
»       }
»       int halfLen1 = (end1-start1)>>>1;
»       int med = start1 + halfLen1;
»       int par2 = find(src1[med], src2, start2, end2);      // binary search
»       PMerge pm1 = new PMerge(tgt, tStart, src1, start1, med, src2, start2, par2);
»       PMerge pm2 = new PMerge(tgt, tStart+ halfLen1+(par2-start2), src1, med, end1, src2, par2,end2);
»       try {
»           pm1.join();
»           pm2.join();
»       } catch (InterruptedException ie) {
»           System.out.println(ie);
»       }
» }
```

# Improvement of Naïve Parallel Sort

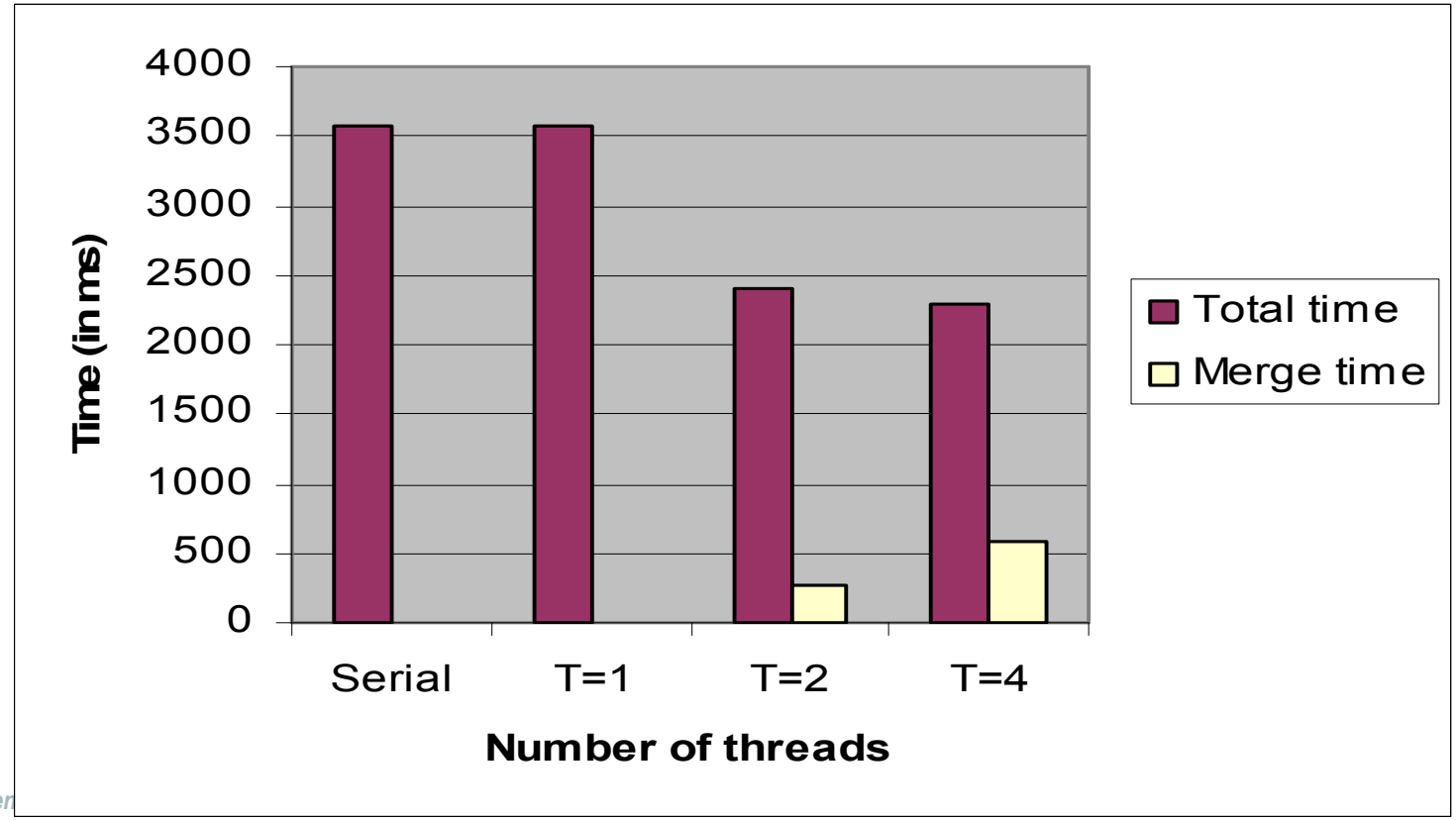
- Replacement of serial merging with parallel merging in the naïve algorithm
- It can work with any sorting algorithms
- Reasonably good performance without much work
  - *O(n*log(n/p)/p + n/p) = O(n*log(n)/p – nlog(p)/p +n/p) = O(n*log(n)/p)*

# Example 4: Improvement of Naïve Parallel Sort

```
for (int i = 0; i < numOfThreads; i++) {
                        int endPos = (i+1)*len;
                        if (endPos > MAX_LENGTH) endPos = MAX_LENGTH;
                        threadPool[i] = new MergeSort1(i*len, endPos); // java.util.Arrays.sort()
                        threadPool[i].start();
            }

            for (int i = 0; i < numOfThreads; i++) {threadPool[i].join();}
            for (int i = 0; i < (numOfThreads-1); i++) {
                        String[] tgt = PMerge.pmerge(stringArray, 0, (i+1)*len, stringArray, (i+1)*len,
(i+2)*len);
                        System.arraycopy(tgt, 0, stringArray, 0, tgt.length);
            }

}
```

# Improvement of Naïve Parallel Sort: Performance

- Configuration: 4-core, 2.66Ghz, 4G RAM with Windows Vista 64-bit. The to be sorted array has 2.4M elements with “String” type, and each String is of length between 1-10.
