COS597C: Assignment 2

Problem 1

1 Submission Instructions

Email a SINGLE tgz containing EVERYTHING that you wish to submit.
To:rarun@princeton.edu, sinha@princeton.edu
Cc: august@cs.princeton.edu, dpw@cs.princeton.edu
Subject line should start with "COS597C:".

2 Problem 1

A sequential C implementation of the quicksort algorithm (quicksort.c) and a client of the sorting routine (client.c) are provided.

Your task is to implement a parallel version of the quicksort routine using pthreads by filling in the skeleton method in quicksort_par.c (see algorithm below). A good pthreads primer is available at
https://computing.llnl.gov/tutorials/pthreads/
Your implementation will be tested against a different client (similar to the provided client) that will sort pseudo-randomly generated arrays of different sizes. Note that the provided client program does NOT check the correctness of your sorting routine. To build the client with the sequential sorting routine, type "make seq". This creates the "client_seq" program. To build the client with the parallel sorting routine, type "make par". This creates the "client_par" program. Both of these programs have the following command line options:

- -t [num threads] : optional argument (positive integer, default is 1)
- -n [number of array elements] : optional argument that can be varied to sort arrays of different sizes (positive integer, default is 50000000)
- -h : Display help

An execution of the client will output the wall clock execution time of the sorting routine in seconds (measured using `clock_gettime`). Use this number to compute speedups.

Submit quicksort_par.c along with a report containing the following sections.

2.1 ALGORITHM

Quicksort identifies a pivot element in the array, and partitions the array into subarrays such that the first partition contains all elements smaller than the pivot while the second partition contains all elements greater than or equal to the pivot. Then quicksort is recursively called on each of the two subarrays. One way to parallelize the quicksort algorithm is to assign each of the recursive calls to quicksort to a different thread. You are expected to implement this parallel algorithm.

The primary impediment to getting "good" speedup is load imbalance among the threads. To understand this, consider the `getPivot` implementation provided. It uses the middle element of the array as the pivot. Using this element as the pivot may not partition the array into equally sized subarrays. Consequently, a naive parallelization using the provided `getPivot` implementation would be suboptimal. Your parallel algorithm should address this problem of determining how to keep a given number of hardware threads (specified via the [num threads] argument to client_par) busy in a useful way to maximize speedup.

Provide pseudo code for your parallel algorithm (including the `getPivot` heuristic in case you modify it) and give an informal proof of its correctness. Provide an analysis of the time complexity of the sequential and parallel algorithms using asymptotic notation.

2.2 EXPERIMENTAL RESULTS

In this section, you should compare the actual running times to their theoretical complexity. You are free to use any machine with a sufficient number of hardware contexts (for example, 8). Provide details of the machine in your report. Those who do not have access to a suitable machine can use c2.cs.princeton.edu. It has 8 hardware contexts.

Show:

1. Speedup (y-axis) vs. #threads (x-axis): For the default data size, show the speedup for the parallel version as you vary the number of threads (1,2,4,6,8,...). The speedup should be calculated with respect to the sequential implementation. Also, report the execution time of the sequential implementation.

2. Speedup (y-axis) vs. data size: For #threads equal to eight (or maximum on your machine), show the speedup over sequential implementation by varying n (size of array) by orders of magnitude between 5000 and 500000000.

For the two experiments above, compare the theoretically expected performance and the experimentally obtained performance.

COS597C: Assignment 2

Problem 2

1 Submission Instructions

You can submit your answer to this problem on paper in class on October 14 or email a soft copy as part of the single tgz for this assignment (see submission instructions for Problem 1 above).

2 Problem 2

2.1 Serializability

There are several properties which a programmer desires in a parallel program, such as serializability, linearizability, etc. These properties are helpful in reasoning about the correctness of the program under arbitrary scheduling of threads. The following problem attempts to give a deeper insight into the property of serializability.

A history can be defined as the sequence of accesses of global shared variables (you can think of history as a trace which logs the accesses made to shared variables during the execution of the parallel program). For instance,

$$h = W_1[x,y]\ R_1[y]\ R_2[y]\ W_2[x].$$

In this history, the global accesses of two atomic sections (often referred to as transactions) are recorded. W and R stand for write and read respectively. The subscripts for W and R stand for the transaction indices. The variables inside the square brackets denote the shared variable accesses.

The first transaction (T_1) writes variables x and y and then reads variable y. Similarly, the second transaction (T_2) reads y and writes x. (Assume that no two transactions are executed in the same thread.) Observe that history h is serial since the accesses do not overlap.

Often, the given history is not serial but still it is serializable. To show that a given history is serializable, it is necessary to show that it is equivalent to a serial history. Two histories are equivalent iff there exists a sequence of steps, where on each step the positions of two adjacent accesses can be swapped. The permitted swapping operations between two adjacent accesses are the following.

1) Two adjacent reads.
2) Two adjacent read and write accesses with disjoint sets of variables being accessed.
3) Two adjacent writes with disjoint sets of variables being accessed.

