Parallel Performance Analysis Tools

Gordon Gibb; g.gibb@epcc.ed.ac.uk

Reusing this material

Outline

- Motivations
- Discussion of CrayPAT and Scalasca
- Outline example code
- CrayPAT Usage
- Scalasca Usage

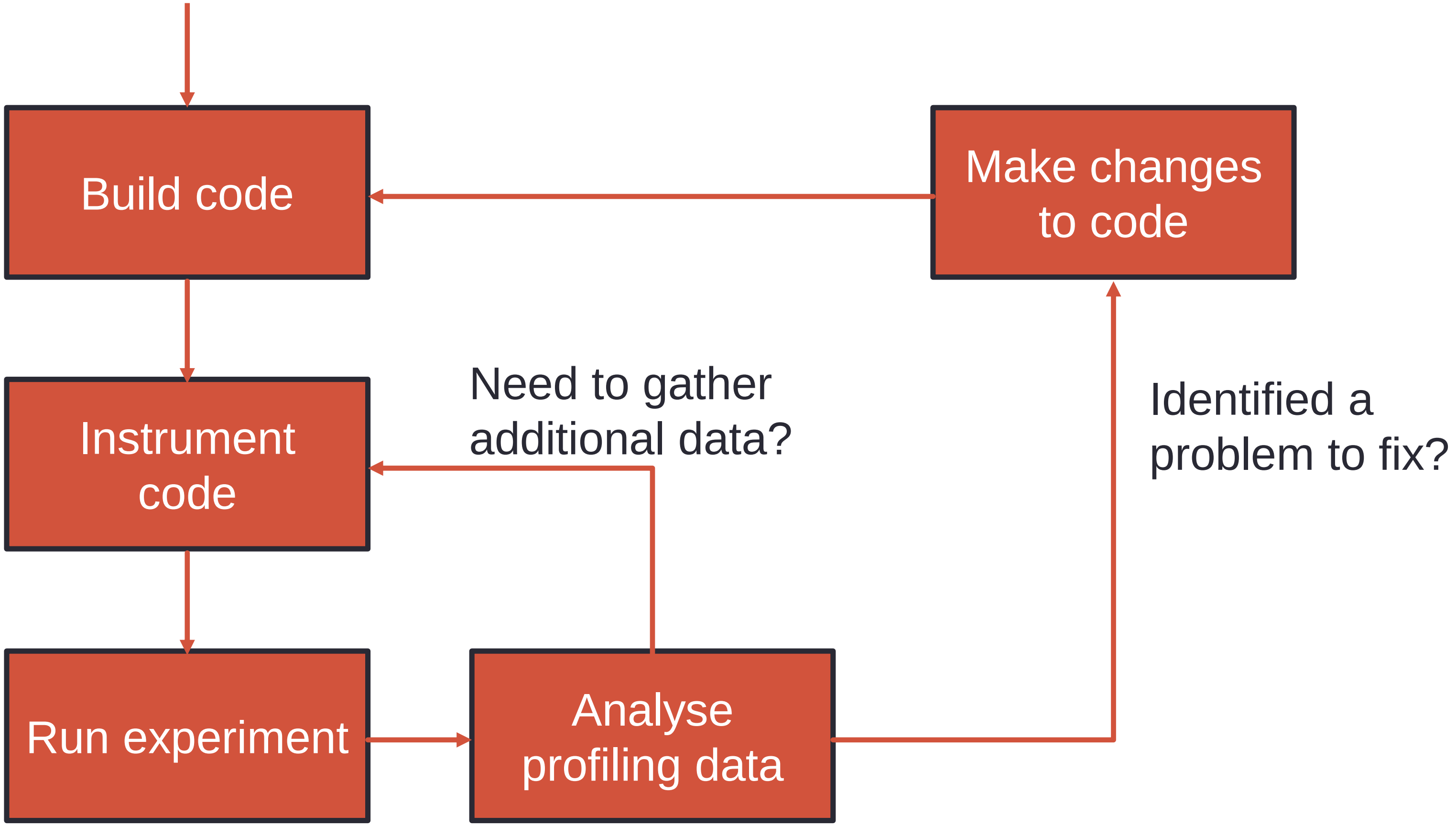
Motivations – What is Profiling?

