High Performance Programming - Individual Assignment

Optimizing the quicksort algorithm

Johan Rideg

2020-03-24

1 Preface

For a long time I have wanting to dive deeper into sorting algorithms, especially quicksort and now I have a great excuse to indulge myself. Getting an intuition for why quicksort is so fast and why it even works was not immediately apparent but is something that grew over time as I got deeper into the development.

2 Introduction

Quicksort is a recursive *divide and conquer* algorithm i.e. it works by recursively divide the array into smaller subsets and sort these subsets separately. We will see in detail how the divisions are done which are the very essence of quicksort and why it performs so efficiently. Quicksort is today recognized as one of the fastest sorting algorithms, outperforming other $\mathcal{O}(n \log n)$ algorithms like mergesort if implemented well. It was first developed by Tony Hoare in 1959, and is now used in many standard libraries across multiple programming languages.

Quicksort works by selecting a pivot. The pivot can be any element of the array and there exists many strategies to choose the pivot "in a good way". When a pivot is determined, move all elements less than the pivot to the left of the pivot and elements greater than the pivot to the right of the pivot. This is called *partitioning* and there exists multiple partitioning schemes. When an array is partitioned, it can be split into two arrays where partitioning is performed until we either end up with an array with one element or choose to do something smarter...

All test were performed on a UNIX system using `Intel Xeon E5520 @2.27 GHz` with the compiler `gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0`. The code was compiled using `gcc -g -Wall -O3 -c Quicksort.c` unless stated otherwise.

3 Methology

3.1 Preparatory Work

We must first have some methods for creating, copying and printing arrays which we wish to not be sorted by default. The `generateArray` function allocates memory for an array of size n dynamically and assigns each element an integer, specifically the elements index. This results in an array on the form:

$$\{0, 1, 2, ..., n-1\}$$

The array clearly need to be shuffled to be interesting for our purposes. This is done by calling `srand` to set a seed for `rand` function. We then loop over each element, starting from the top and switch its place with a randomly selected element from the left. Decrement until the second element is reached.

The backbone of a sorting algorithm is the `swap` function which takes two numbers and swaps them. We will see this function in many places of the code. In total, the preparatory work resulted in eight simple functions which made the development of quicksort very convenient.

```c
// Swaps two integers.
void swap(int* a, int* b)

// Shuffles an array using rand().
// arr: array.
// n:   array size.
void shuffle(int* arr, const int n)

// Prints an array to teminal in a column.
void printArrVert(int* arr, const int n)

// Prints an array to terminal horizontally.
void printArr(int* arr, const int n)

// Prints a subset of an arr from lo to hi inclusive.
void printSubArr(int* arr, const int lo, const int hi)

// Generates an ordered array [0, 1, 2, 3, ..., n-1].
int* generateArr(const int n)

// Makes a copy of an array.
int* copyArr(int* oldArr, const int n)

// Verifies that an array is sorted.
void verifySorted(int* arr, const int n)
```

3.2 Developing Quicksort

Quicksort is comprised of two main parts: The recursive function and the partition function, which the latter is of particular interest. Being a quicksort novice, I found Lomuto's partitioning scheme easy to both comprehend and to implement. Lomuto's partitioning uses a pivot which is chosen to be the rightmost element of the array. A counter i is used to keep track of the leftmost element greater or equal to the pivot. We then loop over the array from left to right checking if current element is less than the pivot. If it is less than the pivot, swap it with element i and increment i. When the whole array has been iterated over swap the pivot with the leftmost element greater or equal to the pivot. This results in an array where all elements less than the pivot is placed in the left end of the array, followed by the pivot and then all elements greater or equal to the pivot is placed to the right of the pivot. Lastly the index of the pivot is returned and a complete partition has been performed. The result of a partitioning is the following array:

$$\{e_1 < p, \quad e_2 < p, \quad ... \quad e_{i-1} < p, \quad \textbf{pivot}, \quad e_{i+1} >= p, \quad e_{i+2} >= p, \quad ... \quad e_{n-1} >= p\}$$

An interesting characteristic of Lomuto's partitioning is that when a partition has been performed, the pivot is at its final index and does not need to be passed on into the recursion.

```
// Lomuto partition scheme: Choose last element as pivot.
int partitionLomuto(int* arr, const int lo, const int hi)
{
    const int pivot = arr[hi];
    int i = lo;

    for (int j=lo; j<=hi; j++)
    {
        if (arr[j] < pivot)
        {
            swap(&arr[i], &arr[j]);
            i += 1;
        }
    }

    swap(&arr[i], &arr[hi]);
    return i;
}
```

Having completed the implementation of the `partitionLomuto` function, the next step is to implement the recursive function `quicksortLomuto`.

```
// Quicksort recursion using Lomuto's partitioning scheme.
void quicksortLomuto(int* arr, const int lo, const int hi)
{
    if (lo < hi)
    {
        int p = partitionLomuto(arr, lo, hi);
        quicksortLomuto(arr, lo, p-1);
        quicksortLomuto(arr, p+1, hi);
    }
}
```

We now have a very basic quicksort algorithm on which to apply optimization strategies.

