Practical Parallel Programming –

a plan for a B.S. course on how to design efficient parallel algorithms.

Arne Maus
Department of Informatics,
University of Oslo
Norway
arnem@ifi.uio.no

Stein Gjessing,
Department of Informatics,
University of Oslo
Norway
steing@ifi.uio.no

***Abstract*—This paper describes a new course in parallel programming at the University of Oslo that emphasis thread based parallel programming in Java on multicore computers. The quality of a parallel algorithm is evaluated by its speedup, since the main reason for making a parallel algorithm is to create a faster program. This course stresses that there are many possible correct parallel algorithms for a given problem. Since the course focuses on measured efficiency, a more theoretical approach like the PRAM (Parallel Random Access Memory) model is not considered. However, we are examining those practical factors that contribute most to the total running time – like the startup time of a thread based solution, the Java JIT-compilation, the number of synchronizations, and the locality of data in the caches. A partly new pattern for designing a parallel algorithm by dividing it into fully parallel parts with barrier synchronization between each part is taught. Instead of synchronization, when updating shared variables, each thread, if possible, receives a copy of the shared data that it updates locally and without synchronization. In the next part in the algorithm these partial updates are consolidated. This pattern works on many, but of course not on all problems. Also, many, but not all of the problems of parallel programming are covered, as they do not all apply to treads that essentially all execute the same code. This paper presents the first and last problem given to the students in the course, and we show alternative solutions for their speedup. This course is given for the first time in the spring semester 2014, thus this paper presents the plans for the course.**

Keywords-Parallel algorithms, Java, Threads, Speedup, Barrier synchronization, undergraduate course.

I. INTRODUCTION

Since every computer and most smart phones one buys these days have more than one processing core, parallel programming is forced upon us. The chip manufacturers can not deliver what we really want, which is ever faster processors. The heat generated on a chip increases at least linearly with the clock frequency, and will make it malfunction and eventually melt above 4 GHz with today's technology and air cooling. Instead standard multi core processors with 2-16 processor cores are now manufactured, but non standard processors with from 50 to 100 cores are also available [15,16], and the race for more processing cores on a chip doesn't stop there. Each of these cores has the processing power of a single CPUs sold some 5−10 years ago. Many of these cores are also hyper-threaded, where some of the circuitry is duplicated such that each core can run two threads concurrently with no or little interference. Also, we see today standard servers with up to 4 such hyper-threaded multi core processors, each with 8(16) hyper-threaded cores, meaning that up to 32(64) threads can run in parallel. The conclusion to all this parallelism is that if we want to create faster programs, we must design parallel algorithms that exploit these new machines.

In this paper we show how we created a course at the undergraduate level that demonstrates that there are more than one way to parallelize an algorithm on a multi core CPU using Java threads. We want to teach the students paradigms for dealing with shared variables in parallel programs and how one can speed up algorithms. We do this by showing the students many solutions to a number of problems. In this paper we demonstrate this approach using two well known problems: First finding the maximum value in an integer array of length n, and secondly, sorting n integers using the ordinary Right Radix (LSD – Least Significant Digit) algorithm. The first problem is also a sub problem of the second.

Paradigms for parallel programming are abundant [1, 2, 4, 5]. Lots of textbooks are written with these words in its title. Usually the books present a full exposure to parallelization, almost always covering MPI style of programming in large grids of workstations using message passing, OpenMP [20] on multi-platform shared memory multiprocessing programming in C, C++, and Fortran, but usually also covering Java style programming with threads on multi core shared memory computers. The main distinction between the two common ways to parallelize a program is covered: Either by partitioning data and let each processor do all operations on its part of the data elements (map-reduce), or by (nested) for-loops that are parallelized by letting each processor do some operations on all data. It has also been proposed that parallel programming should take lessons from databases where most concurrency problems already have been dealt with [7]. Textbooks also treat parallelization on special purpose processors, like the Single-Instruction-Multiple-Data GPUs that are shown to achieve remarkable speedup of more than 100 in special cases when parallelizing standard operations on large matrixes, but also shown to be very slow when executing programs with conditional branching (if-statements) or

working on small matrixes of less than 100 rows and columns.

