DD2443 H23

Group 7: Lukas Sebastian Beinlich, Aleksei Veresov

September 2023

Compiling and Executing

All tasks can be compiled and executed using the following commands:

```
cd src/
javac *.java
java MeasureMain <SorterName> <Threads> <Array Size> <Warm-up rounds>
    <Measurement rounds> <PRNG seed>
```

The particular sorter name can be seen in the corresponding task section of the report. The data for plots were collected on PDC Dardel using the script `src/job.sh`. The resources were allocated via `salloc -n 1 -t 0:30:00 -A edu23.dd2443 -p shared`, since the main partition was overloaded. This means that the gathered data is a bit unreliable (i.e., it depends on the state of other tasks executed on Dardel, while in main partition tasks have exclusive access to a node), and to counter this, three measurements were done with a gap of several hours between them.

1 Sequential Sort

We chose **quicksort** for implementation in all tasks.

The sorting consists of two parts. First, we split the array into two parts: we chose a pivot point (particularly in the implementation below, the last element is used as the pivot), then we place all elements of the array less than the pivot before it, and we place all elements greater than the pivot after it. Second, we sort both of these subarrays.

The `split` function can be found in `src/Auxiliary.java`, and the sorting function itself can be found in `src/SequentialSort.java` and is:

```java
public void sort(int[] arr, int begin, int end)
{
    if (end <= begin) // no elements to sort
        return;

    int split_position = Auxiliary.split(arr, begin, end);
    sort(arr, begin, split_position - 1);
    sort(arr, split_position + 1, end);
}
```

It is important to have base case for our recursion (the if with return), and since empty arrays or arrays of one element are already sorted, we can safely return in these cases.

2 Amdahl's Law

To estimate Amdahl's law, we use a few assumptions:

- runtime of quicksort is on average $n \cdot log(n)$
- the pivot chosen divides the array to sort in two parts of the same length

We first take a look at the usual run of the sorting for two threads. Here we start by partitioning the array first into a higher and lower region. This means that every value is touched once (and around half of them are swapped) and we have a runtime scaling with n, where n is the amount of values to be sorted. This work is sequential.

After the partitioning is done, the lower and upper half can be sorted independently. Sorting both partitions using quicksort has an average runtime of $2 \cdot \frac{n}{2} \cdot log(\frac{n}{2})$. This can be done in parallel on two threads, so for two threads we have the following:

$$s_2 = \frac{n}{n + 2 \cdot \frac{n}{2} \cdot log(\frac{n}{2})} = \frac{1}{1 + log(\frac{n}{2})}$$
$$p_2 = \frac{2 \cdot \frac{n}{2} \cdot log(\frac{n}{2})}{n + 2 \cdot \frac{n}{2} \cdot log(\frac{n}{2})} = \frac{log(\frac{n}{2})}{1 + log(\frac{n}{2})}$$
$$S(2) = \frac{1}{s_2 + \frac{p_2}{2}} = \frac{1}{\frac{1}{1+log(\frac{n}{2})} + \frac{1}{2} \cdot \frac{log(\frac{n}{2})}{1+log(\frac{n}{2})}} = \frac{2 + 2log(\frac{n}{2})}{2 + log(\frac{n}{2})}$$

