Debugging, Profiling and Tuning

Prasad Maddumage

Sr. Research Scientist, HPC Consulting

Corning Inc.

2023 Virtual Residency, June 28

WARNING!

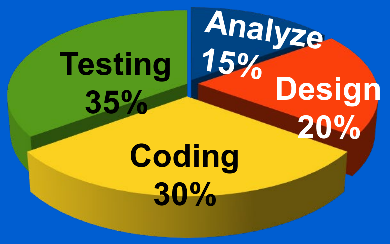
- I am NOT an expert, just someone who volunteered to talk about this topic!
- This is NOT a lecture!
- Please interrupt me during the session when you have questions / comments
- I assume you are somewhat familiar with programming under Linux
  - Parallel programming experience preferred

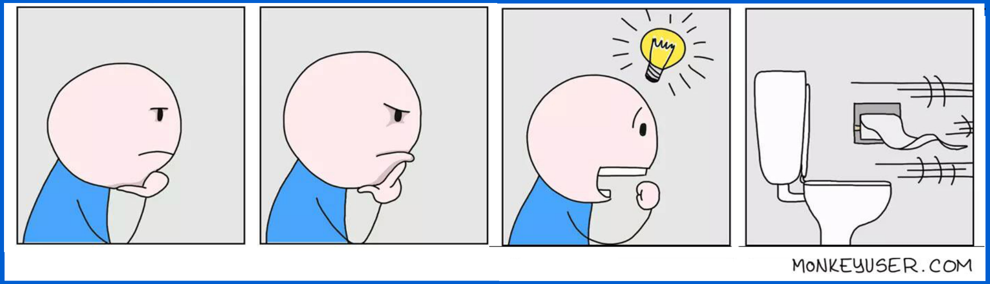
Overview

- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers
- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Debugging

- Detecting and removing of existing and potential errors (‘bugs’) in a software that can cause it to behave unexpectedly or crash. To prevent incorrect operation of a software
  - Syntax errors, segmentation faults (invalid memory access), I/O errors, ...
- Debugger : A tool that helps you debug (it doesn’t debug for you)
  - CLI (Command Line Interface) based
    - write/printf, gdb, valgrind (memory issues), …
    - Effectively pinpoint problems, works with serial/parallel codes
    - Need to remember commands, not user friendly
  - GUI (Graphical User Interface) debuggers
    - TotalView, DDT, Intel Inspector, ...
    - Powerful and user friendly
  - **ChatGPT**





- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Using Compiler Flags

- Compilers can help debugging without a debugger
- Almost all debuggers require the code to be compiled with -g flag
- There are other compiler flags that can identify potential issues
  - During compile time
  - During **runtime**
- May not be as reliable as using a debugger
  - Vendor dependent
  - Version dependent

Using Compiler Flags – Compile time

- -Wall : (gnu, Intel - C/C++), -warn all (Intel - Fortran)
  - Detect uninitialized variables
  - Find unused parameters (variables, functions, labels, ...)
  - Implicit function declaration in C /C++ (declare before use a function)
- -Wextra : (gnu) enables extra warning flags in addition to -Wall
  - -Wall -Wextra : detects unused but set variables
- -Werror : (gnu) compilation stops at warnings
  - Treat warnings as errors
- -Wuninitialized : (gnu) Warn at compiling time if a variable is used without first being initialized
  - -check-uninit , -check unint (Intel) Runtime checking of undefined variables

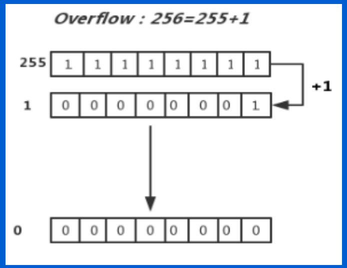
Using Compiler Flags – Runtime

- -g : embed debug information to the binary (parts of the source itself)
- -fcheck=bounds : (gfortran) check array indices are within the declared range
  - -check bounds / -CB (Intel)
- -fcheck=all : (gfortran) checks for invalid modification of loop iteration variables, memory allocation, bounds, etc

Using Compiler Flags – Runtime

- -ftrapv : (gnu C/C++)detects integer overflow and abort the program

Odometer analogy



- -ffpe-trap=invalid,zero,... : (gfortran, gcc by default) detects and aborts the program
  - invalid: invalid floating point operation √-1
  - zero: division by zero
  - overflow: overflow in a floating point operation
  - underflow: underflow in a floating point operation etc

```
>>> 5e-324 - 1e-324
5e-324
```

Using Compiler Flags

```
$ gfortran -o oflow oflow.f90
$ ./oflow
  2147483647   1.7976931348623157E+308 -2147483648  Infinity
```

```
$ gfortran -ffpe-trap=overflow -o oflow oflow.f90
$ ./oflow

Program received signal SIGABRT: Process abort signal.
...
Aborted (core dumped)
```

```fortran
program main
  integer :: n, i
  real*8 :: x

  x = HUGE(1.d0)
  n = HUGE(1)
  print *, n, x, n + 1, x + x

end program main
```

```
$ gcc -o oflow_c oflow.c
$ ./oflow_c
-2147483648
```

```
$ gcc -ftrapv -o oflow_c oflow.c
$ ./oflow_c
Aborted (core dumped)
```

```c
#include <limits.h>
#include <stdio.h>
int main(void){
        int i = INT_MAX;
        return printf("%d\n",i + 1);
}
```

Debug support from MPI Compilers

- Setting certain environment variables enable MPI to output information helpful for debugging applications during runtime
- Open MPI
  - mpi_param_check : If true, checks MPI function values for illegal values such as NULL
  - mpi_abort_delay : If nonzero, prints hostname and process ID of the process invoked MPI_ABORT
- MVAPICH2
  - MV2_DEBUG_SHOW_BACKTRACE : Show backtrace when a process fails on errors like *Segmentation fault, Bus error, Illegal Instruction, Abort* etc
- -g flag is not needed for these to work

- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Debugger Basics

- Debugger: Program that helps you run a software in a controlled way to help you find and fix bugs
- Breakpoint: Pauses execution of processes
  - Unconditional: always pause
  - Conditional: pauses only if a condition is satisfied
  - Evaluation: pause and execute a code fragment when reached
- Watchpoint: monitors a variable and pauses execution when its value changes
- Backtrace: List of function calls currently active in a process
- Frame: (stack frame) Contains arguments given to a function, its local variables, and the address at which the function is executing
  - There is always one or more frame(s) associated with a running program

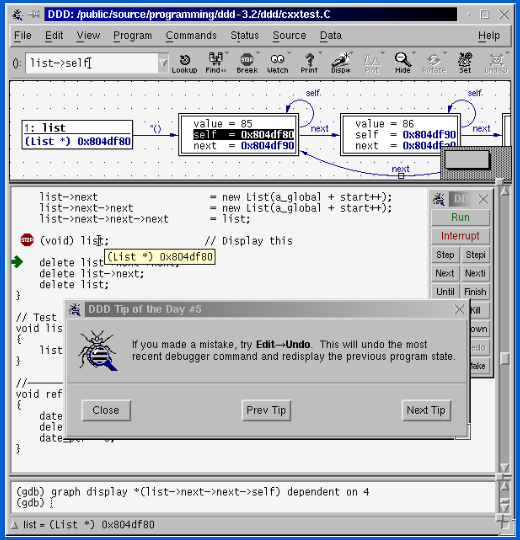
- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

gdb

- gdb is the GNU project debugger
- Supports C, C++, Fortran, Assembly, Go, OpenCL, etc
- Capabilities
  - Start a program
  - Make a program stop on specified conditions
  - Examine the program once its stopped
  - Change variable values of a program while its running to examine the effect (on bugs)

- Terminal based (text only) debugger
  - The GUI front end of gdb is DDD (Data Display Debugger)
  - Latest version of DDD was released on 05/10/2023. However, the previous release was in 2009!
    - Not worked as intended in most new systems until last month



Serial debugging with gdb

```
$ gfortran trap.f90 -g -o trap
$ gdb trap
```

```
(gdb) break 13
Breakpoint 1 at 0x11de: file trap.f90, line 13.
(gdb) break 15
Breakpoint 2 at 0x1234: file trap.f90, line 15.
(gdb) run
13                area = 0.5 * (sin(a) + sin(b))
(gdb) print a
$1 = 0
(gdb) p area
$2 = -209808
(gdb) next
14                DO i = 1, n-1
(gdb) p area
$3 = -4.37113883e-08
(gdb) continue
Continuing.
```

```
1  ! Caculate area under sine curve
2
3  PROGRAM trapz
4    IMPLICIT none
5    INTRINSIC :: sin
6    REAL :: a, b, h, area
7    INTEGER :: i, n
8    n = 100
9    a = 0
10   b = 4.0 * atan(1.0)
11
12   h = (b - a) / n
13   area = 0.5 * (sin(a) + sin(b))
14   DO i = 1, n-1
15      area = area + sin(a + i*h)
16   END DO
17   area = h * area
18   PRINT *, "Area = ", area
19 END PROGRAM trapz
```

```
Breakpoint 2, trapz () at trap.f90:15
15                area = area + sin(a + i*h)
(gdb) p area
$4 = -4.37113883e-08
(gdb) continue
Continuing.

Breakpoint 2, trapz () at trap.f90:15
15                area = area + sin(a + i*h)
(gdb) p area
$5 = 0.0314107165
(gdb) clear
Deleted breakpoint 2
(gdb) c
Continuing.
 Area =    1.99983561
[Inferior 1 (process 13270) exited normally]
(gdb) q
```

Serial debugging with gdb

```
(gdb) run
Starting program:

Program received signal SIGSEGV, Segmentation fault.
0x00005555555552bc in test () at test.f90:10
10                    x(i) = i
(gdb) backtrace
#0  0x00005555555552bc in test () at test.f90:10
(gdb) frame 0
#0  0x00005555555552bc in test () at test.f90:10
10                    x(i) = i
(gdb) print i
$1 = 30141
(gdb) print x
$2 = (1, 2, 3, 4, 5)
```

```fortran
program test
  implicit none

  integer :: i
  integer, allocatable :: x(:)

  allocate(x(5))

  do i = 1, 100000
     x(i) = i
  end do

end program test
```

Useful gdb Commands

- break *location* / thread *thread#* / if *condition*
- clear *function/breakpoint …* : Remove all or selected breakpoints
- step *count* : Pause the program after executing a *count* number of source line(s). Stops at each line of any functions called within a line
- next *count* : Same as step but does not stop when inside a function
- skip *function / file* : Prevent gdb from running a function or source file

Useful gdb Commands

- reverse-step : Run the program backward until it reaches the start of a different source line
- list : Print lines (at line #, function, before/after last line, …)
- set var *variable=value* : Change a variable value during the debugging session
- info locals : Display the local variable values in the current frame

Core Dump Analysis

- A core dump is a file containing part of the application's memory when the process terminates unexpectedly
  - Core dumps may be produced on-demand (eg: by a debugger) or automatically upon termination (crash)
- A core file can be opened and examined using gdb
  - ```
    $ gdb -e program_name -c core_dump_name
    ```
    OR
    ```
    $ gdb program_name
    (gdb) core core_dump_name
    ```
  - Use bt / frame / list / info locals / print etc to pin point the cause
- gcore can create a manual core dump of any process

```
$ gcore -o core_file_name process_id
```
```

```
$ gdb oflow /var/lib/apport/coredump/core._oflow.1000.b92dc8f9-2041-46b6-a112-455c25153497.53671.5260506
GNU gdb (Ubuntu 13.1-2ubuntu2) 13.1
...
Enable debuginfod for this session? (y or [n]) n
Debuginfod has been disabled.
...
Core was generated by `./oflow'.
Program terminated with signal SIGABRT, Aborted.
#0  __pthread_kill_implementation (no_tid=0, signo=6, threadid=<optimized out>) at ./nptl/pthread_kill.c:44
...
(gdb) bt
#0  __pthread_kill_implementation (no_tid=0, signo=6, threadid=<optimized out>) at ./nptl/pthread_kill.c:44
#1  __pthread_kill_internal (signo=6, threadid=<optimized out>) at ./nptl/pthread_kill.c:78
...
#5  0x000055b66343c1e1 in __addvsi3 ()
#6  0x000055b66343c189 in main () at oflow.c:5
(gdb) frame 6
#6  0x000055b66343c189 in main () at oflow.c:5
5                       return printf("%d\n",i + 1);
(gdb) q
```

- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Parallel (MPI) Debugging

- High Performance Computing (HPC) involves using more than a single node to solve a problem
  - A common way to do this is using MPI (Message Passing Interface)
  - MPI programs often need to be debugged in a cluster environment
- Using gdb
  - Attach gdb to each process of an *already running* job
  - Interactive job with all or some ranks run under gdb (interactive debugging)
  - Submit a batch job so that all or some ranks run under gdb (non-interactive)
- TotalView and DDT
  - GUI debuggers are user friendly and offer convenience
  - Expensive!