The focus of our course is how to get speedup on a shared memory computer with one or more multi core CPUs, each having multilevel caches – i.e. on non uniform memory access machines. We assume k (hyperthreaded) cores and for simplicity we then test our problems with k threads. We also assume that we program this in Java [6] with threads, but we assume that the mechanisms demonstrated here carries over to thread programming in other imperative languages like C, C++ or C#. All our problems are initially defined by a sequential algorithm. This algorithm usually consists of m steps ($m \geq 1$) – often in the form of some loops. We demonstrate different parallel paradigms to the students by generating alternative parallel solutions to the same problem, and demonstrate their relative efficiency (or lack of such), by comparing their execution times with the sequential algorithm and the other parallel solutions. The approach taken in this course is practical in the sense that we know that starting and stopping threads and synchronizing on access to shared data-elements create overhead that is not present in the sequential algorithm, and also that all modern computers have a non uniform memory because of the caches [13,14]. Therefore the programs recorded running times with random data is reported and are the yard-stick used for evaluating their efficiency.

When making this course at the Department of Informatics, University of Oslo, we already had three courses that cover:

1. Parallel algorithms using MPI on a grid of workstations – like the main university cluster with about 10000 cores.
2. The use of special chips like GPUs and the Cell processor.
3. A theoretical course on using finite state machine logic to model concurrent systems (one example: The Dining Philosophers) and then generate Java programs from this model.

Our course comes in addition to these courses, since we think that there is room for a course in Practical Parallel Programming stressing the speedup of algorithms in Java. After all it is real, measured speedup that is the "proof of the pudding" in parallel programming.

II. FOUR KINDS OF OVERHEAD FROM PARALLELIZATION

There are at least four different forms of overhead when doing parallel computations in a language like Java.

The first kind of overhead is concerned with creating and starting the k threads. In some of the examples in the course we use a common technique creating a pool of k threads by using two CyclicBarriers [10]. The overhead from this startup stage is in the order of 1-4 milliseconds (k = 4-64) and are not included in the reported running times, since this pool of threads can be reused as many times as we want for running a parallel algorithm.

The most important overhead in parallel programming is in theory covered by Amdahl's law [3] – the limitation of the speedup of an algorithm by its sequential parts. The second to fourth kinds of overhead are covered by this law, or practical considerations of the law.

The second kind of overhead we find when m >1 (the number of steps in the algorithm), because then the program has to synchronize the threads between each step, typically because one thread has to read what the other threads have produced (written) in the previous step. In the two examples used in this paper, the numbers of such steps range from 2 to 12, the synchronization is done with the help of a (third) CyclicBarrier object. The overhead introduced by this synchronization is of course included in the reported running times. Note that this overhead is also fairly constant and independent of n, the size of the problem. The number of steps is 2 in finding the maximum value, a constant, while the number steps of both the sequential and parallel version of the Right Radix sorting algorithm is proportional to log n, because we sort on more digits the larger the maximum number to be sorted is. In practice, with sorting with 1 to 3 digits, this will be from 6 to 18 synchronization steps.

The third overhead is caused by synchronization in parallel programs by access to shared variables. These are variables in the program that are written by at least one thread and at the same time read by at least one other thread. Both reading and writing to such variables has to be synchronized, as any standard textbooks will tell us, [4, 6, 8]. This is the overhead we will tell the students to eliminate or minimize. Often the number of such synchronizations is proportion to n, the size of the problem, if we are forced to do synchronization for each thread´s access.

The fourth overhead is JIT (Just In Time) compilation. The first time an algorithm runs, its execution time is a sum of a JIT compilation and a mixed interpreted and compiled execution. Also, a JIT compilation is partial and might kick in the second (or third) time the algorithm runs. The first time a parallel algorithm runs it typically takes $6 - 10$ millisec. longer because of the JIT compilation. All figures in this paper are from the median of 11 runs, hence factoring out the JIT compilation from the results. We also note that we get a more effective JIT compilation of methods that are called many times than of the same loops running the same number of times inside a lager method that is only called once. This is an observed effect we are not yet able to explain.

To conclude this section, we emphasis that measuring the execution time of a Java program is not a straight forward exercise, and one never gets exactly the same result twice. But since the whole point of making a parallel algorithm is to make a faster algorithm, there seems to be no alternative to measurements if we want to conclude that the parallel version is faster, The students will learn by own experience that a parallel version of an algorithm is certainly much

harder to design than a sequential one. In the examples demonstrated to the students in this course, the parallel versions of the programs have at least double the number of lines of code compared to the corresponding sequential programs.

