# Parallel Sorting

**Homework 3:** Will be posted soon.

Sorting is probably the most important problem in computing. There are many sequential sorting algorithms:

* Bubble sort
* Selection sort
* Insertion sort
* Merge sort
* Quick sort
* Heap sort
* Radix sort
* And many more

### A Difficult Problem to Parallelize

* There is one array
* all the elements of the array may be relocated many times.

### Two sorting algorithms

Two recursive sorting algorithms seem to be suitable for parallel execution:

* Merge sort
* Quick sort

### Sequential Recursive mergesort algorithm

### Sequential Recursive mergesort implementation is given below.

|  |
| --- |
| /\*\*  \* sequential mergesort  \* @author auyar  \*/  public class SeqMergeSort {  public static void mergeSort(long dizi[], int indis1, int indis2) {  if (indis1 == indis2) {  return;  }  int orta = (indis1 + indis2) / 2;  mergeSort(dizi, orta + 1, indis2);  mergeSort(dizi, indis1, orta);  merge(dizi, aux, indis1, orta + 1, indis2);  }  public static void merge(long d1[], long aux[], int i1, int i2, int last){  int indis1 = i1;  int indis2 = i2;  int indis3 = i1;  while(indis1<i2 && indis2<=last){  if(d1[indis1] < d1[indis2]){  aux[indis3] = d1[indis1];  indis1++;  indis3++;  }else{  aux[indis3] = d1[indis2];  indis2++;  indis3++;  }  }    while(indis1<i2){  aux[indis3++] = d1[indis1++];  }    while(indis2<=last){  aux[indis3++] = d1[indis2++];  }    for (int i = i1; i <= last; i++) {  d1[i] = aux[i];  }  }    public static long aux[];  public static void main(String[] args) {  long dd[] = {50, 70, 45, 30, 34, 78, 56, 10};  aux = new long[dd.length];  mergeSort(dd, 0, dd.length - 1);  print(dd);  }  public static void print(long[] dd) {  for (int i = 0; i < dd.length; i++) {  System.out.print(dd[i] + ", ");  }  System.out.println("");  }  } |

### Parallel Mergesort algorithm with Barriers

If there are n elements and m cores, then each core can sort a subsection of the array independently.

**blockSize = n/m**

* First block is sorted by the first thread,
* Second block is sorted by the second thread,
* Third block is sorted by the third thread,
* Etc.

**Assumption 1:** Assume that the number of elements to be sorted is divisible by the number of threads. So every block size is the same for all threads.

**Assumption 2:** Also assume that the number of threads is 2^k where k is an integer. So, always two blocks of equal size is merged.

**Merge Operations**

**First Merge operations:** Consecutive blocks are merged.

If there are k blocks, there will be k/2 merge operations. These operations can be handled by even numbered threads.

**Second merge operations:**

Consecutive blocks are merged. This time there will be only k/4 merge operations.

The threads that are the multiple of 4 may perfrom the merge operation.

**Barrier**

1. Threads wait at the barrier when they sorted their sub arrays.
2. Threads wait at the barrier when they merged their blocks.
3. Some threads loop without doing any merging.

**Shared variables**

**Array to be sorted**: this is a shared array. Different threads access different parts of the array. No synchronization is necessary.

**Auxillary array:** Multiple threads may work on the shared auxillary array. But each works on a different part of the thread. No synchronization is necessary.

**Merge Rounds**

When there are m threads:

1. First round of merge operations will be performed by m/2 threads
2. Second round of merge operations will be performed by m/4 threads
3. Third round of merge operations will be performed by m/8 threads
4. ….
5. Last round of merge operations will be performed by m/m threads

**Number of Merge Rounds**

Overall, there will be logm rounds of merging.

If we have 2 threads, there will be 1 round of merging.

If we have 4 threads, there will be 2 round of merging.

If we have 8 threads, there will be 3 round of merging.

Etc.

**Which threads will perform merge operations**

First round merging can be performed by even bumber of threads.

