UNIVERSITY OF OSLO
Department of Informatics

Investigating Different Concurrency Mechanisms in Java

Master thesis

Petter Andersen
Busterud

August 2012

Abstract

With the continual growth in number of cores on the Central Processing Unit (CPU), developers will need to focus more and more on concurrent programming to get the desired performance boost that in the past have come logically with the increased clock-rate.

Today there are numerous of different libraries and mechanisms for synchronization and parallelization in Java, and in this thesis we will attempt to test the efficiency and run time of two different types of sorting algorithms on machines with multi-core processors using the concurrent tools provided by Java.

We will also be looking into the overhead that occur by using the various mechanisms, to help establish the correlations between the mechanisms overhead and performance gain.

Acknowledgement

This thesis concludes my Master's Degree in *Informatics: Programming and Networks* with the Department of Informatics at the University of Oslo. It has been a long and different, but positive experience.

First I want to thank my supervisor Arne Maus, who offered me this thesis and have been providing valuable feedback along the way.

I also like to thank my friends and family who has been patient with me all these years, and the support and encouragement they given me throughout this period.

University of Oslo, August 2012
Petter Andersen Busterud

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Background

The Central Processing Unit (CPU) is one of the main components we have in computer systems, and may be looked at as the *brain* in our computer. It is the CPU which performs the necessary processing and calculations of instructions for the computer.

Moore's law is today one of the most known "laws" in computer science. It all started in the mid-1960s when Gordon E. Moore released an article in the *Electronics* magazine[1], where he described a long going trend where the number of transistors that could be inexpensively placed on an integrated circuit doubled roughly every two years. He foresaw that this would continue for at least ten years, which by then would mean around mid-1970s. But amazingly enough his statement continued to be true 40 years later, and it is still realistic to think that it will keep doing so.

Looking back at the years that has gone we see that the CPU originally consisted of only one single processing core, where the performance throughout the years have had an exponential growth because of a combination of Moore's law and the increasing of the clock frequency speed (i.e. more instructions can execute per second). But in the later years we have seen another trend rise, the multi-core CPU. This trend is due to the limitation on of how high the clock frequency could go before having a **huge** impact on energy consumption and heat production compared to the performance gain. And that is why the manufactures started producing CPUs with multi-core architecture, thus continue to increase the overall system performance. So while a single core CPU only could execute a

single sequence of instructions, multi-core could execute many sequences. "Many" of course being how many cores the CPU has; dual- (2), quad- (4) or octa-core (8) can today be seen on every modern laptop and desktop computers. And on the server side we see all possible combination ranging from 16- to 128-cores (and more), we will sure be seeing even more in the future.

While the first dual-core processor hit the desktop market in 2005, Intel had already used another idea of increasing the CPU performance some years earlier. The introduction of **Hyper-Threading** was brought to the market appearing on the *Pentium 4* processor in 2002. In short the Hyper-Threading is to physically duplicate certain sections of the processor (i.e. architectural state), but not the main execution resources. Resulting in giving a processor core the ability to schedule and assign resources to two threads at once, but only compute one at any given time, and with this theoretically increasing the performance. Intel claimed that they would get a performance boost around 15 to 30% [2] compared to other non-Hyper-Threaded CPUs and only increase the size of the *die* with 5%.

But with the introduction of multi-core the developers have had to change their mindset when writing program code. Not only would they have to divide the program in such a way that each part could run concurrently, they also had to think about synchronization when more than one core have access to shared data. Taking advantage of multi-core to get the desired performance increase, these concurrent parts will have to run in parallel and all necessary sequential fraction of the program needs to be kept to a minimum (Amdahl's law).

Java, a programming language developed by James Gosling at Sun Microsystems and released in 1995, have since the beginning offered multi-threading and synchronized methods. In the earliest versions, Java only had a few low-level primitives that dealt with monitor and synchronization of threads. In 2004 almost a decade later, version 5 of Java was released introducing concurrency utilities. These utilities were designed to be used as building blocks for concurrent classes or applications, offering a number of advantages for the developer. These concurrency utilities have since then received many improvements and additions in new version releases, with the latest being Java 7 released July, 2011.

1.2 Motivation

The motivation for this thesis is to look into already existing sequential sorting algorithms for Java, and explore the potential performance gain by parallelizing these using the concurrency packages with its classes and interfaces included in Java. Thus trying to take advantage of the multi-core architecture and letting us test the efficiency and run time of these implementations.

Creating implementations using various tools will also give us the ability to compare the different results. Maybe some tools are more efficient at certain tasks then others, how easy is it to create these implementations and what precautions should one think about when writing concurrent programs. These are some of the questions we would like to get answered by working with the subject of this thesis.

1.3 Thesis Outline

Chapter 2 - *Concurrency and Threads in Java* provides background information on concurrency in general. Here we will be giving an introduction to commonly used terms, look at the collection of different classes and interfaces to help dealing with concurrency and the usage of threads; the main component for concurrent programming in Java.

Chapter 3 - *Test Environment* gives an overview of how the data used for testing are structured and generated. What the thoughts behind benchmarking the different implementations are, and what the specification of the hardware and software that is used for producing and testing for this thesis. There are also a section about the Java Virtual Machine and what necessary tuning that had to be done.

Chapter 4 - *Sorting Algorithms* goes through the choices made for picking what kind of algorithms we wanted to work with for this thesis. Why chose sorting algorithms and what types of these to experiment with. The chapter also gives an insight in how these algorithms work and how to construct a regular sequential implementation of the different algorithms.

Chapter 5 - *Experiments* will attempt on implementing different parallel versions of the algorithms in Chapter 4, using the built-in concurrency packages in Java. These implementations will be measured and compared against a sequential algorithm to see how well they perform against each other.

Chapter 6 - *Threads and Overhead* looks into the overhead imposed by using the various tools / mechanisms in Java. First there is an introduction to how the measurements were done on the two operating systems of choice; Windows and Linux. Then there is an overview of the test results with the different mechanisms, and finally a comparison of the results.

Chapter 7 - *Discussion* is a more in depth talk on the experiments and overhead chapters, bringing some theoretical viewpoints.

Chapter 8 - *Conclusions* summarizes the results from the previous chapters. And will go through some general advices when construction parallel versions of already existing sequential algorithms. Finally give some suggestions on further research on the topic; things that we would like to have done, but were never able to set time for.

Chapter 2

Concurrency and Threads in Java

The Java class `Thread` is the whole foundation for all Java concurrency frameworks. This class makes it easy for the developer to create and execute threads. Initially a thread is identical to normal Java classes, but with the exception of having a *runnable* task which can be executed concurrently with the main program. When these threads are called it is up to the Java Virtual Machine (JVM) and the operating system (OS) to decide which processor core will execute the thread.

Before version 5 of Java was introduced the only way to synchronize threads was through the low-level concurrency mechanisms; `synchronized`, `volatile`, `wait()`, `notify()` and `notifyAll()`. They are all difficult to use correctly and the potential for common concurrent threats like deadlock and starvation is high. Java 5 (originally known as Java 1.5) changed this by including a new package named `java.util.concurrent` (including two sub-packages `atomic` and `locks`), which allows a more high-level synchronization on threads. These packages have been updated and improved by including new classes and interfaces in Java 6 and the current version 7, giving the developer more tools to use with concurrent programming.

While many of the subjects are covered in this thesis there are still a lot more to go around, and if one want to get a more deeper knowledge and insight in the basic concepts of concurrency, learn techniques for building and composing thread-safe classes, how to use the concurrency building blocks in `java.util.concurrent` and recommendation on performance optimization. There are many books and articles covering these subjects; one

of the best-selling and personally recommend book is *Java Concurrency in Practice* by Brian Goetz[3], which also includes Doug Lea as one of the authors; the man behind *JSR 166: Concurrency Utilities*[4] the concurrency APIs for Java.

2.1 Concurrency Concepts

This section will present a brief introduction to the different terms used for concurrency and Java in general.

A program can be called **concurrent** if it is divided in such a way that two or more threads can progress at *some* time. This does not mean that they have to progress simultaneously, but that they can be swapped in and out by the operating system on a single core[5].

Parallel on the other hand is when we have two or more threads that progress at the same time, of course requiring multiple cores and where each thread are assigned to a separate core.

Computation that is performed in a thread is called a **task**. This task can also be a smaller part of a bigger problem (i.e. *divide-and-conquer*) that can be executed in parallel and combined with other tasks to create the complete solution.

A **thread** can only run one task at any given time. But two or more threads can run two or more tasks in parallel to speed up the computation of a problem.

Granularity in the terms of concurrency means the number of tasks the computation for a problem is divided into. There are two sub-terms used for granularity; Fine-grained and Coarse-grained. The finer the granularity, the smaller each individual task is and coarse being the opposite.

Atomicity or **atomic operation** is when an action executes one single instruction even though it may be a set of operations executing, an example is incrementing a number where it first will *Get* the number, then *Increment* it and finally *Store* the new value. The only possible outcome for an atomic operation is success or failure.

Context Switching is performed when the operative system switches out a running thread in favor for another thread that is allowed to execute instead. This is done by a scheduler whenever a single CPU runs more than one thread, to get a concurrent behavior. Context switches causes big

performance penalties and the operative system therefore generally tries to avoid them.

2.2 Concurrency in Java

As an introduction to concurrency in Java, we will in this part of the thesis go through some of the packages currently included in Java SE 7[6] which were released July, 2011. The new version includes some additions to the concurrent package; some of these are the interface `TransferQueue` and the classes `ForkJoin`, `ThreadLocalRandom` and `Phaser`. The following lists show these packages and its classes:

`java.util.concurrent`

- AbstractExecutorService
- ArrayBlockingQueue
- ConcurrentHashMap
- ConcurrentLinkedDeque
- ConcurrentLinkedQueue
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- CountDownLatch
- CyclicBarrier
- DelayQueue
- Exchanger
- ExecutorCompletionService
- Executors
- ForkJoinPool
- ForkJoinTask
- ForkJoinWorkerThread
- FutureTask
- LinkedBlockingDeque
- LinkedBlockingQueue
- LinkedTransferQueue
- Phaser
- PriorityBlockingQueue
- RecursiveAction
- RecursiveTask
- ScheduledThreadPoolExecutor
- Semaphore
- SynchronousQueue
- ThreadLocalRandom
- ThreadPoolExecutor

`java.util.concurrent.atomic`

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicIntegerFieldUpdater
- AtomicLong
- AtomicLongArray
- AtomicLongFieldUpdater
- AtomicMarkableReference
- AtomicReference
- AtomicReferenceArray
- AtomicReferenceFieldUpdater
- AtomicStampedReference

`java.util.concurrent.locks`

- AbstractOwnableSynchronizer
- AbstractQueuedLongSynchronizer
- AbstractQueuedSynchronizer
- LockSupport
- ReentrantLock
- ReentrantReadWriteLock

These packages are commonly useful when dealing with concurrent programming along with the many classes and interfaces to help dealing with concurrency and parallelism. In the following subsections there will be given a brief introduction to the classes and interfaces used throughout this document, explaining their usage, what kind of problem they usually solve and give a few code examples.

While we wanted to go through every tool seen in these lists, testing them and get to know how they all work. It was however not possible for the time set of for this thesis. If one is interested to learn more about each individual tool, using the Java Platforms own API specification is a good start[6].

2.2.1 ThreadPool, a Task Execution Framework

When working with threads one could create a new thread for every task that is needed to be done and then proceed to tear them down after execution time. But not only would that create extra work for the developer, this would in most cases also result in poor performance (i.e. large overhead), especially when creating many threads for a lot of small tasks. This is where the **executor** framework in Java comes in to play. This framework executes tasks in separate threads, handling the creation and termination of threads and task so the developer does not have to it herself.

The `ThreadPoolExecutor` can be used to create an instance with the number of threads we wish to have as a parameter. With the two variables `corePoolSize` and `maximumPoolSize` we choose how many threads that should be available at any given time, and what the maximum number of threads allowed should be. By submitting tasks to the `ThreadPoolExecutor` much performance would be gained compared to creating and executing in a new thread.

To determine the number of threads one should use when creating a `ThreadPool`, the usually optimal solution for compute-intensive tasks is to use $N_{cpu} + 1$ [3]. This can easily be obtained with the function

`Runtime.getRuntime().availableProcessors()` which will give out the number of cores included Hyper-Threads. A lot of configuration can be done to the `ThreadPoolExecutor` along with rejection policies, thread factories and other configuration is explained well in the API documentation for Java[6].

2.2.2 Future

A `Future<>` instance is used as an asynchronous response of a computation. The asynchronous methods for executing threads of `ThreadPoolExector` and `ForkJoinPool` returns `Future<>` instances representing the result of a task handled by their worker thread(s). The blocking method `Future<>.get()` returns the generic result of the computation when finished. This means that a task can be started asynchronously and then some work can be performed and when the result of the task is needed the application can block until it is available.

2.2.3 Atomicity

A solution to keep variable data consistence is using the package **java.util.concurrent.atomic**. In this package there are a lot of different classes with the possibility to create thread-safe variables, for the data types like `Boolean` and `Int`; and `Arrays` of these types.

By creating these variables one would not have to worry about threads starting to write over each other. But by using values that updated atomically you would have to trade the simplicity of assigning variables with common operators (i.e. `+` / `-` / `*` ... etc), with using methods like `Get()` and `Set()` to update the values. In the Listing 2.1 there is a short demonstration of a *counter* class using `AtomicInteger` to store the value.

```
1  /** Short example with two Threads incrementing and decrementing
     on the same variable in a counter using the AtomicInteger */
3
   import java.util.concurrent.atomic.*;
5
   public class Atomic {
7    public static void main(String[] args) throws Exception {
       Counter counter = new Counter();
9      Thread t1 = new Thread(new Increment(counter));
       Thread t2 = new Thread(new Decrement(counter));
11     t1.start();   // Construct and start an increment Thread
       t2.start();   // Construct and start an decrement Thread
13     t1.join(); t2.join(); // When both threads are done
```

```
    System.out.println("Counter: " + counter.getCounter());
  }          // t1: inc 1000 times, t2: dec 600 times
}            //  Counter should end as 400

class Counter {
  private AtomicInteger anInt = new AtomicInteger();
  public void increment() { anInt.getAndIncrement(); }
  public void decrement() { anInt.getAndDecrement(); }
  public int getCounter() { return anInt.get(); }
}

class Increment implements Runnable {
  private Counter counter;
  public Increment(Counter count) { this.counter = count; }
  public void run() {
    for (int i = 0; i < 1000; i++) { this.counter.increment(); }
  }
}

class Decrement implements Runnable {
  private Counter counter;
  public Decrement(Counter count) { this.counter = count; }
  public void run() {
    for (int i = 0; i < 600; i++) { this.counter.decrement(); }
  }
}
```

Listing 2.1: AtomicInteger with two Threads

Even though there are no atomic classes for floating-point numbers one can still use the `AtomicInteger` and `AtomicLong` to store values for `float` and `double` by using `Float.floatToIntBits` and `Double.doubleToLongBits` conversion.

As of today one of the new classes that may appear in Java 8 is `AtomicDouble`[7], the JAR-file (Java ARchive) is available to download from the "Concurrency JSR-166 Interest Site"[4] and can be used with Java 7. The new version is expected to release summer 2013.

2.2.4 Locks

With concurrency we are almost always bound to get issues at one point when more threads are working on the same data. When threads are updating and retrieving values from the same object a situation can occur that could render the data's consistency, giving out wrong values to the user.

One solution to this problem could be to use locks. Locks can be used to make only one thread handling the object at any given time, and not let other threads handle the data while it is still being used. The library

java.util.concurrent.locks has methods helping out with this. This package contains the interface **Lock** which allows us to do just that.

There is also an extension for this interface that lets you have both a Read- and Write-lock, which is called **ReadWriteLock**. Since reading of the data still keeps the consistency, a Read-lock could be assigned to multiple threads accessing the same variable. When a thread would want to update the data, it will then ask for the write lock.

2.2.5 Queue and Deque

In some situations we could have threads doing different tasks then others. A usual example is the *Producer/Consumer scenario*. In the **java.util.concurrent** package we have the interface **BlockingQueue** which can help out. This interface let threads add and remove objects from a queue, which easily can be used in said scenario. As one or more threads contribute to the queue by adding objects, other threads could remove and handle them. When constructing and using a BlockingQueue it will always have a maximum number of objects the queue can contain. But the interface includes methods for waiting to input into the queue, it also let consumers wait for new objects if the queue gets empty. The example in Listing 2.2 shows how one could solve the *Producer/Consumer problem* by using a BlockingQueue.

```
1  class Producer implements Runnable {
     private final BlockingQueue queue;
3    Producer(BlockingQueue q) { queue = q; }
     public void run() {
5      // Producing
     }
7  }

9  class Consumer implements Runnable {
     private final BlockingQueue queue;
11   Consumer(BlockingQueue q) { queue = q; }
     public void run() {
13     // Consuming
     }
15 }

17 class ProducerConsumer {
     void main() {
19     BlockingQueue q = new SomeQueueImplementation();
       /** Creating and starting threads */
21   }
   }
```

Listing 2.2: Producer/Consumer example

There are also some extended classes based on the BlockingQueue. In the concurrent utility we have **LinkedBlockingQueue** that acts like a regular LinkedList and uses the first inn, first out approach (*FIFO*). We also have **PriorityBlockingQueue**, which will priorities the objects put into the queue. This will of course only let you put object that can be compared to each other, and object will be handed out by priority and not FIFO.

Other queue types includes **DelayQueue**, that let you delay an object before it is allowed to be handed out. We also have **SynchronousQueue** and **ArrayBlockingQueue**.