Consider another history (*h'* say).

$$h' = R_1[x]\ R_2[x]\ W_1[x]\ W_2[y].$$

As an illustration: $h' = R_1[x]\ R_2[x]\ W_1[x]\ W_2[y]$

$\Xi\ R_1[x]\ R_2[x]\ W_2[y]\ W_1[x]$ (W_1 and W_2 are swapped)

$\Xi\ R_2[x]\ R_1[x]\ W_2[y]\ W_1[x]$ (R_1 and R_2 are swapped)

$\Xi\ R_2[x]\ W_2[y]\ R_1[x]\ W_1[x]$ (R_1 and W_2 are swapped)

Therefore, h' is equivalent to a serial history and hence serializable.

We consider history *h''* which is not serializable.

$$h'' = R_1[x]\ W_2[x]\ W_2[y]\ R_1[y]$$

We prove that *h''* is not serializable by contradiction. Assume *h''* is serializable. Then there are two possibilities:

Case 1: (T_1 precedes T_2). In that case *h''* should be equivalent to h_s'' where,

$$h_s'' = R_1[x]\ R_1[y]\ W_2[x]\ W_2[y]$$

But, $R_1[y]$ and $W_2[y]$ cannot swap positions in h''. Therefore, *h''* cannot be equivalent to h_s''.

Case 2: (T_2 precedes T_1). In that case *h''* should be equivalent to h_s'' where,

$$h_s'' = W_2[x]\ W_2[y]\ R_1[x]\ R_1[y]$$

But, $R_1[x]$ and $W_2[x]$ cannot swap positions in h''. Therefore, *h''* cannot be equivalent to h_s''.

Hence, h'' is not serializable.

Which of the following histories are serializable? Justify your answer. [If serial, show the sequence of swaps among the adjacent accesses leading to a serial history. If not, assume the history is serializable and try to prove by contradiction.]

1. $h_1 = R_1[x]\ W_2[x]\ R_1[y]$.

2. $h_2 = R_1[x]\ W_2[x,y]\ R_1[y]$.

3. $h_3 = R_1[x]\ R_2[x]\ W_1[x,y]\ W_2[y]$.

4. $h_4 = R_3\ [x]\ W_1\ [x]\ R_2\ [y]\ W_3\ [y]$

2.2 Memory Consistency Models – Constraint Graph Modeling

Memory consistency model determines the ordering of memory operations. The result of multi-threaded program is dependent on this order of the memory operations. Consider the following example. This code attempts to ensure mutual exclusion assuming SC (sequential consistency). A legal execution trace is:

(1.1)→(1.2) →(2.1) →(2.2).

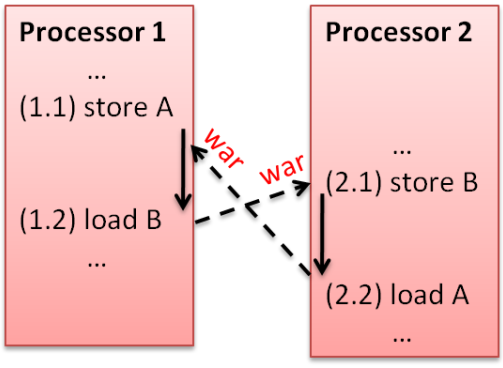
Therefore, only Thread 1 enters critical section. However, another possible (illegal) execution trace that violates SC is:

(1.2)→(2.1)→(2.2)→(1.1).

In this execution, both the threads may enter the critical section simultaneously.

A violation of the assumed memory consistency model can be detected by constructing a constraint graph (CG). A constraint graph is a directed graph that models memory ordering constraints and dependence constraints for a given trace. The vertices in CG are defined by the accesses made to the global shared variables. An intra-thread edge (u, v) in CG denotes that event u must happen before event v. A directed inter-thread edge (u,v) in CG denotes the dependence edges (read-after-write (RAW), write-after-read (WAR), write-after-write (WAW)) such that u happened before v in the trace. A cycle in CG implies violation of memory consistency model.

In the aforementioned example, for the illegal trace ((1.2)→(2.1)→(2.2)→(1.1)) a cycle is found in its CG (load = read, store = write).



We introduce another popular memory consistency model used in Sun SPARC – Total Store Order (TSO). TSO can be characterized as follows:

```
Processor-1
    …
(1.1) A = 1;
(1.2) if (B == 0)
    {
      // critical section
      …
    }
```

```
Processor-2
    …
(2.1) B = 1;
(2.2) if (A == 0)
    {
      // critical section
      …
    }
```

1. All instructions commit in program order, except "write u; read v" can commit as "read v; write u".
2. Commit of write can be delayed.
3. "write u; write v" cannot be re-ordered. "write u; read u" cannot be re-ordered.

Construct the constraint graph for the following program execution assuming TSO and check if there is any cycle. (Hint: Vertices are already given. You have to draw the inter- and intra-thread edges as defined earlier)