Latest versions available on Github:

UL HPC tutorials: https://github.com/ULHPC/tutorials
UL HPC School: http://hpc.uni.lu/hpc-school/
1 Introduction

2 Debugging and profiling tools

3 Conclusion
Main Objectives of this Session

This session is meant to show you some of the various tools you have at your disposal on the UL HPC platform to:

understand + solve development & runtime problems
This session is meant to show you some of the various tools you have at your disposal on the UL HPC platform to:

understand + solve development & runtime problems

During the session we will:

- discuss what happens when an application runs out of memory and how to discover how much memory it actually requires.
- see debugging tools that help you understand why your code is crashing.
- see profiling tools that show the bottlenecks of your code - and how to improve it.
This session is meant to show you some of the various tools you have at your disposal on the UL HPC platform to:

**understand + solve development & runtime problems**

During the session we will:
- discuss what happens when an application runs **out of memory** and how to discover how much memory it actually requires.
- see **debugging tools** that help you understand **why your code is crashing**.
- see **profiling tools** that show the **bottlenecks of your code** - and how to improve it.

**Knowing what to do when you experience a problem is half the battle.**
Summary

1 Introduction

2 Debugging and profiling tools

3 Conclusion
Debugging and profiling tools

Tools at your disposal (I)

Common tools used to understand problems

- Do you know what time it is?
  - `/usr/bin/time -v` is just magic sometimes

- Don’t remember where you put things?
  - `Valgrind` can help with your memory issues

- Is your application firing on all cylinders?
  - with `htop` green means go! (red is bad)

- Got stuck?
  - `strace` can tell you where you are and how you got there

Some times simple tools help you solve big issues.
Debugging and profiling tools

Tools at your disposal (II)

HPC specific tools - Arm (prev. Allinea)

- Arm DDT (part of Arm Forge)
  - Visual debugger for C, C++ and Fortran threaded and // code
- Arm MAP (part of Arm Forge)
  - Visual C/C++/Fortran profiler for high performance Linux code
- Arm Performance Reports
  - Application characterization tool
Debugging and profiling tools

Tools at your disposal (II)

**HPC specific tools - Arm (prev. Allinea)**

- Arm DDT (part of Arm Forge)
  - Visual debugger for C, C++, and Fortran threaded and // code
- Arm MAP (part of Arm Forge)
  - Visual C/C++/Fortran profiler for high performance Linux code
- Arm Performance Reports
  - Application characterization tool

**Arm tools are licensed**

- License check integrated in SLURM: `scontrol show license`
- Ask for licenses at job submission with e.g. `srun -L forge:16`

2017 software set lists the Arm tools under the previous *Allinea* name, the 2018 set will have them under *Arm*. 
HPC specific tools - Intel

- **Intel Advisor**
  - Vectorization + threading advisor: check blockers and opport.

- **Intel Inspector**
  - Memory and thread debugger: check leaks/corrupt., data races

- **Intel Trace Analyzer and Collector**
  - MPI communications profiler and analyzer: evaluate patterns

- **Intel VTune Amplifier**
  - Performance profiler: CPU/FPU data, mem. + storage accesses

Intel tools are licensed. All come as part of Intel Parallel Studio XE - Cluster edition!
Debugging and profiling tools

Tools at your disposal (III)

HPC specific tools - Intel

- Intel Advisor
  - Vectorization + threading advisor: check blockers and opport.
- Intel Inspector
  - Memory and thread debugger: check leaks/corrupt., data races
- Intel Trace Analyzer and Collector
  - MPI communications profiler and analyzer: evaluate patterns
- Intel VTune Amplifier
  - Performance profiler: CPU/FPU data, mem. + storage accesses

Intel tools are licensed

All come as part of Intel Parallel Studio XE - Cluster edition!
Debugging and profiling tools

Tools at your disposal (IV)

HPC specific tools - Scalasca & friends

- **Scalasca**
  - Study behavior of // apps. & identify optimization opport.

- **Score-P**
  - Instrumentation tool for profiling, event tracing, online analysis.

- **Extra-P**
  - Automatic performance modeling tool for // apps.

Free and Open Source!

See other awesome tools at http://www.vi-hps.org/tools
HPC specific tools - Scalasca & friends

- Scalasca
  → Study behavior of // apps. & identify optimization opport.

- Score-P
  → Instrumentation tool for profiling, event tracing, online analysis.

- Extra-P
  → Automatic performance modeling tool for // apps.