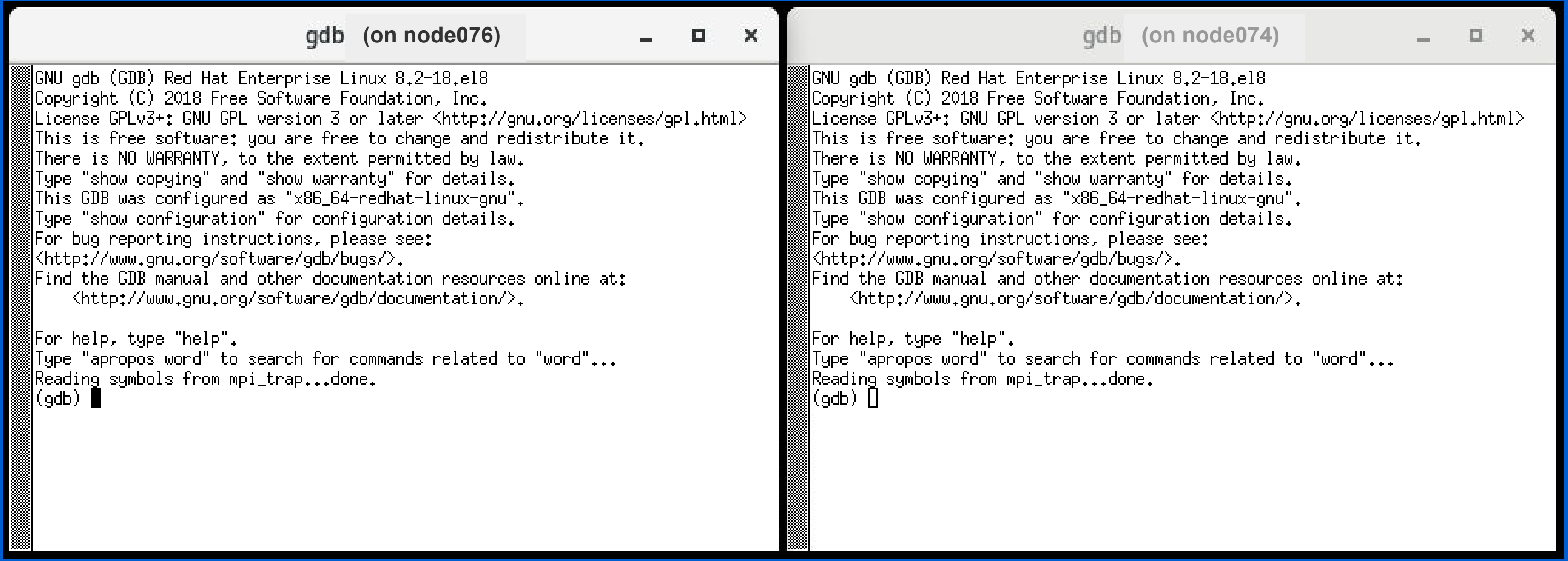
Parallel Debugging with gdb

- Attach to already running job
  - `$ gdb program_name process_id` OR
    ```
    $ gdb program_name
    $ gdb> attach process_id
    ```
  - Need to login (ssh) to the compute node and find the process id first
    - Use top (-u to display processes for a given user)
  - After attaching, any gdb command can be used
  - Interactive debugging
  - Can only debug one (misbehaving) process at a time
- gdbserver is used to remotely debug applications
  - Command line interface (CLI) only, no GUI
  - This is left as an advanced topic

Interactive Parallel Debugging with gdb

- Requires X11 forwarding support from scheduler
  - If set up, use --x11 flag with SLURM or -X with PBS when making a reservation

```
$ salloc -n 2 --x11
$ export MPIGDB="xterm -e gdb –args"
$ mpirun $MPIGDB mpi_trap
```



Interactive parallel debugging with gdb

gdb (on node076)

```
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-18.el8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mpi_trap...done.
(gdb) br 48
Breakpoint 1 at 0x400fb7: file mpi_trap.c, line 48.
(gdb)
```

gdb (on node074)

```
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-18.el8
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mpi_trap...done.
(gdb) br MPI_Comm_rank
Breakpoint 1 at 0x400e60
(gdb)
```

```
47    /* Get my process rank */
48    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Interactive parallel debugging with gdb

gdb (on node076)

```
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mpi_trap...done.
(gdb) br 48
Breakpoint 1 at 0x400fb7: file mpi_trap.c, line 48.
(gdb) r
Starting program: /home/maddumagph/software/test/mpi_trap
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x1555509eb700 (LWP 1791467)]
[New Thread 0x1555507ea700 (LWP 1791468)]

Thread 1 "mpi_trap" hit Breakpoint 1, main (argc=1, argv=0x7fffffffa638)
    at mpi_trap.c:48
48          MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-189.1.el8.x86
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 110
(gdb) 
```

gdb (on node074)

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mpi_trap...done.
(gdb) br MPI_Comm_rank
Breakpoint 1 at 0x400e60
(gdb) watch local_int
No symbol "local_int" in current context.
(gdb) r
Starting program: /home/maddumagph/software/test/mpi_trap
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x1555509eb700 (LWP 2074455)]
[New Thread 0x1555507ea700 (LWP 2074456)]

Thread 1 "mpi_trap" hit Breakpoint 1, 0x0000155554c86be0 in PMPI_Comm_rank ()
   from /cm/shared/userapps/opensource-22/milan/__spack_path_placeholder__/__spa
ck_path_placeholder__/__spack_path_placeholder__/__spac/linux-rocky8-zen3/gcc-11
.2.0/openmpi-4.1.1-hdwjt74h3iqnjvpdhvnwzoq273bo4ib3/lib/libmpi.so.40
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-189.1.el8.x86
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 0
(gdb) 
```

```
47      /* Get my process rank */
48      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Interactive parallel debugging with gdb

gdb (on node076)

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mpi_trap...done.
(gdb) br 48
Breakpoint 1 at 0x400fb7: file mpi_trap.c, line 48.
(gdb) r
Starting program: /home/maddumagph/software/test/mpi_trap
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x1555509eb700 (LWP 1791467)]
[New Thread 0x1555507ea700 (LWP 1791468)]

Thread 1 "mpi_trap" hit Breakpoint 1, main (argc=1, argv=0x7fffffffa638)
    at mpi_trap.c:48
48          MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-189.1.el8.x86
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 110
(gdb) n
51          MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
(gdb) p my_rank
$2 = 1
(gdb)
```

gdb (on node074)

```
No symbol "local_int" in current context.
(gdb) r
Starting program: /home/maddumagph/software/test/mpi_trap
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
[New Thread 0x1555509eb700 (LWP 2074455)]
[New Thread 0x1555507ea700 (LWP 2074456)]

Thread 1 "mpi_trap" hit Breakpoint 1, 0x0000155554c86be0 in PMPI_Comm_rank ()
   from /cm/shared/userapps/opensource-22/milan/__spack_path_placeholder__/__spa
ck_path_placeholder__/__spack_path_placeholder__/__spac/linux-rocky8-zen3/gcc-11
.2.0/openmpi-4.1.1-hdwjt74h3iqnjvpdhvnwzoq273bo4ib3/lib/libmpi.so.40
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-189.1.el8.x86
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 0
(gdb) n
Single stepping until exit from function PMPI_Comm_rank,
which has no line number information.
main (argc=1, argv=0x7fffffffa728) at mpi_trap.c:51
51          MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
(gdb) p my_rank
$2 = 0
(gdb)
```

```
47    /* Get my process rank */
48    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
```

Interactive parallel debugging with gdb

gdb (on node076)

```
Thread 1 "mpi_trap" hit Breakpoint 1, main (argc=1, argv=0x7fffffffa638)
    at mpi_trap.c:48
48          MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-189.1.el8.x86
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 110
(gdb) n
51          MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
(gdb) p my_rank
$2 = 1
(gdb) watch local_int
Hardware watchpoint 2: local_int
(gdb) c
Continuing.

Thread 1 "mpi_trap" hit Hardware watchpoint 2: local_int

Old value = 0
New value = 7.8749999999985425
main (argc=1, argv=0x7fffffffa638) at mpi_trap.c:64
64          if (my_rank != 0) {
(gdb) 
```

gdb (on node074)

```
.2.0/openmpi-4.1.1-hdwjt74h3iqnjvpdhvnwzoq273bo4ib3/lib/libmpi.so.40
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-189.1.el8.x86
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 0
(gdb) n
Single stepping until exit from function PMPI_Comm_rank,
which has no line number information.
main (argc=1, argv=0x7fffffffa728) at mpi_trap.c:51
51          MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
(gdb) p my_rank
$2 = 0
(gdb) watch local_int
Hardware watchpoint 2: local_int
(gdb) c
Continuing.

Thread 1 "mpi_trap" hit Hardware watchpoint 2: local_int

Old value = 0
New value = 1.1249999999998934
main (argc=1, argv=0x7fffffffa728) at mpi_trap.c:64
64          if (my_rank != 0) {
(gdb) 
```

```
   /* Length of each process' interval of
    * integration = local_n * h.  So my interval
    * starts at: */
   local_a = a + my_rank * local_n * h;
   local_b = local_a + local_n * h;
   local_int = Trap(local_a, local_b, local_n, h);
```

Interactive parallel debugging with gdb

gdb (on node076)

```
Thread 1 "mpi_trap" hit Breakpoint 1, main (argc=1, argv=0x7fffffffa638)
    at mpi_trap.c:48
48          MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-189.1.el8.x86
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 110
(gdb) n
51          MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
(gdb) p my_rank
$2 = 1
(gdb) watch local_int
Hardware watchpoint 2: local_int
(gdb) c
Continuing.

Thread 1 "mpi_trap" hit Hardware watchpoint 2: local_int

Old value = 0
New value = 7.8749999999985425
main (argc=1, argv=0x7fffffffa638) at mpi_trap.c:64
64          if (my_rank != 0) {
(gdb)
```

gdb (on node074)

```
_64 libnl3-3.5.0-1.el8.x86_64
(gdb) p my_rank
$1 = 0
(gdb) n
Single stepping until exit from function PMPI_Comm_rank,
which has no line number information.
main (argc=1, argv=0x7fffffffa728) at mpi_trap.c:51
51          MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
(gdb) p my_rank
$2 = 0
(gdb) watch local_int
Hardware watchpoint 2: local_int
(gdb) c
Continuing.

Thread 1 "mpi_trap" hit Hardware watchpoint 2: local_int

Old value = 0
New value = 1.1249999999998934
main (argc=1, argv=0x7fffffffa728) at mpi_trap.c:64
64          if (my_rank != 0) {
(gdb) c
Continuing.
```

- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers
- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Non-interactive Parallel Debugging with gdb

- No special scheduler setup is necessary
- Need to wait until end of the run to find results

```
mpirun -np 4 gdb --batch -q -x commands.txt mpi_trap
```

```
break Trap
run
print my_rank
info locals
continue
```

```
double Trap(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {

   double estimate, x;
   int i;

   estimate = (f(left_endpt) + f(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(x);
   }
   estimate = estimate*base_len;

   return estimate;
```

Non-interactive Parallel Debugging with gdb

```
Breakpoint 1 at 0x40116b: file mpi_trap.c, line 111.
Breakpoint 1 at 0x40116b: file mpi_trap.c, line 111.
Breakpoint 1 at 0x40116b: file mpi_trap.c, line 111.
Breakpoint 1 at 0x40116b: file mpi_trap.c, line 111.
…
Thread 1 "mpi_trap" hit Breakpoint 1, Trap (left_endpt=0.74999999999999989,  right_endpt=1.4999999999999998, trap_count=25000000, base_len=2.9999999999999997e-08) at mpi_trap.c:111
Thread 1 "mpi_trap" hit Breakpoint 1, Trap (left_endpt=2.25, right_endpt=3, trap_count=25000000, base_len=2.9999999999999997e-08) at mpi_trap.c:111
…
111       estimate = (f(left_endpt) + f(right_endpt))/2.0;
111       estimate = (f(left_endpt) + f(right_endpt))/2.0;
…
$1 = 2
estimate = 0
x = 2.0740954862918865e-317
i = 0
$1 = 3
estimate = 0
x = 2.0740954862918865e-317
i = 0
…
```

```
break Trap
run
print my_rank
info locals
continue
```

```
102 double Trap(
103         double left_endpt  /* in */,
104         double right_endpt /* in */,
105         int    trap_count  /* in */,
106         double base_len    /* in */) {
107
108     double estimate, x;
109     int i;
110
111     estimate = (f(left_endpt) + f(right_endpt))/2.0;
112     for (i = 1; i <= trap_count-1; i++) {
113         x = left_endpt + i*base_len;
114         estimate += f(x);
115     }
116     estimate = estimate*base_len;
117
118     return estimate;
```