Second round merging can be performed by threads divisible by 4.

Third round merging can be performed by threads divisible by 8.

..

Last round merging can be performed by the first thread.

**Managing Active Threads**

We can keep track of active threads only.

With every iterations, number of active threads will be halved.

**Managing Thread ids**

We can assign new thread ids to active threads.

With every iterations, the id of each thread will be halved.

Unactive threads are assigned a big id. They loop idle.

**Stop Condition**

When the number of active threads become 0, merging stops.

**Keeping Track of Block Sizes**

Initially, the block size is the sequentially sorted subsections.

With every round of merging, block size is doubled.

|  |
| --- |
| import java.util.Arrays;  import java.util.concurrent.BrokenBarrierException;  import java.util.concurrent.CyclicBarrier;  /\*\*  \* each thread finds a local max.  \* in the reduction part, global max is found.  \*  \* @author Ahmet Uyar  \*/  public class MergeSortWithBarriers extends Thread {  private CyclicBarrier barrier;  int index;  public MergeSortWithBarriers(CyclicBarrier barrier, int index) {  super("thread " + index);  this.barrier = barrier;  this.index = index;  }  @Override  public void run() {  try {  int blockSize = array.length / numberOfThreads;  int first = index \* blockSize;  int last = (index + 1) \* blockSize;  Arrays.sort(array, first, last);  System.out.println(index + " sub sorts finished: ");  barrier.await();    int newid = index;  int activeThreads = numberOfThreads/2;  while(activeThreads>0){  if(newid %2 == 0)  newid = newid/2;  else  newid = 11111111;    if(newid<activeThreads){  int start = first;  int second = start+blockSize;  int third = second+blockSize;  merge(array, start, second, third);  }  blockSize \*= 2;  activeThreads = activeThreads/2;  barrier.await();  }    if(index == 0){  endTime = System.currentTimeMillis();  long duration = endTime - startTime;  System.out.println("par time: "+duration);  isSorted();  }  System.out.println(this.getName() + " has crossed the barrier");  } catch (InterruptedException ex) {  System.out.println("exception error message: " + ex.getMessage());  ex.printStackTrace();  } catch (BrokenBarrierException ex) {  System.out.println("exception error message: " + ex.getMessage());  ex.printStackTrace();  }  }    static int numberOfThreads = 16;  static long array[] = new long[1600000];  public static long aux[] = new long[1600000];  static long array2[] = new long[1600000];  static long startTime, endTime;  public static void main(String args[]) {  arrayInit();    startTime = System.currentTimeMillis();  Arrays.sort(array2);  endTime = System.currentTimeMillis();  long duration = endTime - startTime;  System.out.println("seq time: "+duration);  //creating CyclicBarrier with 3 parties i.e. 3 Threads needs to call await()  CyclicBarrier cb = new CyclicBarrier(numberOfThreads, null);    startTime = System.currentTimeMillis();  //starting each of thread  MergeSortWithBarriers threads[] = new MergeSortWithBarriers[numberOfThreads];  for (int i = 0; i < threads.length; i++) {  threads[i] = new MergeSortWithBarriers(cb, i);  threads[i].start();  }  System.out.println("main thraed has finished. ");  }  public static void arrayInit() {  java.util.Random r = new java.util.Random(20);  for (int i = 0; i < array.length; i++) {  array[i] = r.nextLong();  array2[i] = array[i];  }  }  public static void print() {  for (int i = 0; i < array.length; i++) {  System.out.println(i+": "+array[i]);  }  }    public static void isSorted() {  for (int i = 0; i < array.length-1; i++) {  if(array[i] > array[i+1]){  System.out.println("not sorted");  System.out.println(i+": "+array[i]);  System.out.println((i+1)+": "+array[i+1]);  return;  }  }  System.out.println("array is sorted.");  }    public static void merge(long d1[], int i1, int i2, int last){  System.out.println("in merge: "+i1 +", "+i2+", "+last);  int indis1 = i1;  int indis2 = i2;  int indis3 = i1;  while(indis1<i2 && indis2<last){  if(d1[indis1] < d1[indis2]){  aux[indis3] = d1[indis1];  indis1++;  indis3++;  }else{  aux[indis3] = d1[indis2];  indis2++;  indis3++;  }  }    while(indis1<i2){  aux[indis3++] = d1[indis1++];  }    while(indis2<last){  aux[indis3++] = d1[indis2++];  }    for (int i = i1; i < last; i++) {  d1[i] = aux[i];  }  }  } |

