Investigating the Benefits of Parallel Processing for Binary Search

A. Paul Mullins Ph.D., Advisor1, B. Gennifer Elise Farrell 2, and C. Ronald Baldwin 3

1, 2, 3 Department of Computer Science, Slippery Rock University, Slippery Rock, Pennsylvania, USA

Abstract - Research explored possible benefits of parallel programming for the binary search, specifically relating to how divisions of threads in a binary search affect efficiency of the algorithm. A unique search, named Congruent Binary Search (CBS), was written with the goal of avoiding the pitfalls of the standard parallel binary search. It was hypothesized the CBS would run in $O(\log\left(\frac{N}{p}\right))$, where $N$ is number of elements in the search and $p$ is number of processors. Virtually no speed increase of the CBS was found when compared to standard serial and paralyzed binary search. Research failed to find strong evidence that the CBS offers a performance boost over other binary searches. The binary search is not a good candidate for parallel programming.

Keywords: Parallel programming, binary search, undergraduate research project

1 Introduction

This undergraduate research project asks if the binary search can benefit from parallel programming. This research investigates if any speed up factor will be equivalent to the number of threads used. Current implementations of the parallel binary search tend to split an array into small arrays many times over. An example is included in “Efficient Parallel Binary Search on Sorted Arrays” by Danny Z. Chen [1]. Chen discusses an algorithm that uses two arrays A and B with n numbers and m numbers that has a time complexity of $O(\frac{n \log\left(\frac{n}{m}\right)}{\log(m)})$. Though this is a valid approach there are some key points where an alternative may be more desirable. One such alternative is one that sorts in place and does not require the overhead of splitting data into multiple arrays. This most notably causes an increase in space complexity.

The Congruent Binary Search was written as an alternative. In this algorithm one only needs to use a single array for storage; implementation is much simpler by nature, and in theory should have a significant speed boost.

Furthermore, this experiment is important because there is currently little related research established. According to many on stack overflow, the community is currently divided on how this should be done, or if it should be done at all [2]. This obvious inconsistency in the programming community gives ample reason for testing.

For the experiment, the Congruent Binary Search is created and tested against its serial version, and the seemingly most common algorithm for parallel binary search. The execution times are compared and speed factor is calculated, using the serial binary searches execution time for the same data sets. It is hypothesized that the Congruent Binary Search will improve execution time. It is hypothesized the execution time of the Congruent Binary Search will $O(\log\left(\frac{N}{p}\right))$, where $p$ is the number of processors. The array is split by even and odd indexes. To split the array in half is foolish, because on the first iteration of a serial binary search one of these halves would be thrown out.

2 Report

First, two classes are created for testing. The first class has an array of size one hundred million. Each index in the array contains an integer with an assigned value of its address. This is done for an easy way to attain a large sorted array. Objects are then created which are initialized with the value that is being searched for in this array. Then there are methods that perform the serial binary search and the Congruent Binary Search.

The Congruent Binary Search utilizes one array and two threads. One thread searches on even indexes for the value and the other searches only odd indexes. One thread searches the first array the other searches the second.

The second class contains two arrays, each with a size of fifty million. It is initialized same way as the first class.

After class creation, a test application was created that runs each search with the same number of elements. All algorithms were searching for the same value. This is done sequentially in a for loop so each search can be executed and timed many times over. Each search was saved to a file specified before execution begins. Each individual execution of each search was recorded with its type, time in nanoseconds, and number of threads. After all iterations were complete, the average is taken from each type of search. To find speedup factor, average execution time of each search was used. The serial algorithm was divided by the Congruent Binary Search algorithm to get its speedup factor. Then, the alternative parallel algorithm average execution time is divided by that of the Congruent Binary Search algorithm. Because the Congruent Binary Search is an enhancement of
the two established algorithms, Amdahl’s law is used to calculate speedup of the Congruent Binary Search compared to both established algorithms. Thus, by formula (1), speedup is found.

\[
\frac{\text{Average execution time pre – enhancement}}{\text{Average execution time post – enhancement}} = \text{speedup}
\]  

(1)

In case the thread scheduling by the operating system was affecting the results, each test was run individually. Results were approximately the same. Additionally, all three algorithms were running at once with one iteration and the results were inconsistent; the very first execution is slower than expected. This can be explained by the Java Virtual Machine warm up. The first iteration isn’t useful but it is negligible. The many trials executed assure a more precise answer, by the law of large numbers. That is why execution times are recorded for a million trials.

To calculate time complexity, the relationship between each threads’ iterations is used to hypothesize the Congruent Binary Search true execution time. A value is chosen on either end of the array: aiming to set up a worst-case scenario. Thus, 0 stored at index 0 is searched for. After analyzing all the data, it was concluded that the parallel search is not a good candidate for a parallel implementation.

3 Data

Tables (i) and (ii) record execution time for the two parallel algorithms that are compared. Table (iii) compares the Congruent Binary Search to the Serial Binary Search. Serial binary search is approximately five times faster, although not depicted in the table data.

