Linaro DDT - A Parallel Debugger at Scale

Rudy Shand
Linaro Ltd

HPC development Solutions from Linaro

Build reliable and optimized code on multiple Server and HPC architectures

Linaro Forge combines

Linaro DDT

Market leading, simple to use HPC debugger for C/C++, Fortran and Python applications.

Linaro MAP

Effortless performance analysis for experts and novices alike.

Linaro Performance Reports

At a glance, single-page, application performance summary.

Performance Engineering for any architecture, at any scale

gdb under the hood

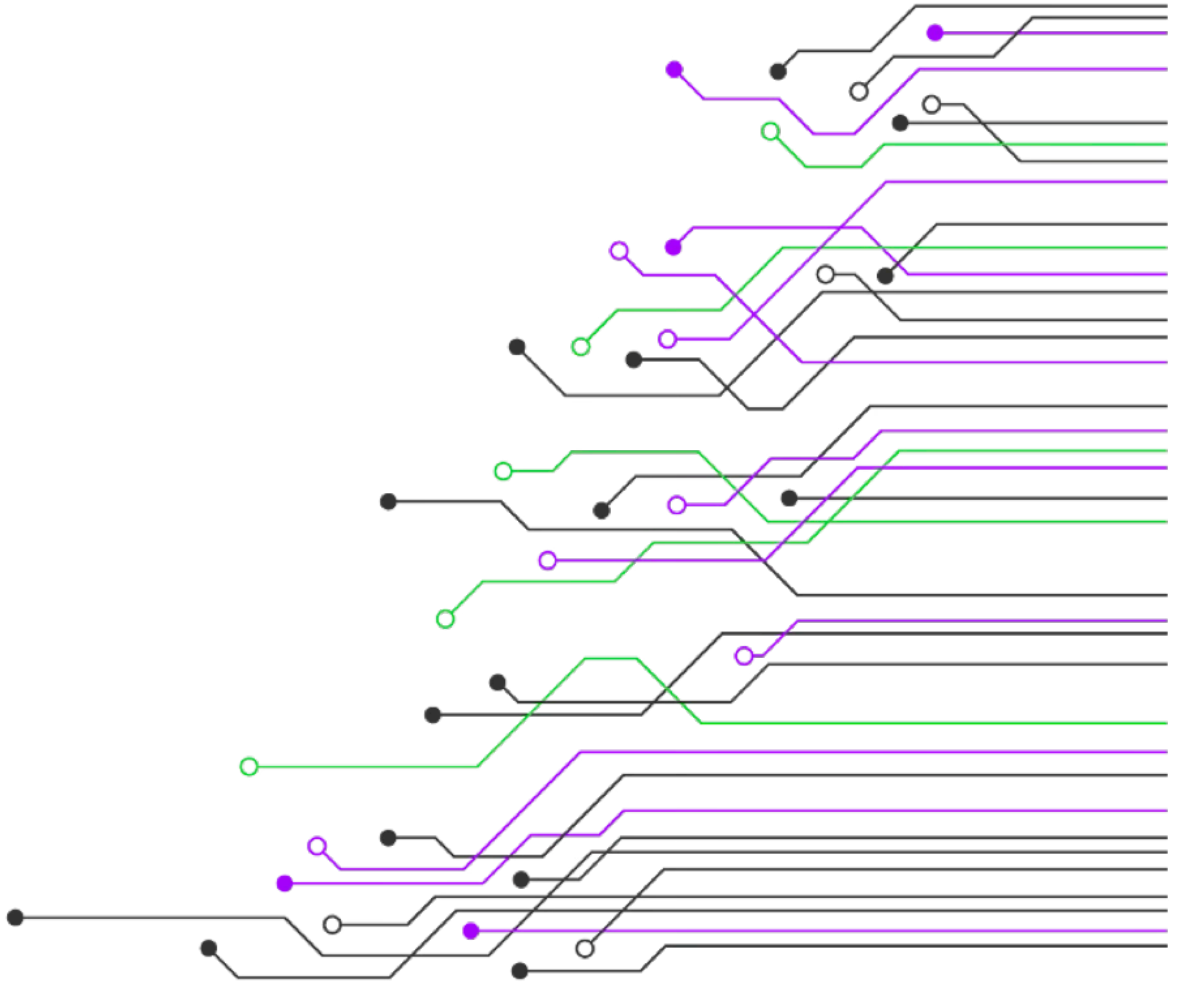
Leveraging the gdb community

GDB is the underlying technology

- Linaro upstreams patches to the gdb community
- Forge team raises and fixes gdb bugs

Nimble at supporting new technologies

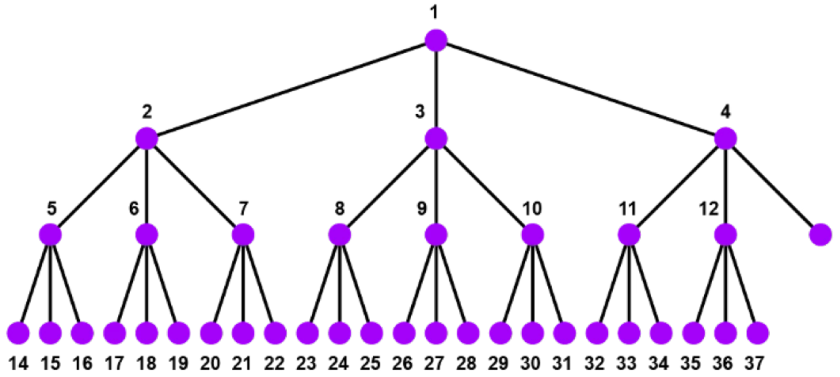
- Rely on hardware vendors to add gdb support
- Rely on software consortiums to add gdb support
- Helps us to stay current with technologies
- Provides a state of the art debugger

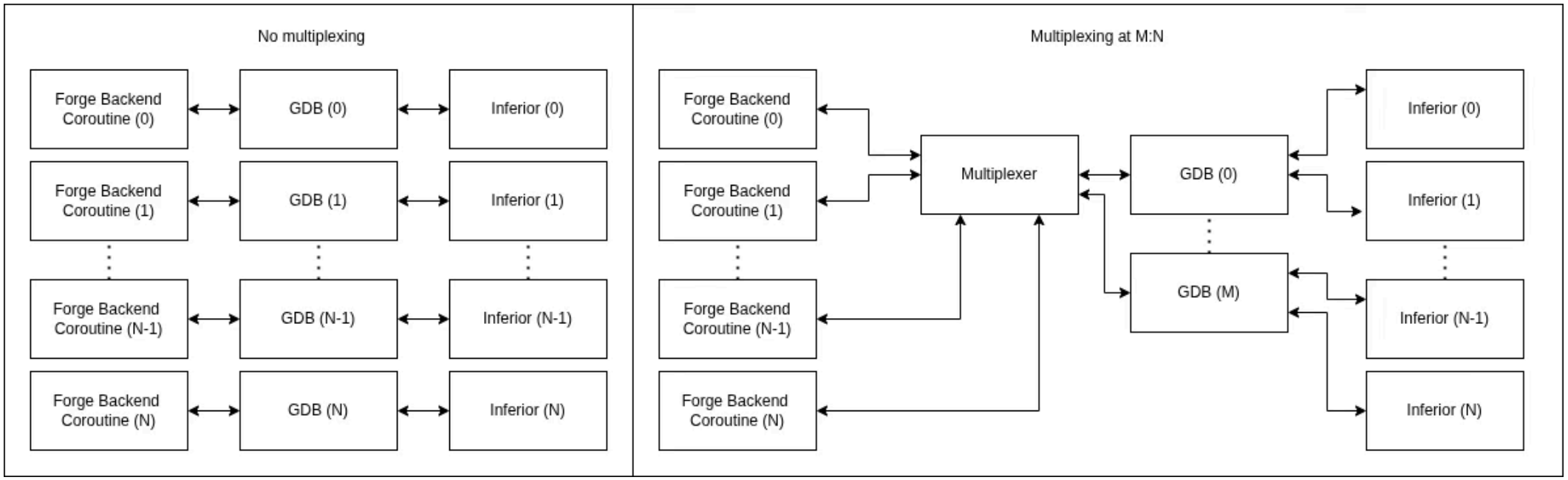


Scaling gdb

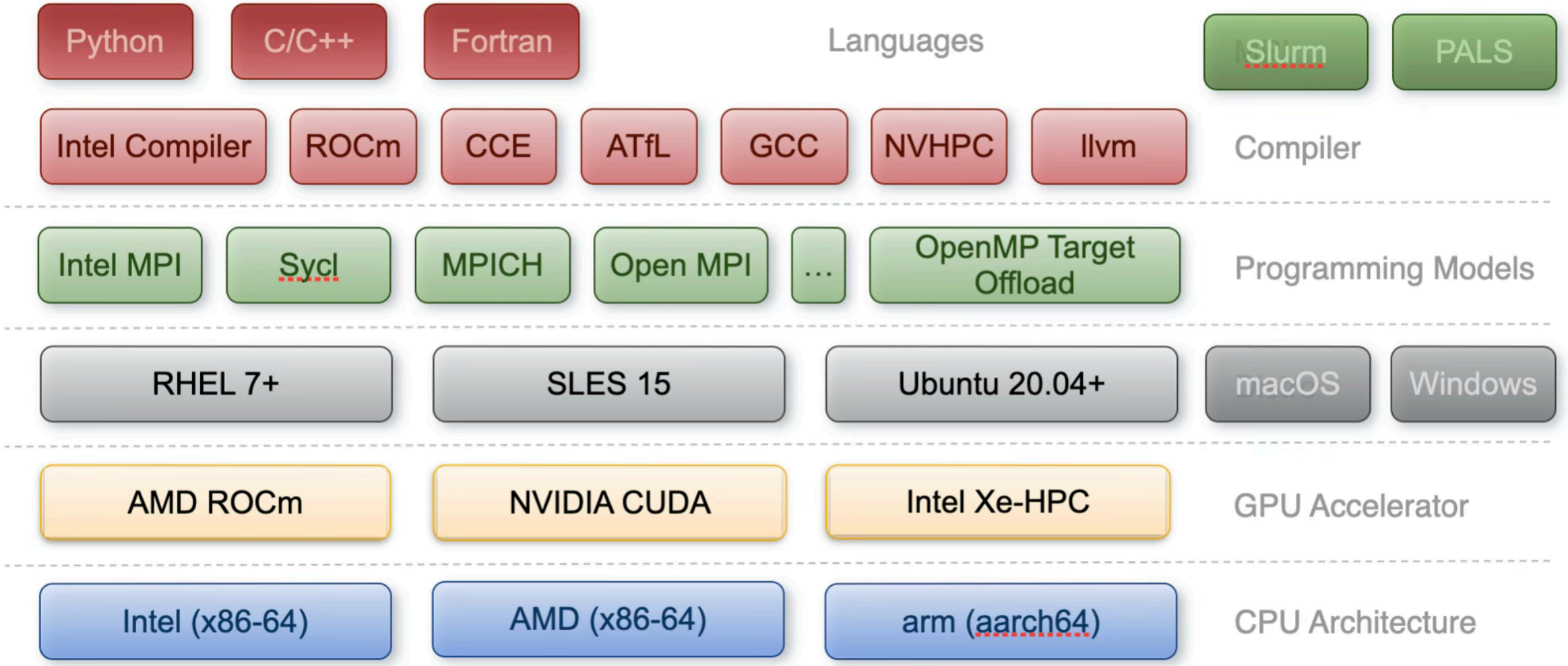
Tree network topology

- Tree server design is how Forge is able to scale
- Send bulk commands and merge responses
- Aggregate the data instead of broadcasting thousands of responses
- Multiplex for tooling memory reduction (up to 80% reduction)





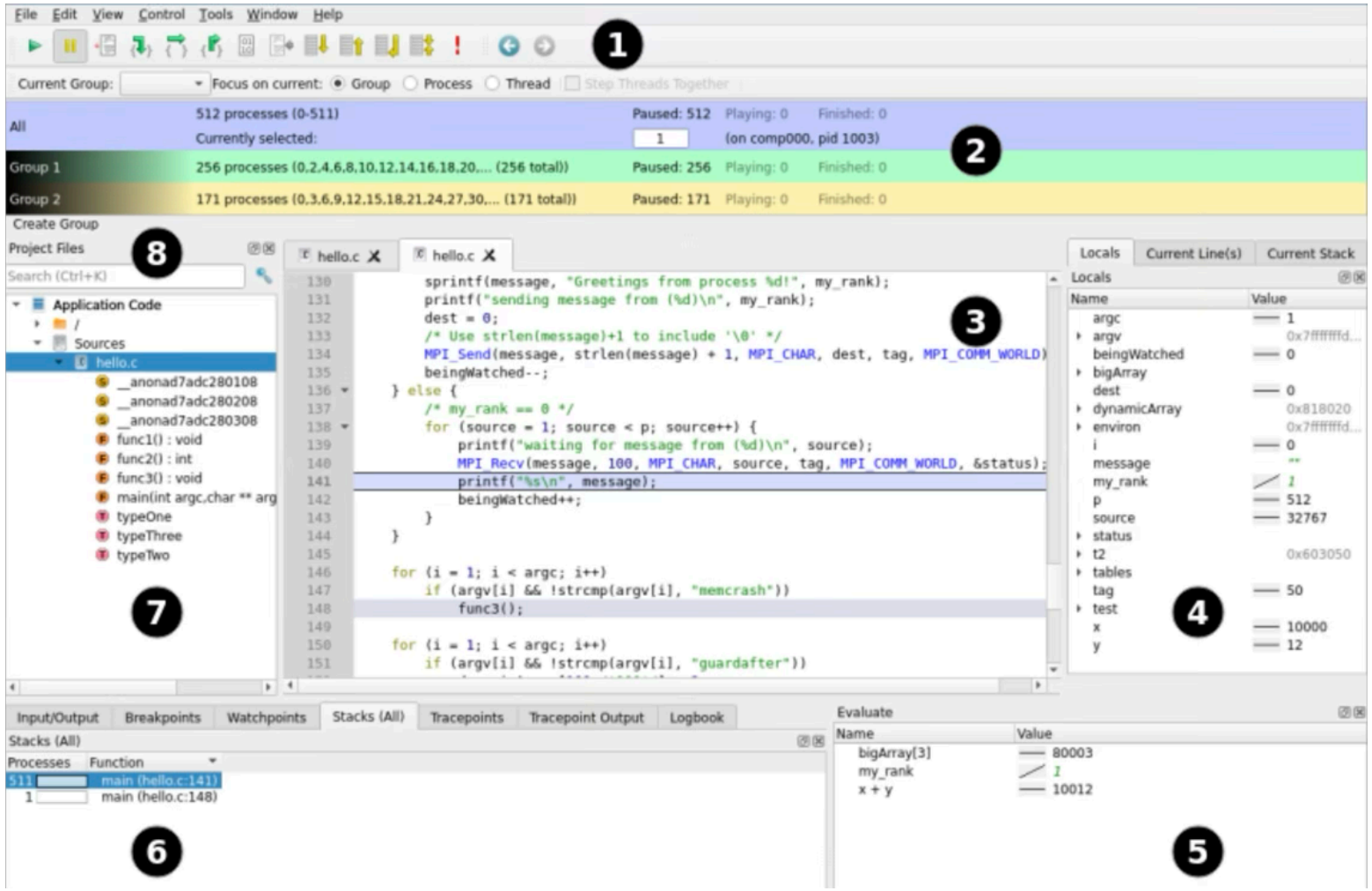
DDT Supported platforms



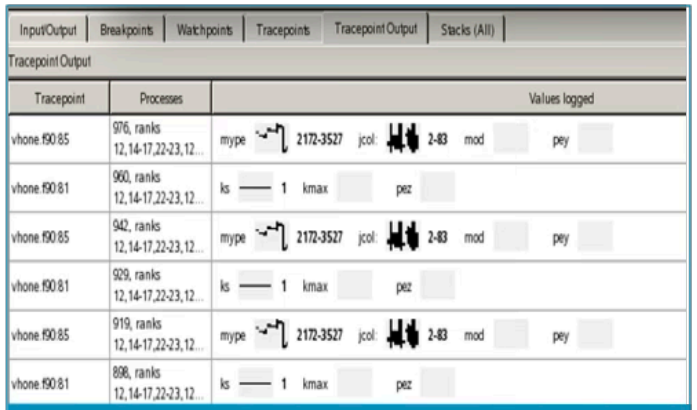
DDT User Interface

- 1 Process controls
- 2 Process groups
- 3 Source Code view
- 4 Variables
- 5 Evaluate window
- 6 Parallel Stack
- 7 Project files
- 8 Find a file or function

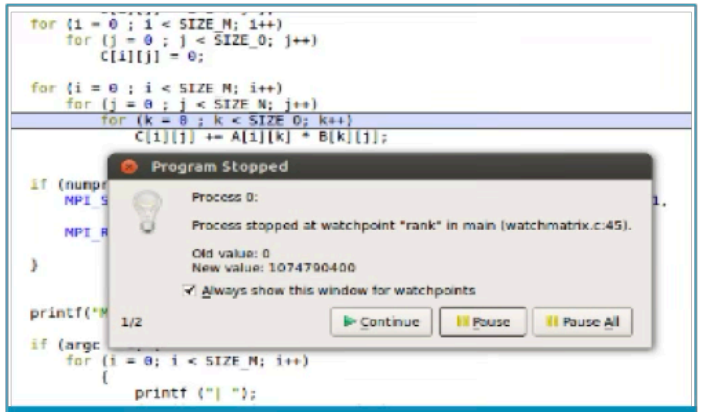
Compile with -g -O0



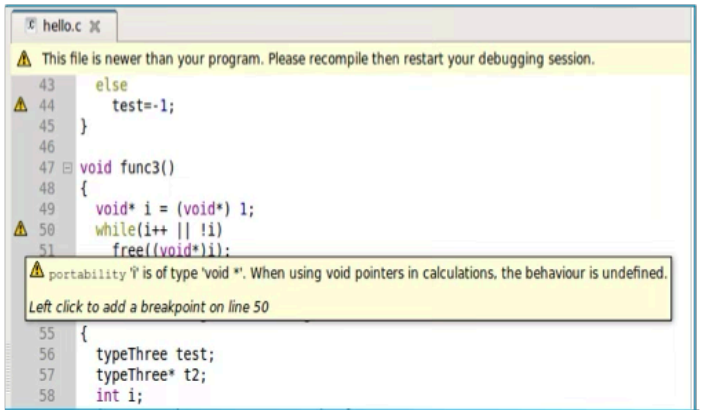
DDT Highlights



The scalable print alternative



Stop on variable change

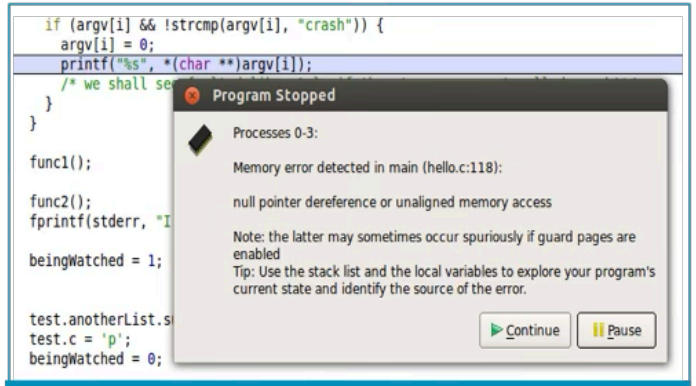


Static analysis warnings on code errors

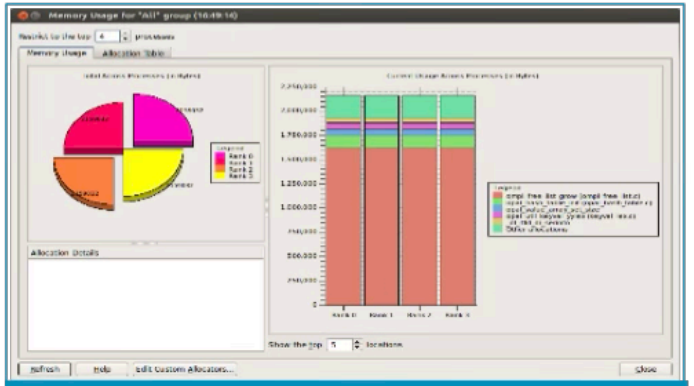
Memory Debugging

```
# Enable reading of debug environment variables:
host:~/demo> export NEOReadDebugKeys=1

# Disable RTLD_DEEPBIND, so that malloc/free calls bind to
# our memory debug library, instead of glibc:
host:~/demo> export NEODisableDeepBind=1
```

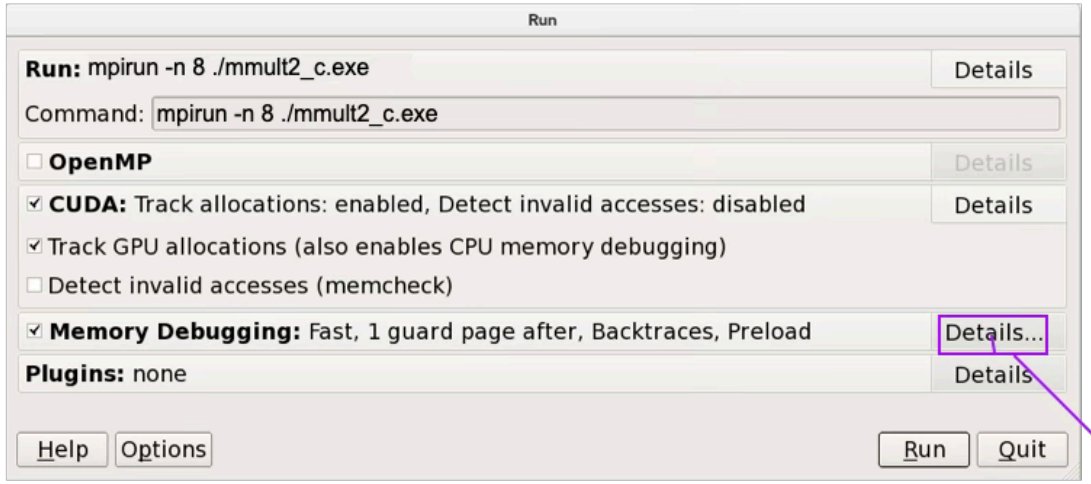


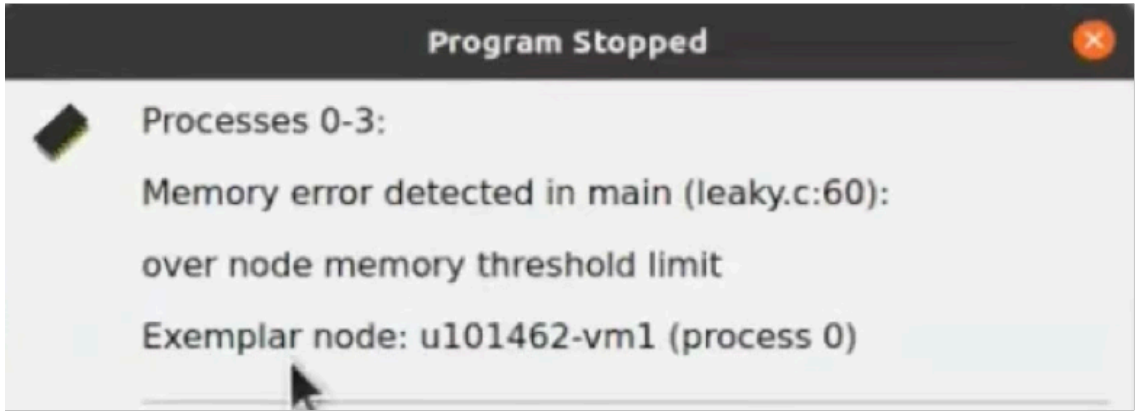
Detect read/write beyond array bounds



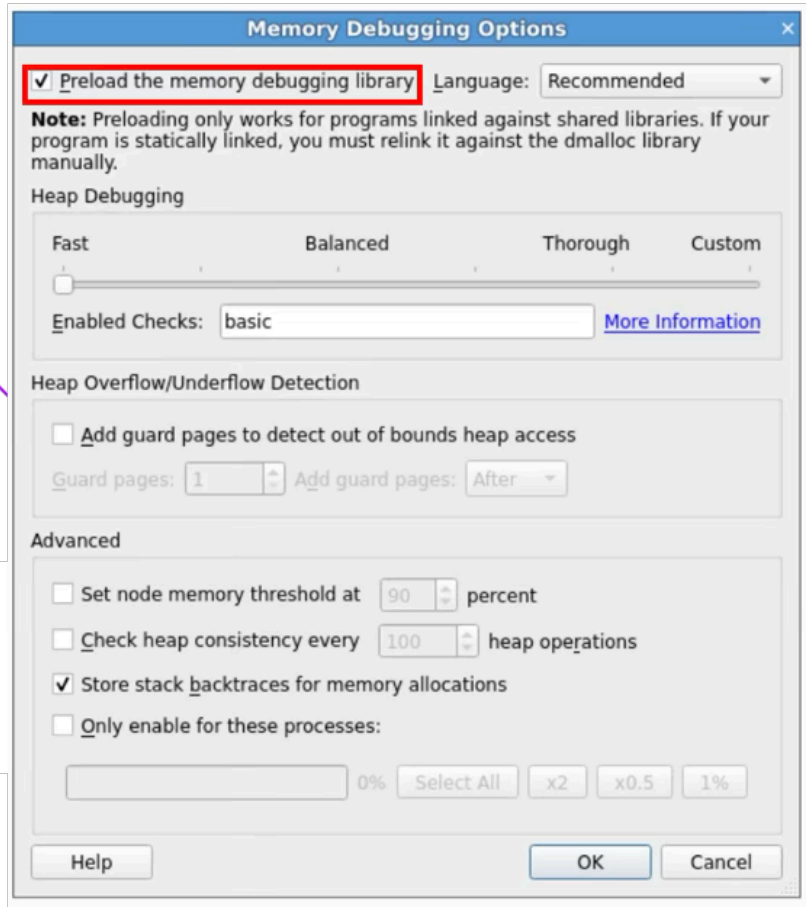
Detect stale memory allocations

Memory Debugging





When manual linking is used, untick “Preload” box



MDA Viewer

What does your data look like at runtime?

View arrays

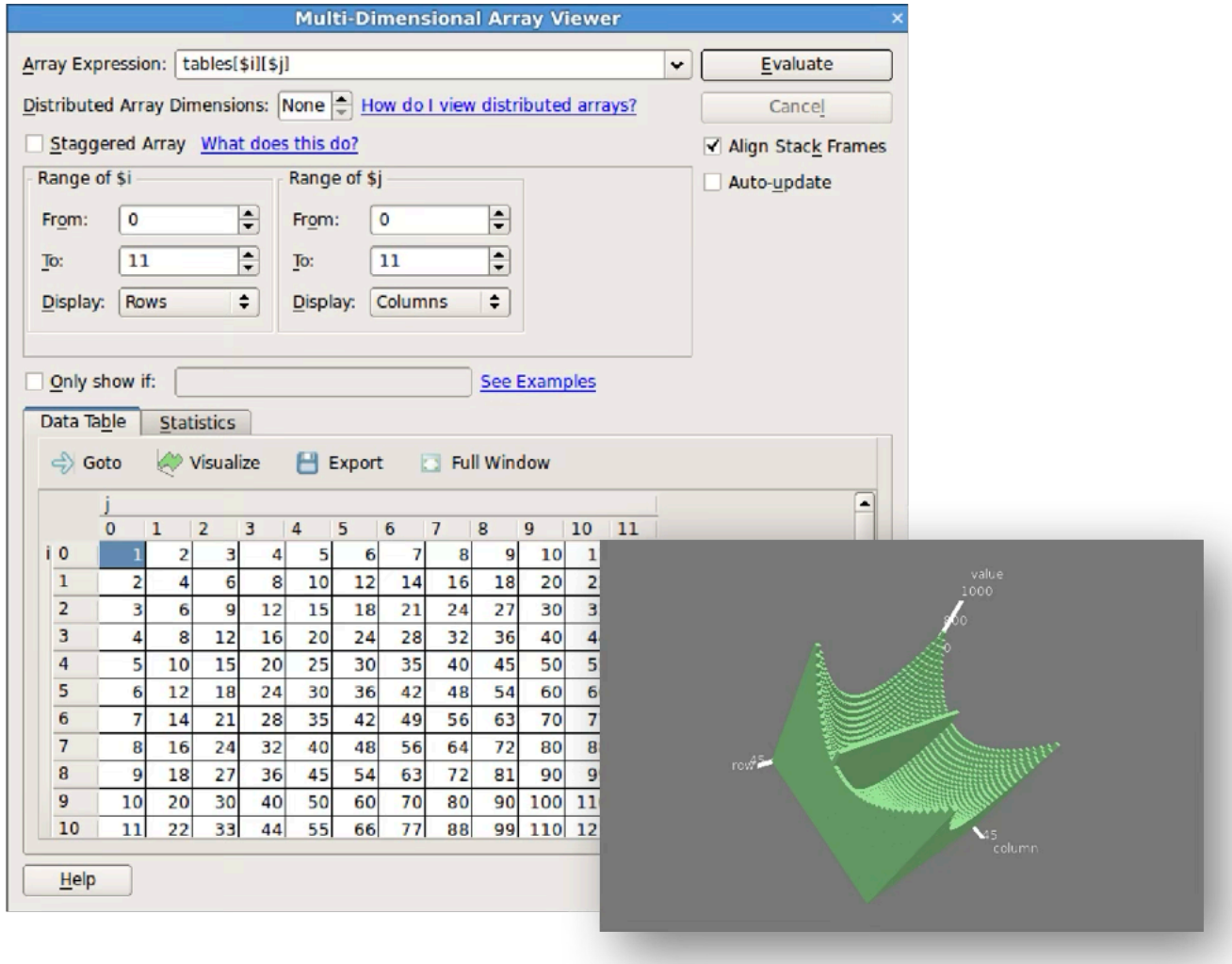
- On a single process
- Or distributed on many ranks

Use metavariables to browse the array

- Example: $i and $j
- Metavariables are unrelated to the variables in your program
- The bounds to view can be specified
- Visualise draws a 3D representation of the array

Data can also be filtered

- "Only show if": $value>0 for example $value being a specific element of the array



GPU Debugging Terminology

	NVIDIA GPU	AMD GPU	Intel GPU
Name	CUDA	ROCm	Xe
Language	CUDA C/C++ and Fortran	HIP	SYCL
Execution Unit	Warp	Wavefront	Sub-group
EU Size	32 Threads	64 Threads	8, 16 or 32 Threads
EU GDB	…	GDB Thread	GDB Thread
EU Thread	…	Lane	Lane (work-item)
Forge GPU Thread	Lane	Lane	Lane

Debugging Intel Xe GPUs

Using Linaro DDT

Debug code simultaneously on the GPU and the CPU

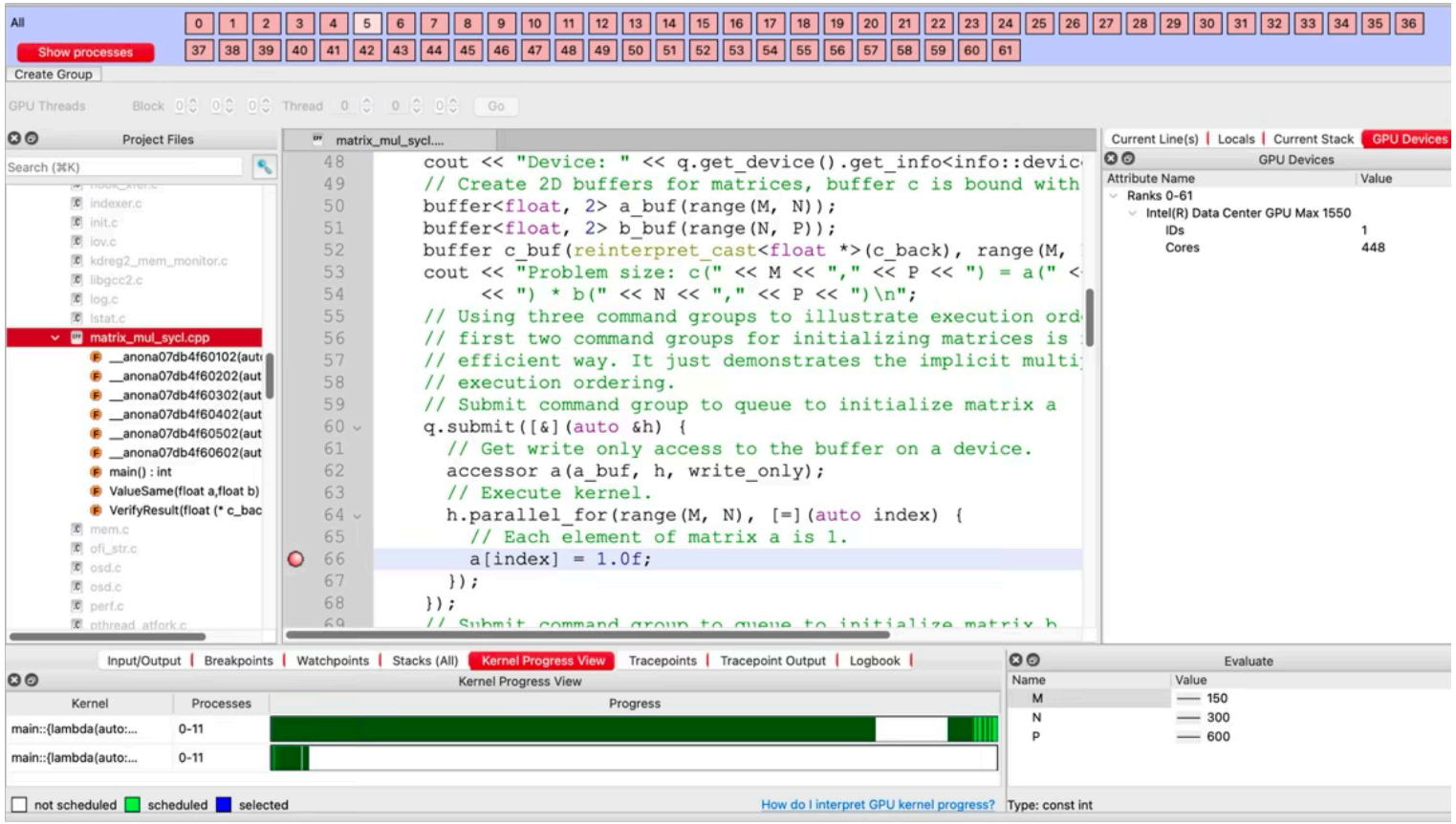
Controlling the GPU execution:

- All active threads in a Sub-group will execute in lockstep. Therefore, DDT will step16 threads at a time.
- Play/Continue runs all GPU threads
- Pause will pause a running kernel

Key (additional) GPU features:

- Kernel Progress View
- GPU thread in parallel stack view
- GPU Thread Selector
- GPU Device Pane

Kernels must be compiled with the -g and -O0 flags



Debugging Nvidia GPUs

Using Linaro DDT

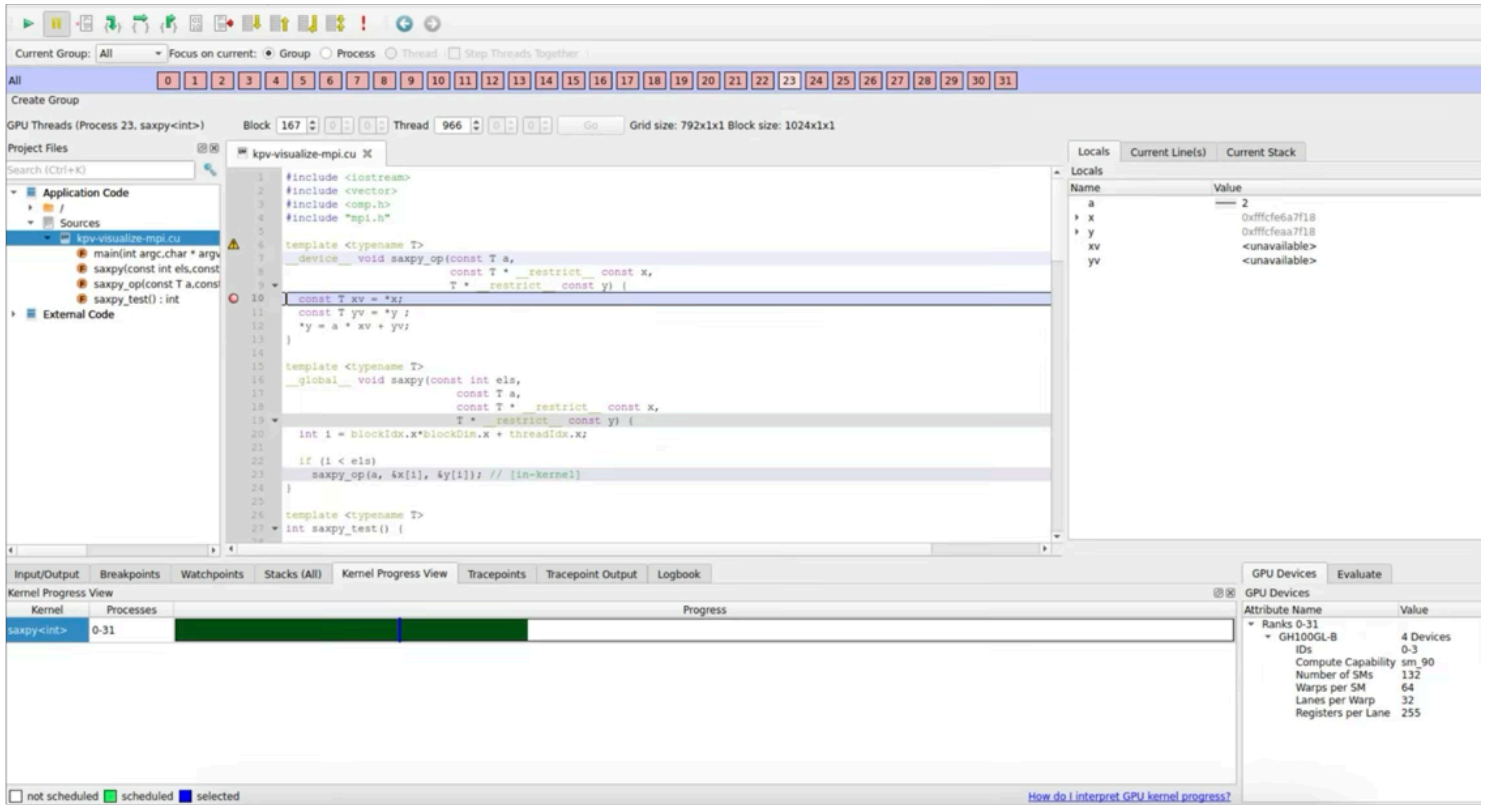
Controlling the GPU execution:

- All active threads in a warp will execute in lockstep Therefore, DDT will step 32 threads at a time.
- Play/Continue runs all GPU threads
- Pause will pause a running kernel

Key (additional) GPU features:

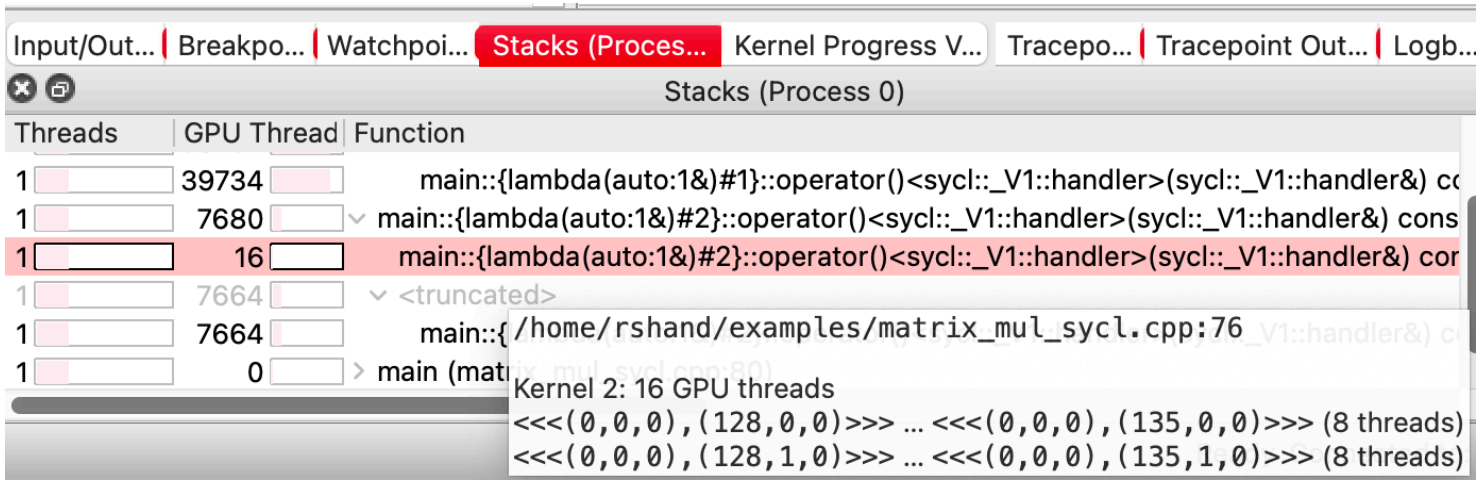
- Kernel Progress View
- GPU thread in parallel stack view
- GPU Thread Selector
- GPU Device Pane

For NVIDIA's nvcc compiler, kernels must be compiled with the -g and -G flags



Parallel Stack view

Display location and number of threads



Click stack item

- Select GPU Thread
- Update variable display
- Move Source Code Viewer

Tooltip displays

- GPU Thread Ranges
- Size of each range

Case Study: Vienne Supercomputing Center (VSC)

Developing a GPU version of an already existing serial algorithm (Marching Tetrahedra). https://github.com/HliasGit/tetra

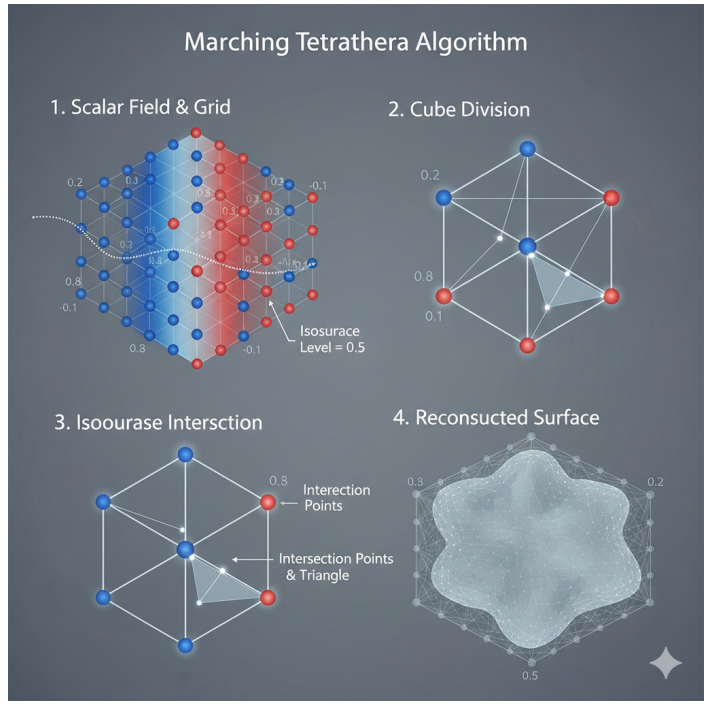
- The aim of the algorithm is to extract the approximation of an iso-surface from a scalar field.
- The scalar field (whose dimensions are DX,DY,DZ) is represented as a 3D tensor (of floats)
- Linearise (flatten) the scalar field to use it in a simpler way on GPU

Used three kernels to develop the GPU version:

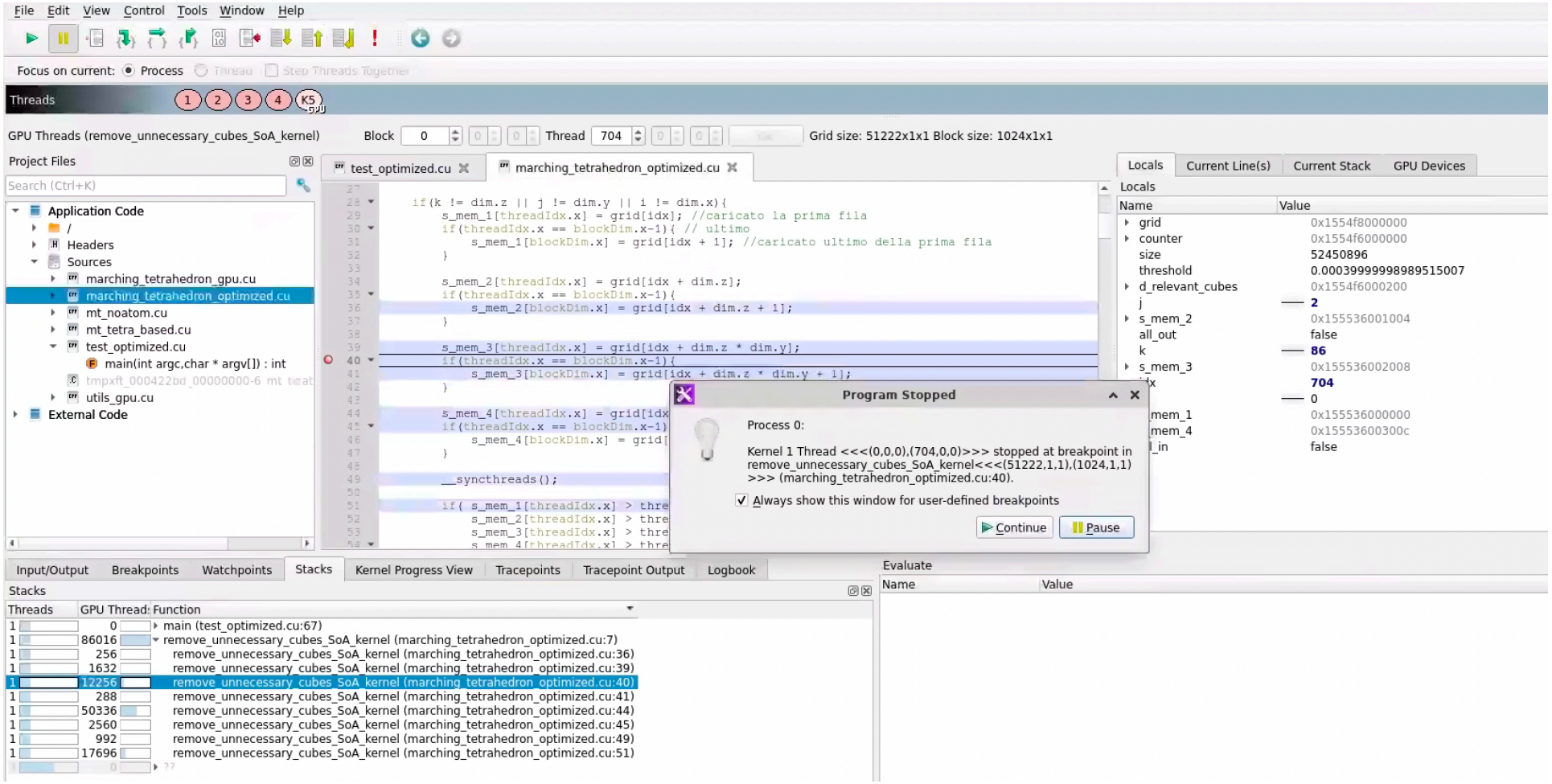
- The first one is a preprocessing step to reduce the computational complexity.
- Second and the third ones do the needed computation for the algorithm to work.

For each point in the scalar field, it needs to form a "cube" by accessing its 8 neighbours. Due to the Z-major linearization, only the neighbor in the Z-direction (index + 1) is adjacent in memory. Accessing neighbors in the X-direction (e.g., index + DZ*DY) or Y-direction (e.g., index + DZ) requires calculated memory offsets.

This approach led to "CUDA illegal memory access" errors. The problem was a failure to correctly handle boundary conditions. When a thread processed a point on the "edge" of the 3D domain (like at x = DX-1 or y = DY-1), these offset calculations were trying to access memory outside the allocated array. This boundary-checking logic was not intuitive, used Linaro DDT debugger to pinpoint these out-of-bounds access.



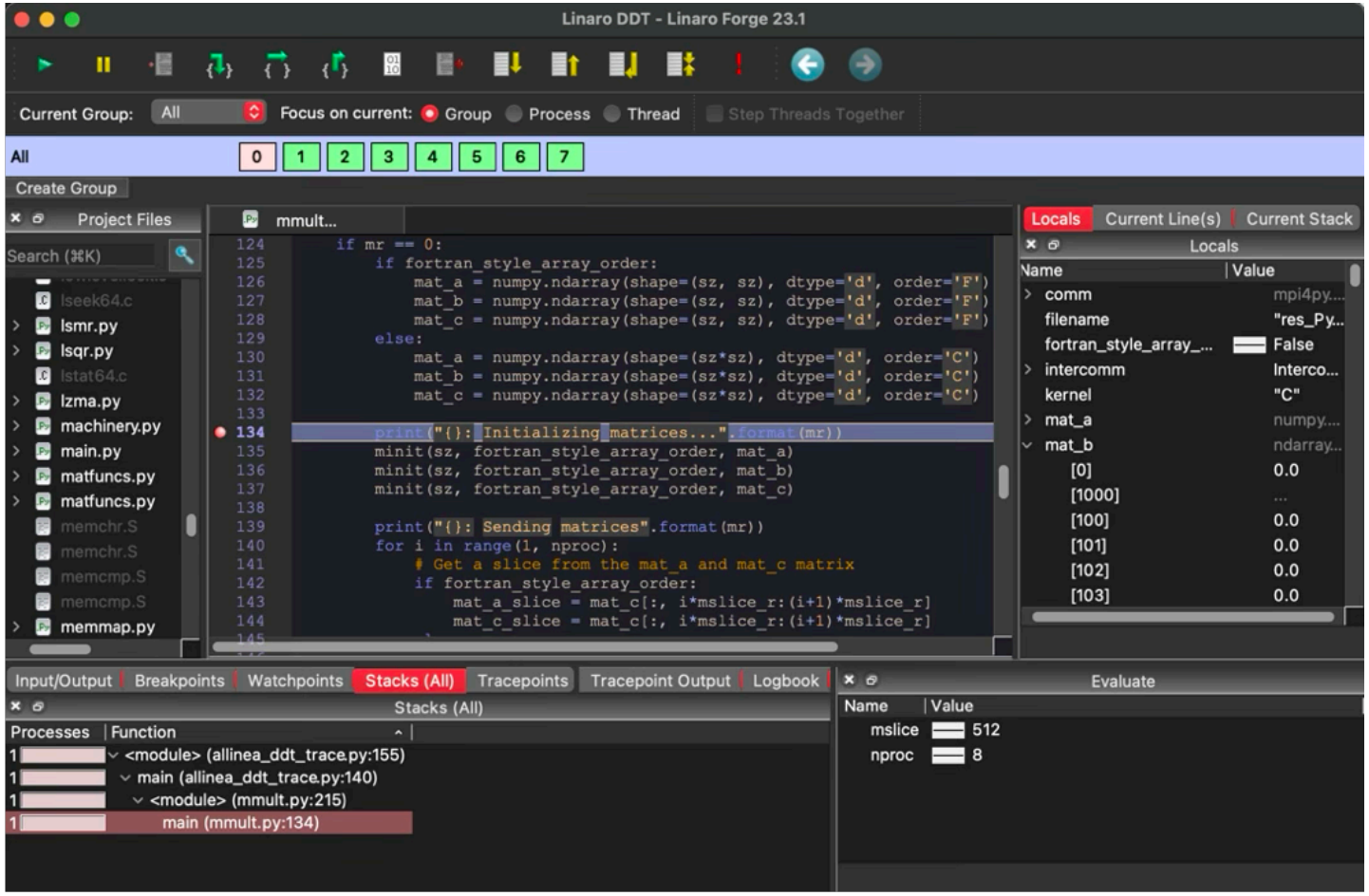
Spotting illegal memory access with Linaro DDT



Python Debugging

- Debug Features
  - Sparklines for Python variables
  - Tracepoints
  - MDA viewer
  - Mixed language support
- Improved Evaluations:
  - Matrix objects
  - Array objects
  - Pandas DataFrame
  - Series objects
- Python Specific:
  - Stop on uncaught Python exception
  - Show F-string variables in “Current Line” display
  - Mpi4py, NumPy, SciPy

ddt --connect mpirun -n 8 python3
%allinea_python_debug% ./mmult.py



DDT offline mode

Run the application under DDT and halt or report when a failure occurs

You can run the debugger in non-interactive mode

- For long-running jobs / debugging at very high scale
- For automated testing, continuous integration…

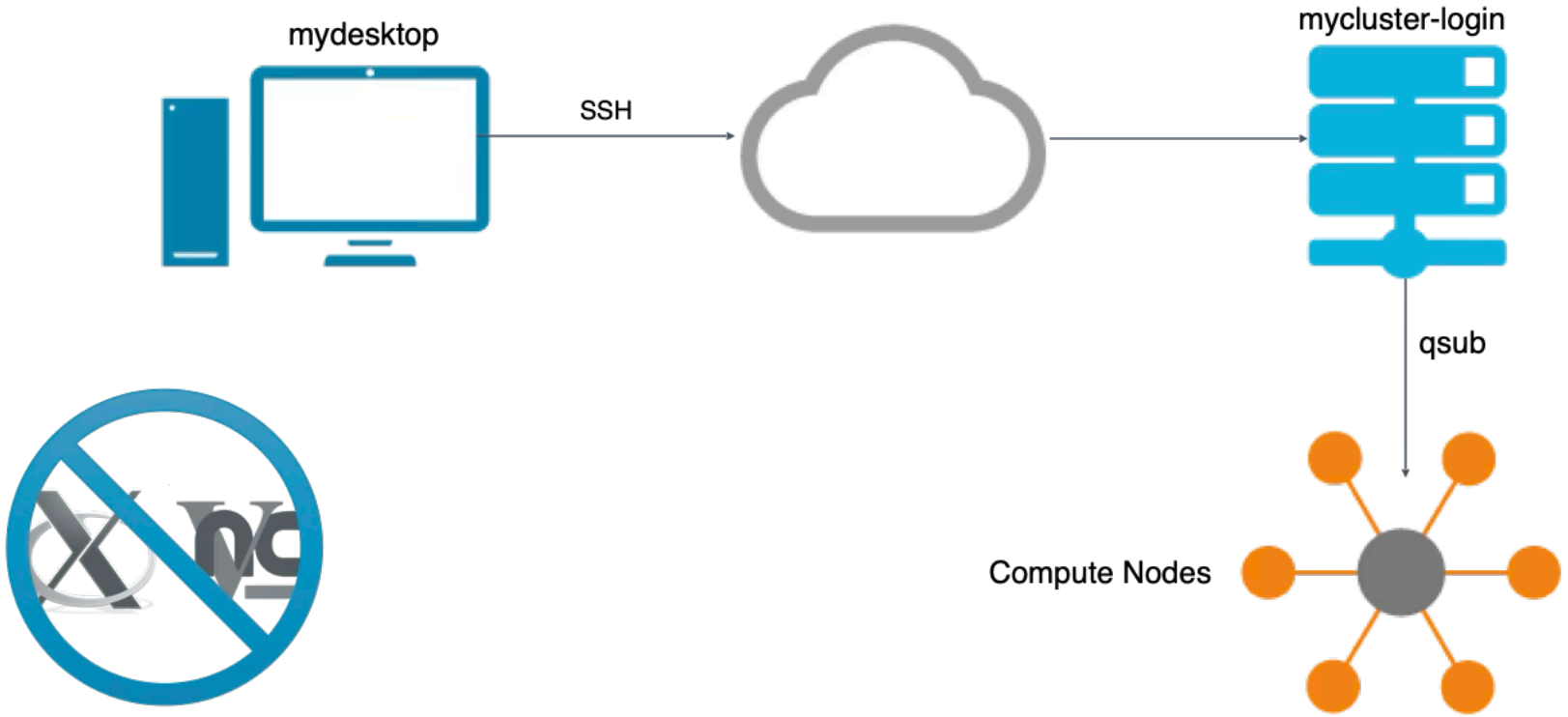
To do so, use following arguments:

- $ ddt **--offline --output=report.html** mpirun ./jacobi_omp_mpi_gnu.exe
  - **--offline** enable non-interactive debugging
  - **--output** specifies the name and output of the non-interactive debugging session (HTML or Txt)
  - Add **--mem-debug** to enable memory debugging **and memory leak detection**

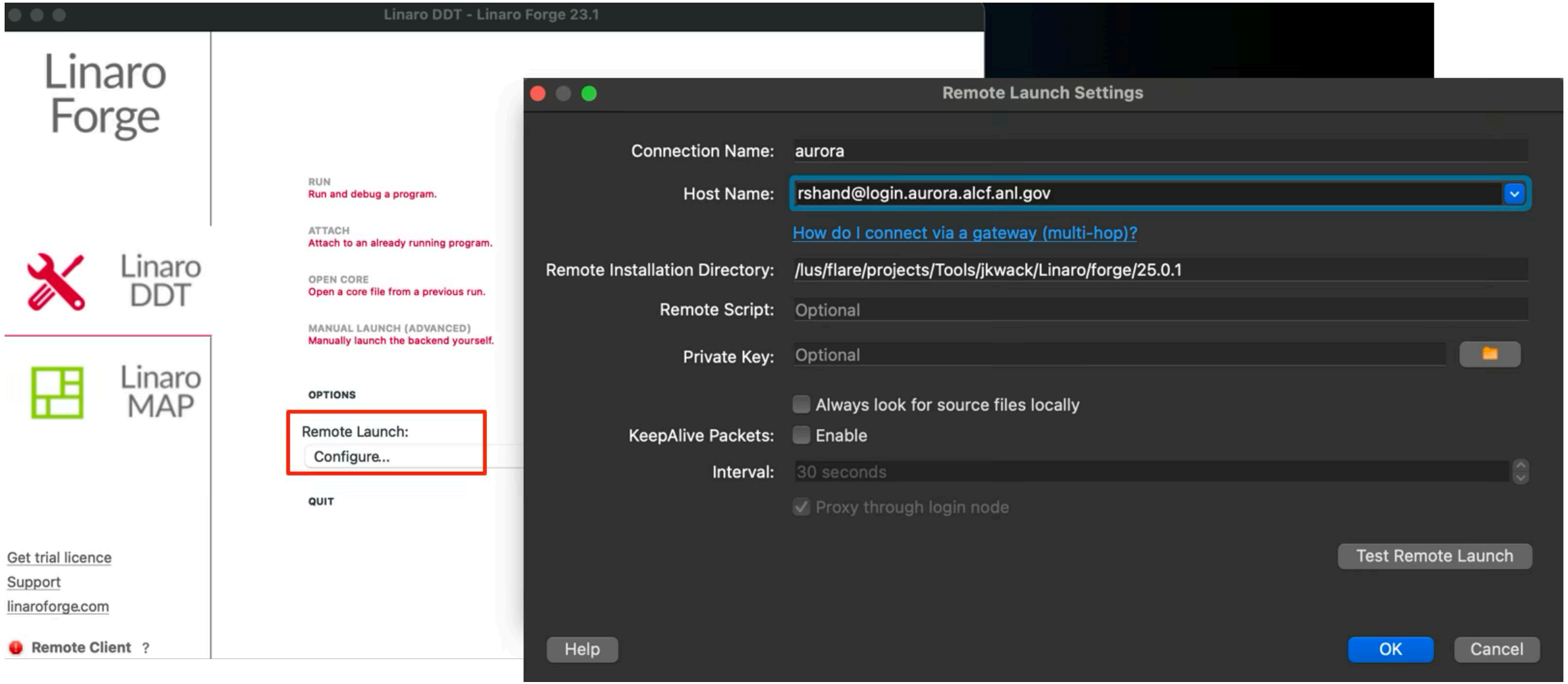
```
ddt --offline -o jacobi_omp_mpi_gnu_debug.txt \
                                 --trace-at _jacobi.F90:83,residual \
                                 mpirun ./jacobi_omp_mpi_gnu.exe
```

The Forge GUI and where to run it

Forge provides a powerful GUIs that can be run in a variety of configurations



Remote connection to Aurora



Cheat sheet

Training material

1. *Getting the examples*
   cp /flare/ATPESC2026/EXAMPLES/linaro-ddt/linaro-forge-training.tar.gz
   tar -xf linaro-forge-training.tar.gz

2. *Set the path to the forge training folder*
   export FORGE_TRAINING=<path_to_training_folder>

Forge Client (On local machine)

Install Forge client *https://www.linaroforge.com/downloadForge*

Running with a batch script

qsub $FORGE_TRAINING/submit.sh

Interactive Session

qsub -I -l select=1 -l walltime=01:00:00 -l filesystems=home:flare -A ATPESC2026 -q ATPESC

Forge commands

ddt --connect	*# Reverse connect*
ddt --offline	*# Run DDT without GUI*
map --profile	*# Profile without GUI*
perf-report	*# Generate Performance Report*

Guides

DDT userguide
Debugging on Aurora
Hands-on

Debugging Intel Xe GPUs on Aurora

cd $FORGE_TRAINING/correctness/gpu-intel-mmult

mpicxx --intel -fsycl -I$ONEAPI_ROOT/dev-utilities/latest/include -g -O0 matrix_mul_sycl.cpp -o matrixmul

qsub -I -l select=1 -l walltime=01:00:00 -l filesystems=home:flare -A ATPESC2026 -q ATPESC

https://docs.alcf.anl.gov/aurora/debugging/ddt-aurora/#invoking-the-ddt-server-from-aurora

ddt --np=24 --connect --mpi=generic --mpiargs="--ppn 12 --envall" ./matrixmul

ARGONNE TRAINING PROGRAM ON EXTREME-SCALE COMPUTING

Produced by Argonne National Laboratory, a U.S. Department of Energy Laboratory managed by UChicagoArgonne, LLC under contract DE-AC02-06CH11357.

Special thanks to the National Energy Research Scientific Computing Center (NERSC) and Oak Ridge Leadership Computing Facility (OLCF) for the use of their resources during the training event.

The U.S. Government retains for itself and others acting on its behalf a nonexclusive, royalty-free license in this video, with the rights to reproduce, to prepare derivative works, and to display publicly.