### Another Run method

We can compute the same thing by using mod operations.

We can have a variable to keep the mod amount at every iteration.

It starts with 2 and doubled with every iteration.

If a thread number is fully divisibale by this variable, it performes merge operation.

Threads are not assigned new ids in this method.

|  |
| --- |
| public void run() {  try {  int blockSize = array.length / numberOfThreads;  int first = index \* blockSize;  int last = (index + 1) \* blockSize;  Arrays.sort(array, first, last);  System.out.println(index + " sub sorts finished: ");  barrier.await();    int activeThreads = numberOfThreads/2;  int modAmount = 2;  while(activeThreads>0){  if(index%modAmount==0){  int start = first;  int second = start+blockSize;  int third = second+blockSize;  merge(array, start, second, third);  }  blockSize \*= 2;  activeThreads = activeThreads/2;  modAmount = modAmount \*2;  barrier.await();  }    if(index == 0){  endTime = System.currentTimeMillis();  long duration = endTime - startTime;  System.out.println("par time: "+duration);  isSorted();  }  System.out.println(this.getName() + " has crossed the barrier");  } catch (InterruptedException ex) {  System.out.println("exception error message: " + ex.getMessage());  ex.printStackTrace();  } catch (BrokenBarrierException ex) {  System.out.println("exception error message: " + ex.getMessage());  ex.printStackTrace();  }  } |

### Parallel Mergesort algorithm with Barriers 2

We can modify the parallel merge sort above and remove the two restrictions that we assumed.

**Data size can be any length:** The number of elemenets to be sorted may be any length. It may not be fully divisible by the number of threads.

We can make the last block a little longer or shorter. We make it shorter. All other blocks are the same size.

**Second:** the number of threads may be any number, not just 2^k.

This means that in some iterations, there may be an odd number of blocks to be merged.

When there are odd number of block, the last block is not merged in that iteration.

Consequently, the size of the last block may be much smaller than the others.

However, the size of other blocks will all be the same.

When merging, we set the last element differently for the last thread.