- Examine the behaviour of the code

- Pick out any subroutines/functions that cause slowdown or have unusual behaviour

- Two types:
  1. Sampling (periodically queries running code to determine what function the code is in)
  2. Tracing (adds instructions into the code that report when entering/leaving functions, and various statistics)

Motivations – What is Profiling?



Picking an Example to Analyse

- Profiling generates a lot of extra data, and can cause your code to run more slowly
  - Need to choose a reasonably short example, but:
    - Program execution must be representative of a production run
    - Must be long enough to hide start-up and finalisation costs
    - Should include all the I/O of a normal job

- A good choice is something like a benchmark problem that takes a few minutes to run on a node/handful of nodes

Motivations - Why Profile?

- For developers:
  - Understand what the most time-consuming parts of the program are
  - Understand communication patterns and problems
    - E.g. load imbalance, synchronisation costs
  - Tool to help direct development efforts to give maximum benefits
- For users?
  - Understand why your program performs in a certain way
  - Help with choice of appropriate parameters, MPI processes…

Profilers: CrayPAT and Scalasca

- In this course we will consider two parallel performance analysis tools; CrayPAT and Scalasca

- With each tool you
  1. Instrument your code (typically during building)
  2. Run your code
  3. Analyse results

CrayPAT

+ Various levels of detail
+ Extreme customisibility for expert users

- Only available on Cray Platforms
- GUI is not particularly useful

Scalasca

+ Open source
+ Portable
+ Allows you to determine early/late senders etc…
+ Useful GUI (Cube)

- Unable to trace CUDA, SHMEM events or OpenMP nested parallelism

Example Test Code - CFD

- In this tutorial we will use a simple MPI code to demonstrate parallel performance analysis

- A computational fluid dynamics (CFD) code is employed, which calculates the flow of fluid within a cavity with an inlet in one side, and an outlet on another.

- The code can calculate the inviscid or viscid fluid flow.

Example Test Code - CFD

- Solves Poisson's Equation for the streamfunction:

$$\frac{\partial^2\psi}{\partial x^2}+\frac{\partial^2\psi}{\partial y^2}=-\zeta$$

$$u_x=\frac{\partial\psi}{\partial y};\;\; u_y=-\frac{\partial\psi}{\partial x};\;\; \zeta=\frac{\partial u_y}{\partial x}-\frac{\partial u_x}{\partial y}$$

