README for week 8 in IN3200/IN4200

Exercise 1) Dependent Tasks

a) Two Workers

There is a lot of dependency between the tasks here, so the amount of work that can be done in parallel is limited. To minimize idle time task 3 should be completed as fast as possible to open up tasks 6, 7 and 8. Tasks 9, 10, 11 and 12 must be done in order, so nothing is gained by having an extra worker for the last few task. One possible schedule is

Time	1	2
$t = 0$	T1	T2
$t = 1$	T3	T4
$t = 2$	T6	T7
$t = 3$	T8	T5
$t = 4$	T9	Done
$t = 5$	T10	
$t = 6$	T11	
$t = 7$	T12	
$t = 8$	Done	

The two workers need a minimum of 8 hours.

b) Three Workers

The only possible speedup we can gain here is when task 3 is completed and we need to complete tasks 6, 7 and 8 before starting the last serial part. From this we can already say that the minimum time required is 7 hours. For completeness, a possible schedule:

Time	1	2	3
$t = 0$	T1	T2	Idle
$t = 1$	T3	T4	T5
$t = 2$	T6	T7	T8
$t = 3$	T9	Done	Done
$t = 4$	T10		
$t = 5$	T11		
$t = 6$	T12		
$t = 7$	Done		

Exercise 2) Output

The static schedule with a chunk size of 10 means that each process is given 10 iterations each in a round robin way until there are no more iterations left. There are 4 threads, and $100/10 = 10$ chunks to share. This means that two processes will get 3 chunks each while the last two only gets 20.

Thread	0	1	2	3
iter	1-10	11-20	21-30	31-40
iter	41-50	51-60	61-70	71-80
iter	81-90	91-100		

Each thread will make a call to `printf`, but the order is undetermined before run time. We can however calculate the value of `my_sum` for each thread when `printf` is called. For thread 0

$$S_0 = \sum_{n=1}^{10} n + \sum_{n=41}^{50} n + \sum_{n=81}^{90} n = 1365$$

Likewise for threads 1, 2 and 3 we get

$$S_1 = 1665 S_2 = 910 S_3 = 1110$$

The last call to `printf` is outside the parallel region and will only be called by the master thread. `total_sum` is a reduction variable and will have the value $S_T = S_0 + S_1 + S_2 + S_3 = 5050$

If we run the program and compile it we see that we do get this result, and running it several times will show that the order of the threads output changes.

```
From thread No.0: my_sum=1365
From thread No.2: my_sum=910
From thread No.3: my_sum=1110
From thread No.1: my_sum=1665
Total sum=5050
```

Exercise 3) odd-even

The functions doing the sorting are all defined in the header `oddeven.h`.

a) Serial Implementation

The serial implementation is given in the pseudocode and is fairly simple to implement. One thing to note though is that the pseudocode assumes that indexing starts at 1. In C indexing starts with 0, so the index of the elements in a must be reduced by one. e.g. $a_{2j+1} \rightarrow a_{2j}$ and likewise for the other elements.

b) Odd Length Lists

Examining the algorithm carefully reveals that the last index included in the sorting is $\lfloor n/2 \rfloor \times 2 + 1$. If n is odd (not divisible by 2) then $\lfloor n/2 \rfloor \times 2 \neq n$. Using zero indexing like in C we get the same result just shifted one value towards zero. To solve this issue we can add a remainder term to the even step, making the even step iterate over one more value of `j`, catching the last index.

c) Earlier Termination

One way to accomplish this is to realize that if no swaps where made in either the *even* or the *odd* step, then the array must be sorted. Adding a couple of flags that we check after making both steps we can terminate if no changes where made.

d) OpenMP Parallelization

With this algorithm we can't do the iterations of the outermost loop in parallel. We have to do the odd and even steps in order. We can however parallelize each step. If we just add `#pragma omp parallel for` to the two inner loops however we will get a lot of overhead. The threads will be repeatedly spawned and killed/put to sleep, incurring lots of overhead for each iteration over `i`. To alleviate this we can put the entire algorithm inside a parallel region. The flags we are using to see if any changes where made in the odd and even steps should now be reduction variables so that the threads only break the outer loop if none of the threads made any changes. Finally we add a barrier right before we let a single process set the changed flags to zero for the next iteration. If we don't add this barrier we might end up with some processes breaking out of the inner loop while others continue with the next iteration.