Free and Open Source!
See other awesome tools at http://www.vi-hps.org/tools
DDT features

- **Parallel debugger**: threads, OpenMP, MPI support
- Controls processes and threads
  - step code, stop on var. changes, errors, breakpoints
- Deep **memory debugging**
  - find memory leaks, dangling pointers, beyond-bounds access
- C++ debugging – including STL
- Fortran – including F90/F95/F2008 features
- See vars/arrays **across multiple processes**
- Integrated editing, building and **VCS integration**
- Offline mode for **non-interactive debugging**
  - record application behavior and state

Full details at Arm HPC Tools: Forge-DDT
Debugging and profiling tools

Arm DDT - on ULHPC

Modules
- On all clusters: module load tools/AllineaForge
- Caution! May behave differently between:
  - Debian+OAR (Gaia, Chaos) and CentOS+SLURM (Iris)

Debugging with DDT

1. Load toolchain, e.g. (for Intel C/C++/Fortran, MPI, MKL):
   - module load toolchain/intel

2. Compile your code, e.g. mpiicc $code.c -o $app

3. Run your code through DDT (GUI version)
   - iris: ddt srun ./$app
   - gaia/chaos: ddt mpirun -hostfile $OAR_NODEFILE ./$app

4. Run DDT in batch mode (no GUI, just report):
   - ddt --offline -o report.html --mem-debug=thorough ./$app
Debugging and profiling tools

Arm DDT - interface
Debugging and profiling tools

Arm MAP - highlights

MAP features

- Meant to show developers **where&why code is losing perf.**
- **Parallel profiler**, especially made for MPI applications
- Effortless profiling
  - no code modifications needed, may not even need to recompile
- Clear **view of bottlenecks**
  - in I/O, compute, thread or multi-process activity
- Deep insight in **CPU instructions affecting perf.**
  - vectorization and memory bandwidth
- **Memory usage over time** – see changes in memory footprint
- Integrated editing and building as for DDT

Full details at Arm HPC Tools: Forge-MAP
Debugging and profiling tools

Arm MAP - on ULHPC

Modules
- On all clusters: module load tools/AllineaForge
- Caution! May behave differently between:
  ✠ Debian+OAR (Gaia, Chaos) and CentOS+SLURM (Iris)

Profiling with MAP

1. Load toolchain that built your app., e.g.
   → module load toolchain/intel

2. Run your code through MAP (attached, GUI version)
   → iris: map srun ./app
   → gaia/chaos: map mpirun -hostfile $OAR_NODEFILE ./app

3. Run MAP in batch mode (no GUI, create .map file):
   → iris: map --profile srun ./app
Debugging and profiling tools

Arm MAP - interface

Input/Output | Project Files | Main Thread Stacks | Functions
---|---|---|---

Main Thread Stacks

<table>
<thead>
<tr>
<th>Total core time</th>
<th>MPI</th>
<th>Function(s) on line</th>
</tr>
</thead>
<tbody>
<tr>
<td>67.4%</td>
<td>MPI</td>
<td>main</td>
</tr>
<tr>
<td>31.1%</td>
<td>MPI_Finalize</td>
<td></td>
</tr>
<tr>
<td>1.9%</td>
<td>MPI_Barrier</td>
<td></td>
</tr>
<tr>
<td>1.3%</td>
<td>MPI_Finalize</td>
<td></td>
</tr>
<tr>
<td>&lt;0.1%</td>
<td>Others</td>
<td></td>
</tr>
<tr>
<td>&lt;0.1%</td>
<td>Others</td>
<td></td>
</tr>
</tbody>
</table>

Showing data from 1,176 samples taken over 28 processes (42 per process)
Debugging and profiling tools

Arm Perf. Reports - highlights

Performance Reports features

- Meant to answer **How well do your apps. exploit your hw.?**
- Easy to use, on unmodified applications
  - outputs HTML, text, CSV, JSON reports
- One-glance view if application is:
  - well-optimized for the underlying hardware
  - running **optimally at** the given **scale**
  - affected by I/O, networking or threading **bottlenecks**
- Easy to integrate with continuous testing
  - programatically improve performance by continuous profiling
- **Energy metric** integrated
  - using RAPL (CPU) for now on iris
  - IPMI-based monitoring may be added later

Full details at Arm HPC Tools: Perf. Reports

V. Plugaru & UL HPC Team (University of Luxembourg)
Debugging and profiling tools

Arm Perf. Reports - on ULHPC

Modules

- On all clusters: module load tools/AllineaReports
- Caution! May behave differently between:
  - Debian+OAR (Gaia, Chaos) and CentOS+SLURM (Iris)
  - Gaia: can collect GPU metrics
  - Iris: can collect energy metrics