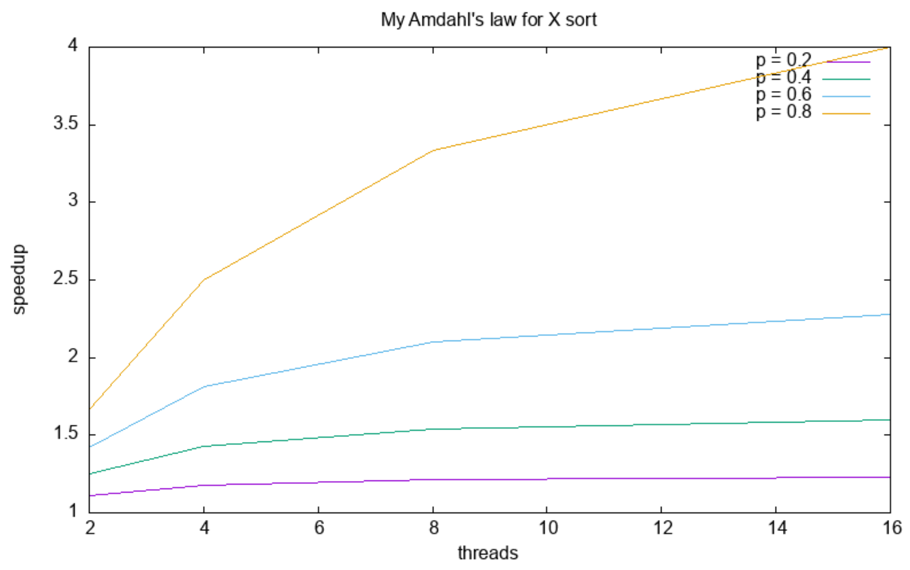
For a higher number of threads, we only reach true parallelism after enough partitions have been done, so we have to iterate multiple times over the entire array, but the second iteration is twice as fast, as the lower and upper half can be done independently. This means we have a runtime scaling of $1.5n$ for 4 threads, $1.75n$ for 8 threads and $1.875n$ for 16 threads. After this we have 4/8/16 partitions of equal size that have to be sorted using quicksort in parallel. Using the same calculation as above we get the following

$$s_4 = \frac{1.5n}{1.5n + 4 \cdot \frac{n}{4} \cdot log(\frac{n}{4})} = \frac{1}{1.5 + log(\frac{n}{4})}$$
$$p_4 = \frac{4 \cdot \frac{n}{4} \cdot log(\frac{n}{4})}{1.5n + 4 \cdot \frac{n}{4} \cdot log(\frac{n}{4})} = \frac{log(\frac{n}{4})}{1.5 + log(\frac{n}{4})}$$
$$S(4) = \frac{1}{\frac{1}{1.5+log(\frac{n}{4})} + \frac{1}{4} \cdot \frac{log(\frac{n}{4})}{1.5+log(\frac{n}{4})}} = \frac{6 + 4log(\frac{n}{4})}{4 + log(\frac{n}{4})}$$
$$S(8) = \frac{14 + 8log(\frac{n}{8})}{8 + log(\frac{n}{8})}$$
$$S(16) = \frac{30 + 16log(\frac{n}{16})}{16 + log(\frac{n}{16})}$$

This means we have a speedup as seen in the table below:

n	S(2)	S(4)	S(8)	S(16)
100	1.459	2.6147	3.2497	3.3266
1,000	1.574	2.9497	4.1024	4.7759
10,000	1.649	3.1543	4.6955	5.9277
100,000	1.701	3.2921	5.1320	6.8651
1,000,000	1.7402	3.3913	5.4666	7.6429
10,000,000	1.7701	3.4661	5.7313	8.2987

In total, we find Amdahl's law applicable to our case, so we don't propose any changes to it. This is the graph of it:



3 Thread start() and join()

The code for this task can be found in `src/ThreadSort.java`.