III. Essential knowledge for the students

In order to make a correct and at the same time effective parallel algorithm in Java, at the end of the course the student must:

i) Know how to make an effective sequential solution.

ii) Know the race-condition problem and how a many-level memory system makes each thread often see different values of shared variables.

iii) Know that the java byte code compiler, the JVM and JIT compilation, all makes optimizations, where statements are inter-changed or deferred. Java promises an 'as if sequential' semantics; i.e. the compiler should give us a program that produces the same results as if the statements were executed in the order in which they occur in the program. But the compiler can use any optimization technique that keeps this semantics, such as reordering or deferring statements and method calls.

iv) Understand the necessity that all threads synchronize on the same synchronizing variables in order to clean up, i.e. execute the deferred actions and flush all shared values to main memory so all threads see the same contents of memory.

v) Know that all threads working on the algorithm must have terminated before the results of the algorithm is reported (the termination problem).

vi) Have a guide to how to treat shared variables. To make things easy for the students, the following rule is taught on accessing shared variables before a synchronization:

   a. If no thread is writing to a variable, all threads can read it.
   b. If one thread is writing to a variable, only that thread can read that variable.
   c. No two threads can write to a shared variable in the same section (between two synchronizations).

Included in an in depth knowledge of how to make a good parallel algorithm, knowledge about concurrency is needed in addition to the knowledge about how to speed up a concurrent algorithm. A producer-consumer example is taught to illustrate that more complex reasoning and rules that must be applied when dealing with a concurrent program. It is stressed that even though this is a course in parallelization of algorithms, also the problems that exist in a concurrent programming environment must be mastered. Also functional decompositions are demonstrated by letting the students do calculation of Pi as a sum of a set of arctan-series [22] – each of these series can be calculated in parallel, but since in practice one of the most efficient such formula (5.19 in [22]) only contains 3 arctan series, the maximum speedup we get is less than 3. Functional decomposition seldom scales with k, the number of cores.

IV. Four ways to deal with shared variables

As described above, we teach the students that write access to shared variables has to be controlled. These variables often describe some essential part of our algorithm, and most algorithms have such variables (quicksort and mergesort doesn't have them explicitly and are what is commonly known as 'embarrassingly parallel' problems). Four possible ways to deal with these shared variables are emphasized:

1) **Sequential update.** If we have more than one step in our algorithm, some problems are such that it, even in the parallel version, is best to isolate one step (usually the first or last), and perform this step sequentially, when the rest of the steps are performed in parallel. The threads can then test their threadIndex and let $thread_0$ do this sequential step alone. Of course this is not optimal, but if all the other steps can be parallelized, this might in total be a speedup, even though we now fall into the realm of Amdahl's law [3].
2) **Synchronized access.** Shared variables may be updated in parallel threads, but each access (read or write) has to be protected by a synchronizing mechanism. The original Java definition had synchronized methods but since Java 1.5 more lightweight mechanisms like CyclicBarrier, Semaphore and AtomicIntegerArray have been introduced in the libraries java.util.concurrent and java.util.concurrent.atomic. We teach the students these synchronizing primitives.
3) **Duplicate data**. Shared variables may be duplicated to each thread. Then each thread can work locally on its subset of the variables with unsynchronized updates. When partial updates from all threads have been performed, the data may be recombined to give the result of these steps of the algorithm after a synchronization.
4) **Several ways to partition data**. One would usually think of a data partition of a problem as $thread_0$ taking ownership of the first n/k^{th} part of the data elements in the problem; $thread_1$ the next n/k^{th} elements, etc. We encourage the students to think alternatively. For example, another way to partition data may be, if we know their maximum value m, to let $thread_0$ own the elements with the m/k smallest values, $thread_1$ the elements with the next m/k smallest elements, etc. We can choose one or the other of these strategies, or both,

meaning that in some of the sections of the algorithm, $thread_i$ will work on the i'th set of element, and in other sections $thread_i$ works on the i'th set of the values.

We will in the two following examples, that are part of the course, employ both these techniques and demonstrate their usefulness.

V. EXAMPLE 1: FINDING THE MAXIMUM OF N INTEGERS

This problem is very simple, and the sequential code is a five-liner with one shared variable, max, as shown in Program1.

```
int findMax(int [] a) {
  int max = a[0];
  for (int i = 1; i < a.length; i++)
     if (a[i] > max) max =a[i];
  return max;
}
```

Program 1. Sequential code for finding the largest value in an array a. The array a[] is initialized randomly (uniform (0:n-1)), before the call to findMax. The choice of which random function to use is not important here.

```
synchronized void testAndUpdate (int elem) {
      if (elem > max) max = elem;
   }
  int [] a ;
  int max = 0;

  int FindM (int left, int right) {
      for (int i = left; i <= right; i++)
         testAndUpdate(a[i]);
     return  max;
   }
```

