Programming Assignment III

Due on Friday, March 17, 2023 @ 2pm

100 points

This is a warm-up simple multithreaded programming assignment using **ThreadMentor**.

The Prefix Sum Problem

Given a sequence of n integers x_0, x_1, x_2, ..., x_{n-1}, the ***prefix sum*** of this sequence is another sequence

$y_0 = x_0$,
$y_1 = y_0 + x_1 = x_0 + x_1$,
$y_2 = y_1 + x_2 = x_0 + x_1 + x_2$,
...
$y_{n-1} = y_{n-2} + x_{n-1} = x_0 + x_1 + ... + x_{n-1}$.

For example, if the sequence contains the following five numbers: 1, 5, 3, 6, 8, the prefix sum is 1, 6 = 1+5, 9 = 1+5+3, 15 = 1+5+3+6, 23 = 1+5+3+6+8. This is a very simple problem and can be solved as follows:

```
y[0] = x[0];
for (i = 1; i < n; i++)
     y[i] = y[i-1] + x[i];
```

Or, if we wish to compute the prefix sum in the same array, it is simply:

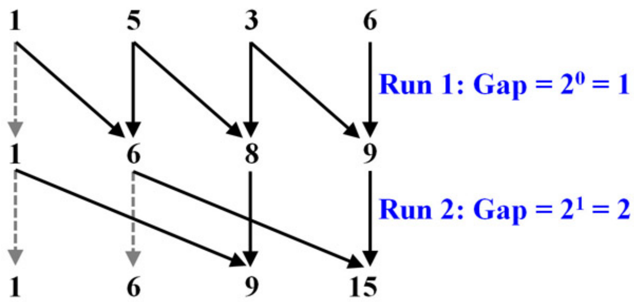
```
for (i = 1; i < n; i++)
     x[i] = x[i-1] + x[i];
```

The total number of additions is n-1, and this is an $O(n)$ algorithm.

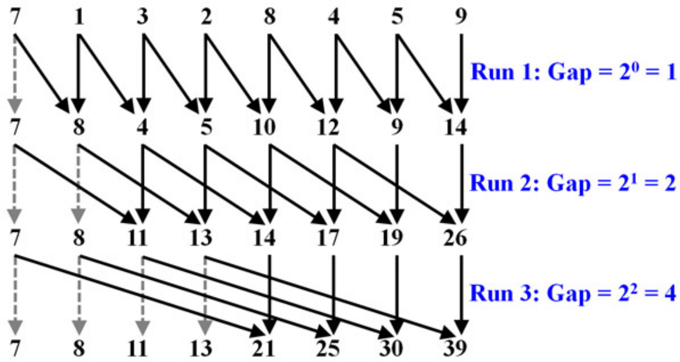
How could the prefix sum be computed in a concurrent way? Well, we do not have the proper synchronization mechanism to solve this problem so far. But, we still can solve this problem in an inefficient way with all the correct ideas in place.

For simplicity, we will assume that the number of integers is always a power of 2. That is, $n = 2^k$ for some $k > 0$. In other words, $k = \log_2(n)$.

Consider a sequence of four integers: 1, 5, 3 and 6. In this case, we have $n = 4 = 2^2$ (*i.e.*, $n = 4$ and $k = 2$). For four numbers ($n = 4$), it takes 2 ($k = 2$) runs to complete the work. The first run calculates the sum of two adjacent numbers (*i.e.*, the gap between two numbers being $2^0 = 1$). Thus, the resulting sequence is 1, 6 = 1+5, 8=5+3, 9=3+6. The second run calculates the sum of two numbers that are 2 positions away (*i.e.*, gap = 2^1). From the result of the first run 1, 6, 8, 9, the result of the second run is 1 (copied), 6 (copied), 9=1+8, 15=9+6. The following is a diagram showing the computation process. This diagram uses solid arrows to indicate the additions and light color arrows for copying.



Now consider a sequence of 8 numbers: 7, 1, 3, 2, 8, 4, 5, 9 (*i.e.*, $n = 8$ and $k = \log_2(8) = 3$). This takes three runs to complete the prefix sum. The gaps are $1 = 2^0$ for the first run, $2 = 2^1$ for the second run, and $4 = 2^2$ for the third run. For the first run, we use the adjacent elements, yielding 7, 8=7+1, 4=1+3, 5=3+2, 10=2+8, 12=8+4, 9=4+5, 14=5+9. For the second run, the gap is $2 = 2^1$ and we have 7 (copied), 8 (copied), 11=7+4, 13=8+5, 14=4+10, 17=5+12, 19=10+9, 26=12+14. For the third run, the gap is $4 = 2^2$ and the final result is 7 (copied), 8 (copied), 11 (copied), 13 (copied), 21=7+14, 25=8+17, 30=11+19, 39=13+26. The diagram below shows the details of this process