3.3 Optimizing Quicksort

3.3.1 Choosing an Effective Partitioning Scheme

When tasked with optimizing a well known algorithm such as quicksort, it becomes clear that there exists a smorgasbord of optimization choices available. Another partitioning scheme, namely Hoare's, seemed like a good place to start, promising three times less `swap` calls than Lomuto's on avarage.

We begin by choosing a pivot, for example the middle element. Hoare uses two indices initialized at the ends of the array and moves them towards eachother. This is done by letting the leftmost index, i, move right until it finds an element greater than or equal to the pivot. It then waits for the rightmost index, j, to move left until it encounters an element less than or equal to the pivot. When both indices have found the *inversion*, swap the elements and proceed until the two indices meet. At that point the algorithm returns the meeting point where the pivot is now located.

```c
// Basic hoare partitioning. No median of three strategy is applied.
int partitionBasicHoare(int* arr, const int lo, const int hi)
{
    // Choose middle element as pivot.
    const int pivot = arr[(hi+lo)/2];

    // Initialize indices at the ends of the array.
    int i = lo - 1;
    int j = hi + 1;

    // Repeat until indices meet.
    while (1)
    {
        // Increment i until arr[i] exceeds pivot.
        do
            i++;
        while (arr[i] < pivot);

        // Decrement j until pivot exceeds arr[j].
        do
            j--;
        while (arr[j] > pivot);

        // If indices meet, return j.
        if (i >= j)
            return j;

        // Inversion has been found. Swap current elements.
        swap(&arr[i], &arr[j]);
    }
}
```

A new quicksort version is written. Note that when we call quicksortHoare on the partition with values less than the pivot, we also include the pivot in that partition. This is because unlike Lomuto's partitioning scheme, the pivot may not be in its final position.

```c
// Quicksort recursion using Hoares's partitioning scheme.
void quicksortBasicHoare(int* arr, const int lo, const int hi)
{
    if (lo < hi)
    {
        int p = partitionBasicHoare(arr, lo, hi);
        quicksortBasicHoare(arr, lo, p);
        quicksortBasicHoare(arr, p+1, hi);
    }
}
```

Compiling with only `funroll-loops` as optimization flag and comparing the performance against Lomuto's yield a speedup of about 1.3 times which is lower than expected. Even stranger, when compiling with the `-O3` optimization flag, both methods perform roughly the same. The test was performed five times on identical arrays whose size ranged from 2,000 to 10,000,000.

3.3.2 Quicksort-Insertionsort Hybrid

One drawback quicksort is that it behaves poorly on "small" (<50) arrays compared to simple $\mathcal{O}(N^2)$ algorithms such as the `insertionsort`. One approach is that when operating on a large array and the recursion reaches a partition of a small size, let `insertionsort` perform the sorting instead and back out of the current recursion branch. This is what was hinted at the end of the introduction. Let us create an `insertionsort` function.

```c
// Fast insertion sorting algorithm. Used for small partitions.
// lo: lowest index of the array;
// hi: highest index of the array;
void insertionSort(int* arr, const int lo, const int hi)
{
    int tmp;
    int j;
    for (int i=lo+1; i<hi+1; i++)
    {
        tmp = arr[i];
        j = i-1;

        while(j>=0 && arr[j]>tmp)
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = tmp;
    }
}
```

It is important that the `&&`-operator must use short-circuit evaluation which it does in C. If not when we try to evaluate `arr[j]>tmp` with a `j` smaller than `0`, we are likely to get an `arrayOutOfBounds` error. With a working `insertionsort` we can modify our `quicksort` algorithm.

```c
// Quicksort recursion using Hoares's partitioning scheme
// utilizing insertion sort for small partitions.
void quicksortHoare(int* arr, const int lo, const int hi)
{

    if (lo < hi)
    {
        // If array is small, use insertion sort.
        if (hi-lo < 50)
        {
            insertionSort(arr, lo, hi);
            return;
        }
        int p = partitionHoare(arr, lo, hi);
        quicksortHoare(arr, lo, p);
        quicksortHoare(arr, p+1, hi);
    }
}
```

The optimum size for which to switch to insertion sort depends on a number of factors, some of which are the machine, how the code is implemented and how the code is compiled. Finding the optimum size is done empirically by simply iterating through some trial values and check where the execution time dips. A file is written each run with timings. For this particular environment, this indicated that choosing 50 as a cutoff provides the best performance increase, providing a speedup of 1.2 times.

3.3.3 Pivot Choice

Naively choosing a pivot, be it the middle element or at random creates inconsistencies in the size of the left and right side of a partitioned array. For example if we have a shuffled array of ten elements as shown below and the pivot is chosen to be the middle element, **7**, we would end up with one significantly larger partition and one smaller as shown below.

$$\{3,\quad 8,\quad 0,\quad 4,\quad \mathbf{7},\quad 9,\quad 1,\quad 5,\quad 2,\quad 6\}$$

$$\{3,\quad 8,\quad 0,\quad 4,\quad 1,\quad 5,\quad 2,\quad 6\quad \mathbf{7}\} \qquad\qquad \{9\quad 8\}$$

Ideally we would like to choose the median element. Calculating the median is however an expensive process and is not something we can realistically do and expect better performance. By creating two equal sized partitions, the recursion should reach the same depth for all partitions, and the time taking partitioning an array on depth d would be cn/d resulting in the best possible performance.

We can employ the *median of three* method of choosing a pivot. A non-deterministic method would pick three elements at random and choose their median value as a pivot. Choosing elements at random also decreases the risks of having bad performance on some specific arrays. This way of choosing a pivot was implemented but chosen not to be used due to `srand` and `rand` functions being too slow. One approach that was not attempted was to build a pseudorandom number generator specific for this problem. Eventually, a deterministic median of three pivot choice was implemented. This worked considerably faster than the non-deterministic one and chooses the pivot by finding the median of the leftmost, middle and rightmost element of the array.

```c
// Deterministic median of three.
// Finds the median of an array by sampling at three points.
int medianOfThree(int* arr, const int lo, const int hi)
{
    int left = arr[lo];
    int mid = arr[(hi+lo)/2];
    int right = arr[hi];

    if (right<left)
        swap(&right, &left);
    if (mid<left)
        swap(&mid, &left);
    if (right<mid)
        swap(&right, &mid);

    return mid;
}
```

3.3.4 Parallelizing Quicksort with Pthreads.

At the same recursion level, each process of sorting a partition is independent of each other, which means that they may be performed in parallel. One approach is to partition up until a maximum recursion level and store each partition in a global array of `threadData_t` structures referencing a partition, a thread, and the thread's ID.

```
// Preprocess by taking more samples to lower the risks of unbalanced load between threads.
// the global args array holds a threadData_t struct for each thread.
const int maxLevel = 4;
int threadCounter = 0;
threadData_t args[16];
void preprocess(int* arr, const int lo, const int hi, const int level)
{
    if (level < maxLevel)
    {
        int p = partitionHoareBalanced(arr, lo, hi);
        preprocess(arr, lo, p, level+1);
        preprocess(arr, p+1, hi, level+1);
    }
    else
    {
        threadData_t data;
        data.arr = arr;
        data.lo = lo;
        data.hi = hi;
        data.threadID = threadCounter;

        args[threadCounter] = data;
        threadCounter++;
    }
}
```

This solution is simple and has its risks. The largest of the partitions at maximum recursion level will restrict minimum execution time. Choosing a maximum recursion level of 4 will result in sixteen partitions to be sorted by each of the 16 threads. Therefore, it is especially important that the pivots chosen are close to the true median to ensure a good load balance. Without a good approximate median, quicksort will produce inconsistent results using this parallelization scheme. At one run it may provide a near optimal speedup while at another run provide close to no speedup or even degrade performance slightly due to the overhead when creating and joining threads.

This is solved by the `partitionHoareBalanced` function. It works the same way as the regular Hoare partitioning with a slight difference. It samples the array at larger number of equidistant points and chooses the median of those values as a pivot. This reduces the risks that the partition sizes deviate significantly from eachother. The extra computation required is well worth it since this will only be done $2^{maxLevel}$ times. This proved to drastically help performance consistency without a noticeably impact on the best execution time. The parallelized quicksort first resets the `threadData_t` array and proceeds to create the partitions at maximum recursion level. When all partitions have been created, the functions spawns a thread to sort each partition sequentially. Finally, all threads are joined and the global variable `threadCounter` is reset.

```c
// Creates threads and assigns their workload.
void quicksortParall(int* arr, const int lo, const int hi)
{
    // Reset argument array.
    for (int i=0; i<16; i++)
    {
        args[i].thread = 0;
        args[i].threadID = 0;
        args[i].arr = NULL;
        args[i].lo = 0;
        args[i].hi = 0;
    }

    // Partition array with low risk of unbalanced load.
    preprocess(arr, lo, hi, 0);

    // For all leaf partitions let a thread perform sequential quicksort.
    for(int i=0; i<threadCounter; i++)
        pthread_create(&args[i].thread, NULL, parallHelper, (void*)&args[i]);

    // Join threads and reset thread counter.
    void* status;
    for(int i=0; i<threadCounter; i++)
        pthread_join(args[i].thread, &status);

    threadCounter = 0;

}
```

