Homework 1
CSC410 - Parallel Computing

Aaron G. Alphonsus

September 28, 2018

1. The identity values for each operator is as follows:

&&	1
||	0
|	0
^	0

2. We can do this by declaring a private variable inside the `parallel` block and initializing it with the identity value of the operator we are using. We then use the `critical` pragma before updating the global variable.

3. (a)
```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i) shared (a, b, x)
    for(i=0; i<(int) sqrt(x); i++) {
        a[i] = 2.3*i;
        if (i < 10) b[i] = a[i];
    }
```

   (b) This seems to be a sequential program. It looks like the objective is to start from the beginning of the `a` array and set each value to `2.3*i`. The program should stop after the first instance of `a[i] < b[i]` however, if we parallelize this, the flag could be set at the wrong time and cause an early loop termination.

   (c)
```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i) shared (a, n)
    for(i=0; i<n; i++)
        a[i] = foo(i);
```

   (d)
```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i) shared (a, b, n)
    for(i=0; i<n; i++) {
        a[i] = foo(i);
        if(a[i] < b[i]) a[i] = b[i];
    }
```

   (e) Similar to 3(b), this is not suitable for parallel execution because it could cause an early loop termination when one of the threads executes the `break` statement.

   (f)
```
dotp = 0;
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i) shared (dotp, a, b, n) \
    reduction(+: dotp)
    for(i=0; i<n; i++)
        dotp += a[i]*b[i];
```

(g)
```
# pragma omp parallel for num_threads(thread_count) \
    default(none) private(i) shared (a, k)
    for(i=k; i<2*k; i++)
        a[i] = a[i] + a[i-k];
```

(h) This is similar to problem 3(g) but we need to be careful. Groups of `k` positions may be filled in parallel but each subsequent group of `k` positions depend on the preceding group

4. Given that this is an `m`-stage pipeline and the task has `m` sub-tasks, when `Task 1` comes in, it is completed after `m` cycles. However, as the pipeline has now filled up, every cycle that follows completes another task. So the completion times of `Task 2`, `Task 3`, and `Task 4` are `m+1`, `m+2`, and `m+3` respectively.

   Therefore, for an `m`-stage pipeline executing `m` sub-tasks that each require 1 unit of time, `n` tasks can be processed in `m+n-1` time.

5. If the address of the nodes in a hypercube has n bits, it can have 2^n nodes at most and each node will have n edges.

   Perform an `XOR` between `u` and `v`. Count all the 1 bits. This gives you the minimum number of steps to reach `v`. You may start from either end and flip each bit if they are different, keep them same if they match.

```
for each bit of u
    if it differs from the bit of v in the same position
        change the bit // i.e. move to the neighboring node
    // else stay at the same node

e.g. u = 110, v = 011
        Move to: 111
        Move to: 111
        Move to: 011
```