Asynchronous Algorithms

Guy Blelloch

April 20, 2009

These are notes from 3 lectures in the class 15-499 on parallel algorithms.

1 Introduction

So far we have considered algorithms for which for the most part the results do not depend on how the tasks are scheduled. In particular except for some limited use of concurrent writes we have not allowed any race conditions. Without race conditions the results of parallel instructions do not depend on each other so the interleaving generated by the scheduler or by processor delays do not affect what the instructions return. Such "deterministic" parallelism makes it easier to analyze the correctness and timing of algorithms. In particular we need only consider a single interleaving to prove correctness.

In some situations, however, it is necessary to write algorithms for which the results depend on how instructions on parallel threads are interleaved. Such *asynchronous* algorithms are often necessary when programming interactive systems where requests are coming from the "outside world". They are also often required at a low-level to implement the run-time systems needed to implement deterministic parallelism.

In analyzing the correctness of asynchronous algorithms one needs to consider all possible outcomes that might arise from different rates of progress among the processes. To simplify this analysis one typically assumes that individual instructions take place atomically allowing one to consider the sequential semantics under all possible interleavings of the instructions. Even with this simplification the analysis can be very difficult since there can be an exponential number of interleavings. Furthermore this assumption is not always true in practice. It is possible on some real machines, for example, for one processor to write a value into location l_1 and then location l_2, and for a second processor to see location l_2 written first and then location l_1. There is no interleaving of instructions from the processors that leads to a sequential semantics that is consistent with this behavior. Because of these issues, asynchronous algorithms are and notoriously difficult to get correct and one needs to be very careful about being precise in defining the semantics of the underlying system.

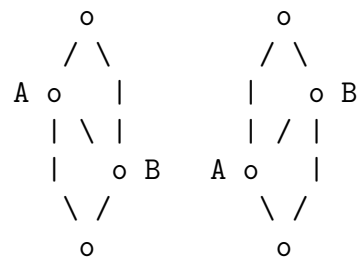
Beyond issues of correctness there is also the question of running time. Even if we assume there are no deadlocks, there is the potential for live locks (where a processor keeps trying to get a resource but never gets it) or simply the question of comparing the asymptotic running

time of two algorithms. Given that we assume that different processors can make different rates of progress, how do we analyze the time taken by an algorithm or an operation?

In the following discussion we assume that all processes are interacting through a shared memory. In the simplest form we will assume a fixed number P processes, but more generally we will describe a computation as a dependence graph, as we have for deterministic algorithms. However, unlike deterministic algorithms, we allow arbitrary race conditions—parallel tasks can write to and read from shared memory locations. Because of these race conditions the DAG itself can depend on the interleaving of instructions. Consider the following example:

parallel do
  lock(x); A; unlock(x);
  lock(x); B; unlock(x);
end parallel do

where the **Parallel do** means that each line executes in parallel with the other lines. It can lead to two DAGS depending on which of the two parallel tasks grabs the lock first.



It is easy to create examples where one interleaving creates a DAG of depth $O(1)$ and another creates a DAG of depth $O(n)$.

2 Interleavings and sequential consistency

Consider the following code where := is the assignment operator.

$x := 0$;
parallel do
  $y := 0$; $x := 2$;
  $z := 0$; $x := 1$;
end parallel do

We can ask what are valid outcomes of this code. Which of the following outcomes for the final values of the three variables seem reasonable?

	x	y	z
a	0	0	0
b	1	0	0
c	2	0	0
d	2	1	0
e	2	2	0
f	1	1	2

Case a might not seem like a reasonable outcome since either the 1 or the 2 should be written. Case e might not seem like a reasonable outcome since it would imply 2 is written to x before x is written to y, which is out of order with respect to the first line. What about case f?

To get a handle on what are valid outcomes we first consider the notion of sequential consistency. Later we will look at related notions of linearizability and serializability. All these notions are based on the idea of that the only valid outcomes correspond to some interleaving of the basic operations. Such interleavings imply two things: (1) each of the operations appear to take place atomically (all at once) and (2) that certain ordering constraints, such as the order within a process, are maintained.

Consider a shared state S that supports a set of dynamic operations O each of which takes an argument and returns a result and possibly updates the state: $o : S \times A \rightarrow S \times R, o \in O$. The states we will consider will mostly be data types such as a stack with push and pop operations, a dictionary with search, insert and delete, or simply a memory cell with read and write. We refer to an operation along with its arguments and return values as an *operation instance*. For an operation `op` with arguments `args` and no return values we will use the notation `op(args)` and if it also has return values `rets` we will use `op(args) -> rets`.