(ii) Average Found Through Single Iterations (Two Threads)

<table>
<thead>
<tr>
<th>Trial</th>
<th>Standard Parallel Binary Search</th>
<th>Congruent Binary Search</th>
<th>Search Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>143980 ns</td>
<td>165702 ns</td>
<td>1000</td>
</tr>
<tr>
<td>2</td>
<td>143050 ns</td>
<td>140568 ns</td>
<td>1000</td>
</tr>
<tr>
<td>3</td>
<td>163530 ns</td>
<td>148635 ns</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>149256 ns</td>
<td>153290 ns</td>
<td>1000</td>
</tr>
<tr>
<td>5</td>
<td>159496 ns</td>
<td>168495 ns</td>
<td>1000</td>
</tr>
<tr>
<td>6</td>
<td>160737 ns</td>
<td>154220 ns</td>
<td>1000</td>
</tr>
<tr>
<td>7</td>
<td>160116 ns</td>
<td>143981 ns</td>
<td>1000</td>
</tr>
<tr>
<td>8</td>
<td>153290 ns</td>
<td>152980 ns</td>
<td>1000</td>
</tr>
<tr>
<td>9</td>
<td>148324 ns</td>
<td>157014 ns</td>
<td>1000</td>
</tr>
<tr>
<td>10</td>
<td>150807 ns</td>
<td>142739 ns</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table ii: Difference: 496.2 ns in favor of even/odd split

Speedup = 153258.2ns/152762.4ns = 1.0032  
(virtually no speed increase)

(iii) Iterations Per Thread

<table>
<thead>
<tr>
<th>N</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>61</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>75</td>
<td>4</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>150</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Table iii: This data table lists the number of iterations for each thread, where n is the length of the array being searched.

4 Congruent Binary Search for Evens and Odds

One thread handles even indexes and one thread handles odd indexes. For a thread of even and odd indexes, ‘left’ is set to the first index for even or (first + 1) for odd. Furthermore ‘right’ to the last even index for even and the last odd index for odd. Then the midpoint is calculated. With the even thread midpoint is checked that it is even and odd in the odd thread midpoint is checked that it is odd. This is done via the modulus operator. If needed, the value of midpoint changes so that it remains within its proper range. For example, it is possible to find an odd midpoint between two even indexes. This is not allowed, because the midpoint must always be located within the even thread. If the midpoint isn’t even, it is shifted one space to the left.
Either ‘left’ or ‘right’ are assigned a new index, depending on the value at the midpoint index when compared to the current value being searched for. With these new ‘left’ and ‘right’ points, the process of calculating the midpoint is repeated, cutting the array in half each iteration. The iteration stops when either ‘left’ is greater than ‘right’ (indicating value was not found within thread) or midpoint is equal to the value being searched for (indicating that the value was found).

The above process is shown in the code in figures 1 and 2.

```
//does binary search on even indexes
public void binaryThread1() {
    int right = 0;
    int mid = this.data.length - 1;
    if(right % 2 == 0)
        right--;
    mid = (int)((right + left) / 2);
    if(mid % 2 == 0)
        mid--;
    //cut off half values depending on if mid point is
    //greater or less than the value we are searching for
    while (left <= right & flag == false)
    {
        if(this.searchVal == data[mid])
            flag = true;
        else if (this.data[mid] > searchVal)
        {
            right = mid - 2; // only access evens
            mid = (int)((right + left) / 2);
            if(mid % 2 == 0)
                mid--;
        }
        else
        {
            left = mid + 2;
            mid = (int)((right + left) / 2);
            if(mid % 2 == 0)
                mid--;
        }
    }
}
```

Figure 1: This is the code for the section of the Congruent Binary Search that handles the thread of even indexes

```
@override
//does binary search on odd indexes
public void run() {
    int left = 1;
    int right = this.data.length - 1;
    if(right % 2 == 0)
        right--;
    mid = (int)((right + left) / 2);
    if(mid % 2 == 0)
        mid--;
    while (left <= right & flag == false)
    {
        //cut off half values depending on if mid point
        if(this.searchVal == data[mid])
            flag = true;
        else if (this.data[mid] > searchVal)
        {
            right = mid - 2;
            mid = (int)((right + left) / 2);
            if(mid % 2 == 0)
                mid--;
        }
        else
        {
            left = mid + 2;
            mid = (int)((right + left) / 2);
            if(mid % 2 == 0)
                mid--;
        }
    }
}
```

Figure 2: This is the code for the section of the Congruent Binary Search that handles the thread of odd indexes

### 4.1 Generalizing the congruent Binary Search

Currently, established research has only been performed on an array divided into two threads. The Congruent Binary Search can easily split an array into any number of even number threads. It can be generalized to an odd number of divisions, but these divisions are not desirable for the experiment at hand. An odd number of divisions is undesirable because each thread will contain both even and odd indexes. This causes two problems: The Congruent Binary Search works on threads of just even or just odd indexes. Accommodating for the case of a thread with both even and odd indexes would require a different algorithm. Secondly, even if the Congruent Binary Search could work within one of these threads with both even and odd indexes, new code would have to be written to accommodate the fact that midpoint can now be a decimal. This would be done by adding a call to the floor method.

