Parallel Programming Languages & Systems
Exercise 1

This exercise sheet accounts for 20% of the course final mark. The deadline for electronic submission is ***12:00 (noon) on Thursday 14th March 2024****. You must use the filename as indicated in the exercise. Your program must compile and execute correctly on DICE, using the commands described here.* ***It is your responsibility to check this.***

Good Scholarly Practice:
Please remember the good scholarly practice requirements of the University regarding work for credit. You can find guidance at the School page

https://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct

This also has links to the relevant University pages.

You will write a C program, parallelized with Pthreads, which implements a parallel algorithm for the prefix sum problem. Good marks will be awarded for following the specification, for appropriate use of Pthreads operations and for general clarity of code, presentation (for example appropriate commenting) and discussion. Your program will be tested on a normal DICE workstation. Depending upon the specification of the workstation you run on, you may not see any real performance improvement (for example, if it only has one core!). This doesn't matter – the goal here is to correctly implement the given algorithm.

Parallel Prefix Sum

Given a sequence of integers, its prefix sum is a sequence of the same length in which the i^{th} value is the sum of the first i values of the original sequence. For example, the prefix sum of the sequence $2, 3, 0, 2, 4, 8$ is $2, 5, 5, 7, 11, 19$. Computing a prefix sum sequentially is trivial: we simply loop over the sequence, adding items and storing partial results as we go along.

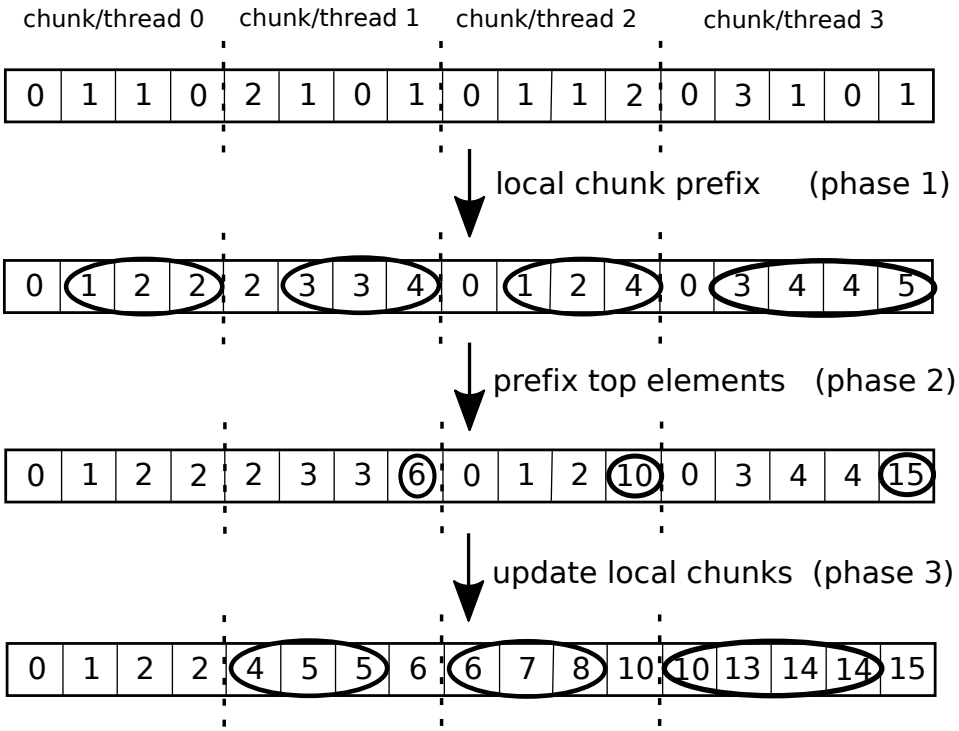
However, it is also possible to compute a prefix sum in parallel, using shared memory and threads. Given an n item sequence stored in an array, and p worker threads, each thread takes responsibility for a roughly equal-sized chunk of the sequence. The algorithm has three phases. In the first and third phases only the responsible worker thread for a chunk will write new values to its chunk. In the second phase only worker thread 0 is active, and makes updates at various places in the whole array. All updates are made in place (i.e. there no auxiliary arrays or copies made). The phases of the algorithm are as follows.

1. Every worker thread performs a prefix sum sequentially across its chunk of data, in place.

2. Worker thread 0 performs a sequential prefix sum *using just the highest indexed position from each chunk*, in place. Other worker threads simply wait.