- Merge time has been substantially reduced. When number of threads increases to 4, some marginal performance gain.



# Naïve Algorithm Not Good Enough

- Still a lot of room to improve
  - *Need to work the parallel algorithm inside out*
- The good old Divide and Conquer technique
- Recursive and Parallel
  - *Study the classical recursive algorithm and make it parallel*
- Both merge-sort and quick-sort can be made into parallel quite easily
  - *Simply change the divide-and-conquer recusiveness into divide-and-conquer parallelism*

# Overview

- Divide and Conquer
- Bubble Sort
- Naïve Parallel Sort
- Merging
- ***Parallel Mergesort***

# MergeSort: Classical and Parallel Algorithms

- Divide-and-Conquer

```
1. Mergesort(A) {
2.    Divide a given array A into two parts B and C
3.    Mergesort(B);
4.    Mergesort(C);
5.    Merge B and C into a new array D;
6.    Copy D to A
7. }
```

- Divide-and-Conquer in Parallel

```
1. Mergesort(A) {
2.    Divide a given array A into two parts B and C
3.    Spawn Mergesort(B);
4.    Spawn Mergesort(C);
5.    join(); // waiting spawned threads to finish
6.    Parallel Merge B and C into a new array D
7.    Copy D to A
8. }
```

# Example 5: Parallel Mergesort

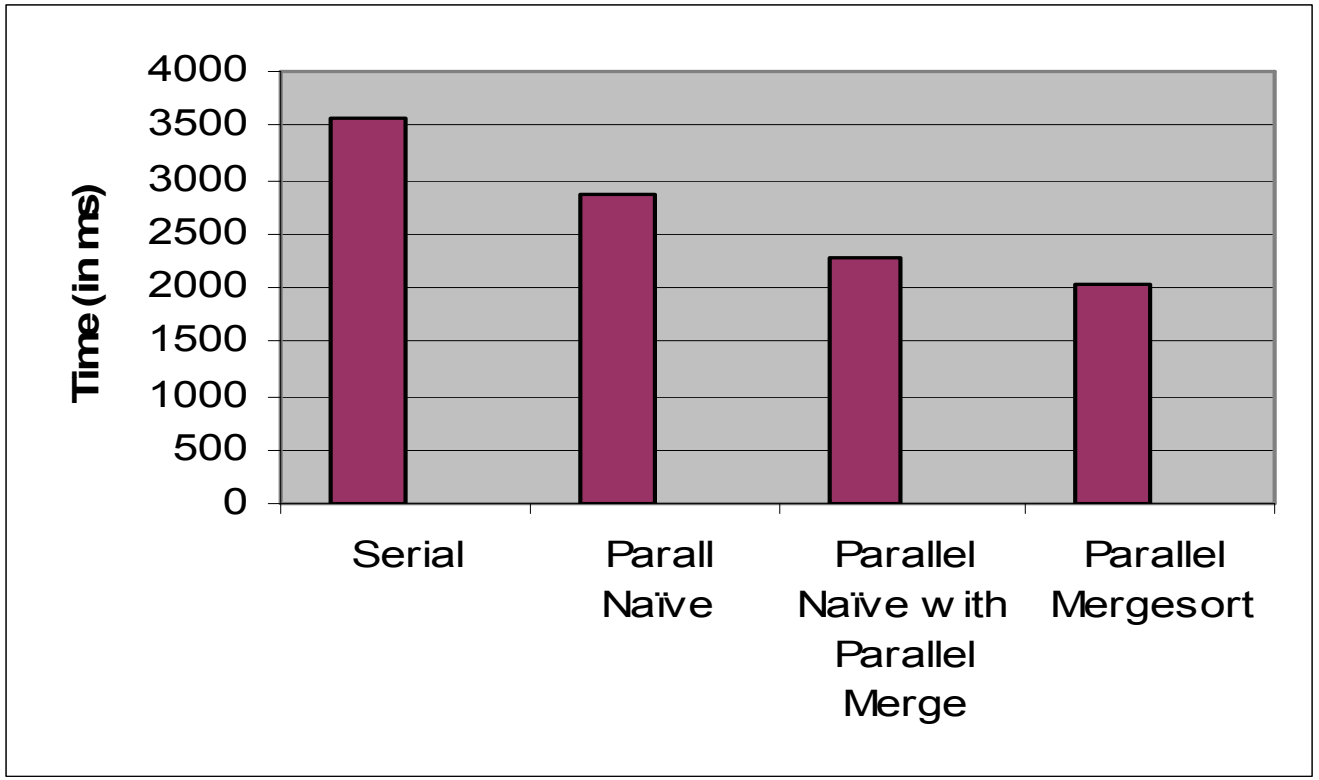
```
» private final void pSort() {
»         if (endPos <= startPos)
»                 return;
»         if ((endPos - startPos) < (SERIAL_THRESHOLD+1)) {
»                 Arrays.sort(stringArray, startPos, endPos);
»                 return;
»         }
»         int halfLen = (endPos - startPos) >>>1;