Contrary to a regular queue, where we are only able to handle one input and one output, a double-ended queue (*Deque*) lets us add and remove from both sides, making it a combination of a *FIFO-list*, and a last inn, first out (*LIFO-list*) with both a head and a tail. There are currently two types of deques in the concurrent package; they are called **BlockingDeque** and **LinkedBlockingDeque**.

2.2.6 Synchronize

With all the classes and methods in the concurrent utility, there is also some that can be used to synchronize two or more threads working simultaneously. This is to make sure we obtain correct runtime order.

The **CountDownLatch** is a class that let the developer create a gate with an integer as a parameter. This integer let the developer choose how many times the method *countDown* need to be called before opening the gate. When threads approach the gate they will have to wait for the countdown to reach zero before they can pass it.

CountDownLatch can only be used once. When it is created and the countdown has reached zero it will not lock the gate again and threads can pass through freely. So if we want to reset the countdown we would have to use another class called **CyclicBarrier**.

One of the classes included in Java 7 was the **Phaser**. The Phaser is an updated version of the previously mentioned CountDownLatch and CyclicBarrier. This updated version gives the developer a new functionality to change the value of the "countdown" dynamically while the program is running, giving the option to increase or decrease the number of needed calls to the Phaser.

2.2.7 Fork/Join

Another of the new features that was implemented in Java 7 is the **Fork/-Join**-framework, a framework that is similar to the `ThreadPoolExecutor`. It has its own kind of thread pool like `ThreadPoolExecutor`, so it takes advantage of reusing threads by submitting tasks.

Fork/Join is based on the idea of *Divide and Conquer*; a problem that is repeatedly spilt into smaller bits until some limit where the problem is small enough, then each sub-problem are solved and combined to form the full solution. The structure is illustrated in Listing 2.3 [8].

```
Result compute(Task task ) {
  if (problem is small) {
    Solve sequential
  }
  else {
    split into: Task left, Task right
    fork(compute(left), compute(right))
    join the results of subtasks
    compose results
  }
}
```

Listing 2.3: Divide and Conquer Structure

The basic idea for the Fork/Join-framework is the creation of new tasks every time an *Divide and Conquer* algorithm splits the data into two sub-problems. These tasks can then be solved in parallel by letting the framework divide them among threads. This framework is very light-weight as it is optimized to solve such problems where usually the only synchronization is when tasks wait for other sub-tasks to finish, resulting in a framework that scales very well.

Work-Stealing

The Fork/Join-framework also implements **Work-stealing** which also should give it good scalability and performance. The work-stealing is based on the producer/consumer design pattern with the use of deques.

The work-stealing scheduler operates as follows. Each worker thread has its own individual deque, a double-ended queue (see previous subsection). The queue works in a way that subtasks generated from running tasks on a worker are `pushed` into that works queue, and the worker `pops` tasks in LIFO order. If a worker has no tasks left in its own queue, it will pick

another workers queue randomly and try `taking` (”stealing”) a task in FIFO order[8].

Using LIFO for `popping` with its own thread and FIFO when another thread `takes` makes it an optimal solution for recursive divide-and-conquer algorithms. It takes advantage of that these algorithms generate much larger tasks early on and smaller tasks towards the end, resulting in less need for stealing work from each other.

Figure 2.1 shows a possible scenario which illustrates how the work-stealing could apply:

- **At time = 1**, Worker Thread-1 is idle and pops the last task that was put into its own queue (Task 3). Worker Thread-N is busy computing Task 4.
- **At time = 2**, Worker Thread-1 is busy computing Task 3. Worker Thread-N is idle and checks its own queue, but as it is empty he takes (*”Steals”*) the first task that was put into a randomly picked queue, in this case Task 1 from Thread-1’s queue.
- **At time = 3**, Worker Thread-1 is still busy computing Task 3. Worker Thread-N is busy computing Task 1 which in the process produces a new subtask (Task 5) that gets pushed into its own queue.

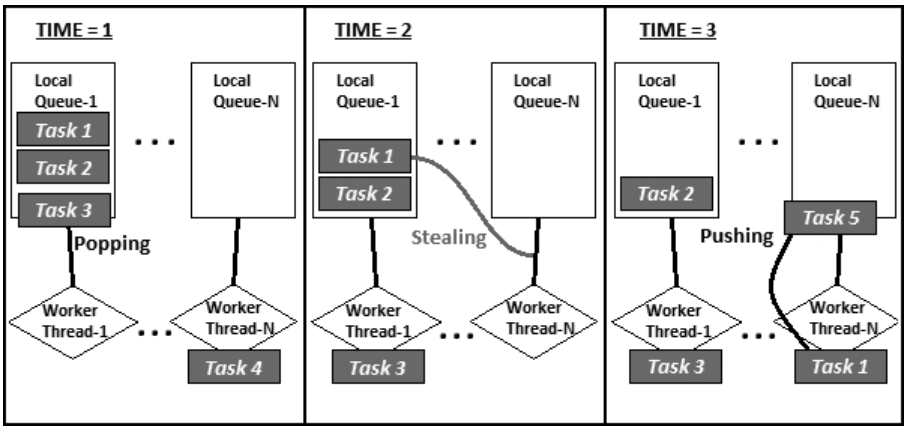

Figure 2.1: Work-Stealing

Chapter 3

Test Environment

The following chapter explains the test environment used for the next Chapters 4 and 5. We will here be going through how the test data is build up, how the measurement are done and the specifications of hardware and software used to produce the test results.

With this given information one should then be able to reproduce the collected data, or maybe even create some comparable results by using different hardware and/or software.

3.1 Test Data Structure

When benchmarking it is reasonable to do more than one test run on each dataset. This is mainly because other background processes on the computer may interrupt ongoing computation and may cause a relative huge impact on measurements, especially when sorting smaller dataset. When sorting 100 000 elements sequentially, which only takes a few milliseconds on a modern computer, a small delay in the computation could cause the measured time to be many times higher than expected.

To get a wide range of test results, the test data structure consists of many different sizes. The results shown in the different graphs in this thesis are generated by the structure that can be seen in Listing 3.1. This structure contains array-sizes ranging from 1000 to 100 million elements and also includes the number of times each array is sorted to get a more precise result. The array-sizes were defined using **Powers of Ten** incremented by $^1/_4$.

$$10^{n*{}^{1}/_{4}}, 12 \leq n \leq 32$$

With n ranging from 12 to 32 we get a good collection of array-sizes between 1000 and 100 million to test.

```
static int[][]  dataArray = new int[][] {
/*{Array-size, N runs} */
  {     1 000, 2 500}, {     1 778, 2 500}, {     3 162, 2 500},
  {     5 623, 2 500}, {    10 000, 2 500}, {    17 782, 2 500},
  {    31 622, 1 250}, {    56 234, 1 250}, {   100 000,   750},
  {   177 827,   500}, {   316 227,   250}, {   562 341,   150},
  { 1 000 000,   100}, { 1 778 279,    75}, { 3 162 277,    60},
  { 5 623 413,    50}, {10 000 000,    40}, { 17 782 794,    24},
  {31 622 776,    16}, {56 234 132,     8}, {100 000 000,     4},
};
```

Listing 3.1: Test Data Structure

For the smallest arrays-sizes a limit to 2500 test runs have been set, this should let the computation process "warm-up" and also giving a large collection of test data. As the array-sizes increase, we have chosen to decrease the number of test runs. The main reason to do this is that as the array grows so does the computation time, and with an increased computation time the impact of other background processes will have a smaller effect. The time for sorting up to 100 million elements also starts taking a couple of seconds, so doing many tests runs start taking unnecessary amount of time.

3.1.1 Generating Data

With defined sizes of the arrays containing test data, the next step would be to fill these arrays with actual data appropriate for sorting. The most common types of sorting would be to sort positive integers (i.e. $\{1, 2, 3, ...\}$), and in Java one can then use `short`, `int` or `long` as the variables.

Of these data types `short` can only hold values up to around 32 thousand ($2^{15} - 1$) which with arrays ranging up to 100 million elements, the number of duplicate values would be very high. `long` on the other hand would be overkill with the ability to store values up to $2^{63} - 1$, but would also be a doubling of bit-size compared to `int` which also would have impact memory and performance wise. The choice of using 32-bit `int` as elements is best suited for the array-sizes chosen, the signed `int` in Java may contain positive values from 1 to about 2.1 billion ($2^{31} - 1$).

In Listing 3.2 a basic function is shown, this function will take two parameters; the wanted size of the array and a "seed" for the random data creation. After creating the random number generator `rdm` and construct `array` with the desired size, a for-loop will fill each index of the array with random values ranging from 1 to `array.length`. By using the same seed when initializing an array with a specific size, this function will create an array which will contain identical values in each index every time. This is useful when wanting to benchmark the algorithms more than once and/or on other computers and still have the same starting point.

```
int[] initArray(int size, int seed) {
  Random rdm = new Random(seed);
  int[] array = new int[size];

  for (int i = 0; i < array.length; i++) {
    array[i] = rdm.nextInt(array.length) + 1;
  }

  return array;
}
```

Listing 3.2: Initialize an array with data

If one want to have completely random data each time, the `Random(seed)` can easily be changed to just `Random()` which then uses the systems clock rather than a given `seed`, ensuring randomized data with each and every use. But since we want to produce fair and comparable data to be used for all implementations in this thesis, we are going to use the same `seed` every time.

3.2 Benchmarking

3.2.1 Method

To benchmark the algorithms in this thesis, the main method used is practical measurements of run time. By measuring and comparing the time it takes to sort the variety of different array-sizes, as seen in Listing 3.1, we can see how well they perform compared to each other.

In Java there are currently two built-in functions which let the user retrieve time, and with a start and stop time we are then able to see how much time was spent performing the computation. These functions are:

- `System.currentTimeMillis()`
- `System.nanoTime()`

`System.currentTimeMillis()` is based on the computers system-clock. This function actually returns the difference (in milliseconds) between the current time and midnight, January 1, 1970 UTC. This can be used to measure elapsed time but the function has some weaknesses; the system-clock is in no way perfect, it may drift off and occasionally needs to be corrected. How often the system-clock *ticks*, increasing the unit of time, also depends on the underlying operating system.

Included in Java 5 was the function `System.nanoTime()`, when this function is called it returns a number in nanoseconds from a fixed but arbitrarily chosen point in time, a time that may also be in the future. Since the purpose for this function is to measure elapsed time, it is unaffected by the small corrections done by currentTimeMillis.

Another possible measuring method would be **profiling**. With a good profiling tool you will not only find answer to the execution time, but also get a better overview to find:

- What methods are called the most?
- What methods are using the largest percentage of time?
- What methods are calling the most-used methods?
- What methods are allocating a lot of memory?

There are a lot of both commercial and non-commercial profiling tools for Java out there, and Java even has an included lightweight alternative called `HPROF` that is capable of presenting CPU usage and heap allocation statistics.

Since this thesis is mainly focused on investigating different concurrent mechanisms in Java, we are not really after implementing the *"perfect"* algorithm. But interested in measuring how the different mechanisms perform compared to each other. So **profiling** the different implementations to find CPU usage and memory leaks is not really an issue. Comparing `System.nanoTime()` and `System.currentTimeMillis()`, we find that `nanoTime()` is the best alternative if one does not have a program that have to rely on system-date.

3.2.2 Median or Average

As mentioned in the previous Subsection 3.1 - *Test Data Structure* we want to do many test runs on each dataset, but how should one treat this run time collection of test runs. Two of the easiest and best suited solutions here would be to either:

A. Sum all test results and get the average value.

B. Sort all the results and pick the median value.

Since the purpose is to summarize a set of test runs by a single typical value. The *average* would not be the best alternative in this case, as the average is more sensitive if non-typical values would occur. When measuring with only `System.nanoTime()` other background processes on the computer could cause the sorting process to get delayed and may be giving a much higher measured time than usual.

In the following Listings 3.3, 3.4 and 3.5 the `ExecutorService` implementation of Quicksort was tested with an array-size of 50 000 and with three different number of runs; 10, 100 and 1 000. The idea was to see how the average and median time differ from each other.

```
Size: 50000 | Runs: 10 | Seed: 3141592 | Cores: 8

QuickParExec:
1.  Run : 12.238287 ms
2.  Run : 6.884612 ms
3.  Run : 4.980912 ms
4.  Run : 3.635897 ms
5.  Run : 3.645102 ms
6.  Run : 3.791753 ms
7.  Run : 3.657373 ms
8.  Run : 3.651544 ms
9.  Run : 3.681611 ms
10. Run : 3.631295 ms
------------------------------
Average time: 4.979838 ms   // (totalTime / runs)
Median time : 3.657373 ms   // sort(Times), pick(runs / 2)
==============================
```

Listing 3.3: Quicksort; 10 runs

The first thing to notice is that the first three runs always takes longer time, which results in causing a great impact on the average times when few test runs are done (1.32 ms “slower” than median with 10 runs).

```
Size: 50000 | Runs: 100 | Seed: 3141592 | Cores: 8

QuickParExec:
1.  Run : 12.492011 ms
2.  Run : 6.870806 ms
3.  Run : 4.918938 ms
4.  Run : 3.707996 ms
5.  Run : 3.669339 ms
* * *
98. Run : 3.660442 ms
99. Run : 3.659521 ms
100.Run : 3.674861 ms
------------------------------
Average time: 3.805993 ms   // (totalTime / runs)
Median time : 3.664123 ms   // sort(Times), pick(runs / 2)
==============================
```

Listing 3.4: Quicksort; 100 runs

When increasing the number of runs we see much less impact, but the average value still does not give out a typical run time.

```
Size: 50000 | Runs: 1000 | Seed: 3141592 | Cores: 8

QuickParExec:
1.  Run : 12.32051 ms
2.  Run : 6.873567 ms
3.  Run : 4.979072 ms
4.  Run : 3.663509 ms
* * *
998.  Run : 3.650931 ms
999.  Run : 3.681304 ms
1000. Run : 3.657681 ms
------------------------------
Average time: 3.690158 ms   // (totalTime / runs)
Median time : 3.658294 ms   // sort(Times), pick(runs / 2)
==============================
```

Listing 3.5: Quicksort; 1000 runs

Even though one could increase the number of test runs and that the impact will be much less with the increased array-size. The conclusion is to use median time to get the typical value for a test run, and it is these values that we have used for future drawing of graphs and calculating the differences in the next chapters.

3.3 Hardware and Software

With the idea of making many different implementations of the sorting algorithms as discussed in Chapter 4, it is necessary to keep using the

same hardware with every test run. This we do in order to ensure that the measured times can be correctly compared to each other, giving us the information on how well each implementation perform.

3.3.1 Specification

For the development and benchmarking the following hardware in Table 3.1 has been used. Even though it would be very interesting to see how well each implementation would have worked on different kind of architecture and with a wide variety of N-cores CPUs, it was not possible with the time available for this thesis.

CPU	Intel Core i7 920 @ 3.5GHz Quad-Core
RAM	6 GB DDR3 @ 1333MHz
OS	Windows 7 64-bit / Linux Mint 13 64-bit

Table 3.1: Hardware used for Experiments

The hardware is used within a modern desktop computer, and where almost all the parts are from early 2009. The CPUs stock Clock Speed is 2.66 GHz, but have been overclocked to 3.5 GHz which will result in a bit faster computation time making it in more up to par with the current generation of Intel Core i7 models.

In Figure 3.1[9] we have an overview of cache sizes, memory controller and the *QuickPath Interconnect* speed. QuickPath Interconnect replaces what may be better known as the Front Side Bus (FSB). The i7 CPU is in the microprocessor family know as **Bloomfield** under the **Nehalem** microarchitecture.

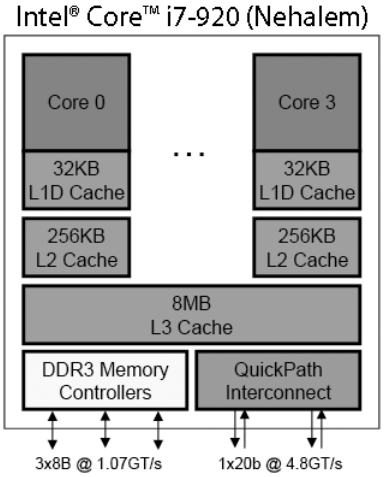

Figure 3.1: Nehalem Architecture

While the main platform used for this thesis is Windows 7 for developing and testing, Chapter 6 also includes some overhead testing on Linux. The Linux distribution of choice ended up being Linux Mint 13, which is based on Ubuntu 12.04 with the current kernel version 3.2.

The Java versions used for benchmarking and testing are Java SE 7 (build 1.7.0_03) and Java SE 6 (build 1.6.0_31) acquired from Oracle's own website under Windows. While the same versions and builds are used under Linux, the runtime environment is `OpenJDK` an open-source implementation of the 6th and 7th edition of the Java SE Platform.

3.3.2 Cache

To reduce the average time to access memory, the CPU has its own cache. The cache is a memory bank between the main memory and the CPU, which stores copies of the data from the most frequently used main memory locations. The cache is spilt into multiple levels with today's modern processors usually up to three levels, which varies in access times and sizes.

When reading and writing to cache, the most used terms are *cache hit* and *cache miss*. A cache hit is when a cache access successfully finds the requested data. A cache miss is when a cache access failed to find the requested data. Larger caches have better hit rates, but usually longer latency.

In Figure 3.2 the application known as CPU-Z from the developer CPUID have been used. This application let the user see the time consumption in CPU-cycles for each cache-level when it read / write to it, and also gives the information on the cache-size for each level.