Using Performance Reports

1. Load toolchain that you run your app. with, e.g.
   - module load toolchain/intel

2. Run your application through Perf. Reports
   - iris: perf-report srun ./$app
   - gaia/chaos: perf-report mpirun -hostfile $OAR_NODEFILE ./$app

3. Analysis by default in .html and .txt indicating also run config.
Debugging and profiling tools

Arm Perf. Reports - output (I)

Command: srun gmx_mpi mdrun -s bench_rnase_cubic.tpr -nsteps 10000
Resources: 1 node (28 physical, 28 logical cores per node)
Memory: 126 GiB per node
Tasks: 28 processes, OMP_NUM_THREADS was 0
Machine: iris-053
Start time: Sun Jun 11 2017 20:13:58 (UTC+02)
Total time: 19 seconds
Full path: /mnt/irisgpfs/apps/resif/data/production/v0.1-20170602/
default/software/bio/GROMACS/2016.3-intel-2017a-hybrid/bin

Summary: gmx_mpi is **Compute-bound** in this configuration

- **Compute**: 54.6%
- **MPI**: 45.4%
- **I/O**: 0.0%

Time spent running application code. High values are usually good. This is **average**, check the CPU performance section for advice.

Time spent in MPI calls. High values are usually bad. This is **average**, check the MPI breakdown for advice on reducing it.

Time spent in filesystem I/O. High values are usually bad. This is **negligible**, there's no need to investigate I/O performance.

This application run was **Compute-bound**. A breakdown of this time and advice for investigating further is in the **CPU** section below.

### CPU

- A breakdown of the 54.6% CPU time:
  - Single-core code: 5.5%
  - OpenMP regions: 94.5%
  - Scalar numeric ops: 5.2%
  - Vector numeric ops: 44.2%
  - Memory accesses: 50.6%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

### MPI

- A breakdown of the 45.4% MPI time:
  - Time in collective calls: 33.5%
  - Time in point-to-point calls: 66.5%
  - Effective process collective rate: 426 MB/s
  - Effective process point-to-point rate: 419 MB/s

Most of the time is spent in point-to-point calls with an average transfer rate. Using larger messages and overlapping communication and computation may increase the effective transfer rate.
Debugging and profiling tools

Arm Perf. Reports - output (II)

CPU
A breakdown of the 54.6% CPU time:

- Single-core code: 5.5%
- OpenMP regions: 94.5%
- Scalar numeric ops: 5.2%
- Vector numeric ops: 44.2%
- Memory accesses: 50.6%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

I/O
A breakdown of the 0.0% I/O time:

- Time in reads: 0.0%
- Time in writes: 0.0%
- Effective process read rate: 0.00 bytes/s
- Effective process write rate: 0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

MPI
A breakdown of the 45.4% MPI time:

- Time in collective calls: 33.5%
- Time in point-to-point calls: 66.5%
- Effective process collective rate: 426 MB/s
- Effective process point-to-point rate: 419 MB/s

Most of the time is spent in point-to-point calls with an average transfer rate. Using larger messages and overlapping communication and computation may increase the effective transfer rate.

OpenMP
A breakdown of the 94.5% time in OpenMP regions:

- Computation: 99.5%
- Synchronization: 0.5%
- Physical core utilization: 100.0%
- System load: 101.9%

OpenMP thread performance looks good. Check the CPU breakdown for advice on improving code efficiency.

Memory
Per-process memory usage may also affect scaling:

- Mean process memory usage: 76.6 MB
- Peak process memory usage: 86.6 MB
- Peak node memory usage: 11.0%

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.

Energy
A breakdown of how the 0.899 Wh was used:

- CPU: 100.0%
- System: not supported %
- Mean node power: not supported W
- Peak node power: not supported W

The whole system energy has been calculated using the CPU energy usage.

System power metrics: No Allinea IPMI Energy Agent config file found in (null). Did you start the Allinea IPMI Energy Agent?
Debugging and profiling tools

**Intel Advisor - highlights**

**Advisor features**

- Vectorization Optimization and Thread Prototyping
- Analyze vectorization opportunities
  - for code compiled either with Intel and GNU compilers
  - SIMD, AVX* (incl. AVX-512) instructions
- Multiple data collection possibilities
  - loop iteration statistics
  - data dependencies
  - memory access patterns
- Suitability report - predict max. speed-up
  - based on app. modeling

**Full details at** software.intel.com/en-us/intel-advisor-xe
Debugging and profiling tools

Intel Advisor - on ULHPC

Modules

- On iris/gaia/chaos: module load perf/Advisor

Using Intel Advisor

1. Load toolchain: module load toolchain/intel
2. Compile your code, e.g. mpiicc $code.c -o $app
3. Collect data e.g. on gaia:
   
   mpirun -n 1 -gtool "advixe-cl -collect survey \
   -project-dir ./advisortest:0" ./$app

4. Visualise results with advixe-gui $HOME/advisortest
Debugging and profiling tools

Intel Advisor - interface

Vectorization Advisor

Vectorization Advisor is a vectorization analysis toolset that lets you identify loops that will benefit most from vector parallelism, discover performance issues preventing effective vectorization and characterize your memory vs. vectorization bottlenecks with Advisor Roofline model automation.