|  |
| --- |
| /\*\*  \* parallel iterative merge sort  \*  \* The number of elemenets to be sorted may be any length.  \* This means that the last block may be a little longer or shorter.  \* We make it shorter. All other blocks are the same size.  \*  \* the number of blocks may be any number, not just 2^k.  \* This means that there may be an odd number of blocks to be merged.  \* So, there may be one last block that may not be merged when others are merged.  \* In addition, the size of the last block may be much smaller than the others.  \* However, the size of other blocks will be the same.  \*  \* @author auyar  \*/  import java.util.Arrays;  import java.util.concurrent.BrokenBarrierException;  import java.util.concurrent.CyclicBarrier;  public class MergeSortWithBarriers2 extends Thread {  private CyclicBarrier barrier;  int index;  public MergeSortWithBarriers2(CyclicBarrier barrier, int index) {  super("thread " + index);  this.barrier = barrier;  this.index = index;  }  public void run() {  try {  double bs = array.length / (double)numberOfThreads;  int blockSize = (int)Math.ceil(bs);  int first = index \* blockSize;  int last = (index + 1) \* blockSize;  if (index + 1 == numberOfThreads) {  last = array.length;  }    // each thread sorts its own range independently and waits  Arrays.sort(array, first, last);  barrier.await();    //merge each of two consecutive ranges  // even threads merge, odd threads do nothing  int newThreadNo = index;  int newTotalThreads = numberOfThreads;  System.out.println("newThreadNo: "+newThreadNo);    while(newTotalThreads > 1){  if(newThreadNo%2 == 1)// if threadNo is odd,  newThreadNo = 1111111111;  else  newThreadNo = newThreadNo/2;  newTotalThreads = (int)Math.ceil(newTotalThreads/2.0);    if(newThreadNo< newTotalThreads){  first = newThreadNo\*blockSize\*2;  int middle = first + blockSize;  if(middle < array.length){  last = first + 2\*blockSize;  if(newThreadNo+1 == newTotalThreads)  last = array.length;  System.out.println("merge newThreadNo: "+newThreadNo + ", first: "+ first+" middle: "+middle+" last: "+last);  merge(array, first, middle, last);  }  }  blockSize = blockSize\*2;  System.out.println("newThreadNo: "+newThreadNo);    barrier.await();  }    // first thread will print the duration  // and checks whether the array is sorted  if(index == 0){  endTime = System.currentTimeMillis();  long duration = endTime - startTime;  System.out.println("par time: "+duration);  isSorted();  }    } catch (InterruptedException ex) {  System.out.println("exception error message: " + ex.getMessage());  ex.printStackTrace();  } catch (BrokenBarrierException ex) {  System.out.println("exception error message: " + ex.getMessage());  ex.printStackTrace();  }  }  static int numberOfThreads = 4;  static long array[] = new long[1000000];  static long array2[] = new long[1000000];  static long aux[] = new long[1000000];  static int counter = 0;    static long startTime, endTime;    public static void main(String args[]) {    System.out.println("array.length: "+array.length);    arrayInit();    startTime = System.currentTimeMillis();  Arrays.sort(array2);  endTime = System.currentTimeMillis();  long duration = endTime - startTime;  System.out.println("seq time: "+duration);    //no sequential part after they reach at the barrier  CyclicBarrier cb = new CyclicBarrier(numberOfThreads, null);    startTime = System.currentTimeMillis();  //starting each of thread  MergeSortWithBarriers2 threads[] = new MergeSortWithBarriers2[numberOfThreads];  for (int i = 0; i < threads.length; i++) {  threads[i] = new MergeSortWithBarriers2(cb, i);  threads[i].start();  }  }  public static void arrayInit() {  java.util.Random r = new java.util.Random(20);  for (int j = 0; j < array.length; j++) {  array[j] = r.nextLong();  array2[j] = r.nextLong();  }  }    public static void isSorted() {  for (int j = 0; j < array.length-1; j++) {  if(array[j] > array[j+1]){  System.out.println("array is not sorted");  System.out.println(j+ ": "+ array[j]);  System.out.println((j+1)+ ": "+ array[j+1]);  return;  }  }    System.out.println("array is sorted. ");  }    public static void merge(long d1[], int i1, int i2, int last){  int indis1 = i1;  int indis2 = i2;  int indis3 = i1;  while(indis1<i2 && indis2<last){  if(d1[indis1] < d1[indis2]){  aux[indis3] = d1[indis1];  indis1++;  indis3++;  }else{  aux[indis3] = d1[indis2];  indis2++;  indis3++;  }  }    while(indis1<i2){  aux[indis3++] = d1[indis1++];  }    while(indis2<last){  aux[indis3++] = d1[indis2++];  }    for (int i = i1; i < last; i++) {  d1[i] = aux[i];  }  }    } |

### Fork-Join Thread Programming

### For recursive programming, some library classes are provided to do parallel programming.

### The general framework is as follows:

|  |
| --- |
| if (my portion of the work is small enough or there are enough threads)  do the work directly  else  split my work into two pieces  invoke the two pieces and wait for the results  combine the results if necessary |

This method constructs two threads recursively if necessary. It waits for the constructed threads to finish.

### Recursive and parallel sum calculation

### We would like to find the sum of elements in an array. We want to do it recursively and in parallel.

### We define a global array.

### We define a threshold, when there are less elements, each thread calculates the answer. Recursion stops. It does not continue to construct two new threads.

