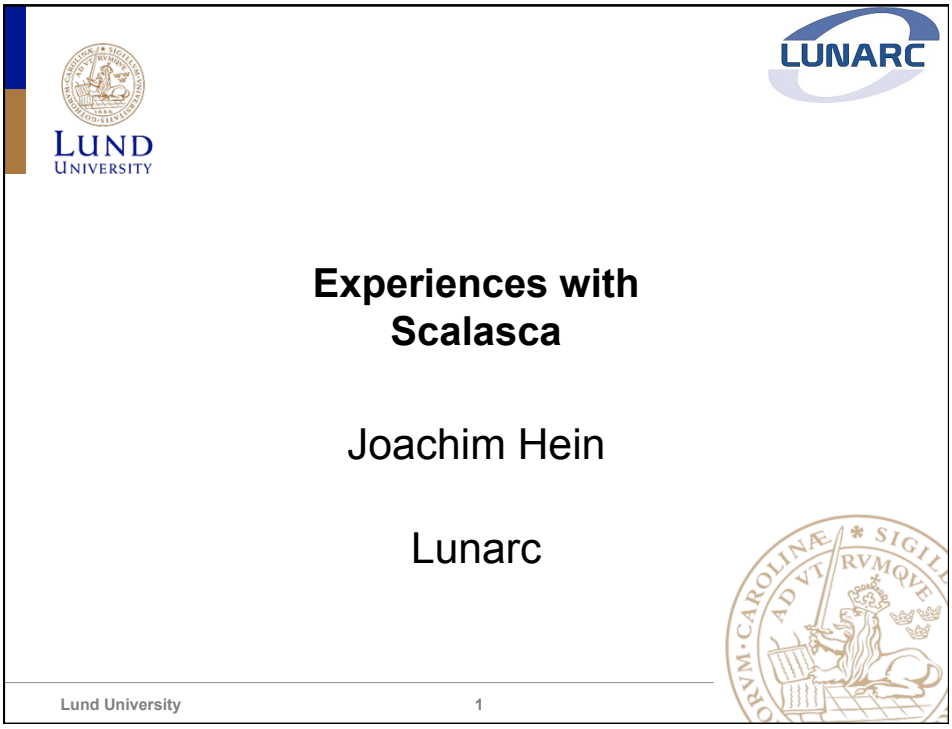
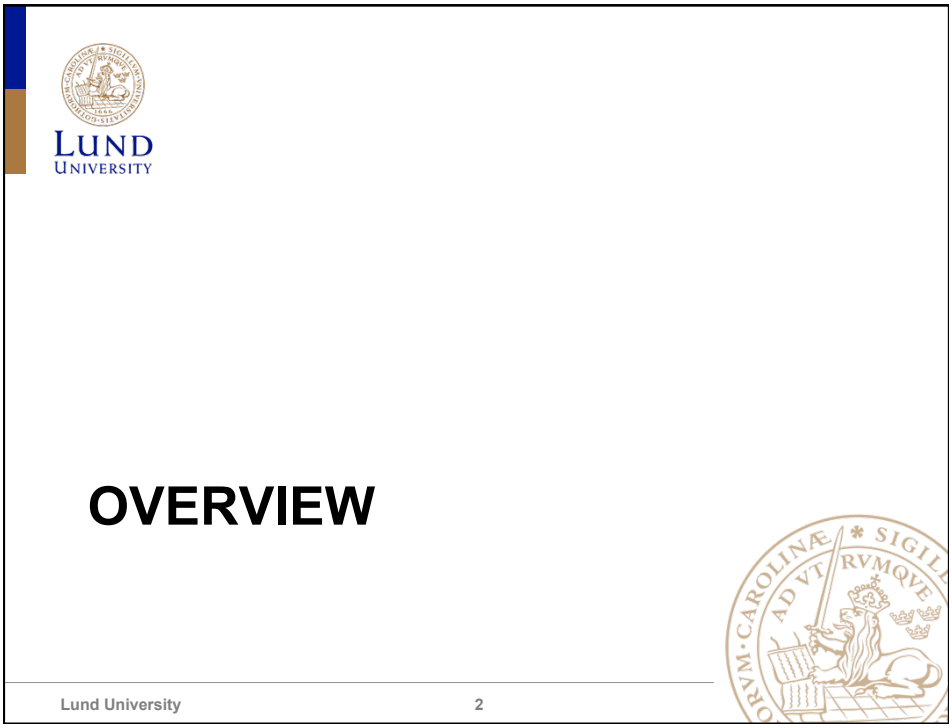

