README for week 9 in IN3200/IN4200

Exercise 1) Work Division

a) 6 Workers

In two days our six workers can finish nine task. Labeling the “super” workers with an S we get.

T	1S	2S	3S	4	5	6
0	T1	T2	T3	T4	T5	T6
1	T7	T8	T9			
2	Done	Done	Done	Done	Done	Done

So to finish 20 tasks we need four days for the first 18 tasks. and then one more for the last two if we assign two super workers to the last tasks.

b) Generalization

To solve this problem we can deal with the m tasks in two stages. We know that p workers plus p “super” workers can complete $3p$ tasks in two days, the super worker complete $2p$ tasks, and the other workers complete p. So first we deal with the part of m that is divisible by $3p$. Using integer division this is just `2*(m/(3*p))`. Now we are stuck with a remainder `r = m%(3*p)`. There are three different possibilities:

- $r = 0$: In this case m is divisible by $3p$ and the first term is the complete solution.
- $0 < r \leq p$: If the remainder is less than or equal to p, the super workers can handle the remainder in one day, and we get one extra day.
- $r > p$: In this last case the super workers can't handle the remainder in one day and we therefore get two extra days.

Note: The remainder is always less than $3p$ because it is defined as the remainder from a division by $3p$. We will never have a case where the added days are more than 2.

When implementing this in code you can add this as a `if, else if, else` block. There are other ways to do this as well. One expression that also works is

```
int time =  m/(3*p) + (m + 3*p - 1)/(3*p) + ((m%(3*p))>p?1:0);
```

With integer division the first term become $\lfloor m/3p \rfloor$ and the second $\lceil m/3p \rceil$, assuming positive m and p. The last term just adds an extra unit of time if the remainder is greater than the number of super workers (? is called the ternary operator).

Exercise 2) Find Errors

There are several errors in this short program:

- Swapping stack pointers: *u* and *v* are allocated on the stack. These kind of pointers can't be swapped around in the same way that pointers to the heap can. To solve this we allocate them with `malloc`.
- `nowait` in inner loop: With the no wait clause, we will end up with a thread swapping the pointers before all threads are done with writing to `v`. Remove `nowait`.
- All threads swap the pointers: Without `nowait` there is a barrier at the end of the parallel loop, but all the threads execute the code after the loop anyway. Swapping the pointers each time a thread goes through that region. To avoid this we add a `#pragma omp single` region.
- Init of first and last value in *v*, `v[0] = v[N-1] = 0`: In the loop that does the computation the values of `v[0]` and `v[N-1]` are never updated. But after the pointer swap these values are used in the next iteration, meaning that the first iteration gets different boundary conditions from all the others. We should set `v[0] = u[0]` and `v[N-1] = u[N-1]`.

The source code in `w9_e2_errors.c` shows the resulting code.

Exercise 3) Parallelization

The exercise asks us to just parallelize the code, but it is often useful to think about some serial optimizations that can be done.

We see that the values that *a* are initialized to are never used (except the first one), so there is no need to do that computation. We don't need to store the values of *b* either, we can just set the elements of *a* to their final value immediately. For the last improvement we can combine the two loops into one.

```
a2[0] = 10.0;
for (i=0; i<N-1; i++) {
    a2[i+1] = cos(20.0+sin(0.1*i));
    s += a2[i];
}
```

Now we have reduced our problem to only parallelize this one loop. There is a dependency from one iteration to the next that would give us a race condition. This is easily fixed by updating *s* using the same index we update *a*, and changing the range of the iterator i. s become a reduction variable, but the parallelization is otherwise trivial.

```
a3[0] = 10.0;
s += a3[0];
#pragma omp parallel for reduction(+:s)
for (i=1; i<N-1; i++) {
    a3[i] = cos(20.0+sin(0.1*(i-1)));
    s += a3[i];
}
// This is just here to make sure that a3 == a2
a3[N-1] = cos(20.0+sin(0.1*(N-2)));
```

Notice that we add up the first element outside the loop so we don't miss it, and that we subtract one from i inside the sine.

