Parallel Quicksort vs Serial Quicksort

A Brief Analysis of an OpenMP Quicksort vs a Traditional Quicksort

By: Zak Kastl

# Introduction

Quicksort was published in 1961 and was notable for its easy implementation and its speed, qualities that still make it the de-facto sort in several C libraries. As computers become smaller, faster and with multiple cores located on the motherboard, it is wise to make use of parallel processing to reduce the time it takes to sort large arrays of values. This will be more important as Big Data continues to process quantities of data in the terabyte range or higher. With that in mind, sorting arrays of or larger would be a common occurrence.

OpenMP is an industry standard in parallel processing. It is a set of compiler pragmas that allow efficient parallel processing using a minimum of commands and changes to existing code. This seems an opportune position to place within quicksort. This paper will examine two versions of quicksort, one created in serial, and one expanded to parallel. It will examine the time to sort arrays ranging from the very small to the very large and will examine the performance of the two algorithms.

# Test System Specifications

While big data systems will work with massive systems, more modest multicore systems can benefit from the enhancements brought about by multiprocessing. Below is the specifications for the test system:

* Processor: Intel Core i5-6300U CPU @ 2.40GHz, 2 Cores, 4 Logical Processors
* OS: Microsoft Windows 10 Pro
* Physical RAM: 8.00GB
* C Compiler Version: mingw32-gcc 4.9.3-1 / Microsoft Visual C++ 2017 (MSVC)
* OpenMP Version: 4.8.1-4 (MinGW)
* Microsoft Office 365

# Experiment

Source code written in C is designed to run Quicksort and a slightly modified version of Quicksort that implements the fewest OpenMP pragmas necessary to parallelize Quicksort. The original Quicksort is an implementation of the author’s own design following the basic algorithm of Quicksort with the pivot being selected as the middle element of the array.

The parallel version of Quicksort comes from a version of Quicksort designed by Eduard Lopez. It uses the same basic principle as the traditional Quicksort with the same pivot, but allows for explicit variables in its partitioning scheme that can be compared. One of the major issues for the experiment was the problem of Stack Overflow errors that occur when the array size is very large. The issue was solved when a recursion cutoff was implemented to avoid creating too many threads due to recursion.

Another problem observed was that MSVC only allows for OpenMP 2.0, with no plans to standardize to the latest version (as listed in the specifications) any time soon. Despite the cutoff, MSVC still threw Stack Overflow errors with very large array sizes. The program was then switched to MinGW, an implementation of GCC for Microsoft Windows. This allowed for the OpenMP 4.x pragmas, and the program was modified accordingly. The largest change in this was the reintroduction of the *omp task* pragma, used in Lopez’s original code.

The results of the experiment use this pragma instead of the *omp sections* pragma. The only difference between tasks and sections for this code is that, because tasks will allow code following the tasks section to be executed without waiting for all of the threads to return. This difference is noticed when attempting to review the program’s progress. The output can be delayed using sections while the other threads finish their sorts. Therefore for this paper, the *omp tasks* pragma was left in, but the sections pragma was left as a code comment for future study. Since the parallelization of quicksort is not affected by sections vs pragmas, it should not affect the results.

Before and after sorting an array with the different Quicksorts, the program uses *omp\_get\_wtime* for timing analysis. This timing for both sorts along with the array size is then reported to the standard output window, which can be redirected into an output file using the command line.

# Results

The program was then run, with the standard output redirected into a Comma Separated Value (CSV) format file. The program prints the size of the array and the time to sort that array both in the traditional and parallel Quicksorts. This CSV file was then imported into Microsoft Excel in order to utilize the powerful statistical analysis tools provided. Incrementing the size of the array by 10,000 in each iteration created 1000 separate data points and these points were charted in Excel as a pair of scatter plots.

Figure 1 - Quicksort Times for Randomized Integer Arrays

The data above shows the sort times for each size of array. Due to the effect an unsorted array might have on its sort time in Quicksort, there is some variation in the sort times, which increase as the array size grows. The Yellow and Green lines are lines of best fit for each plot of data points.

