Parallel Quicksort vs Serial Quicksort

A Brief Analysis of an OpenMP Quicksort vs a Traditional Quicksort

# 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 list of compiler pragmas that allow efficient parallel processing using a minimum of commands and changes to legacy code. This seems an opportune position to place within quicksort. This paper will examine two versions of quicksort, one created in serial, and one in parallel. It will examine the time to sort arrays ranging from the very small to the very large and will examine the potential speedup of the two quicksorts.

# Test System Specifications

While big data systems will work with massive systems, slower systems can benefit from the enhancements brought about by multiprocessing.

# 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 4

#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.

\*/

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