CpSc 418 **Homework 4** **Solution Set**

1. **Reduce and Scan (75 points)**
   Implement each of the operations below using Erlang (with the `wtree` module) **and** MPI (using `MPI_Reduce`, `MPI_Scan`, and `MPI_Op_create`).

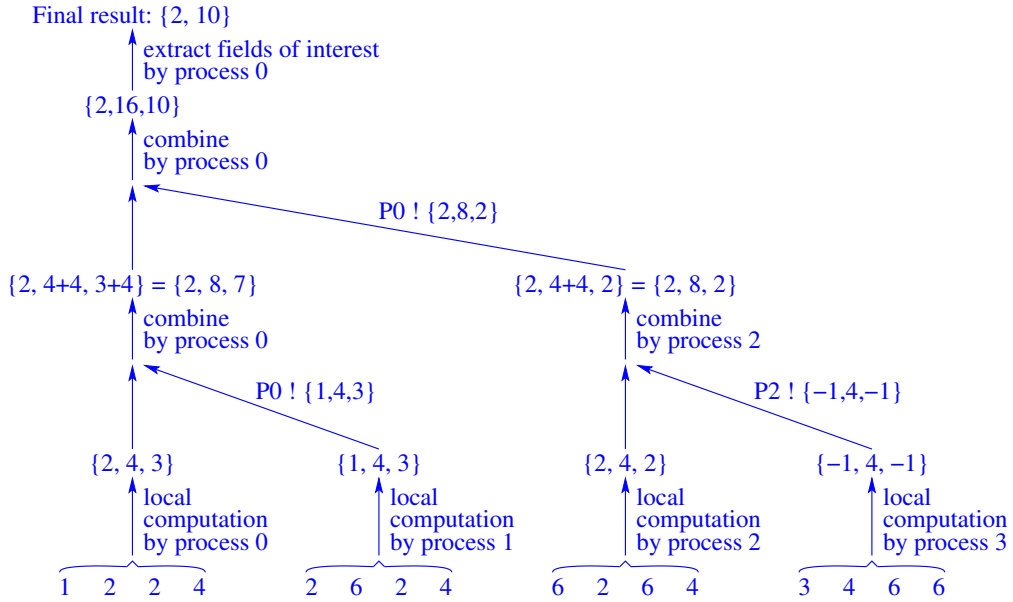
   (a) Find element. **(25 points)**

   i. Draw a picture. **(5 points)**
   Given an array, A, of N elements, and a special value, q, define

   $$\{first, last\} \quad = \quad index(q, A)$$

   Where *first* is the smallest integer, $i \in 1, \ldots, N$ such that $A_i = q$, and *last* is the largest integer in $i \in 1, \ldots, N$ such that $A_i = q$. If no element of A is equal to q, then *first* is $+\infty$, and *last* is $-\infty$. Draw a diagram that shows how this computation can be performed using a reduce operation with four processes where each process initially holds four consecutive elements of A.

   Solution:



   Note: My solution is based on the example from the problem statement with $q = 2$. I assumed four processes, where each process initially holds four elements of the array, A. I changed the value of A_{13} from 2 to 3 to get my example to illustrate what happens if a process has no array elements that match the key. The value passed up the tree is a tuple of the form {`First, Length, Last`}, where

   `First` is the index of the first occurrence of q in the subtree (or -1 if q does not occur in the subtree);

   `Last` is the index of the last occurrence of q in the subtree (or -1 if q does not occur in the subtree);

   `Length` is the total number of elements in the subtree.

   ii. Erlang version: (**10 points**)
   Solution: see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/hw/hw4.erl`.

   iii. MPI version: (**10 points**)
   Solution: see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/hw/hw4.c`.