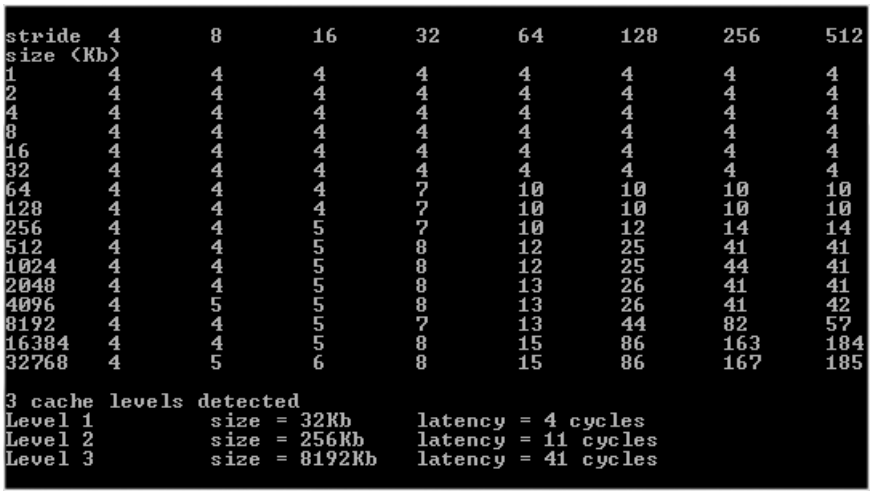

Figure 3.2: Cache Latency

- **Level 1-cache (L1)** is the fastest cache available on a CPU, but also the smallest cache, and is the first cache that gets checked. If it hits, the processor proceeds at high speed. In modern processors this cache is actually split into two caches of equal size, one to store program data and the other to store instructions.
- **Level 2-cache (L2)** will be the next cache to be checked if L1 misses. L2 is bit larger compared to L1, but alas a bit slower. It also stores both program data and instructions.
- **Level 3-cache (L3)** takes approximately ten times longer to access than L1. While the other caches are individual to each core (shared with Hyper-Thread), this cache is usually shared between all cores.
- **Main memory (RAM)**, finally we may need to read / write to the main memory. Here the access time are usually a few hundreds CPU-cycles, on our CPU it is around $\sim$185 [10].

3.4 Java Virtual Machine (JVM)

Java programs are usually compiled into a format known as Java byte-code. Byte-code is comparable to the format that our computers understand natively, but no mainstream processor understands Java byte-code. Instead compiled Java programs usually require a translation layer for them to run the code. And it is this layer that we call a Java Virtual Machine, or short just JVM. This is a standardized execution environment developed by Sun Microsystems (now merged with Oracle Corporation) that Java programs may run within.

3.4.1 Tuning

The JVM has more than hundreds of options that can be tuned to tweak performance or to customize the behavior of a running virtual machine. At start-up there are three categories of options that can be given to the virtual machine:

- **Standard options**: They are supported for any version of JVM. Though some options are specific to different operating system.
- **-X options**: Are non-standard, means that they do not guaranteed to be supported on all JVM implantations, and may also change without

notice in new releases of JDK.

- **-XX options**: Also non-standard, but are also not stable and not recommended for casual use.

For concurrent programming most of the options are with `-XX`, meaning that when we tune the JVM with these options we must keep in mind that it is a certain risk and that the possibility of the option used on the current JVM may not exist on another.

But some of the `-X` options are also useful for concurrent programming even though it was not what they were mainly designed for. Two of these options are the setting of the initial heap size; where the default initial heap size is set to $^1/_{64}$ of the machine's physical memory or some reasonable minimum, and the maximum heap size; which by default is set to $^1/_4$ of the physical memory with an upper threshold at 1 GB. These two is important when running memory intense programs and can be set by using the options below:

- `-Xms <initial java heap size>`
- `-Xmx <maximum java heap size>`

This is two of the options that are used for this thesis as the different implementations require much more memory, than the default sizes set by the JVM, when sorting arrays up to 100 million elements.

Since each thread also gets its own memory stack, parallel implantations with a lot of threads will also have an increase in memory requirements. As of Java 6, in a Windows environment this stack size is default set to 320kb in 32-bit VM and 1024kb in 64-bit VM[11]. This will also be an issue with `ThreadPools`; even though we may have a set amount of threads the application may create many task instances again rising the memory requirement.

Chapter 4

Sorting Algorithms

The programming language **Java** is the language of choice, mainly because it is a language that I always wanted to get more familiar with, it offers a great collection of tools when dealing with concurrency and it is also a modern and popular language which has been covered in many books and papers. This combination gave a smoother access to the world of concurrent and parallel programming in Java.

To test out the various frameworks and some of its classes presented in Chapter 2 - **Concurrency and Threads in Java**. We needed to have some algorithms with different characteristics to try out, and for each algorithm a couple of different implementations should be considered to be able to try out some varieties of tools. This would let us see the effect of different parallel implementations of the algorithm, and getting some comparable results.

4.1 Which Sorting Algorithms

The choices of sorting algorithms have been based on trying to find types which let us use a wide range of different tools from the concurrency package in Java. Therefore we are after algorithms that achieve the following:

- Sorting in parallel is feasible. The algorithms does not necessary have be specifically constructed for parallel computing, as we want to see the effect of going from sequential to parallel. But we should be able run at least some parts of the algorithms concurrently, so that we can expect some performance boost with multi-core.

- Using two distinct algorithms that uses **shared** and **non-shared** data should help us to be able to use a larger variety of tools.
- Popular or at least commonly known sorting algorithms and not some obscure implementation of these.

4.2 Quicksort

Quicksort is a sorting algorithm developed by Charles Antony Richard Hoare or more commonly known as Tony Hoare. While he was abroad as a student at *Moscow State University*, working in a project on machine translation, he developed this specific algorithm to be used in sorting translated words. The algorithm was presented as an article in the *The Computer Journal*[12] in 1962, and is a comparison sorting algorithm that is based on the *divide-and-conquer* paradigm.

The most common implementations of Quicksort are so called *in-place* version and also uses recursion. Each recursive call needs to store some local variables on the stack, which usually grows up to $\mathcal{O}(logn)$ in memory-space. The algorithm sorts an input of n element with the average of $\mathcal{O}(nlogn)$ comparisons. In a worst-case scenario $\mathcal{O}(n^2)$ comparisons would be needed to sort the input sequence, though this behavior is rare.

The pseudocode for a Quicksort algorithm is written in Listing 4.1, and are based on the code from the book *Introduction to Algorithms*[13].

```
QUICKSORT(A, p, r)
  if p < r
    q = PARTITION(A, p, r)
    QUICKSORT(A, p, q - 1)
    QUICKSORT(A, q + 1, r)


PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
    if A[j] <= x
      i = i + 1
      exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

Listing 4.1: Quicksort pseudocode

The fundamental steps for the sorting are rather simple:

1. Pick an element `q`, the pivot, from the array `A`.
2. Partitions the remaining elements into those greater than and less than the pivot `q`.
3. And recursively repeat the process on the partitions, until the array is completely sorted.

There are at least two things that make Quicksort a suitable algorithm for parallelism. *First* it does not have any shared data, meaning no need for synchronization, as the pivot split the array in two each iteration.

Second each part of the array would recursively call Quicksort dividing it in such a way that they can be assigned for the available resources on the system.

4.2.1 Implementation

Based on the previous mentioned pseudocode of Quicksort, the code in Listing 4.2 is a rather straight forward implementation of Quicksort in Java.

```
class QuicksortSequential {
  void quicksort(int[] array, int left, int right) {
    if (left < right) {
      int pivotIndex = partition(array, left, right);
      quicksort(array, left,  pivotIndex - 1);
      quicksort(array, pivotIndex + 1, right);
    }
  }

  int partition(int[] array, int left, int right) {
    pivotValue = array[right];
    int index = left;

    for (int i = left; i < right; i++) {
      if (array[i] <= pivotValue) {
        swap(array, i, index);
        index++;
      }
    }

    swap(int[] array, index, right);
    return index;
  }

  void swap(int[] array, int left, int right) {
    int temp = array[left];
    array[left] = array[right];
    array[right] = temp;
  }
}
```

Listing 4.2: Sequential Quicksort

With only some variable declaration, name differences and the included swap function, one will easy notice the similarity in the code with Listing 4.1. But one problem with this sequential implementation is *when* the worst-case scenario happens. When the pivotValue (line number 11 in Listing 4.2) is set by the rightmost (or could also be leftmost) integer in the array, worst-case would actually be to sort an already sorted array and the same goes if it is in reverse order.

There is more than one solution to help dealing with this problem, we can:

A. *Choose the integer in middle.*

B. *Choose the median of three integers[14].*

C. *Choose a random integer from the array.*

Listing 4.3 show the usage with option **A**, which is used for the Quicksort implementations in this thesis.

```
int pivotValue = array[(left + right) / 2];
swap(array, (left + right) / 2, right);
```

Listing 4.3: Pivot Select

To make the Quicksort algorithms more like the built-in `Arrays.sort()` in Java, we have included an insertion sort for small arrays. The current threshold for when to use insertion sort is set to **47**, the same threshold currently used in `Arrays.sort()` for Java 7 (`build 1.7.0_03`)[15].

```
void insertionSort(int[] array, int left, int right) {
  for (int i = left + 1; i <= right; i++) {
    int a = array[i];
    int j;
    for (j = i - 1; j >= left && a < array[j]; j--) {
      array[j + 1] = array[j];
    }
    array[j + 1] = a;
  }
}
```

Listing 4.4: Insertion Sort

The Listing 4.4 includes the code used in all Quicksort implementation in this thesis, and is called when an array (or part of an array while sorting) contains 47 or less elements. By including this we should see a performance boost as insertion sort is very efficient for small arrays.

4.2.2 Test Run

To compare how well this implementation of Quicksort are to the `Arrays.sort()`, a test run has been done. What size and type of data that is used to create the graph are described in greater detail in Chapter 3 - **Test Environment**.

As shown in Figure 4.1, the sequential Quicksort implementation without the insertion sort usually runs at 75 to 90% speed compared to the built-in `Arrays.sort()`. But by including the insertion sort with the same threshold, the computation speeds are very much alike. The `Arrays.sort()` is a Dual-Pivot Quicksort implementation by Vladimir Yaroslavskiy[16].

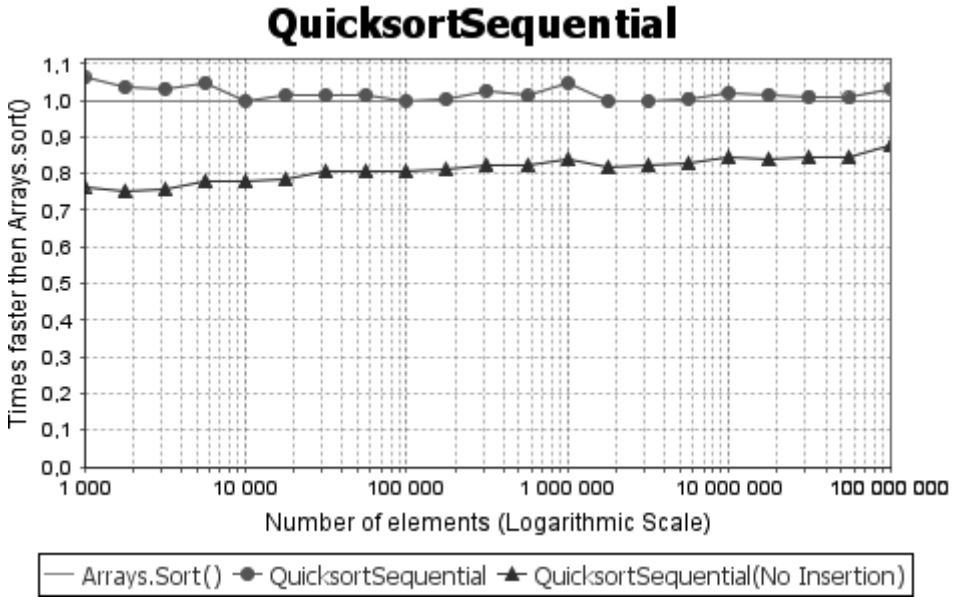

Figure 4.1: Sequential Quicksort

4.3 LSD-Radixsort

Contrary to Quicksort, Radixsort is a non-comparative sorting algorithm that can be dated back as far as the 1890s. Radixsort being a part of the work by Herman Hollerith, an American statistician, who back then worked on *tabulating machines*. But it was later converted to a computer algorithm in 1954 at Massachusetts Institute of Technology (MIT) by Harold H. Seward.

There are essentially two types of Radixsort. The first is the Least-Significant-Digit (LSD), which sort with keys from rightmost digit to the

leftmost digit. The second one is the Most-Significant-Digit (MSD) which goes the other way around. Both illustrated in Figure 4.2.

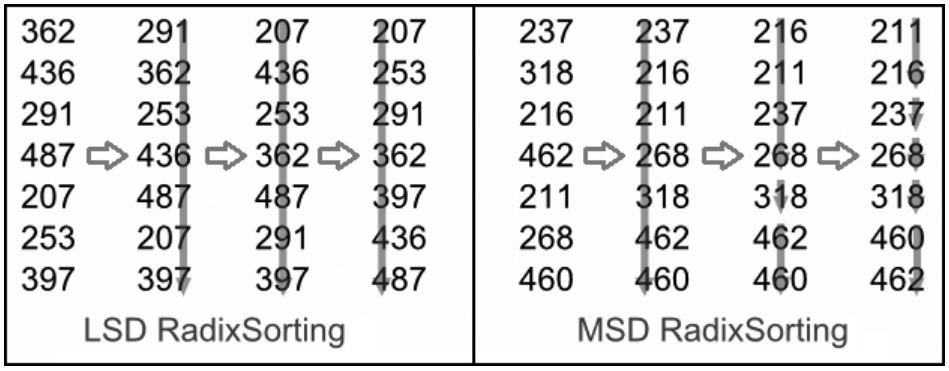

Figure 4.2: LSD- / MSD-Radixsort

In this thesis we have chosen the LSD implementation. With Radixsort we will also have **shared**-data, resulting in that we will have to use tools for synchronization when we construct parallel implementations of this algorithm.

There are also no recursive calls, meaning that we would have to look into another way to split the work among the available resources on the system.

4.3.1 Implementation

A sequential algorithm has been implemented using inspiration from the book *Algorithms*[14] by Robert Sedgewick and Kevin Wayne, which in the book uses Radixsort for string sorting. The code in Listing 4.5 is an implementation for sorting the variables we chose; `int`.

```
class LSDRadixSequential {
  void radixsort(int[] array) {
    int max = 0, bits = 1, length = array.length;

    // Find max value in array
    for (int i = 0; i < length; i++) {
      if (array[i] > max) max = array[i];
    }

    while (max >= 1<<bits) bits++;

    int[] temp  = new int[length];
    int[] count = new int[1<<bits];

    // Compute frequency counts
    for (int i = 0; i < length; i++) {
      count[array[i]+1]++;
    }
```

```
20      // Compute frequency cumulates
        for (int i = 1; i < (1<<bits)-1; i++ ) {
22        count[i] += count[i-1];
        }
24
        // Distribute the records
26      for (int i = 0; i < length; i++) {
              temp[count[array[i]]++] = array[i];
28      }

30      // Copy back
        for (int i = 0; i < length; i++) {
32            array[i] = temp[i];
        }
34    }
  }
```

Listing 4.5: Sequential LSD-Radixsort

The algorithm does the sorting in what can be defined as five steps:

1. Start of by finding the maximum value in the whole `array`. And finding how many bits this value consist of in the binary numeral system.
2. Count how many elements it is of each value in the `array`.
3. Add up values in `count`, accumulating the values.
4. Distribute the values sorted to `temp`.
5. Copy back the elements from the temporary array to the original.

2-Digit

By using only one digit with one full pass of the algorithm to completely sort the array, we get a very fast algorithm for small arrays[10]. But as the array grow in size following our test data structure; the 1-digit Radixsort will continuously decrease in performance.

To *"slow"* down this process we can extend the algorithm to use more passes, and we did this by implementing a 2-digit version of the LSD-Radixsort. While this version will be slower on small array, it will perform much better on larger ones, overall giving a better performance.