A *sequential history* is a total order (sequence) of operation instances.

A sequential history is *legal* (or consistent) if it obeys the semantics of all data types that are involved.

For example for a stack:

```
H: push(3); push(5); pop() -> 5; pop() -> 3
```

is consistent, but

```
H: push(3); push(5); pop() -> 3; pop() -> 5
```

is not since the elements are popped in the wrong order.

A *parallel history* is a partial order (DAG) of operation instances. The partial ordering represents any ordering constraints among parts of the computation. The idea is that we will only need to consider interleavings that are consistent with that partial order. The simplest form of parallel history we will consider is a set of P processors for which the operations are totally ordered within each processor but there is no ordering among processors. We will

refer to this as a *simple parallel history* and most of the examples in this section will be of this form. A *total ordering* of a parallel history (simple or not) is any sequential history that is consistent with the partial ordering (i.e. any topological sort of the DAG).

Definition 1 *A parallel history H is* sequentially consistent *if there exists a legal total ordering of H.*

In the example at the beginning of this section the outcomes b, c, d are sequentially consistent but a, e, f are not.

Sequential consistency is a global property of a history. In particular, just because the operations on each data object in a history are sequentially consistent does not imply that when looked at together the operations are sequentially consistent. The definition is therefore more useful for describing a property of a set of built-in primitive operations (e.g. memory operations) than it is in proving correctness of the implementation of operations on data types. We will return to this later, but for now will consider how sequential consistency can be used to implement some simple algorithms using memory operations.

Given a sequentially consistent memory what can we do with it. It turns out that with just read and write operations implementing many basic operations such as locks, stacks, queues inefficient, difficult, or even impossible under certain assumptions. Because of this many machines have been augmented with additional atomic memory operations. Here we consider three such operations: TestAndSet, FetchAndAdd, Swap, and CompareAndSwap. The CompareAndSwap is also known as CAS or Compare & Exchange. The integer versions of these operations are defined as follows:

```
int TestAndSet(int *p) {
  int r = *p
  if (*p == 0) then *p=1;
  return r;}

int FetchAndAdd(int *p, int v) {
  int r = *p;
  *p = r + v;
  return r; }

int Swap(int *p, int v) {
  int r = *p;
  *p = v;
  return r; }

int CompareAndSwap(int *p, int v1, int v2) {
  int r = *p;
  if (*p == v1) then *p = v2;
  return r; }
```

We note that all of these operations have a similar form and they all fall into the class of Read-Modify-Write operations. Also note that TestAndSet is really a special case of CompareAndSwap where `v1` is 0 and `v2` is 1.

One might first ask how hard is it to implement these operations with just reads and writes. It is possible to do this by implementing a lock and then executing the instructions under a lock. This, however, is inefficient and as discussed later can lead to problems if a processor stops while executing the code in the lock. Having built-in atomic versions of these operations that are sequentially consistent greatly simplifies the design of asynchronous algorithms.

Our first example is how to implement a FetchAndAdd. First lets consider using the code given above. In analyzing the possible outcomes we have to consider the possibility that many concurrent threads are executing a FetchAndAdd on the same location. The instructions of these threads might be interleaved in arbitrary ways in the total ordering. Consider the following code:

```
v = 0;
In parallel do:
  x = FetchAndAdd(&v,1)
  y = FetchAndAdd(&v,1)
```

If the instructions are interleaved in the following ordering:

```
P1: int r = *p;  (r = 0)
P2: int r = *p;  (r = 0)
P1: *p = *p + r; (*p = 1)
P2: *p = *p + r; (*p = 2)
```

the results are inconsistent with any sequential ordering of the two calls to FetchAndAdd since both calls will return 0. We now consider a correct implementation that uses a CompareAndSwap.

```
int FetchAndAdd(int *p, int v) {
  int o;
  do {o = *p;}
  while (CompareAndSwap(*p,o,o+v) != o);
  return o; }
```

What makes this implementation correct is that the CompareAndSwap will only change the value in `*p` in the case that it has the old value `o`. If a second process interleaves updates to the location `*p` between reading it and the CompareAndSwap then the CompareAndSwap will fail and return the new value instead of `o` and the do loop will be repeated. The loop will only complete if the CompareAndSwap properly replaces `o` with `o+v` in `*p`. One way to argue the correctness of the algorithm is to note that as long as the CompareAndSwap fails there is no side effect, but than when it succeeds the operation appears like

it happened atomically at the point of the CompareAndSwap. In particular if we look at any interleaving of instructions it will always be consistent with a sequential ordering of the FetchAndAdd operations where they took place at the point of the CompareAndSwap. We will formalize this better in the next section.