The `parallHelper` function takes a `void*` arguments, casts to a `threadData_t` structure and calls the sequential algorithm.

```c
// Intermediate step to call sequential quicksort with a pthread.
void* parallHelper(void* args)
{
    threadData_t* data = (threadData_t*)args;
    int* arr = (*data).arr;
    int lo = (*data).lo;
    int hi = (*data).hi;
    quicksortHoare(arr, lo, hi);
    pthread_exit(NULL);
}
```

4 Performance analysis

4.1 General Optimization

The sorting works in-place meaning that the array which is passed will be sorted instead of making a copy and sorting that instead. While it may or may not have been easier to implement an algorithm which creates a copy it would either way have been inefficient in regards to memory. The performance is expected to be worse aswell because the pointer needs to move inbetween a memory span twice the size of the in-line version.

The choice to restrict to integer elements was made for two main reasons. First, integer comparisons and arithmetic are relatively quick compared to other datatypes and still provide a large span of possible numbers (-32,768 to 32,767). Secondly, a `int` occupies less memory than its long or double counterparts, meaning a shorter span of memory is allocated for the array. This leads to shorter pointer moves and thus, better performance.

Declaring constants as `const` may help the compiler to make some optimizations at assembly level. The `const` declaration by itself should not yield any speedup, however, it is good practice to help prevent the developer introducing errors and fascilitates readability.

Setting the attribute `inline` to functions, where applicable, is a great method for a noticeable speedup for this specific problem since most of the code is segmented into small, easaly-read, subroutines, this is essential to achieve a good performance. The performance increase comes from the removal of function overhead by essentially replacing a call to an `inline` function by its code. Care must be taken as this may vary between different codes. `inlining` when the executable is large may worsen the performance because the system spends more time fetching the next code chunk. This is done automatically if the `-O3` optimization flag is passed.

No manual loop unrolling is performed but is done by the compiler by passing the `funroll-loops` flag. This increases the size of the executable. The performance increase comes from the elimination of the loop control instruction at assembly level. This is done automatically if the `-O3` optimization flag is passed.

4.2 Parallel performance

Comparing the parallel performance increase is done by generating an array with a size n and making a copy. The optimized sequential quicksort is timed sorting the original and the optimized parallel quicksort is timed sorting the copy using 16 threads. This is repeated as n is incremented from 2,000 to 100,000,000 with steps of 2000. When all timings have been performed, a result file is written. This is repeated five times and the lowest sorting times for both the sequential and parallel quicksort are chosen at each n to compare against both methods. Looking at figure 1, it becomes apparent that at small n, the sequential performs better than the parallelized version due to decreased overhead when spawning threads. At higher n, the parallel speedup is consistently around 4 times.