### We also have a global variable to hold the sum of all numbers. Update of this variable must be synchronized.

### We assign an id to each thread.

### pool.invoke(fb) in main method waits until all the threads are done.

### invokeAll(th1, th2); waits until these two threads are done.

|  |
| --- |
| import java.util.concurrent.ForkJoinPool;  import java.util.concurrent.RecursiveAction;  /\*\*  \* sume of all elements in an array  \* it performs the calculation recursively and uses parallel programming  \* ahmet uyar  \*/  public class ForkJoinSum extends RecursiveAction {  private int start;  private int length;  private int id;  public ForkJoinSum(int start, int length, int id) {  this.start = start;  this.length = length;  this.id = id;  System.out.println("my id: "+ id + " my start index: "+this.start+" my length: "+this.length);  }  // Average pixels from array, write results into destination.  protected void computeDirectly() {  long result = 0;  for (int index = start; index < start + length; index++) {  result += array[index];  }  addToSum(result);  }  protected void compute() {  if (length < threshold) {  computeDirectly();  return;  }  int split = length / 2;  ForkJoinSum th1 = new ForkJoinSum(start, split, 2\*id);  ForkJoinSum th2 = new ForkJoinSum(start + split, length - split, 2\*id+1);  invokeAll(th1, th2);  }    public static synchronized void addToSum(long partialSum) {  globalSum += partialSum;  }  protected static int threshold = 10000;  protected static long globalSum = 0;  public static int array[] = new int[100000];  // Plumbing follows.  public static void main(String[] args) throws Exception {  arrayInit();  long seqSum = seqSum();  System.out.println("seq sum: "+ seqSum);  System.out.println("Array size is " + array.length);  System.out.println("Threshold is " + threshold);  int processors = Runtime.getRuntime().availableProcessors();  System.out.println(Integer.toString(processors) + " processor"  + (processors != 1 ? "s are " : " is ")  + "available");  ForkJoinSum fb = new ForkJoinSum(0, array.length, 1);  ForkJoinPool pool = new ForkJoinPool();  long startTime = System.currentTimeMillis();  pool.invoke(fb);  long endTime = System.currentTimeMillis();  System.out.println("sum took " + (endTime - startTime)  + " milliseconds.");    System.out.println("par sum: "+globalSum);  if(globalSum == seqSum)  System.out.println("they are the same");    }  public static void arrayInit() {  java.util.Random r = new java.util.Random(20);  for (int j = 0; j < array.length; j++) {  array[j] = r.nextInt();  }  }  public static long seqSum() {  long sum = 0;  for (int j = 0; j < array.length; j++) {  sum += array[j];  }  return sum;  }  } |