LUND
UNIVERSITY
LUNARC
Experiences with
Scalasca
Joachim Hein
Lunarc
Lund University 1


LUND
UNIVERSITY
OVERVIEW
Lund University 2

Overview

- Parallel profiling tool
  - OpenMP
  - MPI
  - Can be used for serial
- Aims to help with questions like
  - Where is my application spending time
  - Why is it spending time on this
- Fast and convenient to use

Overview (cont.)

- Developed:
  - Forschungszentrum Jülich (Germany)
  - German Research School for Simulation Sciences
- Free but copyright
- Widely available – architectures include:
  - Cray XT/XE
  - IBM Bluegene
  - IBM Power
  - “various” Linux x86/x64 clusters

Compilers

- Works closely with the compiler: C, C++, Fortran
- Supported compilers (some contain “small print”)
  - GCC
  - IBM xlc, xlC, xlf
  - PGI
  - Intel
  - SUN (Fortran only)
  - PathScale compiler (which?)
  - CCE/Cray
  - NEC compiler

Scalasca installations available at Lunarc

- Installed and testing
  - Platon@lunarc
    - Scalasca 1.3.2
    - gcc 4.4
    - openmpi 1.4.1
  - Alarik@lunarc
    - Scalasca 1.4.1/1.4.2
    - gcc 4.6.2
    - openmpi 1.4.4
    - papi 4.2.1

What does it attempt to do

- Event based, e.g.:
  - MPI call starts
  - OpenMP parallel region finishes
  - Subroutine “init” starts
- Recording
  - Runtime summation (time spend in routine)
  - Event tracing (huge files)
- GUI report browser to view the results
  - Some other tools can also digest the data

Three step approach

1. Instrument your program (Source access required)
   - Automatically during compile
   - Manually using an API (Source modification)
   - PDT instrumentation
2. Execute the program – perform the analysis
   - Events are recorded while executing
   - Check: runtime still reasonable
3. Look at the results – examine the result
   - Load into the GUI

Step 1

INSTRUMENTATION

Automatic instrumentation

- Pre-fix your compiler call with either of

```
scalasca -instrument
skin
```

- Example:

```
mpif90 –o myprog.x myprog.f90
```

becomes:

```
scalasca -instrument mpif90 –o myprog.x myprog.f90
skin mpif90 –o myprog.x myprog.f90
```

You get feed back OpenMP example:

```
[jhein@alarik ScalascaPlay]$ skin gcc -O3 -fopenmp -std=c99 \
> -o gauss2d_dynamic.x gauss2d_dynamic.c -lm
INFO: Instrumented executable for OMP measurement
[jhein@alarik ScalascaPlay]$
```

Step 2

ANALYSE APPLICATION PERFORMANCE

Use the analysis tool to run the executable

- MPI example

```
scalasca –analyze mpiexec –np 4 myprog.x

scan mpiexec –np 4 myprog.x
```

- OpenMP code (or serial)

```
scalasca –analyze myprog.x

scan mpiprog.x
```

Comments

- The above creates a summary report
- Have a look at the runtime
  - small routines can make the runtime explode
  - if it is bad: use a filter file
  - if it is really bad: compile relevant file without instrumentation
- The next step can give you hint on culprits
  - Look for high invocation count

Filterfile

- Name routines not to measure in filter file
- Invoke analyser:

```
scan –f filterfile mpiexec –np 128 myprog.x
```

Fortran example: filter file for gcc 4.6.2

```
[jhein@alarik Benchmark]$ more not_to_measure.filt
__particles_sub_MOD_get_rhopswarm_point
__general_MOD_keep_compiler_quiet_i
__general_MOD_keep_compiler_quiet_r4d
__general_MOD_keep_compiler_quiet_i2d
__general_MOD_random_number_wrapper_0
__general_MOD_random_number_wrapper_1
__general_MOD_ran0
__particles_map_MOD_interpolate_quadratic_spline
__particles_MOD_get_frictiontime
```

