,,

Homework 2

MPCS 51100 due: Oct. 31, 2016 @5pm

For problems 1-3, you are free to modify the code if you feel it is necessary to complete the assignment, e.g. add helper functions, header files, etc. Include the number of threads used with your timing reports.

1. Using `p1.c` as a starting point, write a parallel OpenMP version of the work kernel. Compare the runtimes of the two versions.
2. Write a single program with two functions to calculate the product of two double matrices AB=C each of size $N \times N$. Use `p2.c` as a starting point.
   (a) one function which performs the calculation serially (provided)
   (b) second function which performs the same calculation but with OpenMP. Submit a single file `p2.c`, you may use a header file if you wish.
   (c) Compare timings between the serial and parallel versions
3. Complete the program `p3.c` which includes 3 functions to generate the Mandelbrot Set (`https://en.wikipedia.org/wiki/Mandelbrot_set`).:
   (a) a serial version (provided)
   (b) a parallel version with OpenMP using dynamic scheduling
   (c) a parallel version with OpenMP using static scheduling
   (d) compare the runtime of each function. You can report it in a neat table, or a plot.
   (e) include a picture of the resulting Mandelbrot set image as `mandelbrot.png`; do not submit a text file for this.

4. (a) Implement a simple dictionary using a naive (unbalanced) Model 1 binary search tree as the underlying data structure. The dictionary must support the following text based client commands (% indicates the unix prompt):
   i. % add word "definition" add the given word and associated definition to the dictionary. print success or failure message to screen.
   ii. % delete word remove word from dictionary. print success or failure message to screen.
   iii. % find word find definition associated with input word and print to screen
   iv. % print show the contents of the dictionary in key order

   Be sure to handle all errors in a reasonable way (e.g. the program shouldn't crash if the input is incorrect, or a key cannot be found, etc.).

   (b) Redo (a) using a self-balancing binary search tree

   (c) Develop a meaningful test case comparing (a) and (b). You are responsible for defining the test input and displaying and analyzing the results.

5. (a) Write a non-recursive function to verify that a binary tree is a proper binary search tree (i.e., the left subtree of a node contains only nodes with keys less than the node's key, the right subtree of a node contains only nodes with keys greater than the node's key, and both the left and right subtrees are are binary search trees):
   int isBST(Tree_node *tree)$\{\ldots\}$, with the following definition for Tree_node:

```
typedef struct Tree_node{
  struct Tree_node *left, *right;
  int data;
} Tree_node;
```

   (b) Write a recursive version using the same data structure. You are free to choose the function signature.

6. Let $\{L_1, L_2, \ldots, L_k\}$ denote a set of k similar, sorted arrays with average length n. Suppose we wish to locate the bounding indices of some key q in each of the arrays. In other words, a binary search for a key should return either the index of the key if the key is present in the array or the index of the next largest value in the array otherwise. The most obvious solution is to perform a separate binary search on each array, as shown in Figure 1.

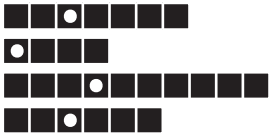

Figure 1: Original arrays where key is located using series of binary searches.

Suppose that instead of locating a single key in the arrays we want to successively locate many different key values. We can use more intricate search techniques which pre-compute some information in order to save time later during the searches. A second approach, which can improve upon

the query time of the naive solution at the cost of a larger memory footprint, involves the creation of a new data structure which requires only a single binary search followed by k lookups using some pre-calculated "pointer" values to identify the location of q in each array. Here, we use the term pointer to refer to an integer index value rather than a C pointer. A new master unionized array is constructed as the ordered union of the values in each of the original arrays. Associated with each value in this unionized array is a series of k pointers (integer indices), one for each of the original arrays, which identify the index in that array that would be returned in a search for that value. Figure 2 depicts the unionized structure and lookup process. Let U denote the unionized array and $\{P_1, P_2, \ldots, P_k\}$ denote the k pointer arrays associated with the original k arrays $\{L_1, L_2, \ldots, L_k\}$. If a binary search for q in U returns index j, the value of the j^{th} element in the pointer array P_i is the index of q in L_i.