To use thread start and join we modified the sort function a bit, so we have the following

```java
private void sort(int[] arr, int begin, int end)
{
    if (end <= begin) // no elements to sort
        return;

    int split_position = Auxiliary.split(arr, begin, end);
    if (available_threads > 1)
    {
        // Do balancing dependant on the size of subarrays:
        // Ceil is used to ensure that at least one new thread will be
            spawned.
        int giveaway_threads = (int) Math
                .ceil((available_threads - 1.0) * (split_position -
                    begin) / (end - begin));
        // Spawn new sorting thread with giveaway_threads available and
             the task to sort lower subarray:
        Thread t = new Thread(new Runnable()
        {
            public void run()
            {
                new ThreadSort(giveaway_threads).sort(arr, begin,
                     split_position - 1);
            }
        });
        t.start();
        // Now we update available threads and recurse:
        available_threads -= giveaway_threads;
        sort(arr, split_position + 1, end);
        try
        { // Each thread waits for its subthread (and, recursively,
             sub...subthreads) to finish:
            t.join();
        }
        catch (Exception e)
        {
            System.out.println("Exception: " + e);
        }
    }
    else
    {
        sort(arr, begin, split_position - 1);
        sort(arr, split_position + 1, end);
    }
}
```

This code basically checks if more threads are available to fork and if yes then it creates a new thread to sort the lower half of the partition while itself sorts the higher part. Afterward, it waits for the other half to finish. If no more threads are available to spawn, both halves are done in the current thread.

Important to note though is that when the partition is not balanced, one half will be done much faster than the other, but the available threads will not be reused in the other half. This limits performance improvements as described in task 2 only on specifically crafted arrays (where the assumptions hold true).

4 ExecutorService

The code for this task can be found in `src/ExecutorServiceSort.java` and `src/ExecutorServicePhaserSort.java`, since we developed two different ways to use ExecutorService here.

First, `src/ExecutorServiceSort.java`. In this case the idea is to use AtomicInteger to count number of placed pivot points. The sign of completion of execution is then this AtomicInteger becomes equal to array length (i.e., all elements are in their place). To wait for the count active waiting was used, i.e., while loop with check of this equality. This approach is quite inefficient in terms of resource usage, but it was chosen for its simplicity. The main thread is thus not counted as a thread actually doing sorting.

The important part here is to increase placed pivots count in case we have an array with one element, so the sorting algorithm was modified with additional if before if-return for base cases:

```
if (end == begin)
    pivots.incrementAndGet();
if (end <= begin)
    return;
...
```

Another important part is that in order to achieve suitable performance, we use sequential sort if the size of subarray is lower than certain threshold (lines 57-79 of `src/ExecutorServiceSort.java`).

The other implementation (`src/ExecutorServicePhaserSort.java`) is using Phaser, which counts the number of currently running jobs, and acts as a barrier for the main thread to wait until all submitted threads finish as follows

```
public void sort(int[] arr)
{
    final Phaser phaser = new Phaser(1); // register self
    executor = Executors.newFixedThreadPool(threads);
    phaser.register();
    Runnable task = () -> {
        this.sort(arr, 0, arr.length - 1, phaser);
        phaser.arriveAndDeregister();
    };
    executor.execute(task);
    try
    {
        phaser.arriveAndAwaitAdvance();
        executor.shutdown(); // wait for all tasks to end and close
        executor.awaitTermination(10, TimeUnit.SECONDS); // this should
            never wait
    }
    catch (Exception e)
    {}
}
private void sort(int[] arr, int begin, int end, Phaser phaser)
{
    if (end - begin <= 0) // only one (or zero) elements, return
        return;
    int split = Auxiliary.split(arr, begin, end);

    boolean manual_low = false;
    if ((split - 1) - begin > threshold)
    {
        //sorting without new threads
        final Phaser phaser2 = phaser.getRegisteredParties() > 60000 ?
            new Phaser(phaser) : phaser; // check if we have to many
            registered things in the phaser, if yes create a new one
        phaser2.register();
        Runnable task1 = () -> {
            this.sort(arr, begin, split - 1, phaser2);
            phaser2.arriveAndDeregister();
        };
        executor.execute(task1);
    }
    else
        manual_low = true;
    boolean manual_high = false;
    if (end - (split + 1) > threshold)
    {
        //sorting without new threads
        final Phaser phaser2 = phaser.getRegisteredParties() > 60000 ?
             new Phaser(phaser) : phaser; // check if we have to many
             registered things in the phaser, if yes create a new one
        phaser2.register();
        Runnable task2 = () -> {
            this.sort(arr, split + 1, end, phaser2);
            phaser2.arriveAndDeregister();
        };
        executor.execute(task2);
    }
    else
        manual_high = true;
    if (manual_low)
        this.sort(arr, begin, split - 1, phaser);
    if (manual_high)
        this.sort(arr, split + 1, end, phaser);
}
```