In general, if we have $n = 2^k$ numbers, k runs are needed to complete the prefix sum computation. The following is a possible sequential algorithm:

```
for (stage = 1, gap = 1; stage <= k; stage++, gap *= 2) {
     for (i = 0; i < n-1; i++) {
          if (i - gap < 0)
               x[i] = x[i];
          else
               x[i] += x[i-gap];
     }
}
```

Note that the inner loop has some copy operations which can be eliminated completely as shown below:

```
for (stage = 1, gap = 1; stage <= k; stage++, gap *= 2) {
     for (i = gap; i < n-1; i++) {
          x[i] += x[i-gap];
     }
}
```

How many additions are there in the above algorithm? Note that run 1 uses gap $1 = 2^0$, run 2 uses gap $2 = 2^1$, run 3 uses gap $4 = 2^2$, etc. In general, run h uses gap 2^{h-1}. Moreover, if the gap is 2^{h-1}, then the first 2^{h-1} elements in the array are not used, and the number of additions is $n - 2^{h-1}$. Because there are k runs in total, where $n = 2^k$, the total number of additions used is

$$(n - 2^0) + (n - 2^1) + (n - 2^2) + \ldots + (n - 2^{k-1})$$

Rearranging the terms, we have

$$k\times n - (2^0 + 2^1 + 2^2 + \ldots + 2^{k-1})$$

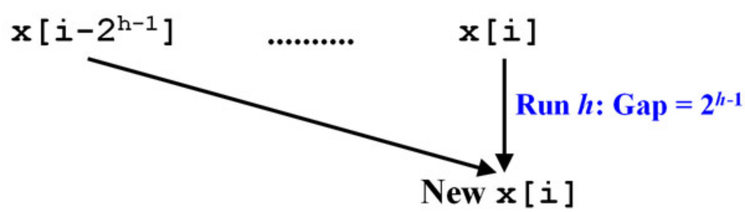
The second part is a geometric progression, which can be computed as follows:

$$2^0 + 2^1 + 2^2 + \ldots + 2^{k-1} = (2^k-1)/(2-1) = 2^k-1$$

Therefore, the total number of additions is $k\times n - 2^k + 1$. Because $n = 2^k$ and $k = \log_2(n)$, the total number of additions is $n\log_2(n) - n + 1$, and we have an $O(n\log_2(n))$ algorithm. This is slower than the simple $O(n)$ algorithm discussed at the beginning of this page. The major advantage of this slower version is that it can be made concurrent!

Making the Prefix Sum Computation Concurrent!

For run h, we need to execute n-2^{h-1} additions, one addition per pair of elements that are 2^h away. Then, why don't we assign a CPU/thread for each pair? This is exactly what we should do. We could create n threads T_0, T_1, T_2, ..., T_{n-1}. Thread T_i computes the value of $x_i + x_{i-2^{h-1}}$ for run h. This is shown in the diagram below:



Can the new value of x_i be stored back to x_i? The answer is NO! (**Why**?) To overcome this problem, we need another array to store the intermediate results. The array to be used is a 2-dimensional one of k+1 rows and n columns, where $k = \log_2(n)$: **`B[k,n]`**. Initially, we have the input elements stored in row 0 of **`B[0,*]`** (*i.e.*, **`B[0,i]`** = x_i for i = 0, 1, 2, 3, ..., n-1). Then, in run h, n threads T_0, T_1, T_2, ..., T_{n-1} are created so that thread T_i computes $x_i + x_{i-2^{h-1}}$, stores the result in **`B[h,i]`**, and exits. In this way, the last row of **`B[k,*]`** contains the prefix sum results, where $k = \log_2(n)$ is the number of needed runs.

The Algorithm:

From the above, we are able to quickly develop an algorithm to do a concurrent prefix sum computation. It is summarized as follows:

1. Suppose the input array **`x[*]`** has $n = 2^k$ numbers.
2. Prepare an array **`B[*,*]`** of k+1 rows and n columns.
3. Initialize the 0-th row of **`B[*,*]`** so that it contains the numbers of the input array **`x[*]`**. More precisely, **`B[0,j] = x[j]`** for j = 0, 1, 2, ..., n-1.
4. Iterate k times (*i.e.*, **`i`** = 1, 2, 3, ..., k). For iteration i, do the following:
   A. Create n threads T_0, T_1, T_2, ..., T_{n-1}.
   B. If **`j-2`**$^{\mathtt{i-1}}$ is less than 0, thread T_j simply copies **`B[i-1,j]`** to **`B[i,j]`**. Otherwise, thread T_j computes **`B[i-1,j]`** + **`B[i-1,j-2`**$^{\mathtt{i-1}}$**`]`** and saves the result to **`B[i,j]`**.
   C. After this, thread T_j terminates.
5. After all k iterations complete, the desired prefix sum is on the k-th row of array **`B[k,*]`**