3. Every worker thread (except thread 0) adds the final value from the preceding chunk into every value in its own chunk, except the last position (which already has its correct value after phase 2), in place.

The figure below illustrates the contents of the array as they are changed by each phase, for an example with an array of length 17 and 4 threads. The dashed lines indicate the boundaries between chunks (threads 0, 1 and 2 have chunks of size 4, while thread 3 has a chunk of size 5). The ovals indicate the positions which have been updated during the previous phase. There is another example at the end of this document.



The Exercise

You must

1. Implement a C with Pthreads version of the prefix sum algorithm described above.
2. Write a brief explanation of what you have done, as a multiline comment in your C source file.

In order to avoid wasting time with I/O in C, and in order to make the marking task a little easier, you are provided with a template program which you should expand. This can be copied from the course Learn site, where it is called `ex1Starter.c`. **You should call your copy of the file** `BXXXXXX.c`, where `BXXXXXX` is your examination number. The starter file as provided contains a program which creates some random data and computes its prefix sum sequentially. It then calls the parallel prefix sum function on a copy of the same random data. Finally, it checks whether the two results agree. Since the parallel prefix sum function is currently empty, the results don't agree and the program reports an error. You can compile and test the program as follows, on DICE (the data will be generated randomly for each run)

```
[mic]: gcc -o BXXXXXX BXXXXXX.c -DNITEMS=10 -DNTHREADS=4 -DSHOWDATA=1 -lpthread
[mic]: ./BXXXXXX
initial data          :  1 3 4 2 4 4 1 4 4 2
sequential prefix sum :  1 4 8 10 14 18 19 23 27 29
parallel prefix sum   :  1 3 4 2 4 4 1 4 4 2
Error: The sequential and parallel prefix sum arrays don't match.
```

As you can see, three values are defined in the compilation command: `NITEMS` is the size of arrays to create, `NTHREADS` is the number of threads to create, and `SHOWDATA` controls whether or not the arrays are displayed (value `1`) or not (value `0`). You might want to switch off display when testing with large arrays.

Writing the Program

Your task is to implement the function called `parallelprefixsum`. You can add other functions and definitions as you choose, but **do not change any of the other code provided**. Your `parallelprefixsum` function (and/or any helper functions you write)

- should create and run `NTHREADS` worker threads, which implement the algorithm described above. `NTHREADS` does not include the original main thread, which plays no role in the core of the algorithm (it should simply create the worker threads, then wait for them to complete). The worker threads should be created once for the entire algorithm, not recreated for each phase, or any other such scheme.
- should work for any value of `NITEMS` such that `NITEMS >= NTHREADS`
- should use chunks which are roughly equal in size, in the sense that each thread should have a chunk of size at least $\left\lfloor \frac{NITEMS}{NTHREADS} \right\rfloor$ (i.e. the floor function of `NITEMS` divided by `NTHREADS`). Surplus items can be handled as you choose - an obvious scheme, as in the examples in this document, is to assign them to the final chunk. For example, with 3 threads and 11 items, the first two chunks would have 3 items and the last chunk would have 5 (i.e. $\left\lfloor \frac{11}{3} + 2) \right\rfloor$). With 3 threads and 101 items, the first two chunks could have 33 items and the last chunk could have 35 (i.e. $\left\lfloor \frac{101}{3} + 2) \right\rfloor$).

Multiple Argument Threads

As we have seen in lectures, one of the challenging features of Pthreads is that a thread can only be passed a single argument when it is created. You may find that you want to pass your threads more than one argument. The file `multiArgumentThreads.c` shows how this effect can be achieved, by creating a packet of arguments, passed to the thread via a pointer.

Writing the Discussion

Having completed the program, you should write a short description (of around half a page) explaining your implementation strategy. You should explain how you created threads, how,

where and why you synchronized them, and any other interesting features of your implementation. This discussion should be written as a multi-line comment in your source file.

Submitting your Work

Submit your work on Learn as a single file called `BXXXXXX.c` where `BXXXXXX` is your exam number.

Testing your Code

Your code will be tested on a regular DICE machine, by compiling and running it for a range of values of `NITEMS` and `NTHREADS`. Notice that the code as provided for you will exit the program if run with values which are too large - your code won't be tested with values outside these bounds.

Another Example

Here is another example of the algorithm in action, this time for 3 threads and an array of length 10. Each thread takes responsibility for a chunk of size 3, except thread 2 which has a chunk of size 4.