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 have become smaller, faster and containing multiple, 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 petabyte range or higher. With that in mind, sorting arrays of elements 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 parallelization of code 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 i7-2600K CPU @ 3.40GHz, 4 Cores, 8 Logical Processors
* OS: Microsoft Windows 10 Pro
* Physical RAM: 16.00GB
* C Compiler Version: mingw32-gcc 4.9.3-1
* OpenMP Version: 4.8.1-4 (MinGW)
* Microsoft Office 2016

# Experiment

The source code is written in C and is implements two versions of Quicksort: a traditional version that partitions an array using a central pivot, and a parallel version that is differentiated solely in that it implements several OpenMP pragmas designed to parallelize it. The parallel version of Quicksort was designed by Eduard Lopez, and the link to his code is in Appendix B. The serial version is a modification of this parallel version, with only the pragmas removed.

The main issue with the experiment was a stack overflow error that occurred when array sizes were very large, greater than 10 million elements. This problem was observed using the Microsoft Visual C++ compiler (MSVC). Regardless of the method used to parallelize it, the algorithm would begin to fail with very large arrays, but still within the limits of 32 bit signed integers.

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 mitigations to avoid them, 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.

Originally, to keep the program using the OpenMP 2.0 standard, the *omp task* pragma was replaced with the *omp sections* pragma, however the sections pragma was deemed unfit for the task. Therefore, the experiment changed the compiler to the GNU Compiler Collection (GCC) which does support OpenMP 4.x.

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 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 both algorithms run at efficiently, near linear speed. However, it is clear that the parallel quicksort performs faster than the serial quicksort, minor at first, but increases quickly. The question is then, “How much faster is the parallel version?” We can possibly find this out by dividing the two best fit slope values , which gives a value of 2.6. This is not, however the true speedup of the program. This only tells us how much faster the serial algorithm is increasing over the parallel version. Examining the values at various array sizes shows an increase in speed of about 3 times faster. An array size of 9990000 gives a speed up of ~3.10 times, a much better value.

What if the program progresses further? A second experiment was run, where the ProgressiveSorting method, which drives the various array sizes was altered to let the array size start with 2 elements, increasing by a factor of 2 each iteration, until the array is near 2^32-1. At 134217728 elements, the speed up is ~3.8 times. Figure 2 shows array sizes as set to powers of 2.

Figure 2 -Quicksort Timing Data Using Array Sizes of 2^x

From this we can see that this linear progression continues on similar to that of Figure 1. It is unlikely that we will achieve 8x speedup with this algorithm.

# Conclusion

The ideal speedup for an algorithm moving from serial to parallel is a speedup of *N*, where *N* is the number of processors in the machine. Quicksort could possibly achieve this speedup, but it would take time and effort to implement. What is more amazing is that with the addition of 4 pragmas, once can achieve a speedup of nearly 4 times the algorithm’s original speed. That is the power of OpenMP. Other techniques could possibly be faster than the one outlined here, but the simplicity of OpenMP can have a powerful effect on software.

While we may not get the maximum amount of performance from retrofitting algorithms with OpenMP, there is an industry that relies on code that is not changed often, if ever. OpenMP was made for this. With the addition of 4 pragmas, and the linking of *omp.h* a legacy software could quadruple its speed. Utilizing other multiprocessing techniques may be useful for better performance, but only for software that allows change.

In the end, OpenMP and other parallel programming techniques are powerful tools in the software engineer’s toolbox and should always be considered when writing software where performance is critical. The above figures may not show much of a difference with small values, but in this world of Big Data, the above values may be small subsets of massive databases. One needs all of the performance one can get.

# 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 8

#define MAX\_LEN 262144000

/\*

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. The only difference in the programs is that one possess

OpenMP pragmas.

\*/

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 \*/

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.\*/

void ProgressiveSorting()

{

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

int list\_size;

for(list\_size = 2; list\_size <= MAX\_LEN; list\_size\*=2) {

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 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--;

}

}

}

Quicksort(A, low, j);

Quicksort(A, i, high);

}

}

/\* This method was inspired by:

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

\*/

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 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>