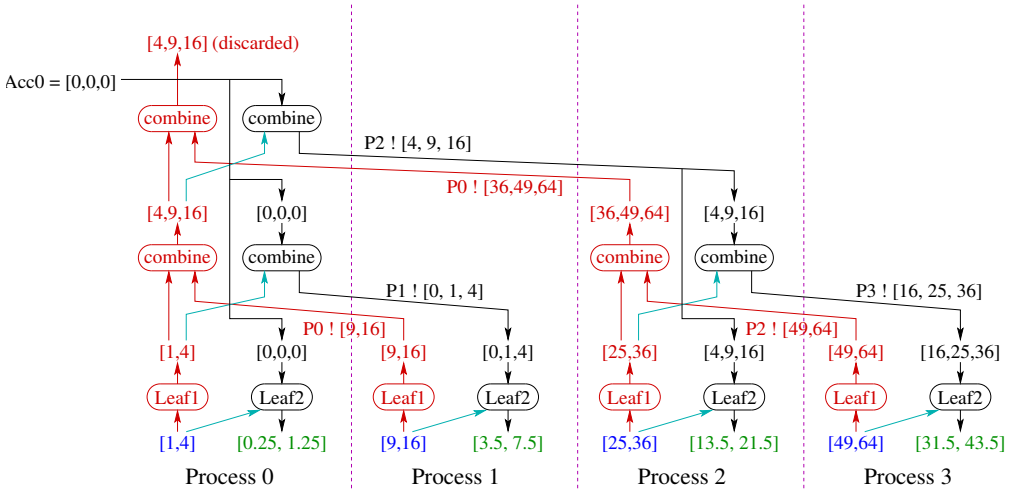
(b) Rolling average. **(25 points)**

i. Draw a picture. **(5 points)**
Given an array, A, of N elements, the M-element rolling average of A is the array B where

$$B_k = \frac{1}{M} \sum_{i=\max(1,k-(M-1))}^{k} A_i$$

Draw a diagram that shows how this computation can be performed using a scan operation with four processes where each process initially holds two consecutive elements of A, and each process will hold two elements of B at the end of the reduce. You can draw your diagram neatly by hand, scan it, and include it in `hw4.pdf`, or you can draw it using a drawing program of your choice, export it as a PDF file, and include it in `hw4.pdf`.

Solution:



Notes: To compute a m-way rolling average, each subtree sends its m last values to its parent node. If the subtree has fewer than m values, then it sends all of its values. In the downward phases, each parent node sends to each of its children the m values that are to the right of that child.

- The original data is shown in blue.
- The propagation of values up the tree is shown in red.
- The propagation of values down the tree is shown in black. The value "reused" in the up and down computations is indicated with a cyan arrow.
- The final result is shown in green.
- The `Leaf1` function produces the m last elements of the node, or all of the elements if there are fewer than m.
- The `Combine` function produces the last m last elements of the concatenation of its left and right operands. If the total number of elements of hte operands is less than m, then `Combine` produces the concatenaton of the its two operands. For the "downward" computation, the operand from the parent is the "left" operand.
- The `Leaf2` function computes the rolling average given the values from the left of the node and its own values.

ii. Erlang version: (**10 points**)
Solution: see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/hw/hw4.erl`.

iii. MPI version: (**10 points, Extra Credit**)
Solution: see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/hw/hw4.c`.

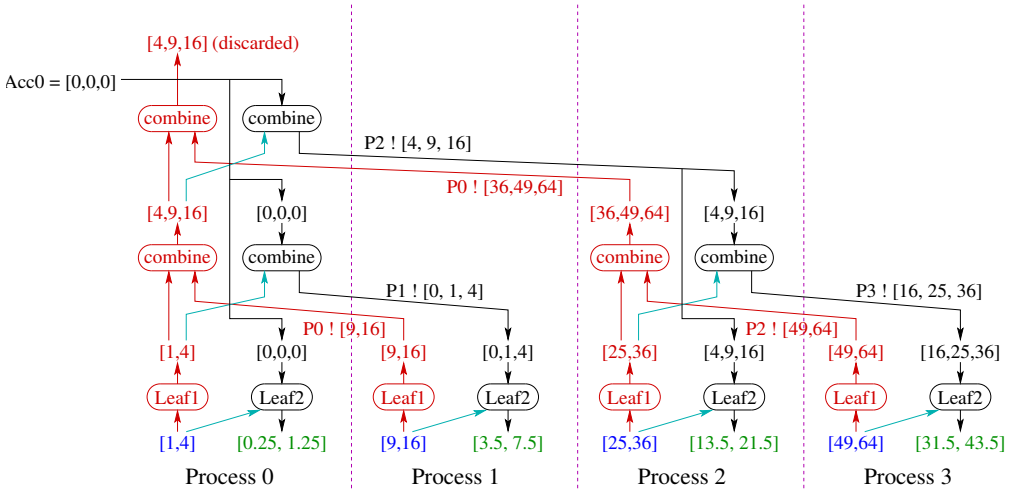
(c) Credit Card balance (**25 points**) Consider a credit-card account that is opened on day 0 with a balance of \$0.00. Let T be a list of transactions, where each transaction is a tuple (d, v); d is an integer, the date on which the transaction took place; and v is the amount of the transaction. If v is positive, it is a *purchase*, which increases the balance owed on the account. If v is negative, it is a *payment*, which decreases the balance owed. For any positive integer, n, we compute the balance on day n in two steps:

$$\begin{aligned} \text{balance}(0) &= 0 \\ \text{balance}(n) &= (1+r) * \text{balance}(n-1) + \sum_{(n,a)\in T} a, \quad \text{the "true" balance} \\ \text{acctbal}(n) &= \text{round}(\text{balance}(d), 0.01), \quad \text{rounded to the nearest penny} \end{aligned}$$

where r is the daily interest rate, and $\text{round}(x, p)$ rounds x to the nearest multiple of p.

i. Draw a picture. **(5 points)**
Draw a diagram that shows how the computation of the account balance after each transaction can be performed using a scan operation with four processes where each process initially holds two consecutive elements of T, and each process will hold two elements of B at the end of the scan.

Solution:



Notes: note that the figure is the same as for Q1.b, I just had to change the values being passed between the processes. On the upward pass, each leaf computes its final balance assuming that the starting balance is 0. It sends a tuple {`FinalDate, FinalBalance`} to its parent. The combine operation applies the interest to the left balance according to the difference between the right final date and the left final date:

$$\mathit{CombinedBalance} = \mathit{LeftBalance} * (1 + \mathit{Rate})^{\mathit{RightDate} - \mathit{LeftDate}} + \mathit{RightBalance}$$

The downward computation uses the same `Combine` function (or course) to determine the account balance and date to the left of each leaf, and from this, each leaf computes the balance following each transaction.

ii. Erlang version: (**10 points**)
Solution: see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/hw/hw4.erl`.

iii. MPI version: (**10 points**)
Solution: see `http://www.ugrad.cs.ubc.ca/~cs418/2012-1/hw/hw4.c`.

2. Test-and-set **(30 points)**
In class and on homework 3, we considered mutual exclusion algorithms for which the only atomic (i.e. indivisible) operations were memory reads and memory writes. Modern machines provide other instructions, where a simple one is `tas` ("test-and-set"). In particular,

```
tas $Rdst, $Rptr
```

reads the memory location at the address given by register `$Rptr`, stores the value read in register `$Rdst`, and sets the content of the memory location to `1`.

(a) Using `tas` for mutual exclusion (**10 points**)
Show that the following code guarantees mutual exclusion for N threads indicated by their respective program counters, $\texttt{PC}_0, \ldots, \texttt{PC}_N$.

```
initially:  flag = false;
PCi=0:  while(true) {
PCi=1:    non-critical code
PCi=2:    while(tas(&flag));
PCi=3:    critical section
PCi=4:    flag= false;
PCi=5:  }
```

where `tas(&flag)` performs a test-and-set on address of `flag` and returns the value that had been stored in `flag`. Following the Peril-L convention, `flag` is underlined in the code above to indicate that it is a global variable.

To show that this code guarantees mutual exclusion, let

$$ncrit \quad = \quad |\{i \mid \texttt{PC}_i \in \{3,4\}\}|$$

Now show that I_N is an invariant of the program where:

$$I_N \quad = \quad (\texttt{flag} = (ncrit = 1)) \wedge (ncrit \leq 1)$$

Finally, write a *short* explanation of why I_N implies that at most one thread is in its critical section at any given time.