### Getting the result of each thread by the parent thread

### Instead of accumulating the answer in a global shared variable, parent threads can get the results of each thread.

### The value of the first thread represents the global sum.

|  |
| --- |
| import java.util.concurrent.ForkJoinPool;  import java.util.concurrent.RecursiveAction;  /\*\*  \* sum of all elements in an array  \* it performs the calculation recursively and uses parallel programming  \* each thread gets the results of the children thread  \* ahmet uyar  \*/  public class ForkJoinSum2 extends RecursiveAction {  private int start;  private int length;  private int id;  private long result;    public ForkJoinSum2(int start, int length, int id) {  this.start = start;  this.length = length;  this.id = id;  System.out.println("my id: "+ id + " my start index: "+this.start+" my length: "+this.length);  }  // Average pixels from array, write results into destination.  protected void computeDirectly() {  result = 0;  for (int index = start; index < start + length; index++) {  result += array[index];  }  }  protected void compute() {  if (length < threshold) {  computeDirectly();  return;  }  int split = length / 2;  ForkJoinSum2 th1 = new ForkJoinSum2(start, split, 2\*id);  ForkJoinSum2 th2 = new ForkJoinSum2(start + split, length - split, 2\*id+1);  invokeAll(th1, th2);  result = th1.result + th2.result;  }    protected static int threshold = 10000;  public static int array[] = new int[100000];  // Plumbing follows.  public static void main(String[] args) throws Exception {  arrayInit();  long seqSum = seqSum();  System.out.println("seq sum: "+ seqSum);  System.out.println("Array size is " + array.length);  System.out.println("Threshold is " + threshold);  int processors = Runtime.getRuntime().availableProcessors();  System.out.println(Integer.toString(processors) + " processor"  + (processors != 1 ? "s are " : " is ")  + "available");  ForkJoinSum2 fb = new ForkJoinSum2(0, array.length, 1);  ForkJoinPool pool = new ForkJoinPool();  long startTime = System.currentTimeMillis();  pool.invoke(fb);  long endTime = System.currentTimeMillis();  System.out.println("sum took " + (endTime - startTime)  + " milliseconds.");    System.out.println("par sum: "+fb.result);  if(fb.result == seqSum)  System.out.println("they are the same");    }  public static void arrayInit() {  java.util.Random r = new java.util.Random(20);  for (int j = 0; j < array.length; j++) {  array[j] = r.nextInt();  }  }  public static long seqSum() {  long sum = 0;  for (int j = 0; j < array.length; j++) {  sum += array[j];  }  return sum;  }  } |

### Stopping recursion by using the number of threads

### When there are enough number of threads, we can stop constructing new threads.

### For example,

### if the program is running on an 16 core system, it can construct 16 threads on the leaves,

### if the program is running on an 8 core system, it can construct 8 threads on the leaves,

### etc.

### To construct 16 threads, the threads that have the id less that 16 construct two new threads, the ones that have 16 or more thread ids will compute directly.

### Similarly, to construct 8 threads, the threads that have the id less that 8 construct two new threads, the ones that have 8 or more thread ids will compute directly.

|  |
| --- |
| import java.util.concurrent.ForkJoinPool;  import java.util.concurrent.RecursiveAction;  /\*\*  \* sum of all elements in an array  \* it performs the calculation recursively and uses parallel programming  \* each thread gets the results of the children thread  \* stop recursive construction of threads by using the number of threads  \* ahmet uyar  \*/  public class ForkJoinSum3 extends RecursiveAction {  private int start;  private int length;  private int id;  private long result;    public ForkJoinSum3(int start, int length, int id) {  this.start = start;  this.length = length;  this.id = id;  System.out.println("my id: "+ id + " my start index: "+this.start+" my length: "+this.length);  }  // Average pixels from array, write results into destination.  protected void computeDirectly() {  result = 0;  for (int index = start; index < start + length; index++) {  result += array[index];  }  }  protected void compute() {  if (id >= numberOfThreads) {  computeDirectly();  return;  }  int split = length / 2;  ForkJoinSum3 th1 = new ForkJoinSum3(start, split, 2\*id);  ForkJoinSum3 th2 = new ForkJoinSum3(start + split, length - split, 2\*id+1);  invokeAll(th1, th2);  result = th1.result + th2.result;  }    public static int array[] = new int[100000];  public static int numberOfThreads = 8;  // Plumbing follows.  public static void main(String[] args) throws Exception {  arrayInit();  long seqSum = seqSum();  System.out.println("seq sum: "+ seqSum);  System.out.println("Array size is " + array.length);  System.out.println("number Of Threads is " + numberOfThreads);  int processors = Runtime.getRuntime().availableProcessors();  System.out.println(Integer.toString(processors) + " processor"  + (processors != 1 ? "s are " : " is ")  + "available");  ForkJoinSum3 fb = new ForkJoinSum3(0, array.length, 1);  ForkJoinPool pool = new ForkJoinPool();  long startTime = System.currentTimeMillis();  pool.invoke(fb);  long endTime = System.currentTimeMillis();  System.out.println("sum took " + (endTime - startTime)  + " milliseconds.");    System.out.println("par sum: "+fb.result);  if(fb.result == seqSum)  System.out.println("they are the same");    }  public static void arrayInit() {  java.util.Random r = new java.util.Random(20);  for (int j = 0; j < array.length; j++) {  array[j] = r.nextInt();  }  }  public static long seqSum() {  long sum = 0;  for (int j = 0; j < array.length; j++) {  sum += array[j];  }  return sum;  }  } |