- Available in both C and Fortran

Example Test Code - CFD

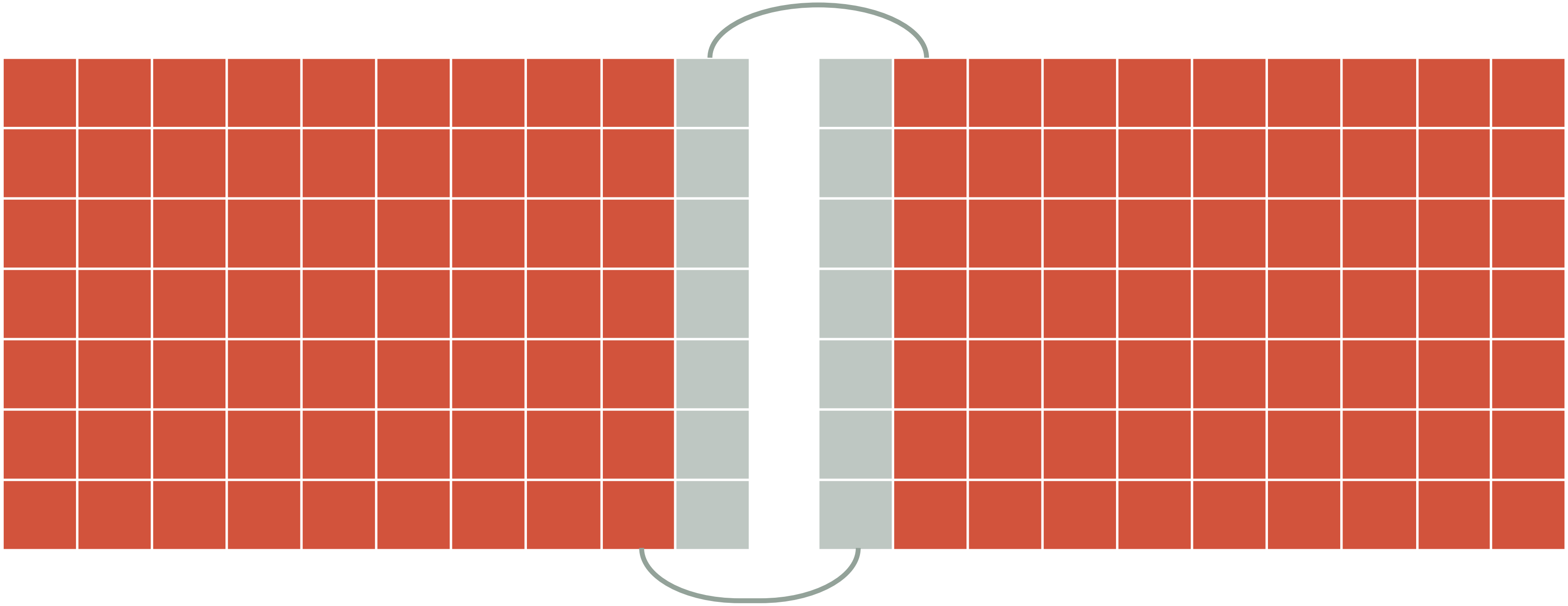
- Iterate until convergence

$$\psi_{i,j}^{\text{new}} = \frac{1}{4}\left(\psi_{i-1,j}^{\text{old}} + \psi_{i+1,j}^{\text{old}} + \psi_{i,j-1}^{\text{old}} + \psi_{i,j+1}^{\text{old}}\right) - \zeta_{i,j}^{\text{old}}$$

$$\begin{aligned}\zeta_{i,j}^{\text{new}} =& \frac{1}{4}\left(\zeta_{i-1,j}^{\text{old}} + \zeta_{i+1,j}^{\text{old}} + \zeta_{i,j-1}^{\text{old}} + \zeta_{i,j+1}^{\text{old}}\right)\\ &- \frac{R_e}{16}\left[\left(\psi_{i,j+1}^{\text{old}} - \psi_{i,j-1}^{\text{old}}\right)\left(\zeta_{i+1,j}^{\text{old}} - \zeta_{i-1,j}^{\text{old}}\right)\right.\\ &\left. - \left(\psi_{i+1,j}^{\text{old}} - \psi_{i-1,j}^{\text{old}}\right)\left(\zeta_{i,j+1}^{\text{old}} - \zeta_{i,j-1}^{\text{old}}\right)\right]\end{aligned}$$

Example Test Code - CFD

- Parallelised in the x (C) or y (Fortran) directions



- Halos transferred via MPI_Sendrecv

Example Test Code - CFD

- The code can be found on the course web pages

- To run it, use

```
aprun –n [nprocs] ./cfd <scale> <numiter> <Re>
```

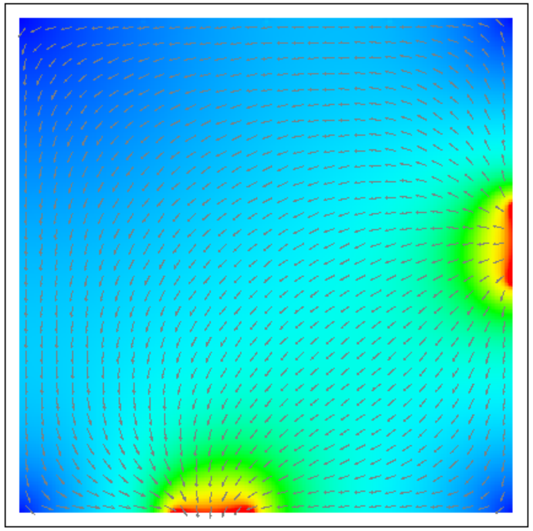
Where

- nprocs is the number of MPI processes
- scale scales the size of the box (32 x scale cells)
- numiter is the number of iterations
- Re (optional) is the Reynolds number ($0 \leq Re < 3.7$)

