Parallel Binary Search Using OpenMP

Dharak Vasavda

d_vasavda@u.pacific.edu

May 4, 2020

1 Problem

Binary Search is a divide-and-conquer searching algorithm which searches for an input value in a sorted array. The average-case time complexity for Binary Search is $\mathcal{O}(logn)$. Considering that it is a divide-and-conquer search where the input array is repeatedly divided into equal halves, can the search operation be made even faster using OpenMP parallelization?

Prior Solutions

Parallel Binary Search implementations with OpenMP do exist, however; rarely is there any specifically real-world timing data comparison between a serial and parallel implementation. Due to such circumstance, this scientific paper aims to compare the real-world performance between a parallel and serial implementation. It also aims to investigate any real-world performance differences between a serial and parallel approach to Binary Search.

2 Design

Typical Serial Binary Search

Typically, a serial approach to non-recursive Binary Search algorithm goes as follows:

Listing 1. Serial Binary Search

```
while (first <= last) {
    int middle = first + (last - first) / 2;

    // Check if search value is present at middle
    if (randomNums[middle] == searchVal)
        return middle;

    // If search value greater, ignore left half
    if (randomNums[middle] < searchVal) {
        first = middle + 1;
    }

    // If search value is smaller, ignore right half
    else
        last = middle - 1;
}
```

Parallel Binary Search Design

The sample code block below may assist with understanding the parallel solution.

Listing 2. OpenMP Binary Search

```
#pragma omp parallel num_threads(num_threads)
    {
#pragma omp sections
        {
        /* Function parameters:
            binarySearch_openmp(first_index, last_index, search_value);
        */

#pragma omp section
            thread_one = binarySearch_openmp(0, quarter_slice, searchVal);
#pragma omp section
            thread_two = binarySearch_openmp(quarter_slice + 1, middle, searchVal);
#pragma omp section
            thread_three = binarySearch_openmp(middle + 1, quarter_slice * 3, searchVal);
#pragma omp section
            thread_four = binarySearch_openmp((quarter_slice * 3) + 1, last, searchVal);
        }
    }
```

Parallel Algorithm Explanation

Each call to the function responsible for binary search is now given the input of a new array size, where each input array is a quarter size of the original input array. From there, each thread will simultaneously perform a search operation on a quarter section of the array. Once a single thread finds the desired value to search for, all threads will stop the searching process and return the index of the search value.

The pragma directive of choice here is using sections, so that each task can run on its own thread. The idea behind this algorithm is to reduce the search array size and reduce the number of operations, while each thread can perform each operation in parallel.

3 Experiments

Data Sources

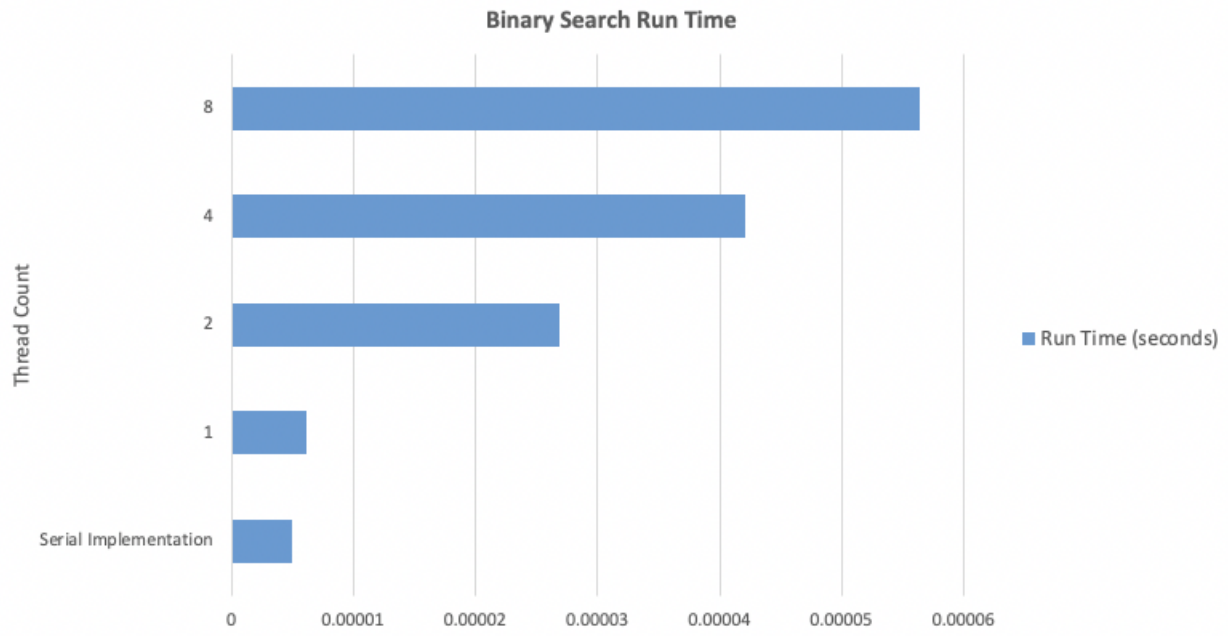
All data was generated using an integer generator. All numbers are placed in ascending order into a file which will be used by the operation code. The data is inserted into an array in the source code to perform Binary Search experimentation upon.

