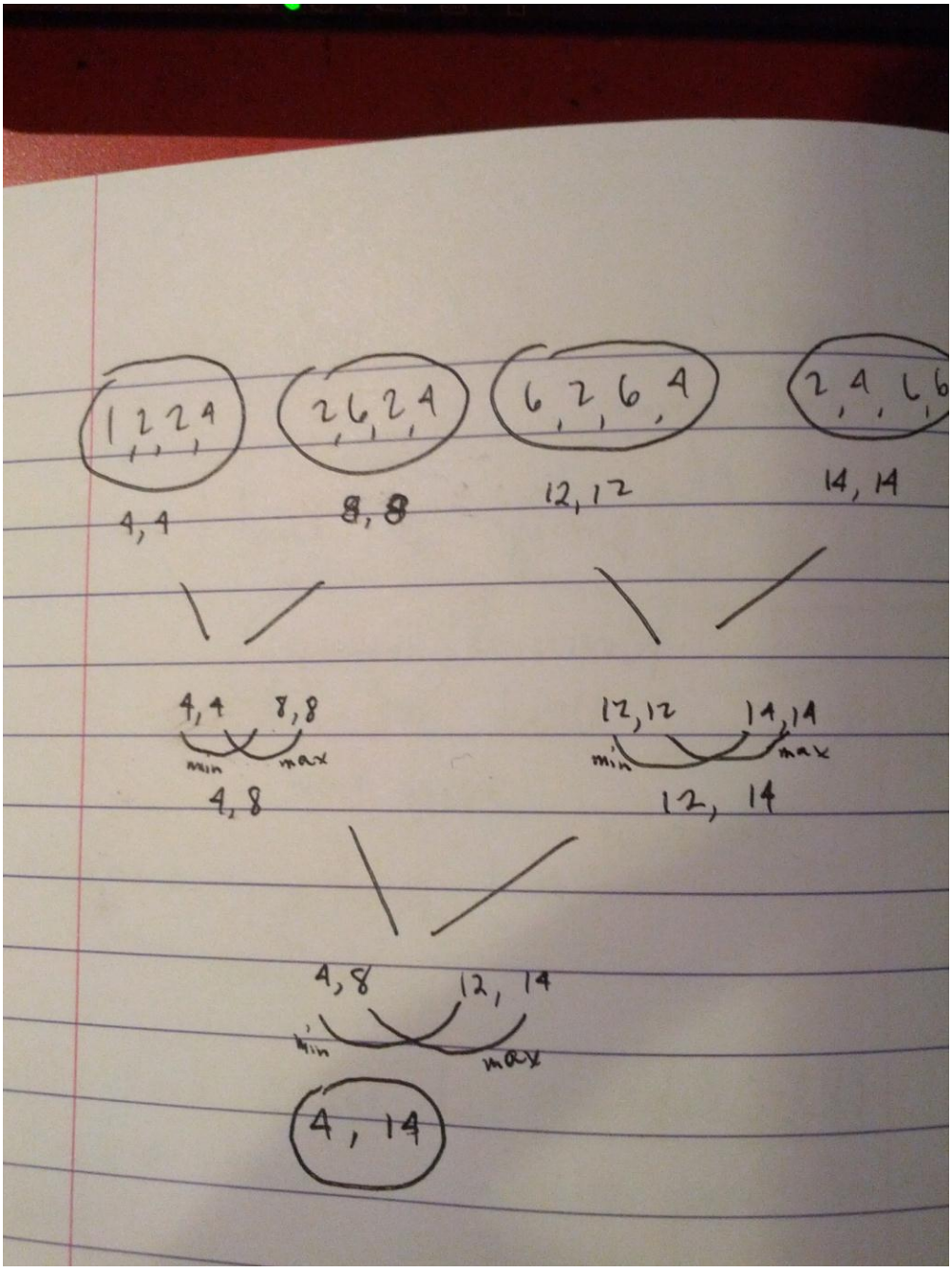
Yelnil Gabo
70179064 g3d6

Note: hw4.c has a main that runs my MPI functions