One property of this algorithm is that it can be the case that a processor loops forever always failing on the CompareAndSwap. This is sometimes called live-lock. We will look at this later.

Our next example is how to implement a lock using FetchAndAdd.

```
int serving = 0;
int ticketNum = 0;

void Lock() {
  ticket = FetchAndAdd(ticketNum,1);
  while (ticket > serving); }

void Unlock() { serving = serving + 1;}
```

This algorithm works like the tickets often used at bakeries. At such bakeries the customer picks up a ticket from a dispenser at the entrance. Each ticket has a number on it that is one more than the previous one that was dispensed. On the wall is a counter which says which number is currently being served. As soon as the customer is finished being served the counter is increased and the customer with the next number is served. Only one person is served at a time since only one person has each number. In the code the variable `serving` specifies who is currently being served (inside the lock) if any, and the variable `ticketNum` specifies the next number available at the dispenser. Inside the lock code the `ticket` variable indicates the ticket number for the specific user.

Unlike the implementation of FetchAndAdd using CompareAndSwap this algorithm is somehow fair. It is not possible for a user to wait inside the while loop indefinitely while other users enter the lock and leave it. In particular the "customers" are served in the order in which they arrive (pick up their tickets). On the other hand if a processor gets stopped in a lock it can block any progress.

We can compare the algorithm above with a version of the Bakery algorithm by Leslie Lamport given in Figure 1. This version only uses reads and writes to memory, but it is more complicated and much less efficient. This algorithm allows multiple users to have the same ticket (kept in `ticket[i]` for user `i`) but breaks ties among users by the user id—in particular if two users have the same ticket number then the one with a smaller i has precedent.

3 Performance

We have already mentioned the concept of fairness and the ability to always make progress. We typically want some way to know that an asynchronous algorithm is making progress

```
int flag[P] = 0;
int ticket[P] = 0;

void Lock() {
  int j,maxV=0;
  i = threadID();
  flag[i] = 1;
  for (j=0; j<P; j++) maxV = max(maxV,ticket[j]);
  ticket[i] = MaxV + 1;
  for (j=0; j<i; j++) while(flag[j] && (ticket[j] <= ticket[i]));
  for (j=i+1; j<P; j++) while(flag[j] && (ticket[j] < ticket[i]));
}

void Unlock() { flag[threadID()] = 0;}
```

Figure 1: Lamport's Bakery algorithm using just reads and writes.

even if we can't specifically bound the time. Such progress criteria include the following:

Deadlock free: Some operation must finish in a finite number of steps if all threads have the opportunity to work.

Lockout free: (Also starvation free:) All operations must finish in a finite number of steps if all threads have the opportunity to work.

Lock free: Some operation must finish in a finite number of steps even if other threads are stopped.

Wait free: All operations must finish in a finite number of steps even if other threads are stopped.

These criteria can actually be organized along two dimensions, one having to do with fairness (whether all operations need to make progress or just some of them), and the other with blocking (whether operations can make progress even if other threads are stopped).

	Blocking	**Non-Blocking**
Not-Fair	deadlock-free	lock-free
Fair	lockout-free	wait-free

The blocking criteria can be important since threads delays can happen in practice when the operating system or scheduler decides to context switch a thread out in exchange for another. If the thread that is being swapped out is holding a lock then the swap will block all other threads that try to acquire that lock until it is swapped back in. If the scheduler is not designed carefully this can actually cause deadlocks. Such swapping out of threads

during critical sections can often be avoided when holding a lock for a short period. This can be done by, for example, waiting until the thread leaves the critical section before scheduling another thread.

We consider one more criteria:

Obstruction free: An operation must finish in a finite number of steps if it is the only one running.

This criteria is weaker than lock-free but not comparable to lockout-free. If all processes die except one, then obstruction free says that that one process must make progress. This is not true with lockout-free since if a process dies during a critical region it could stop another from making progress. On the other hand if even two processes are working then obstruction free makes absolutely no guarantee. The operations could block forever. We will mention later that many transactional memory systems are obstruction free but nothing stronger.