Exercise 4) Reading a Web Graph

There are two parts to this exercise that we can look at separately, reading the file, and finding the top entries. There is a header, `week9.h`, associated with this exercise.

One thing to note is the usage of a structure.

```
typedef struct graph{
    int n_nodes;
    int n_edges;
    int* in;
    int* out;
    int* self;
} webGraph;
```

`typedef` is used to give this structure the name "webGraph" in our code. If you want to declare a new web graph structure you can do it in one of two ways:

```
struct graph graphA;
webGraph graphB;
```

There is not much gained from using a structure in this program, but the concept is very useful when dealing with variables and arrays that are related to each other. e.g when trying to create a matrix on the CRS format.

Reading the File

We create a file pointer and open the file using

```
FILE * fptr = fopen(argv[1], "r");
if (fptr==NULL) {
    printf("Could not open file.\n");
    return 1;
}
```

The `if` test triggers if the file could not be opened. `return 1;` will make the program exit the `main` function and terminate the program.

The file is in ASCII, meaning that the numbers in the “FromNodeId” and “ToNodeId” must be interpreted as ASCII, and converted to integers. Using fscanf this is handled behind the scenes for us. The link is to a C++ site, but the info is accurate for plain C as well.

The two first lines:

```
# Directed graph (each unordered pair of nodes is saved once): web-NotreDame.txt
# University of Notre Dame web graph from 1999 by Albert, Jeong and Barabasi
```

These lines are not of interest to our program, so we just want to move the file pointer forward past these two lines. `fscanf` takes in a formatting string that is used to tell the function what to read, and what to do with it. We just want to increment our file pointer to the next line. To do this with `fscanf`

```
fscanf(fptr, "%*[^\n]\n"); // Throw away the first line.
```

For those that know regular expressions this will look familiar. The “%” sign means that there is something we want to match/find. The `*` means that we don’t want to save this field/value. Then we have the actual expression we want to find. `[^\n]\n` means any number of any type of characters, except “newline”, then a “newline”. This will match any line, and increment the file pointer to the next line. So we do this twice to skip the first two lines.

The third line:

```
# Nodes: 325729 Edges: 1497134
```

On the third line we want to read in the number of nodes and edges in to some variables in our program. There are several different formatting strings that would give the same results here but one simple variant is

```
fscanf(fptr, "%*c %*s %d %*s %d %*[^\n]\n", &n_nodes, &n_edges);
```

- `%*c`: Single character, don’t store.
- `%*s`: String without spaces, of any length, don’t store.
- `%d`: Integer, should be stored.

The final part `%*[^\n]\n`, identical to what we used for the first two lines, and might not be necessary on your system. It is just there to avoid problems with different line terminators on different operating systems. It guarantees that the file pointer is incremented to the next line.

The fourth line we skip just like the first two, and then we can go on with reading the edge data. This is typically done in a while loop.

```c
int from, to;
while (fscanf(fptr, "%d %d", &from, &to)!=EOF){
    // Record the from and to values.
}
```

Find Top K

One way to achieve this is to sort the arrays (with odd-even transposition for example), and then just return the first k elements. This however, is very slow. Therefore we should be using a selection algorithm instead.

One variant that works very well here is “heap selection”. Add the first k elements of the array to the “heap”. Then for each element in the array, compare it to the smallest value in the heap. If the element is bigger, overwrite the value in the heap and find the new smallest value. Continue until you are through the entire array.

Using this approach we only need to run trough the array once. In the source code the variant shown is one where the indexes of the biggest array elements are kept in the heap instead of the actual values themselves.

The solution only outputs the greatest values from the array containing the number of inbound links, but the approach is identical for the outbound links.

ps: The output is not sorted either. But that is left as an exercise for the reader.