```
1 int bit1 = bits/2,
      bit2 = bits-bit1;
3
  radixsort2(array, temp, bit1, 0);
5 radixsort2(temp, array, bit2, bit1);
```

Listing 4.6: Sequential 2-Digit LSD-Radixsort

With the code in Listing 4.6 we show how we split the 1-pass digit, `bits`, into a 2-pass with two equally sized smaller digits. We have then put the sorting steps into its own method, calling it two times with the necessary parameters to do the two full passes now required to sort the array.

4.3.2 Test Run

In Figure 4.3 a test run was done for the 1-digit and 2-digit LSD-Radixsort, comparing how well they performed against each other and `Arrays.sort()`.

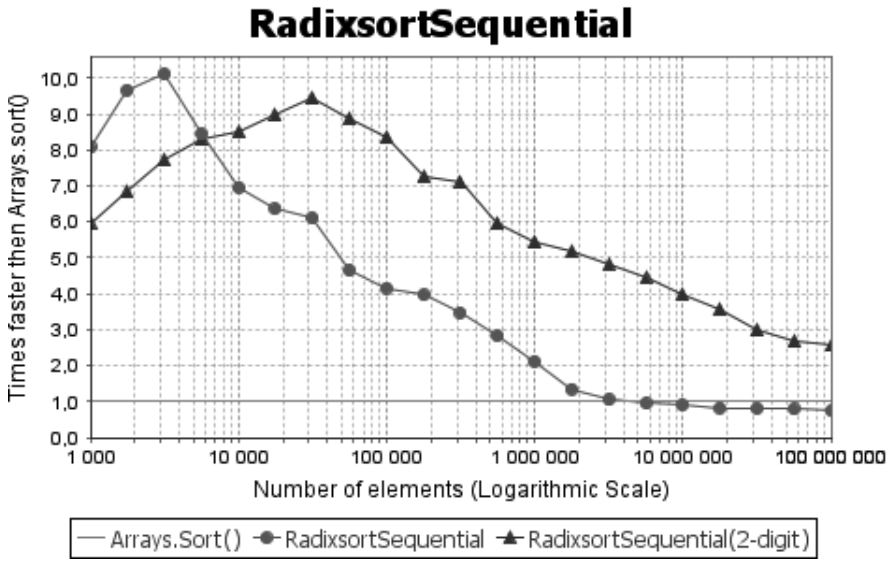

Figure 4.3: Sequential Radixsort

From the graph we can see that 1-digit Radixsort performs **very** well on small arrays, reaching a peak at ∼3000 elements, but decreases in performance as the array grows. And start to perform worse than `Arrays.sort()` with ∼5 million elements.

The 2-digit implementation start of slower, but keeps on climbing and reaching maximum performance at ∼30 000 elements. The 2-digit also performed better than `Arrays.sort()` on large arrays. Comparing the two implementations of Radixsort we see that the 2-digit version have the best overall performance and is the one we chose to implement in parallel.

Chapter 5

Experiments

The goal of the following experiments has been to try out many of the various tools included in the Java concurrency packages. This to see if it is easy and effective to create parallel versions of common sorting algorithms, and what the actual performance gain can be compared to a regular sequential implementation.

For this thesis it is the two algorithms from the previous Chapter 4 that we will create parallel implementations of; **Quicksort** and **2-digit LSD-Radixsort**. This chapter includes the ideas, how we implemented the parallel versions with different tools and finally the tests comparing the results with the built-in `Arrays.sort()`.

5.1 Parallel Quicksort Implementations

With the divide-and-conquer pattern that Quicksort is built on, it is no need for synchronization as the array that we are going to sort will be divided for each recursive call and which will result in that we have no shared data. With no share data and an algorithm that split into smaller tasks for us, we only need to have tools for executing these tasks in parallel.

Going through Java API documentation, we ended up with three different tools to create parallel implementations of Quicksort. The parallelization will be achieved with the following scenarios:

- An *"naive"* attempt by only creating new **Threads**.
- Using the included **ExecutorService** in Java.
- Using the new **Fork/Join**-framework included in Java 7.

This will let us see how well the implementations by just creating new threads for every task versus the reusing of threads in a pool perform compares to each other. We will also see if the new framework Fork/Join does have any advantages compared with the older ExecutorService.

5.1.1 Naive

The naive attempt differs from the sequential Quicksort in that way that each time Quicksort recursively calls itself; it instead creates new threads which then call. And by creating these new threads, each core (included Hyper-Threads) on the current system can then be assigned the available threads and do the computation in parallel, which then should result in increasing the overall performance. The Listing 5.1 contains code for this implementation.

But with larger and larger arrays to sort, even more threads will be created in the naive attempt of Quicksort. As creating threads causes overhead and with a new thread for each recursive call, the algorithm would result in poor performance if we do not limit the creation process somehow.

```
1  class QuicksortParallel implements Runnable {
     private final int[] array;
3    private final int left, right;
     final static int INSERTION_SORT_THRESHOLD = 47;
5
     public QuicksortParallel(int[] arr, int l, int r) {
7      // Constructor
     }
9
     public void run() {
11     quicksort(array, left, right);
     }
13
     void quicksort(int[] array, int left, int right) {
15     if (right-left <= INSERTION_SORT_THRESHOLD) {
         insertionSort(array, left, right);
17     }

19     else {
         int pivotIndex = partition(array, left, right);
21       Thread t1 = new Thread(
             new QuicksortParallel(array, left, pivotIndex - 1));
23       Thread t2 = new Thread(
             new QuicksortParallel(array, pivotIndex + 1, right));
25
         t1.start();
27       t2.start();
         try {
29         t1.join(); t2.join();
         } catch (InterruptedException e) {}
```

```
31      }
    }
33
    int partition(int[] array, int left, int right) {
35    int pivotValue = array[(left + right) / 2)];
      swap(array, (left + right) / 2, right);
37    int index = left;

39    for (int i = left; i < right; i++) {
        if (array[i] <= pivotValue) {
41        swap(array, i, index);
          index++;
43      }
      }
45
      swap(array, index, right);
47    return index;
    }
49

51  void swap(int[] array, int left, int right) {
      int temp = array[left];
53    array[left] = array[right];
      array[right] = temp;
55  }
}
```

Listing 5.1: Parallel Naive Quicksort

To reduce the number of threads to create we can try by finding the optimal granularity. This is no simple task when using different systems with various numbers of cores, clock-speed and memory bandwidth to run the algorithm. One approach would be to do a live profiling to give the desired granularity for each system. But as the test environment in this thesis is mainly focused on one system there is an easier solution. In Listing 5.2 is a simplified alternative, using a hardcoded threshold to limit the creation of threads.

```
final static int LIMIT = 50000;
2
void quicksort() {
4 ...
  if (right-left <= LIMIT) {
6   /** If current part of the array-size is less than the set
    LIMIT, we call quicksort() recursive with current thread */
8 }
  else {
10  /** Else we create two new threads to call quicksort() */
  }
12 }
```

Listing 5.2: Granularity; limit the creation of threads for Quicksort

The result of using this IF-check is that it will also reduce the number of additional threads to create, and on this naive implementation we will only have one thread when the array-sizes is less or equal to 50 000 elements.

Test Run

After we have including the hardcoded threshold to the code in Listing 5.1, the next step is doing a test run and comparing it to `Arrays.sort()`. The test can be seen in Figure 5.1.

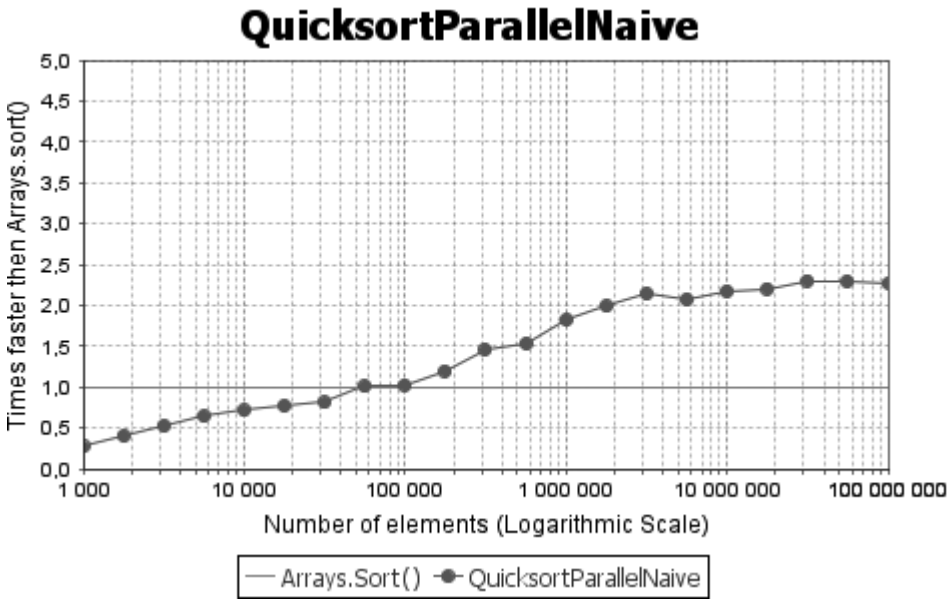

Figure 5.1: Parallel Naive Quicksort

Because we now start the sorting by allocating a new thread for the `QuicksortParallel` class, the overhead will cause such an impact on the measured times that it performs worse when sorting less than ∼50 000 elements. A solution to this problem could have been to do add an array length check before deciding to create and run the parallel implementation, and rather call a sequential Quicksort implementation to get the result from Figure 4.1.

As the number of elements goes above 50 000, the algorithms starts create new threads for the recursive calls. Threads get assigned to different cores on the CPU by the Java Virtual Machine and Operating System. The outcome of this is the performance increase we see in the graph, with the highest at ∼3 million elements being 2.3 times faster than `Arrays.sort()` on our quad-core (8 w/Hyper-Thread) CPU.

5.1.2 ExecutorService

The creation process is a bit different with **ExecutorService** since we have to create a `ThreadPool` with a fixed amount of threads that should be available, and we need a `Future` to keep track of every submitted task. The `Future` will also need a checkpoint to see when the sorting is completed.

An example of starting Quicksort with ExecutorService can be seen in Listing 5.3. The static integer `CORES` when we create the `ThreadPool` are initialized using `Runtime.getRuntime().availableProcessors()` to get the current systems available processors (included Hyper-Threads). On the system used for this test this would mean **eight**.

```
// Create the fixed Threadpool for the ExecutorService
final ExecutorService pool = Executors.newFixedThreadPool(CORES);
//  and a future to keep track of when tasks are done
List<Future> futures = new Vector<Future>();



// Instantiate the Quicksort implemented with ExecutorService
QuicksortExecutor QE = new QuicksortExecutor
      (QuickArray, 0, QuickArray.length-1, futures, pool);

// Submit the task to the Threadpool and add it to Future
futures.add(pool.submit(QE));

// Wait for all tasks to complete
while(!futures.isEmpty()) {
  Future topFeature = futures.remove(0);
  try {
    if(topFeature!=null)topFeature.get();
  } catch(...) { ... }
}
pool.shutdown();
```

Listing 5.3: ExecutorService Start Process

When all tasks are complete this means that we are done sorting, and finally we will shut down the `ThreadPool`.

The rest of the implementation is rather identical to the *naive* attempt, with only some additions of parameters and the submission of new Quicksort calls to the `ThreadPool`. The code can be seen in Listing 5.4.

```
class QuicksortExecutor implements Runnable {
  ExecutorService pool;
  List<Future> futures;

  ...

  void quicksort(int[] array, int left, int right) {
    if (right-left <= INSERTION_SORT_THRESHOLD) {
      insertionSort(array, left, right);
    }

    else {
      int pivotIndex = partition(array, left, right);

      if (right-left <= LIMIT) {
        quicksort(array, left,  pivotIndex - 1);
        quicksort(array, pivotIndex + 1, right);
      }

      else {
        futures.add(pool.submit(new QuicksortExecutor
            (array, left,  pivotIndex - 1, futures, pool)));
        futures.add(pool.submit(new QuicksortExecutor
            (array, pivotIndex + 1, right, futures, pool)));
      }
    }
  }
  ...
}
```

Listing 5.4: Parallel Quicksort using ExecutorService

Test Run

As seen in Figure 5.2; compared to the naive attempt we get a much better performance boost on the test run using the ExecutorService. The main reason for this increased performance is that with a fixed `ThreadPool`, the algorithm only creates the fixed amount of threads at the beginning and then reuses these threads when new tasks are submitted.

Considering that we achieved a computation time of almost 4.75 times faster than the sequential implementation, we think it is a pretty good outcome on the quad-core CPU tested with here. And with some tweaking to both the algorithm and the Java Virtual Machine, we think it should be able to break the 5 times mark fairly easily.

As with the naive attempt the overhead will also here have such an impact to the measured times that it is less efficient on small arrays. Another solution to the problem could be to combining the algorithm with another that is more ideal for these kinds of sizes.

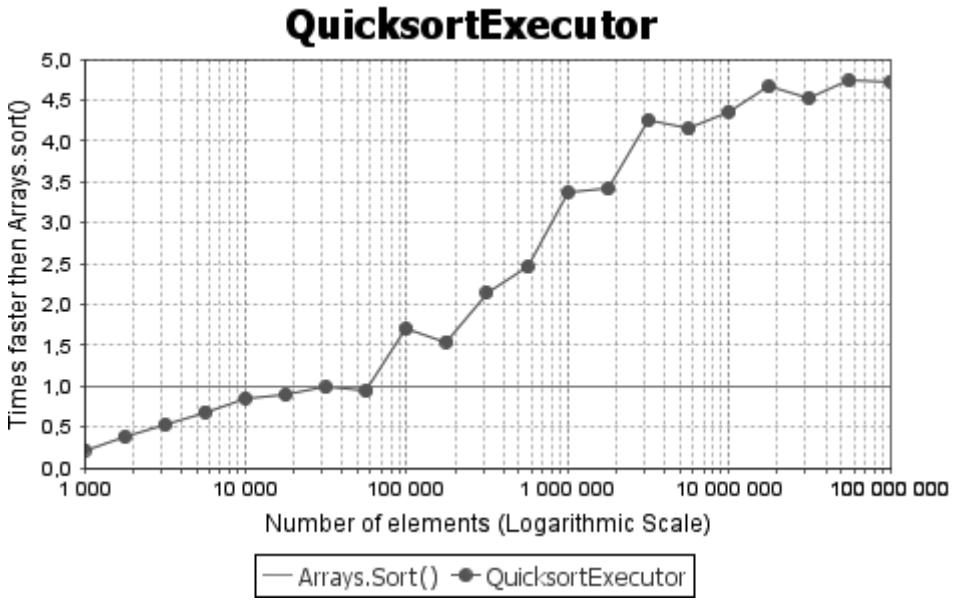

Figure 5.2: Parallel Quicksort using ExecutorService

5.1.3 Fork/Join

One of the new and very interesting feature introduced in Java 7; is the **Fork/Join**-framework which is described in greater detail in Subsection 2.2.7.

As with the ExecutorService we create a fixed pool to contain the available worker threads, and then `submit` tasks to this pool. But while the submitting with ExecutorService submits a value-returning task for execution and return a `Future` representing the pending results of the task. The Fork/Join uses `invoke` on tasks, this performs the given task and returning its result upon completion. This means that there is no need to use `Future` to keep track of the tasks.

Creating the Fork/Join-framework only requires specifying the number of desired threads for the pool (You can actually just call `ForkJoinPool()`, which uses `Runtime.getRuntime().availableProcessors()` as default), example like we did with ExecutorService:

```
ForkJoinPool QParFJ = new ForkJoinPool(CORES);
```

Rest of the implementation can be seen in Listing 5.5.

```
class QuicksortForkJoin extends RecursiveAction {

  ...

  protected void compute() {
    quicksort(array, left, right);
  }

  void quicksort(int[] array, int left, int right) {
    if (right-left <= INSERTION_SORT_THRESHOLD) {
      insertionSort(array, left, right);
    }

    else {
      int pivotIndex = partition(array, left, right);

      if (right-left <= LIMIT) {
        quicksort(array, left, pivotIndex - 1);
        quicksort(array, pivotIndex + 1, right);
      }
      else {
        invokeAll(
          new QuicksortForkJoin(array, left, pivotIndex - 1),
          new QuicksortForkJoin(array, pivotIndex + 1, right));
      }
    }
  }
  ...
}
```

Listing 5.5: Parallel Quicksort using Fork/Join

Test Run

The Figure 5.3 show the test run with the Fork/Join implementation.

If we compare the different graphs, Fork/Join-framework is actually slower compared to ExecutorService. This is surprising as we thought the Work-Stealing aspect would increase the performance as none of the worker threads would idle (i.e. they got dealt a smaller portion of the array that is sorted and finished earlier).

But over 4 times as fast compared to the sequential implementation, is in our eyes still considered a good performance boost on a quad-core CPU.

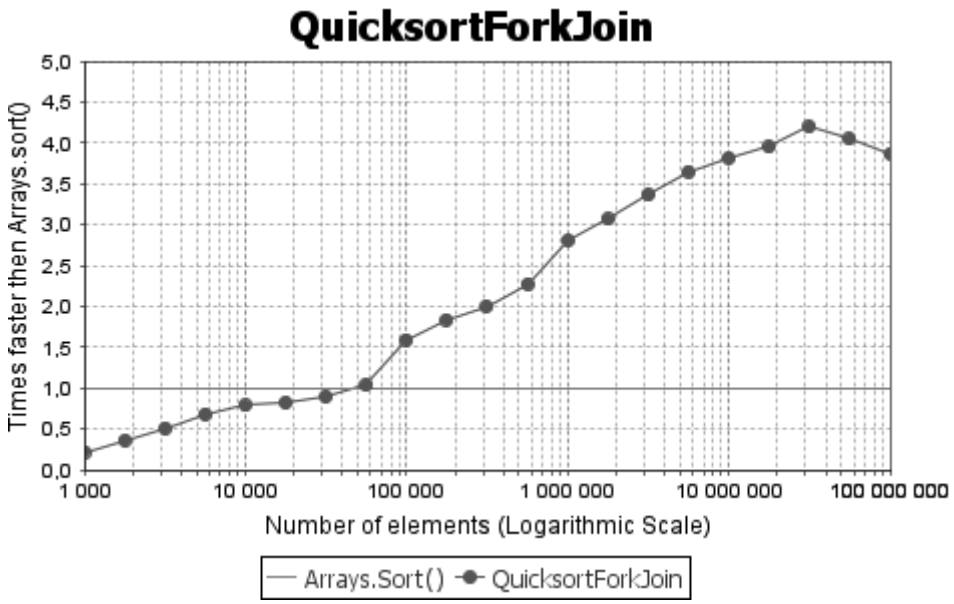

Figure 5.3: Parallel Quicksort using Fork/Join

5.2 Parallel LSD-Radixsort Implementations

Contrary to Quicksort it is necessary to use some sort of synchronization to create parallel versions of LSD-Radixsort. This is because each step in Radixsort has to be completely done before the next operation can take place. We will also have to look into how we divide the work among the available resources and how to treat the shared data for the algorithm.

For the parallel implementations of LSD-Radixsort we will use the same tools, but with various tweaking to try improving the parallel computation time. After going through the Java API documentation we ended up with `CyclicBarrier` for synchronization and `AtomicIntegerArray` for storing the shared data. This allowed us to create the following scenarios:

- CyclicBarrier and Atomicity
- Non-Atomic with Duplicate Reads
- Non-Atomic with Duplicate Data

While there are other mechanisms available for synchronization between the different steps when sorting, we chose `CyclicBarrier` because first of we do want to use tools from the concurrent packages rather than `wait/notify`, `Condition` etc... Secondly we also want to reduce the number of declarations and reuse them, which would not be possible with the CountDownLatch.

Another possible mechanism would be to use Phaser, but we chose not to use this because we do not need the dynamic functionality it provides, and while doing some minor testing we found that the performance was almost identical compared to CyclicBarrier.

5.2.1 CyclicBarrier and Atomicity

With the overhead that occur by creating threads for computing in parallel, we want to check the length of the array before we start sorting to determine if we should sort the array sequentially (2-digit LSD-Radixsort from Section 4.3.1) or in parallel.

As we did with Quicksort (Listing 5.2) we choose how to sort by setting a `LIMIT` as seen in Listing 5.6. Here we do an IF-check to see if the arrays length is less or equal to the set threshold, which are the same value used in the parallel implementations of Quicksort, 50 000, a LIMIT that was confirmed to be a good balance between overhead and sorting time on the previous test runs.