Scalability. So far we have only considered measures having to do with completion in "a finite number of steps". That finite number could be very large and can depend on the number of processors. Under our definition of even the strongest criteria, wait-free, an operation could fully sequentialize access. Even worse it could take time exponential in the number of processors. Such a weak bound is therefore unlikely to be useful in analyzing the performance of parallel algorithms, especially *scalable* algorithms in which we want the time to be sublinear in the number of processors.

To get a better handle on performance we will use the following definition. Assume that the machine has some set of primitive instructions. Let τ be the longest time taken by any of these instructions (*i.e.*, the time from when the instruction start until the next instruction in the thread starts.) We say an algorithm or operation takes m instructions if it runs in $m\tau$ time.

4 Linearizability

We now return to the issue of how to more formally prove the correctness of concurrent operations. For example we gave an informal proof for the implementation of FetchAndAdd with CompareAndSwap and now we look at this more carefully. This will be important when defining concurrent versions of stacks and queues. First let us consider some parallel histories involving two queues, A and B. In each of the following H_1 and H_2 are the history of two parallel threads and H_S is some legal interleaving of the histories.

```
H_1: A.Enq(x); A.Enq(y); A.Deq() -> x
H_2: A.Deq() -> y;
H_S: A.Enq(x); A.Enq(y); A.Deq() -> x; A.Deq() -> y

H_1: B.Enq(x); B.Eng(y); B.Deq() -> y;
H_2: B.Deq() -> x
```

```
H_S: B.Enq(x); B.Enq(y); B.Deq()->x; B.Deq()->y

H_1: A.Enq(x); B.Enq(x); A.Enq(y); B.Eng(y); B.Deq() -> y; A.Deq() -> x
H_2: A.Deq() -> y; B.Deq() -> x
H_S: NONE
```

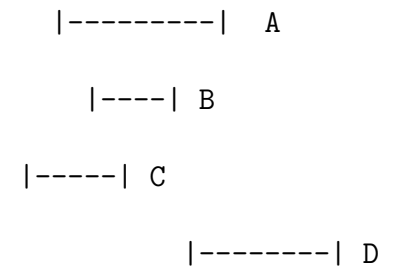
We note that the first two are sequentially consistent since there is a corresponding total ordering of the queue operations. However the third one is not sequentially consistent. The odd thing is that the operations on A and B considered separately in the third parallel history are identical as the first two (e.g. just cross out all operations on A). We therefore have a situation in which if we look at either either one of the queues the operations are sequentially consistent but if we look at a composition of the two it is not. Therefore sequential consistency is non-compositional. This can be problematic for proving correctness based on sequential consistency since just proving correctness for each structure in isolation is not good enough. When put together the operations might no longer be correct. For this reason we define another notion referred to as linearizability which is better for analyzing correctness of concurrent data structures.

Informally *linearizability* states that every operation must appear to take effect atomically at some point between when it starts and when it ends. This is referred to as the *linearization* point. This is basically the argument we made for proving the FetchAndAdd code was correct—we said it appeared to take place atomically at the point at which the CompareAndSwap succeeded. Most linearizable implementations of data types have a specific instruction in the code at which point the operation appears to take place. However, this is not always the case and some implementations the spot in the code that the operation appears to take place will depend on what else is going on in the system.

To formalize the notion of linearizability we have to more carefully define what we mean by between when an operation start and ends. For the purposes of this document it is sufficient to say at some atomic instruction within the code but here we briefly go into the formal definition.

Formally one needs to split every operation into an invocation event (when it starts) and a response event (when it ends). A sequential history is a sequence of pairs of invocation-response events (every invocation is immediately followed by its response). As before, a sequential history is legal if the invocation-responses are consistent with a sequential execution of the data type. A concurrent history is a sequence of invocation and response events but the events don't have to be adjacent. However, the response for an operation must come after the invocation, of course. In a concurrent history we say that an operation B comes after A if the invocation of B comes after the response of A. This defines a partial order on the operations in a concurrent history. Based on these definitions we say that a concurrent history H is linearizable if there is a legal sequential history H' that is consistent with the partial ordering defined by H. This is still a simplification of the full definition which requires considering that some operations might not be complete (i.e. they have been invoked but have not returned).

As an example consider the following operations. In the graph time goes from left to right.



