Exam Assignments 3

Ordered Clause with a Parallel For Loop

The OpenMP `ordered` clause in a parallelized for loop specifies a region of code in the loop that should get executed sequentially, in the order that it would get executed in if the loop was not parallelized.
This means the threads will run the loop in parallel until they hit the `ordered` clause, at which point the code in the clause will be executed sequentially in the order of the loop.

Collapse Clause

The `collapse` clause is used to parallelize nested for loops. When a for loop is nested in another for loop, the `parallel for` pragma will, by default, only parallelize the outer loop. If the inner loop is also to be parallelized, one must add `collapse(2)` to the pragma. The `2` stands for the number of nested loops, so even deeper nests than two can be parallelized.
A side note: If $n_1, n_2, ...n_i$ are the number of iterations in each of the i nested loops, and all loops are collapsed, the total number of parallelized iterations will be $n_1 \cdot n_2 \cdot ... \cdot n_i$.

Reductions

The `reduction` clause is used to combine many values that were computed in parallel into a single result. The programmer specifies the name of the variables that should be combined and the operator to combine them.
Internally, each thread creates a local copy of the variable, initializes it with the appropriate value depending on the operator (`0` for `+`, `1` for `*`, ...) and updates it locally. Afterwards, all local copies are combined into the global variable.

Barrier

A barrier is a construct for synchronizing parallel programs. It is especially prevalent in the fork-join model of parallel programming.
Placing a barrier in the code forces all threads that reach it to stop until all other threads have also gotten there. It thus synchronizes all threads in a computation, and allows the programmer to be sure that after the barrier, all parallel computations have finished their work up to this point.

Differences in Library Routines

`omp_get_num_procs()` returns the number of logical cores that the program can use. `omp_get_max_threads()` returns the number of threads that a new `parallel` construct would use by default. It is set to the number of logical cores by default. `omp_get_num_threads()` returns the number of threads that are recruited into the team and executing the current `parallel` region.

Storage Attributes Private and Firstprivate

Both `private(var)` and `firstprivate(var)` are clauses that cause each thread to create it's own local copy of `var`. However, `private` does not initalize these local copies, while firstprivate initializes them with the value that the variable had in the previous serial part of the code.

Parallelized Computation of π

```
function integrate(thread_num, num_steps, width, sums)
    local_sum := 0
    for thread_num to num_steps schrittweite num_threads:
        addiere das integral an dieser stelle zu local_sum
    sums[thread_num] := local_sum

function main()
    num_steps := 10000000
    width := berechne width
    threads := array aus NUM_THREADS threads
    sums := array aus NUM_THREADS doubles

    starte alle threads mit integrate,
            übergebe dabei eine referenz zu sums
    joine alle threads wieder zusammen
    sum := summe über alle Elemente von sums
    pi := sum * 4 * width
    gebe pi aus
```