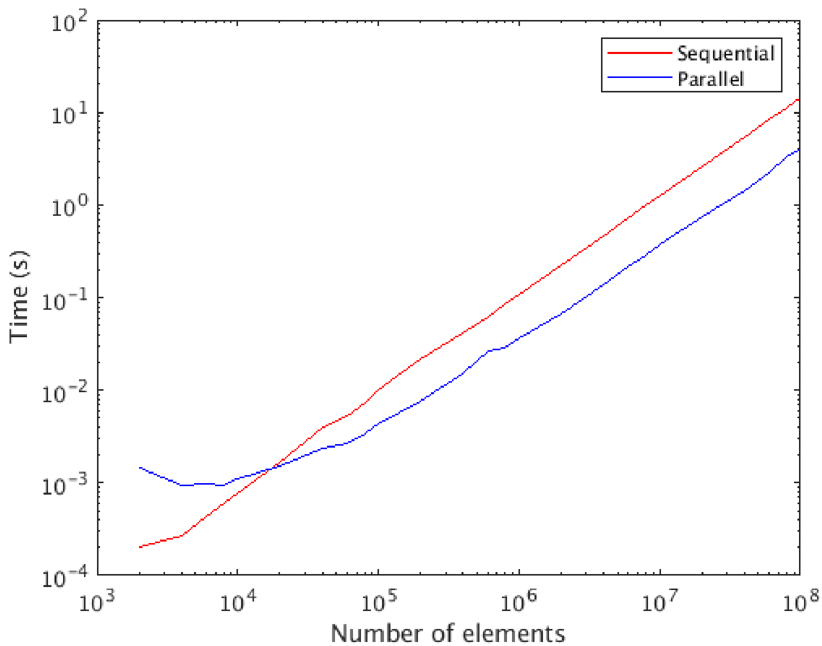

Figure 1: *LogLog plot of the best recorded execution times for the optimized sequential (red) quicksort and the parallelized (blue) version. Best recordings was sampled from a set of five executions.*

4.3 Final performance

Comparing the first version, presented in 3.2, to the final parallelized and optimized quicksort is interesting to see how much of a performance increase was gained in total. The test was performed similarly to when comparing the optimized sequential and parallel algorithms. When compiling with no optimization flags, thee speedup is slightly below 5 times for large arrays. With `-O3` optimization flag, the speedup is closer to 4 times. This is reasonable as the optimized quicksort algorithm is not expected to do as many function calls, such as `swap`, and the benefit of removed function overhead is not as impactful for the optimized quicksort than it is for the basic Lomuto algorithm.

A Appendix - Complete code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <pthread.h>

// For timings.
double get_wall_seconds()
{
  struct timespec tspec;
  clock_gettime(CLOCK_REALTIME, &tspec);

  double seconds = tspec.tv_sec + (double)tspec.tv_nsec/(double)1000000000;
  return seconds;
}

// Swaps two integers.
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Shuffles an array.
// arr: array.
// n:   array size.
void shuffle(int* arr, const int n)
{
    // Set seed for rand().
    srand(time(NULL));

    // Shuffle array.
    for (int i = n - 1; i > 0; i--)
    {
        int j = rand() % (i + 1);
        swap(&arr[i], &arr[j]);
    }
}

// Prints an array to teminal in a column.
void printArrVert(int* arr, const int n)
{
    for (int i = 0; i < n; i++)
    {
        printf("idx %i: %i\n", i, arr[i]);
    }
}

// Prints an array to terminal horizontally.
void printArr(int* arr, const int n)
{
    printf("[");
    for (int i = 0; i < n-1; i++)
    {
        printf("%i, ", arr[i]);
    }
    printf("%i]\n", arr[n-1]);
}

// Prints a subset of an arr from lo to hi inclusive.
```

```c
void printSubArr(int* arr, const int lo, const int hi)
{
    printf("[");
    for (int i = lo; i < hi; i++)
    {
        printf("%i, ", arr[i]);
    }
    printf("%i]\n", arr[hi]);
}

// Generates an ordered array [0, 1, 2, 3, ..., n-1].
int* generateArr(const int n)
{
    int* arr = (int*)malloc(sizeof(int) * n);
    for (int i = 0; i < n; i++)
    {
        arr[i] = i;
    }

    return arr;
}

// Makes a copy of an array.
int* copyArr(int* oldArr, const int n)
{
    int* newArr = (int*)malloc(sizeof(int)*n);
    for (int i=0; i<n; i++)
    {
        newArr[i] = oldArr[i];
    }
    return newArr;
}

// Verifies that an array is sorted.
void verifySorted(int* arr, const int n)
{
    for (int i=1; i<n; i++)
    {
        if (arr[i]<arr[i-1])
        {
            printf("*** Array is NOT sorted. ***\n");
            return;
        }
    }
    printf("*** Array is sorted. ***\n");
}



// === QUICKSORT FUNCTIONS ===

// Fast insertion sorting algorithm. Used for small partitions.
// lo: lowest index of the array;
// hi: highest index of the array;
void insertionSort(int* arr, const int lo, const int hi)
{
    int tmp;
    int j;
    for (int i=lo+1; i<hi+1; i++)
    {
        tmp = arr[i];
        j = i-1;

        while(j>=0 && arr[j]>tmp)
        {
```

```
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = tmp;
    }
}

// Lomuto partition scheme: Choose last element as pivot.
int partitionLomuto(int* arr, const int lo, const int hi)
{
    const int pivot = arr[hi];
    int i = lo;

    for (int j=lo; j<=hi; j++)
    {
        if (arr[j] < pivot)
        {
            swap(&arr[i], &arr[j]);
            i += 1;
        }
    }

    swap(&arr[i], &arr[hi]);
    return i;
}

// Deterministic median of n elements. array is a small array.
// Used when partitioning in the first levels of recursion
// when running multiple threads. Lowers risk of bad load balance.
int medianOfN(int* arr, const int lo, const int hi)
{
    insertionSort(arr, lo, hi);
    return arr[(hi+lo)/2];
}

// Deterministic median of three.
// Finds the median of an array by sampling at three points.
int medianOfThree(int* arr, const int lo, const int hi)
{
    int left = arr[lo];
    int mid = arr[(hi+lo)/2];
    int right = arr[hi];

    if (right<left)
        swap(&right, &left);
    if (mid<left)
        swap(&mid, &left);
    if (right<mid)
        swap(&right, &mid);

    return mid;
}


// Non-deterministic median of three. Returns median of a subset of an array.
// Note: RAND_MAX = 2147483647 on ITC1515 system so we should be safe.
// arr: array for which the median value is to be found in selected interval.
// lo:  lower interval bound.
// hi:  upper interval bound.
int medianOfThreeRand(int* arr, const int lo, const int hi)
{
    srand(time(NULL));
    int vals[3];
    int index;

```

```c
    index = (rand()%(hi-lo+1))+lo;
    vals[0] = arr[index];
    index = (rand()%(hi-lo+1))+lo;
    vals[1] = arr[index];
    index = (rand()%(hi-lo+1))+lo;
    vals[2] = arr[index];

    if (vals[2]<vals[0])
        swap(&vals[2], &vals[0]);
    if (vals[1]<vals[0])
        swap(&vals[1], &vals[0]);
    if (vals[2]<vals[1])
        swap(&vals[2], &vals[1]);

    return vals[1];
}

// Basic hoare partitioning. No median of three strategy is applied.
int partitionBasicHoare(int* arr, const int lo, const int hi)
{
    // Choose middle element as pivot.
    const int pivot = arr[(hi+lo)/2];

    // Initialize indices at the ends of the array.
    int i = lo - 1;
    int j = hi + 1;

    // Repeat until indices meet.
    while (1)
    {
        // Increment i until arr[i] exceeds pivot.
        do
            i++;
        while (arr[i] < pivot);

        // Decrement j until pivot exceeds arr[j].
        do
            j--;
        while (arr[j] > pivot);

        // If indices meet, return j.
        if (i >= j)
            return j;

        // Inversion has been found. Swap current elements.
        swap(&arr[i], &arr[j]);
    }
}

// Hoare partition scheme: Choose middle element as pivot.
int partitionHoare(int* arr, const int lo, const int hi)
{
    // Choose pivot as the median of three elements.
    const int pivot = medianOfThree(arr, lo, hi);

    // Initialize indices at the ends of the array.
    int i = lo - 1;
    int j = hi + 1;

    while (1)
    {
        // Increment i until arr[i] exceeds pivot.
        do
            i++;
        while (arr[i] < pivot);
```

```

        // Decrement j until pivot exceeds arr[j].
        do
            j--;
        while (arr[j] > pivot);

        // If indices meet, return j.
        if (i >= j)
            return j;

        // Inversion has been found. Swap current elements.
        swap(&arr[i], &arr[j]);
    }

}

// Hoare's partitioning scheme: sample the array at regular intervals
// and choose their median as pivot
int partitionHoareBalanced(int* arr, const int lo, const int hi)
{
    // Choose median of any nSamples number of elements.
    const int nSamples = 50;
    const int stepsize = (hi-lo)/(nSamples-1);
    int* pivotArr = (int*)malloc(sizeof(int)*nSamples);
    pivotArr[0] = arr[lo];
    for (int i=1; i<nSamples-1; i++)
    {
        // Sample array at regular intervals.
        pivotArr[i] = arr[lo+i*stepsize];
    }
    pivotArr[nSamples-1] = arr[hi];
    const int pivot = medianOfN(pivotArr, 0, nSamples-1);
    free(pivotArr);

    // Initialize indices at the ends of the array.
    int i = lo - 1;
    int j = hi + 1;

    while (1)
    {
        // Increment i until arr[i] exceeds pivot.
        do
            i++;
        while (arr[i] < pivot);

        // Decrement j until pivot exceeds arr[j].
        do
            j--;
        while (arr[j] > pivot);

        // If indices meet, return j.
        if (i >= j)
            return j;

        // Inversion has been found. Swap current elements.
        swap(&arr[i], &arr[j]);
    }

}

// Quicksort recursion using Lomuto's partitioning scheme.
void quicksortLomuto(int* arr, const int lo, const int hi)
{
    if (lo < hi)
    {
```

```
        int p = partitionLomuto(arr, lo, hi);
        quicksortLomuto(arr, lo, p-1);
        quicksortLomuto(arr, p+1, hi);
    }
}

// Quicksort recursion using Hoares's partitioning scheme.
void quicksortBasicHoare(int* arr, const int lo, const int hi)
{

    if (lo < hi)
    {
        int p = partitionBasicHoare(arr, lo, hi);
        quicksortBasicHoare(arr, lo, p);
        quicksortBasicHoare(arr, p+1, hi);
    }
}

// Quicksort recursion using Hoares's partitioning scheme with median of three
// strategy applied and utilized insertion sort for small partitions.
void quicksortHoare(int* arr, const int lo, const int hi)
{

    if (lo < hi)
    {
        // If array is small, use insertion sort.
        if (hi-lo < 50)
        {
            insertionSort(arr, lo, hi);
            return;
        }
        int p = partitionHoare(arr, lo, hi);
        quicksortHoare(arr, lo, p);
        quicksortHoare(arr, p+1, hi);
    }
}





// === PARALLELIZED QUICKSORT ===

// Struct used to call parallHelper.
typedef struct threadData
{
    pthread_t thread;
    int threadID;
    int* arr;
    int lo;
    int hi;
} threadData_t;

// Intermediate step to call sequential quicksort with a pthread.
void* parallHelper(void* args)
{
    threadData_t* data = (threadData_t*)args;
    int* arr = (*data).arr;
    int lo = (*data).lo;
    int hi = (*data).hi;
    quicksortHoare(arr, lo, hi);
    pthread_exit(NULL);
}

// Preprocess by taking more samples to lower the risks of unbalanced load between threads.
```

```
// the global args array holds a threadData_t struct for each thread to be used later by
    quicksortParall.
const int maxLevel = 4;
int threadCounter = 0;
threadData_t args[16];
void preprocess(int* arr, const int lo, const int hi, const int level)
{
    if (level < maxLevel)
    {
        int p = partitionHoareBalanced(arr, lo, hi);
        preprocess(arr, lo, p, level+1);
        preprocess(arr, p+1, hi, level+1);
    }
    else
    {
        threadData_t data;
        data.arr = arr;
        data.lo = lo;
        data.hi = hi;
        data.threadID = threadCounter;

        args[threadCounter] = data;
        threadCounter++;
    }

}

// Creates threads and assigns their workload.
void quicksortParall(int* arr, const int lo, const int hi)
{
    // Reset argument array.
    for (int i=0; i<16; i++)
    {
        args[i].thread = 0;
        args[i].threadID = 0;
        args[i].arr = NULL;
        args[i].lo = 0;
        args[i].hi = 0;
    }

    // Partition array with low risk of unbalanced load.
    preprocess(arr, lo, hi, 0);

    // For all leaf partitions let a thread perform sequential quicksort.
    for(int i=0; i<threadCounter; i++)
        pthread_create(&args[i].thread, NULL, parallHelper, (void*)&args[i]);

    // Join threads and reset thread counter.
    void* status;
    for(int i=0; i<threadCounter; i++)
        pthread_join(args[i].thread, &status);

    threadCounter = 0;

}

// Used for finding the optimal array size to switch to insertion sort.
int cutoff = 0;
void quicksortInsertTest(int* arr, const int lo, const int hi)
{
    if (lo < hi)
    {
        // If array is small, use insertion sort.
        if (hi-lo < cutoff)
        {
```

```
            insertionSort(arr, lo, hi);
            return;
        }
        int p = partitionHoare(arr, lo, hi);
        quicksortInsertTest(arr, lo, p);
        quicksortInsertTest(arr, p+1, hi);
    }
}

// Used to approximate what size of an array it is more beneficial
// to use insertion sort than to keep recursing.
void optimalInsertCutoff()
{
    const int N = 1000000;
    int* arr;
    int* arrCopy;
    double startTime;
    double insertTime;
    double noInsertTime;

    // Generate two identical arrays.
    arr = generateArr(N);
    shuffle(arr, N);
    arrCopy = copyArr(arr, N);

    // Do a measurement where no insertion sort is used
    cutoff = 0;
    startTime = get_wall_seconds();
    quicksortInsertTest(arr, 0, N-1);
    noInsertTime = get_wall_seconds() - startTime;
    arr = arrCopy;
    arrCopy = copyArr(arr, N);