```
1  final static int LIMIT = 50000;

3  void radixsort(int[] array) {

5    ...

7    if (array.length <= LIMIT) {
       /** If the arrays length is less than set LIMIT,
9        we sort the array sequentialy as seen in Section 4.3 */
     }
11
     else {
13     /** Else we initiate the process to run with worker threads
         and sort the array in parallel with these, Listing 5.7 */
15   }
   }
```

Listing 5.6: Radixsort, when to sort Sequential/Parallel

When we are going to sort arrays that exceed 50 000 elements, we would have to initiate the parallel computation process which use worker threads (classes with runnable) for sorting. But before starting these threads we first need to sequentially find the maximum value for the whole array, declaring the two bits for running the algorithm with 2-digit and create the two `AtomicIntegerArray` as the shared counters.

```
// Find max value in array
for (int i = 0; i < array.length; i++) {
  if (array[i] > max) max = array[i];
}

// Convert the max value to bits
while (max >= 1<<bits) bits++;
//   and split it for 2-digit
bit1 = bits/2;
bit2 = bits-bit1;

count1 = new AtomicIntegerArray(1<<bit1);
count2 = new AtomicIntegerArray(1<<bit2);

// Create and start the worker threads
for (int i = 0; i < CORES; i++) {
  new Thread(new Worker(i)).start();
}

// Wait for all Worker Threads to finish (i.e. sorting is done)
try {
  finished.await();
} catch (...) { ... }
```

Listing 5.7: Radixsort, sequential part

The created worker threads contain a runnable instance that will first find its *"own"* portion of the array which it will sort. The workers are numbered from 0 to CORES-1 (`Runtime.getRuntime().availableProcessors()-1`), which helps us divide the array into equally sized pieces. Overview of the Worker class is seen in Listing 5.8.

```
class Worker implements Runnable {
  int threadNumber, left, right;
  double part;

  Worker(int threadNumber) {
    this.threadNumber = threadNumber;
  }

  public void run() {
    // Divide the array into equally sized pieces
    part = array.length/CORES;
    left = (int)(threadNumber*part);

    //   and making sure they do not overlap
    if (threadNumber == CORES-1) {
      right = array.length-1;
    }

    else {
      right = (int)((threadNumber+1)*part)-1;
    }


```

```
    // Do a 2-pass Radixsort where each worker thread
    //   focus on their "own" part of the array [left...right]
    radixsort2(array, temp, bit1, 0,
          left, right, count1, threadNumber);
    radixsort2(temp, array, bit2, bit1,
          left, right, count2, threadNumber);

    // Wait for all other threads to finish
    try {
      finished.await();
    } catch (...) { ... }
  }
}
```

Listing 5.8: Radixsort, the Worker Threads

We use two CyclicBarrier for synchronization. The first one is `finish` which will not open before every worker thread plus the main thread is done computing, confirming that the array is sorted and we are done. The second is `sync` which we use as a synchronizer between each step when sorting.

Test Run

Like with Quicksort the test run for the Radixsort implementations are done using the previously mentioned system and structure in Chapter 3. Figure 5.4 show how the well the first implementation of a parallel version of Radixsort performed by using CyclicBarrier and Atomicity as tools.

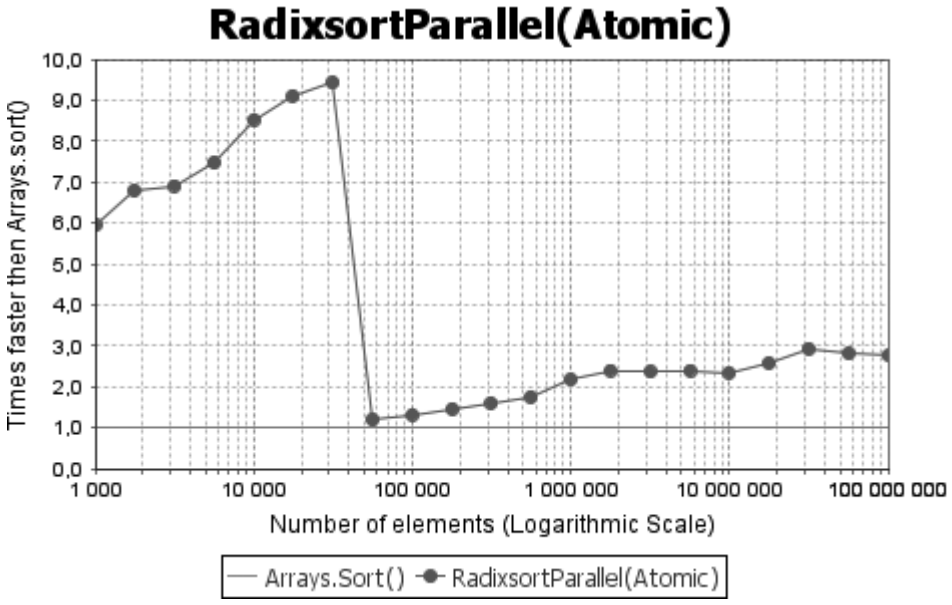

Figure 5.4: Parallel Radixsort using CyclicBarrier and Atomicity

Compared with the sequential 2-digit Radixsort from Chapter 4, Figure 4.3, we see that the beginning of test run is roughly the same. This is of course the result of our limitation of when to start computing in parallel.

As soon as the arrays length exceed 50 000 elements the parallel sorting begin. The measurement shows us that it performs quite badly compared to the sequential implementation (∼9 times faster), and that we have to sort array length above 30 million elements before we see any benefit with the parallel version.

We have reason to believe that the bad performance is a combination of the overhead for creating and calling worker threads, the synchronization using the two `AtomicIntegerArray count[]` and that some steps of the sorting are performed sequentially.

5.2.2 Non-Atomic with Duplicate Reads

To try improving the parallel LSD-Radixsort algorithm, an idea is to remove the need for our synchronization on the count arrays (`count1` and `count2`). Which would also give us opportunity to change out the `AtomicIntegerArray` with regular `int[]`, removing the extra overhead that occur by using atomic variables.

We did this by dividing which values each worker will sort rather than which part of the array (`array[]`) as in the previous implementation. This means that all threads go through the whole array, but only addressing their *"own"* values and as a result no need for synchronization on the `count[]` arrays. The new improved `radixsort2` can be seen in Listing 5.9:

```
void radixsort2(int[] array, int[] temp, int bit1, int bit2,
        int left, int right, int[] count, int threadNumber) {

  // Compute frequency counts for values that this worker owns
  for (int i = 0; i < array.length; i++) {
    int j = (array[i]>>bit2) & (1<<bit1)-1;
    if (left <= j && j <= right) { count[j]++; }
  }

  // Synchronize - wait for all worker threads
   try {
       sync.await();
   } catch (...) { ... }

  // Sequential compute frequency cumulates
  if (threadNumber == 0) {
    int k = 0, l;
    for (int i = 0; i < (1<<bit1); i++) {
```

```
      l = count[i]; count[i] = k; k += l;
    }
  }

  // Synchronize
  try {
    sync.await();
  } catch (...) { ... }

  // Distribute the records for values that this worker owns
  for (int i = 0; i < array.length; i++) {
    int j = (array[i]>>bit2) & (1<<bit1)-1;
    if (left <= j && j <= right) temp[count[j]++] = array[i];
  }

  // Synchronize
  try {
    sync.await();
  } catch (...) { ... }
}
```

Listing 5.9: Radixsort, Duplicate Reads

For the partition of the array each worker will now assign their `left / right` variable by shifting the `bit1 / bit2`. The updated `Run()` implementation for the workers can be seen in Listing 5.10.

```
// Divide the array into equally sized pieces
part = (1<<bit1)/CORES;
leftValue = (int)(threadNumber*part);

//   and making sure they do not overlap
if (threadNumber == CORES-1) {
  rightValue = (1<<bit1)-1;
}
else {
  rightValue = (int)((threadNumber+1)*part-1);
}

// Do a 1.pass Radixsort where each worker thread focus
//   on their "own" value in the array
radixsort2(array, temp, bit1, 0,
      leftValue, rightValue, count1, threadNumber);

/* Partition with (1<<bit2) as above, and do 2.pass */

radixsort2(temp, array, bit2, bit1,
      leftValue, rightValue, count2, threadNumber);
```

Listing 5.10: Radixsort, Duplicate Reads `run`-method

Even though we have to go through the whole array this should reduce the overhead as there is no need to use the `getAndIncrement()`, and there is no synchronization on the array when updating it.

Test Run

Even though we have to go through the whole array this should reduce the overhead for not using atomic variables, and there is no more synchronization on the counters when sorting the array. We see that in Figure 5.5 we have gained a small performance boost overall compared to the previous implementation, but still the sequential Radixsort have a much better performance.

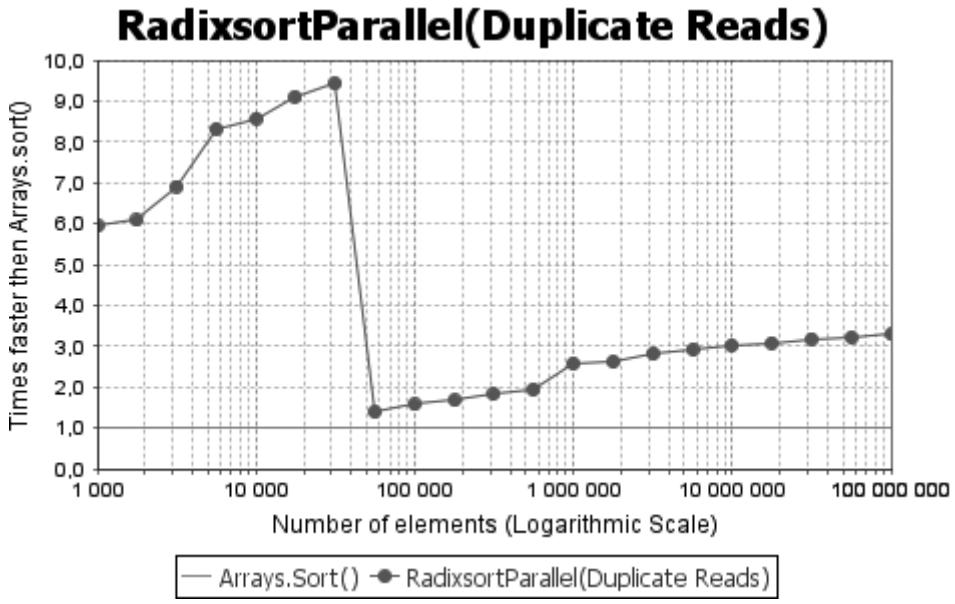

Figure 5.5: Parallel Radixsort with Duplicate Reads

5.2.3 Non-Atomic with Duplicate Data

We suspect that the sequential parts of our Radixsort are the cause of the poor performance, so the next step would be to try parallelizing these parts. The following section and code are based on the work by Arne Maus[17].

The first thing we want to do is to find the maximum value in parallel. We do this by dividing the array into equally sized pieces like we did in Listing 5.8, then each of the workers will go through his part (`[leftIndex...rightIndex]`) and find the local maximum value. After every worker has completed this, we then proceed to pick out the highest maximum value from this collection.

```
// Find local max value in this workes part of the array
for (int i = leftIndex; i <= rightIndex; i++) {
  if (array[i] > max) max = array[i];
}

localMax[threadNumber] = max;

// Synchronization - wait for all worker threads to find localMax
try {
  sync.await();
}  catch (...) { ... }

// Compare and find max value
for (int i = 0; i < CORES-1; i++) {
  if (localMax[i] > max) max = localMax[i];
}
```

Listing 5.11: Radixsort, find maximum value in parallel

Another sequential part from the previous implementations is the computation of frequency cumulates, which was only done by the first worker thread. We have tried to solve this issue by introducing a local count for each worker, which will contain the frequency count for the workers *own* part of the array (`[leftIndex...rightIndex]`).

```
for (int i = leftIndex; i <= rightIndex; i++) {
  localCount[(array[i]>>bit2) & (1<<bit1)-1]++;
}
allCount[threadNumber] = localCount;
```

Listing 5.12: Radixsort, local count

By using the same left / right values from Listing 5.10, each worker can now focus on his *own* values in the array and cumulate these in parallel.

```
// Compute frequency cumulates for values that this worker owns
int k = 0, l;
for (int i = leftValue; i <= rightValue; i++){
  for (int j = 0; j < CORES; j++) {
    l = allCount[j][i]; allCount[j][i] = k; k += l;
  }
}
allValue[threadNumber] = k;

try {
  sync.await();
} catch (...) { ... }

// Accumulate own values
k = 0;
for (int i = 0; i < threadNumber; i++) {
  k += allValue[i];
}
```

```
20 // Update cumulates for values that this worker owns
   for (int i = leftValue; i <= rightValue; i++){
22   for (int j = 0; j < CORES; j++) {
       allCount[j][i] += k;
24   }
   }
```

Listing 5.13: Radixsort, compute frequency cumulates in parallel

The most important factor with this implementation is that each worker has a local version of `count[]` for their own values. This implementation manages to both avoid the synchronization on `count[]` and it also share data and values.

Test Run

With Figure 5.6 we clearly see a performance boost from using this implementation.

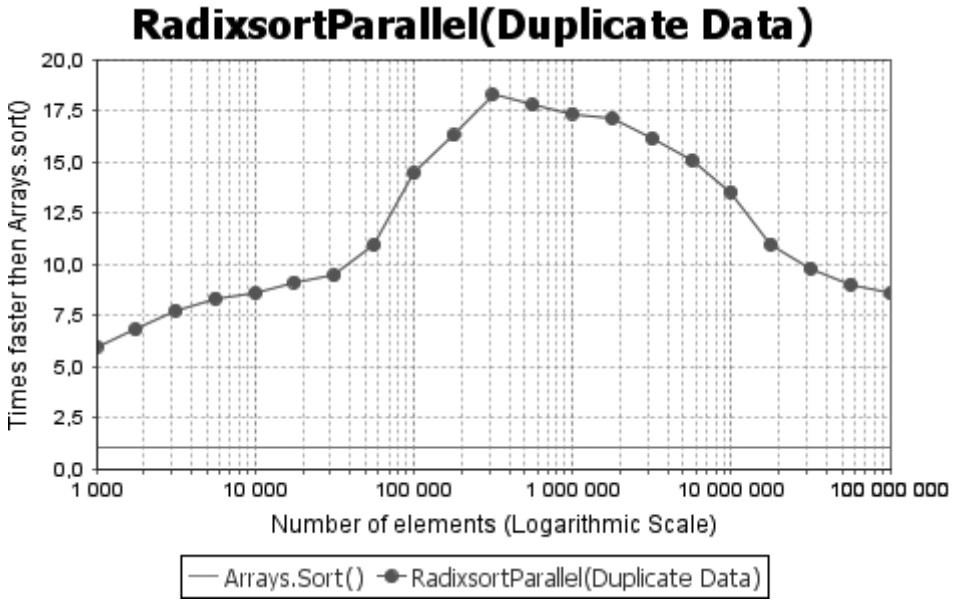

Figure 5.6: Parallel Radixsort with Duplicate Data

One of the most important factor compared to those seen previously is that when we start to run the algorithm in parallel (∼50 000 elements) it keeps on climbing, reaching maximum performance at 300 000 elements and when compared to `Arrays.sort()` more than 18 times faster!

5.3 Test Cases with more Cores

After experimenting and testing on the local system we wanted to expand the testing to systems with more cores to see how well the different implementations scale.

The following tests in this section were done at the Department of Informatics at the University of Oslo. The system we tested on has four octa-core CPUs, each with eight additional threads available via **Hyper-Threading**. This brings the total available threads up to 64. Table 5.1 show the specification of the system, which all is run on a Linux environment.

CPU	4x Intel Xeon L7555 @ 1.87GHz Octa-Core (32 total, 64 w/Hyper-Threading)
RAM	128 GB @ 1066MHz
OS	Red Hat Enterprise Linux 4 64-bit (Kernel: 2.6.18-308.8.2.el5)

Table 5.1: Hardware for Test Cases with more Cores

5.3.1 Java 7 Utilities when using Java 6

Since the current version of Java used at the University is Java 6 (`build 1.6.0_31-b04`) it does not include the Fork/Join-framework, which was part of the Java 7 update. This means that we would have to include an extra package when compiling and running the Java-code at this system. By using the preview version from the *"Concurreny JSR-166 Interest Site"*[4], we can import the extra package called `jsr166y.jar` which includes most of the new classes that appeared in Java 7.

This package can easily be included in the program by implementing the code: `import jsr166y.*;` . And then putting `jsr166y.jar` in the same folder as the Java-code. When compiling and running the Java-code the file has to be included in the Classpath (-cp). Like shown in the following compiling example:

```
javac -cp ".:/home/testing/jsr166y.jar" QuicksortTesting.java
```

As of today JSR-166 Interest Site also have a new package called `jsr166e.jar`; which is a preview version of the new concurrency classes that may appear in Java 8. This package of course requires Java 7.

5.3.2 Quicksort Test Runs

Comparing the graphs in Figure 5.7 with the one from Subsection 4.2.2, we see that this sequential Quicksort is somewhat faster when comparing it to `Arrays.sort()`. The reason for this is that we use an earlier version of Java where the `Arrays.sort()` is less optimized, which is why our sequential Quicksort have around 30% performance *"boost"*.

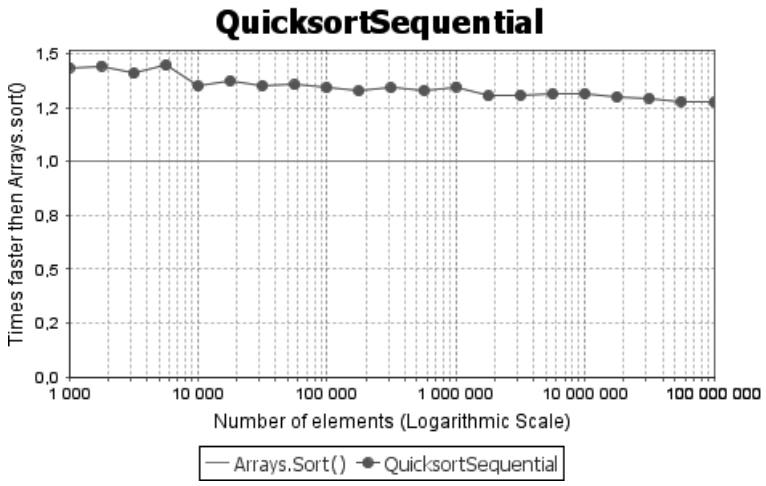

Figure 5.7: Sequential Quicksort on the 32-core CPU

The *naive* parallel implementation of Quicksort as seen in Figure 5.8 ends up being around double as efficient as previous test run (Figure 5.1) with elements from 200 000 to 2 million. But considering that we use 8 times the number of cores, we are far away from a linear scaled performance increase.

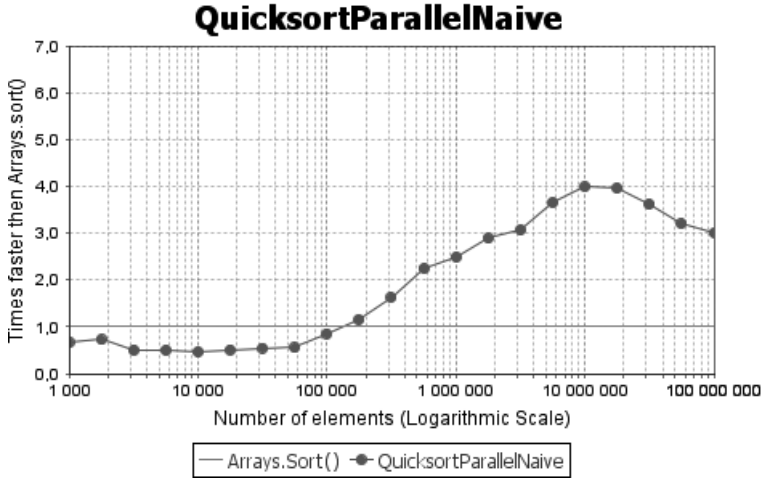

Figure 5.8: Parallel Naive Quicksort with 32-cores (64 w/HT)

With 64 threads the ExecutorService, Figure 5.9, remains as the most effective Quicksort implementation of the once we tested. But again it is far away from what we hoped for with the severely increase in number of cores, and the only really noticeable performance gain was the last test with 100 million elements. Strangely we also see a drop in performance between 1 to 3 million elements.

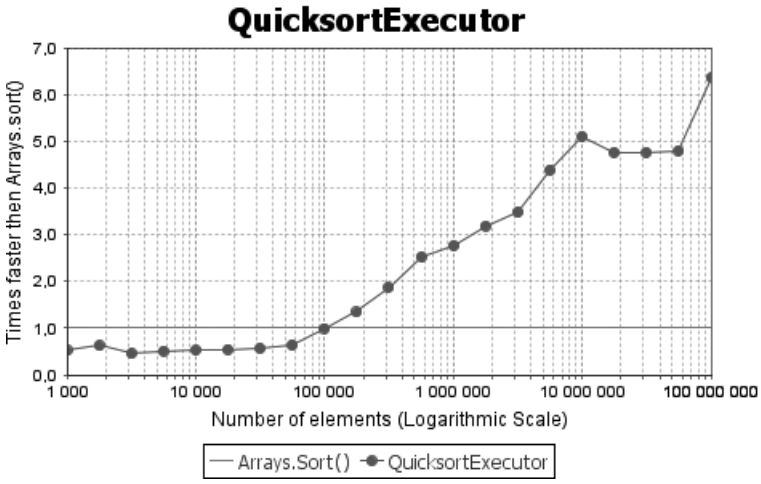

Figure 5.9: Parallel Quicksort using ExecutorService; 32-cores (64 w/HT)

The same can be said of the Fork/Join implementation in Figure 5.10. On average the 32-core test run is actually also less efficient than on our quad-core test when excluding the sequential parts (less than 50 000), with a 288% versus 296% increase in performance.

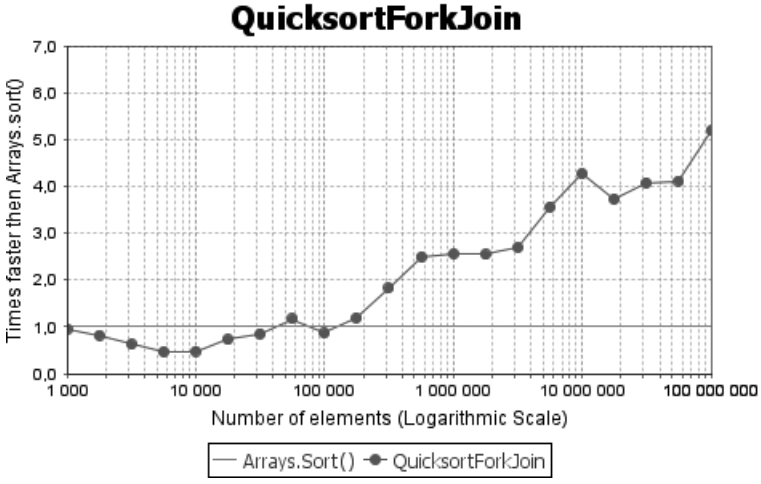

Figure 5.10: Parallel Quicksort using Fork/Join; 32-cores (64 w/HT)

5.3.3 LSD-Radixsort Test Runs

With the sequential LSD-Radixsort test run at the University, we see in Figure 5.11 that because of the less optimized version of `Arrays.sort()` the sequential Radixsort performs *better* than on our own system.

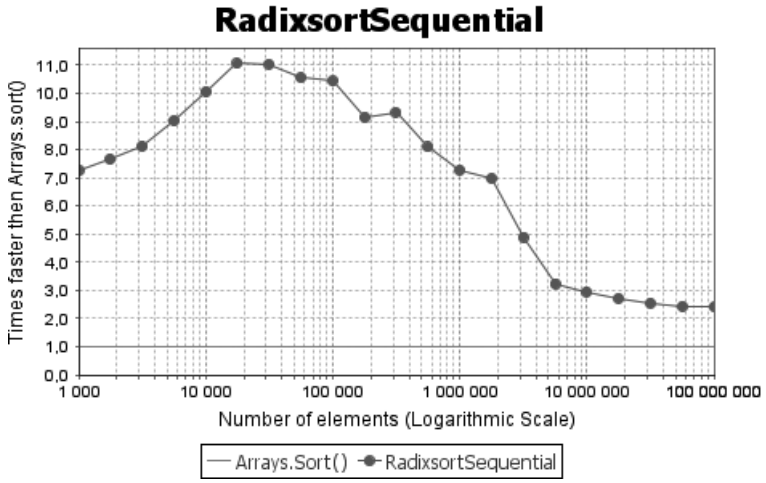

Figure 5.11: Sequential Radixsort on the 32-core CPU

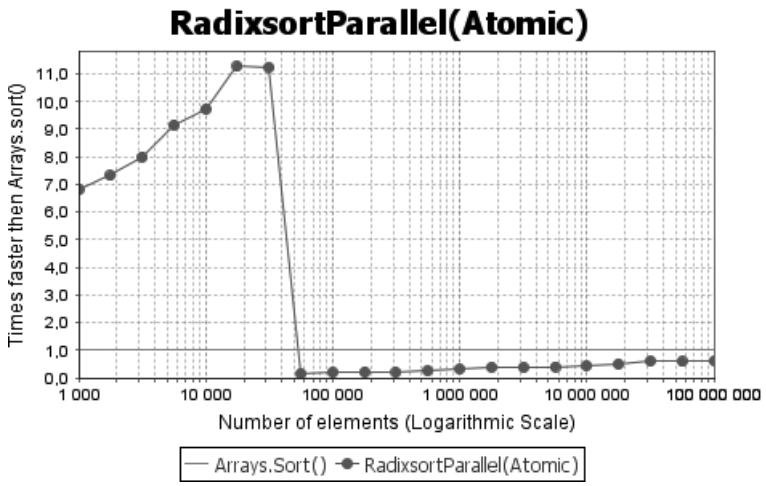

Figure 5.12: Parallel Radixsort using CyclicBarrier and Atomicity; 32-cores

Both test runs in Figure 5.12 (Atomic) and 5.13 (Non-Atomic with Duplicate Reads) shows the same problem as seen in Subsection 5.2.1 / 5.2.2; as soon as the parallel computation begins at 50 000 elements we see a huge drop in performance, which with 32-cores is even worse than on the quad-core test. This strengthen the reason to believe that the overhead for creating and calling workers combined with synchronization and some of the steps

are performed sequentially have a too much negative effect on the sorting times.

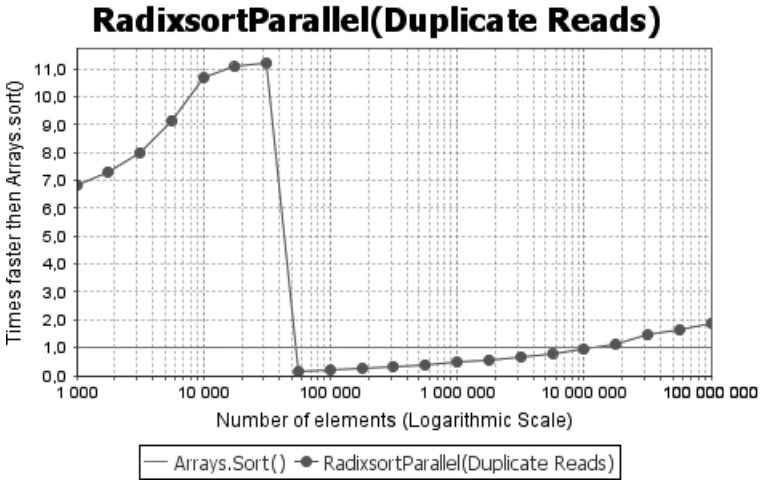

Figure 5.13: Parallel Radixsort with Duplicate Reads; 32-cores (64 w/HT)

Finally on the parallel implementation with duplicate data, Figure 5.14, it starts off slow but reaches very good results as the array grows. It is clear that using a threshold around 50 000 does not go very well with parallel Radixsort on 4x octa-cores, we should have set a larger threshold and gradually increase the number of worker threads to create.

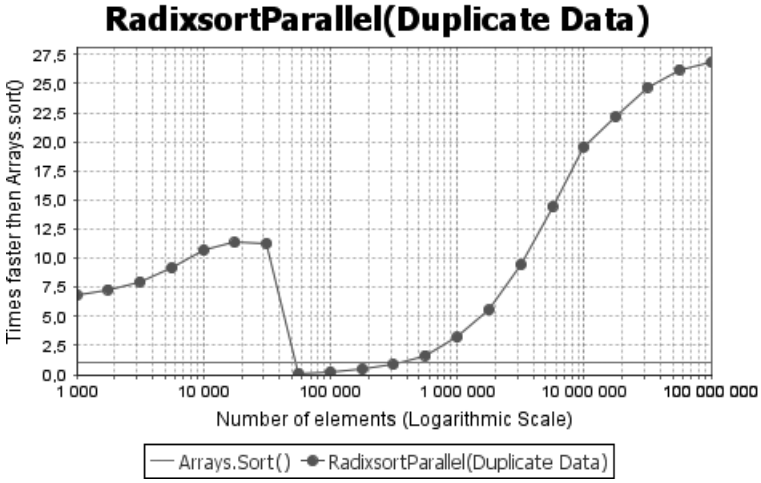

Figure 5.14: Parallel Radixsort with Duplicate Data; 32-cores (64 w/HT)

Chapter 6

Threads and Overhead

To explain the results in Chapter 5, we will in this chapter look closer at **threads** and the **overhead** that occurs from generating and calling them.

Overhead is the extra work the operating system must do to create and manage multiple threads or a concurrent framework. So with a parallel implementation of a program, the overhead could result in a loss in performance if threads and concurrent frameworks are used for too small problems.

By looking into this, it will help us give a few ideas to when we should or should not use threads or a concurrent framework. Listed below are the different mechanisms that we wanted to do some test runs on to measure the overhead that occur when creating and calling:

- **Methods**
- **Classes**
- **Threads**
- **ExecutorService**
- **Fork/Join-Framework**
- **Variables** `(int)`
- **Atomic Variables**

While there are a lot of other mechanisms that would be interesting to look into, the ones that are chosen are the ones previously used in Chapter 5 - **Experiments**. This will also help us figuring out if there are any correlations between the mechanisms overhead and performance gain.

The chapter starts off by going through how the tests are measured on both Windows and Linux, before it presents each of the mechanisms result in its own subsection and finally comparison of the results.

6.1 Measurement Method

6.1.1 Test Program Structure

To test out the different mechanism mentioned at the beginning of this chapter, we first start of by making a program structure for the test runs. This would have to contain the creation and calling of the mechanism, where each part have to include a time for when it starts and when it is done, giving us the computation time which we then can use to get comparable results.

In Listing 6.1 we can see the basic structure that will test out the various mechanisms, creating and calling the number of times we wish to test. When dealing with the `ExecutorService`- and `Fork/Join`-framework, the `ThreadPool` will only be created once at the beginning and each call is a `submit/invoke`.

```
1  // mechanism: may be method, class or thread
   {
3    currentMechanismNr = 0;
     numberOfMechanisms = N; // N: total number of mechanisms
5
     startRunTime = System.nanoTime();
7    startCreateTime[currentMechanismNr] = System.nanoTime();
     /** Execute creation of the mechanism or
9        the framework that will be tested */
     doneCreateTime[currentMechanismNr] = System.nanoTime();
11
     startCallTime[currentMechanismNr] = System.nanoTime();
13   /** Execute call to methodName to start the mechanism */
     totalRunTime = System.nanoTime() - startRunTime;
15 }

17 class className {
     void methodName() {
19     doneCallTime[currentMechanismNr] = System.nanoTime();
       // if more mechanisms being tested, create/call recursively
21     if (currentMechanismNr < numberOfMechanisms-1) {
         int i = ++currentMechanismNr;
23
         startCreateTime[i] = System.nanoTime();
25       /** Execute creation of additional mechanisms */
         doneCreateTime[i] = System.nanoTime();
27
         startCallTime[i] = System.nanoTime();
29       /** Execute call to the additional mechanisms */
       }
31   }
   }
```

Listing 6.1: Overhead Testing Structure

Usually one would fill the `methodName` with code to actually doing something (i.e. Sorting, calculation, merging... etc), but as we only want to do measurements on the overhead, we do not want to have any additional code for this. So the tests will call *"empty"* methods for the different mechanism, giving us the cost of each call.

For variables and atomic variables we will do different operations rather than calls. Each operation will be performed a given number of times in a for-loop, the structure can be seen in Listing 6.2:

```
// operation: Get, Set, Increment or Decrement

startOperations = System.nanoTime();
for (int i = 0; i < numberOfTimes; i++) {
  /** Perform operation N-times */
}
operationsTime = System.nanoTime() - startOperations;
```

Listing 6.2: Overhead Atomic Testing

6.1.2 Windows and Batch-file

In Chapter 3 we discussed how important it is to do more test run to get a more precise result while doing the experiments. For the overhead testing we want to do the same thing; first do many test runs on each of the different mechanisms, and collect the result of each run. Then sort these collections and pick out the median values as mentioned in Subsection 3.2.2 - Median or Average.

To run the tests more than once one would usually just create a loop around the part of the code that is going to be measured, storing each measurement and finally calculate the result. The *"problem"* with doing this while measuring overhead is that the Java Virtual Machine (JVM) does a lot of optimization at runtime. These optimization will impact the measurement and only give us the full cost of overhead the first time we create or call the different mechanisms (`Class`, `Thread`, `ForkJoinPool`... etc). And this is the reason we would want to **"Cold-start"** the test runs each time.

By **Cold-starting** this mean that we want to completely shut down the JVM for each and every test run, so we then can start it up again without **any** of the optimization from the previous run. Since overhead testing takes such a small time to measure and we want to do a lot of runs to get more accurate results, letting the JVM shut down and manually starting up new test would then be a very tedious task. This is where scripting become

handy and in the Windows environment we can then use Windows own shell script **Batch-files** (.bat).

In the following Listing 6.3, we have the script created for the overhead testing done in Windows for this chapter. It start off by assigning two variables; `numberOfMechanisms` and `numberOfTestRun`, both which is commented in the script. Then it compiles the two Java-files; `OverheadTesting.java` which does the actual overhead testing and writing the results to .txt-files, and `OverheadOutput.java` which reads the .txt-files sorting the results and produce graphs and output suitable for tables.

