Assignment 4

by Tobias Hosp, Marcel Huber and Thomas Klotz

Task1

Writing the basic program

To achieve a execution time of at least 2 seconds, by incrementing an integer value, i used a volatile integer variable. Volatile variables can't be cached, so they're always fetched from the memory -> much longer execution time.
With 1900000000 iterations on my local machine respectively 1000000000 on lcc2 with 1 thread and optimization flag -O3, the program needs about 2 seconds to execute completely. I think without volatile, just with a normal int, it is almost impossible to get an execution time that high.

OpenMP 4.0 Affinity

The developer has the opportunity to decide on how many PLACES(hyperthreads, cores, sockets,...) the threads are executed. Places can be set with the Environment variables OMP_PLACES and the num threads variable could then look like this for 8 places: OMP_NUM_THREADS=4,4 (4,4 means nested: first 4 threds are started and then again 4 -> 16 threads; relevant for example below)

PLACES can also be defined by the developer:
OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}" (example)

Predifined places:

- cores: physical cores
- threads: hyperthreads
- sockets: processor package

Affinity policies:

- spread: spread threads evenly among places
- close: place threads near master thread
- master: collocate thread with master thread

Example:

Measurements:

Reference Value:

4 Threads basic program, just with -O3: 2.45598

4 Threads with omp parallel for optimization:

```
#pragma omp parallel for
  for (long i = 0; i < n; i++) {
    inc++;
  }
```

Measure: 0.78777 seconds

4 Threads with omp proc_bind(spread):

```
#pragma omp parallel proc_bind(spread)
  #pragma omp for
    for (long i = 0; i < n; i++) {
      inc++;
    }
```

Measure:

number_of_threads	PLACES=cores(4)	PLACES=threads(4)	PLACES=sockets(4)
2,2	0.75619	0.75606	0.78641

4 Threads with just omp proc_bind(close):

```
#pragma omp parallel proc_bind(close)
  #pragma omp for
    for (long i = 0; i < n; i++) {
      inc++;
    }
```

Measure:

number_of_threads	PLACES=cores(4)	PLACES=threads(4)	PLACES=sockets(4)
4	0.75309	0.75300	0.78591

4 Threads with just omp proc_bind(master):

```
#pragma omp parallel proc_bind(master)
  #pragma omp for
    for (long i = 0; i < n; i++) {
      inc++;
    }
```

Measure:

number_of_threads	PLACES=cores(4)	PLACES=threads(4)	PLACES=sockets(4)
4	2.46937	2.47554	0.73690

-> Master doesn't do anything for cores and threads but works for sockets

Using 2,2 threads omp parallel for and proc_bind(spread) nested with proc_bind(close) before it:

```
#pragma omp parallel proc_bind(spread)
  #pragma omp parallel proc_bin(close)
      #pragma omp for
      for (long i = 0; i < n; i++) {
          inc++;
      }
```

Measure:

number_of_threads	PLACES=cores(4)	PLACES=threads(4)	PLACES=sockets(4)
4	0.00014	0.00010	0.00018

Using 2,2 threads omp parallel for and proc_bind(spread) nested with proc_bind(master) before it:

```
#pragma omp parallel proc_bind(spread)
        #pragma omp parallel proc_bind(master)
            #pragma omp for
            for (long i = 0; i < n; i++) {
              inc++;
            }
```

Measure:

number_of_threads	PLACES=cores(4)	PLACES=threads(4)	PLACES=sockets(4)
4	0.00014	0.00010	0.00020

Conclusion:

So splitting the threads with proc_bind(spread) and proc_bin(close) looks like a really good option to optimize the code, because it went from about 2.4 seconds to 0.00010 seconds.

The version where only proc_bind(master) is used, is in this case not helpful, because it operates like it runs not optimized. proc-bind(master) and also close are good for caching-cases and so on, but i "disabled" it with the volatile int.

Resources:

- https://www.lrz.de/services/compute/courses/x_lecturenotes/mic_workshop_2017/Michael-Klemm-tutorial.pdf
- https://www.openmp.org//wp-content/uploads/openmp-examples-4.0.2.pdf

Task 2

What the code should do

Flush.c spawns 2 threads using the omp parallel pragma. Both of these threads can access the integers 'data' and 'flag'.
The following block gets executed by thread 0:

```
if (omp_get_thread_num()==0) {
         data = 42;
         flag = 1;
      }
```

First the shared variable 'data' is set. After this is done, the variable 'flag' gets set.