    // Compare to when insertion sort is used for different cutoff values and write results to a file.
    FILE *outputFile = fopen("VaryingInsertionCutoffTimings.txt", "w");
    fprintf(outputFile, "nElements,cutoff,noInsertTime,InsertTime,speedup\n");
    for (int i=1; i<3; i++)
    {
        for (int j=1; j<6; j++)
        {
            cutoff = 2*j*10*pow(10, i-1);
            printf("Cutoff: %i\n", cutoff);

            startTime = get_wall_seconds();
            if (cutoff == 1000000)
                printf("Hello!");
            quicksortInsertTest(arr, 0, N-1);
            insertTime = get_wall_seconds() - startTime;
            verifySorted(arr, N);
            fprintf(outputFile, "%i,%i,%3.6f,%3.6f,%3.6f\n", N, cutoff, noInsertTime, insertTime, noInsertTime/insertTime);

            free(arr);
            arr = arrCopy;
            arrCopy = copyArr(arr, N);

        }
    }
}

// Used to perform timings between different versions of quicksort.
// Writes a result file.
void doTimings()
{
```

```c

    int* arr1;
    int* arr2;
    int N;

    double startTime;
    double seqTime;
    double parTime;

    FILE *outputFile = fopen("run1.txt", "w");
    fprintf(outputFile, "nElements,seqTime,parTime,speedup\n");

    // from 1000 to 100000000
    for (int i=1; i<6; i++)
    {
        for (int j=1; j<6; j++)
        {
            // Update N.
            N = (2*j)*1000*pow(10, i-1);

            // Do first timing.
            printf("*** Sorting %i elements\n", N);
            printf(">>> Sorting using sequential quicksort...\n");
            arr1 = generateArr(N);
            shuffle(arr1, N);
            arr2 = copyArr(arr1, N);
            printf("Original: ");
            verifySorted(arr1, N);
            startTime = get_wall_seconds();
            quicksortHoare(arr1, 0, N-1);
            seqTime = get_wall_seconds() - startTime;
            printf("Sorted:   ");
            verifySorted(arr1, N);

            // Do second timing.
            printf(">>> Sorting using parallelized quicksort...\n");
            printf("Original: ");
            verifySorted(arr2, N);
            startTime = get_wall_seconds();
            quicksortParall(arr2, 0, N-1);
            parTime = get_wall_seconds()-startTime;
            printf("Sorted:   ");
            verifySorted(arr2, N);

            // Print to output file.
            fprintf(outputFile, "%i,%3.6f,%3.6f,%3.6f\n", N, seqTime, parTime, seqTime/
    parTime);
            printf("Done\n\n");
            free(arr1);
            free(arr2);
            arr1 = NULL;
            arr2 = NULL;
        }
    }
    fclose(outputFile);
}

// Main driver code.
int main(int argc, char *argv[])
{
    int* arr1;
    int* arr2;

    // Testing parallel sorting.
    int N = 50000000;
```

```
    printf("\n\n>>> PARALLEL TIMINGS:\n");
    printf("    Using array size: %i.\n", N);

    arr1 = generateArr(N);
    shuffle(arr1, N);
    arr2 = copyArr(arr1, N);

    printf("arr1: ");
    verifySorted(arr1, N);
    printf("arr2: ");
    verifySorted(arr2, N);
    printf("\n");

    double startTime;
    printf("Sorting arr1 using sequential quicksort.\n");
    printf("Original arr1: ");
    verifySorted(arr1, N);
    startTime = get_wall_seconds();
    quicksortHoare(arr1, 0, N-1);
    printf("Sorting time: %3.6f seconds.\n", get_wall_seconds() - startTime);
    printf("Verifying...\n");
    printf("Sorted arr1:   ");
    verifySorted(arr1, N);
    printf("\n\n");
    free(arr1);

    printf("Sorting arr2 using parallelized quicksort.\n");
    printf("Original arr2: ");
    verifySorted(arr2, N);
    startTime = get_wall_seconds();
    quicksortParall(arr2, 0, N-1);
    printf("Sorting time: %3.6f seconds.\n", get_wall_seconds() - startTime);
    printf("Verifying...\n");
    printf("Sorted arr2:   ");
    verifySorted(arr2, N);
    printf("\n\n");
    free(arr2);

    pthread_exit(NULL);
}
```

B Appendix - Makefile

```
CFLAGS=-Wall -O3
LDFLAGS=-lm -pthread

Quicksort: Quicksort.o
	gcc -g -o Quicksort Quicksort.o $(LDFLAGS)

Quicksort.o: Quicksort.c
	gcc -g $(CFLAGS) -c Quicksort.c

run: Quicksort
	./Quicksort

debug:
	gdb ./Quicksort

valgrind:
	valgrind --leak-check=full ./Quicksort

clean:
	rm -f ./Quicksort *.o
```