```
:: Maximum number of mechanisms to test using Powers of Two (2^n)
::  32 will test: { 1 , 2 , 4 , 8 , 16 , 32 }
SET /a numberOfMechanisms = 32

:: Number of test runs, more runs result in more precise output
SET /a numberOfTestRun = 2000

:: Compile Java-files
javac OverheadTesting.java
javac OverheadOutput.java

SET /a i = 1
:outerLoop
:: GTR means Greater than (>)
IF %i% GTR %numberOfMechanisms% GOTO outerEnd
  SET /a j = 1
  :innerLoop
  IF %j% GTR %numberOfTestRun% GOTO innerEnd
    :: Start the overhead testing with current
    ::  number of mechanisms to test with as parameter
    java OverheadTesting %i%
    SET /a j = %j% + 1
  GOTO innerLoop
  :innerEnd
  SET /a i = %i% * 2
GOTO outerLoop
:outerEnd

:: Output results to LaTeX and graphs
java OverheadOutput %numberOfTestRun%
echo Test run done.
```

Listing 6.3: Batch-file for Overhead Testing

By simply executing the Batch-file in the same folder as the two Java-files and with the current variables set in the script, overhead testing will be done with the number of Call / Class / Threads : 1, 2, 4, 8, 16 and 32. Where these all will be executed 2000 times to get a large collection of measurements. The final test results produced by `OverheadOutput.java` can be seen in Section 6.2 and 6.3.

6.1.3 Linux and Bash-file

While the main platform used for this thesis is Windows, after running and collection overhead data we thought it would also be interesting to see how much overhead occur on other platforms. This resulted in trying to do the same tests on the Linux platform to get some comparable data, with the Linux distribution of choice Linux Mint 13.

To do the same test on Linux, also meant that we have to create a script-file for this environment too. As Linux do not have the ability to execute Batch-files, we created a similar script using **Bash-files** (.sh) which can be seen in Listing 6.4.

```
#! /bin/bash

# Maximum number of mechanisms to test using Powers of Two (2^n)
#  32 will test: { 1 , 2 , 4 , 8 , 16 , 32 }
numberOfMechanisms=32

# Number of test runs, more runs result in more precise output
numberOfTestRun=2000

# Compile Java-files
javac OverheadTesting.java
javac OverheadOutput.java

i=1
# -le means Less than or equal (<=)
while [ $i -le $numberOfMechanisms ]; do
  j=1
  while [ $j -le $numberOfTestRun ]; do
    # Start the overhead testing with current
    #  number of mechanisms to test with as parameter
    java OverheadTesting $i
    let j+=1
  done
  let i*=2
done