You do not have to create n threads in every iteration. We do it in that way because we do not have the needed mechanism yet. Ignoring the repeated thread creation process, we are able to use n threads, each of which iterates $k = \log_2(n)$ times, to compute the prefix sum of n numbers. If each thread is considered as a CPU, this algorithm means that we are able to use n CPUs to compute the prefix sum of n numbers, and each CPU only iterates $k = \log_2(n)$ times (*i.e.*, $O(\log_2(n))$). This is fast because $O(\log_2(n))$ is faster than $O(n)$. For example, if we have 1024 = 2^{10} numbers, the sequential algorithm requires 1024-1 = 1023 additions on a single CPU while the concurrent one requires 1024 CPUs each of which executes only 10 = $\log_2(1024)$ additions!

Program Specification

Write a program (*i.e.*, the **`main`**) to read in n integers into the array **`x[*]`**, and initialize the array **`B[*,*]`** by copying **`x[*]`** to the 0-th row of **`B[*,*]`**. Then, iterates $k = \log_2(n)$ times. In each iteration, the **`main`** creates n threads as discussed in the previous section and waits for all n threads to exit. Finally, the **`main`** prints out the last (*i.e.*, k-th) row of array **`B[k,*]`**.

Here are a few notes:

- The input array should be read in from **`stdin`**.
- The value of n is always a power of 2 (*i.e.*, $n = 2^k$ for some $k > 0$).
- Array **`B[*,*]`** is global, but array **`x[*]`** is not.
- You can only use the above program structure and the indicated thread creation and thread join. **No other thread and/or process functions can be used for this program.**

Input and Output

The input to your program should be taken from **`stdin`**. Your executable must be named **`prog3`**. The command line looks like the following, where **`input-filename`** is a file from which **`prog3`** reads in the input values:

```
./prog3 < input-filename
```

The input file has the following format, where n is an integer of form 2^k for some integer $k > 0$, and $x_0, x_1, ..., x_{n-1}$, are n integers. You may assume all input values are correct so that you do not have to do error checking.

$$
\begin{array}{l}
n \\
x_0 \quad x_1 \quad x_2 \ \ldots \quad x_{n-1}
\end{array}
$$

Suppose the command line is

```
./prog3 < in.txt
```

and the file `in.txt` has the following lines:

```
8
7 1 3 2 8 4 5 9
```

Some valid test input files are available from the course web page.

Then, your program output should look like the following:

```
Concurrent Prefix Sum Computation              // from main()

Number of input data = 8                       // from main()
Input array:                                   // from main()
   7   1   3   2   8   4   5   9               // from main()
                                               // each number occupies 4 positions
                                               // there has to be k = log2(n) runs
                                               // from main(), do it for each run
Run i:                                         // from main(), run i
     ..........
     Thread j Created                          // from thread j
     ..........
     Thread j computes x[j] + x[j-2^(i-1)]     // from thread j
                                               // thread j fills in the values of
                                               //     j and j-2^(i-1)
                                               //     fill in the values of j and j-2^(i-1)
     Thread j copies x[j]                      // thread j copies if no computation needed
     ..........
     Thread j exits                            // thread j exits
     ..........
Result after run i:                            // from main()
  aa bb cc dd ee ff gg hh                      // from main()
                                               // use the input array format

Final result after run k:                      // from main()
   7   8  11  13  21  25  30  39               // from main()
                                               // use the input array format
```

In the above sample output, the **`main()`** prints out the input and all intermediate result arrays. The **`main()`** iterates **`k`** = $\log_2$(**`n`**) times. Iteration **`i`** uses gap length 2^{i-1}, where **`i`** = 1, 2, ..., **`k`**. Thus, the gaps are 1, 2, 4, 8, 16, ..., $\log_2$(**`n`**)-1. All lines printed by the **`main()`** starts on column 1, and each data value is printed with 4 positions.

In each iteration, **`main()`** creates **`n`** threads, each of which does the following: **(1)** prints a message indicates that it has been created, **(2)** prints the two entries this threads has to add (*e.g.*, **`x[5]+x[3]`** for run 2), and **(3)** terminates. All output from a thread starts on column 6.