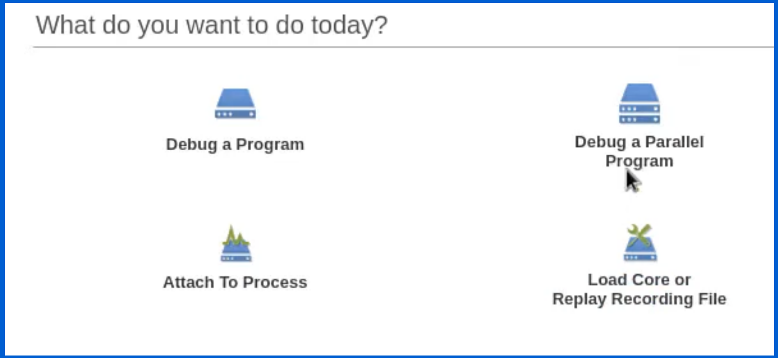
- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Totalview

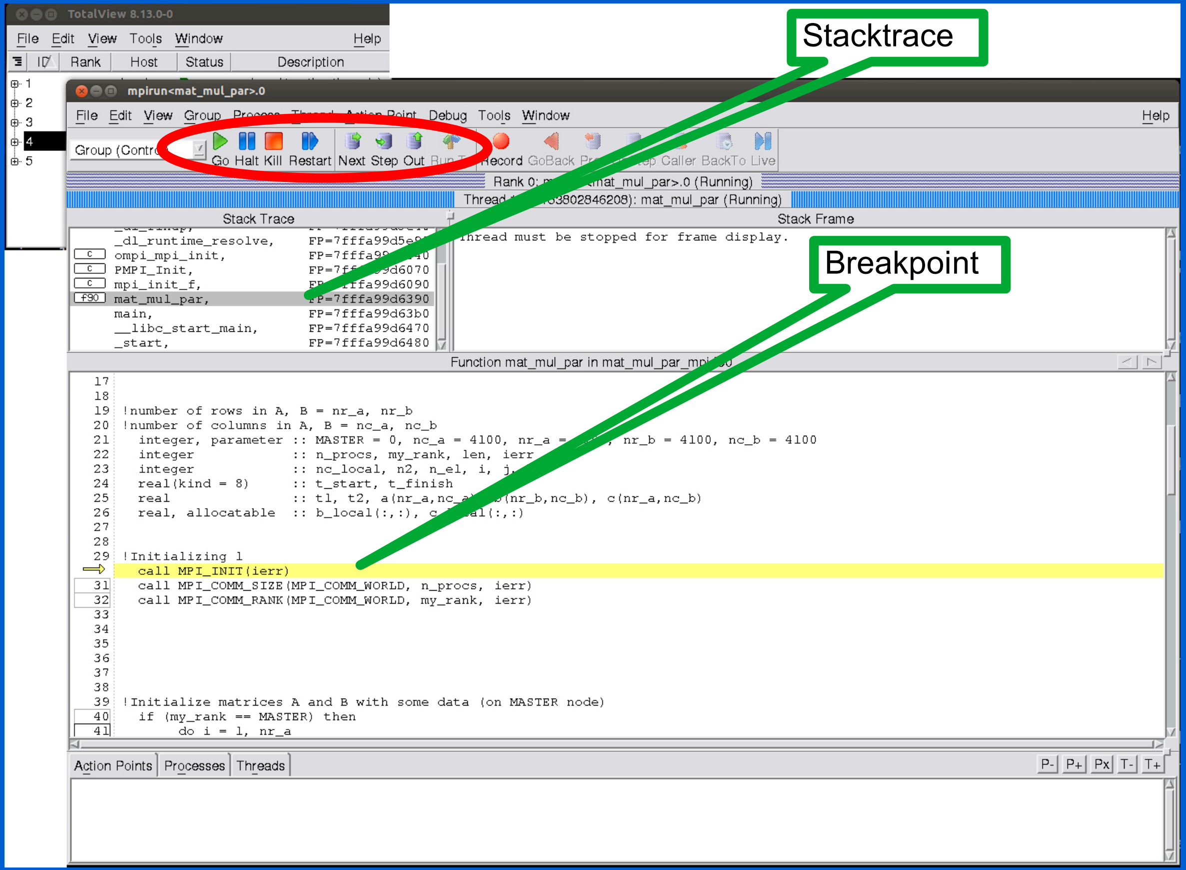
- Debugging and analyzing serial and parallel programs
- Both a GUI and command line interface
- Memory debugging features
- Graphical visualization of array data
- Comprehensive built-in help system
- Recording and replaying running programs
- Sessions Manager for managing and loading debugging sessions

```
$ totalview -args mpirun -np number_of_processes program_name
```



TotalView

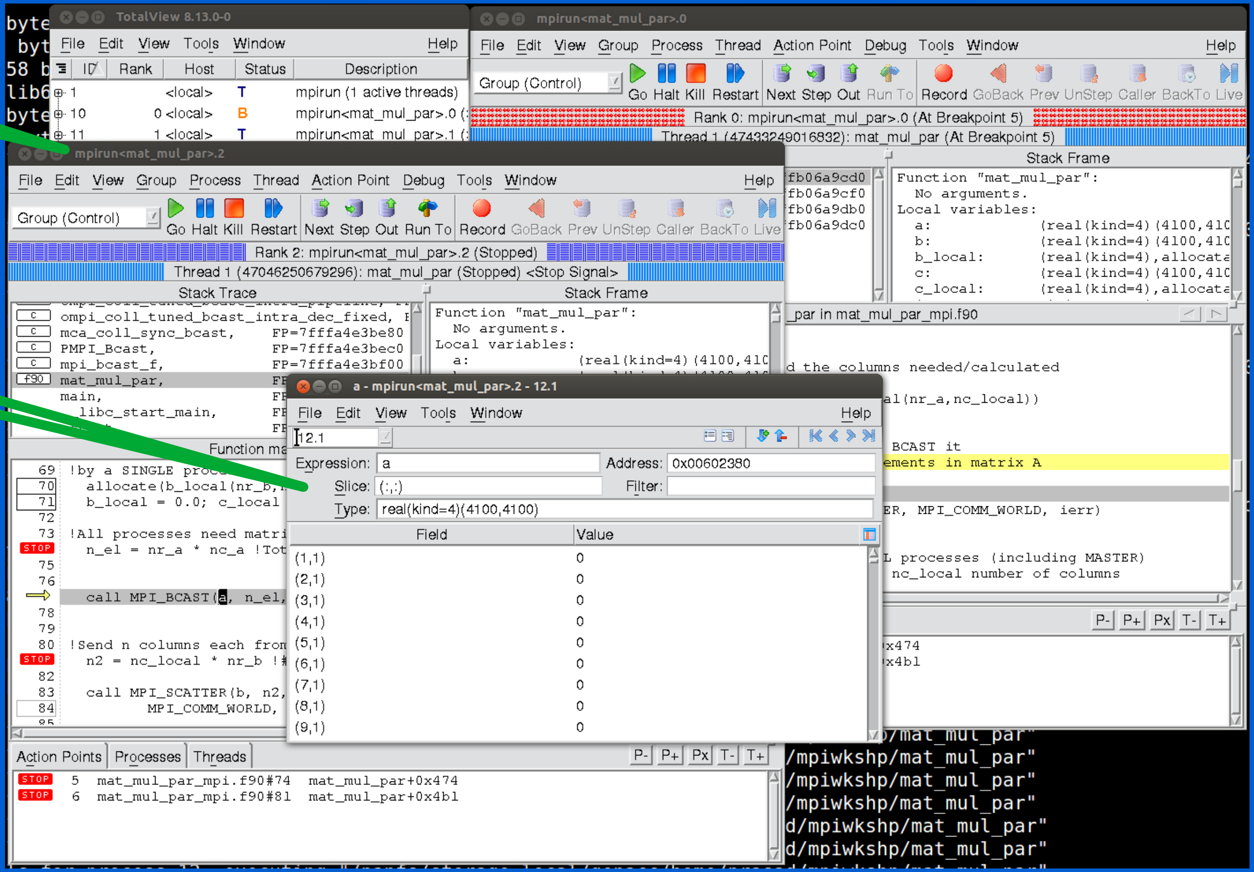
- Process barrier: point to synchronize all processes or threads
- Able to check variable values in different ranks without logging in to that rank
- Batch (non-interactive) debugging using tvscript
- Debugging on a remote host
  - Connect to TotalView server running on a remote system
- CUDA debugger
- Reverse debugging
  - ReplayEngine records all program's activities to be reviewed later



TotalView

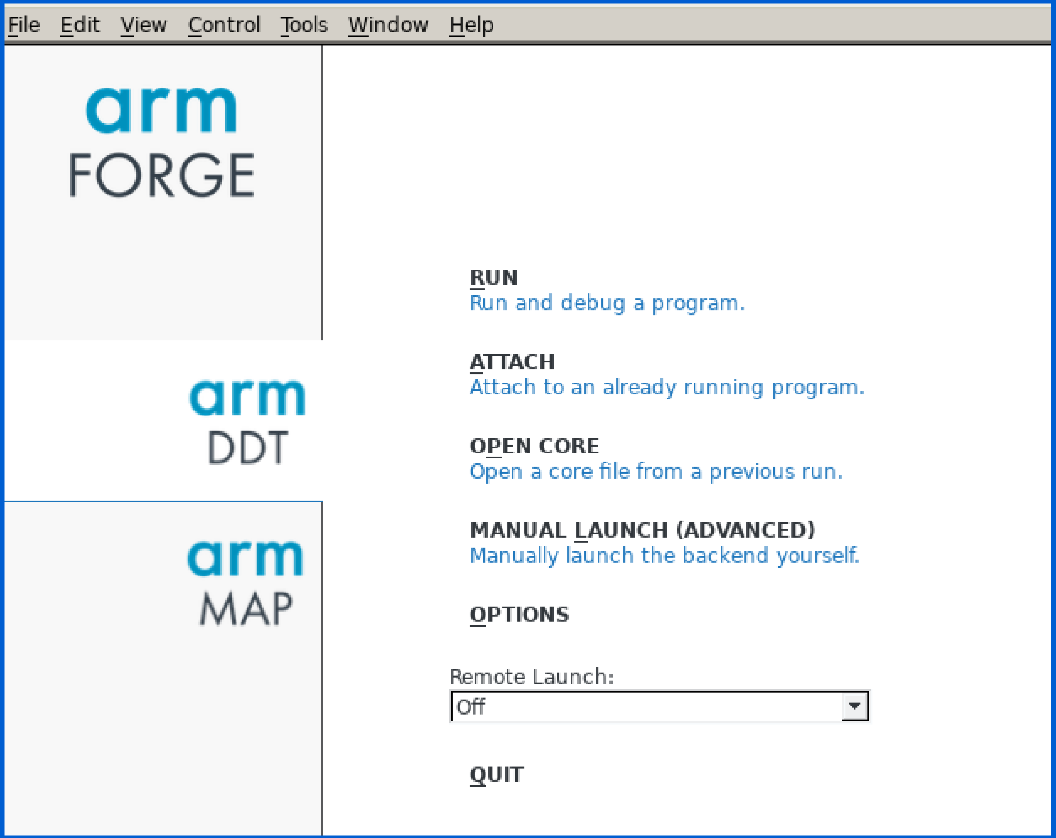
What's happening on each rank

Visualize multidimensional arrays



DDT

- CLI and GUI support
- Interactive and batch debugging
- Attach to an already running program
- Open core dump files
- Memory debugging
- Remote debugging
- CUDA debugging
- Python debugging



- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

CUDA Debugging with gdb

- CUDA-GDB
  - NVIDIA tool for debugging CUDA applications on Linux

```
$ nvcc -g -G foo.cu -o foo

$ pgfortran -g -Mcuda=nordc foo.cuf -o foo
```