Here we can see in the initial call the ExecutorService and Phaser are intialized, while a thread is defined as calling sort, which calls sorts and then deregisters itself on the phaser. Note that before submitting this job, it is important that we already register this job, as it takes a bit for the thread to start and afterward the arriveAndAwaitAdvance() call would just immediately advance, because the ExecutorService has not yet started the job.

The sort function just partitions the array as usual and then submits two new tasks to the ExecutorService. However, the Phaser from java can only register a maximum of 65535 concurrently running threads, which is why we first check if the array to sort is at least of minimum size threshold, otherwise it is better to just quickly sort it on the same thread (this is also because creating and running new threads comes with a bit of an overhead, which leads to terrible performance otherwise). Secondly, we simply create a new (child) phaser, which the parent also has to wait for, if we have reached a certain number of registered jobs on the parent. Note however, that this implementation currently can still register over 65535 threads on the same phaser, as the check how many jobs are registered and the registration of a new one is not an atomic action and due to unfortunate scheduling a lot of jobs can check at the same time before the threshold, but when reaching the phaser.register() line, there might be significantly more than 60000 threads registered. Decreasing this limit could help, as well as increasing the threshold. This is also the reason why we implemented the other solution above after encountering this problem (also the phaser solution is much more efficient).

5 ForkJoinPool and RecursiveAction

The code for this task can be found in `src/ForkJoinPoolSort.java`.

The idea here is to simply invoke a worker which will do a separation and invoke two workers for subarrays recursively:

```
if (end <= begin)
    return;
int split_position = Auxiliary.split(arr, begin, end);
invokeAll(new Worker(arr, begin, split_position - 1),
          new Worker(arr, split_position + 1, end));
```

It is important to note that invoke/invokeAll method is blocking until the task/-tasks given to it has/have finished, so the sort will return only then all created tasks are completed. This is a very simple implementation and very well suited for quicksort, the performance is very good, especially considering there is no switch to sequential sorting after a threshold.

6 ParallelStream and Lambda Functions

The code for this task can be found in `src/ParallelStreamSort.java`.

This solution has a bit of limited parallelism compared to other solutions, i.e., if we see quicksort as a tree of sorts (the first one at top, then its two subsorts, then their 4 subsorts and so on), this solution is allowing parallel execution of sorts of the same depth only. This doesn't result in too much damage to performance, though.

The idea is to organize sorting as a sequence of rounds, in which round all subtasks can execute in parallel. To do it we use `todo` variable. It is important for it to be a concurrent collection, ConcurrentLinkedQueue was chosen, however, other concurrent collections will also work.

The tasks are represented as pairs (two-element arrays), the first element is `begin`, and the second is `end`. Then, the task used to sort the whole array is represented as a task to sort array from 0 to array length - 1:

```
todo.add(new int[] { 0, arr.length - 1 });
```