### MergeSort with Fork-Join Threads

### When the array is divided to each thread equally, each thread performs sequential sorting.

### After two children are done sorting, parent thread merges two sorted sub sections.

### Thread ids are important for the correct auxiliary array.

|  |
| --- |
| import java.util.concurrent.ForkJoinPool;  import java.util.concurrent.RecursiveAction;  import java.util.Arrays;  /\*\*  \* mergesort with Fork-Join Threads  \* Ahmet uyar  \*/  public class MergeSortWithForkJoin extends RecursiveAction {  private long[] mSource;  private int mStart;  private int mLength;  private int id;  public MergeSortWithForkJoin(long[] src, int start, int length, int id) {  mSource = src;  mStart = start;  mLength = length;  this.id = id;  System.out.println("my id: "+ id + " my start index: "+mStart+" my length: "+mLength);  }  // Average pixels from source, write results into destination.  protected void computeDirectly() {  Arrays.sort(mSource, mStart, mStart+mLength);  }  @Override  protected void compute() {  if (mLength <= blockSize) {  computeDirectly();  return;  }  int split = mLength / 2;  MergeSortWithForkJoin th1 = new MergeSortWithForkJoin(mSource, mStart, split, 2\*id);  MergeSortWithForkJoin th2 = new MergeSortWithForkJoin(mSource, mStart + split, mLength - split, 2\*id+1);  invokeAll(th1, th2);  merge(source, th1.mStart, th2.mStart, th2.mStart+th2.mLength);  }    public static long source[] = new long[1000000];  static long source2[] = new long[1000000];  static int cores = Runtime.getRuntime().availableProcessors();  public static long aux[] = new long[source.length];  protected static int blockSize = (int)Math.ceil(source.length/(double)cores);    public static void main(String[] args) throws Exception {  arrayInit();  long startTime = System.currentTimeMillis();  Arrays.sort(source2);  long endTime = System.currentTimeMillis();  long duration = endTime - startTime;  System.out.println("seq time: "+duration);    MergeSortWithForkJoin fb = new MergeSortWithForkJoin(source, 0, source.length, 1);  ForkJoinPool pool = new ForkJoinPool();  startTime = System.currentTimeMillis();  pool.invoke(fb);  endTime = System.currentTimeMillis();  duration = endTime - startTime;  System.out.println("sorting took " + duration + " milliseconds.");    isSorted();  }    public static void arrayInit() {  java.util.Random r = new java.util.Random(20);  for (int j = 0; j < source.length; j++) {  source[j] = r.nextLong();  source2[j] = source[j];  }  }  public static void merge(long d1[], int i1, int i2, int last){  int indis1 = i1;  int indis2 = i2;  int indis3 = i1;  while(indis1<i2 && indis2<last){  if(d1[indis1] < d1[indis2]){  aux[indis3] = d1[indis1];  indis1++;  indis3++;  }else{  aux[indis3] = d1[indis2];  indis2++;  indis3++;  }  }    while(indis1<i2){  aux[indis3++] = d1[indis1++];  }    while(indis2<last){  aux[indis3++] = d1[indis2++];  }    for (int i = i1; i < last; i++) {  d1[i] = aux[i];  }  }    public static void isSorted() {  for (int j = 0; j < source.length-1; j++) {  if(source[j] > source[j+1]){  System.out.println("array is not sorted");  System.out.println(j+ ": "+ source[j]);  System.out.println((j+1)+ ": "+ source[j+1]);  return;  }  }    System.out.println("array is sorted. ");  }  } |