»         int medPos = startPos + halfLen;
»         MergeSort3 ms1 = new MergeSort3(stringArray, startPos, medPos);
»         MergeSort3 ms2 = new MergeSort3(stringArray, medPos, endPos);
»         try {
»                 ms1.join();
»                 ms2.join();
»         } catch (InterruptedException ie) {
»         }
»         String[] tgt = new String[endPos-startPos];
»         PMerge.MAX_LENGTH = MAX_LENGTH;
»         PMerge pm = new PMerge(tgt, 0, stringArray, startPos, medPos, stringArray, medPos, endPos);
»         try {
»                 pm.join();
»         } catch (InterruptedException ie) {
»         }
»         System.arraycopy(tgt, 0, stringArray, startPos, tgt.length);
» }
```

# Example 5: Parallel MergeSort

- A parallel implementation of divide-and-conquer
  - *The recursion is implemented in parallel*
- When the parallelism is reached to its threshold, i.e., the number of threads is larger than the number of physical processors, the sub-array sort is done using Java's `Arrays.sort()`

# Performance of Parallel MergeSort

- With same configuration machine, number of cores is 4, performance comparison among different algorithms.
- Sorting in general is a memory intensive, when more CPUs are added, memory bandwidth becomes bottleneck



# Overview

- Divide and Conquer
- Bubble Sort
- Naïve Parallel Sort
- Merging
- Parallel Mergesort
- ***Parallel Quicksort***

# Quicksort: Classical and Parallel Algorithms

- Divide-and-Conquer

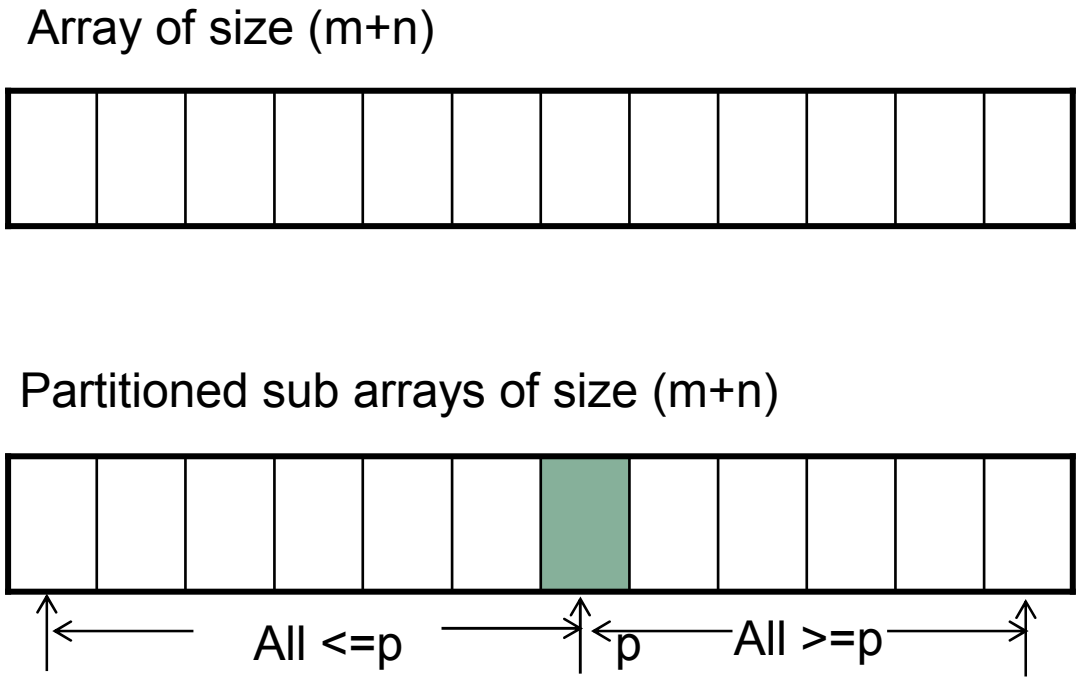
```
1. Quicksort(A, l, r) {