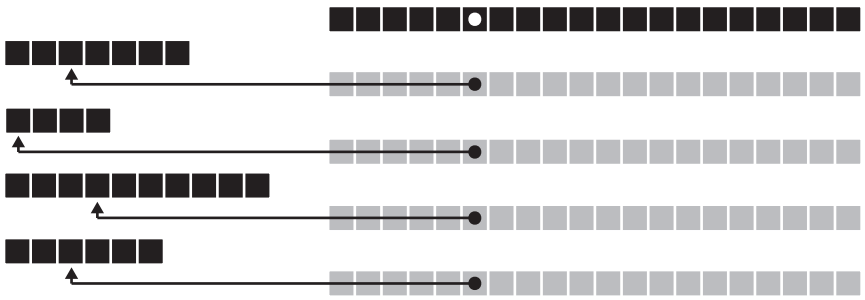

Figure 2: Unionized array structure where key is located using single binary search followed by pointer lookups.

Though this method speeds up the index search process, these pointer arrays can be quite large, and a third solution presents a compromise between memory requirement and execution time. This new data structure consists of the original k arrays, a new set of k augmented arrays, and a pointer (integer) pair for each value in the augmented arrays. The structure is designed to maintain a correspondence both between the adjacent pairs of augmented arrays and between each augmented array and the original array it is associated with.

Let $\{M_1, M_2, \ldots, M_k\}$ denote the set of augmented arrays built from the original arrays $\{L_1, L_2, \ldots, L_k\}$. Furthermore, let each value $x \in M_i$ have two pointers, p_1 and p_2, associated with it. This augmented data structure is implemented in the following manner:

(a) The final augmented array M_k is the same as the final array L_k

(b) The augmented array M_i is a sorted array containing every element in L_i and every second element in M_{i+1}

(c) The pointer p_1 associated with value x in M_i is the index of x in L_i

(d) The pointer p_2 associated with value x in M_i is either the index of x in M_{i+1} or one more than the index of x in M_{i+1}

To locate q in each of the original arrays, a binary search is performed on only the first augmented grid M_1. The pointer p_1 associated with the returned index is the index of q in L_1, and the pointer p_2 is the approximate index of q in the following augmented array M_2. To determine the true

position of q in M_2, a single comparison is made between the value at p_2 and the value at $p_2 - 1$. This process of following the pointers is repeated down to the last augmented array M_k. Figure 3 depicts the augmented array structure and lookup process.

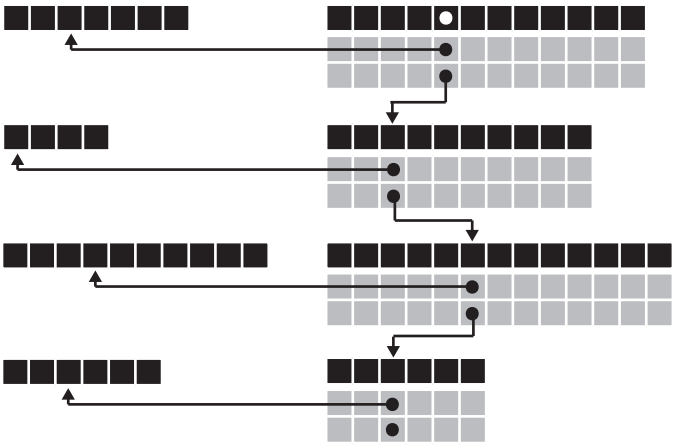

Figure 3: Augmented array structure where key is located using single binary search followed by pointer lookups.