From a glance, we can see that the traditional Quicksort performs at quadratic speed, somewhere in the performance order of ). This is the worst-case performance for quicksort, but easy to implement and understand. The average case of Quicksort runs in time.

In contrast, the parallelized version of Quicksort runs in time. This matches the theoretical best-case performance of the serial version of Quicksort. However, the important point of these results is that the algorithm was not changed aside from a modification to the partitioning scheme that allowed for proper parallelization. Pivot remained the same for both algorithms.

Additionally, there is less variance in the parallel version of quicksort. The of the serial version of Quicksort runs at ~0.93 (a really good fit) versus ~0.97 for the parallel version (a really great fit).

# Conclusion

Just by adding a few OpenMP pragmas, Quicksort has moved from its worst case scenario to its best, all without redesigning the algorithm or pivot. Additionally, the unexplained variance in sorting times, while very low in the traditional Quicksort, is almost negligible with the parallel version.

What can we draw from this? OpenMP is an exceptionally simple and powerful tool to enhance the C programmer’s toolbox. A simple implementation of Quicksort can be improved by an order of magnitude by adding in only 4 pragma directives and allows one to sort arrays of millions of integers in less than 2 seconds.

Of course as with any parallel processing library, OpenMP requires multiple CPU cores to have any effect on performance. It may also require some slight modification to the algorithm to properly use some pragmas. Additionally, while the parallel algorithm does run faster, noticeable performance improvements do not begin until the array approaches 2,000,000 elements. If one is programming and knows that that value is beyond the maximum that will be needed to sort, it can be beneficial to leave quicksort in serial.

The algorithm runs into issues on some compilers involving Stack Overflow errors. However, the benefits far outweigh the extra work required to implement multiprocessing. The modest machine that ran the experiment in this paper is the minimum performance improvement. As one

# Appendix A: Source Code

**Main.c**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <time.h>

#include <omp.h>

#include "ArrayUtils.h"

#include "Sorting.h"

#define NUM\_THREADS 2

/\*

Main Sorting driver method.

This will sort a randomly generated array of integers in the

range of 0-500 in increasing numerical order using two versions of quicksort. The

first is a simple traditional quicksort, performed in serial. The second is a OpenMP

modified version of quicksort that will run in parallel using the number of threads

defined in the above macro. Quicksort selects the pivot that is the midpoint of the array

in both serial and parallel.

\*/

void Sorting(int num\_to\_sort)

{

/\* Declare variables \*/

double start\_time, serial\_execution\_time, parallel\_execution\_time;

int parallel\_faster = 0;

/\* Generate a list of random numbers to sort. \*/

int\* list = RandomList(num\_to\_sort);

/\* Copy the list for comparison to the serial sort. \*/

int\* copy = CopyList(list, num\_to\_sort);

/\* Sort the array using a traditional Quicksort method. Calculate the sort time. \*/

start\_time = omp\_get\_wtime();

Quicksort(list, 0, (num\_to\_sort - 1));

serial\_execution\_time = omp\_get\_wtime() - start\_time;

/\* Sort the copy of the array using the modified openmp quicksort; this should

\* be compatable with OpenMP >=2.0. Calcualte the sort time \*/

start\_time = omp\_get\_wtime();

QuicksortParallel(copy, 0, (num\_to\_sort - 1), NUM\_THREADS);

parallel\_execution\_time = omp\_get\_wtime() - start\_time;

if (!PrintArray(copy, num\_to\_sort, 0))

/\* Report to standard output the number sorted, the times it takes\*/

printf("%d, %lf, %lf\n", num\_to\_sort, serial\_execution\_time, parallel\_execution\_time);

/\* Clean up the heap. \*/

free(list);

free(copy);

}

/\* Special driver function. This function will run the quicksort algorithm from 10^1 to

\* 10^9 values. Each sorting will return whether or not the serial or the parallel version

\* is faster in order to determine just when you should use parallel processing.