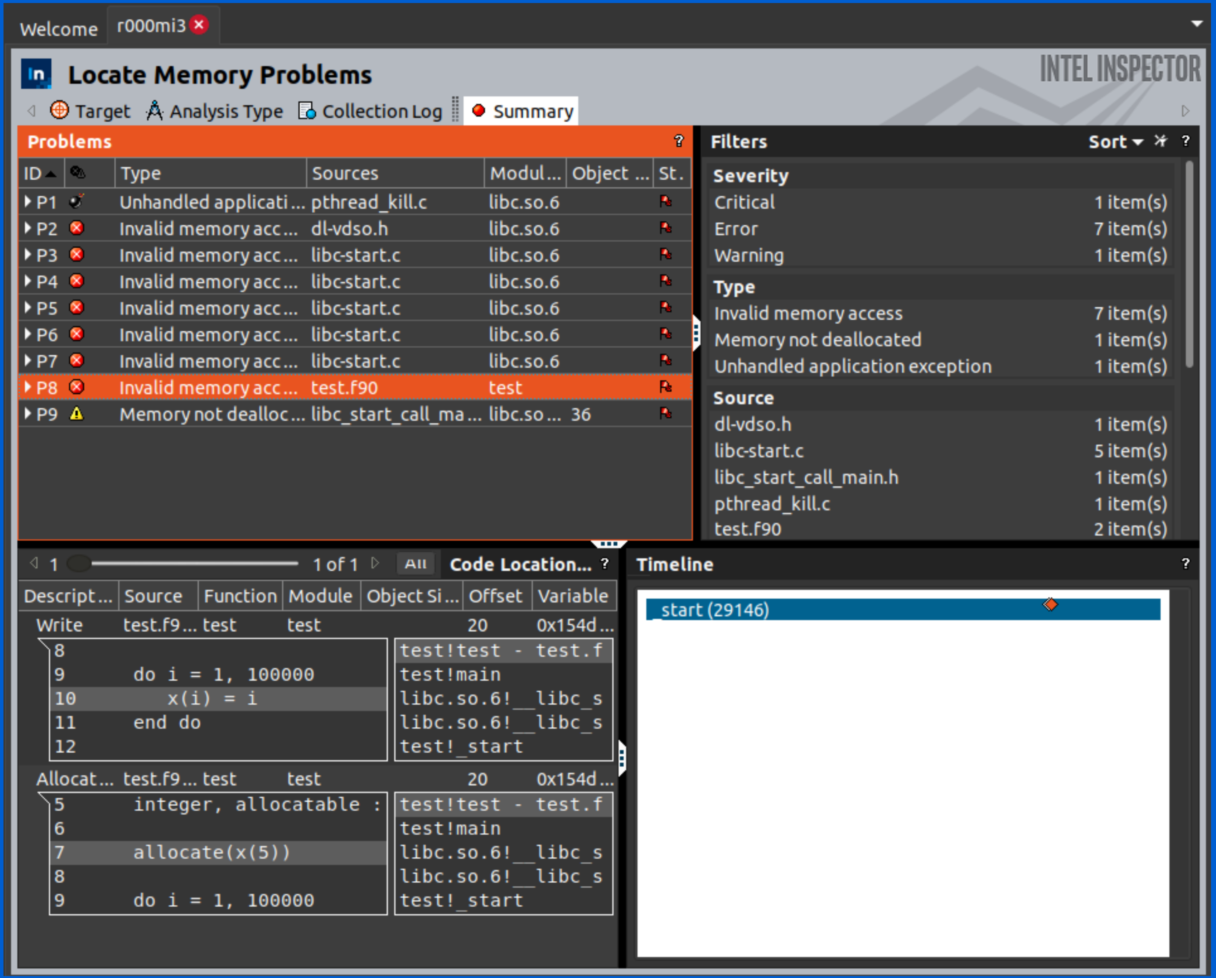
  - Can debug both GPU and CPU code simultaneously
  - CUDA commands in addition to gdb commands
  - MPI is supported
  - Breakpoints supported on GPU and both breakpoints and watchpoints on CPU
    - Breakpoints can be set by symbolically (function name), line number, memory address, conditional, and kernel entry
  - Can switch between threads and inspect program execution
  - Stepping works by advancing all active threads in the warp of focus
  - Remote debugging is possible
  - GPU core dump is supported

Intel Inspector

- Detect memory leaks
  - Locate memory problems
- Locate deadlocks and data races
- GUI (inspxe-gui) and cli (inspxe-cl) versions
- Works with serial and mpi applications

```
srun -n8 inspxe-cl -collect mi3
                  -r my_results my_mpi_app
```

- CLI version results can be visualized with GUI later
- NOT a complete debugger
- Free!



- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Language Specific Debuggers

- Python: pdb
  - Interactive debugging: use python -m pdb source.py
  - Batch/ interactive debugging: use breakpoint() or pdb.set_trace()in source code or python prompt
    - Need to import pdb for pdb.set_trace()
  - Interactive source debugger
  - Supports breakpoints and single stepping at the source line level
  - Inspection of stack frames, source code listing
- R
  - In RStudio
    - Set breakpoints in RStudio or put browser() at the line you want to break
    - This causes R to enter the debug mode
      - Can check current variable stack, traceback the execution, and more

Python Debugging

```
$ python3 -m pdb convert.py
> /home/prasad/Downloads/convert.py(1)<module>()
-> temp = input("Temperature : (e.g., 45F): ")
(Pdb) n
Temperature : (e.g., 45F): 75F
> /home/prasad/Downloads/convert.py(2)<module>()
-> degree = int(temp[:-1])
(Pdb) p degree
*** NameError: name 'degree' is not defined
(Pdb) n
> /home/prasad/Downloads/convert.py(3)<module>()
-> i_convention = temp[-1]
(Pdb) p degree
75
(Pdb) b 10
Breakpoint 1 at /home/prasad/Downloads/convert.py:10
(Pdb) l
  3         i_convention = temp[-1]
  4
  5         if i_convention.upper() == "C":
  6           result = int(round((9 * degree) / 5 + 32))
  7           o_convention = "Fahrenheit"
  8  ->     elif i_convention.upper() == "F":
  9           result = int(round((degree - 32) * 5 / 9))
 10 B         o_convention = "Celsius"
 11         else:
 12           print("Input proper convention.")
 13           quit()
```

```python
temp = input("Temperature : (e.g., 45F): "
degree = int(temp[:-1])
i_convention = temp[-1]

if i_convention.upper() == "C":
  result = int(round((9 * degree) / 5 + 32))
  o_convention = "Fahrenheit"
elif i_convention.upper() == "F":
  result = int(round((degree - 32) * 5 / 9))
  o_convention = "Celsius"
else:
  print("Input proper convention.")
  quit()
print("Temperature in ", o_convention, " is ", result)
```

r(eturn)	Continue until current function return
c(ontinue)	Continue until next breakpoint
j(ump) line_no	Next line to be executed (useful for breaking out of loops)
w(here)	Print the current position and stack trace
a(rgs)	Print args of the current function
q(uit)	Quit pdb

R debugging

RStudio breakpoint
Click left of line number
OR
Press Shift+F9 at the line

```r
best <- 0
for (x in 100:999) {
  for (y in x:999) {
●   candidate <- x * y
    if (candidate > best && palindrome(candidate)) {
      best <- candidate
    }
  }
}
```

Variable values, stacktrace, etc are accessible through Rstudio once program pauses


```r
> rescale <-function(x) {
+ rng <- range(x)
+ browser()
+ (x - rng[1]) / (rng[2] - rng[1])
}
> rescale(c(0,5,10))
Called from: rescale(c(0, 5, 10))
Browse[1]> rng
[1]  0 10
Browse[1]> x
[1]  0  5 10
Browse[1]> s
debug at #4: (x - rng[1])/(rng[2] - rng[1])
Browse[2]> c
[1] 0.0 0.5 1.0
```

```r
g <- function(b) {
  browser()
  h(b)
}
```

breakpoint

```r
g <- function(b) {
  if (b < 0) {
    browser()
  }
  h(b)
}
```

conditional breakpoint

where	Print stack trace of all active function calls
c(ont)	Exit browser, execute the next statement
f	Finish execution of current loop or function
n	Evaluate next line, step over any function calls
s	Evaluate next line, step into any function calls
Q	Exit browser and current evaluation and return to the top-level prompt

- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Profiling and Tuning

- HPC emphasizes on performance of software
  - Being bug-free is not enough
  - Should be able to get maximum performance from the hardware
- Software can be *tuned* to increase efficiency
  - Different compilers, compiler flags (-O2, -O3 etc)
  - Better algorithms
  - Using optimized libraries
- *Profiling* helps find which part(s) a program should be tuned
  - Software profiling: *Dynamic code analysis where a program's behavior is investigated using the data collected during program execution*
    - CPU/memory utilization, frequency of function calls, I/O, MPI library usage, hardware counters, etc.
  - Identify bottlenecks
- Profilers
  - gprof, TAU (Tuning and Analysis Utilities), Intel tools

- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries


No, it's still not right, you've got a memory leak somewhere
Humm...
How about now? I recompiled with different options
Uh, yes, that seems to fix it
What do you use to monitor that? It works really well!
I don't use an app...
The only profiling tool I need is the fan of my laptop!
CommitStrip.com

GNU Profiler - gprof

- Terminal based profiler
- Already exist in most linux distributions
- Produces flat profile and a call graph
  - Flat profile: A breakdown of time spent on each function call
  - Call graph: In what order each subroutine / function was called
- Can profile serial as well as parallel applications

```
$ gfortran thermal.f -pg -o thermal
$ ./thermal
$ gprof thermal
```

```
$ export GMON_OUT_PREFIX='gmon.out-'
$ mpicc thermal_mpi.f -pg -o thermal_mpi
$ mpirun thermal_mpi
$ gprof -s thermal_mpi gmon.out-*
```

Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
44.10	9.04	9.04	86150709	0.00	0.00	energy_
34.08	16.04	6.99	13893157	0.00	0.00	update_
19.75	20.09	4.05	771898	0.00	0.00	sumit_

Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.05% of 20.51 seconds

Index	% time	self	children	called	name
		0.03	20.45	1/1	main [2]
[1]	99.9	0.03	20.45	1	MAIN__ [1]
		0.13	20.08	771780/771780	move_ [3]
		0.16	0.00	771897/771897	locate_ [7]

TAU (Tuning and Analysis Utilities)

- Integrated performance toolkit
  - Instrumentation, measurement, analysis, visualization
  - Performance data management and data mining
  - 20+ year project actively developed by Univ. of Oregon, LANL, Julich
- Open source and FREE
- Works with or without recompiling code
  - Dynamic instrumentation (without recompile) provides limited information
- Uses PAPI to measure hardware counters (cache, FLOPS, ...)
- Serial, parallel, GPU profiling capability
- Works with Fortran, C, C++, UPC, Java, Python
- Low performance overhead (can be compensated runtime)
- Complicated and steep learning curve

TAU

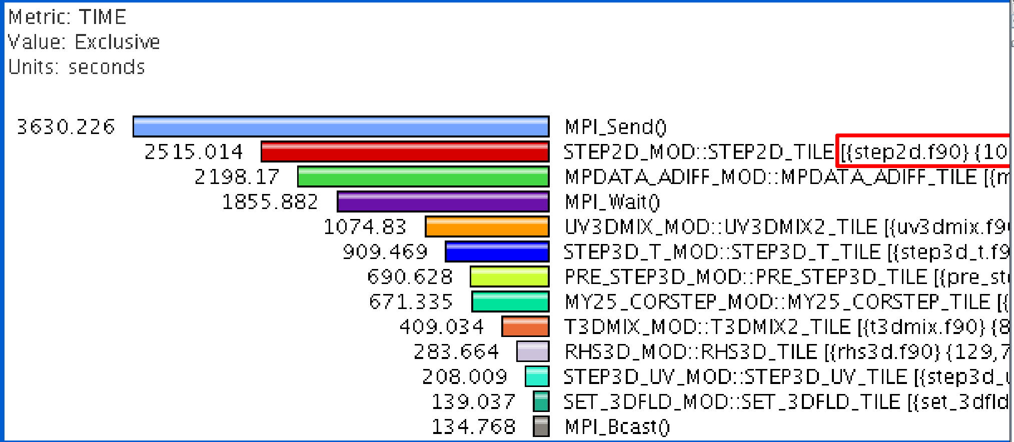
- Instrumentation
  - Source code instrumentation using pre-processors and compiler scripts
    - Instrumentation: Adding code to collect performance, behavior, and resource usage of a program (manually or automatically)
  - Wrapping external libraries (I/O, MPI, Memory, CUDA, OpenCL, pthread, ...)
  - Rewriting the binary
- Measurement
  - Direct: interval events, Indirect: collect samples to profile statement execution
  - Per-process storage of performance data
    - TAU creates one profile file per process in a single location
    - Profile file names look like, profile.0.0.0, profile.1.0.0, ...
  - Throttling and runtime control of low-level events
- Analysis
  - 2D and 3D visualization of profile data using pprof and paraprof
  - Trace conversion & display in external visualizers such as Jumpshot