Result directory

- After the `scan`-run you will find a result directory in your workdirectory
- Example: `epik_gauss2d_dynamic_0x16_sum`
  - Executable: gauss2d_dynamic
  - 16 OpenMP threads
  - summary run
- By default: `scan` aborts when result-directory exists

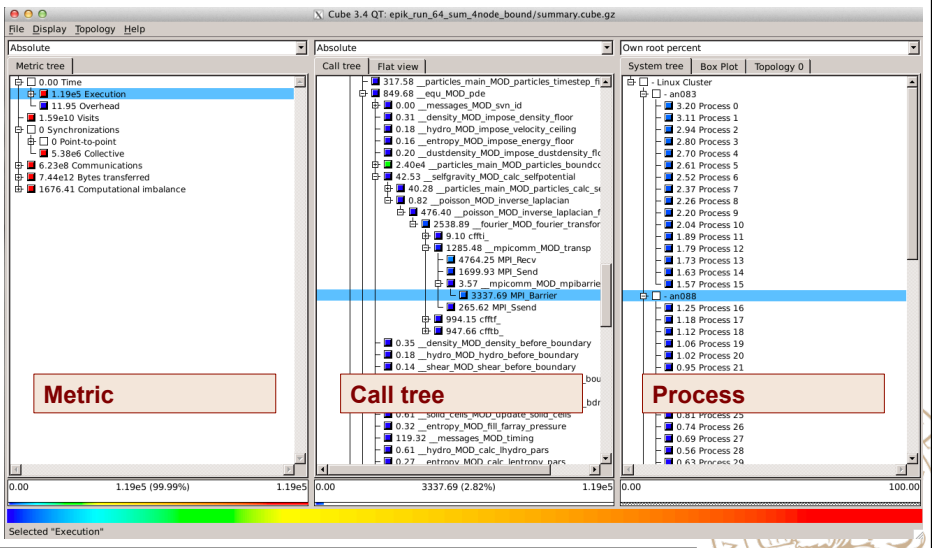
Step 3

EXAMINE WITH THE GUI

Starting the GUI

- The command:
  `scalasca -examine epik_`*`directory`*
  `square epik_`*`directory`*
  - post-processes the data (pattern search)
  - starts the GUI
- Use `-s` option for post-processing only
  - Useful when opening archive from the GUI

GUI window: Time spent in MPI_Barrier



OpenMP case study

VECTOR NORM

Initialisation routine

```
void vectorinit(double *vdata, int leng)

{
#pragma omp master // should have been: for
  for ( int i = 0; i < leng; i++)
    {
      vdata[i] = i;
    }
  return;
}
```

Vectornorm routine

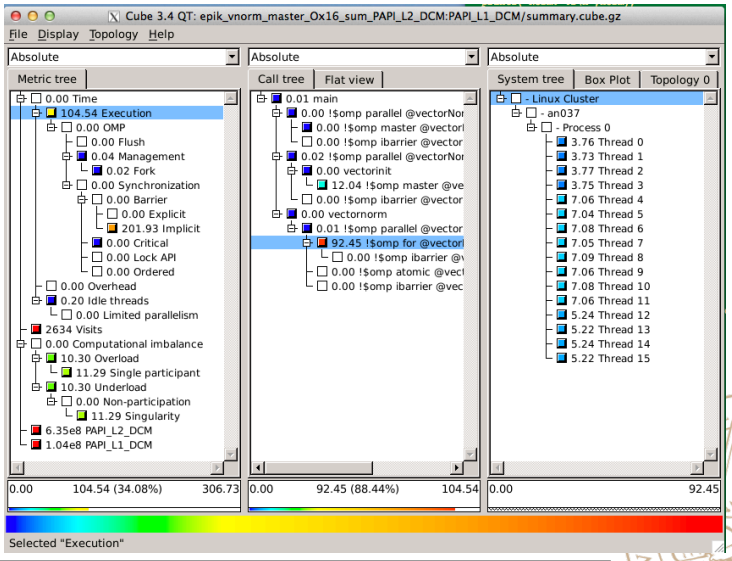
```
double vectornorm(double *vdata, int leng)
{
  double norm = 0.0;
#pragma omp parallel default(none) shared(vdata, norm, leng)
  { double lnorm =0.0;
#pragma omp for
    for (int i = 0; i<leng; i++)
      {  lnorm += vdata[i] * vdata[i];
      }
#pragma omp atomic
    norm += lnorm;
  }
  norm = sqrt(norm);
  return norm;
}
```