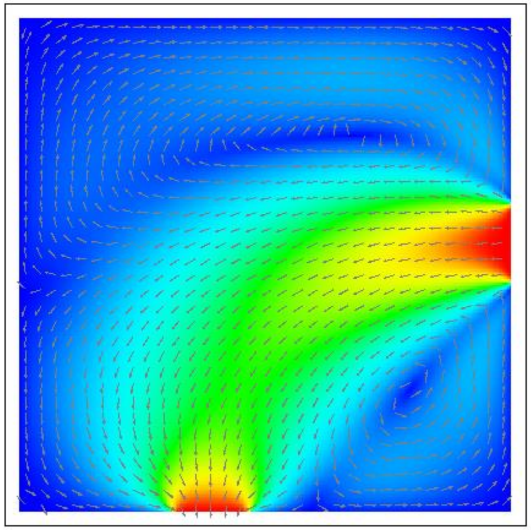
Example Test Code - CFD

- The output can be visualised using:

```
$ gnuplot –persist cfd.plt
```



$R_e = 0$



$R_e = 3.0$

Examples of Performance Tools

- I will now go onto demonstrate CrayPAT and Scalasca on ARCHER using the CFD code.

- Afterwards you will get an opportunity to try using CrayPAT/Scalasca yourselves

- For best results, it is recommended that you to login to ARCHER with an X-windows connection, e.g.

```
$ ssh –X [username]@login.archer.ac.uk
```

Using CrayPAT - Sampling

- Load the CrayPAT modules:

```
$ module load perftools-base
$ module load perftools
```

- Build executable as normal

```
$ make clean; make
```

- Instrument the binary using pat_build

```
$ pat_build ./cfd
```

Using CrayPAT - Sampling

- Instrumentation creates a new binary cfd+pat

- Modify the job submission script to run this new binary, then submit the job

  ```
  $ qsub submit.pbs
  ```

- This will run the cfd code with sampling

Using CrayPAT - Sampling

- Once the job has completed, it will have created an additional file: cfd+pat+<number>.xf

- Generate a human-readable report using pat_report

```
$ pat_report cfd+pat+<number>.xf
```

(You can put this information into a file by using the argument ‘–o <file>’)

Using CrayPAT - Sampling

```
Table 1:  Profile by Function


  Samp% |    Samp | Imb. |  Imb. |Group
        |         | Samp | Samp% | Function
        |         |      |       |  PE=HIDE

 100.0% | 1,906.5 |   -- |    -- |Total
|-----------------------------------------------
|  96.6% | 1,842.0 |   -- |    -- |USER
||----------------------------------------------
||  74.9% | 1,427.2 | 15.8 |  1.5% |jacobistepvort
||  21.0% |   401.0 |  8.0 |  2.6% |main
||==============================================
|   3.3% |    62.5 |   -- |    -- |MPI
||----------------------------------------------
||   3.1% |    58.5 | 25.5 | 40.5% |MPI_Sendrecv
|===============================================
```

Using CrayPAT - Sampling

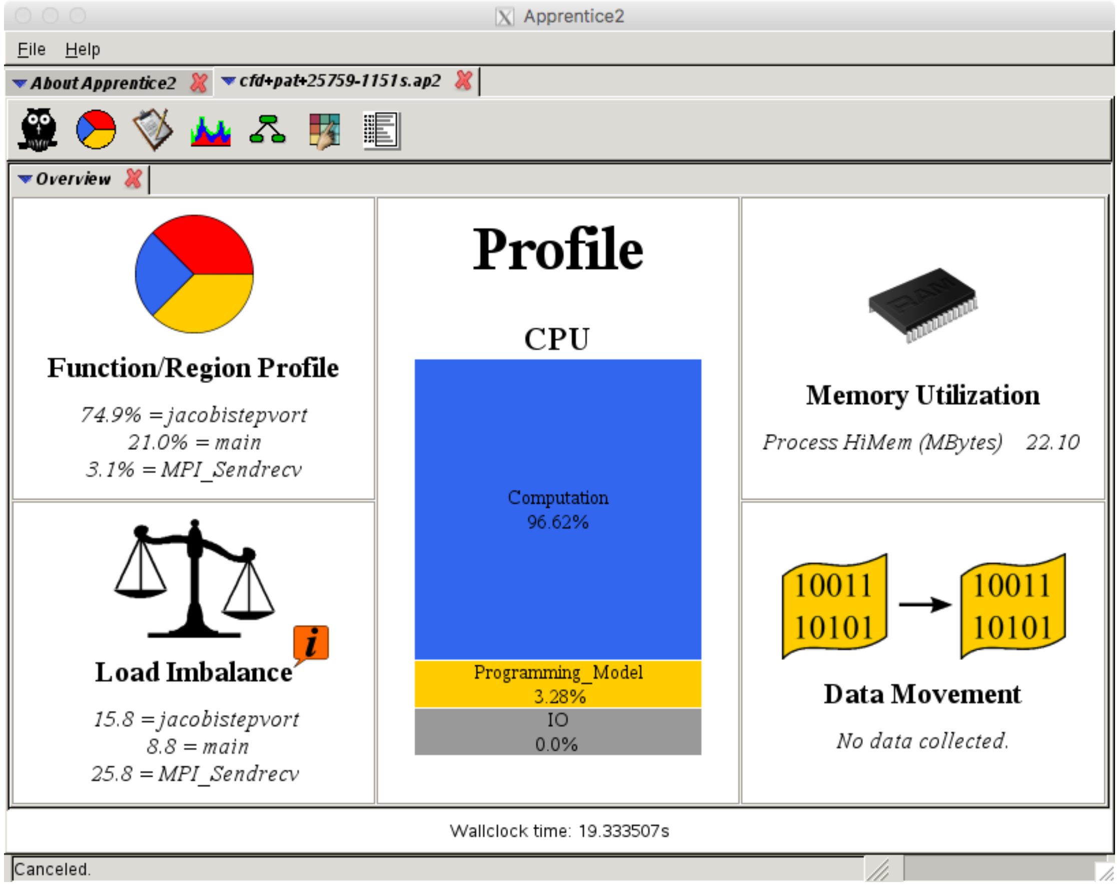
Pat_report also produces two other files; an .ap2 file, and an .apa file:

- The ap2 file acts as an input to the Apprentice2 graphical interface for viewing performance statistics

```
$ app2 <file>.ap2
```

- The apa file contains suggested configuration options for a traced experiment

Using CrayPAT – Apprentice2



Function/Region Profile

74.9% = jacobistepvort
21.0% = main
3.1% = MPI_Sendrecv

Profile

CPU

Computation 96.62%

Programming_Model 3.28%

IO 0.0%

Memory Utilization

Process HiMem (MBytes) 22.10

Load Imbalance

15.8 = jacobistepvort
8.8 = main
25.8 = MPI_Sendrecv

Data Movement

No data collected.

Wallclock time: 19.333507s

Canceled.

Using CrayPAT - Tracing

- Instrument the binary for tracing using the .apa file as an input to pat_build

```
$ pat_build -O cfd+pat+<number>.apa
```

- Modify the job submission script to use the new binary then submit the job

```
$ qsub submit.pbs
```

- View the results data using pat_report as before

```
$ pat_report cfd+apa+<number>.xf
```

- Then use Apprentice2 if desired

```
$ app2 cfd+apa+<number>.ap2
```

Using CrayPAT

- This process can be continued as necessary until the information you need has been obtained/you have gained the desired understanding of your code's performance

- More information on CrayPAT can be found using the commands

```
$ pat_help
$ man intro_pat
$ man pat_build
$ man pat_report
```

Using Scalasca - Sampling

- Load the Scalasca module

```
$ module load scalasca
```

- Instrumentation must be carried out during compilation by prepending scorep to the compiler. For example

```
$ scorep cc -c foo.c  or $ scorep ftn –c foo.f90
```

- Modify the compiler line in Makefile to include scorep:

```
CC = scorep cc
FC = scorep ftn
```

Using Scalasca - Sampling

- It is important to ensure that scorep is used during the linking of the object files.
- Functions/subroutines/files that you do not need/want to instrument do not need to be compiled with scorep
- Build the executable

```
make clean; make
```

Using Scalasca - Sampling

- Modify the submission script to launch the parallel job with scalasca –analyze, e.g.

```
scalasca –analyze aprun –np 4 ./cfd <options>
```

- Submit the job

```
$ qsub submit.pbs
```

- A measurement directory scorep_cfd_4_sum is created during the job's execution which contains all the log files

Using Scalasca - Sampling

- To analyse the output data, first run

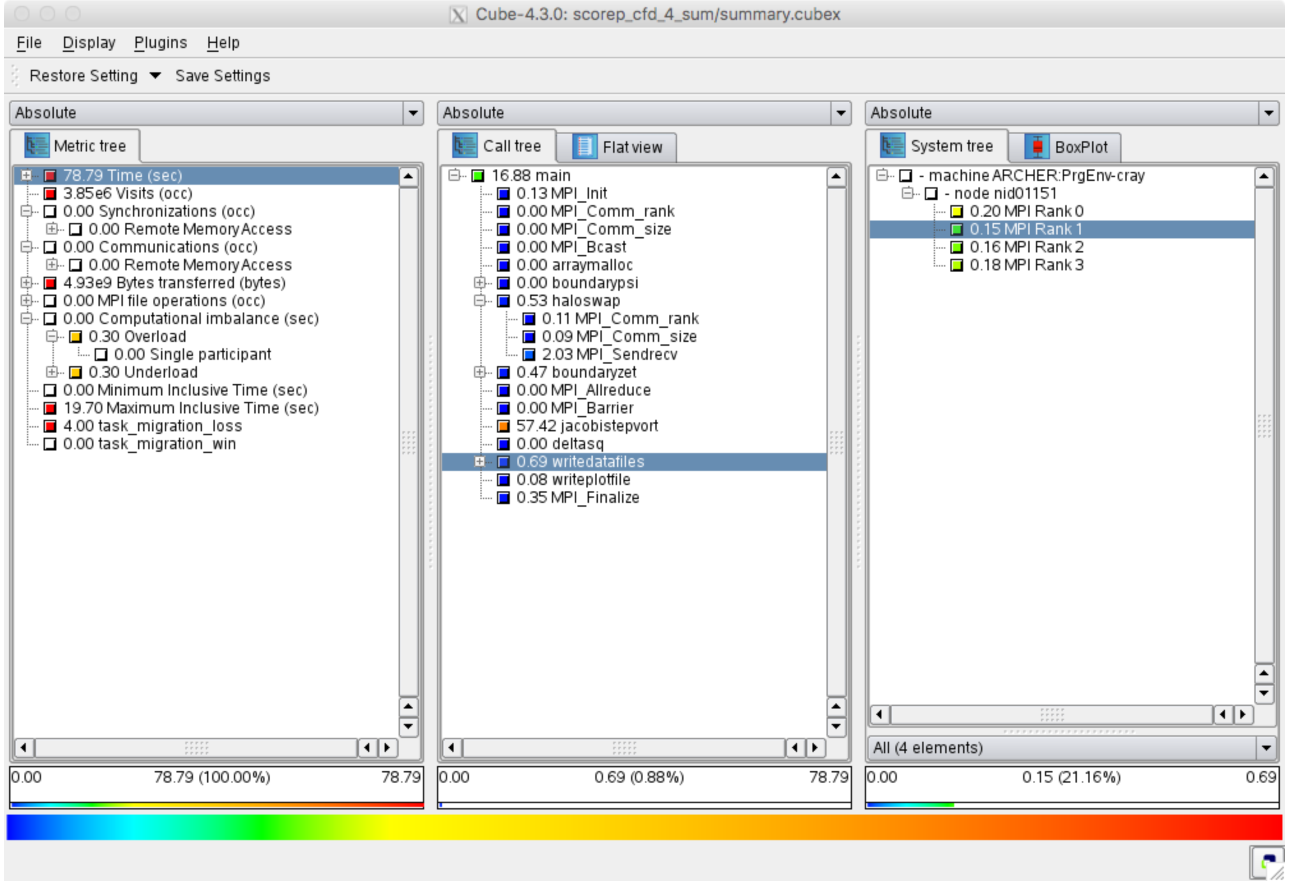
```
$ scalasca –examine scorep_cfd_4_sum
```

- This will open the cube browser, which allows you to examine the code’s timings

- Using the –s option produces a file (scorep.score) that can be used to advise you about setting up a tracing experiment

```
$scalasca –examine –s scorep_cfd_4_sum
```

Using Scalasca - Cube



Metric tree:

```
⊞ 🟥 78.79 Time (sec)
  🟥 3.85e6 Visits (occ)
⊟ □ 0.00 Synchronizations (occ)
    ⊞ □ 0.00 Remote Memory Access
⊟ □ 0.00 Communications (occ)
    ⊞ □ 0.00 Remote Memory Access
⊞ 🟥 4.93e9 Bytes transferred (bytes)
⊞ □ 0.00 MPI file operations (occ)
⊟ □ 0.00 Computational imbalance (sec)
    ⊟ 🟨 0.30 Overload
        □ 0.00 Single participant
    ⊞ 🟨 0.30 Underload
  □ 0.00 Minimum Inclusive Time (sec)
  🟥 19.70 Maximum Inclusive Time (sec)
  🟥 4.00 task_migration_loss
  □ 0.00 task_migration_win
```