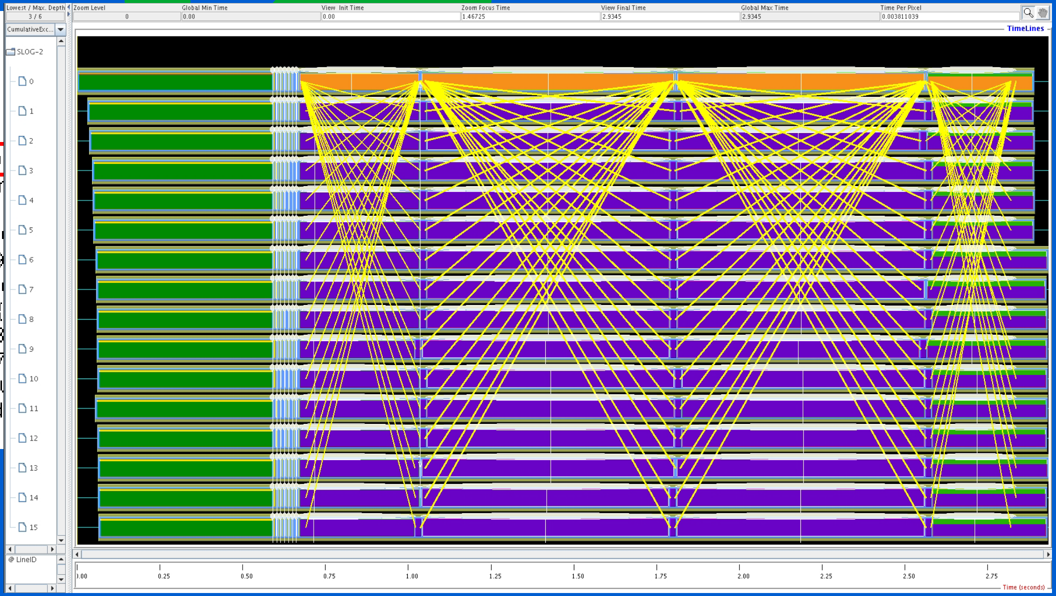
TAU

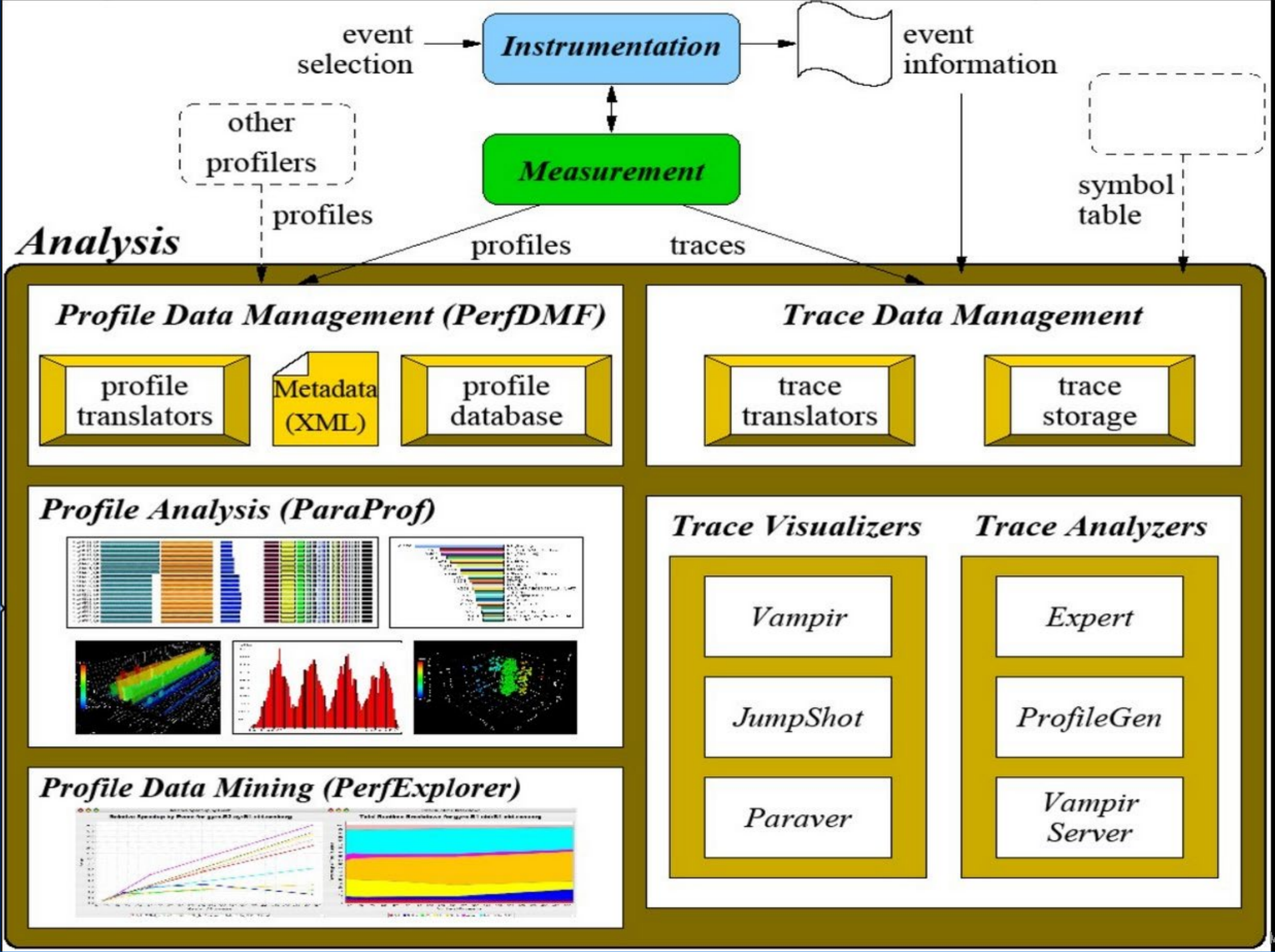
- Profile: statistical summary of all metrics measured
  - Example: Show how much total time & resources each call utilized

- Trace: timeline of events took place
  - Shows when each event happened and where

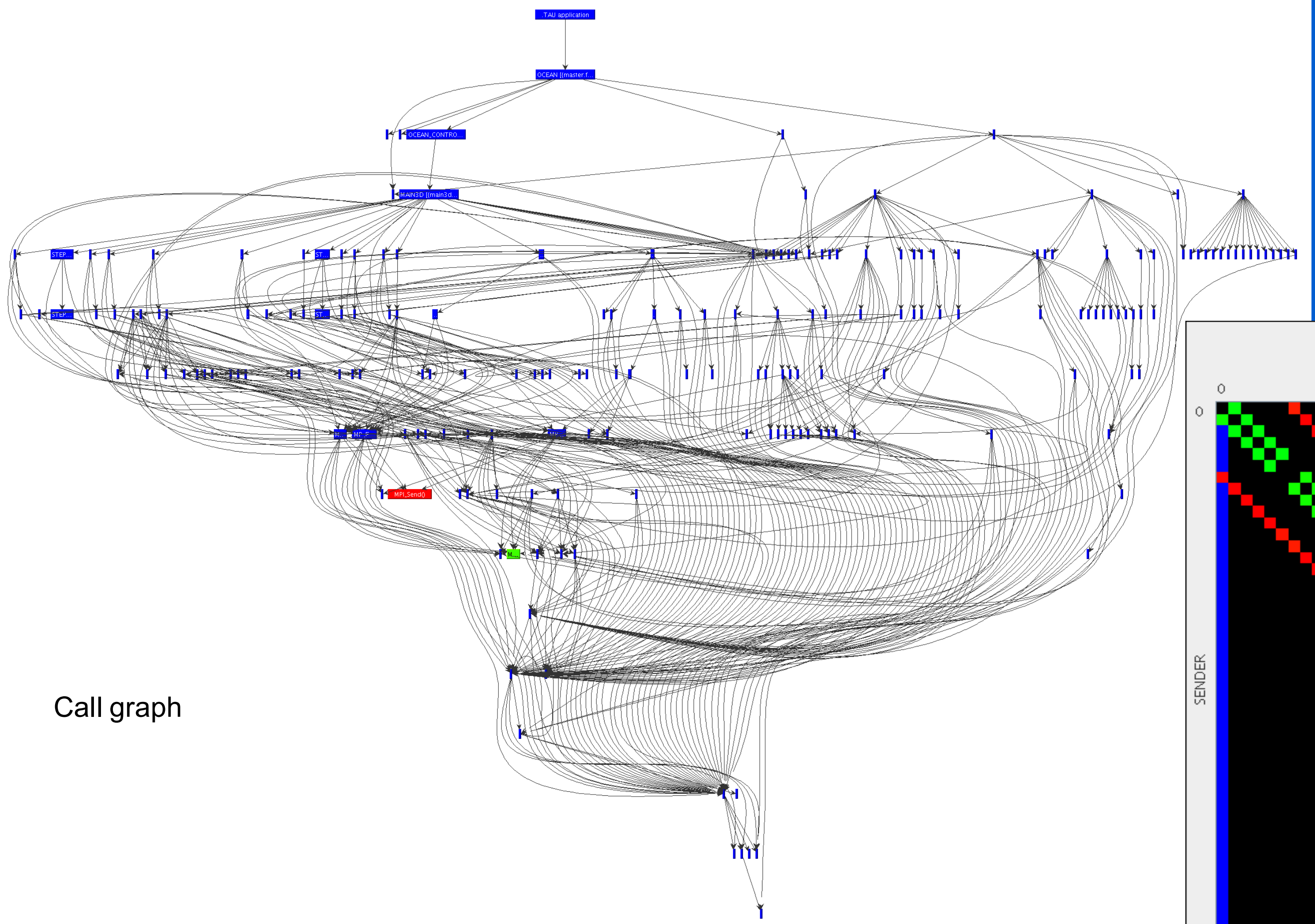
Source file name and location of the function





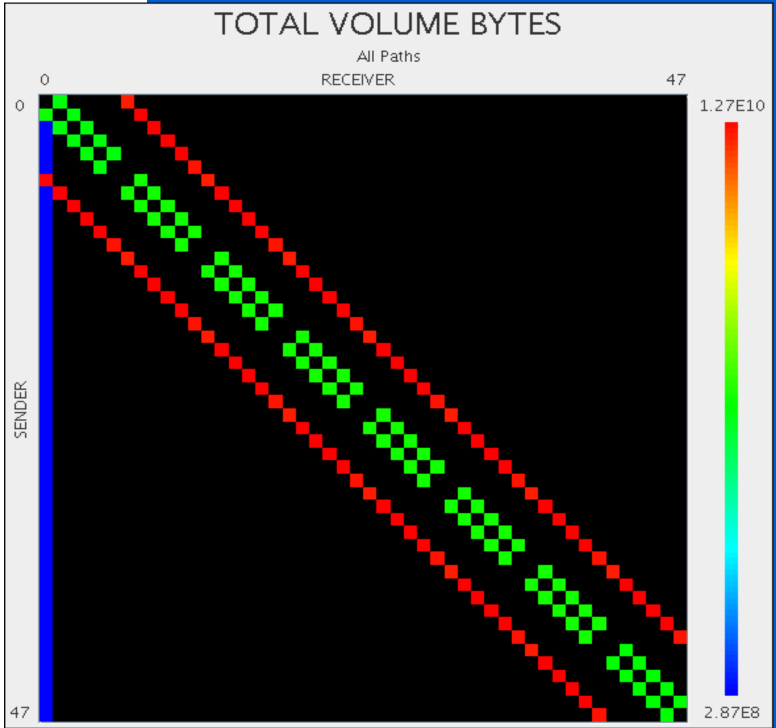

event selection
Instrumentation
event information
other profilers
Measurement
profiles
symbol table
Analysis
profiles
traces
Profile Data Management (PerfDMF)
profile translators
Metadata (XML)
profile database
Trace Data Management
trace translators
trace storage
Profile Analysis (ParaProf)
Trace Visualizers
Trace Analyzers
Vampir
JumpShot
Paraver
Expert
ProfileGen
Vampir Server
Profile Data Mining (PerfExplorer)