Program 2. A first Parallel solution MaxSync, using a synchronized method for accessing and updating max. We assume a wrapping method in each thread calculating the correct partition of a[] for each thread: a[left..right]. Note that we do not duplicate the shared variable max in FindM.

```
  // duplicate shared variable, one for each thread
   int [] localMax = new int [numThreads];

   void findLocalMax(int left, int right, int threadIndex) {
      int max = a[left];
      for (int i = left+1; i <= right; i++){
         if (a[i] > max) max = a[i];
      }
      localMax[threadIndex] = max;
   }

  int findMax (int [] localMax, int numThreads) {
      int max = localMax[0]; // duplication of shared variable
      for (int i = 1; i < numThreads; i++)
            if (localMax[i] > max) max = localMax[i];
      return max;
  }
```

Program 3. A second parallel solution, duplicateMax, that duplicates the shared variable max to each thread. First each thread finds the maximum value in its segment of a[left..right] and posts it in an array localMax. A second max-finding operation (findMax) is then called, when all threads have posted their local max (after a CyclicBarrier synchronization not shown here), and the global max is found.

n	100	6 400	409 600	26 214 400	104 857 600
Sequen-tial	0,000	0,006	0,374	25,92	109,3
Synchr. Method	0,082	0,674	41,684	2719,32	10794,9
Duplicate Data	0,082	0,368	0,237	10,08	32,5

Table 1. Execution time in milliseconds for finding the maximum of n numbers by using a sequential progran, by using a synchronized method and by the duplication of a shared variable.. The tests were performed on an 4(8) core Intel 7i, 2.9 GHz laptop.

We ask the students to run the different versions of their program on different sets of data and discuss the results. In table 1 the running time of the two parallel algorithms in example 1 are compared with the sequential algorithm. Both parallel algorithms distribute n/k of the array a[] to each thread. We notice that the synchronized method algorithm, that does not duplicate the shared max variable in any way, is basically 100 times slower than the sequential algorithm. The duplicateMax − algorithm eventually becomes 3X faster than the sequential algorithm even though it introduces a second step in the algorithm (not before every thread has found its localMax, can the global max be found). This speedup we explain by the duplication of shared data to local variables. The "speed down" of the synchronized method version is explained by the fact the synchronization on all of the data will add much overhead and in addition make the parallel program almost sequential since it has to access all data in sequence. Most students would say that the synchronized algorithm with one max variable is obviously very slow. We ask them to make a third parallel algorithm where each thread first computes the local max (like in program 3), and then update the global max in a synchronized method. The lessons learned form these parallel programs and the discussions about their running times are very important in this course.

VI. EXAMPLE 2: RIGHT RADIX SORTING

The most common form of radix sorting, where one start by sorting on the least significant digits first, is assumed to be well known and can be found in any textbook on sorting [19, 12]. This is a much more complicated algorithm than findMax. There are two set of shared variables in this algorithm, the array a[] to be sorted and an array count[] that keeps track of how many elements in a[] have different digit values. We see that the sequential algorithm (Program 4) has used one of the above stated paradigms by duplication of the elements in a[] to an array b[] to allow correct placement of the elements in a[] in a sorted order by copying from a[] to b[]. The sequential algorithm in Program 4 is taken from [13] and we see that it consists of four stages (A-D). We avoid a

copy back stage b[] to a[] at the end because we have an even number of digits to sort on and swap array parameters a[] and b[].

```
static void radixSort (int [] a) {
        int max = 0, numBit = 1;

        // A) Find max
        for (int i = 0 ; i < a.length ; i++)
            if (a[i] > max) max = a[i];

        while (max >= (1<<numBit)) numBit++;
        int bit1 = numBit/2,
           bit2 = numBit-bit1;

        int [] b = new int [a.length];
        radixSort2( a,b, bit1,0);
        radixSort2( b,a, bit2,bit1);
    }

 static void radixSort2( int [] a, int [] b, int maskLen, int shift){
     int  acumVal = 0, j;
     int mask = (1<<maskLen) -1;
     int [] count = new int [mask+1];

     // B) count=the frequency of each radix value in a[]
     for (int i =0; i < a.length; i++)
       count[(a[i]>>shift) & mask]++;

     // C) Add up in 'count' - accumulated values
     for (int i = 0; i <= mask; i++) {
          j = count[i];
          count[i] = acumVal;
          acumVal += j;
      }

     // D) move numbers in sorted order a to b
     for (int i = 0; i < a.length; i++)
       b[count[(a[i]>>shift) & mask]++] =  a[i];

    }/* end radixSort2 */
```

Program 4. *Sequential Right Radix sorting on two digits. It consists of 4 stages. Stage A is finding the maximum value in a[], which has been examined in the previous example. The next stages B,C and D (which are repeated twice, once for each digit we sort on) have a shared array count[] which keeps track of first how many, and later where the elements with different values of the current digit should be placed sorted on that digit.*

First, in a wrapper method radixSort, the maximum value in array a[0..n] is calculated and the number of bits in each of the two digits are determined (for simplicity two digits are used here). This is the findMax problem from our first example, and we will later in our last parallel solution use the parallel duplicateData solution here. The next three stages are done once for each digit. In stage B we count how many data elements there are of each value. In stage C these values are added together to form a data structure where count[i] points to where the next element with value 'i' should be moved (in the first digit moved from a[] to b[]) – see figure 1. In stage D these moves are made.

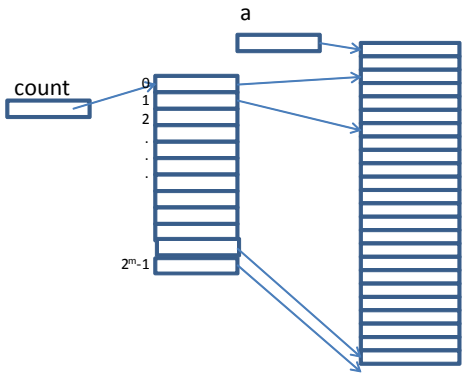

Figure 1. *The use of array count in any radix algorithm when sorting on a digit with* numbit *bits. The illustration is after sorting. We see that there are two elements in a[] with the value 0 on that digit, 4 elements with value 1,…,and 1 element with value* $2^{numbit}-1$.

We will now try two solutions to a parallel LSD Radix algorithm, in the first solution using a synchronization with an 'AtomicIntegerArray count', in the second solution with a duplication of the shared variables count[] to each thread.

We also note that there is a difference between the three stages we try to parallelize. In stage B we need an atomic increment of the different elements of count[]. In phase C we make accumulated sums such that count[i] = sum(count[j]) $\forall j < i$. In stage D the now accumulated count[] is read and incremented. Note also that in step D both a[] and count[] are read and b[] and count[] are modified. That makes it harder to parallelize stage D.

A. A synchronized solution using AtomicIntegerArray

First we decide that each $thread_i$ gets an n/k part of the array. Since the lightweight synchronization primitive AtomicIntegerArray has just the required synchronized operation 'getAndIncrement(int index)' that is needed in stage B, we substitute the increment of count[i] with 'getAndIncrement(i)'. In stage C we don't get any help from the operations in AtomicIntegerArray, so we just let one of the threads ($thread_0$) do it sequentially. But since this is $v=2^{numbit}$ operations, and not n operations (say numbit=10, then v=1024) it might not be all that important.

```
void radixSort2( int [] a, int [] b,  AtomicIntegerArray count,
      int left, int right, int  maskLen, int shift,  int threadIndex,
      boolean firstDigit){

     int  acumVal = 0, j;
     int  mask = (1<<maskLen) -1;

      // B) count=the frequency of each radix value in a
      for (int i = left; i <=right; i++)
         count.getAndIncrement((a[i]>> shift) & mask);

  //SYNC 1 here----------
```

```
 try { sync.await();    // wait on all the threads
 } catch (Exception ex) { return;}

 // C) Sequential operation
 if (threadIndex == 0) {
   // b) Add up in 'count' - accumulated values
    for (int i = 0; i <= mask; i++) {
          j = count.get (i);
          count.set(i, acumVal);
          acumVal += j;
 }    }

 //SYNC 2 here ------------------------
 try { sync.await();    // wait on all the threads
 } catch (Exception ex) { return;}

 if ( firstDigit ){
    // D) move numbers in sorted order a to b
    for (int i = left; i <= right; i++)
       b[count.getAndIncrement((a[i]>>shift) & mask)] = a[i];
 }else if (threadIndex ==0) {
     // sequential operation in all digits except digit 1
    for (int i = 0; i <a.length; i++)
       b[count.getAndIncrement((a[i]>>shift) & mask)] = a[i];
  }

  //SYNC 3 here ------------------------
  try { sync.await();    // wait on all the threads
  } catch (Exception ex) { return;}

}/* end radixSort2 */
```

Program 5. *The AtomicIntegerArray parallel solution to sorting on one digit. The method RadixSort is basically the same as the sequential solution.*

Stage D is the most difficult. Initially one could imagine that this could be done in parallel for all digits, but it is only for the least significant digit that all threads can use the getAndIncrement(i) operation and do this in full parallel.

On the second digit (and all subsequent digits), however, there is an ordering that must be respected. All elements with 0-es in the second digit and 0- also in the first digit must be copied before those elements with 0 in their second digit and 1 in the first digit are copied, and so on. Hence, in stage D it is also necessary to go sequential in all but the first digit and let $thread_0$ do the job. In a two digit sort of a[] we see that 3 of the 6 stages gets executed in parallel and 3 sequentially, and of the stages with n operations, 3 out of the 4 stages (B and D) are executed in parallel.

B. The duplicate shared variables solution

We distribute among the threads the elements in a[] both by value and by index. In stage A, B and D each thread basically works with its n/k part of the elements in a[] such that each thread owns a[leftElem..rightElem]. In stages C1 and C2, however, the algorithm works on all elements with values {leftValue ..rightValue}. First, in C1 a thread accumulates these values for all the local count arrays, and then in C2 updates (after barrier synchronization) these arrays stored in the two-dimensional allCounts[][] for the space needed above its own values. All these values are now read only and all threads can from these data calculate their own local count arrays from these data.

Stage D is now trivial (like the sequential solution) because the localCount in each thread points exactly to where elements (in the element-distribution to the threads) shall be moved, and can be done in parallel by all threads.

We will now be able to both avoid all unnecessary synchronization on shared variables, and all the O(n) stages (A,B,D) will be O(n/k) stages. The stage C will be split into two stages C1 and C2, which adds one barrier synchronization for each digit sorted on. As $v=2^{numbit}$ is the number of possible values in the digit we sort on, both C1 and C2 will be have O(v) operations. Also b[], the array we copy to and from, can for each thread be made of length n/k, and we start with a new stage A2, copying all elements owned by this thread to the shorter array b[] and later copying all b[] back to a[]. Since we also have used the parallel version of findMax in stage A, that is O(n/k), the whole algorithm executes in time O(n/k). We have used no synchronizations of shared data in any of the stages, only between the stages, so the number of synchronizations are O(log n) since we in an actual implementation sort on more digits the larger n is (in practice 1 – 3 digits for 32 bit integers, 1-6 digits for 64 bit integers).

```
 void radixSort( int [] a, int [] b, int leftElem, int rightElem,
   int leftValue, int rightValue,
        int  maskLen, int shift, int threadIndex) {

        int  acumVal=0, j, k, next =0;
        int  mask = (1<<maskLen) -1;
        int [] localCount =new int [1<<maskLen];

        // A1) is perfomed in run, finding  maskLen
       //   and making a local b[0.. (rightElem – leftElem)]
        // A2) new stage, copy relevant part of a[] to b[]
        for (int i = leftElem; i <= rightElem; i++) {
              b[next++] = a[i];
            }
        }

       // B)count=the frequency of each radix value
       //      this thread owns
       for (int i = 0; i <b.length; i++) {
            localCount[(b[i]>> shift) & mask]++;
       }
       allCounts[threadIndex] = localCount;

       //SYNC 1. here --------------------------------
       try {  sync.await();    // stop-start the threads
            } catch (Exception ex) { return;}

      // C1)Add all values owned by this thread into
     // allCounts[0][leftValue..rightValue]and accumulate them

      acumVal =0;
      for ( k = leftValue; k <= rightValue; k++){
          for (int i = 0; i < numThreads; i++) {
               j = allCounts[i][k];
              allCounts[i][k] = acumVal;
```

```
                acumVal +=j;
            }
        }
        allAccumVals[threadIndex] = acumVal;

        //SYNC 2. here --------------------------------
        try { sync.await();    // stop-start the threads
        } catch (Exception ex) { return;}

        // C2) Accumulate own values
        acumVal =0;
        for (int i = 0; i < threadIndex; i++) {
                acumVal += allAccumVals[i];
        }

        // Update own accumulated values for those
        //  values owned by this thread
        for ( k = leftValue; k <= rightValue; k++){
             for (int i = 0; i <numThreads; i++) {
                 allCounts[i][k] += acumVal;
             }
        }

        //SYNC 3. here --------------------------------
        try { sync.await();    // stop-start the threads
        } catch (Exception ex) { return;}

        // D) move own numbers in sorted order a to b
        for (int i = 0; i < b.length; i++) {
           a[localCount[(b[i]>> shift) & mask]++]=  b[i];
        }

        //SYNC 4. here ----------------------------------------
        try {  sync.await();    // start the threads
        } catch (Exception ex) { return;}
}// end radixSort
```

Program 6. *The Duplicate shared variables parallel solution. This is fully parallel for all digits sorted. Stage C is now split in two stages. In C1 it updates the local copies of count for all threads for its own values. Then in C2 these values for all threads are added with the accumulated number of values smaller than its own values. Method RadixSort is also here basically the same as the sequential solution.*

Note that we have in this solution duplicated all shared variables (a[] and count[]) and have distribute elements to threads both by element index and by value. We have also duplicated some instructions. Since this is theoretically an O(n/k) parallel algorithm, we might hope for a linear speedup with the number of processing cores we have. However, previously experiments with a somewhat similar Left Radix algorithm [10] showed that the bandwidth of the memory channel and the caches can make the speedup less spectacular.

VII. Test results for the parallel Right Radix algorithms

We ask the students to evaluate their solutions to their Right Radix programs by running their programs on different multi core computers. Here we show results when the algorithms are implemented in Java 1.6 on a Linux, HP based server with 32 cores, and in Java 1.7 on a Windows 7 based laptop with 4 cores.

The n integer elements to be sorted are drawn from a uniform 0:n-1 distribution. Many other distributions could of course have been chosen, but this is reported to be the most frequently used [21]. Up to 104 million elements are sorted. Based on the maximum number, two arrays of length maximum/2 are sorted, then four arrays of length maximum/4, etc., until 100 arrays is reached, to get more stable performance values when sorting shorter arrays. Time is measured with the Java system call: System.nanoTime().

The overhead for creating the pool of 4 threads for the laptop is measured to 1 milliseconds and 14 milliseconds for the 64 threads on the server (this overhead is not included in the measurement values), while the start/stop barrier synchronization is measured to 0.1 milliseconds on the Laptop and 10 milliseconds on the Server (included in total running times). These large differences in overhead on Barrier synchronization might be attributed to a new and better implementation of the 'java.util.concurrent' library in java 1.7 as opposed to java 1.6. The number of threads in each case is equal to the number of cores, as mentioned in the caption.

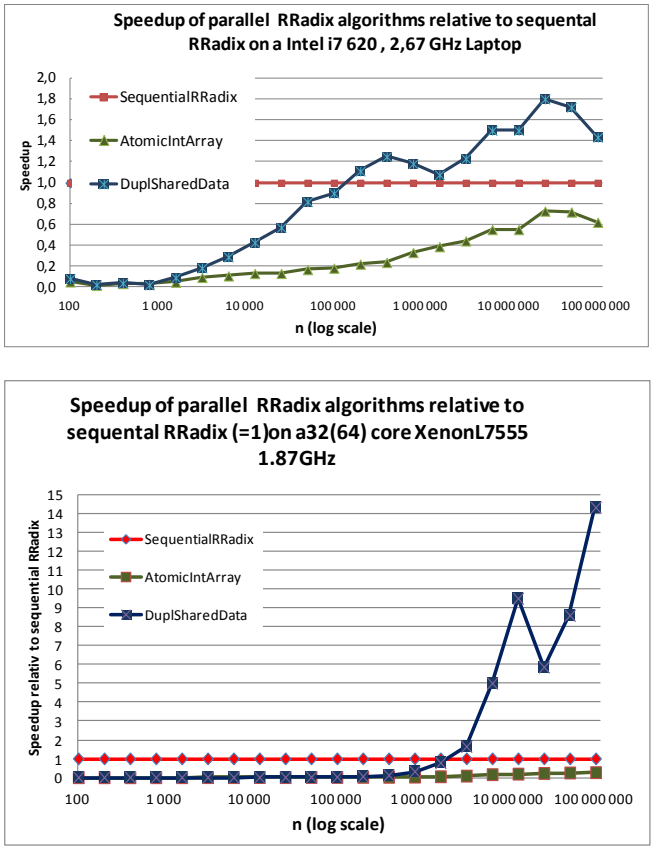

Figure 2. *Test results for a Windows based 4 core laptop (Java 1.7) top, and a 32(64 hyperthreaded) Linux based server(Java 1.6) bottom, measuring the speedup of the 2 parallel RRadix algorithms compared to the sequential algorithm.*

We see that of the tested paradigms, only the duplication of shared data really made a faster algorithm, The paradigm that uses synchronized access to all shared variables, both in the FindMax and the RRadix case, was not effective.