Solution:

Initially: $\texttt{PC}_i = 0$ for all $i \in \{0, \ldots, N-1\}$; thu $ncrit = 0$. Furthermore, `flag` = `false`. Thus, the I_N holds.

$\texttt{PC}_i \in \{0, 1, 3, 5\}$: performing any of these actions leaves $ncrit$ and `flag` unchanged. The invariant is maintained.

$(\texttt{PC}_i = 2) \wedge \neg\texttt{flag}$: By the assumption that I_N holds before performing this action, $ncrit \neq 1$ before performing the action. Thus $ncrit = 0$ before performing the action. (I should have added a clause to the invariant that $ncrit$ is non-negative.) The `TAS` instruction sets `flag` to `true` and "returns" a value of `false` for the while-loop test. This means that after performing this operation: $\texttt{PC}_i = 3$ and `flag` = `true` and $ncrit = 1$. The invariant is maintained.

$(\texttt{PC}_i = 2) \wedge \texttt{flag}$: In this case, performing the action leaves $\texttt{PC}_i$ and `flag` unchanged. The invariant is maintained.

$(\texttt{PC}_i = 4)$: By the assumption that I_N holds before performing this action, $ncrit = 1$ and `flag` = `true` both hold. Performing the action decrements $ncrit$ and clears `flag`. The invariant is maintained.

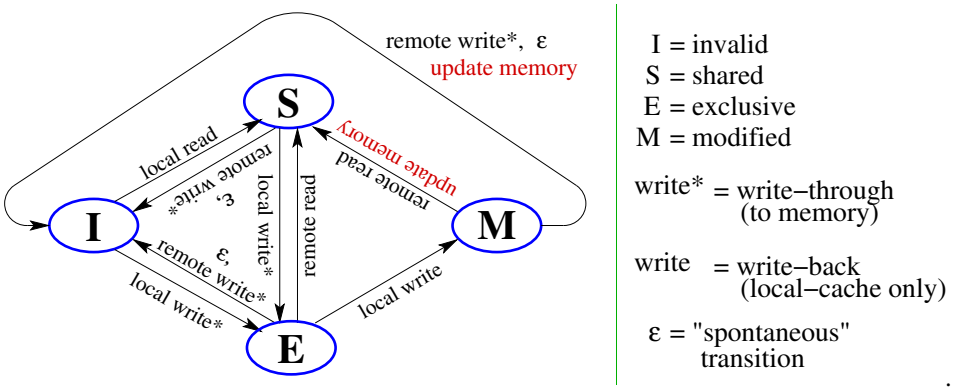


Figure 1: The MESI Cache-Coherence Protocol

This shows that I_N is an invariant.
To see that I_N guarantees mutual exclusion, I will show that if mutual exclusion is violated, then I_N does not hold. If mutual exclusion is violated, then two or more threads whose PC values are 4. This means that $ncrit \geq 2$, and therefore that I_N is false.

(b) Test-and-Set with MESI (**10 points**)
Figure 1 shows the MESI protocol from the October 4 lecture. Show that this protocol is insufficient for implementing the `tas` instruction. In particular, consider two threads that try to perform a test-and-set at the same time. Assume that thread 0 performs the first read. Show that there are no states that their caches can be in after this read that guarantees that thread 0 performs its write before thread 1 performs its read.

Solution:
Simple answer: consider starting execution from a state where both threads have the line for `flag` in their processor's caches in the shared state, and that `flag` = `false`. If I assume that the read and write of the `TAS` are treated as normal reads and writes, then we could have both threads perform a read of `flag` and get `false`. Then both would attempt to write `true` to `flag`. One thread (let's say thread 0) would win the arbitration, move to the exclusive state and update it's cache entry and main memory. The cache for the other thread would invalidate its block for `flag`. Now the other thread (let's say thread 1) would perform its write. It would first transition from invalid to exclusive by loading the cache line from memory, and then invalidate the entries by performing a write-through to memory, and write the value `true` on top of the `true` value that thread 0 already wrote. Because the write simply reloads the cache line and then performs the write, nothing stops the thread 1 from seeing a successful `TAS` and continuing to its critical region.
At this point, you might object:

Objection 1: shouldn't the write fail if the cache is in the invalid state?