TAU



Call graph

Communication matrix



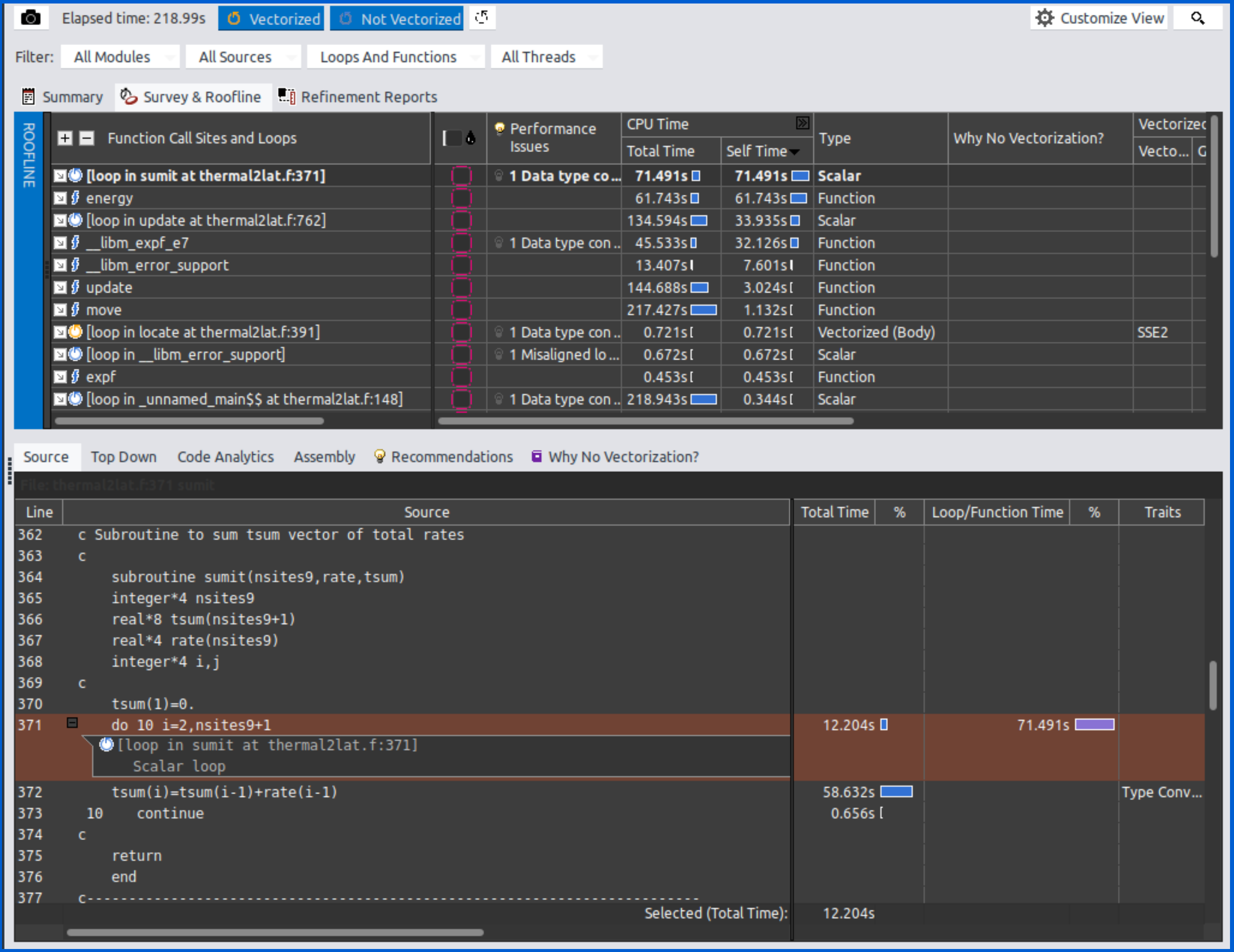
Intel Advisor (FREE)

- Vectorization advisor and Threading advisor
  - Can time-consuming loops able to benefit from vectorization or already vectorized?
  - Compile code with -g
  - Collect data

```
srun -n 1 -c 1 advixe-cl —-collect=survey
        --project-dir=Directory_name          --search-
dir=Directory_name
        --trace-mpi program_name
```

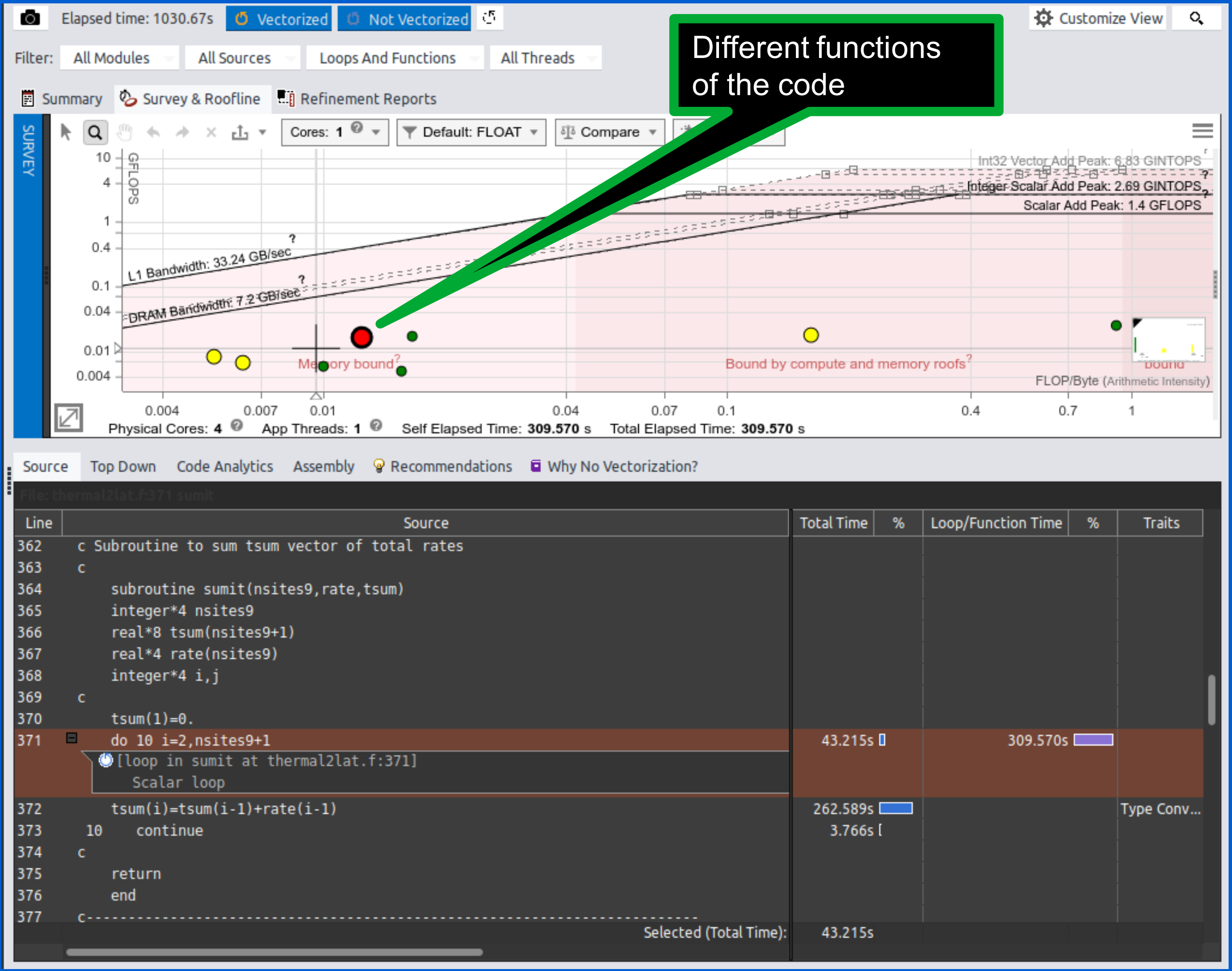
  - Visualize data

```
advisor-gui Directory_name
```



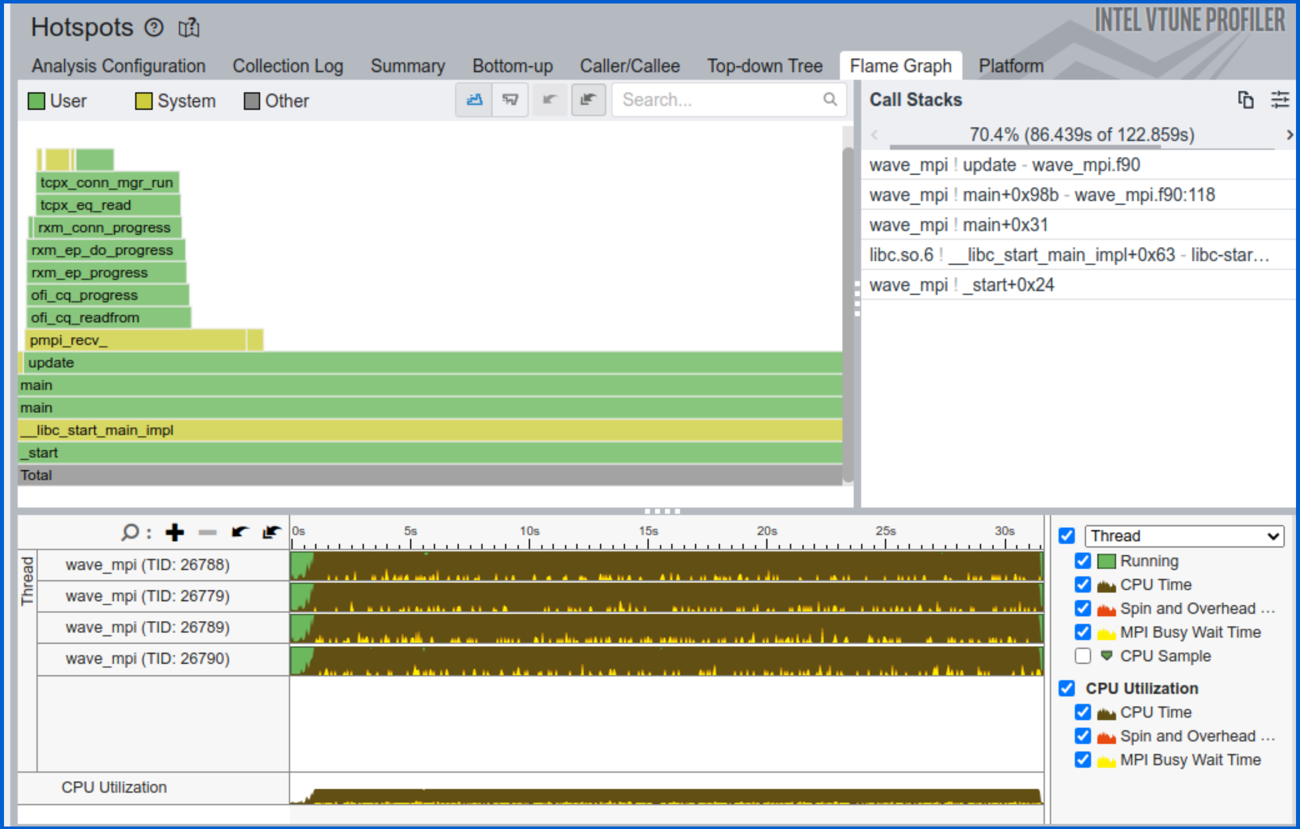
Intel Advisor

- Roofline Analysis for CPU/GPU
  - What is the maximum achievable performance with the hardware used?
  - Does application work optimally on current hardware?
  - If not, what are the best candidates for optimization?
  - Roofline plot shows theoretical limits of computational performance and communication between processors and memory
- Much higher overhead compared to TAU



Intel Vtune (FREE)

- Tune application performance for CPU / GPU
- Profile C, C++, C#, Fortran, OpenCL, Python, Google Go, Java, .NET, Assembly
- Coarse-grained system data for an extended period
- Detailed results mapped to source code
- Multi node (MPI) profiling

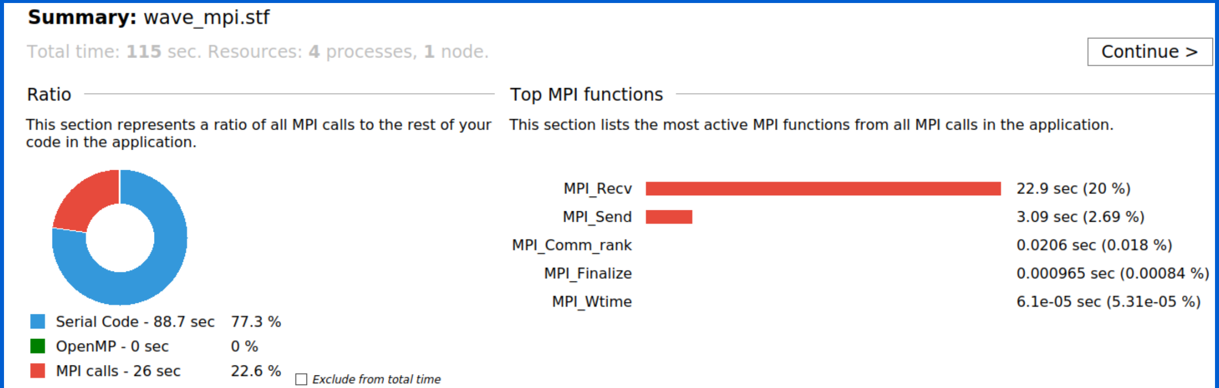


Intel Trace Analyzer (FREE)