2.     s = Partition(A, l, r); s is the split point
3.     Quicksort(A, l, s-1);
4.     Quicksort(A, s+1, r);
5. }
```

- Divide-and-Conquer in Parallel

```
1. Quicksort(A, l, r) {
2.     s = ParallelPartition(A, l, r);
3.     Spawn Quicksort(A, l, s-1);
4.     Spawn Quicksort(A, s+1, r);
5.     join(); // waiting spawned threads to finish
6. }
```

# Quicksort

- The key to the quicksort algorithm is the partition.
- Find a pivot point p such that left are all <= and right are all >=

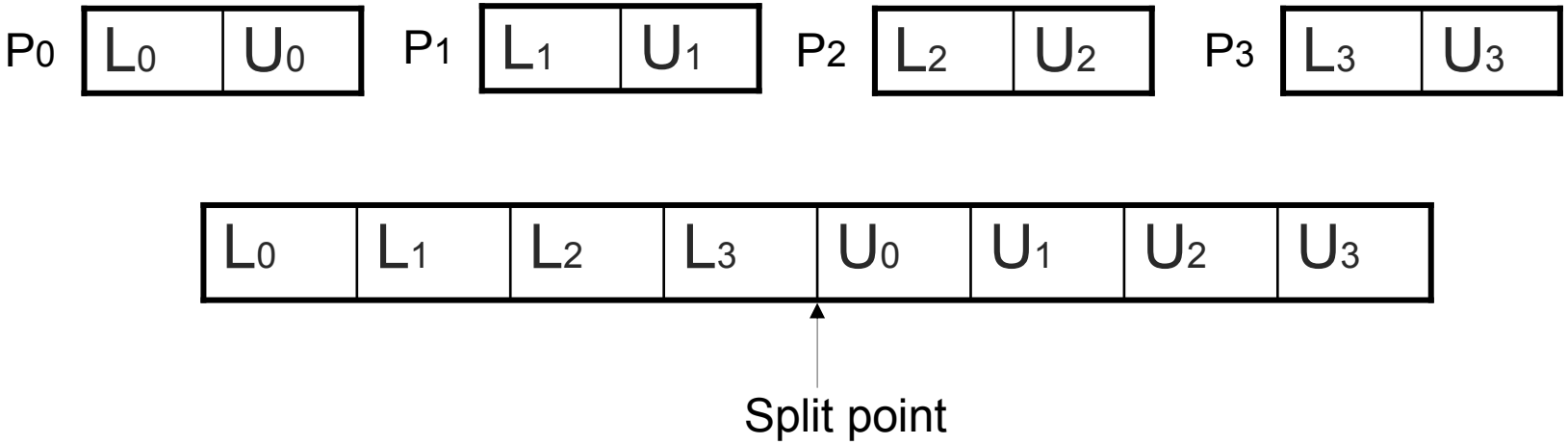


# Parallel Quicksort

- Again, we look at the classical Quicksort algorithm, try to add parallelism into it.
- It's the familiar divide-and-conquer idea used in MergeSort: the recursive divide-and-conquer part is, again, done in parallel.
- Similar to MergeSort where key is to parallelize merging, the key here is to parallelize the partition

# Parallel Partition

- Suppose we have p processors and we divide the entire array into p segments and each processor is assigned a segment.
- We pick a pivot point across all segments. Each processor i try to partition its own segment into two parts, $L_i$ and $U_i$.
- All $L_i$s are merged and $U_i$s are merged.



# Parallel Partition

- The complexity of serial partition is linear. The complexity of serial partition is n/p on average if the segments are evenly distributed.
- But we lose some of desired property of quicksort, such as in-place sort. In the second phase of parallel partition, we have to do merging, which essentially is an array-copy operation.