1a.i

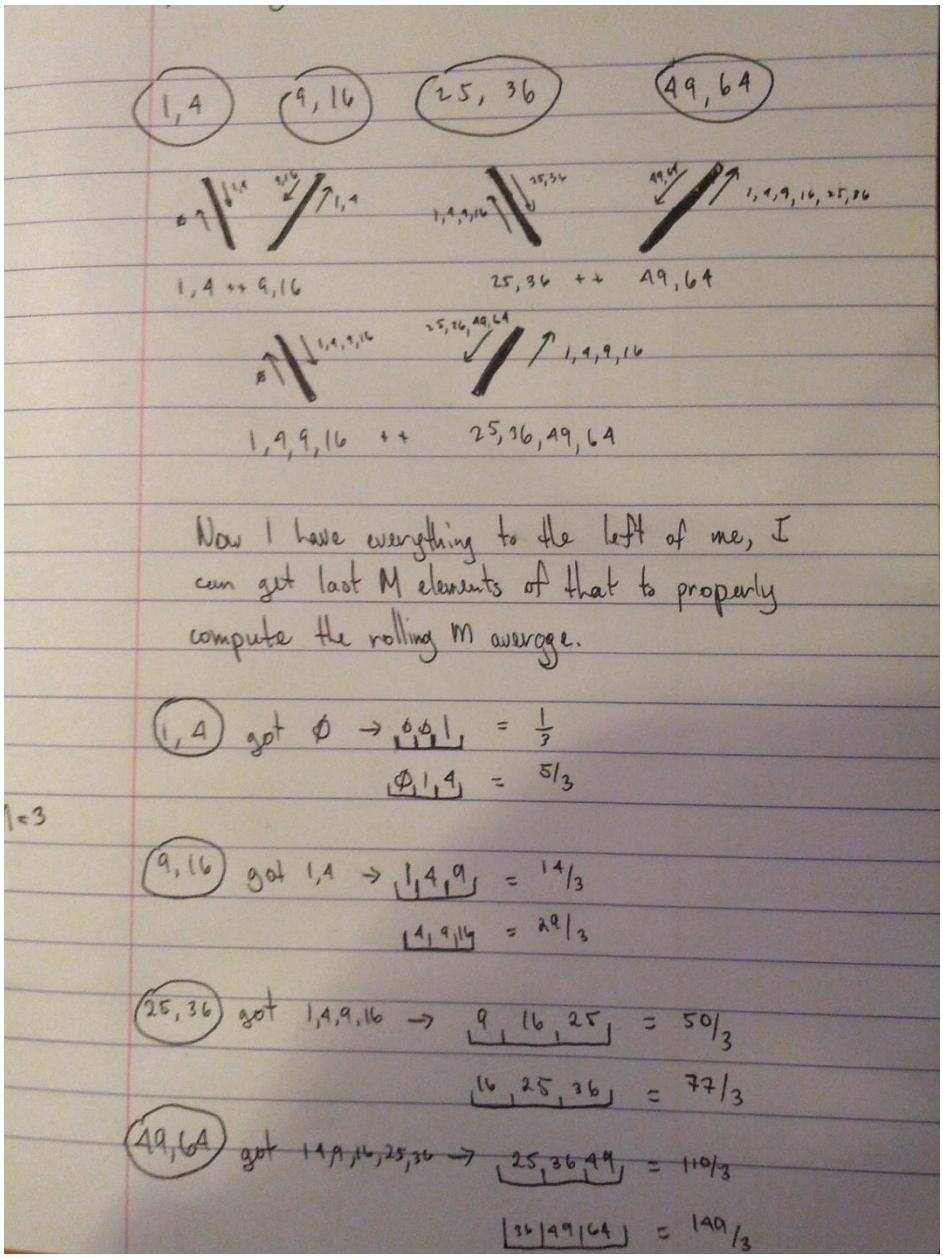


1a.ii hw4.erl

1a.iii hw4.c – I assumed RootProcess here is always 0.