# Output results to LaTeX and graphs
java OverheadOutput $numberOfTestRun
echo Test run done.
```

Listing 6.4: Bash-file for Overhead Testing

It is a rather straight-forward convert from the Batch-file using Linux $\leftrightarrow$ DOS command equivalences. The results can be seen in Section 6.2 and 6.3.

6.1.4 Hardware

Contrary to the hardware used for the Experiments in Chapter 5 (for specifications see Section 3.3), another hardware setup has been used for the overhead testing.

The reason for this decision is that with the previous hardware there was some measurements trouble when using the built-in Java method `System.nanoTime()`. An example is that the output of `System.nanoTime()` when comparing the start and end time of method-calls sometimes resulted in identical values, which means it took *zero* microseconds to call, a precision issue that (hopefully) can be ignored with slower hardware.

An overview of the hardware can be seen in Table 6.1 and are parts of an old *Dell Inspiron 9300* laptop which released in 2004. This hardware have been used for every test run in Section 6.2, and also for measuring the times for the sequential Quicksort comparison in Figure 6.2, Section 6.3. The *"key"* component is the much slower Intel Mobile singe-core processor, which has a much lower clock-speed and smaller cache-sizes. As there is no need for multi-core either, the single-core does not share caches with other cores (or Hyper-Threads).

CPU	Intel Pentium M 760 @ 2.0GHz Single-Core
RAM	2 GB DDR2 @ 533 MHz
OS	Windows 7 32-bit / Linux Mint 13 32-bit

Table 6.1: Hardware used for Overhead Testing

6.2 Overhead Test Results

The following subsections and its tables contain the test results given by the testing in `OverheadTesting.java`, and output from `OverheadOutput.java`. All the measured times used for this subsections are in microseconds (μs).

These subsections also refer to the Section 6.3 when comparing with the different Quicksort computation times.

6.2.1 Methods

We start off by measuring a static **method** that practically does nothing. But as Java also does some optimization when compiling, the method actually

updates a global *dummy* variable so that the method will be called and not just ignored. The measuring takes this into account so the results seen in Table 6.2 and 6.3 is for the overhead cost of the calls only.

- **First Call**: This is the first call done to the method, and is commonly the call that has the most overhead.
- **Additional Calls**: For the additional method-calls (i.e. for 16 calls this is call number 2 to 16), we will see that the JVMs runtime-optimization comes in to play, dropping the overhead for calling the method.
- **Total Run Time**: The final run time given by `totalRunTime` seen at line 14 in Listing 6.1, it should be very close to sum of all operations done.

MethodCall	Number of Method Calls					
	1	2	4	8	16	32
First Call	4.75	4.75	4.75	4.75	4.75	4.75
Additional Calls	-	2.24	2.05	1.96	1.96	1.96
Total Run Time	4.75	6.71	10.7	18.5	34.1	65.4

Table 6.2: Overhead using Methods on Windows (in μs)

MethodCall	Number of Method Calls					
	1	2	4	8	16	32
First Call	4.54	4.54	4.55	4.54	4.54	4.54
Additional Calls	-	1.61	1.49	1.45	1.43	1.42
Total Run Time	4.54	6.08	8.95	14.6	26.0	48.5

Table 6.3: Overhead using Methods on Linux (in μs)

With the first call taking roughly $\sim$4.75 μs on Windows and $\sim$4.54 μs on Linux, and the additional calls reducing the overhead with 50% to 70%. There is really no reason to take any precaution when dealing with only methods. But it is interesting to see that Linux uses less overhead with methods.

6.2.2 Classes

With **Classes** we will have to instantiate a class by allocation memory for the new object and return a reference to that memory. A task that should cause more overhead compared to testing only methods. An overview of the testing can be seen in Table 6.4 for Windows and 6.5 for Linux.

- **Create the Class**: Instantiate the class with the `new` operator, we have to create it before we can call its methods. This is only performed at the beginning of the testing.
- **First Method Call**: The first call to the class's *"empty"* method.
- **Additional Calls**: The rest of the additional call to the class's *"empty"* method.

ClassCall	Number of Classes					
	1	2	4	8	16	32
Create the Class	759	768	774	774	773	777
First Method Call	11	11	11	11	11	11
Additional Calls	-	4	3	2	2	2
Total Run Time	770	788	802	817	848	914

Table 6.4: Overhead using Classes on Windows (in μs)

ClassCall	Number of Classes					
	1	2	4	8	16	32
Create the Class	494	499	499	500	500	501
First Method Call	12	12	12	12	12	12
Additional Calls	-	3	2	2	2	1
Total Run Time	506	519	525	537	560	607

Table 6.5: Overhead using Classes on Linux (in μs)

With the process of instantiate a class and calling its methods we start to notice some overhead, but mainly because of the creation process. For while the first method-call's overhead more than doubled, $\sim$11-12 μs is not much to take notice of. Also again less overhead occur on Linux in total, and creating a class is noticeably much cheaper.

6.2.3 Threads

It is by using **Threads** that we finally can create and run parallel computations, and it will be very interesting to see how much the cost for using this mechanism is. The results for testing threads with Windows and Linux can respectively be seen in Table 6.6 and 6.7.

- **Create the First Thread**: Instantiate a class extended with a `Runnable` object that is passed to the `Thread` constructor, with the cost of doing so.

- **First `Run()` Call**: Executing *`thread`*`.start()`, starting the object's `Run` method to be called in that separately executing thread.
- **Additional Threads**: Identical to "Create the First Thread", for the rest of the specified number of threads.
- **Additional `Run()` Calls**: Identical to "First `Run()` Call", but for the additional threads.
- **Total Thread Create**: Total cost of creating all Threads, First Thread + All Additional Threads.
- **Total `Run()` Call**: Total cost of the First `Run()` Call + All Additional `Run()` Calls.

ThreadCall	Number of Threads					
	1	2	4	8	16	32
Create the First Thread	759	758	764	767	766	772
First `Run()` Call	839	835	836	815	820	846
Additional Threads	-	37	27	24	22	22
Additional `Run()` Calls	-	188	178	180	170	168
Total Thread Create	-	797	850	935	1101	1456
Total `Run()` Call	-	1039	1388	2118	3425	6050
Total Run Time	1614	1841	2244	3062	4554	7533

Table 6.6: Overhead using Threads on Windows (in μs)

ThreadCall	Number of Threads					
	1	2	4	8	16	32
Create the first Thread	491	492	492	492	492	493
First `Run()` Call	185	185	185	185	186	186
Additional Threads	-	29	22	21	20	20
Additional `Run()` Calls	-	92	90	83	82	84
Total Thread Create	-	521	558	636	789	1102
Total `Run()` Call	-	277	455	763	1417	2796
Total Run Time	676	799	1014	1401	2215	3902

Table 6.7: Overhead using Threads on Linux (in μs)

With the introduction of **Threads** we see that overhead start to become more of an issue. In the time it takes to create and call one thread on Windows, Quicksort have already completed sorting 10 000 elements (1318 μs).

As JVMs optimization from already construction the first thread, the additional threads have a much less overhead cost. The same goes for executing the threads.

6.2.4 ExecutorService

ExecutorService is one of the mechanisms in Java that let us reuse a pool of threads. A solution that will become more and more superior as the number of tasks increases when compared to a new thread for each task. Results can be seen in Table 6.8 and 6.9 (Windows / Linux).

- **Create ExecutorService**: Is the set-up time for the ExecutorService, here we create a fixed `ThreadPool` with the specified number of threads as a parameter. In addition create a `Future` to provide the ability to check when *computations* (calls) are completed.
- **First Submit**: The very first submit done to the `ThreadPool`, with the additional overhead for the system to assign the submitted work to an available Thread in the Pool.
- **Additional Submits**: Every additional submit done to the `ThreadPool`.
- **Total Submit**: Total cost of the First Submit + All Additional Submits.

ExecutorCall	Number of Threads in Pool					
	1	2	4	8	16	32
Create ExecutorService	4881	4861	4874	4870	4835	4957
First Submit	2026	2045	2030	2061	2042	2067
Additional Submits	-	231	238	212	203	214
Total Submit	-	2282	2756	3565	5142	8699
Total Run Time	6915	7169	7694	8534	10123	13734

Table 6.8: Overhead using ExecutorService on Windows (in μs)

ExecutorCall	Number of Threads in Pool					
	1	2	4	8	16	32
Create ExecutorService	4174	4204	4241	4328	4428	4608
First Submit	1086	1081	1085	1094	1083	1077
Additional Submits	-	558	292	193	156	147
Total Submit	-	1642	1973	2465	3435	5689
Total Run Time	5675	5845	6216	6810	7926	10430

Table 6.9: Overhead using ExecutorService on Linux (in μs)

ExecutorService is the mechanism that produces the most overhead. In the time it takes to construct and call with only one thread on Windows, Quicksort is almost done sorting 50 000 elements (7980 μs).

One of the reasons that ExecutorService create so much additional overhead is that we have to add a `Future<>` instance for keeping track on when

the threads are done computing, which is necessary for asynchronous tasks used in this thesis.

But while the ExecutorService generate much more overhead compared to regular threads, the `ThreadPool` in ExecutorService let the user submit additional tasks to the `ThreadPool` for the available threads in the pool to compute (see Section 2.2.1). As the number of tasks reaches a certain threshold, the ExecutorService will be cheaper to use compared to creating a new thread for every task.

6.2.5 Fork/Join

One of the new tools in Java 7 is **Fork/Join**, and like the ExecutorService it contains a pool of threads with the ability to submit (invoke) new tasks to it. The Table 6.10 and 6.11 contains the result of doing an overhead test run with this framework.

- **Create ForkJoin**: Is the set-up time for creating the `ForkJoinPool` with the number of threads specified in the tables as parameter, and the creation of `ForkJoin` class extending `RecursiveAction`.
- **First Invoke**: Invoke (*await and obtain result*) the `ThreadPool`, let the first thread perform the first given call and return upon completion.
- **Additional Invokes**: Identical to "First Invoke", for additional calls.
- **Total Invoke**: Total cost of the First Invoke + All Additional Invokes.

ForkJoinCall	Number of Threads in Pool					
	1	2	4	8	16	32
Create ForkJoin	1728	1741	1775	1782	1806	1866
First Invoke	691	690	683	697	688	692
Additional Invokes	-	10	5	3	3	3
Total Invoke	-	700	698	722	735	789
Total Run Time	2450	2462	2488	2526	2579	2769

Table 6.10: Overhead using Fork/Join on Windows (in μs)

The construction of the **Fork/Join**-framework generate more overhead compared the construction of regular threads, but the overhead for the first and the additional calls (invokes) generate less. As a result of this with eight threads and upward, Fork/Join is cheaper. And with the additional ability to work-steal (Section 2.2.7), and the reuse of threads it is far more superior to use with parallel programming.

ForkJoinCall	Number of Threads in Pool					
	1	2	4	8	16	32
Create ForkJoinPool	1841	1850	1871	1897	1945	2077
First Invoke	405	443	434	437	451	462
Additional Invokes	-	9	4	3	2	2
Total Invoke	-	452	447	457	486	527
Total Run Time	2277	2312	2333	2374	2467	2662

Table 6.11: Overhead using Fork/Join on Linux (in μs)

6.2.6 Atomicity

Another of the tools that have been used in the different implementations are the `AtomicInteger` and `AtomicIntegerArray`, thread-safe variables, which we briefly explained in Section 2.2 about **Concurrency in Java**. With these tools and the same hardware used in the previous sub-sections, we have done measurements on the overhead imposed by utilizing these tools.

In the following Tables there are four different kinds of operations that have been tested for the Atomic variables. Even though there are a couple more operations available, it is these ones that are mostly common:

- **GetAndSet N-times**: For the `int` and `int[]` this operation grabs the value N and sets the `int` or `int[0...N]` to this for an N number of times.
- **Get N-times**: Here we call the `Get()` operation for `AtomicInteger` and `AtomicIntegerArray` N-times.
- **Set N-times**: Here we call the `Set(N)` operation for `AtomicInteger` and `AtomicIntegerArray` N-times, setting the value to N each time.
- **Increment N-times**: This operation perform two things, first it `Get` the value from the source and then `Increment` it; increasing the value with 1. This will be done N-times so the final value if it is not an array will be N, if it is an array each `int` will be 1.
- **Decrement N-times**: This operation perform two things, first it `Get` the value from the source and then `Decrement` it; decreasing the value with 1. This will be done N-times so the final value if it is not an array will be -N, if it is an array each `int` will be -1.

Table 6.12 and 6.13 are both results of non-thread-safe variables performing different computation N number of times.

`int`	Number: N					
	10	100	1000	10^4	10^5	10^6
GetAndSet N-times	0.118	0.251	1.70	14.3	159	1691
Increment N-times	0.152	0.258	1.33	12.1	170	1463
Decrement N-times	0.158	0.270	1.34	12.0	170	1460

Table 6.12: Overhead using `int` on Windows (in μs)

`int[]`	Number: N					
	10	100	1000	10^4	10^5	10^6
GetAndSet N-times	0.119	0.288	1.96	22.6	225	2357
Increment N-times	0.339	0.540	2.22	21.2	208	2182
Decrement N-times	0.347	0.547	2.24	21.1	208	2177

Table 6.13: Overhead using Array of `int` on Windows (in μs)

Table 6.14 and 6.15 are both results of thread-safe variables performing different computation N number of times.

`AtomicInteger`	Number: N					
	10	100	1000	10^4	10^5	10^6
Create AtomicInteger	41.7	42.0	41.7	42.0	42.0	41.6
Get N-times	1.03	1.34	4.16	33.5	333	3166
Set N-times	1.25	1.62	4.84	39.3	380	3448
Increment N-times	1.83	3.08	15.2	123	1267	4490
Decrement N-times	1.91	3.15	15.3	123	1280	4531

Table 6.14: Overhead using AtomicInteger on Windows (in μs)

`AtomicIntegerArray`	Number: N					
	10	100	1000	10^4	10^5	10^6
Create Array[N]	138	139	138	140	173	251
Get N-times	1.37	3.03	19.2	163	688	2951
Set N-times	4.37	5.92	19.2	167	1173	3860
Increment N-times	8.57	11.8	42.4	341	2323	5768
Decrement N-times	8.60	11.9	42.3	360	2365	5805

Table 6.15: Overhead using AtomicIntegerArray on Windows (in μs)

Comparing `int` versus `AtomicInteger` there is roughly 3-times as much overhead when using the `Atomic` version.

With arrays we have `int[]` versus `AtomicIntegerArray` which give 2.5-times as much overhead with the `Atomic` version.

6.3 Comparison

The graph seen in Figure 6.1 contains plots of the Total Run Time for each of the mechanisms tested on Windows. This gives us a good overview of how much overhead that occurs by using the different mechanisms in this thesis

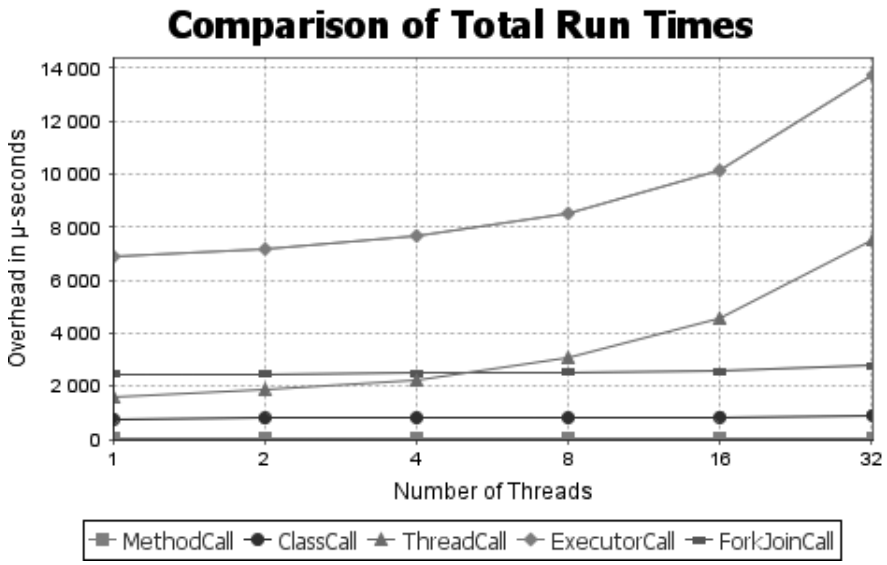

Figure 6.1: Comparison of Total Run Times (in μs)

By using the Sequential Quicksort implementation (without Insertion Sort) from Chapter 4.2, with the hardware specified in the previous Section 6.1.4, we have been able to measure how long it typically takes to sort different number of elements. These measurements are useful when comparing sorting times to the overhead that occur with the different mechanisms in Java.

The Table 6.2 on the right show the time it takes for Quicksort to sort a given number of elements, ranging from 1 000 to 100 000. This implementation is only using one method-call, so the overhead does not have much impact on the sorting time.

Sequential Quicksort

Number of Elements	Sorting Time
1 000	104 μs
2 500	288 μs
5 000	607 μs
7 500	973 μs
10 000	1318 μs
25 000	3632 μs
50 000	7980 μs
75 000	12214 μs
100 000	16980 μs

Figure 6.2: Quicksort Sorting Times

Chapter 7

Discussion

7.1 Amdahl's Law

For us to understand why parallel implementations of algorithms do not scale linearly to the number of available processor cores, we need to reflect on the following *"law"*.

Amdahl's Law[3] says that the theoretically speedup with additional number of cores on a CPU, is based on the proportion of parallelizable and serial components. So for a system to scale linearly to the number of processor cores, it has to be completely parallelized. This is generally not possible as it will always be part of a program that has to run in serial, like splitting up a problem to smaller bits and delegate these to available resources.

With Amdahl's Law we can find the maximum theoretically speedup by using the equation below; we let F be the fraction of the program that must execute serially, and N the number of cores available:

$$speedup \leq \frac{1}{F + \frac{(1-F)}{N}}$$

As the number of cores increases to infinity, we see that the expression reduces to $\frac{1}{F}$. This helps us easily define how much the maximum achievable speedup is, no matter how many cores we have available. For example: if our program does 5% of the computation serially, and the rest 95% with an unlimited number of cores; we still can only achieve a theoretically speedup of measly 20 times! $(\frac{1}{0.05} = 20)$

In Figure 7.1 we show how the speedup scale with number of cores on different parallel portions, and it is easy to see that eventually the improvement by introducing additional cores will flat out the speedup.

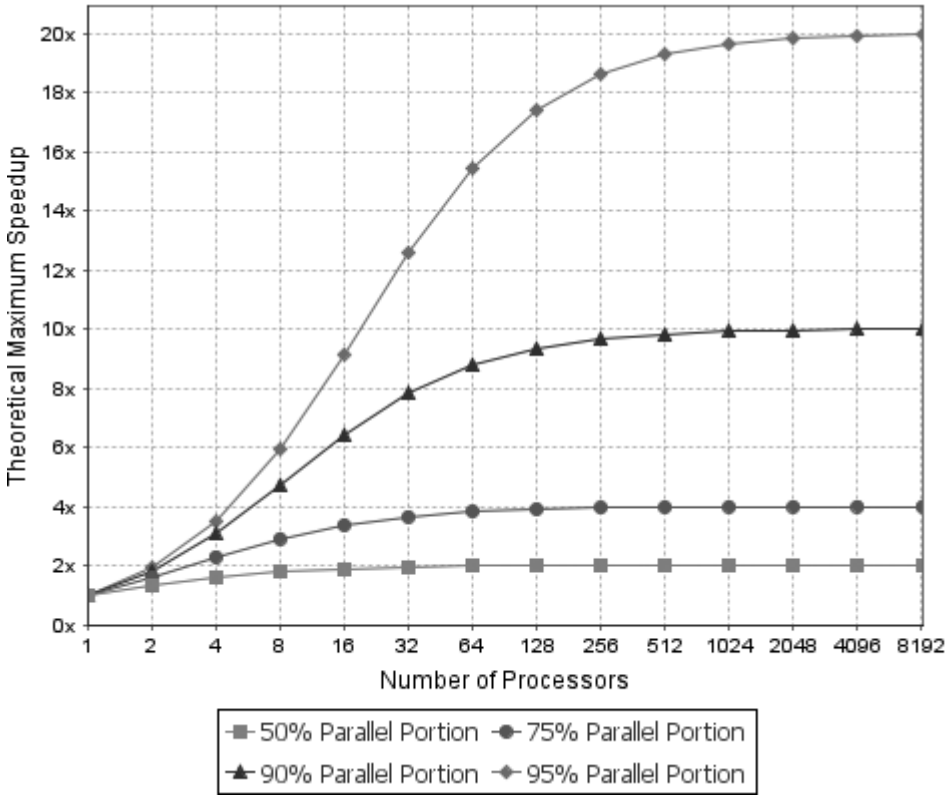

Figure 7.1: Amdahl's Law

Gustafson's law

However in 1988, John L. Gustafson did a reevaluation of Amdahl's Law[18]. He stated that by increasing the data size, it would also increase the parallel work. This results in that the speedup obtained through parallelization increases.

Below is the equation of Gustafson's Law. Here N is the number of cores available, α is the non-parallelizable part of the program.

$$speedup(N) = N - \alpha(N - 1)$$

But it does not necessarily mean that any of them are wrong, but rather that each law is in fact different perspective on the same problem. Amdahl sees

the data sizes as fixed and Gustafson sees the relation as a function of data size.

Of course these two laws are just theoretical viewpoints of parallel problems where no other factors determine the result. But in the real world we have all kind of influences; interruption, overhead, technical drawbacks just to name a few. So we should use the laws as they are meant to, as theoretical viewpoints.

7.2 Test Results Discussion

As we discussed both Amdahl's and Gustafson's laws are what we could achieve in theory and that the overhead is not a part of the equations they produced. With the measurements that we did in Chapter 6 – Threads and Overhead, we see that overhead could have a big part in the extra computation time. So from a developer viewpoint we should always have overhead in our mind when programming parallel implementations.

An interesting point would be to include the overhead in the equations, for example by adding a typical overhead cost from using a specific mechanism M and multiply it with the number of cores:

$$speedup \leq \frac{1}{F + \frac{(1-F)}{N} + (N * M)}$$

The drawback of this is of course that the overhead is not really static and may vary between different machine architecture and operating system. By using the exact same system in the previous overhead testing we saw that with the overhead we gained from using Linux was usually less costly than on Windows.

Chapter 8

Conclusions

The main goal for this thesis was to implement two distinct Sorting Algorithms, one with non-shared data and one with shared data. Each of the algorithms had three different implementations to test out various tools given by `java.util.concurrent`, creating parallel versions of the algorithms. By using different tools, measuring the time it takes to sort an array with them and comparing these results to a regular sequential implementation, we could see how well each implementation performed and learn something from the results.

As overhead is one of the significant factors when dealing with parallelism, we did extensive tests on the various mechanism used for the different implementations. This would help us draw conclusions on how the parallel algorithms respond to overhead and giving us a better insight in when to go from sequential to parallel.

Quicksort

Performance wise with Quicksort the ExecutorService implementation was the fastest overall on both the systems, the one of our own (*quad-core*) and at the University (*4x octa-core*). Even though we measured the Fork/Join-framework to use less overhead when compared with ExecutorService, and of course the work-stealing which cause the worker threads to not idle if another worker have remaining tasks in their queue. So maybe with other algorithms that are emitting much more tasks and where the tasks are more unfair weighted in terms of computation time, we would see that Fork/Join being superior.

Overall we got good parallelization with the quad-core CPU, gradually increasing the performance as the computation time surpass the extra overhead that occur with generating the parallel framework. We could have done the array length check before creating the parallel framework to avoid the overhead when dealing with arrays with less than 50 000 elements.

On the 4x octa-core CPUs we saw that while we got the gradually increasing performance, it was only on par with the quad-core system and not what we really hoped for. Measurements of overhead for the different mechanisms also conclude that we should not gain so much additional overhead to answer for not getting the expected performance. We suspect that the cache, memory and bandwidth limitation when having 4 CPUs working in parallel are the reason for this.

Radixsort

Sequential Radixsort is proven to be very fast compared to the built-in Quicksort based `Arrays.sort()`. The 2-digit LSD-Radixsort is over 6-times faster on average with array-sizes ranging from 1000 to 100 million elements.

On the parallel implementation we gradually increased the efficiency of the algorithm, by eliminating most of the synchronization and sequential parts we ended up with a very fast algorithm. With a quad-core CPU we reached a peak with sorting 300 000 elements 18 times faster than `Arrays.sort()`, and compared with sequential version 2.5 times. That gives us an increase of around 62.5% (31.25% by including Hyper-Threads) more performance per core. This is of course not the case if we decided to use a CPU with more (or less) cores, because there are so many other factors that matters.

This is why we tested with the system at the University. Here the outcome was that we saw a huge drop in performance when spawning 64 worker threads, and that it was not before sorting 5 million or above that this system became more efficient. The problem here is that as soon as parallel computation begins, we create and start all 64 worker threads; suffering a huge cost in additional overhead compared to how much data each of the worker handles. We should have added a limitation on how many workers to create based on how much data each will sort (array size).

Last words

With the research and implementation that has been done for this thesis, the following general conclusions have been drawn when constructing parallel versions of already existing sequential algorithms using the Java concurrency library:

- *Avoid old concurrent tools* - At least for Java, the only tools available before version 5 were a couple of low-level concurrent primitives. These are all difficult to use correctly which potentially could induce dead-lock, starvation or other safety issues. The more modern concurrency tools found in the standard `java.util.concurrent` package, help create a more easy to use and much safer environment when working with concurrent programming. The new version Java 7 brought some interesting and valuable tools to the table. While the Fork/Join-framework performed a bit slower than the older ExecutorService on the tests in this thesis, the work-stealing aspect for Fork/Join should become very valuable with more and wider range of task sizes. With the upcoming Java 8 introduces `LongAdder`, `DoubleAdder` and more.

- *Use specific tools for the job* - Different problems may require different tools to achieve the best scalability and performance. Certain problems like divide-and-conquer may find the Fork/Join-framework, which was specific constructed for these problems, to be the best tool. Or find out if one should use the CountDownLatch or CyclicBarrier as synchronization for the problem.

- *Know when to parallelize* - Parallelize CPU intense parts of a program, but also takes notice of the computation time for these parts. If it takes more time to set-up, spilt into tasks and initiates the threads (**Overhead**) then it takes to just compute the part sequential, the choice is obvious.

- *Try to reduce or remove synchronization* - Shared data can be a problem with parallelization, as the number of threads that needs access to the same data increases so does the number of synchronizations. This will cause a huge effect on the performance as seen with the first parallel implementations of Radixsort, where we on large arrays were up in millions of synchronizations. But by making the shared data local in the last implementation we went down to only 10 synchronizations.

- *Educate for concurrency* - Developing for traditional sequential software varies a lot from implementing concurrent software. Developers need to be aware of the tools available for concurrency on the programming language they use, and get to know various design patterns suitable for concurrency. The ability to write concurrent software that scales well on multi-core architectures, but also be caution on the different safety issues with parallel computations.

8.1 Future Work

As working through this thesis there have come up a lot of things we wish we had time to do, or that we would have approached differently than we did.

Even though we were not really after creating the "perfect" algorithm we would still have liked to do profiling on the test runs, to gain a much better overview of memory and CPU usage, and help gain an overview of the most CPU / memory intensive parts.

It would be interesting to test the ExecutorService- and Fork/Join-framework on other algorithms. To find out if it would conclude the same results we got with Quicksort. Because of our conclusion we have reason to believe that Fork/Join should perform better overall with the less overhead and work-stealing.

Test out more of different mechanisms that the concurrency packages have to offer. Especially the Phaser would be interesting to look into as we only did some small tests with it. But would like to try it out more to see if it gives any advantages, excluding the dynamical functionality it provides, compared to CyclicBarrier. Also look into the new mechanisms in Java 8; AtomicDouble, ConcurrentHashMap, LongAdder, DoubleAdder etc...

The testing done at the University should have been more extensive. When doing test runs with so many cores we should have tried out different settings with fewer cores, to get a better view of the connection between numbers of cores versus the increased performance. We suspected the Cache / Memory / Bandwidth limitation on this system to cause problems, and doing additional measurements and testing on these areas would hopefully give some answers. In addition test out other systems and architectures.

References

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[2] Intel Corp. Desktop Products Group. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–17, February 2002.

[3] Brian Goetz. *Java Concurrency in Practice*. 2006.

[4] Doug Lea. Concurrency JSR-166 interest site. http://g.oswego.edu/dl/concurrency-interest/.

[5] Clay Breshears. *The Art of Concurrency*. 2009.

[6] Java SE 7 API, 2012. http://docs.oracle.com/javase/7/docs/api/.

[7] jsr166e, 2012. http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/.

[8] Doug Lea. A java fork/join framework. *Proceedings of the ACM 2000 Java Grande Conference*, 2000.

[9] David Kanter. Inside nehalem: Intel's future processor and system, 2008. http://realworldtech.com/page.cfm?ArticleID=RWT040208182719.

[10] Arne Maus and Stein Gjessing. A model for the effect of caching on algorithmic efficiency in radix based sortingsort. *The Second International Conference on Software Engineering Advances*, 2007.

[11] Javahotspot FAQ, 2012. http://www.oracle.com/technetwork/java/hotspotfaq-138619.html.

[12] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Third edition, 2009.

[14] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, fourth edition, 2011.

[15] Vladimir Yaroslavskiy, Jon Bentley, and Josh Bloch. Dualpivotquicksort.java, 2011. http://www.docjar.com/html/api/java/util/DualPivotQuicksort.java.html.

[16] Vladimir Yaroslavskiy. Dual-pivot quicksort algorithm, 2009. http://gdtoolbox.com/DualPivotQuicksort.pdf.

[17] Arne Maus. Paradigms for removing unnecessary synchronization in parallel algorithms. *submitted to NIK'2012 (Norwegian Informatics Conference, Bodø)*, 2012.

[18] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[19] Timothy G. Mattson, Beverly A. Sanders, and Berna L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[20] Robert Sedgewick. *Algorithms in Java, Parts 1-4*. Addison-Wesley Professional, third edition, 2002.

[21] Robert Sedgewick. *Algorithms in Java, Part 5*. Addison-Wesley Professional, third edition, 2003.

[22] Mark Allen Weiss. *Data Structures & Problem Solving Using Java*. Third edition, 2006.

Appendix

Source Code

The source code used for producing all the test results seen in this thesis is available at: https://github.com/Busterud/master-thesis/

The code can be navigated and viewed directly online, with the ability to download the whole repository as a zip-file. The test code available is categorized below and with a short description:

Sorting Tests:

- **Quicksort** (`QuicksortTesting.java`): Sequential, Threaded, ExecutorService and Fork/Join implementations tested with Test Data Structure seen in Section 3.1.
- **LSD-Radixsort** (`LSDRadixsortTesting.java`): Sequential and three Parallel implementations using CyclicBarrier and Atomicity, tested with Test Data Structure seen in Section 3.1.
- **Graph Creation** (`GraphCreationSort.java`): Create graph from the data produced by QuicksortTesting.java and LSDRadixsortTesting.java.

Overhead Tests:

- **Overhead** (`OverheadTesting.java`): Methods, Classes, Threads, ExecutorService and Fork/Join overhead testing.
- **OverheadAtomic** (`OverheadAtomicTesting.java`): int, int[], AtomicInteger, AtomicIntegerArray overhead testing.

- **Batch-file for Overhead** (`OverheadTesting.bat`): Test the Overhead and/or OverheadAtomic many times in a Windows environment.
- **Bash-file for Overhead** (`OverheadTesting.sh`): Test the Overhead and/or OverheadAtomic many times in a Linux environment.
- **Graph Creation** (`GraphCreationOverhead.java`): Create graph from the data produced by OverheadTesting.java and OverheadAtomicTesting.java.

Prerequisites

The following tools need to be installed in order to use the test codes:

- Java; Version 1.7 or later for Quicksort and Overhead testing, version 1.6 or later for the rest (http://www.java.com/en/download/)
- JFreeChart; for Graph Creation (http://www.jfree.org/)
- Jsr166y.jar; when using Java 1.6 for Quicksort and Overhead testing (http://g.oswego.edu/dl/concurrency-interest/)