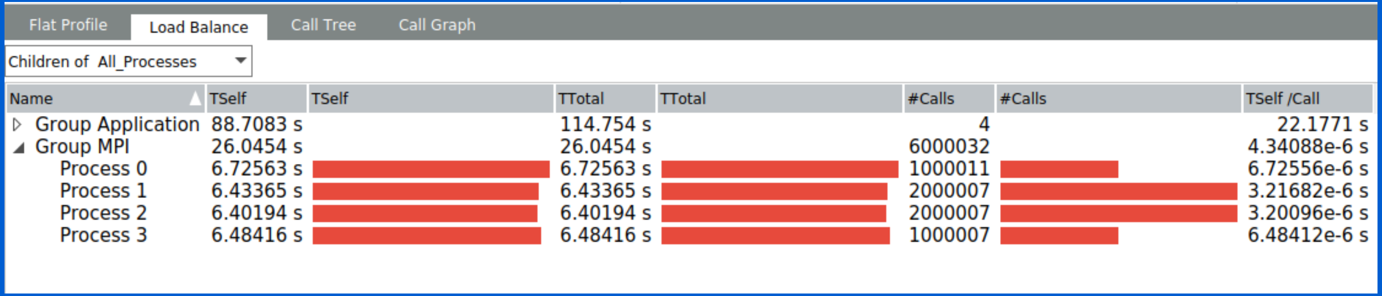
- MPI profiler
- Traces MPI code
- Identifies communication inefficiencies
- To use with Intel MPI (only),

```
$ mpirun -trace -np 4 ./wave_mpi
```

- traceanalyzer gui visualizes generated results




```
$ traceanalyzer wave_mpi.stf
```



Name	TSelf	TSelf	TTotal	TTotal	#Calls	#Calls	TSelf /Call
Group Application	88.7083 s		114.754 s		4		22.1771 s
Group MPI	26.0454 s		26.0454 s		6000032		4.34088e-6 s
Process 0	6.72563 s		6.72563 s		1000011		6.72556e-6 s
Process 1	6.43365 s		6.43365 s		2000007		3.21682e-6 s
Process 2	6.40194 s		6.40194 s		2000007		3.20096e-6 s
Process 3	6.48416 s		6.48416 s		1000007		6.48412e-6 s

Profiling Python

- Two built in profilers: cProfile and profile
  - cProfile is recommended due to low overhead
- Whole program profiling

```
python3 -m cProfile -s tottime numpy_io.py

         3820056 function calls (3805275 primitive calls) in 8.231 seconds
   Ordered by: internal time
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.315    2.315    2.566    2.566 Gio.py:39(run)
        1    0.588    0.588    0.643    0.643 Gtk.py:1(<module>)
    32047    0.468    0.000    0.716    0.000 inspect.py:744(cleandoc)
     5845    0.288    0.000    0.400    0.000 dates.py:305(_dt64_to_ordinalf)
       30    0.214    0.007    0.219    0.007 {built-in method _imp.create_dynamic}
    35070    0.155    0.000    0.242    0.000 _parser.py:83(get_token)
    35146    0.137    0.000    0.137    0.000 {method 'astype' of 'numpy.ndarray' objects}
      282    0.134    0.000    0.134    0.000 {method 'read' of '_io.BufferedReader' objects}
…
```

- Targeted profiling
  - Only profile a selected parts (functions etc) of a code

```
import cProfile
pr = cProfile.Profile()
pr.enable()
# ... your code/function to profile ...
pr.disable()
pr.print_stats()
```

Profiling Python

- Line profiling
  - Only profile selected lines of a code

```
$ pip install line_profiler
```

```
$ kernprof -l -v prfact.py
 Please Enter any Number: 2544
Wrote profile results to prfact.py.lprof
Timer unit: 1e-06 s

Total time: 0.00451784 s
File: prfact.py
Function: prfct at line 5

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     5                                           @profile
     6                                           def prfct(n):
     7      2543       2039.2      0.8     45.1      for i in range(2, n + 1):
     8      2524       2350.7      0.9     52.0          if(n % i == 0):
     9        19         15.5      0.8      0.3              isprime = 1
    10        42         57.5      1.4      1.3              for j in range(2, (i //2 + 1)):
    11        26         23.4      0.9      0.5                  if(i % j == 0):
    12        16         11.3      0.7      0.2                      isprime = 0
    13        16         20.2      1.3      0.4                      break
```

-l : line by line
-v : visualize results

```python
import line_profiler

Number = int(input(" Please Enter any Number: "))

@profile
def prfct(n):
    for i in range(2, n + 1):
        if(n % i == 0):
            isprime = 1
            for j in range(2, (i //2 + 1)):
                if(i % j == 0):
                    isprime = 0
                    break

prfct(Number)
```

Profiling Python

- Memory profiling
  - Keep track of memory usage

```
$ pip install memory_profiler
```

```
import memory_profiler

Number = int(input(" Please Enter any Number: "))

@profile
def prfct(n):
    for i in range(2, n + 1):
        if(n % i == 0):
            isprime = 1
            for j in range(2, (i //2 + 1)):
                if(i % j == 0):
                    isprime = 0
                    break

prfct(Number)
```

```
$ python3 -m memory_profiler prfact.py
 Please Enter any Number: 2588
Filename: prfact.py

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     5  21.875 MiB  21.875 MiB           1   @profile
     6                                         def prfct(n):
     7  21.875 MiB   0.000 MiB        2588       for i in range(2, n + 1):
     8  21.875 MiB   0.000 MiB        2587           if(n % i == 0):
     9  21.875 MiB   0.000 MiB           5               isprime = 1
    10  21.875 MiB   0.000 MiB         327               for j in range(2, (i //2 + 1)):
    11  21.875 MiB   0.000 MiB         325                   if(i % j == 0):
    12  21.875 MiB   0.000 MiB           3                       isprime = 0
    13  21.875 MiB   0.000 MiB           3                       break
```

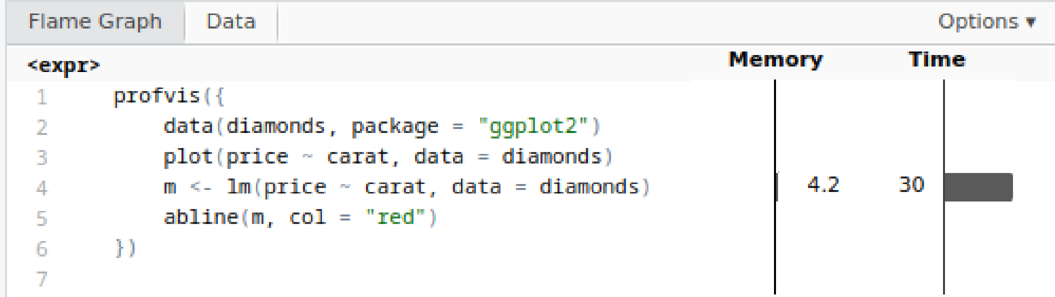
Profiling R

- Select Rstudio's built in Profile > Start Profiling menu and run the R code
- Enclose the function or code with profvis function
- Enables a user to:
  - Measure time and memory
  - Find bottlenecks

```
library(profvis)
profvis({
  data(diamonds, package = "ggplot2")
  plot(price ~ carat, data = diamonds)
  m <- lm(price ~ carat, data = diamonds)
  abline(m, coln0 = "red")
})
```

Flame Graph | Data | Options ▾

Code	File	Memory (MB)		Time (ms)
.rs.GEcopyDisplayList		0	2.6	6960
▼ plot.formula		0	5.6	4040
▼ plot.default		0	5.6	4040
▶ plot.xy		0	0.2	3480
deparse1		0	1.6	470
▶ localAxis		0	0.0	80
range		0	3.7	10
▶ .rs.saveGraphics		0	1.0	540
lazyLoadDBfetch		0	7.7	100
▶ base::try		0	0.2	50
▶ lm	<expr>	0	4.2	30
▶ model.frame.default		0	3.9	20

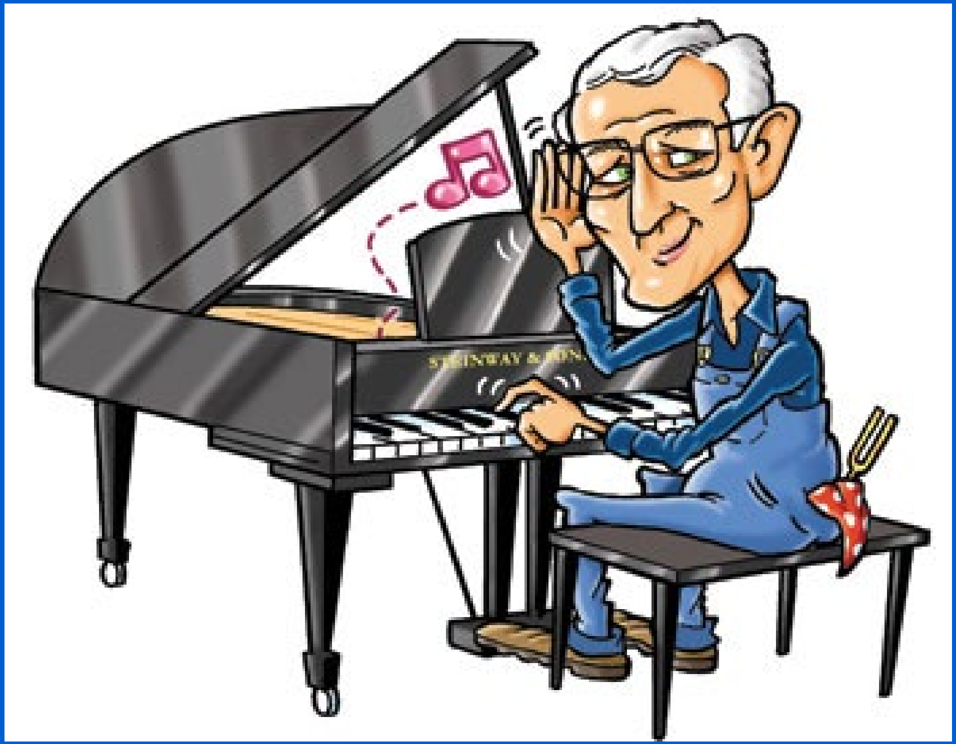


- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Tuning Applications

- Code tuning is the process of manually optimizing a program to lower its runtime requirements (runtime, memory, disk space, ...)
  - Better algorithms
  - Different compiler flags (-O2, -O3 etc)
  - Using different compilers
  - Using optimized libraries
  - Vectorizing loops
  - Using non-blocking MPI calls
    - Hide latency



https://magazine.foriowa.org/archive/archive-story.php?ed=true&storyid=1568

Use Compiler Flags

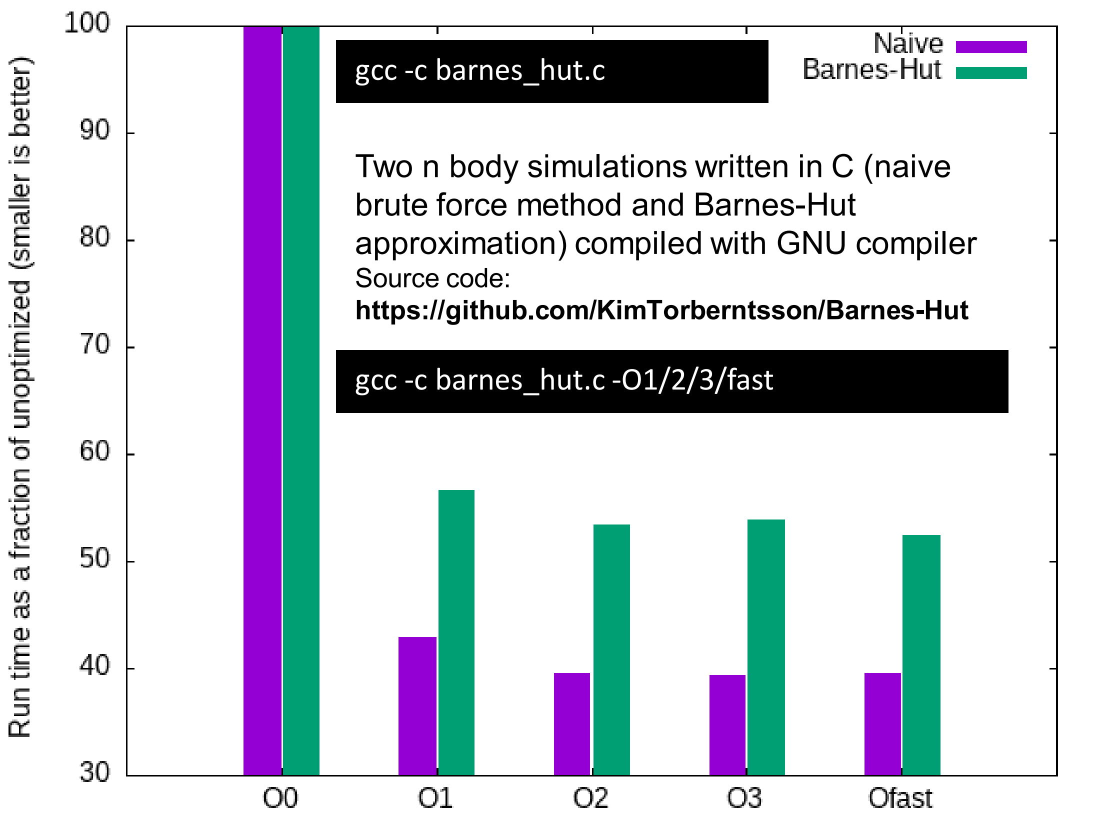
- -march=cpu-type : Generate instructions for the machine type cpu-type
  - Exploits various capabilities in different CPUs, support for different instruction sets, different ways of executing code, etc to generate optimized binary for a target CPU
  - cpu-type = native : Use processor type of the compiling machine (local machine installation, compiling for a homogeneous cluster etc)
  - cpu-type = sandybridge, haswell, skylake, znver2, etc : Compile for Intel Sandy Bridge, Haswell, Skylate, AMD zen2, etc
  - cpu-type = core-avx2 (Intel compiler): Compile for a for processors that supports Advanced Vector Extensions 2

Use Compiler Flags

- -O : Vectorization, scalar optimizations, loop optimizations, inlining, …
  - Too aggressive optimizations may affect computtional accuracy

option	optimization level
-O0	optimization for compilation time (default)
-O1 or -O	optimization for code size and execution time
-O2	optimization more for code size and execution time
-O3	optimization more for code size and execution time
-Os	optimization for code size
-Ofast	O3 with fast none accurate math calculations

https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html



Modular Assembly Quality Analyzer and Optimizer (MAQAO)

- A profiling tool, binary disassembler, and code quality analyzer
  - A user friendly performance analysis and optimization framework
  - Provides reports and hints for code optimization
  - Analyzes production binary
- Binary can be freely downloaded from https://www.maqao.org
- Analyzing applications and generating a report

```
maqao oneview –create-report=one –binary=../test/wave –mpi_command=”mpirun -np 2”
```

- This will run the binary with the given mpi command and generate the results
  - Default report format is html and can be configured to text / excel etc
- Focuses on memory alignment, loop interchange, loop strides, etc

MAQAO

MAQAO | Global | Application | Functions | Loops | Topology

wave - 2023-06-15 18:59:50 - MAQAO 2.17.0

Help is available by moving the cursor above any ? symbol or by checking MAQAO website.

Global Metrics

Total Time (s)		519.16
Profiled Time (s)		517.72
Time in analyzed loops (%)		99.6
Time in analyzed innermost loops (%)		99.5
Time in user code (%)		99.6
Compilation Options Score (%)		25.0
Perfect Flow Complexity		1.00
Array Access Efficiency (%)		83.3
Perfect OpenMP + MPI + Pthread		1.00
Perfect OpenMP + MPI + Pthread + Perfect Load Distribution		1.00
No Scalar Integer	Potential Speedup	2.26
	Nb Loops to get 80%	2
FP Vectorised	Potential Speedup	1.46
	Nb Loops to get 80%	1
Fully Vectorised	Potential Speedup	10.9
	Nb Loops to get 80%	2
FP Arithmetic Only	Potential Speedup	3.20
	Nb Loops to get 80%	2

Compilation Options

Source Object	Issue
▼ wave	
▼ wave_mpi.f90	
○	-O2, -O3 or -Ofast is missing.
○	-march=x86-64 is used but it should be replaced by a more architecture specific option or -march=native.
○	-funroll-loops is missing.

Array Access Efficiency: Percentage of Unit Stride access

FP vectorized: Performance gain if all FP arithmetic operations were vectorized

Fully vectorized: Performance gain if all the FP arithmetic operations + Load/Store instructions were vectorized

Code clean check

Detected a slowdown caused by scalar integer instructions (typically used for address computation). By removing them, you can lower the cost of an iteration from 14.00 to 6.75 cycles (2.07x speedup).

Workaround

- Try to reorganize arrays of structures to structures of arrays
- Consider to permute loops (see vectorization gain report)

Vectorization

Your loop is probably not vectorized. Only 12% of vector register length is used (average across all SSE/AVX instructions). By vectorizing your loop, you can lower the cost of an iteration from 14.00 to 1.32 cycles (10.60x speedup).

Details

Store and arithmetical SSE/AVX instructions are used in scalar version (process only one data element in vector registers). Since your execution units are vector units, only a vectorized loop can use their full power.

Workaround

- Try another compiler or update/tune your current one:
  - recompile with ftree-vectorize (included in O3) to enable loop vectorization and with fassociative-math (included in Ofast or ffast-math) to extend vectorization to FP reductions.
- Remove inter-iterations dependences from your loop and make it unit-stride:
  - If your arrays have 2 or more dimensions, check whether elements are accessed contiguously and, otherwise, try to permute loops accordingly
  - If your loop streams arrays of structures (AoS), try to use structures of arrays instead (SoA)

Execution units bottlenecks

gain | potential | hint | expert

Complex instructions

Detected COMPLEX INSTRUCTIONS.

Details

These instructions generate more than one micro-operation and only one of them can be decoded during a cycle and the extra micro-operations increase pressure on execution units.

- ADD: 1 occurrences

Slow data structures access

Detected data structures (typically arrays) that cannot be efficiently read/written

Details

- Constant non-unit stride: 2 occurrence(s)

Non-unit stride (uncontiguous) accesses are not efficiently using data caches

Workaround

- Try to reorganize arrays of structures to structures of arrays
- Consider to permute loops (see vectorization gain report)

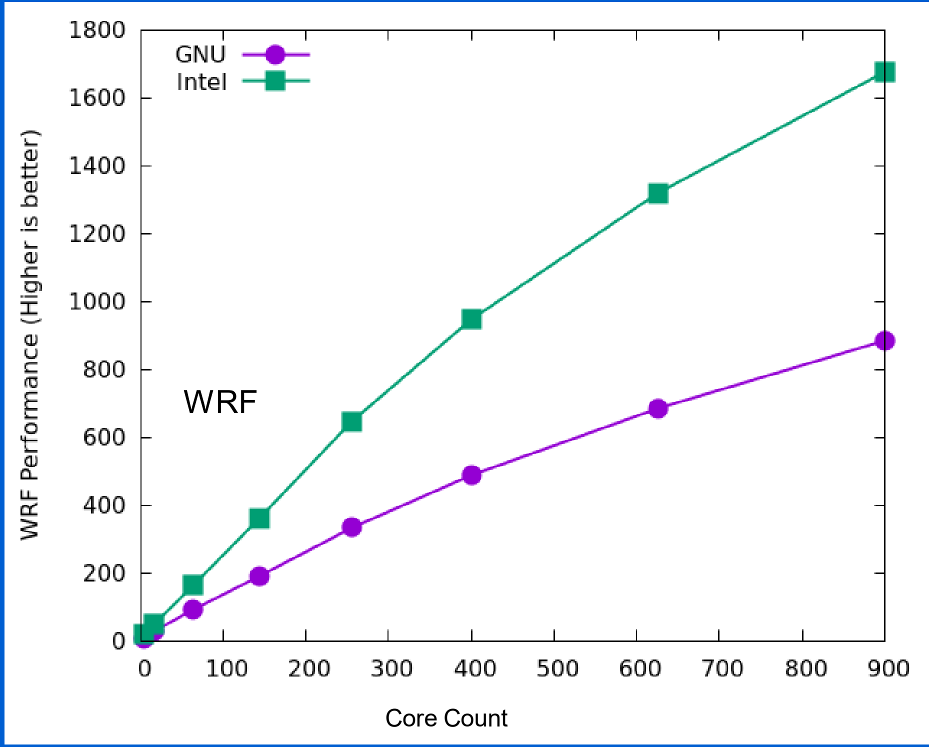
Conversion instructions

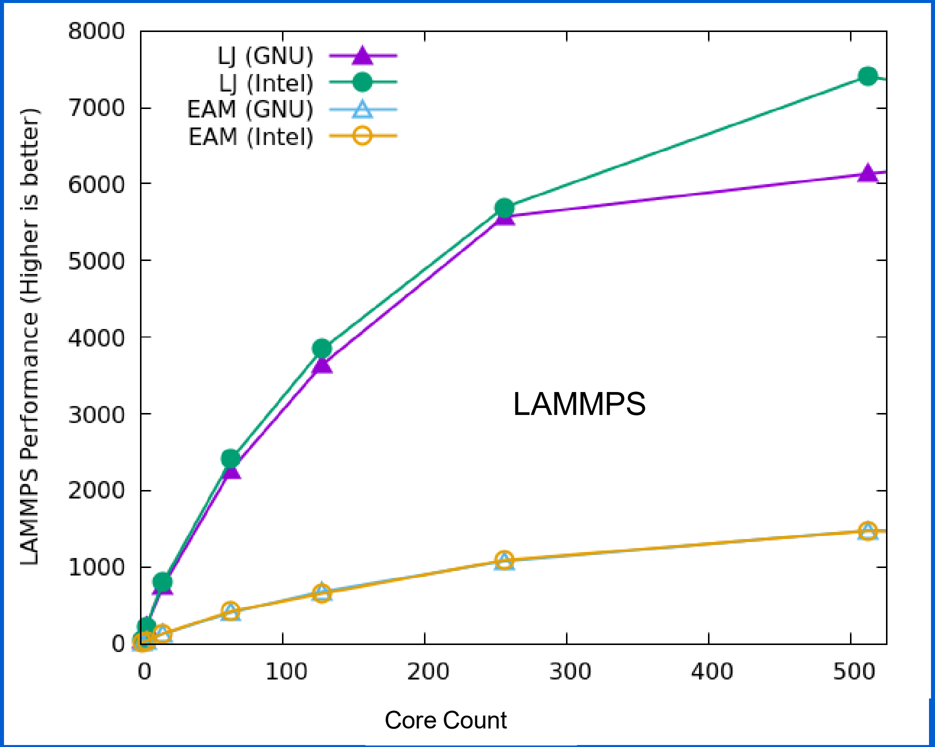
- Debugging
  - Debugging using Compiler Flags
  - Debugger Basics
  - gdb
  - Serial Debugging with gdb
  - Parallel (MPI) Debugging
    - Parallel Debugging with gdb
    - Interactive Parallel Debugging with gdb
    - Non-interactive Parallel Debugging with gdb
    - Totalview and DDT
  - CUDA Debugging with gdb
  - Intel Inspector
  - Language Specific Debuggers

- Profiling and Tuning
  - Profiling
    - GNU Profiler - gprof
    - TAU
    - Intel Tools
    - Profiling Python and R
  - Tuning Applications
    - Use Compiler Flags
    - MAQAO
    - Try Different Compilers
    - Use Performance Optimized Libraries

Try Different Compilers

- Different compilers (GNU vs Intel vs other) *may* yield different performance
  - OpenMPI vs MVAPICH2 vs Intel MPI





Use Performance Optimized Libraries

- Solving most problems numerically involves performing similar tasks
  - Vector operations (dot product, norm, …)
  - Matrix operations (solving systems of equations, matrix product, …)
  - Fourier Transform, parallel input/output

- Numerical libraries are developed to perform these basic tasks optimally
  - Highly tuned for performance, different hardware for decade(s)
  - Well documented and easy to use
    - Most become community standards (eg. *FFTw*)
  - Use these libraries as building blocks to develop applications
    - **Never write your own solvers!**

Numerical Libraries

- BLAS
  - Basic Linear Algebra Subprograms
    - Written in Fortran, provides C bindings
  - Provides a **standard** interface to vector, matrix-vector, matrix-matrix routines that have been optimized for various computer architectures
  - Implementations: OpenBLAS, BLIS (BLAS-like Library Instantiation Software), ATLAS (Automatically Tuned Linear Algebra Software), Intel Math Kernel Library (IMKL), Accelerate, cuBLAS (cuda BLAS), GotoBLAS, …
- LAPACK
  - Linear Algebra PACKage: Built on top of BLAS
  - Designed to solve system of linear equations, eigenvalue problems, singular value problems, LU factorization, etc
  - ScaLAPACK: Parallel version of LAPACK

Numerical Libraries

- SuiteSparse
- Armadillo
- PROJ
- GTS
- TRILINOS
- TA-Lib
- deal.II
- GEOS
- PETSc
- SLEPSc
- HYPRE
- Scotch
- BLAS
- LAPACK
- FFTw
- SuperLU
- ScaLAPACK
- Eigen
- ELSI
- NNPACK
- ARPACK
- MUMPS
- OGDI
- GDAL
- sundials
- CGAL
- HDF
- netCDF

Numerical Libraries

- Try to use libraries widely used and still active / supported
  - Most issues were identified and fixed
  - Community support
- Test different libraries if available and check performance
  - Performance may differ depending on usage, hardware, etc
  - Use libraries built / tuned for your hardware architecture

Thank you