VIII. WHAT TO TEACH THE STUDENTS

From the large set of examples that the students have to solve, including the two examples shown in this paper, we can conclude that the students should have this knowledge at the end of the course:

a. Many parallel algorithms are transformations of effective and correct sequential algorithms.
b. There might be many correct parallel solutions to a given problem − only those that have a speedup > 1 should be considered.
c. The number of synchronizations on shared data in parallel algorithms must *not* be of the same order as the algorithm. It must be either a constant, like the number of parts of the algorithm, or less than the order of the algorithm, say O(log n) if the algorithm is O(n).
d. Synchronized methods should be used with extreme care in a parallel program.
e. Solutions where shared variables are duplicated to each thread and written back after synchronization should always be considered.

IX. CONCLUSIONS

This paper describes an undergraduate course that takes a practical approach to parallel programming by teaching the students to parallelize algorithms and measure their execution times. The main point is to get a speedup > 1, and this is usually achieved by careful treatment of shared data. In most examples, also examples not mentioned in this paper, the best solution is usually to copy these variables to local unshared variables in each thread. Throughout this course we demonstrate many problems encountered in parallel and concurrent programming and, if possible, try to give the students 'foolproof' methods for dealing with them.

In addition, these investigations have produced PRRadix, an effective parallel version of the Right Radix (LSD radix) algorithm. The full Java code of PRRadix is downloadable from [17]. Previously, effective parallel versions of the Left Radix (MSD radix) algorithm have been reported [9, 10].

X. REFERENCES

[1] Horacio González-Vélez and Mario Leyton A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Software: Practice and Experience Volume 40, Issue 12, pages 1135-1160, November/December 2010.

[2] Uzi Vishkin Using simple abstraction to reinvent computing for parallelism, Communications of the ACM Volume 54 Issue 1, January 2011, pages 75-85, NY, USA, doi>10.1145/1866739.1866757

[3] http://en.wikipedia.org/wiki/Amdahl%27s_law

[4] I. Foster. Desining and Building Parallel Programs, Addison Wesley, 1996, available at http://www.mcs.anl.gov/dbpp

[5] D. Caromel and M. Leyton. Fine tuning algorithmic skeletons, In 13th International Euro-Par Conference: Parallel Processing, volume 4641 of Lecture Notes in Computer Science, pages 72–81. Springer-Verlag, 2007.

[6] D. Lea. Concurrent Programming in Java: Design Principles and Patterns, Addison-Wesley Longman Publishing Co., Inc.,Boston, MA, USA, 1996.

[7] Justin E. Gottschlich et al. Shifting the Parallel Programming Paradigm, ISAAC 2009, Lecture Notes in Computer Science, Vol. 5878, Springer Verlag.

[8] Timoty G. Mattson, Beverly A. Sanders, Berna L. Massingill Patterns for Parallel Programming, Addison-Wesley, Boston, MA, 2005

[9] Shin-Jae Lee, Minsoo Jeon, Dongseung Kim, Andrew Sohn, Partitioned Parallel Radix Sort, Journal of Parallel and Distributed Computing, Volume 62, Issue 4, April 2002, Pages 656-668, ISSN 0743-7315, DOI: 10.1006/jpdc.2001.1808.

[10] Arne Maus. A full parallel radix sorting algorithm for multicore processors, Norwegian Informatics Conf, Tromsø, Norway, 2011 (ISBN 82-91116-45-8)

[11] J. JaJa, Introduction to Parallel Algorithms, Addison–Wesley, Reading, MA, 1992.

[12] Donald Knuth,The Art of Computer Programming, Volume 3: Sorting and Searching, Second Edition. Addison-Wesley, 1998.

[13] Arne Maus and Stein Gjessing: A Model for the Effect of Caching on Algorithmic Efficiency in Radix based Sorting, The Second International Conference on Software Engineering Advances, ICSEA 25.Aug. 2007, France

[14] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. 11th ACM Symposium of Discrete Algorithms, pages 829–838, 2000.

[15] www.intel.com/go/terascale

[16] (Tile-GX 100 core) http://www.Tilera.com

[17] Arne Maus' sorting homepage: http://www.heim.ifi.uio.no/~arnem/sorting/

[18] Message Passing Interface 4.0: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf , 2013

[19] Mark Allen Weiss Datastructures & Algorithm analysis in Java, Addison Wesley, Reading Mass., 1999

[20] http://openmp.org/wp/

[21] Private communication with Donald Knuth

[22] J.Arndt and C. Haenel. π unleached. SpringerVerlag, Berlin, 2001