Code sequence from caller

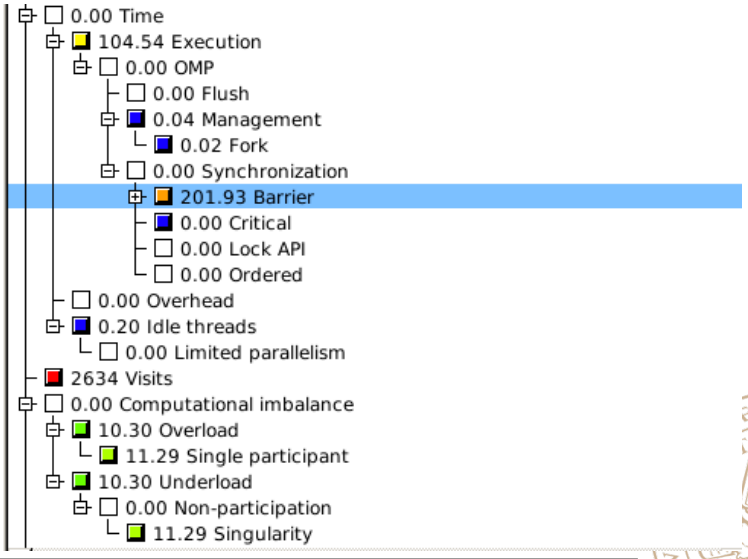
```
 for (int loops=0; loops < loop_count; loops++)
    {
#pragma omp parallel default(none) shared(vect, vleng)
      {
        vectorinit( vect, vleng);
      }

      norm = vectornorm(vect, vleng);
    }
```

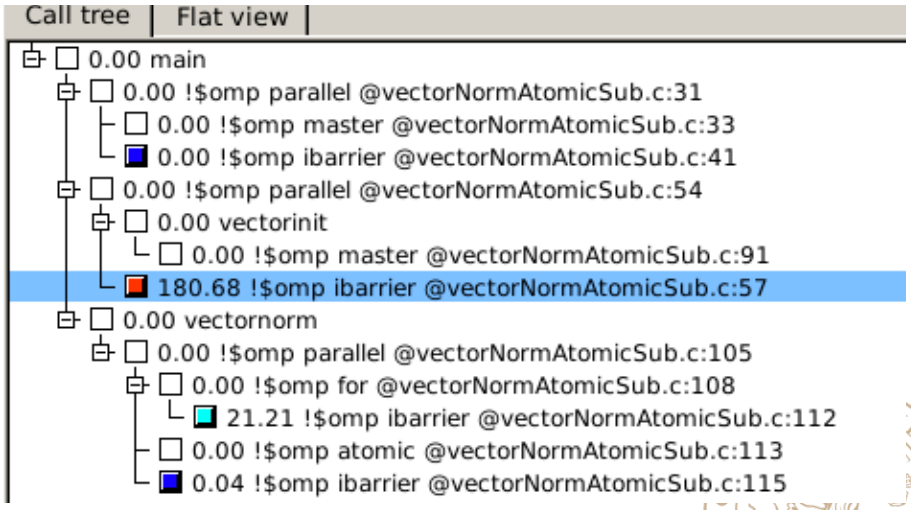
GUI after unfolding



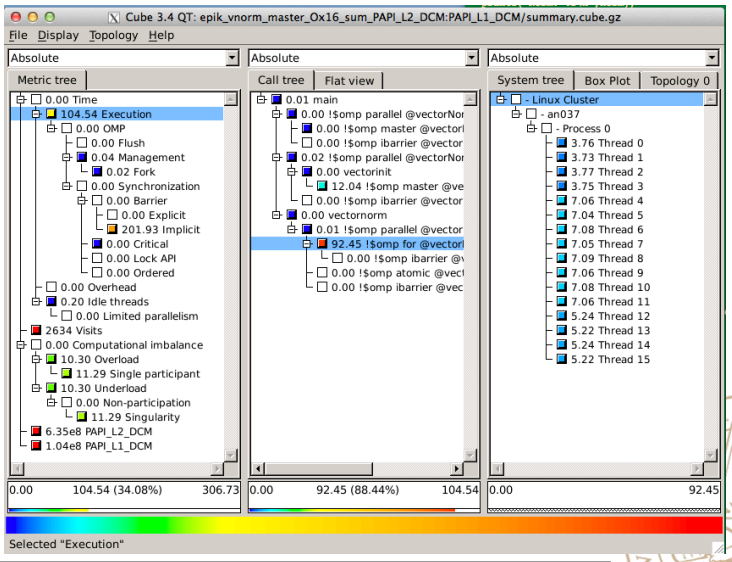
Details on “Metric” window (left)