The second block gets executed by thread 1 in **parallel**:

```
else if (omp_get_thread_num()==1) {
         while (flag < 1) {
         }
         printf("flag=%d data=%d\n", flag, data);
      }
```

The thread waits until 'flag' is set to 1 by thread 0 and afterwards prints both variables.

What the code actually does

In theory the logical order of events is correct and thread 1 should print the variables at some point, **however** this behavior can be observed only most of the time.

The code has been compiled with `-O3` and gcc version 8.2.0.
For all the following results, the code has been executed at least 1000 times using a shell script.

It usually results in the expected output, but every now and then the program doesn't produce output and seems to stick in the endless loop.

An attempt to explain the behaviour

The odd behavior can be explained with the **OpenMP memory model**. In OpenMP, every thread has its own private memory space, which is not necessarily synchronized with the main memory. This means, that every read/write operations could potentially read/write from/to other data sources, e.g. registers.
In our case this would mean that it is not guaranteed that thread 0 writes `flag` or `data` to main memory, nor is it guaranteed that thread 1 reads `flag` or `data` from main memory.

One possible explanation for our codes behavior is a race condition.
Assume that thread 1 reads `flag` only once from main memory - right before thread ones code execution starts. If thread 0 is already done with its task at this point, thread 1 reads `flag = 1` from main memory and immediately leaves the while loop. However, if thread 0 did not write `flag` to the main memory yet, thread 1 reads `flag = 0` and continues with the loop. After that, thread 1 reads `flag` only from registers, which means that it doesn't see updates to `flag` anymore and stays in the loop forever.

First attempt to fix the code

The first approach to fix this problem with flush directives looked like this:

```
int main() {
   int data, flag = 0;
   #pragma omp parallel num_threads(2)
   {
      if (omp_get_thread_num()==0) {
         data = 42;
         flag = 1;
         //guarantee that flag and data are stored in main memory
         #pragma omp flush(flag, data)
      }
      else if (omp_get_thread_num()==1) {
         //guarantee that flag is read from main memory the first time of loop execution
         #pragma omp flush(flag)
         while (flag < 1) {
           //guarantee that flag is read from main memory all the time (equals volatile
flag?).
            #pragma omp flush(flag)
         }
        //guarantee that data is read from main memory
         #pragma omp flush(data)
         printf("flag=%d data=%d\n", flag, data);
```

```
      }
   }
   return 0;
}
```

At first glance this code seems perfectly fine, but after the finding of a suspicious stackoverflow entry closer examination of the specification of OpenMP, several more errors were found.

- It is not guaranteed that flag gets flushed after data in thread 0, which could lead to invalid data getting read in thread 1.
- As long as code is 'independent', the compiler can move it around freely. This is very problematic for the flush operations, since there is no guarantee that they are performed in the order in which they appear in the code.

Second attempt to fix the code

The order of flush statements is only guaranteed to be as written in the code for statements that share at least one variable. As an example, consider the following:

```
...
int a = 3;
int b = 0;
#pragma omp flush(a)
b = 4;
#pragma omp flush(b)
...
```

Since the `#pragma` statements refer to different variable sets, it can not be guaranteed that the flush of a is performed before the flush of b. Instead, the following code ensures that there is no reordering between the flushes:

```
...
int a = 3;
int b = 0;
#pragma omp flush(a,b)
b = 4;
#pragma omp flush(b)
...
```

The following code is the final version of flush.c, which is (*hopefully*) error free:

```
#include <omp.h>
#include <stdio.h>

int main() {
   int data, flag = 0;
   #pragma omp parallel num_threads(2)
   {
      if (omp_get_thread_num()==0) {
         data = 42;
         //flush both flag and data to ensure that data gets written before flag
```

```
            #pragma omp flush(flag, data)
            flag = 1;
            //flush flag to ensure that flag is in main memory
            #pragma omp flush(flag)
        }
        else if (omp_get_thread_num()==1) {
            #pragma omp flush(flag, data)
            while (flag < 1) {
                //includes data to prevent reordering the for loop after the flush statement in
line line 25
                #pragma omp flush(flag, data)
            }

            //includes flag to ensure that the flush happens after the while loop
            #pragma omp flush(data)
            printf("flag=%d data=%d\n", flag, data);
        }
    }
    return 0;
}
```

What is still unclear

In the stackoverflow thread, the last flush in the program also flushes `flag`, and a comment over the last `printf` states that flag is not defined at this point. We do not see a reason why this change is necessary and therefore we did not include this in our solution.

Resources

Following websites were very helpful during the research for this exercise:

- stackoverflow
- the OpenMp API specification
- eamples of flush from OpenMP