

# OpenMP mini-app practical

Ben Cumming

Swiss National Supercomputing Center (CSCS)



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 answers

# 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 answers
    - **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

```
> svn checkout <or> git pull
> cd <SummerSchool2014path>/miniapp/openmp
> ls
cxx fortran
> cd cxx
> make
> OMP_NUM_THREADS=1 aprun ./main -cc none 128 128 100 0.01
<note time to solution and conjugate gradient iterations>
> OMP_NUM_THREADS=1 aprun ./main -cc none 256 256 100 0.01
<note time to solution and conjugate gradient iterations>
```

I choose the C++ version here



# 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=8 aprun ./main -cc none 128 128 100 0.01
...
=====
                Welcome to mini-stencil!
version :: Fortran90 OpenMP with 8 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 often and run with 8 threads to check that you are still getting the right answer
    - it is harder if you do them all at once and then find an error
- 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  $n_x \times n_y$  grid points
- How about the boundary loops?
  - how many grid points does each one touch?

# Step 4: testing

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




# Thanks for your attention

- Good luck with the rest of the course.



# Code Walkthrough

- There are three modules of interest
  - [main.f90/main.cpp](#) : initialization and main time stepping loop
  - [linalg.f90/linalg.cpp](#) : the BLAS level 1 (vector-vector) kernels and conjugate gradient solver
  - [operators.f90/operators.cpp](#) : the stencil operator for the finite volume discretization



the vector-vector kernels and diffusion operator are the only kernels that have to be parallelized

# parameter notes

- Compile

```
> make
```

- Run interactively (use salloc beforehand)

```
> aprun ./main 128 128 100 0.01
```

- the grid is 128 x 128 grid points
- take 100 time steps
- run simulation for t=0.01

- Or run batch job

```
> sbatch job.todi  
... when job is finished ...  
> cat job.out
```



# Output

```
=====
version :
mesh
time :
```

The number of conjugate gradient iterations, which should always be constant for a given mesh size and time parameters. Can be used to check that changes to the code are still getting the correct result. **There will be small variations due to the imprecise nature of floating point operations.**

```
=====
step 1 required 4 iterations for residual 7.21951e-07
step 2 required 4 iterations for residual 7.9975e-07
...
step 99 required 12 iterations for residual 7.21951e-07
step 100 required 12 iterations for residual 7.21951e-07
```

time to solution

```
-----
simulation took 1.58408 seconds
8127 conjugate gradient iterations, at rate of 5130.43 iters/second
920 newton iterations
-----
```

Goodbye!

best way to compare different implementations



# Visualize the answer

- The application generates two data files with the final solution: `output.bin` and `output.bov`
- There is a script for automagically visualizes for you

```
> ls output.*
output.bin output.bov
> ./make_viz.sh
```

```
...
```

```
===== running visit to generate image =====
```

```
VisIt: Message - Saved phi_image.0001.png
```

```
===== drawing phi_image.0001.png =====
```

requires X-windowing  
make sure you connect with "ssh -X"

- configure programming environment
- call visualization code `Visit` to generate the image
- open `ImageMagick` to render the image

# Questions?

- You will become more familiar with the mini-app code over the summer-school.