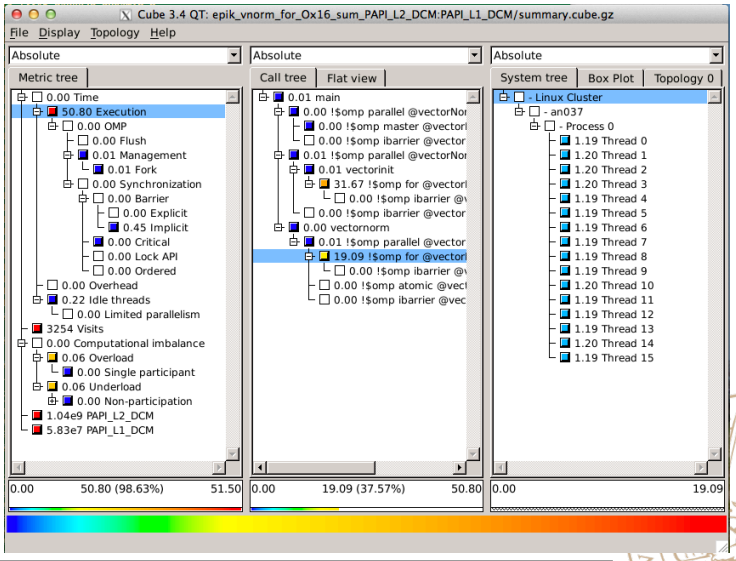
Call tree window (centre)



GUI after unfolding – before fix



GUI after unfolding – after fix



LUND UNIVERSITY

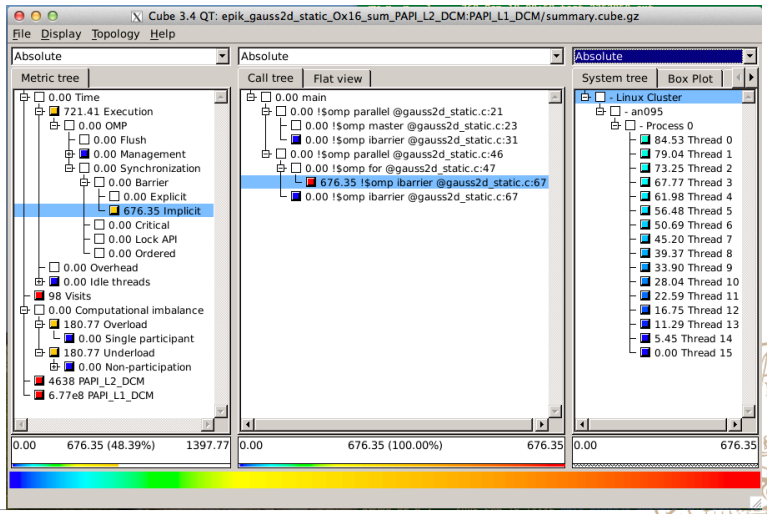
OpenMP case study

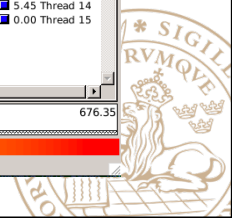
SUMMING A TRIANGULAR REGION

Code:

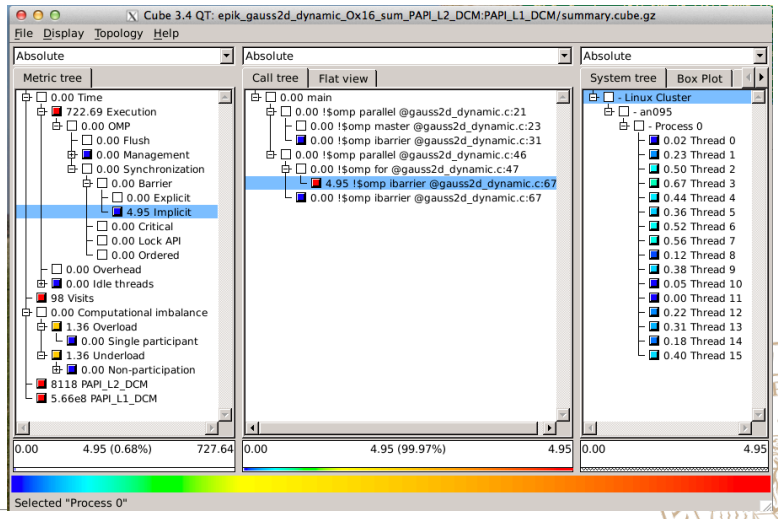
```
#pragma omp parallel default(none) shared(integral)
#pragma omp for private(xpos, ypos) \
  reduction(+:integral)  schedule(static)
  for ( int i = 0; i < NSIZE; i++)
    { xpos = (i+0.5) * STEPSIZE;
      for (int j = 0; j < i; j++ )
       {
         ypos = (j+0.5) * STEPSIZE;
         double xvecSquare = xpos*xpos + ypos*ypos;
         integral += exp(-xvecSquare) * STEPSIZE*STEPSIZE;
       }
    }
```

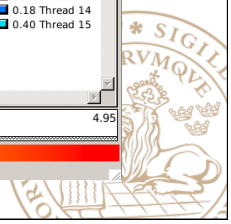
GUI window: Static schedule





GUI window: Dynamic schedule (fixed)



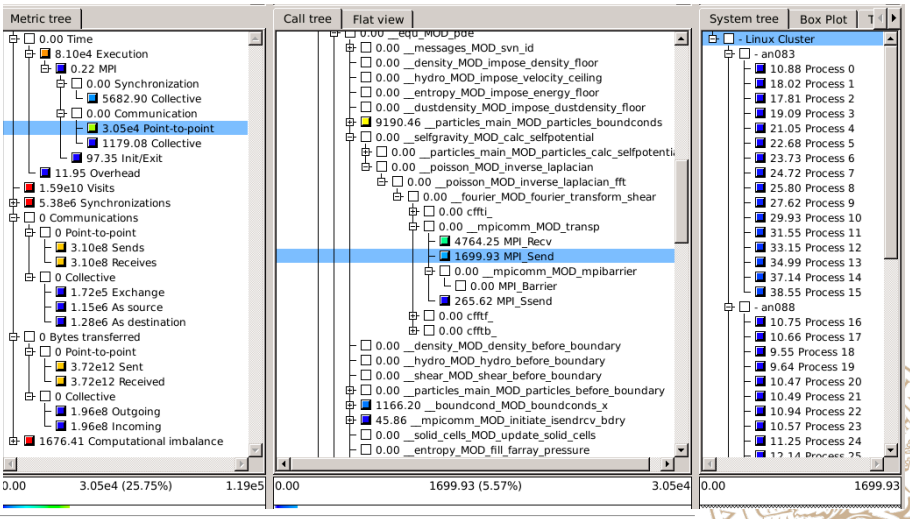


LUND UNIVERSITY

MPI example

PENCIL CODE

GUI for Fortran-MPI code: Pencil



Things I am missing

- Support for OpenMP tasks (scalasca 1.4.2, July 2012)
- Sampling – this helps with OO-style coding
  - Though one would loose some features

Further features

- PAPI event counters integrated
  - Specify `scan` option: `-m`
  - Treat them with great care
    - e.g. L2 miss on streams
    - Hardware event counter problem, not Scalasca
- When using tracing you get more features
  - e.g.: “late sender” or “late receiver” in MPI P2P
  - Care with memory consumption and disk-space
  - Care with not-so-global clock

Summary

- Portable and free tool
  - IBM and Cray in-house tools unavailable on other kit
- You can learn a lot about a code
- Definitely got more robust with time
- In theory easy to use: 3 step approach
  - Need to exclude small methods (e.g. C++)
  - Sometimes it just breaks (so do other tools)