Data Type	Integer
Size	200,000,000
Memory Type	fixed-size array
Order	Ascending
Sorted	Yes

Table 1. Data Source Information

Results

Below, contains both timing results between Parallel Binary Search and Parallel Binary Search using OpenMP. Timing data was recorded using *omp_get_wtime*(). Four runs were taken per entry and timing data was averaged. The operation was a Binary Search, where the user will input any number between 1 and 200,000,00 to search for.



Thread Count	Run time (seconds)
1 (Serial)	∼0.000005

Table 2. Serial Binary Search Timing

Thread Count	Run time (seconds)
1	0.0000061
2	0.0000269
4	0.0000421
8	0.0000564

Table 3. Parallel Binary Search Timing

4 Analysis

Parallel Time Complexity	$\mathcal{O}(log(n/k))$
Speedup Estimate	$\frac{log(n^2)}{k}$
Efficiency Estimate	$\frac{log(n^2)}{k^2}$

Table 4. Theoretical Estimates of Parallel Binary Search

Real-World Results

Referring to the data displayed in Table 2 and Table 3, the serial implementation of Binary Search is significantly faster than the parallel implementation. As the number of threads performing Binary Search increased, the operation took exponentially longer. There are a few reasons I can hypothesize as to why these results occurred.

Effectivity Limits of Parallel Solutions

One crucial idea to note, when finding a parallel approach to a problem, is that the serial operation needs to be costly enough to computer resources to parallelize. Often times, complex calculations such as matrix multiplication or dot product are used to demonstrate parallel effectivity, for a specific reason. That is, the operation needs to be costly enough to outweigh the performance repercussions of a parallel solution. Specific performance repercussions such as thread spawning, excessive function calls, voltage and thermal limitations across several threads, and cache hierarchy issues can all play a part in slowing down performance. In the case of Binary Search, the operation simply is not costly enough to make useful out of a parallel approach. OpenMP is not a solution to all problems.

5 Implementation

Building and Running This Application

To build the project and run experiments with it:

1. cd into project directory
2. Run the integer generator, which will output all integer data into input1.txt
3. Compile the C file
4. Run the compiled file

```
> cd openmp-binary-search
> python3 integer_generator.py
> clang -Xpreprocessor -fopenmp -lomp binarySearch_openmp.c -o binarySearch_openmp
> ./binarySearch_openmp
```

Running Experiments

There are several factors which need to be accounted for before running experiments and gathering data for a similar program:

- Ensure multiple data points are calculated for averaging
- Ensure you are running costly enough operations
- Ensure operation data set is large enough
- Keep aware that high-resolution timers are not always fully accurate for each floating point
- Therefore, do research before choosing a timer
- Create a stable thermal environment for the CPU before collecting data

After following the above awareness points, the data collection goes as follows:

1. Generate all numbers using the random number generator
2. Pick several numbers of threads in powers of 2
3. Run the program with each thread count, four times total
4. Average out each run
5. Use the average as the calculated run time

6 Code Appendix

Listing 3. *binarySearch_openMP.c*

```
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include <sys/time.h> // High resolution timer
#include <omp.h>

const int ARR_SIZE = 200000000; // Add 200,000,000 elements into array
int randomNums[ARR_SIZE];       // Array of random integers
int parallel_search_found = 0;
double omp_get_wtime(void);

// High resolution timer
inline uint64_t rdtsc()
{
    uint32_t lo, hi;
    __asm__ __volatile__(
        "xorl %%eax, %%eax\n"
        "cpuid\n"
        "rdtsc\n"
        : "=a"(lo), "=d"(hi)
        :
        : "%ebx", "%ecx");
    return (uint64_t)hi << 32 | lo;
}

/*
    void readInputFile() -
        Takes contents from file input1.txt and places it into array randomNums[]
*/
void readInputFile()
{
    FILE *inputFile;
    inputFile = fopen("input1.txt", "r");

    int i;

    if (inputFile == NULL)
    {
        printf("Error Reading File input1.txt\n");
        exit(0);
    }

    printf("Populating randomNums[] with data from input1.txt...\n");
    printf("(This may take a while)\n");
    // Place file contents into array randomNums[]
    for (i = 0; i < ARR_SIZE; i++)
    {
        fscanf(inputFile, "%d,", &randomNums[i]);
    }

    // // Print a few values from array to verify elements
```

```
    // for (int i = 0; i < 4; i++) {
    //     printf("%d\n", randomNums[i]);
    // }

    printf("Finished inserting %d elements into randomNums[]\n", ARR_SIZE);

    fclose(inputFile);

    printf("\n");
}

/*
    void binarySearch_serial() -
        Performs serial binary search on randomNums[]
*/
int binarySearch_serial(int searchVal)
{

    int num_elements = ARR_SIZE;
    int first = 0;
    int last = num_elements - 1;

    while (first <= last)
    {
        int middle = first + (last - first) / 2;

        // Check if search value is present at mid
        if (randomNums[middle] == searchVal)
            return middle;

        // If search value greater, ignore left half
        if (randomNums[middle] < searchVal)
        {
            first = middle + 1;
        }

        // If search value is smaller, ignore right half
        else
            last = middle - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

/*
    void binarySearch_openmp() -
        Performs parallel binary search using OpenMP on randomNums[]
*/
int binarySearch_openmp(int first, int last, int searchVal)
{
    // // Print current thread number
    // int tid = omp_get_thread_num();
    // printf("Hello World from thread = %d\n", tid);
```

```
    while (first <= last)
    {
        int middle = first + (last - first) / 2;

        // Check if search value is present at mid
        if (randomNums[middle] == searchVal)
            return middle;

        // If search value greater, ignore left half
        if (randomNums[middle] < searchVal)
        {
            first = middle + 1;
        }

        // If search value is smaller, ignore right half
        else
            last = middle - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}

/*
    void serial_work() -
        Performs all work related to Serial code, including:
            - All print statements
            - Timing of Serial work
            - Binary search function calls
*/
void serial_work(int searchVal)
{
    int result;
    double start, end, total_time;

    printf("\n****** Now beginning Serial work ******\n\n");

    printf("Starting binary search...\n");

    start = omp_get_wtime();
    // Perform serial Binary search with timing
    result = binarySearch_serial(searchVal); // Binary search here

    end = omp_get_wtime();
    total_time = end - start;

    printf("Work took %f seconds\n", total_time);

    // Print results of serial Binary search
    if (result != -1)
    {
        printf("Element %d found! At index %d\n", searchVal, result);
    }
```

```
    else
    {
        printf("Element %d not found\n", searchVal);
    }
    printf("\n");
}

/*
    void parallel_work() -
        Performs all work related to Parallelized code, including:
            - All print statements
            - Timing of parallel work
            - Binary search function calls
*/
void parallel_work(int searchVal, int num_threads)
{
    int result;
    double start, end, total_time;

    // For use with Parallel search
    int first = 0;
    int last = ARR_SIZE - 1;
    int middle = first + (last - first) / 2;

    // Array will be sliced into sections
    int thread_one, thread_two, thread_three, thread_four;
    int quarter_slice = middle / 2;

    printf("\n****** Now beginning Parallel work with OpenMP ******\n\n");

    printf("Starting binary search...\n");

    start = omp_get_wtime();

#pragma omp parallel num_threads(num_threads)
    {
#pragma omp sections
        {
        /* Function parameters:
            binarySearch_openmp(first_index, last_index, search_value);
        */

#pragma omp section
            thread_one = binarySearch_openmp(0, quarter_slice, searchVal);
#pragma omp section
            thread_two = binarySearch_openmp(quarter_slice + 1, middle, searchVal);
#pragma omp section
            thread_three = binarySearch_openmp(middle + 1, quarter_slice * 3, searchVal);
#pragma omp section
            thread_four = binarySearch_openmp((quarter_slice * 3) + 1, last, searchVal);
        }
    }

    end = omp_get_wtime();
    total_time = end - start;
```

```
    printf("Work took %f seconds\n", total_time);

    // Print results of serial Binary search
    if (result != -1)
    {
        printf("Element %d found! At index %d\n", searchVal, result);
    }
    else
    {
        printf("Element %d not found\n", searchVal);
    }
    printf("\n");
}

int main(int argc, char *argv[])
{
    int searchVal, num_threads;

    printf("Enter value to search for: \n");
    scanf("%d", &searchVal);

    printf("How many threads to run on: \n");
    scanf("%d", &num_threads);

    // Read contents of input file and populate array
    readInputFile();

    // Perform all serial work
    serial_work(searchVal);

    // Rewrite contents to array for consistency purposes
    readInputFile();

    // Perform all parallelized work
    parallel_work(searchVal, num_threads);

    return 0;
}
```

Listing 4. *integer_generator.py*

```
'''
Generates random integers and places all data into input1.txt

Each integer will be writtin into a new line in input1.txt

'''
import random

nums_to_generate = 200000000
lower_bound = 0
upper_bound = nums_to_generate


output_file = open("input1.txt", "w" )

print("Generating", nums_to_generate, "random integers...")
print("(This may take a while)")

for i in range(nums_to_generate):
    line = str(i)
    output_file.write(line)
    output_file.write('\n')

print("Random numbers generated:", nums_to_generate)
print("Generated numbers range from:", lower_bound, "to", upper_bound)
print("Random number generation completed.\n")

output_file.close()
```