Submission Guidelines

General Rules

1. Use ThreadMentor (TM) to complete this assignment. Example TM code and makefile are available in ~jmayo/public/cs3331/tm/.
2. All programs must be written in C++.
3. Submit your code through Canvas.
4. Unix filenames are case sensitive, `THREAD.cpp`, `Thread.CPP`, `thread.CPP`, etc are *not* the same as `thread.cpp`

5. We will use the following approach to compile and test your programs:

```
make noVisual      <-- make your program w/o visulization
./prog3 < in.txt   <-- test your program
```

   This procedure may be repeated a number of times with different input files to see if your program works correctly. Note that the binary created by your makefile must be named prog3.

6. Your implementation should fulfill the program specifications as stated. Any deviation from the specification may cause you to lose a significant number of points.
7. Include a `README` that indicates the number of slip days you are using

Program Style and Documentation

It's important for our course TA to look at your code to identify problems that may not appear from the program output. Otherwise, you may believe you have a good solution to the problem, but instead there is a significant misunderstanding. To help our TA, please use the following guidelines. Helping the TA understand your code will help you get a better grade!

1. For each file, the first piece should be a program header to identify yourself like this:

```
// -----------------------------------------------------------
// NAME : John Smith                         User ID: xxxxxxxx
// DUE DATE : mm/dd/yyyy
// PROGRAM ASSIGNMENT #
// FILE NAME : xxxx.yyyy.zzzz (your unix file name)
// PROGRAM PURPOSE :
//    A couple of lines describing your program briefly
 // -----------------------------------------------------
```

   Here, User ID is the one you use to login. It is *not* your social security number nor your M number.

   For each function in your program, include a simple description like this:

```
// -----------------------------------------------------------
// FUNCTION  xxyyzz : (function name)
//     the purpose of this function
// PARAMETER USAGE :
//    a list of all parameters and their meaning
// FUNCTION CALLED :
```

```
//    a list of functions that are called by this one
// ---------------------------------------------------------
```

Program Specifications

Your program must follow exactly the requirements of this programming assignment. The following is a list of potential problems.

1. Your program does not use the indicated algorithms/methods to solve this problem.
2. Your program does not follow the structure given in the specifications. For example, your program is not divided into the functions and files indicated in the specification.
3. Incorrect output format. This maay cost you some points depending on how serious the violations are. The grader will make a decision. Hence, carefully check your program output against the required one.
4. Your program does not achieve the goal of maximum parallelism.

Program Correctness

If your program compiles and runs, we will check its correctness. We normally run your program with several sets of input data, one posted on this programming assignment page (the public one) and the others prepared by the grader (the private ones). You program must deliver correct results for all data sets. Depending on the seriousness of the problem(s), a significant deduction may be applied.

The `README` File

Submit a `README` that answers the following questions:

1. Are there any race conditions in this prefix sum computation? Why?
2. Can the result of `x[i]+x[i-2`$^{h-1}$`]` of run `h` be saved back to `x[i]`? Explain your findings as clearly as possible.
3. The `main()` creates `n` threads in each iteration and wait for them to complete. There is a significant amount of time in creating and joining threads. If you are allowed to use extra variables/arrays and busy waiting, can you just create `n` threads and let them do all the work without the use of a temporary array `B[*,*]`? Suggest a solution and discuss its correctness.
4. Prove rigorously (proof by induction, not by example) that this algorithm does compute the prefix sum correctly.

You should elaborate in your answers and provide details. When answering the above questions, make sure each answer starts with a new line and have the question number (*e.g.*, Question X:) clearly shown. Separate two answers with a blank line. This will make it much easier for the TA to grade the papers.

`README` must be a plain text file. We do not accept files produced by any word processor. Moreover, watch for very long lines. More precisely, limit the length of each line to no more than 80 characters with the Return/Enter key for line separation. Missing this file, submitting non-text file, file with long lines, or providing incorrect and/or vague answers will cost you many points. Suggestion: Use a Unix text editor to prepare your `README` rather than a word processor. Many editors will export a text file.

Final Notes

1. Your submission should include the following files:
    - File `thread.h` that contains all class definitions of your threads.
    - File `thread.cpp` contains all class implementations of your threads.
    - File `thread-main.cpp` contains the main program.
    - File `Makefile` is a makefile that compiles the above three files to an executable file `prog3` without visualization. Your makefile **should** make sure all paths are correct. Do not assume the grader knows your local path!
    - The `README` file.

   Note also that without following this file structure your program is likely to compile but not run, and, as a result, you may get low grade. Therefore, before submission, check if you have the proper file structure and a correct makefile.