Program metrics

- Elapsed Time: 247.46s
- Vector Instruction Set: SSE, SSE2
- Number of CPU Threads: 1

Loop metrics

- Total CPU time: 246.84s
- Time in vectorized loops: 176.70s
- Time in scalar code: 70.04s

Vectorization Gain/Efficiency

- Vectorized Loops Gain/Efficiency: 3.81x
- Program Approximate Gain: 1.58x

Top time-consuming loops

- Loop 
  - Start Time: 70.624s
  - Total Time: 70.624s

Collection details

- Platform information
  - MPI rank: 0
  - CPU Name: Intel(R) Xeon(R) CPU X5670 @ 2.93GHz
  - Frequency: 2.93 GHz
  - Logical CPU Count: 12
  - Operating System: Linux
  - Computer Name: gaia-100.gaia-cluster.uni.lu
Debugging and profiling tools

Scalasca & friends - highlights

Scalasca features

- Scalable performance analysis toolset
  - for large scale // applications on 100,000s of cores
- Support for C/C++/Fortran code with MPI, OpenMP, hybrid
- 3 stage workflow: instrument, measure, analyze
  - at compile time, run time and resp. postmortem
- Score-P for instrumentation + measurement, Cube for vis.
  - Score-P can also be used with Periscope, Vampir and Tau
- Facilities for measurement optimization to min. overhead
  - by selective recording, runtime filtering

Full details at http://www.scalasca.org/about/about.html
Debugging and profiling tools

Scalasca - on ULHPC

Modules

- On iris/gaia/chaos:

  module load perf/Scalasca perf/Score-P

Using Scalasca

1. Load toolchain: `module load toolchain/foss`
2. Compile your code, e.g. `scorep mpicc $code.c -o $app`
3. Collect data e.g. on gaia: `scan -s mpirun -n 12 ./$app`
4. Visualise results with square `scorep_$app_12_sum`
   - or generate text report: `square -s scorep_$app_12_sum`
   - ... and print it: `cat scorep_$app_12_sum/scorep.score`
Debugging and profiling tools

Scalasca visualisation with Cube-P
Summary

1. Introduction

2. Debugging and profiling tools

3. Conclusion
Conclusion

Now it’s up to you

Easy right?
Conclusion

Now it’s up to you

Easy right?

Well not exactly.
Now it’s up to you

Easy right?

Well not exactly. Debugging always takes effort and real applications are never trivial.
Now it’s up to you

Easy right?

Well not exactly. Debugging always takes effort and real applications are never trivial.

But we do guarantee it’ll be /easier/ with these tools.
Conclusion and Practical Session start

We’ve discussed
- A couple of small utilities that can be of big help
- HPC oriented tools available for you on UL HPC

And now..

Short DEMO time!
Conclusion

Conclusion and Practical Session start

We’ve discussed

- A couple of small utilities that can be of big help
- HPC oriented tools available for you on UL HPC

And now..

Short DEMO time!

Your Turn!
Hands-on start

We will first start with running HPCG (unmodified) as per:


... your tasks:

1. perform a timed first run using unmodified HPCG v3.0 (MPI only)
   - use /usr/bin/time -v to get details
   - single node, use ≥ 80 80 80 for input params (hpcg.dat)

2. run HPCG (timed) through Allinea Perf. Report
   - use perf-report (bonus points if using iris to get energy metrics)

3. instrument and measure HPCG execution with Scalasca

Remember: pre-existing reservations for the workshop:

- ‘hpschool’: Iris cluster resv. (use 
  --reservationname=hpcschoolday1)
- 4354151: Gaia cluster regular nodes (use -t inner=4354151)
High Performance Computing @ uni.lu

Prof. Pascal Bouvry
Dr. Sebastien Varrette
Valentin Plugaru
Sarah Peter
Hyacinthe Cartiaux
Clement Parisot

University of Luxembourg, Belval Campus
Maison du Nombre, 4th floor
2, avenue de l’Université
L-4365 Esch-sur-Alzette
mail: hpc@uni.lu

1 Introduction
2 Debugging and profiling tools
3 Conclusion