Call tree:

```
⊟ 🟩 16.88 main
    🟦 0.13 MPI_Init
    🟦 0.00 MPI_Comm_rank
    🟦 0.00 MPI_Comm_size
    🟦 0.00 MPI_Bcast
    🟦 0.00 arraymalloc
  ⊞ 🟦 0.00 boundarypsi
  ⊟ 🟦 0.53 haloswap
        🟦 0.11 MPI_Comm_rank
        🟦 0.09 MPI_Comm_size
        🟦 2.03 MPI_Sendrecv
  ⊞ 🟦 0.47 boundaryzet
    🟦 0.00 MPI_Allreduce
    🟦 0.00 MPI_Barrier
    🟧 57.42 jacobistepvort
    🟦 0.00 deltasq
  ⊞ 🟦 0.69 writedatafiles
    🟦 0.08 writeplotfile
    🟦 0.35 MPI_Finalize
```

System tree:

```
⊟ □ - machine ARCHER:PrgEnv-cray
    ⊟ □ - node nid01151
          🟨 0.20 MPI Rank 0
          🟩 0.15 MPI Rank 1
          🟩 0.16 MPI Rank 2
          🟩 0.18 MPI Rank 3
```

All (4 elements)

0.00	78.79 (100.00%)	78.79
0.00	0.69 (0.88%)	78.79
0.00	0.15 (21.16%)	0.69

Using Scalasca - Tracing

Examining the scorep.score file in the measurement directory reveals information on the estimated final disk usage and memory usage of a trace

```
Estimated aggregate size of event trace:                    128MB
Estimated requirements for largest trace buffer (max_buf): 32MB
Estimated memory requirements (SCOREP_TOTAL_MEMORY):       34MB
(hint: When tracing set SCOREP_TOTAL_MEMORY=34MB to avoid intermediate flushes
 or reduce requirements using USR regions filters.)


type  max_buf[B]    visits    time[s]  time[%] time/visit[us]  region
ALL   33,493,662   3,848,767   78.79    100.0          20.47   ALL
MPI   22,401,846   2,000,134    2.95      3.7           1.47   MPI
USR    7,491,672   1,248,609   57.90     73.5          46.37   USR
COM    3,600,144     600,024   17.95     22.8          29.91   COM
```

Using Scalasca - Tracing

- To trace the code, alter your job submission script to contain:

```
scalasca –analyze –q –t aprun –np 4 ./cfd <options>
```

- Don’t forget to also set SCOREP_TOTAL_MEMORY in the script as suggested in the .score file:

```
export SCOREP_TOTAL_MEMORY=34MB
```

Using Scalasca - Tracing

- A new directory scorep_cfd_4_trace is created, and the results can be examined using

```
$ scalasca -examine scorep_cfd_4_trace
```

- This time, more information is present, such as that on late senders/receivers.

Using Scalasca - Tracing

- If the estimated disk/memory usage for tracing is too high, you may need to consider to avoid tracing certain functions by using a filter file:

```
SCOREP_REGION_NAMES_BEGIN
  EXCLUDE
     jacobistepvort
     MPI_Sendrecv
SCOREP_REGION_NAMES_END
```

- Usage:

```
scalasca –examine –f filter.txt aprun ...
scalasca –analyze –q –t –f filter.txt aprun ...
```

Using Scalsca

- More information can be found on the Scalasca website

  http://www.scalasca.org

- In particular their user's guide:

  http://apps.fz-juelich.de/scalasca/releases/scalasca/2.3/docs/UserGuide.pdf

Practical: CFD

- Try out using CrayPAT and/or Scalasca to investigate the performance of the CFD code

- Options:
    - Try using different values for scale, and investigate turning viscosity on and off
    - How does the profile change when running on large numbers of processes?
    - Terminate calculation based on a tolerance value (see comments in code), investigate only computing this infrequently
    - Investigate using serialised Send / Receive functions (see alternative boundary source files) instead of Sendrecv