Debugging and Tuning

Prasad Maddumage

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, hardware issues
- Terminal based debugging
  - write/printf, gdb/idb, valgrind (memory issues), …
  - Can effectively pinpoint problems, works with serial and parallel codes
  - Need to remember commands, need recompiling codes
- GUI debuggers
  - TotalView, DDT, Intel Inspector (GUI and cli)
  - Powerful and user friendly

gdb Serial Debugging

```
$ 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
```

gdb Serial Debugging

```
(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)
```

```
program test
  implicit none

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

  allocate(x(5))

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

end program test
```

- Core dump analysis

```
$ gdb <path to binary> <path to core dump>
```

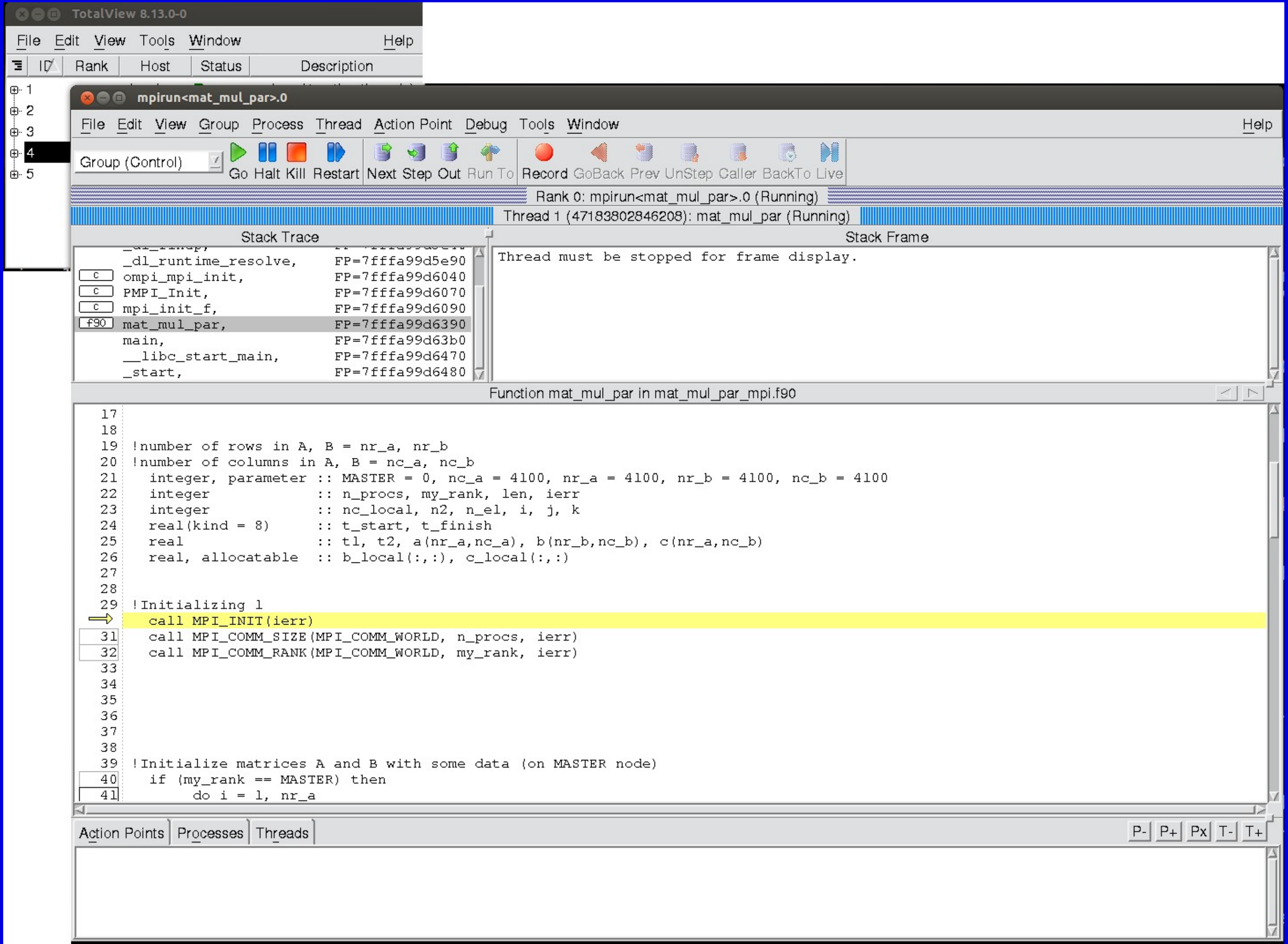
  - Use `bt / frame / list / info locals / print` etc to pin point the cause

