OpenMP mini-app practical

Vasileios Karakasis

Swiss National Supercomputing Center (CSCS)

Slides prepared by Ben Cumming

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Adding OpenMP to the mini-app

- The aim of this practical is to take the serial mini-app implementation, and make it faster with OpenMP
  - while still getting the correct answer!

Hints

- Before starting find two sets of input parameters that converge for the serial version
  - note the time to solution
    - you want this to get faster as you add OpenMP
    - NOTE: but you might have to add quite a few directives before things actually get faster
  - note the number of conjugate gradient iterations
    - use this to check after you add each directive that you are still getting the right answer
    - NOTE: remember that there will be some small variations because floating point operations are not commutative

First test

- Get the code, by checking it out from github

```
> git pull
> cd <SummerSchool2017path>/miniapp/openmp
> ls
cxx fortran
> cd cxx
> make
> srun ./main 128 128 100 0.01
<note time to solution and conjugate gradient iterations>
> srun ./main 256 256 200 0.01
<note time to solution and conjugate gradient iterations>
```

I choose the C++ version here

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Step 1

- replace the welcome message in main.cpp/main.f90 with a message that tells the user
  - that this is the openmp version
  - how many threads it is using

```
> make
> OMP_NUM_THREADS=12 srun -n1 -c12 –hint=nomulithread ./main 256 256  200 0.01
...
=================================================
           Welcome to mini-stencil!  version ::
Fortran90 OpenMP with 12 threads
```

Step 2: Linear Algebra

- Open `linalg.cpp/f90` and add directives to the functions subroutines `ss_XXXX`
  - do one or two at a time
    - recompile frequently and run with 12 threads to check that you are still getting the right answer
- Once finished with that file, did your changes make any improvement?
  - compare the 128x128 and 256x256 results

Step 3: the diffusion stencil

- The final step is to parallelize the stencil operator in operators.cpp/f90.
- The nested for/do loop is an obvious target
  - it covers nx*ny grid points
- How about the boundary loops?

Step 4: testing

- how does it scale at different resolutions?
  - 32x32
  - 64x64
  - 128x128
  - 256x256
  - 512x512
  - 1024x1024
- Advanced C++:
  - can you implement first touch memory allocation in the C++ version?
    - requires adding just one OpenMP directive

Extras

- can you implement first touch memory allocation
  - requires adding just one OpenMP directive in C++
- does the stencil kernel vectorize?
  - look at Cray and Intel compiler vectorization reports
  - can you make it vectorize?

Thanks for your attention

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre