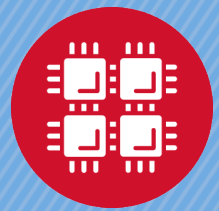
# Performance Tuning Workshop

Samuel Khuvis

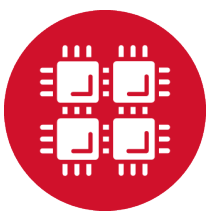
Scientific Applications Engineer, OSC

October 29, 2019



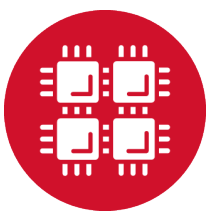
# Workshop Set up

- Workshop – set up account at `my.osc.edu`
  - If you already have an OSC account, sign in to `my.osc.edu`
  - Go to Project
  - Project access request
  - PROJECT CODE = PZS0724
- Slides are the workshop website:
  `https://www.osc.edu/~skhuvis/opt19_fall`

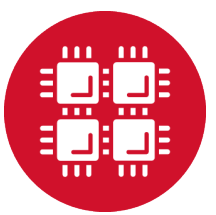


# Outline

- Introduction
- Debugging
- Hardware overview
- Performance measurement and analysis
- Help from the compiler
- Code tuning/optimization
- Parallel computing

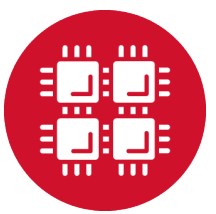


# Introduction



# Workshop Philosophy

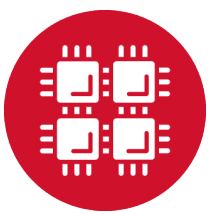
- Aim for "reasonably good" performance
- Discuss performance tuning techniques common to most HPC architectures
  - Compiler options
  - Code modification
- Focus on serial performance
  - Reduce time spent accessing memory
- Parallel processing
  - Multithreading
  - MPI



# Hands-on Code

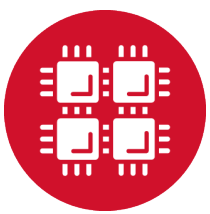
During this workshop, we will be using a code based on the HPCCG miniapp from Mantevo.

- Performs Conjugate Gradient (CG) method on a 3D chimney domain.
- CG is an iterative algorithm to numerically approximate the solution to a system of linear equations.
- Run code with `mpiexec -np <numprocs> ./test_HPCCG nx ny nz`, where `nx`, `ny`, and `nz` are the number of nodes in the x, y, and z dimension on each processor.
- Download with `git clone git@code.osu.edu:khuvis.1/performance2019_handson.git`
- Make sure that the following modules are loaded: `intel/18.0.3 mvapich2/2.3`



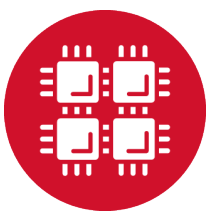
# More important than Performance!

- Correctness of results
- Code readability/maintainability
- Portability - future systems
- Time to solution vs execution time

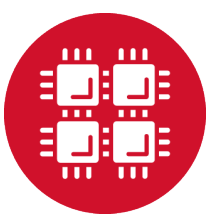


# Factors Affecting Performance

- Effective use of processor features
  - High degree of internal concurrency in a single core
- Memory access pattern
  - Memory access is slow compared to computation
- File I/O
  - Use an appropriate file system
- Scalable algorithms
- Compiler optimizations
  - Modern compilers are amazing!
- Explicit parallelism

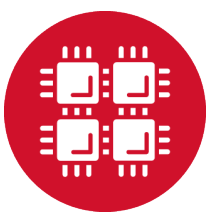


# Debugging



# What can a debugger do for you?

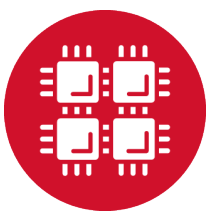
- Debuggers let you
  - execute your program one line at a time ("step")
  - inspect variable values
  - stop your program at a particular line ("breakpoint")
  - open a "core" file (after program crashes)
- HPC debuggers
  - support multithreaded code
  - support MPI code
  - support GPU code
  - provide a nice GUI



# Compilation flags for debugging

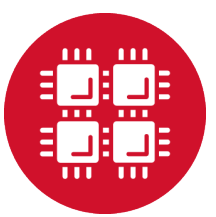
For debugging:

- Use `-g` flag
- Remove optimization or set to `-O0`
- Examples:
  - `icc -g -o mycode mycode.c`
  - `gcc -g -O0 -o mycode mycode.c`
- Use `icc -help diag` to see what compiler warnings and diagnostic options are available for the Intel compiler
- Diagnostic options can also be found by reading the `man` page of `gcc` with `man gcc`

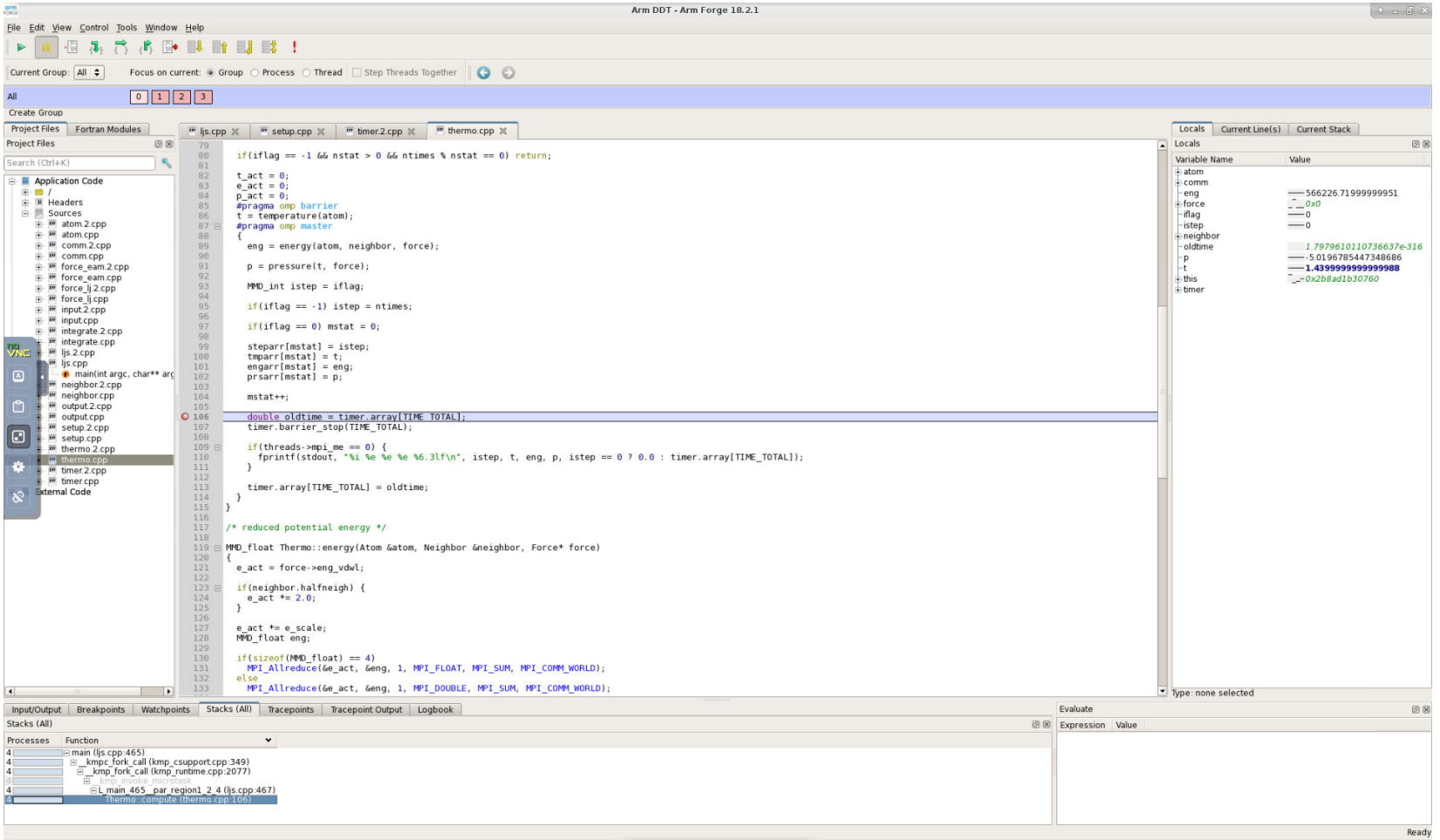


# ARM DDT

- Available on all OSC clusters
  - `module load arm-ddt`
- To run a non-MPI program from the command line:
  - `ddt --offline --no-mpi ./mycode [args]`
- To run a MPI program from the command line:
  - `ddt --offline -np num_procs ./mycode [args]`



# ARM DDT

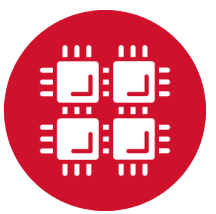

Arm DDT - Arm Forge 18.2.1
File Edit View Control Tools Window Help
Current Group: All
Focus on current: Group Process Thread Step Threads Together
Create Group
Project Files Fortran Modules
Application Code
Headers
Sources
atom.2.cpp
atom.cpp
comm.2.cpp
comm.cpp
force_eam.2.cpp
force_eam.cpp
force_lj.2.cpp
force_lj.cpp
input.2.cpp
input.cpp
integrate.2.cpp
integrate.cpp
ljs.2.cpp
ljs.cpp
main(int argc, char** arg
neighbor.2.cpp
neighbor.cpp
output.2.cpp
output.cpp
setup.2.cpp
setup.cpp
thermo.2.cpp
thermo.cpp
timer.2.cpp
timer.cpp
External Code
ljs.cpp
setup.cpp
timer.2.cpp
thermo.cpp
if(iflag == -1 && nstat > 0 && ntimes % nstat == 0) return;
t_act = 0;
e_act = 0;
p_act = 0;
#pragma omp barrier
t = temperature(atom);
#pragma omp master
{
eng = energy(atom, neighbor, force);
p = pressure(t, force);
MMD_int istep = iflag;
if(iflag == -1) istep = ntimes;
if(iflag == 0) mstat = 0;
steparr[mstat] = istep;
tmparr[mstat] = t;
engarr[mstat] = eng;
prsarr[mstat] = p;
mstat++;
double oldtime = timer.array[TIME_TOTAL];
timer.barrier_stop(TIME_TOTAL);
if(threads->mpi_me == 0) {
fprintf(stdout, "%i %e %e %e %6.3lf\n", istep, t, eng, p, istep == 0 ? 0.0 : timer.array[TIME_TOTAL]);
}
timer.array[TIME_TOTAL] = oldtime;
}
}
/* reduced potential energy */
MMD_float Thermo::energy(Atom &atom, Neighbor &neighbor, Force* force)
{
e_act = force->eng_vdwl;
if(neighbor.halfneigh) {
e_act *= 2.0;
}
e_act *= e_scale;
MMD_float eng;
if(sizeof(MMD_float) == 4)
MPI_Allreduce(&e_act, &eng, 1, MPI_FLOAT, MPI_SUM, MPI_COMM_WORLD);
else
MPI_Allreduce(&e_act, &eng, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
Locals Current Line(s) Current Stack
Variable Name Value
atom
comm
eng 566226.71999999951
force 0x0
iflag 0
istep 0
neighbor
oldtime 1.7979610110736637e-316
p -5.0196785447348686
t 1.4399999999999988
this 0x2b8ad1b30760
timer
Type: none selected
Input/Output Breakpoints Watchpoints Stacks (All) Tracepoints Tracepoint Output Logbook
Stacks (All)
Processes Function
4 main (ljs.cpp:465)
4 __kmpc_fork_call (kmp_csupport.cpp:349)
4 __kmp_fork_call (kmp_runtime.cpp:2077)
4 kmp_invoke_microtask
4 L_main_465__par_region1_2_4 (ljs.cpp:467)
4 Thermo::compute (thermo.cpp:106)
Evaluate
Expression Value
Ready

# Hands-on - Debugging with DDT

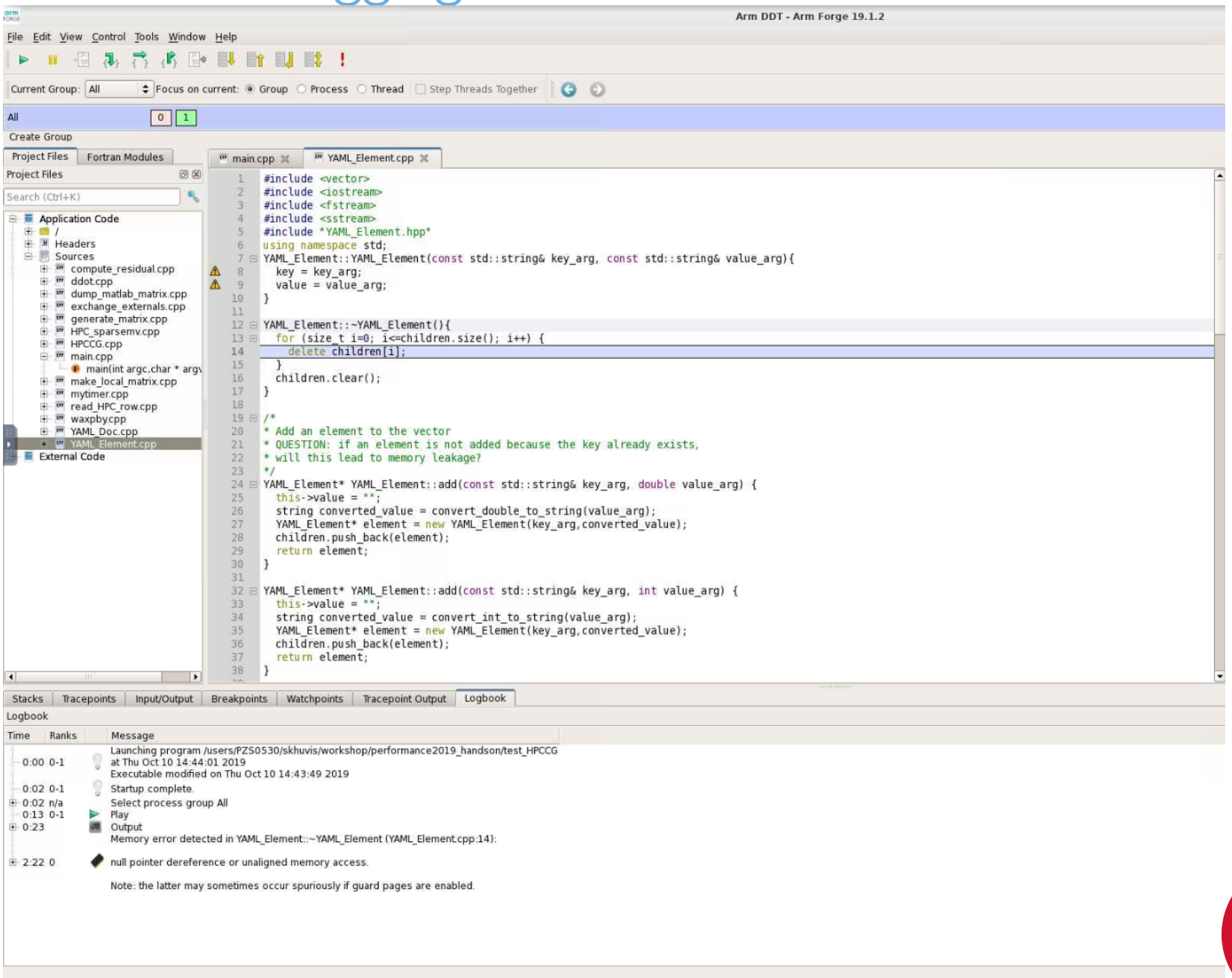
- Compile and run the code:

```
make
mpiexec -np 2 ./test_HPCCG 150 150 150
```

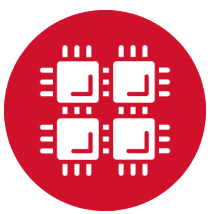
- Debug any issues with ARM DDT:
  - Set compiler flags to `-O0 -g` (`CPP_OPT_FLAGS` in Makefile), then recompile
  - `make clean; make`
  - `module load arm-ddt`
  - `ddt -np 2 ./test_hpcg 150 150 150`



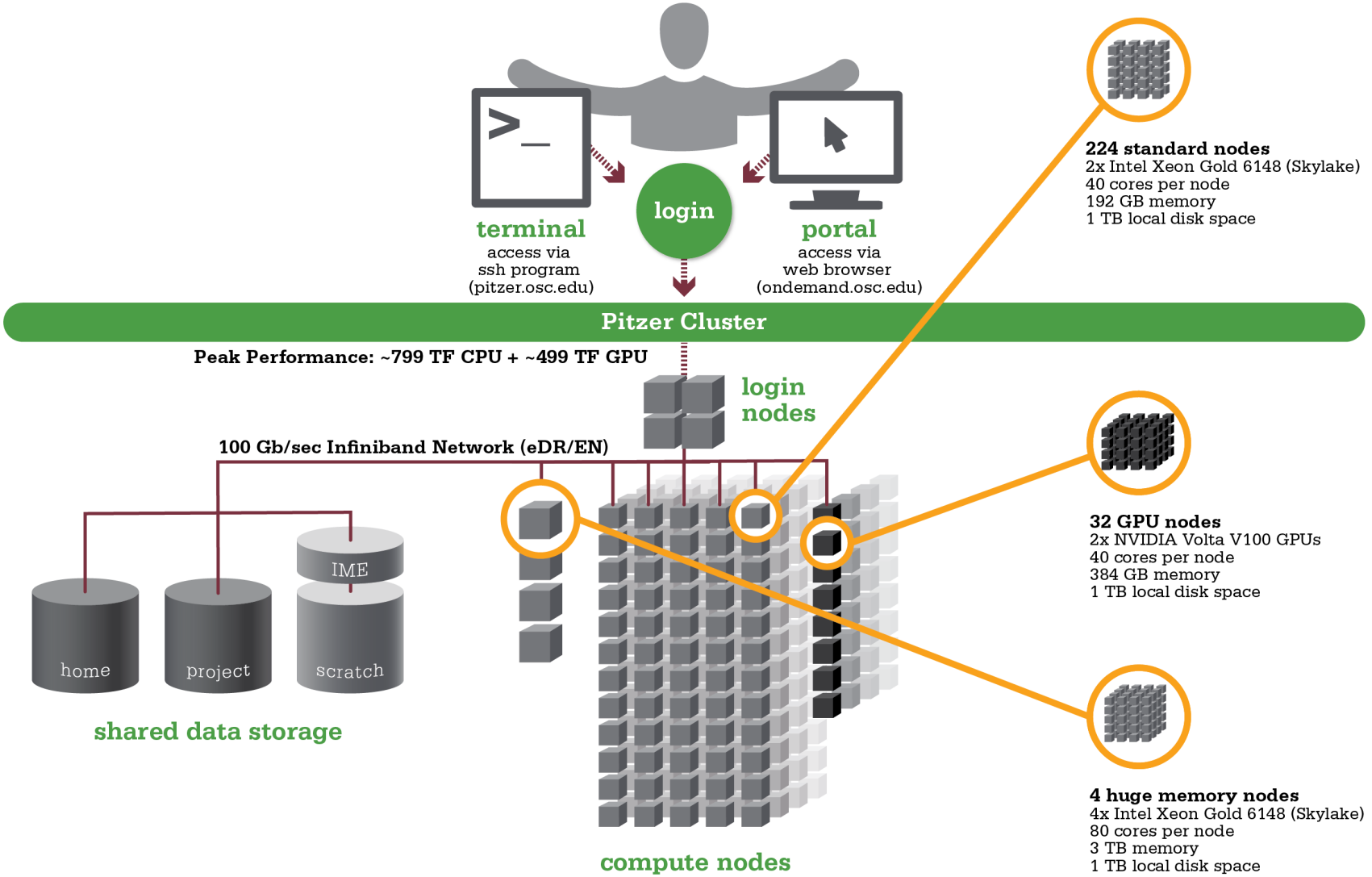
# Hands-on - Debugging with DDT - Solution



# Hardware Overview

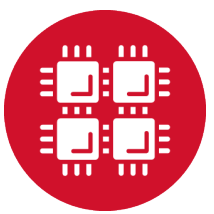


# Pitzer Cluster Specification

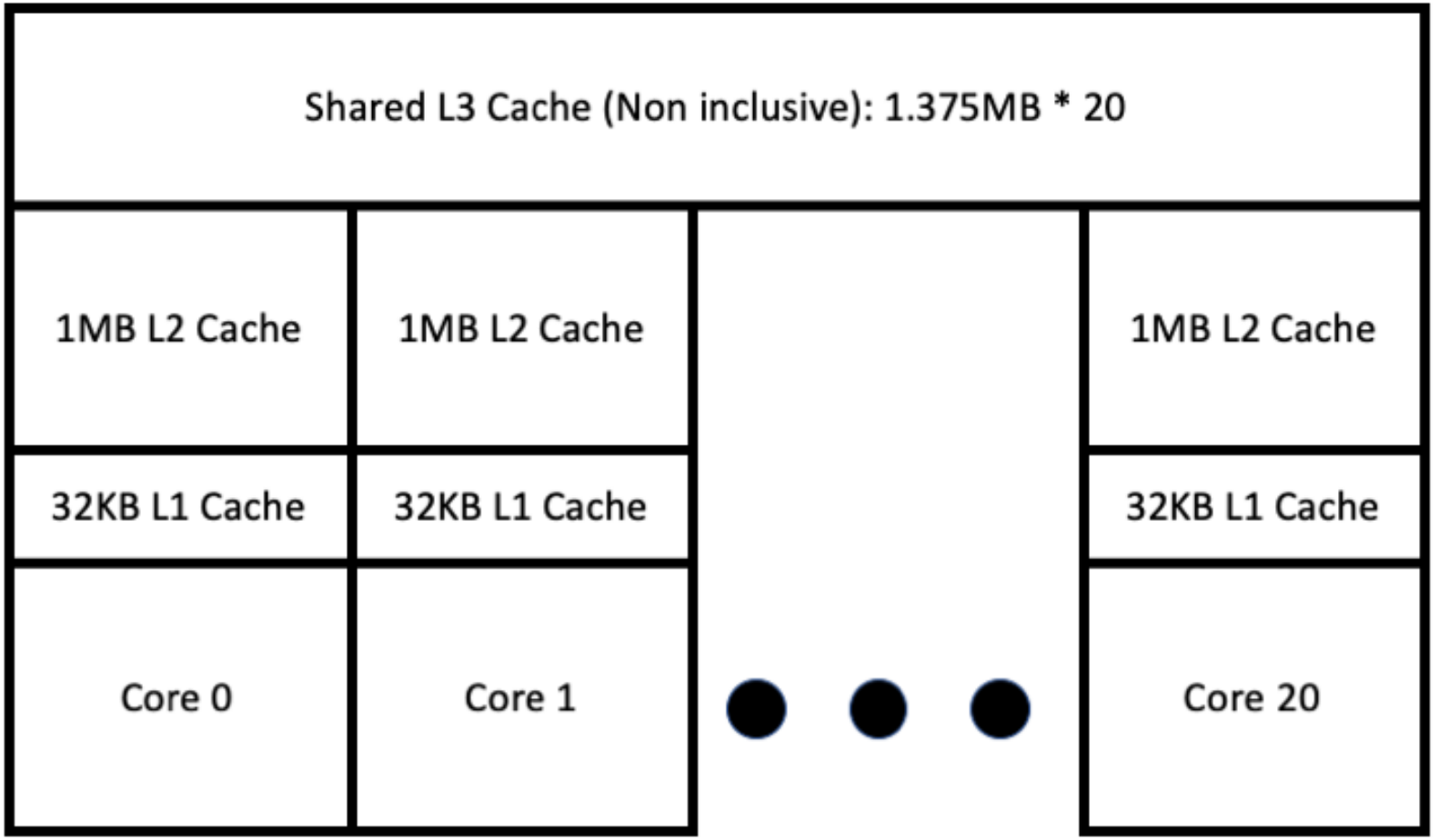


# Pitzer Cache Statistics

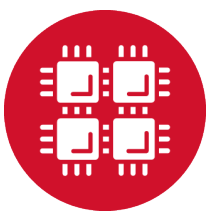
| Cache level | Size (KB) | Latency (cycles) | Max BW (bytes/cycle) | Sustained BW (bytes/cycle) |
|---|---|---|---|---|
| L1 DCU | 32 | 4–6 | 192 | 133 |
| L2 MLC | 1024 | 14 | 64 | 52 |
| L3 LLC | 28160 | 50–70 | 16 | 15 |



# Pitzer Cache Structure

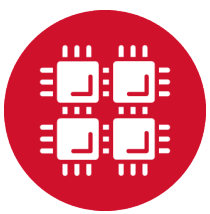



- L3 cache bandwidth is $\sim 5x$ bandwidth of main memory
- L2 cache bandwidth is $\sim 20x$ bandwidth of main memory
- L1 cache bandwidth is $\sim 60x$ bandwidth of main memory



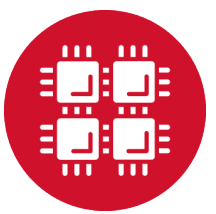
# Some Processor Features

- 40 cores per node
  - 20 cores per socket * 2 sockets per node
- Vector unit
  - Supports AVX512
  - Vector length 8 double or 16 single precision values
  - Fused multiply-add
- Hyperthreading
  - Hardware support for 4 threads per core
  - Not currently enabled on OSC systems

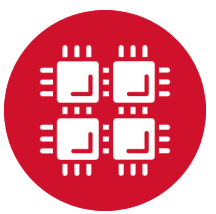


# Keep data close to the processor - file systems

- **NEVER DO HEAVY I/O IN YOUR HOME DIRECTORY!**
  - Home directories are for long-term storage, not scratch files
  - One user's heavy I/O load can affect all users
- For I/O-intensive jobs
  - Local disk on compute node (not shared)
    - Stage files to and from home directory into $TMPDIR using the `pbsdcp` command (i.e. `pbsdcp file1 file2 $TMPDIR`)
    - Execute program in $TMPDIR
  - Scratch file system
    - /fs/scratch/username or $PFSDIR
    - Faster than other file systems
    - Good for parallel jobs
    - May be faster than local disk
  - For more information about OSC's filesystem see osc.edu/supercomputing/storage-environment-at-osc/available-file-systems
  - For example batch scripts showing use of $TMPDIR and $PFSDIR see osc.edu/supercomputing/batch-processing-at-osc/job-scripts

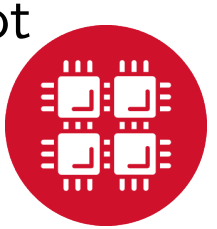


# Performance measurement and analysis



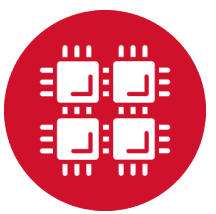
# What is good performance

- FLOPS
  - Floating Point OPerations per Second
- Peak performance
  - Theoretical maximum (all cores fully utilized)
  - Pitzer - 720 trillion FLOPS (720 teraflops)
- Sustained performance
  - LINPACK benchmark
    - Solves a dense system of linear equations
    - Pitzer - 543 teraflops
  - STREAM benchmark
    - Measures sustainable memory bandwidth (in MB/s) and the corresponding computation rate for vector kernels.
    - Applications are often memory-bound, meaning performance is limited by memory bandwidth of the system
    - Pitzer - Copy: 299095.01 MB/s, scale: 298741.01 MB/s, add: 331719.18 MB/s, triad: 331712.19 MB/s
- Application performance is typically much less than peak/sustained performance since applications usually do not take full advantage of all hardware features.



# Performance Measurement and Analysis

- Wallclock time
  - How long the program takes to run
- Performance reports
  - Easy, brief summary
- Profiling
  - Detailed information, more involved

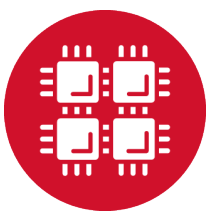


# Timing - command line

- Time a program
  - `/usr/bin/time` command

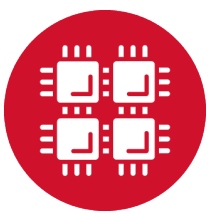
```
/usr/bin/time j3
5415.03user 13.75system 1:30:29elapsed 99%CPU \
(0avgtext+0avgdata 0maxresident)k \
0inputs+0outputs (255major+509333minor)pagefaults 0 swaps
```

  - Note: Hardcode the path - less information otherwise
- `/usr/bin/time` gives results for
  - user time (CPU time spent running your program)
  - system time (CPU time spent by your program in system calls)
  - elapsed time (wallclock)
- % CPU = (user+system)/elapsed
- memory, pagefault, and swap statistics
- I/O statistics



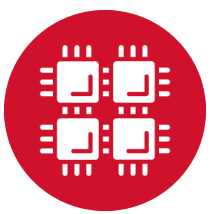
# Timing routines embedded in code

- Time portions of your code
  - C/C++
    - Wallclock: time(2), difftime(3), getrusage(2)
    - CPU: times(2)
  - Fortran 77/90
    - Wallclock: SYSTEM_CLOCK(3)
    - CPU: DTIME(3), ETIME(3)
  - MPI (C/C++/Fortran)
    - Wallclock: MPI_Wtime(3)



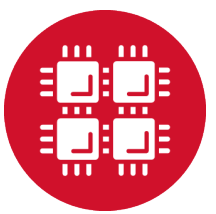
# Profiling Tools Available at OSC

- Profiling tools
  - ARM Performance Reports
  - ARM MAP
  - Intel VTune
  - Intel Trace Analyzer and Collector (ITAC)
  - Intel Advisor
  - TAU Commander
  - HPCToolkit



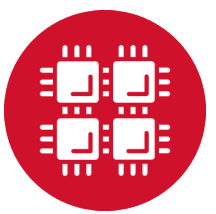
# What can a profiler show you?

- Whether code is
  - compute-bound
  - memory-bound
  - communication-bound
- How well the code uses available resources
  - Multiple cores
  - Vectorization
- How much time is spent in different parts of the code



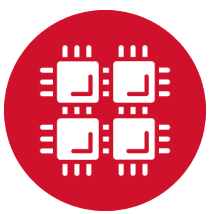
# Compilation flags for profiling

- For profiling
  - Use `-g` flag
  - Explicitly specify optimization level -On
  - Example: `icc -g -O3 -o mycode mycode.c`
- Use the same level of optimization you normally do
  - Bad example: `icc -g -o mycode mycode.c`
    - Equivalent to `-O0`



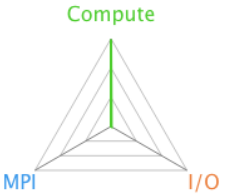
# ARM Performance Reports

- Easy to use
  - "-g" flag not needed - works on precompiled binaries
- Gives a summary of your code's performance
  - view report with browser
- For a non-MPI program:
  - `module load arm-pr`
  - `perf-report --no-mpi ./mycode [args]`
- For an MPI program:
  - `perf-report -np num_procs ./mycode [args]`



arm PERFORMANCE REPORTS

| | |
|---|---|
| Command: | /fs/project/PZS0720/skhuvis/SETSM/setsm dataset/WV01_15MAY080613301-P1BS-102001003C02A600.tif dataset/WV01_15MAY080614188-P1BS-102001003EA5DA00.tif out -outres 8 -projection ps |
| Resources: | 1 node (40 physical, 40 logical cores per node) |
| Tasks: | 1 process, OMP_NUM_THREADS was 28 |
| Machine: | p0165.ten.osc.edu |
| Start time: | Fri Dec 28 2018 14:13:20 (UTC-05) |
| Total time: | 372 seconds (about 6 minutes) |
| Full path: | /fs/project/PZS0720/skhuvis/SETSM |




## Summary: setsm is Compute-bound in this configuration

| | | |
|---|---|---|
| Compute | 99.3% | Time spent running application code. High values are usually good. This is **very high**; check the CPU performance section for advice |
| MPI | 0.0% | Time spent in MPI calls. High values are usually bad. This is **very low**; this code may benefit from a higher process count |
| I/O | 0.7% | Time spent in filesystem I/O. High values are usually bad. This is **very low**; however single-process I/O may cause MPI wait times |

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

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

### CPU

A breakdown of the 99.3% CPU time:

| | |
|---|---|
| Single-core code | 44.5% |
| OpenMP regions | 55.5% |
| Scalar numeric ops | 21.8% |
| Vector numeric ops | 4.4% |
| Memory accesses | 43.7% |

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

Little time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

### MPI

A breakdown of the 0.0% MPI time:

| | |
|---|---|
| Time in collective calls | 0.0% |
| Time in point-to-point calls | 0.0% |
| Effective process collective rate | 0.00 bytes/s |
| Effective process point-to-point rate | 0.00 bytes/s |

No time is spent in MPI operations. There's nothing to optimize here!

## I/O

A breakdown of the 0.7% I/O time:

| | | |
|---|---|---|
| Time in reads | 71.4% |  |
| Time in writes | 28.6% |  |
| Effective process read rate | 2.88 GB/s |  |
| Effective process write rate | 3.23 GB/s |  |

Most of the time is spent in read operations with a high effective transfer rate. It may be possible to achieve faster effective transfer rates using asynchronous file operations.

## OpenMP

A breakdown of the 55.5% time in OpenMP regions:

| | | |
|---|---|---|
| Computation | 78.5% |  |
| Synchronization | 21.5% |  |
| Physical core utilization | 70.0% |  |
| System load | 57.9% |  |

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 | 1.16 GiB |  |
| Peak process memory usage | 3.70 GiB |  |
| Peak node memory usage | 8.0% |  |

The peak node memory usage is very low. Larger problem sets can be run before scaling to multiple nodes.

## Energy

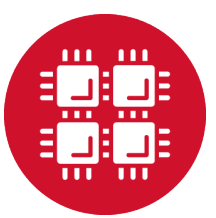
A breakdown of how the 19.1 Wh was used:

| | | |
|---|---|---|
| CPU | 100.0% |  |
| System | not supported % | |
| Mean node power | not supported W | |
| Peak node power | 0.00 W | |

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

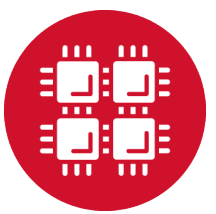
System power metrics: No Arm IPMI Energy Agent config file found in /var/spool/ipmi-energy-agent. Did you start the Arm IPMI Energy Agent?

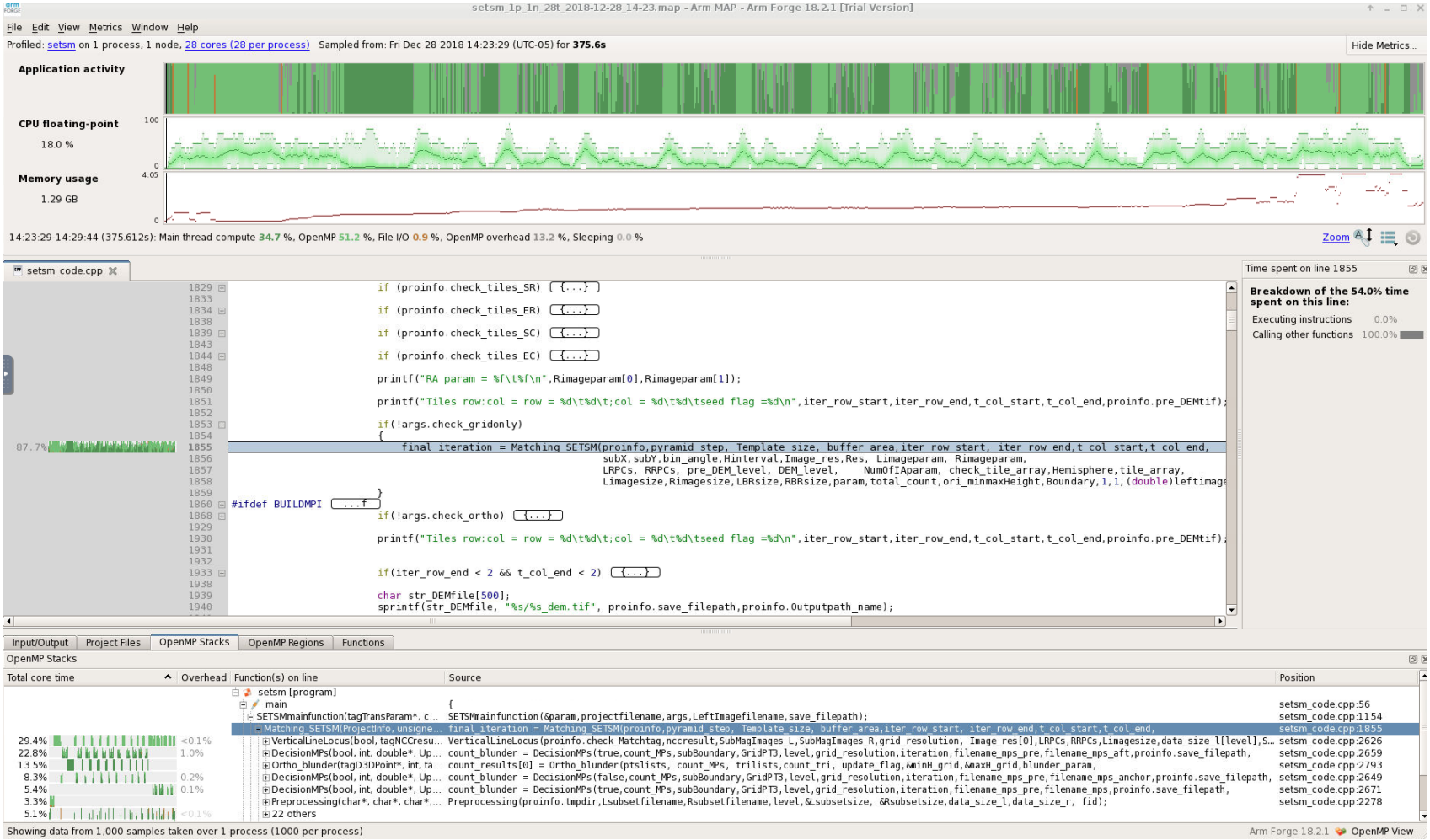




# ARM MAP

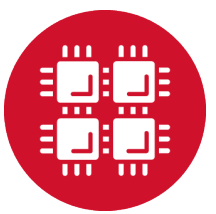
- Interpretation of profile requires some expertise
- Gives details about your code's performance
- For a non-MPI program:
  - module load arm-map
  - `map --profile --no-mpi ./mycode [args]`
- For an MPI program:
  - `map --profile -np num_procs ./mycode [args]`
- View and explore resulting profile using ARM client




setsm_1p_1n_28t_2018-12-28_14-23.map - Arm MAP - Arm Forge 18.2.1 [Trial Version]
File Edit View Metrics Window Help
Profiled: setsm on 1 process, 1 node, 28 cores (28 per process) Sampled from: Fri Dec 28 2018 14:23:29 (UTC-05) for 375.6s
Hide Metrics...
Application activity
CPU floating-point
18.0 %
Memory usage
1.29 GB
14:23:29-14:29:44 (375.612s): Main thread compute 34.7 %, OpenMP 51.2 %, File I/O 0.9 %, OpenMP overhead 13.2 %, Sleeping 0.0 %
Zoom
setsm_code.cpp
Time spent on line 1855
Breakdown of the 54.0% time spent on this line:
Executing instructions 0.0%
Calling other functions 100.0%
Input/Output
Project Files
OpenMP Stacks
OpenMP Regions
Functions
OpenMP Stacks
Total core time
Overhead
Function(s) on line
Source
Position
setsm [program]
main
SETSMmainfunction(tagTransParam*, c...
Matching SETSM(ProjectInfo, unsigne...
VerticalLineLocus(bool, tagNCCresu...
DecisionMPs(bool, int, double*, Up...
Ortho_blunder(tagD3DPoint*, int, ta...
DecisionMPs(bool, int, double*, Up...
DecisionMPs(bool, int, double*, Up...
Preprocessing(char*, char*, char*,...
22 others
Showing data from 1,000 samples taken over 1 process (1000 per process)
Arm Forge 18.2.1 OpenMP View

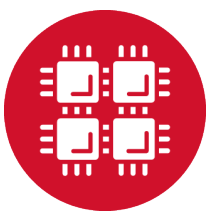
# More information about ARM Tools

- www.osc.edu/resources/available_software/software_list/ARM
- www.arm.com

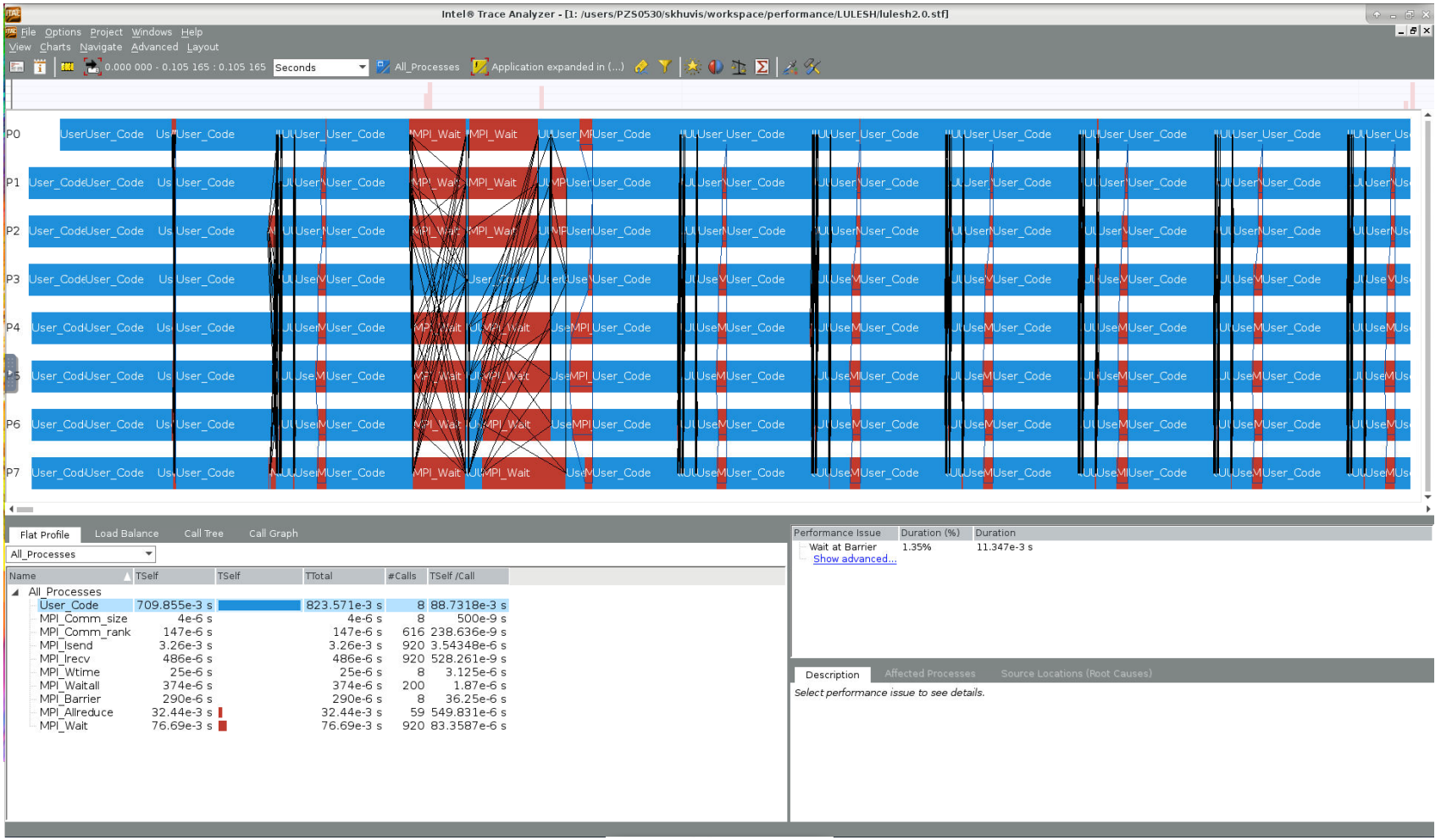


# Intel Trace Analyzer and Collector (ITAC)

- Graphical tool for profiling MPI code (Intel MPI)
- To use:
  - `module load intelmpi # then compile (-g) code`
  - `mpiexec -trace ./mycode`
- View and explore existing results using GUI with `traceanalyzer`:
  - `traceanalyzer <mycode>.stf`

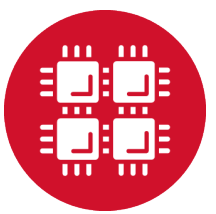


# ITAC GUI

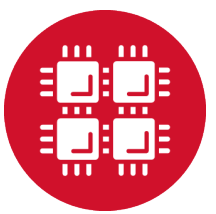

Intel® Trace Analyzer - [1: /users/PZS0530/skhuvis/workspace/performance/LULESH/lulesh2.0.stf]
File Options Project Windows Help
View Charts Navigate Advanced Layout
0.000 000 - 0.105 165 : 0.105 165
Seconds
All_Processes
Application expanded in (...)
P0
P1
P2
P3
P4
P5
P6
P7
User_Code
MPI_Wait
Flat Profile
Load Balance
Call Tree
Call Graph
All_Processes
Name
TSelf
TTotal
#Calls
TSelf /Call
All_Processes
User_Code 709.855e-3 s 823.571e-3 s 8 88.7318e-3 s
MPI_Comm_size 4e-6 s 4e-6 s 8 500e-9 s
MPI_Comm_rank 147e-6 s 147e-6 s 616 238.636e-9 s
MPI_Isend 3.26e-3 s 3.26e-3 s 920 3.54348e-6 s
MPI_Irecv 486e-6 s 486e-6 s 920 528.261e-9 s
MPI_Wtime 25e-6 s 25e-6 s 8 3.125e-6 s
MPI_Waitall 374e-6 s 374e-6 s 200 1.87e-6 s
MPI_Barrier 290e-6 s 290e-6 s 8 36.25e-6 s
MPI_Allreduce 32.44e-3 s 32.44e-3 s 59 549.831e-6 s
MPI_Wait 76.69e-3 s 76.69e-3 s 920 83.3587e-6 s
Performance Issue
Duration (%)
Duration
Wait at Barrier 1.35% 11.347e-3 s
Show advanced...
Description
Affected Processes
Source Locations (Root Causes)
Select performance issue to see details.

# Profiling - What to look for?

- Hot spots - where most of the time is spent
  - This is where we'll focus our optimization effort
- Excessive number of calls to short functions
  - Use inlining! (compiler flags)
- Memory usage
  - Swapping, thrashing - not allowed at OSC (job gets killed)
- CPU time vs wall time (% CPU)
  - Low CPU utilization may mean excessive I/O delays

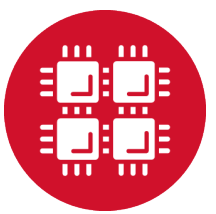


# Help from the compiler



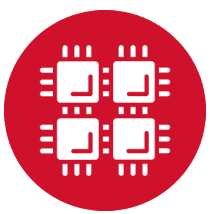
# Compiler and Language Choice

- HPC software traditionally written in Fortran or C/C++
- OSC supports several compiler families
  - Intel (icc, icpc, ifort)
    - Usually gives fastest code on Intel architecture
  - Portland Group (PGI - pgcc, pgc++, pgf90)
    - Good for GPU programming, OpenACC
  - GNU (gcc, g++, gfortran)
    - Open source, universally available



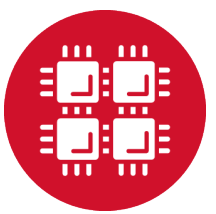
# Compiler Options for Performance Tuning

- Why use compiler options?
  - Processors have a high degree of internal concurrency
  - Compilers do an amazing job at optimization
  - Easy to use - Let the compiler do the work!
  - Reasonably portable performance
- Optimization options
  - Let you control aspects of the optimization
- Warning:
  - Different compilers have different default values for options



# Compiler Optimization

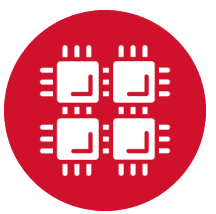
- Function inlining
  - Eliminate function calls
- Interprocedural optimization/analysis (ipo/ipa)
  - Same file or multiple files
- Loop transformations
  - Unrolling, interchange, splitting, tiling
- Vectorization
  - Operate on arrays of operands
- Automatic parallelization of loops
  - Very conservative multithreading



# What compiler flags to try first?

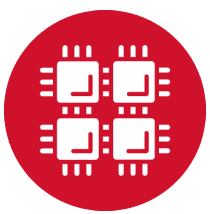
- General optimization flags (-O2, -O3, -fast)
- Fast math
- Interprocedural optimization/analysis

- Profile again, look for changes
- Look for new problems/opportunities



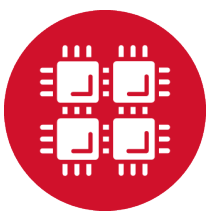
# Floating Point Speed vs. Accuracy

- Faster operations are sometimes less accurate
- Some algorithms are okay, some quite sensitive
- Intel compilers
  - Fast math by default with -O2 and -O3
  - Use `-fp-model precise` if you have a problem (slower)
- GNU compilers
  - Precise math by default with -O2 and -O3 (slower)
  - Use `-ffast-math` for faster performance



# Interprocedural Optimization/Inlining

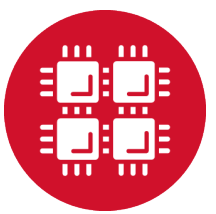
- Inlining
  - Replace a subroutine or function call with the actual body of the subprogram
- Advantages
  - Overhead of calling the subprogram is eliminated
  - More loop optimizations are possible if calls are eliminated
- One source file
  - Typically automatic with -O2 and -O3
- Multiple source files compiled separately
  - Use compiler option for compile and link phases



# Optimization Compiler Options - Intel compilers

| | |
|---|---|
| `-fast` | Common optimizations |
| `-On` | Set optimization level (0,1,2,3) |
| `-ipo` | Interprocedural optimization, multiple files |
| `-O3` | Loop transforms |
| `-xHost` | Use highest instruction set available |
| `-parallel` | Loop auto-parallelization |

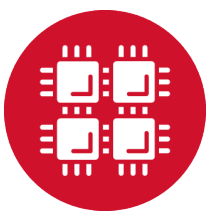
- Don't use **-fast** for MPI programs with Intel compilers
- Use same compiler command to link for **-ipo** with separate compilation
- Many other optimization options are available
- See **man** pages for details
- Recommended options:
  `-O3 -xHost`
- Example:
  `ifort -O3 program.f90`



# Optimization Compiler Options - PGI compilers

| | |
|---|---|
| `-fast` | Common optimizations |
| `-On` | Set optimization level (0,1,2,3,4) |
| `-Mipa` | Interprocedural analysis |
| `-Mconcur` | Loop auto-parallelization |

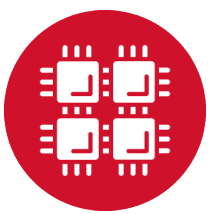
- Many other optimization options are available
- Use same compiler command to link for **-Mipa** with separate compilation
- See **man** pages for details
- Recommended options: `-fast`
- Example:
```
pgf90 -fast
program.f90
```



# Optimization Compiler Options - GNU compilers

| | |
|---|---|
| `-On` | Set optimization level (0,1,2,3) |
| N/A for separate compilation | Interprocedural optimization |
| `-O3` | Loop transforms |
| `-ffast-math` | Potentially unsafe float pt optimizations |
| `-march=native` | Use highest instruction set available |

- Many other optimization options are available
- See **man** pages for details
- Recommended options:
  `-O3 -ffast-math`
- Example:
  `gfortran -O3 program.f90`

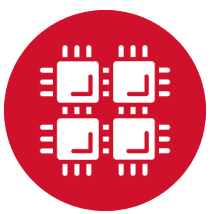


# Hands-on – Compiler options

- Compile and run with different compiler options.

```
time mpiexec -np 2 ./test_HPCCG 150 150 150
```

- Which compiler options give the best performance?



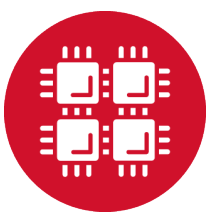
# Hands-on – Compiler options – Sample times

- Compile and run with different compiler options.

```
time mpiexec -np 2 ./test_HPCCG 150 150 150
```

- Which compiler options give the best performance?

| Option | Time |
|---|---|
| -g | 129 |
| -O0 -g | 129 |
| -O1 -g | 74 |
| -O2 -g | 74 |
| -O3 -g | 74 |

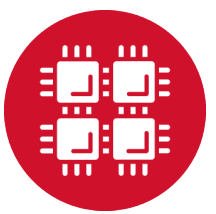


# Hands-on - Performance Report

Now that you have selected the optimal compiler flags, get an overview of the bottlenecks in the code with the ARM performance report.

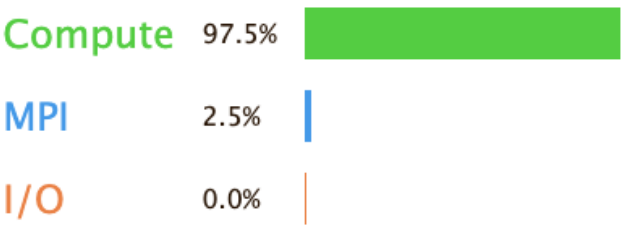
```
module load arm-pr
perf-report -np 2 ./test_HPCCG 150 150 150
```

Open the html file in your browser to view the report. What are the bottlenecks in the code?



# Hands-on - Performance Report

## Summary: test_HPCCG is Compute-bound in this configuration




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

Time spent in MPI calls. High values are usually bad.
This is **very low**; this code may benefit from a higher process count

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.

As very little time is spent in MPI calls, this code may also benefit from running at larger scales.

### CPU

A breakdown of the 97.5% CPU time:

| | |
|---|---|
| Scalar numeric ops | 30.6% |
| Vector numeric ops | 0.2% |
| Memory accesses | 69.2% |

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

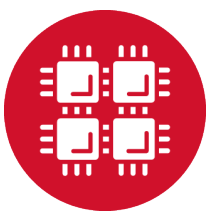
Little time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

### MPI

A breakdown of the 2.5% MPI time:

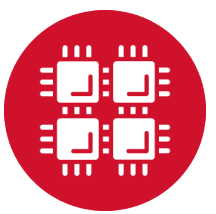
| | |
|---|---|
| Time in collective calls | 79.6% |
| Time in point-to-point calls | 20.4% |
| Effective process collective rate | 74.4 bytes/s |
| Effective process point-to-point rate | 126 MB/s |

Most of the time is spent in collective calls with a very low transfer rate. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.



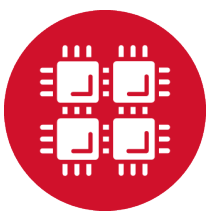
# Compiler Optimization Reports

- Let you understand
  - how well the compiler is doing at optimizing your code
  - what parts of code need work
- Generated at compile time
  - Describe what optimizations were applied at various points in the source code
  - May tell you why optimizations could not be performed



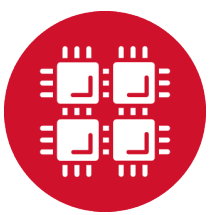
# Compiler Optimization Reports

- Intel compilers
  - `-qopt-report`
  - Output to a file
- Portland Group compilers
  - `-Minfo`
  - Output to stderr
- GNU compilers
  - `-fopt-info`
  - Output to stderr by default



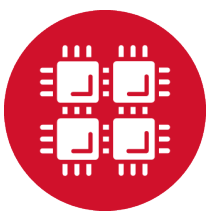
# Sample from an Optimization Report

```
LOOP BEGIN at laplace-good.f(10,7)
   remark #15542: loop was not vectorized: inner loop was already vectorized

   LOOP BEGIN at laplace-good.f(11,10)
   <Peeled loop for vectorization>
   LOOP END

   LOOP BEGIN at laplace-good.f(11,10)
      remark #15300: LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at laplace-good.f(11,10)
   <Remainder loop for vectorization>
      remark #15301: REMAINDER LOOP WAS VECTORIZED
   LOOP END

   LOOP BEGIN at laplace-good.f(11,10)
   <Remainder loop for vectorization>
   LOOP END
LOOP END
```



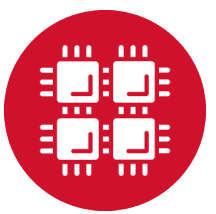
# Hands-on - Optimization Report

Add the `-qopt-report=5` compiler flag and recompile to view an optimization report.



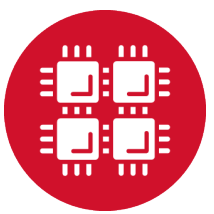
# Hands-on - Optimization Report - Solution

```
LOOP BEGIN at HPC_sparsemv.cpp(72,3)
   remark #25096: Loop Interchange not done due to: Imperfect Loop Nest (Either at Source or due to other Compiler Transformations)
   remark #25451: Advice: Loop Interchange, if possible, might help loopnest. Suggested Permutation : ( 1 2 ) --> ( 2 1 )
   remark #15344: loop was not vectorized: vector dependence prevents vectorization
   remark #15346: vector dependence: assumed FLOW dependence between y[i] (74:7) and x[*(cur_inds+j*4)] (84:11)
   remark #15346: vector dependence: assumed ANTI dependence between x[*(cur_inds+j*4)] (84:11) and y[i] (74:7)

   LOOP BEGIN at HPC_sparsemv.cpp(83,7)
      remark #15344: loop was not vectorized: vector dependence prevents vectorization
      remark #15346: vector dependence: assumed FLOW dependence between y[i] (84:11) and y[i] (84:11)
      remark #15346: vector dependence: assumed ANTI dependence between y[i] (84:11) and y[i] (84:11)
      remark #25439: unrolled with remainder by 2
   LOOP END

   LOOP BEGIN at HPC_sparsemv.cpp(83,7)
   <Remainder>
   LOOP END
LOOP END
```

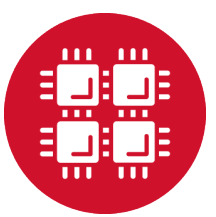


# A word about algorithms

- Problem-dependent - can't generalize
- Scalability is important
  - How computational time increases with problem size
- Replace with an equivalent algorithm of lower complexity
- Replace home-grown algorithm with call to optimized library

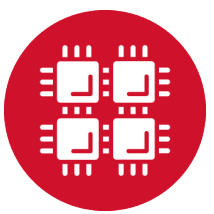


# Code tuning and optimization



# Code modifications for Optimization

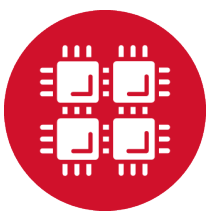
- Vectorization
  - Vectorizable loops
  - Vectorization inhibitors
- Memory optimizations
  - Unit stride memory access
  - Efficient cache usage



# Vectorization/Streaming

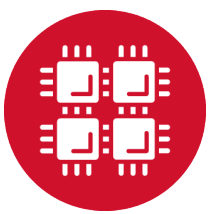
- Code is structured to operate on arrays of operands
  - Single Instruction, Multiple Data (SIMD)
- Vector instructions built into processor (AVX512, AVX, SSE, etc.)
  - Vector length 16 single or 8 double precision on Pitzer
- Best performance with unit stride
- Fortran 90, MATLAB have this idea built in
- A vectorizable loop:

```
do  i=1,N
   a(i)=b(i)+x(i)*c(i)
end  do
```



# Vectorization Inhibitors

- Not unit stride
  - Loops in wrong order (column-major vs. row-major)
    - Usually fixed by the compiler
  - Loops over derived types
- Function calls
  - Sometimes fixed by inlining
  - Can split loop into two loops
- Too many conditionals
  - "if" statements
- Indexed array accesses (i.e. `a[b[i]]`)

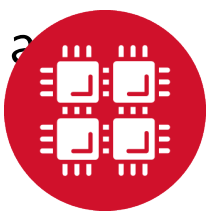


# Hands-on - Vectorization

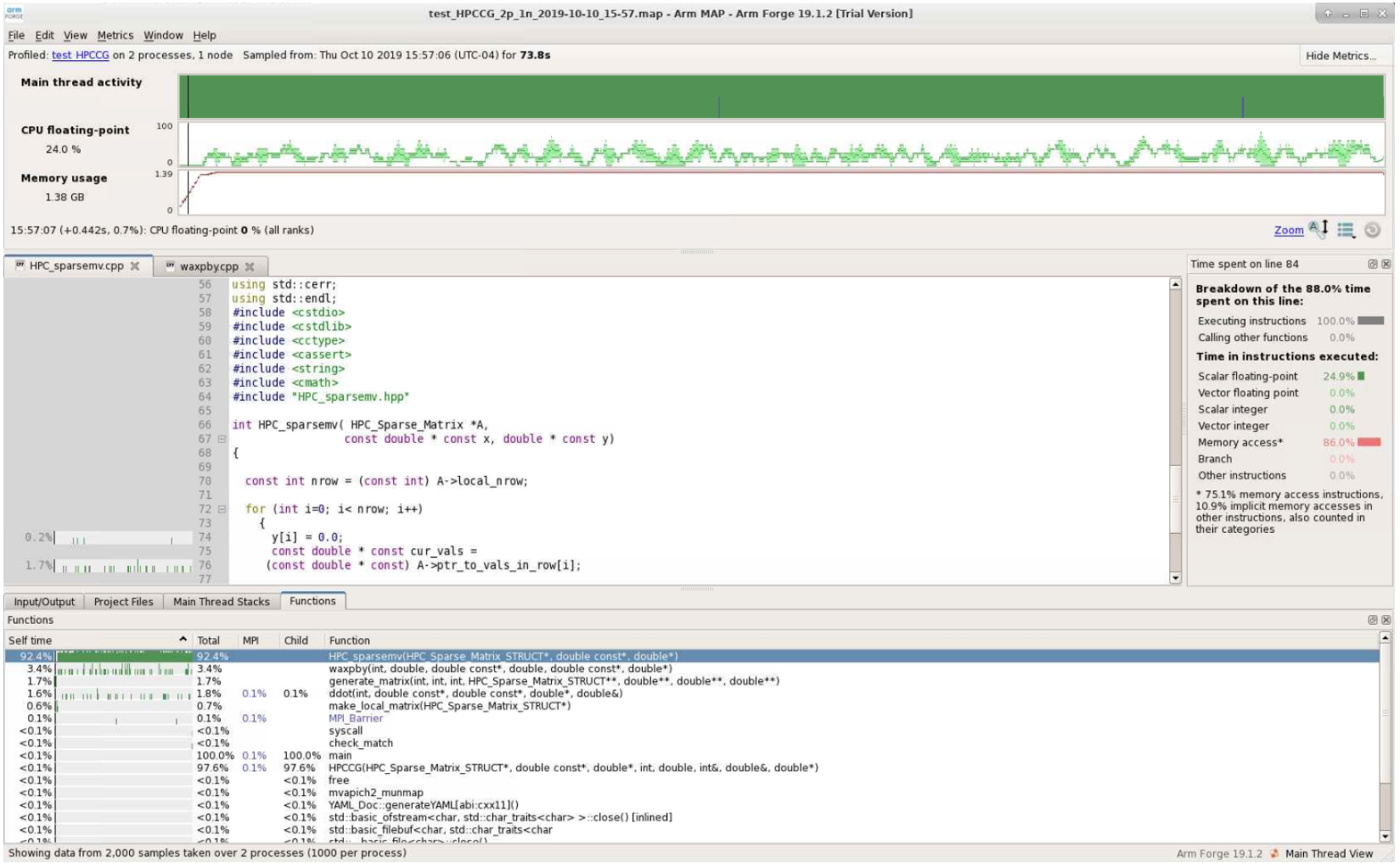
- Make sure that you are compiling with -xHost to ensure optimal vectorization.
- Look at the most expensive function in the code using ARM MAP.

```
module load arm-map
map -np 2 ./test_HPCCG 150 150 150
```

- Check the optimization report generated by the compiler (with `-qopt-report=5`) to see if any of the loops in this function is not being vectorized.
- Modify the code to enable vectorization and rerun the code. Do these changes improve performance?
- Hint: You should see a recommendation in the optimization report to add the `-qopt-zmm-usage=high` command-line option for your function. Make sure to add it to the Makefile.
- Hint: Try replacing an assignment to an array element with a temporary variable to enable vectorization.



# Hands-on - Vectorization - Solution



# Hands-on - Vectorization - Solution

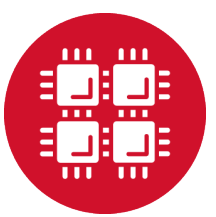
Replace lines 83–84 of `HPC_sparsemv.cpp`

```
for (int j=0; j < cur_nzz; j++)
  y[i] += cur_vals[j]*x[cur_inds[j]];
```

with

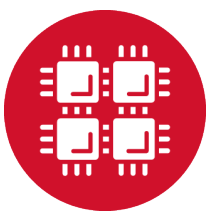
```
for (int j=0; j < cur_nzz; j++)
  sum += cur_vals[j]*x[cur_inds[j]];
y[i] = sum;
```

Reduces runtime from 74 seconds to 56 seconds.



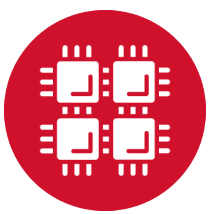
# Unit Stride Memory Access

- Often the most important factor in your code's performance!!!
- Loops that work with arrays should use a stride of one whenever possible
- C, C++ are ***row-major***, in a 2D array, they store elements consecutively by row:
  - First array index should be outermost loop
  - Last array index should be innermost loop
- Fortran is ***column-major***, so the reverse is true:
  - Last array index should be outermost loop
  - First array index should be innermost loop
- Avoid arrays of derived data types, structs, or classes (i.e. use struct of arrays (SoA) instead of arrays of structures (AoS))



# Data Layout: Object-Oriented Languages

- Arrays of objects may give poor performance on HPC systems if used naively
  - C structs
  - C++ classes
  - Fortran 90 user-defined types
- Inefficient use of cache - not unit stride
  - Can often get factor of 3 or 4 speedup just by fixing it
- You can use them efficiently! Be aware of data layout
- Data layout may be the only thing modern compilers can't optimize

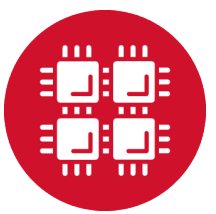


# Efficient Cache Usage

- Cache lines
  - 8 words (64 bytes) of consecutive memory
  - Entire cache line is loaded when a piece of data is fetched
- Good example - Entire cache line used
  - 2 cache lines used for every 8 loop iterations
  - Unit stride

```
real*8 a(N), b(N)
do i=1,N
   a(i)=a(i)+b(i)
end do
```

```
2 cache lines:
a(1),a(2),a(3),...a(8)
b(1),b(2),b(3),...b(8)
```

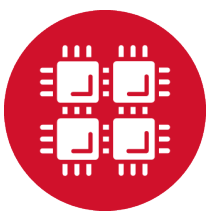


# Efficient Cache Usage - Cache Lines (cont.)

- Bad example - Unneeded data loaded
  - 1 cache line loaded for *each* loop iteration
  - 8 words loaded, only 2 words used
  - Not unit stride

```
TYPE :: node
   real*8 a, b, c, d, w, x, y, z
END TYPE node
TYPE(node) :: s(N)
do i=1,N
   s(i)%a = s(i)%a + s(i)%b
end do
```

```
cache line:
a(1),b(1),c(1),d(1),w(1),x(1),y(1),z(1)
```

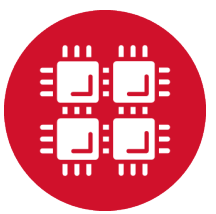


# Hands-on - Memory utilization

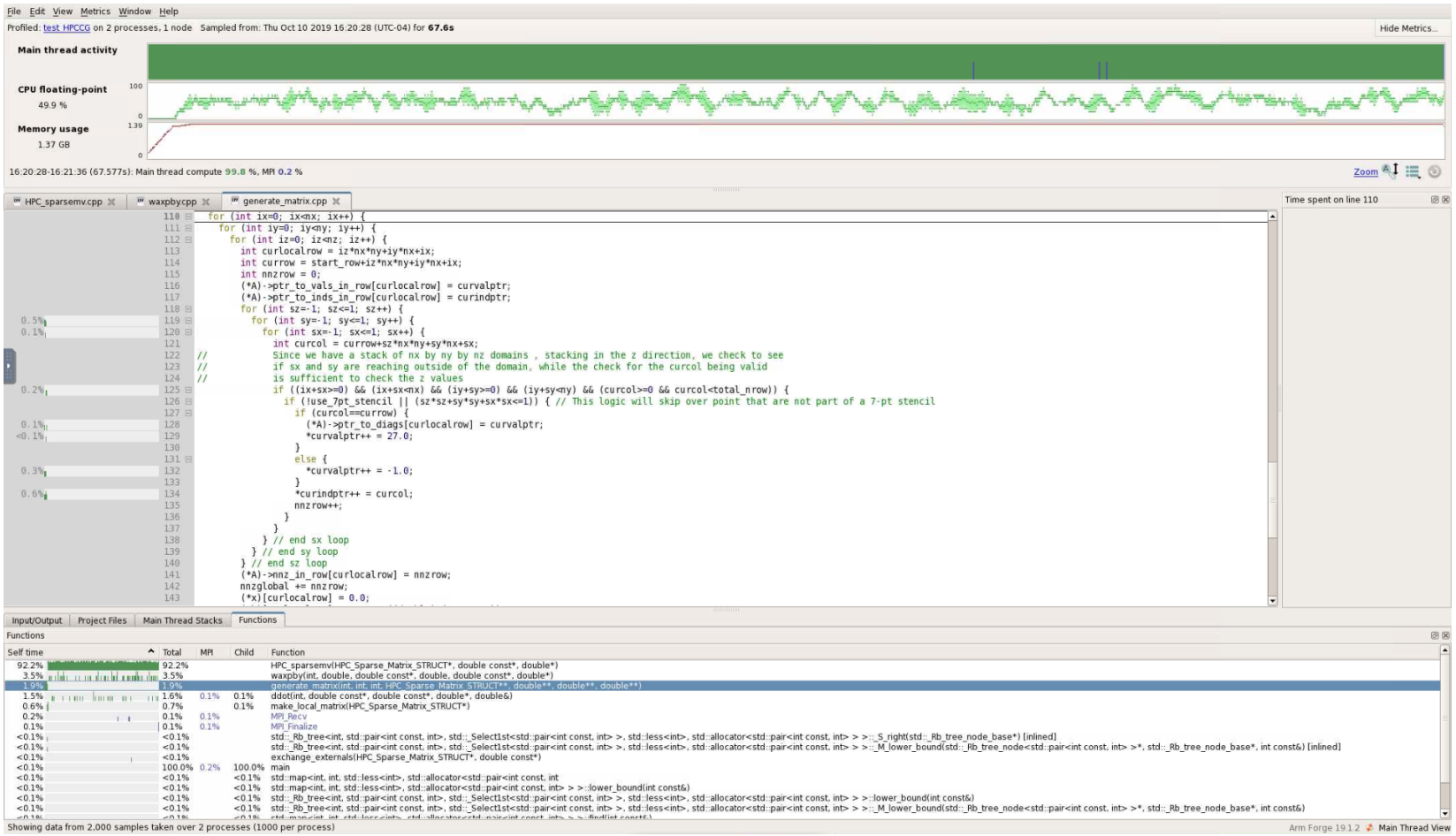
- Use ARM MAP to identify the most expensive parts of the code.

```
module load arm-map
map -np 2 ./test_HPCCG 150 150 150
```

- Look for any inefficient memory access patterns.
- Modify the code to improve memory access patterns and rerun the code. Do these changes improve performance?
- Hint: Look for nested loops that are not ordered correctly.



# Hands-on - Memory utilization - Solution



# Hands-on - Memory utilization - Solution

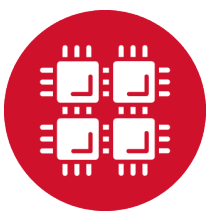
Replace lines 110–113 of `generate_matrix.cpp`:

```
for (int iz=0; iz<nz; iz++) {
  for (int iy=0; iy<ny; iy++) {
    for (int ix=0; ix<nx; ix++) {
```

with:

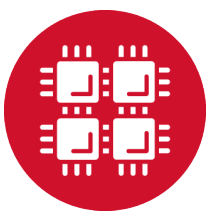
```
for (int ix=0; ix<nx; ix++) {
  for (int iy=0; iy<ny; iy++) {
    for (int iz=0; iz<nz; iz++) {
```

Reduces runtime form 56 seconds to 19 seconds.



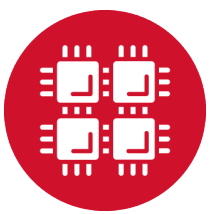
# Optimized Mathematical Libraries

- MKL (Intel Math Kernel Library)
  - BLAS
  - LAPACK
  - FFT
  - Vectorized transcendental functions (sin, cos, exp)
- AI libraries
  - Intel MKL-DNN
  - Intel DAAL
  - CuDNN
- FFTW
- ScaLAPACK
- SuperLU
- ... and many others



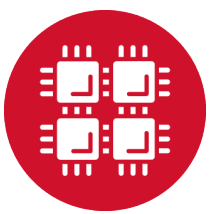
# Profiling Interpreted Languages

- Most interpreted languages have their own profiling tools
- For example, Python has cProfile and R has Profvis
- Performance considerations:
  - Vectorization
  - Efficient memory utilization
  - Using appropriate data structures
  - Use built-in functions where possible
  - Best practices for the language

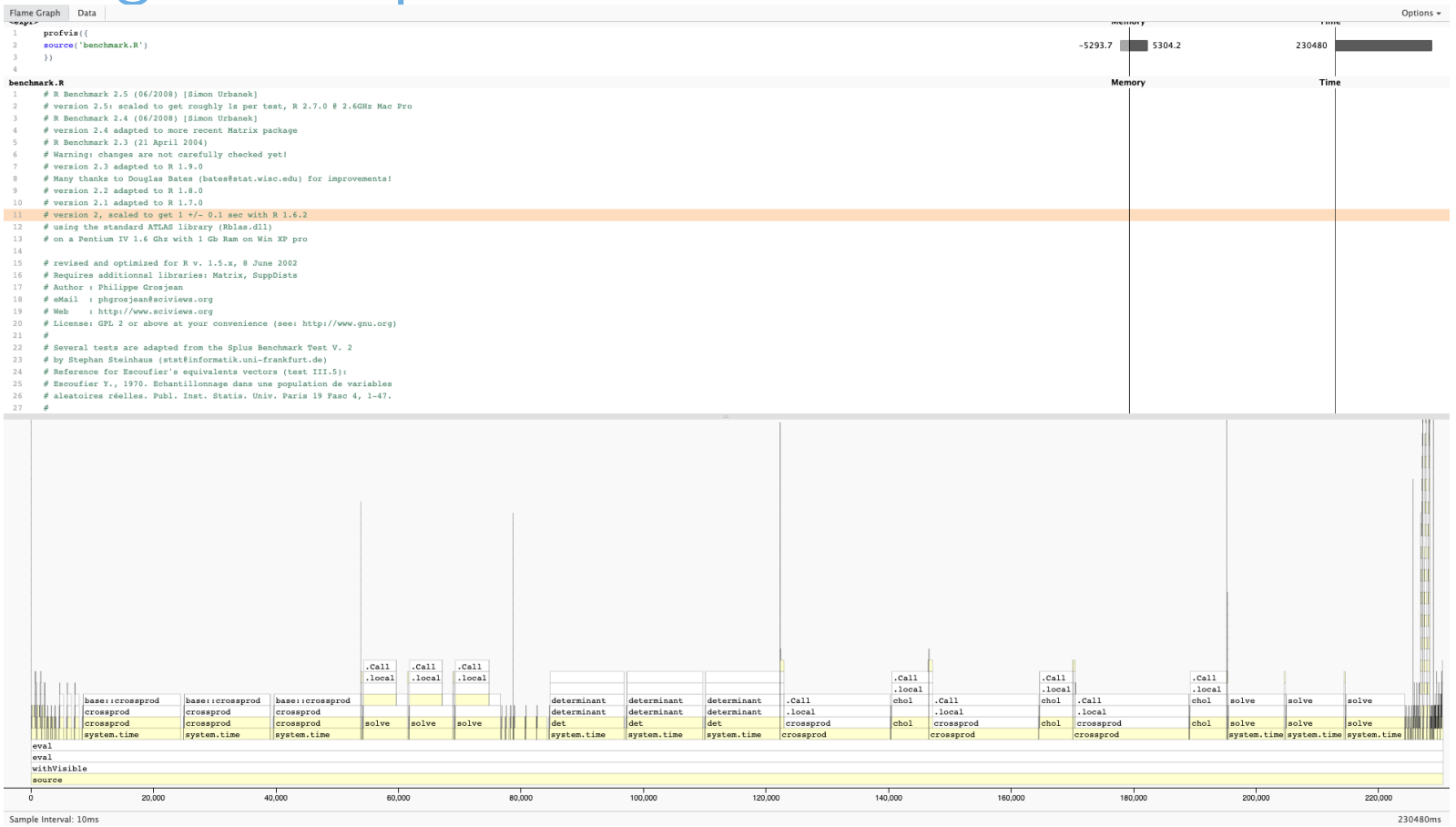


# Profiling Python with cProfile

```
skhuvis@pitzer-login01:~$ python -m cProfile -s time poisson.py
         320447 function calls (319459 primitive calls) in 8.904 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    8.018    8.018    8.018    8.018 linalg.py:327(solve)
        1    0.141    0.141    0.141    0.141 {matplotlib._delaunay.linear_interpolate_grid}
        1    0.093    0.093    0.096    0.096 numerictypes.py:81(<module>)
        4    0.060    0.015    0.279    0.070 __init__.py:1(<module>)
        1    0.038    0.038    8.313    8.313 poisson.py:37(poisson)
        1    0.027    0.027    0.384    0.384 __init__.py:106(<module>)
     1546    0.024    0.000    0.179    0.000 poisson.py:27(A_e)
     3094    0.019    0.000    0.022    0.000 defmatrix.py:191(__getitem__)
        1    0.018    0.018    0.036    0.036 overrides.py:1(<module>)
     1546    0.017    0.000    0.030    0.000 linalg.py:486(inv)
        1    0.013    0.013    0.013    0.013 hashlib.py:73(<module>)
    23196    0.012    0.000    0.015    0.000 defmatrix.py:169(__array_finalize__)
        1    0.011    0.011    0.016    0.016 __init__.py:15(<module>)
        1    0.011    0.011    0.050    0.050 __init__.py:7(<module>)
     1546    0.011    0.000    0.011    0.000 poisson.py:59(f)
        1    0.011    0.011    0.023    0.023 numeric.py:1(<module>)
        1    0.010    0.010    8.905    8.905 poisson.py:6(<module>)
     4640    0.009    0.000    0.013    0.000 {numpy.concatenate}
    15583    0.009    0.000    0.009    0.000 {numpy.array}
     1547    0.009    0.000    0.029    0.000 index_tricks.py:36(ix_)
     6184    0.009    0.000    0.009    0.000 {_warnings.warn}
     9278    0.009    0.000    0.017    0.000 shape_base.py:83(atleast_2d)
     1546    0.009    0.000    0.017    0.000 linalg.py:2040(det)
     1546    0.008    0.000    0.026    0.000 poisson.py:32(b_e)
        1    0.007    0.007    0.019    0.019 npyio.py:1(<module>)
     1546    0.007    0.000    0.043    0.000 defmatrix.py:794(getI)
        1    0.007    0.007    0.009    0.009 cbook.py:4(<module>)
        2    0.007    0.003    0.020    0.010 __init__.py:45(<module>)
     1546    0.007    0.000    0.007    0.000 {method 'reduce' of 'numpy.ufunc' objects}
        1    0.006    0.006    0.025    0.025 index_tricks.py:1(<module>)
```

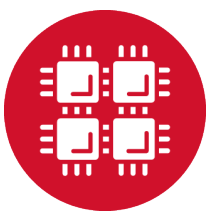


# Profiling R with profvis



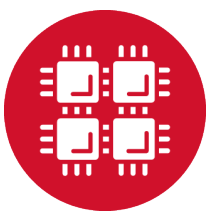

```
> install.packages('profvis')
> library('profvis')
> profvis({
source('benchmark.R')
})
```

# Parallel computing



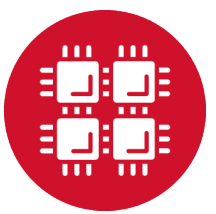
# Parallel Computing

- Multithreading
  - Shared-memory model (single node)
  - OpenMP support in compilers
- Message Passing Interface (MPI)
  - Distributed-memory model (single or multiple nodes)
  - Several available libraries
- GPUs



# What is OpenMP?

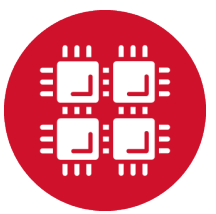
- Shared-memory, threaded parallel programming model
- Portable standard
- A set of compiler directives
- A library of support functions
- Supported by vendors' compilers
  - Intel
  - Portland Group
  - GNU
  - Cray



# Parallel loop execution - Fortran

- Inner loop vectorizes
- Outer loop executes on multiple threads

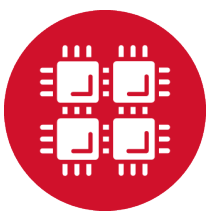
```
PROGRAM omploop
INTEGER, PARAMETER :: N = 1000
INTEGER i, j
REAL, DIMENSION(N,N) :: a, b, c, x
... ! Initialize arrays
!$OMP PARALLEL DO
do j=1,N
  do i=1,N
    a(i,j)=b(i,j)+x(i,j)*c(i,j)
  end do
end do
!$OMP END PARALLEL DO
END PROGRAM omploop
```



# Parallel loop execution - C

- Inner loop vectorizes
- Outer loop executes on multiple threads

```
int main()
{
  int N = 1000
  float *a, *b, *c, *x
... // Allocate and initialize arrays
#pragma omp parallel for
  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++) {
      a[i*N+j]=b[i*N+j]+x[i*N+j]*c[i*N+j]
    }
  }
}
```



# Compiling a program with OpenMP

- Intel compilers
  - Add the `-qopenmp` option

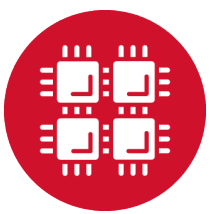
```
ifort -qopenmp ompex.f90 -o ompex
```

- gnu compilers
  - Add the `-fopenmp` option

```
gcc -fopenmp ompex.c -o ompex
```

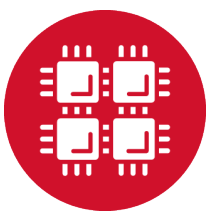
- Portland group compilers
  - Add the `-mp` option

```
pgf90 -mp ompex.f90 -o ompex
```



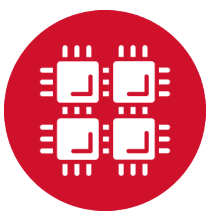
# Running an OpenMP program

- Request multiple processors through PBS
  - Example: `nodes=1:ppn=40`
- Set the `OMP_NUM_THREADS` environment variable
  - Default: Use all available cores
- For best performance run at most one thread per core
  - Otherwise too much overhead
  - Applies to typical HPC workload, exceptions exist



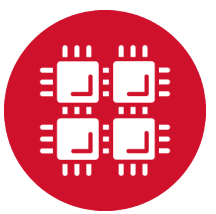
# Running an OpenMP program - Example

```
#PBS -N omploop
#PBS -j oe
#PBS -l nodes=1:ppn=40
#PBS -l walltime=1:00

cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=40
/usr/bin/time ./omploop
```



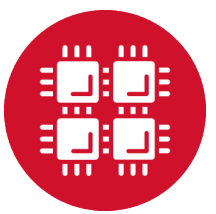
# More Information about OpenMP

- www.openmp.org
- OpenMP Application Program Interface
  - Version 4.5, November 2015
  - http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf
- OSC will host an XSEDE OpenMP workshop on November 5, 2019.
- Self-paced tutorial materials available from https://portal.xsede.org/online-training



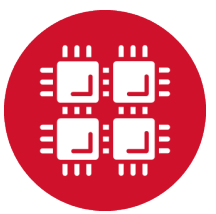
# What is MPI?

- Message Passing Interface
  - Multiple processes run on one or more nodes
  - Distributed-memory model
- A message passing library
- A run-time environment
  - *mpiexec*
- Compiler wrappers
- Supported by all major parallel machine manufacturers

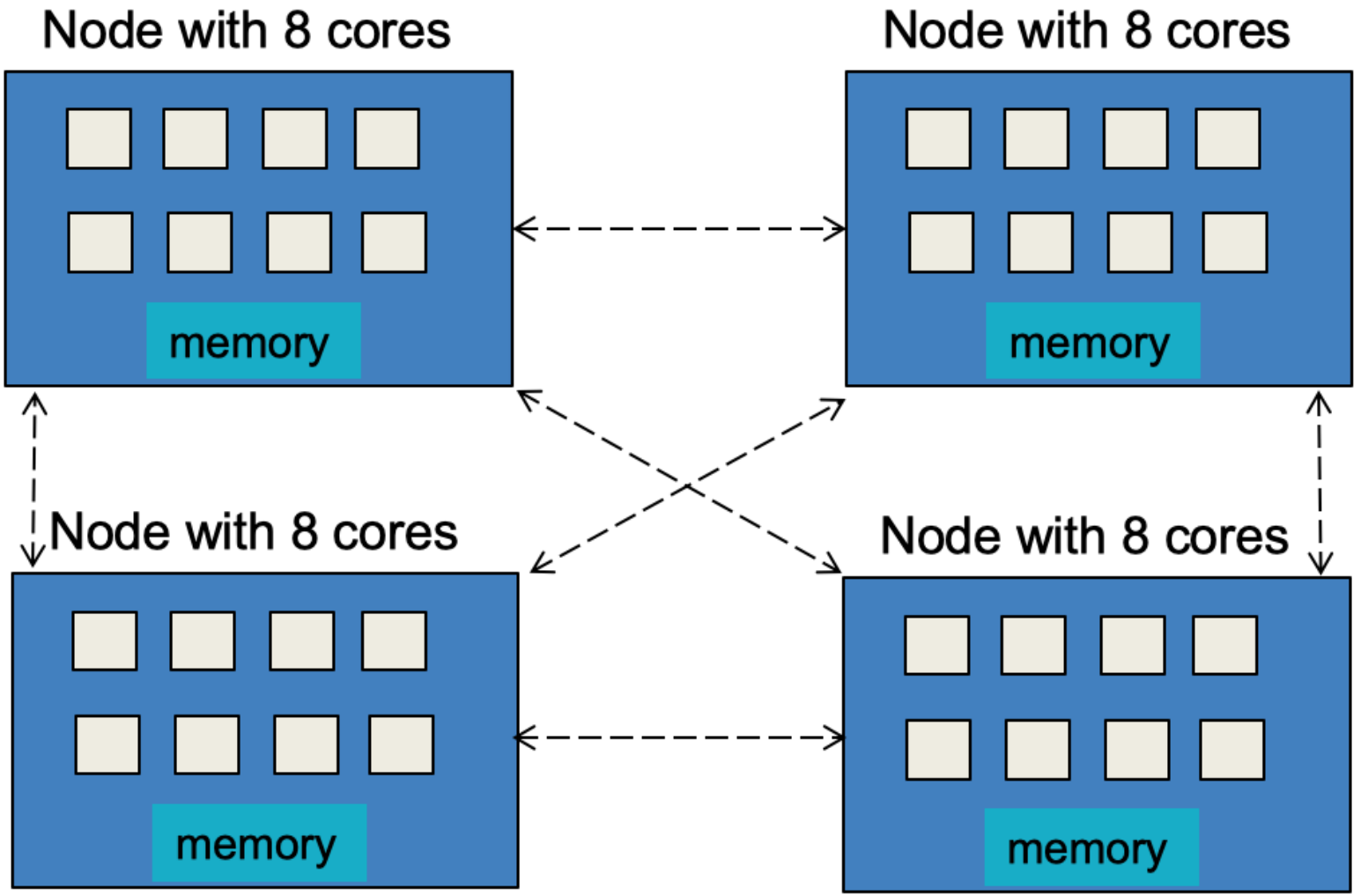


# MPI Functions

- MPI has functions for point-to-point communication (i.e. `MPI_Send`, `MPI_Recv`)
- MPI also provides a number of functions for typical collective communication patterns, including:
  - `MPI_Bcast`: broadcasts value from root process to all other processes
  - `MPI_Reduce`: reduces values on all processes to a single value on a root process
  - `MPI_Allreduce`: reduces value on all processes to a single value and distributes the result back to all processes
  - `MPI_Gather`: gathers together values from a group of processes to a root process
  - `MPI_Alltoall`: sends data from all processes to all processes

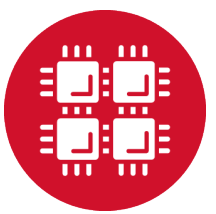


# OpenMP vs. MPI



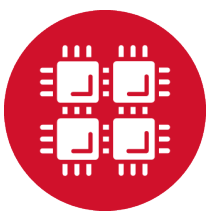
# A simple MPI program

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank,size

    MPI_Init(&argc,&argv)
    MPI_Comm_rank(MPI_COMM_WORLD,&rank)
    MPI_Comm_size(MPI_COMM_WORLD,&size)
    printf("Hello from node %d of %d\n",rank,size)
    MPI_Finalize()
    return(0)
}
```



# MPI Implementations Available at OSC

- mvapich2
  - default
- Intel MPI
  - available only with Intel compilers
- OpenMPI

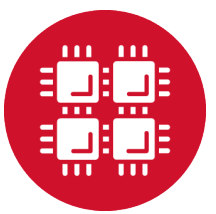


# Compiling MPI programs

- Compile with the MPI compiler wrappers
  - `mpicc`, `mpicxx`, and `mpif90`
  - Accept the same arguments as the compilers they wrap

```
mpicc -o hello hello.c
```

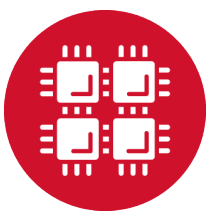
- Compiler and MPI implementation depend on modules loaded



# Running MPI programs

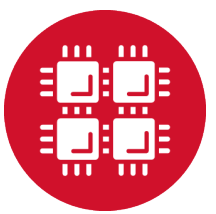
- MPI programs must run in batch only
  - Debugging runs may be done with interactive batch jobs
- ***mpiexec***
  - Automatically determines execution nodes from PBS
  - Starts the program running, 2x40=80 copies

```
#PBS -N mpi_hello
#PBS -j oe
#PBS -l nodes=2:ppn=40
#PBS -l walltime=1:00

cd $PBS_O_WORKDIR
mpiexec ./hello
```



# More Information about MPI

- www.mpi-forum.org
- MPI: A Message-Passing Interface Standard
  - Version 3.1, June 4, 2015
  - http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf
- OSC will host an XSEDE MPI workshop on September 3–4, 2019.
- Self-paced tutorial materials available from https://portal.xsede.org/online-training

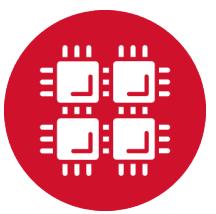


# Hands-on - MPI

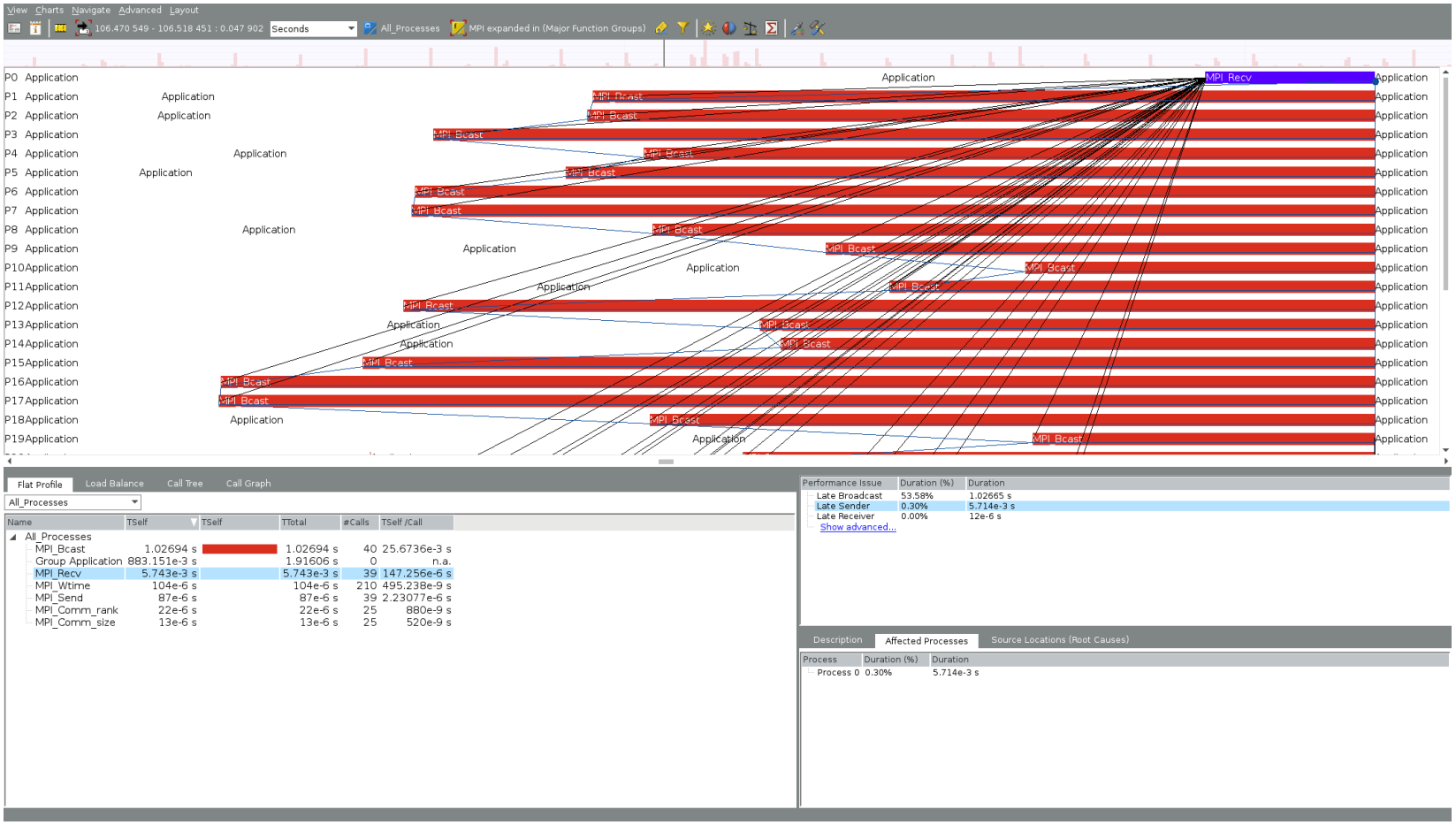
- Use ITAC to get a timeline of the run of the code.

```
module load intelmpi
mpiexec -trace -np 40 ./test_hpcg 150 150 150
traceanalyzer <stf_file>
```

- Look at the Event Timeline (under Charts).
- Do you see any communication patterns that could be replaced by a single MPI command?
- Hint: Ungroup the MPI group, zoom in on one of the MPI communications (black lines) and try to find the MPI calls in the code.



# Hands-on - MPI - Solution



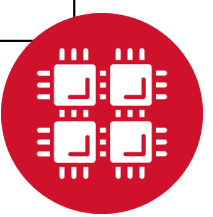
# Hands-on - MPI - Solution

Replace lines 82–89 of `ddot.cpp`:

```
MPI_Barrier(MPI_COMM_WORLD);
if(rank == 0)
  for(dst_rank=1; dst_rank<size; dst_rank++)
    MPI_Send(&global_result, 1, MPI_DOUBLE,
             dst_rank, 1, MPI_COMM_WORLD);
if(rank != 0)
  MPI_Recv(&global_result, 1, MPI_DOUBLE, 0, 1,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
MPI_Barrier(MPI_COMM_WORLD);
```

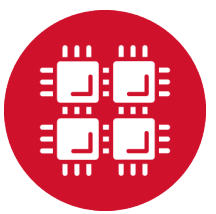
with

```
MPI_Allreduce(&local_result, &global_result,
              1, MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD);
```



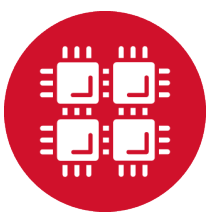
# GPU-Accelerated Computing

- GPU = Graphics Processing Unit
  - Can be used to accelerate computation
- OSC clusters have some nodes with NVIDIA GPUs
- Many-core processors
  - more cores than multi-core
- Can be programmed with CUDA
  - low level
- PGI and GNU compilers support OpenACC
  - easier than CUDA
  - similar to OpenMP



# Summary: What should you do with your code?

- Experiment with compiler optimization flags
- Profile it
- Read optimization reports
- Analyze data layout, memory access patterns
- Examine algorithms
  - Complexity
  - Availability of optimized version
- Look for potential parallelism and any inhibitors to parallelism
  - Improve vectorization



# Resources to get your questions answered

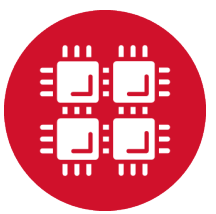
FAQs: osc.edu/resources/getting_started/supercomputing_faq
HOW TOs: osc.edu/resources/getting_started/howto

Performance Collection Guide:
osc.edu/resources/getting_started/howto/howto_collect_performance_data_for_your_program

Office Hours:
go.osu.edu/rc-osc Tuesdays 1-3 p.m. or Weekdays 4-5 at Pomerene Hall

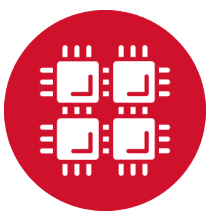
System updates:

- Read Message of the Day on login
- Follow @HPCNotices on Twitter



# Other Sources of Information

- Online manuals
  - `man ifort`
  - `man pgc++`
  - `man gcc`
- Related workshop courses
  - www.osc.edu/supercomputing/training
- Online tutorials from Cornell
  - https://cvw.cac.cornell.edu/
- oschelp@osc.edu



OH·TECH

Ohio Technology Consortium

A Division of the Ohio Department of Higher Education


info@osc.edu

osc.edu

twitter.com/osc

oh-tech.org/blog

facebook.com/ohiosupercomputercenter

linkedin.com/company/ohio-supercomputer-center