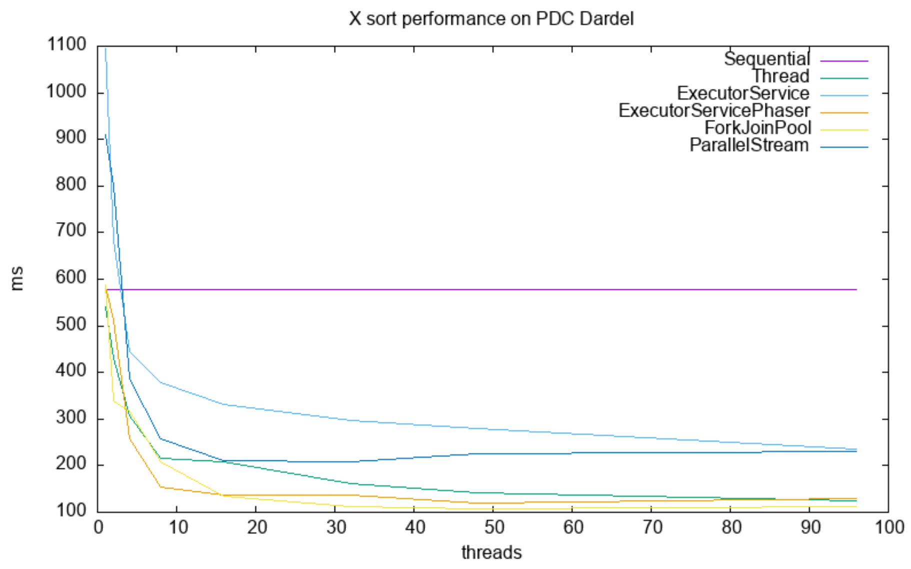
Then, ForkJoinPool is used to limit the amount of created tasks. Inside it, we spin in a loop until the next round of sorting has no more tasks:

```
while (todo.size() > 0)
{
    Collection<int[]> old_todo = todo;
    todo = new ConcurrentLinkedQueue<int[]>();

    old_todo.parallelStream().forEach((int[] range) -> {
        int begin = range[0];
        int end = range[1];

        if (end <= begin)
            return;
        if (end - begin < threshold)
        {
            new SequentialSort().sort(arr, begin, end);
            return;
        }

        int split_position = Auxiliary.split(arr, begin, end);

        todo.add(new int[] { begin, split_position - 1 });
        todo.add(new int[] { split_position + 1, end });
    });
}
```

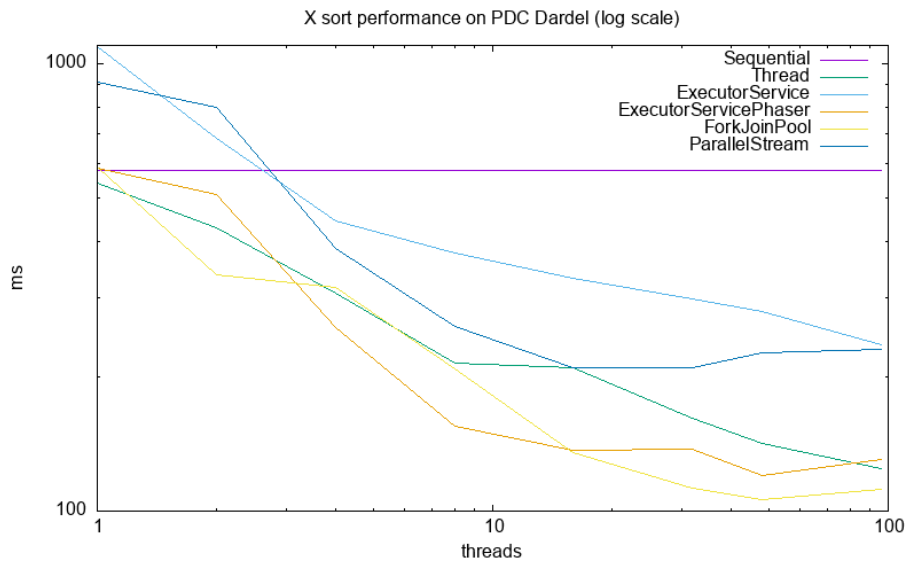
The tasks are allowed to be executed in parallel, since our concurrent collection is converted to parallel stream. Each task is just doing the split and adding two new tasks into the next round's `todo` if the size of the task's subarray is big enough, or does sequential sort otherwise.