As a simple example, consider the problem where you have the following four arrays and where you wish to perform a query for $q = 4.0$ in each array:

1.3	2.2	**4.1**	7.3	8.8
4.8	5.1	7.4		
2.6	2.8	**4.7**	8.2	
1.1	**5.2**	6.1	6.7	9.2

Using the naive method, you would simply perform a binary search on each array (returning 4.0 if it exists or the next largest value otherwise). To solve this problem using the second method, the following unionized array structure would be created:

1.1	1.3	2.2	2.6	2.8	**4.1**	4.7	4.8	5.1	5.2	6.1	6.7	7.3	7.4	8.2	8.8	9.2
0	*0*	*1*	*2*	*2*	***2***	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*4*	*4*	*4*	*5*
0	*0*	*0*	*0*	*0*	***0***	*0*	*0*	*1*	*2*	*2*	*2*	*2*	*2*	*3*	*3*	*3*
0	*0*	*0*	*0*	*1*	***2***	*2*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*3*	*4*	*4*
0	*1*	*1*	*1*	*1*	***1***	*1*	*1*	*1*	*1*	*2*	*3*	*4*	*4*	*4*	*4*	*4*

The unionized array is shown on top with the pointer arrays below in italics. In this case, you would perform a single binary search for q on U which would return the index of 4.1. You would then use the corresponding pointers to locate the key in the original arrays, finding its position to be index 2 in L_1, index 0 in L_2, index 2 in L_3, and index 1 in L_4.

Finally, the third solution involves creating the following augmented array structure:

The four augmented arrays are shown with their pointers in italics below them. An initial binary search for q is done on the first augmented array M_1, returning the index of 4.1. The first pointer,

1.3	2.2	**4.1**	4.8	5.2	7.3	8.2	8.8
0	*1*	***2***	*3*	*3*	*3*	*4*	*4*
1	*1*	***1***	*1*	*3*	*5*	*5*	*7*

2.8	**4.8**	5.1	5.2	7.4	8.2
0	***0***	*1*	*2*	*2*	*3*
1	***3***	*3*	*3*	*5*	*5*

2.6	2.8	**4.7**	5.2	6.7	8.2
0	*1*	***2***	*3*	*3*	*3*
1	*1*	***1***	*1*	*3*	*5*

1.1	**5.2**	6.1	6.7	9.2
0	***1***	*2*	*3*	*4*
0	***0***	*0*	*0*	*0*

$\mathbf{2}$, tells us the index of $\boldsymbol{q}$ in $\boldsymbol{L_1}$. The second pointer, $\mathbf{1}$, tells us the approximate index of $\boldsymbol{q}$ in $\boldsymbol{M_2}$, i.e. a binary search for $\boldsymbol{q}$ in $\boldsymbol{M_2}$ would return either $\mathbf{1}$ or one position before, $\mathbf{0}$. We perform a single comparison of $\boldsymbol{q}$ to the element at index $\mathbf{0}$ in $\boldsymbol{M_2}$ and find that $\boldsymbol{q}$ is greater, so the correct index is $\mathbf{1}$. Now we know $\boldsymbol{q}$ is located at position $\mathbf{1}$ in $\boldsymbol{M_2}$, and we look at the corresponding pointers to locate $\boldsymbol{q}$ at index $\mathbf{0}$ in $\boldsymbol{L_2}$ and at either index $\mathbf{3}$ or $\mathbf{2}$ (one less that $\mathbf{3}$) in $\boldsymbol{M_3}$. After a single comparison we find that $\boldsymbol{q}$ is at position $\mathbf{2}$ in $\boldsymbol{M_3}$. Continuing in this manner we can locate $\boldsymbol{q}$ in the remaining arrays.

For this problem, you will be provided with some data, the arrays $\{\boldsymbol{L_1}, \boldsymbol{L_2}, \ldots, \boldsymbol{L_k}\}$. The file $\boldsymbol{arrays.txt}$ contains $\boldsymbol{k = 68}$ arrays of doubles of varying length. Each line of the file contains a single array. The first value of each line is the length of the array; the remaining values are the elements of the array. Read in the arrays and implement each of the three data structures described above. Then, write a lookup kernel which for some large number of iterations at each step generates a random double between the minimum and maximum values of the arrays and locates it in each array. Conduct a performance analysis comparing the three different search techniques.