Objection 2: didn't the problem state that a failure can occur no matter what state the thread 0's cache is in after it performs its read?

With respect to objection 1, we don't want a write to fail if the cache line is invalid – otherwise normal writes would fail when they incur a cache miss, and this would make programming very painful. We could consider having the `TAS` fail if the cache line is invalid when the write is performed. That seems to work.
With respect to objection 2: yes, the problem should handle the more general case. I gave a simple answer first, and the simple version should get full credit. Now, consider what happens if thread 0 can do it's read and tell its cache to move to some other state (perhaps invalidating the other cache lines at the same time). No matter what state the caches end up in, thread 1 can do

it's read following the usual MESI protocol. Next, one of the threads does its write. This gets us back to the original answer. If each thread requires the cache to be in the state that it left it in after its read for the write to go through and `TAS` to succeed, then

(c) Extending MESI (**10 points**)
Now, add a fifth state to the MESI protocol that we will label **T** in diagrams in honour of the test-and-set instruction. We will add a new operation called "read-with-intent-to-write" that is used for the read operation of a test-and set, and brings the cache into the **T state**. Draw the state-diagram for the five-state protocol that supports test-and-set. Your diagram should have states **M**, **E**, **S**, **I**, and **T**. Show the transitions for local-read, local write, remote-read, remote-write, local-read-with-intent-to-write, remote-read-with-intent-to-write, and ϵ. To make the transition labels legible, you may use the following abbreviations:

lr: read by the local processor

lw: write by the local processor

lx: read-with-intent-to-write by the local processor

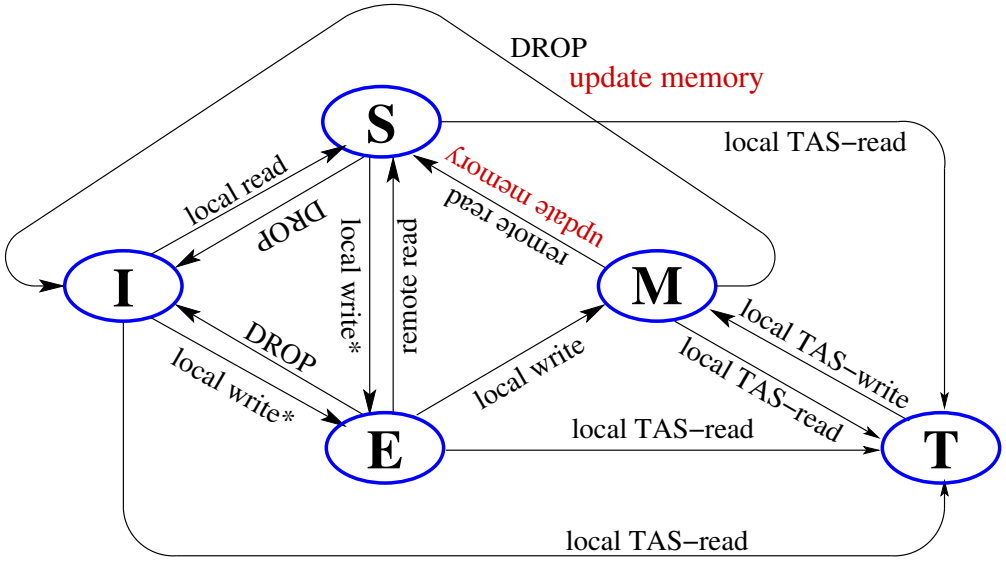
rr: read by another (i.e. remote) processor

rw: write by another (i.e. remote) processor

rx: read-with-intent-to-write by another (i.e. remote) processor

ϵ: Spontanous transition (always allowed)

Solution:



Note: Drop = remote write*, remote TAS−read, or ε

Observe that once a cache enters state **T** it stays there until a local `TAS`-write has been performed. In particular, it won't move to the other states while waiting for the `TAS`-write, and other caches will be blocked from moves to **S**, **E**, or **T** while this cache is in state `T`.