UL HPC School 2017
PS4b: Debugging, profiling and performance analysis

UL High Performance Computing (HPC) Team
V. Plugaru

University of Luxembourg (UL), Luxembourg
http://hpc.uni.lu
Latest versions available on Github:

UL HPC tutorials:  
https://github.com/ULHPC/tutorials

UL HPC School:  
http://hpc.uni.lu/hpc-school/

PS4b tutorial sources:  
https://github.com/ULHPC/tutorials/tree/devel/advanced/debugging
Summary

1. Introduction

2. Debugging and profiling tools

3. Conclusion
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 problems.

During the hands-on session you will:

- see what happens when an application runs out of memory and how to discover how much memory it actually requires.
- use debugging tools to understand why your code is crashing.
- use profiling tools to understand the (slow) performance of your code - and how to improve it.

Knowing what to do when you experience a problem is half the battle.
Debugging and profiling tools

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.
Tools at your disposal (II)

**HPC specific tools**

- Allinea DDT (part of Allinea Forge)
  - Visual debugger for C, C++, and Fortran threaded and \(/\) code
- Allinea MAP (part of Allinea Forge)
  - Visual C/C++/Fortran profiler for high performance Linux code
- Allinea Performance Reports
  - Application characterization tool
Tools at your disposal (II)

**HPC specific tools**

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

**Allinea tools are licensed**

Make sure enough tokens available to profile/debug your code in the requested configuration (#cores)!
- license check will be integrated in SLURM
- ... so your jobs will be able to wait for it to be available
Debugging and profiling tools

Allinea DDT - highlights

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 allinea.com/products/ddt/features
Debugging and profiling tools

Allinea DDT - on ULHPC

Modules

- On iris: module load tools/AllineaForge
- On gaia/chaos: module load Allinea/Forge
  ➔ we’ll synchronize the software set to match iris soon

Debugging with DDT

1. Load toolchain, e.g.
   ➔ iris: module load toolchain/intel
   ➔ gaia/chaos: module load toolchain/ictce

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

3. Run your code through DDT
   ➔ iris: ddt srun ./app
   ➔ gaia/chaos: ddt mpirun -hostfile $OAR_NODEFILE ./app
Debugging and profiling tools

Allinea DDT - interface
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 [allinea.com/products/map/features](http://allinea.com/products/map/features)
Debugging and profiling tools

Allinea MAP - on ULHPC

Modules

- On iris: module load tools/AllineaForge
- On gaia/chaos: module load Allinea/Forge

Profiling with MAP

1. Load toolchain that built your app., e.g.
   - iris: module load toolchain/intel
   - gaia/chaos: module load toolchain/ictce

2. Run your code through MAP
   - iris: map srun ./$app
   - gaia/chaos: map mpirun -hostfile $OAR_NODEFILE ./$app
Debugging and profiling tools

Allinea MAP - interface
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 allinea.com/products/allinea-performance-reports
Debugging and profiling tools

Allinea Perf. Reports - on ULHPC

**Modules**

- On iris: module load tools/AllineaReports
- On gaia/chaos: module load Allinea/Reports

**Using Performance Reports**

1. Load toolchain that you run your app. with, e.g.
   - iris: module load toolchain/intel
   - gaia/chaos: module load toolchain/ictce

2. Run your application through Perf. Reports
   - iris: perf-report srun ./${app}
   - gaia/chaos: perf-report mpirun -hostfile $OAR_NODEFILE ./${app}
Debugging and profiling tools

Allinea Perf. Reports - output (I)

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

Allinea 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.

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.

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!

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 76.8 MiB
- Peak process memory usage 86.6 MiB
- 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?
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.
Conclusion

Now it’s up to you

Easy right?

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

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.
We’ve discussed

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

And now..

Short DEMO time!

Your Turn!
Thank you for your attention...

Questions?

The UL High Performance Computing (HPC) Team
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