This can be explained by the concurrent history:

$$H : I_C, I_A, I_B, R_C, R_B, I_D, R_A, R_D$$

where I_x indicates an invocation for x and R_x indicates its response. A sequential history that is consistent with H is:

$$H' : I_A, R_A, I_B, R_B, I_C, R_C, I_D, R_D \ .$$

5 More Asynchronous Algorithms

In this section we will consider some algorithms for implementing stacks and a simple memory allocator. Consider the following array implementation of a stack.

```
int A[n];
int top;

void push(val y) {
  int j
  j = fetchAdd(&top, 1)
  A[j] = y; }

val pop() {
  int k;  val x;
  k = fetchAdd(&top, -1)
  if (k <= 0) {
    top = 0;
    x = EMPTY; }
  else x = A[k-1];
  return x; }
```

This code actually does not work, but the reason is subtle. Can you figure it out? First consider just having pushes. Is it correct? Then consider whether it is correct if there are just pushes. Then consider whether it is correct if there are just pops. Finally consider having concurrent pushes and pops. For example, consider the following interleaving of instructions:

```
struct block { valtype val; block* next; }

block* top = NULL;

void free(block *p) {    // push p onto front of list
  block *n;
  do {
     n = top;
     p->next = n;
  while (CAS(&top, n, p) != n); }

block *alloc() {         // pop first element of list
  block *n,*p;
  do {
     n = top;
     p = n->next;
  while (CAS(&top, n, p) != n);
  return n; }
```

Figure 2: An implementation of memory allocation using a concurrent stack. This implementation is almost but not correct due to the ABA problem.

```
j = fetchAdd(&top,1); // from push
k = fetchAdd(&top,-1); // from pop
x = A[k-1]; // from pop
A[j] = y; // from push
```

Figure 2 gives another implementation of stacks based on lists. This version is part of a memory allocator that pushes blocks onto a free list using `free` and pops them from the list using `alloc`. The code seems like it could work, but it still has a problem. Consider the following list

```
top -> [x, ] -> [y, ]
```

Now we have one thread do an `alloc()` and a second thread do an `alloc()` which returns `x` followed by a `free(z)` and `free(x)`. Note that the second free is on the element that was allocated. What can go wrong?

Consider the following interleaving:

```
T1 :  n = top;
T1 :  p = n->next;
T2 :  x = alloc(); free(z); free(x);
T1 :  n = CAS(&top, n, p);
```

At the end we have the list

```
top -> [y, ]
```

and we have lost the `z` that was inserted. This is called the ABA problem and is very common with protocols involving a compare and swap (or even without). The value of top goes from A (x) to B (z) and back to A (x). When the compare and swap happens on P1 it sees an A and goes ahead even though there was a B at some time in between.

Here is a fix. Assume top is a double word with an actual pointer and a counter. Use a double word CompareAndSwap which does a compare and swap on two adjacent words. Then change alloc to:

```
struct pointer_t {block *ptr, int cnt};
pointer_t* top;

block *alloc() {          // pop first element of list
  pointer_t *n,*p;
  do {
     n = top;
     p.ptr = (n.ptr)->next;
     p.cnt = n.cnt+1;
  while (CAS2(&top, n, p)); }
```

Why does this work? What do we need to assume?

6 Room Synchronizations

So far we have not bounded the time for operations on our data structures. In fact our lock-free structures all sequentialize access to the data structure and are therefore not scalable under our earlier definition. In this section we are interested in data structures that support asynchronous parallel accesses that are both scalable and linearizable, although not necessarily non blocking. Furthermore we are interested in proving bounds on the time needed to access the data structure, at least under well-specified assumptions.

For this purpose we use a synchronization structure called room synchronizations which is useful for certain data structures. In a room synchronization there are m rooms (typically just a few). A user can request to enter a room i using `enterRoom(`i`)` and will potentially be blocked until it is allowed in at some later time. After entering the user executes some code in the critical region and then leaves the room using `exitRoom()`. The primitives allow any number of users to occupy a given room at the same time, but no two processes can occupy different rooms at the same time. Therefore if some set of processes are in room 1, then a process requesting room 2 will have to wait for them all to leave before it can enter. Once all processes leave room 1, then all processes waiting on room 2 can enter that room.