7 Performance comparisons

As was stated, we had to use shared partition of Dardel, which means that some measures were affected by other tasks running on Dardel, and thus some measurements had too high deviation to be trusted. To counter this, we ran measurments three times with a time gap between them. Then the values with the lowest standard deviation were used for the final plot.

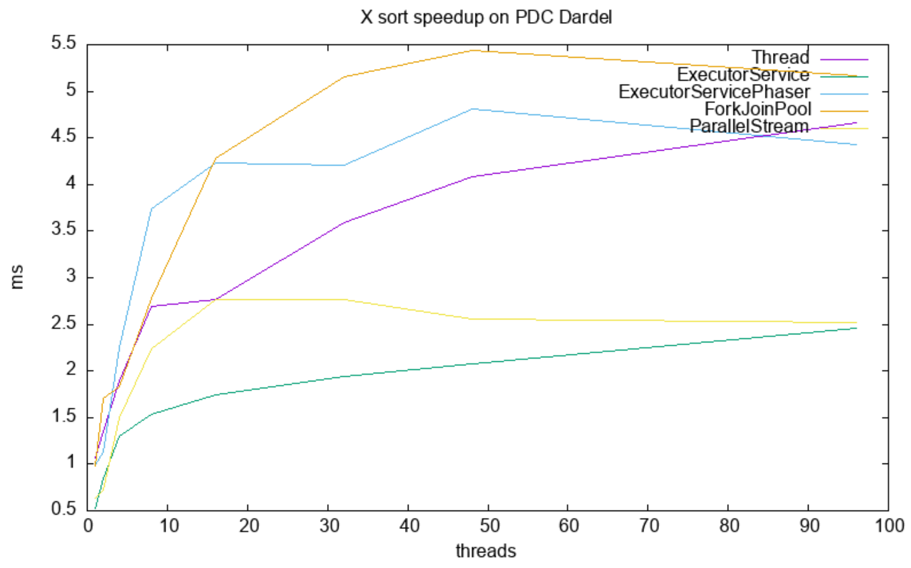


For a more clear picture of what is happening around a small and big numbers of threads, there is also a log-plot of the same data:



The raw data can be found in `data/res`, and the purified data for plot is in `data/pdc.dat`. The execution were done according to `src/job.sh`, so each sort were supplied with the same seed (37) and amount of elements in an array (5,000,000). We chose 30 as both warm up and measuring rounds number.

This is a graph of speedups:



8 Reflection

The easiest method to implement were (at first) the ForkJoinPool (our favorite, because of the simplicity and how well suited it is to this problem) and the ExecutorService, as they are very similar. However, it quickly became apparent, that the ExecutorService has some caveats, if you want to measure how long it takes, as, at least in our implementation, there is no (native) way to wait until all threads are done, while new threads are still able to be submitted. To solve this problem took us a long time and is also why we have two solutions, as the Phaser-solution was efficient, but can crash if parameters are chosen wrongly. In the start() and join() solution, it was a bit tricky to count the number of running threads, and the perfomance is also not optimal, as a partition that is significantly smaller than another may finish quickly, but the threads are then idle and not used by the other partitions.

On the plots we can see, that all sorts have more or less the same shape, except ParallelStream, which peaks at 32 threads. This is probably due to its more synchronized nature, so the peak is achieved earlier than with others methods. The peak happens than redundant efforts for synchronization of threads become bigger than payoff from them.

The fastest methods are ExecutorService with Phaser and ForkJoinPool, although the first one has threshold, while the latter does not, so ForkJoinPool in the end is the fastest. Thread method is also quite fast, although it is still slower than the two others. ExecutorService with AtomicInteger is the slowest method, which is expected due to synchronization efforts happening in the AtomicInteger implementation. However, it still shows a stable increase in performance with growth of threads count.

So in total, ForkJoinPool with RecursiveAction seems to be both fastest and easiest to implement, so it is our favorite method at the moment.