Yelnil Gabo
70179064 g3d6

1b.i

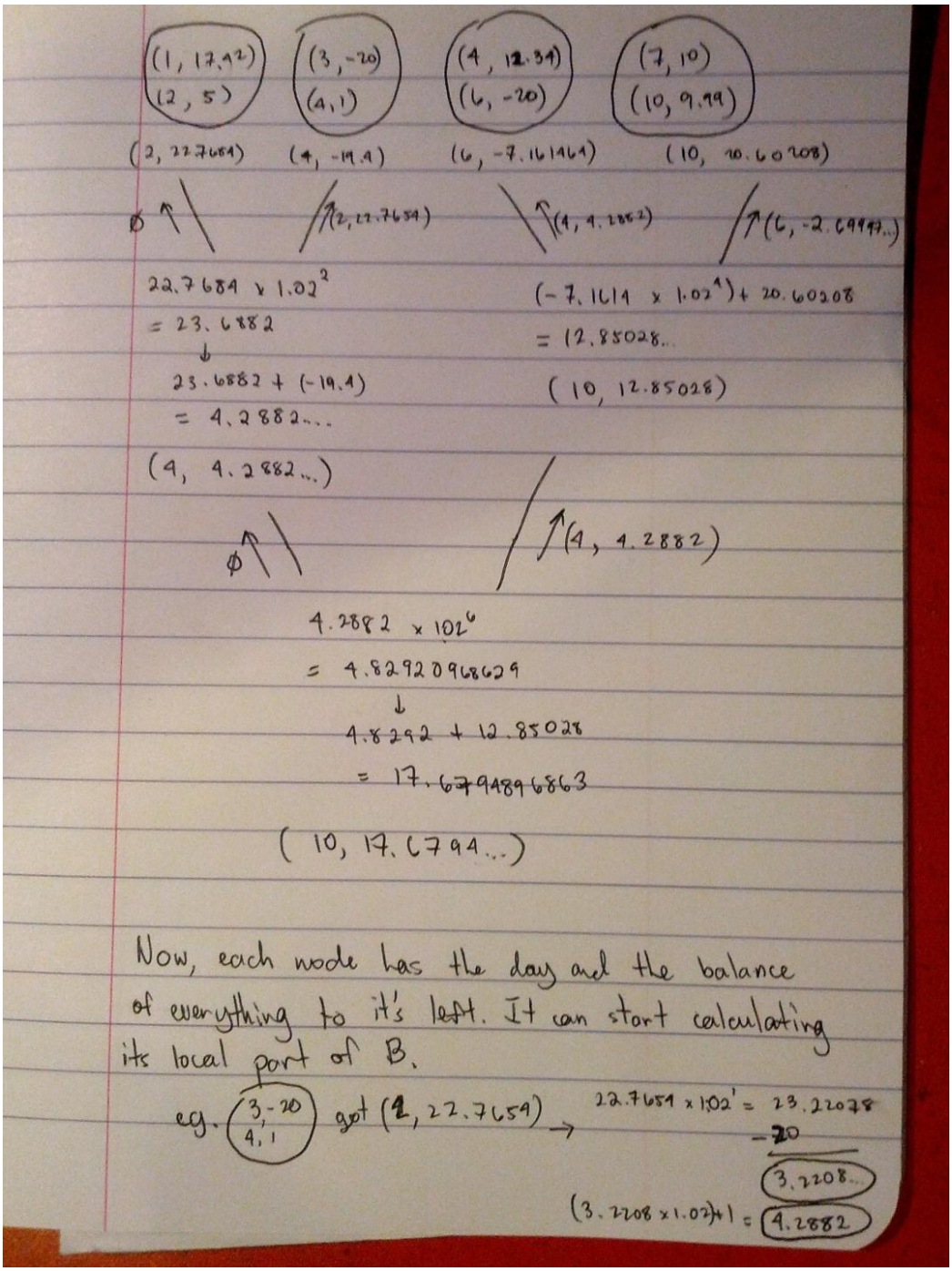


1bii. hw4.erl

1biii. hw4.c – Didn't perform scan for this, had to manually send to neighbor the data.

Yelnil Gabo
70179064 g3d6

1ci.



1cii. hw4.erl

1ciii. hw4.c – I made the last node send the final answer to all the other nodes.

Yelnil Gabo
70179064 g3d6

2a.

I = Invariant = ((flag == (ncrit == 1)) ^ (ncrit <= 1))

Let flag == (ncrit == 1) be A, and (ncrit <= 1) be B. Ncrit means how many guys in PC ϵ {3,4}.

Invariant holds initially. Flag is 0 at first.

When ncrit==0, means everyone is in PCs that are 0,1,2,5. At 0,1,2 the flag is not being changed and remains false. B then is true and A is true. Invariant holds.

When ncrit is 1, that means there is someone inside the critical section of PC ϵ 3,4. Let's call that guy, D. When D is in PC = 3. He just executed "while(tas(&flag));" To get out of that while loop tas instruction must have returned 0/false. This also means it D set the flag to 1/true, which blocks everyone else from coming through. At this moment, A is true and B is true. So invariant still holds. At pc =4, D is about to set the flag to false/0, but at the moment flag has not changed in value. Meaning invariant continues to hold. The catch is, at PC=3 D set the flag to true/1, blocking everyone else from passing through. This prevents ncrit from going over 1.

So, the invariant continues to hold with all parts of the code. I is an invariant.

The invariant implies that there is only one thread in its critical section since when that specific thread successfully got out of the while loop. It blocked everyone else from coming through with the use of the tas instruction.

2b.

There are two threads trying to do the test and set instruction. Assume they both start in the shared state and the value in the memory location we are trying to read/write to is zero (0). First thread 0 does his first read and gets a 0 from that. Now, thread 1 does a read and also gets a 0 from his read. Both are still in shared state. Then, thread 0 continues with its write. He invalidates everyone else, goes to exclusive state and writes one (1) to the memory location. Thread 1 now has an invalid cache line. Thread 1 still has to write though so, from the invalid state, reloads the memory location and does a local write of one (1), and goes to exclusive. The problem is, both thread 1 and thread 0 had both read zero (0) so both of them returns zero (0).

Yelnil Gabo
70179064 g3d6

2c.