Session 2: Tasking

Christian Terboven, Paul Kapinos
IT Center, RWTH Aachen University
Seffenter Weg 23, 52074 Aachen, Germany
{terboven, kapinos}@itc.rwth-aachen.de

Abstract

This document guides you through the exercises. Please follow the instructions given during the lecture/exercise session on how to login to the cluster.

Linux: The prepared makefiles provide several targets to compile and execute the code:

- debug: The code is compiled with OpenMP enabled, still with full debug support.
- release: The code is compiled with OpenMP and several compiler optimizations enabled, should not be used for debugging.
- run: Execute the compiled code. The `OMP_NUM_THREADS` environment variable should be set in the calling shell.
- clean: Clean any existing build files.

The provided *.slurm is a batch script that you can use to submit a batch job to run on a Perlmutter compute node. Submit the job via command “sbatch *.slurm”, and check the output file after it is run.

You can also run an interactive batch job via “salloc” to get on a compute node. For example:

```
% salloc -N 1 -q interactive -C cpu -t 30:00
<will land on a compute node>
%  export OMP_NUM_THREADS=8
% ./mycode.exe
```

Please refer to https://github.com/NERSC/openmp-series-2024/blob/main/Session-1-Introduction/Using-OpenMP-Compilers-on-Perlmutter-CPUs-May2024.pdf for more details on Using various OpenMP compilers on Perlmutter and running jobs.

1 First steps with Tasks: Fibonacci

During this exercise you will examine the Tasking feature introduced in OpenMP 3.0.

Exercise 1: Go to the `fibonacci` directory. This code computes the Fibonacci number using a recursive approach – which is not optimal from a performance point of view, but well-suited for this exercise.

Examine the `fibonacci` code. Parallelize the code by using the Task concept of OpenMP 3.0. Remember: The *Parallel Region* should reside in main() and the fib() function should be entered the first time with one thread only. You can compile the code via '`make [debug|release]`'.

(optional) During the presentation you have heard that creating tasks after a certain roadblock is inefficient. Implement that idea into you code and stop creating new tasks when n is smaller than 30. For the last fibonacci numbers write a separate function which continues to compute in serial execution.

2 Reasoning about Work-Distribution

Go to the `for` directory. Compile the `for` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where *procs* denotes the number of threads to be used.

Exercise 1: Examine the code and think about where to put the parallelization directive(s). First, utilize the `task` construct.

Exercise 2: Examine the code and think about where to put the parallelization directive(s). Now, utilize the `taskloop` construct.

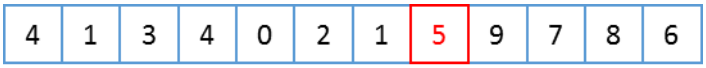
3 Quicksort

Quicksort is a recursive algorithm which, in this case, is used to sort an array of random integer numbers. How it works is described in the following steps.

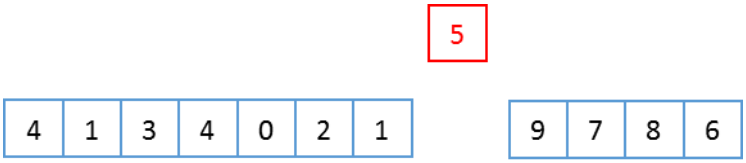
A pivot element is chosen. The value of this element is the point where the array is split in this recursion level.

5	8	1	7	4	9	2	1	0	3	4	6

All values smaller than the pivot element are moved to the front of the array, all elements larger than the pivot element to the end of the array. The pivot element is between both parts. Note, depending on the pivot element the partitions may differ in size.



Both partitions are sorted separately by recursive calls to quicksort.



The recursion ends, when the array reaches a size of 1, because one element is always sorted.

Go to the `quicksort` directory. Compile the `Quicksort` code via '`make [debug|release]`' and execute the resulting executable via '`OMP_NUM_THREADS=procs make run`', where procs denotes the number of threads to be used.

Exercise 1: The partitions created in step 3 can be sorted independent from each other, so this could be done in parallel. Use OpenMP Tasks to parallelize the quicksort program.

Exercise 2: Creating tasks for very small partitions is inefficient. Implement a cut-off to create tasks only if enough work is left. E.g. when more than 10k numbers have to be sorted, a task can be created, for smaller arrays no task is created.

Hint: You can add if clauses to the task pragmas.

Exercise 3: The if clause needs to be evaluated every time the function is called, although the array size does not exceed 10k elements on a lower level. Implement a serial_quicksort function and call this function when the array gets too small. This can help to avoid the overhead of the if clause.