\* This will be not 100% accurate, due to the random nature of the list and its effect

\* on quicksort's performance. \*/

void ProgressiveSorting()

{

printf("Array Size, Sort Time with Quicksort in Series, Sort Time with Quicksort in Parallel\n");

int list\_size;

for(list\_size = 10000; list\_size < 10000000; list\_size+=10000) {

Sorting(list\_size);

}

}

/\* Main method. \*/

int main(int argc, char\* argv[])

{

ProgressiveSorting();

return 0;

}

**ArrayUtils.h**

#pragma once

int\* CopyList(int\* list, int size)

{

int\* copy = (int\*)malloc(size \* sizeof(int));

for (int i = 0; i < size; i++) {

copy[i] = list[i];

}

return copy;

}

int PrintArray(int \*A, int size, int print)

{

int outOfOrder = 0;

int prev = -1;

int i;

for (i = 0; i < size; i++) {

if (A[i] < prev)

outOfOrder = 1;

prev = A[i];

}

//printf("%c", (outOfOrder ? 'Y' : 'N'));

return outOfOrder;

}

int\* RandomList(int num\_rands)

{

/\* Initialize values \*/

srand((unsigned int)time(NULL));

int \*list = (int\*)malloc(num\_rands \* sizeof(int));

for (int idx = 0; idx < num\_rands; idx++) {

list[idx] = rand() % 500;

}

return list;

}

**Sorting.h**

#pragma once

/\* Declarations \*/

void Swap(int \*a, int \*b);

void Quicksort(int \*A, int low, int high);

void QuicksortParallel(int \*A, int low, int high, int num\_threads);

void QSP\_internal(int \*A, int low, int high, int cutoff);

/\* Implementations \*/

/\* Serial Quicksort method. Uses the array's midpoint as the starting

\* pivot. \*/

void Quicksort(int \*A, int low, int high)

{

if (low < high)

{

int pivot = (low + high) / 2;

int index = low;

while (index < pivot)

{

if (A[index] < A[pivot])

index++;

else {

Swap(&A[index], &A[pivot - 1]);

Swap(&A[pivot - 1], &A[pivot]);

pivot--;

}

}

Quicksort(A, low, pivot - 1);

Quicksort(A, pivot + 1, high);

}

}

/\* This method was inspired by:

\* https://github.com/eduardlopez/quicksort-parallel/blob/master/quicksort-omp.h

\* Lopez's code is a slight modification of the serial quicksort method, shown in this file.

\* This code utilizes OpenMP >= 3.0 to divide the separate recursive portions of quicksort

\* into separate tasks. Since my version of C (MSVC) only supports OpenMP 2.0, a change was

\* required.

\*

\* In this version, a modification of the version located at the link. The tasks pragma

\* was replaced with a sections pragma. It was a slight modification, but increases the

\* compatability with different compilers. \*/

void QuicksortParallel(int \*A, int low, int high, int thread\_count)

{

int cutoff = 1000;

#pragma omp parallel num\_threads(thread\_count)

{

#pragma omp single nowait

{

QSP\_internal(A, low, high, cutoff);

}

}

}

/\* Internal method to be used in the parallelization. \*/

void QSP\_internal(int \*A, int low, int high, int cutoff)

{

int i = low;

int j = high;

int pivot = A[(low + high) / 2];

{

/\* Partition\*/

while (i <= j) {

while (A[i] < pivot)

i++;

while (A[j] > pivot)

j--;

if (i <= j) {

Swap(&A[i], &A[j]);

i++;

j--;

}

}

}

if (((high - low) < cutoff)) {

if (low < j)

QSP\_internal(A, low, j, cutoff);

if (i < high)

QSP\_internal(A, i, high, cutoff);

}

else {

/\*#pragma omp sections

{

#pragma omp section

QSP\_internal(A, low, j, cutoff);

#pragma omp section

QSP\_internal(A, i, high, cutoff);

}\*/

/\*#pragma omp task

QSP\_internal(A, low, j, cutoff);

#pragma omp task

QSP\_internal(A, i, high, cutoff);

}

}

/\* Simple Swap method. \*/

void Swap(int \*a, int \*b)

{

int c = \*a;

\*a = \*b;

\*b = c;

}

# Appendix B: References

<https://github.com/eduardlopez/quicksort-parallel/blob/master/quicksort-omp.h>

<https://en.wikipedia.org/wiki/Quicksort>