For splitting the index into p threads, where p is an element of \(2N\), let each thread be a subset of the congruency classes \([0]_p,[1]_p,\ldots,[p-1]_p\). Let each thread be the only subset of an individual congruency class. Do not allow for more than one thread to be a subset of each congruency class.
For each thread, set Left to the first most element of the congruency class it is a subset of. So the index \( p(0)+0, p(0)+1, \ldots p(0)+(n-1) \) will be Left for each respective thread. Set Right to the last (i.e. largest) index for each thread.

Within each thread, the algorithm will iterate through either even or odd indexes. Because the array is divided between an even number of threads, there will never be a thread that contains both even and odd indexes. It is possible that the midpoint between the Left and the Right indexes is not even when within a thread of even indexes. It is possible that the midpoint between two odd indexes is even.

The general form for finding the midpoint, and determining if its value must be changed as to remain within the thread in question, is as follows:

Let ‘a’ = first address in each respective thread.
Let ‘remainder’ = ‘midpoint’ \( \text{mod} \) p
If (‘midpoint’ \( \text{mod} \) p \( \neq \) a)
Then mid = mid – (remainder-a)
Else, midpoint remains the same.

When midpoint \( \text{mod} \) p is equal to a, midpoint is located within the thread in question and its value does not need to be adjusted.

Following the form of the Congruent Binary Search, outlined above, either ‘left’ or ‘right’ are assigned a new index, depending on the value at the midpoint index when compared to the current value being searched for. An array divided into \( p \) threads will need Left and Right values that can change in increments of \( p \). For example, in the even and odd two threaded search, Left and Right’s values can change in increments of two. For an array divided into four threads, Left and Right’s values can change in increments of four.

The search terminates under the same conditions as the above form. The iteration stops when either ‘left’ is greater than ‘right’ or midpoint is equal to the value being searched for.

5 Conclusions

The hypothesis that the Congruent Binary Search algorithm could offer a performance boost to the binary search was not supported. In fact, any speed up was negligible because the same execution time was also achieved with the prior method. Some possibilities for error can be accounted for by the fact that Java is not particularly good for parallel programming. In Java, all threads run within the Java Virtual Machine. Unfortunately, it is not possible to see what the Operating System is doing in terms of handling the threads. The Java threads may have had such low priority that the operating system interrupted them often for other system tasks that have higher priority in the system queue. To get more accurate results a different environment may be better. An ideal environment is a parallel programming model that uses explicit parallelism where the programmer has more control over memory management as well as task management.

Additionally, the research fails to find evidence that the Congruent Binary Search offers a performance boost when compared to the serial binary search and the standard parallel binary search. In fact, the serial Binary search outperformed both parallel algorithms. This could be because of underlying low level processes that are unseen. However, it is important to remember that in some cases serial execution is the most performant, especially for smaller tasks. With such tasks the multithreading overhead resulting from parallelization becomes more noticeable.

The theoretical time complexity of the Congruent Binary Search algorithm is determined to be \( O(\log \left( \frac{k}{2} \right)) \). This contradicts the actual time complexity of \( \log (n + k) \) where \( k \) is a constant for thread overhead. This is due to the fact that even when an array is split by even and odd indexes, by the halving of the array, only approximately one iteration is saved than when compared to an array double the size. An example is the time complexity of an array of 100,000,000, performed under the serial binary Search and the Congruent Binary Search, respectively: \( \log (2 \times 100,000,000) = 26.58 \) and \( \log \left( \frac{2 \times 100,000,000}{\left( \frac{100,000,000}{2} \right)} \right) = 25.58 \). Thus, the two threads are still doing roughly twenty-six iterations apiece despite half the number of elements in each search. This can be seen in table (iii), a data table that lists the iterations for each thread for several different values of \( n \), as well as the number of iterations for the serial version. Therefore, if there was no overhead of using threads in parallel time, complexity would be the same as serial’s time complexity, \( O(\log(n)) \). Unfortunately, the overhead of threads can be particularly large and not efficient in some languages. The actual time complexity is calculated to be \( O(\log(n + k)) \) where \( k \) is the overhead for a single thread.

The idea of accessing indexes by even and odd indexes may have some potential in algorithms that are constant or quadratic in time, because of their steeper curve than that of logarithmic algorithms. The Congruent Binary Search algorithm is parallel but not performant. It is concluded binary search is a bad candidate for parallel programming.

For future work, there are currently a few avenues of investigation for the potential effectiveness of the Congruent Binary Search algorithm. Research is being conducted on the effects of changes to the Congruent Binary Search algorithm, utilizing its generalized form. These effects are: splitting the array into a larger number of threads, and a adding a ‘flag’ that allows communication between all threads. Also, current research is exploring adding counter variables to the code to
compare each algorithm’s number of instructions. This offers a way of comparing efficiency in terms of instruction rather than speed.

6 References