Figure 3 gives some code for implementing room synchronizations. The function `createRooms(m)` creates a room synchronization structure with m rooms. Although we will not prove it here,

assuming a FetchAndAdd is a primitive instruction, this implementation guarantees that no user will wait more than $m(t_r + k\tau)$ time to enter or exit a room, where m is the number of rooms, t_r is the maximum time any user spends in a room, τ is the maximum time taken by any instruction, and k is a constant. Therefore as long as every user only spends constant time in a room entering a room will take constant time (here "constant" refers to the time to execute a constant number of instructions). We will make use of this to prove time bounds.

We now look at how we can use room synchronizations to implement a parallel stack. At the start of Section 5 we considered an implementation of a stack that used FetchAndAdd. If you recall the problem with that implementation is when the instructions from a push and a pop are interleaved. In fact we argued that if we just had pushes, or just had pops then the code would be correct. This therefore suggests that we could use room synchronizations with a separate room for pushes and pops. This idea leads to the following code.

```
#define PUSHROOM 0
#define POPROOM 1

void push(val y) {
  enterRoom(PUSHROOM);
  int j = fetchAdd(&top,1);
  A[j] = y;
  exitRoom(); }

val pop () {
  val x;
  enterRoom(POPROOM);
  int k = fetchAdd(&top,-1);
  if (k <= 0) {
    top = 0;
    x = EMPTY;  }
  else x = A[k-1];
  exitRoom();
  return x; }
```

Note that the only difference is wrapping the push and pop each with its own `enterRoom` and `exitRoom`. The rooms prevent the instructions from a push and a pop from being interleaved but allows any number of processes to push or pop simultaneously. Now we see that the code within each room only involves a constant number of instructions so based on the $m(t_r + k\tau)$ bounds on the time to enter and exit a room given earlier, we have a bound $2(k_1\tau + k\tau)$ for the time taken to enter or exit the room by the push and pop. Therefore the overall time for either a push or pop is a constant.

```
struct rooms {
  int *wait, *grant, *done;
  int numRooms, active, activeRoom;
} Rooms_t;

Rooms_t *createRooms(int m) {
  Rooms_t *r = (Rooms_t *) malloc(sizeof(Rooms_t));
  r->wait = (int *) calloc(m, sizeof(int));
  r->grant = (int *) calloc(m, sizeof(int));
  r->done = (int *) calloc(m, sizeof(int));
  r->numRooms = m;
  r->active = 0;
  return r; }

void enterRoom(Rooms_t *r, int i) {
  int myTicket = FetchAndAdd(&r->wait[i],1) + 1;
  while (myTicket > r->grant[i]) {
    if (testSet(&r->active)) {
      r->activeRoom = i;
      int currWait = r->wait[i];
      r->grant[i] = currWait;
      return;
    } } }

int exitRoom(Rooms_t *r) {
  int ar = r->activeRoom;
  int myDone = FetchAndAdd(&r->done[ar],1) + 1;
  if (myDone == r->grant[ar]) {
    for (int k = 0, newAr = ar; k < r->numRooms; k++) {
      newAr = (newAr + 1) % r->numRooms;
      int currWait = r->wait[newAR];
      if (currWait - r->grant[newAR] > 0) {
         r->activeRoom = newAr;
         r->grant[newAr] = currWait;
         return; } }
    r->active = 0; } }
```

Figure 3: Code for implementing Room Synchronizations.

7 Serializability

Serializability is different from linearizability or sequential consistency in that it considers blocks of operations and makes them atomic as a group.

Consider the following code for reversing the order of a stack.

```
while (!empty(s1))
 x = pop(s1)
 push(s2,x)
```

Now lets assume that this was being run by multiple processes at the same time. Even if the `push` and `pop` are linearizable, the result might be a stack with the elements in the "wrong" order. For example starting with a, b, c, d in `s1` could end up with d, c, a, b in `s2`. Even worse it could cause an exception since the `pop` could be applied to an empty stack. Note, however, that if the `empty`, `pop` and `push` could be executed atomically as a group on each processor then the code would work properly.

Serializability assumes you have a set of atomic regions (sometimes called transactions) each with a sequence of atomic operations. The outcome has to act such that there is some ordering of the regions that leads to a legal operation sequence. As with sequential consistency, the atomic operations are most often assumed to be reads and writes to memory.

Basic serializability puts no constraints on the ordering of the atomic regions.

Strict Serializability. This is serializability but it adds a precedence graph (DAG) to the atomic blocks (transactions). In this case there has to be an ordering of the blocks that obeys the precedence graph that leads to a legal operation sequence.