Parallel (MPI) Debugging

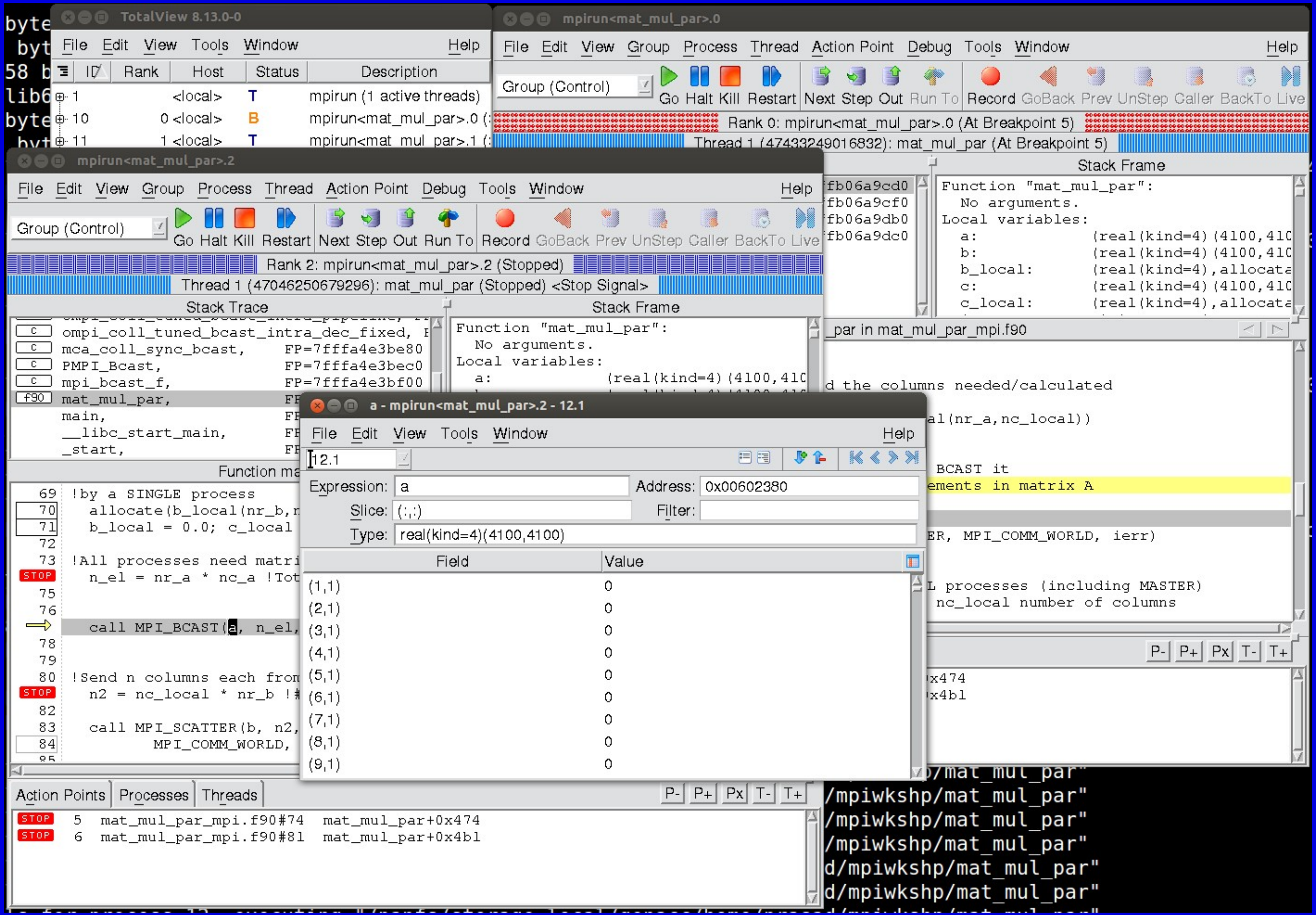
- Attach gdb to each process of a running job and examine
  - ```
    $ gdb attach <pid>
    ```
  - Can submit a (SLURM) job with gdb commands (break etc) supplied
- TotalView and DDT
  - GUI debuggers offer convenience
  - Expensive!
- Intel Inspector
  - Memory / thread checker
  - `inspxe-gui` and `inspxe-cl`
    ```

TotalView

- Recompile with -g
- `totalview <your binary>`
- Open TotalView and use startup dialog to choose the binary
- Attach TotalView to an already running job



TotalView



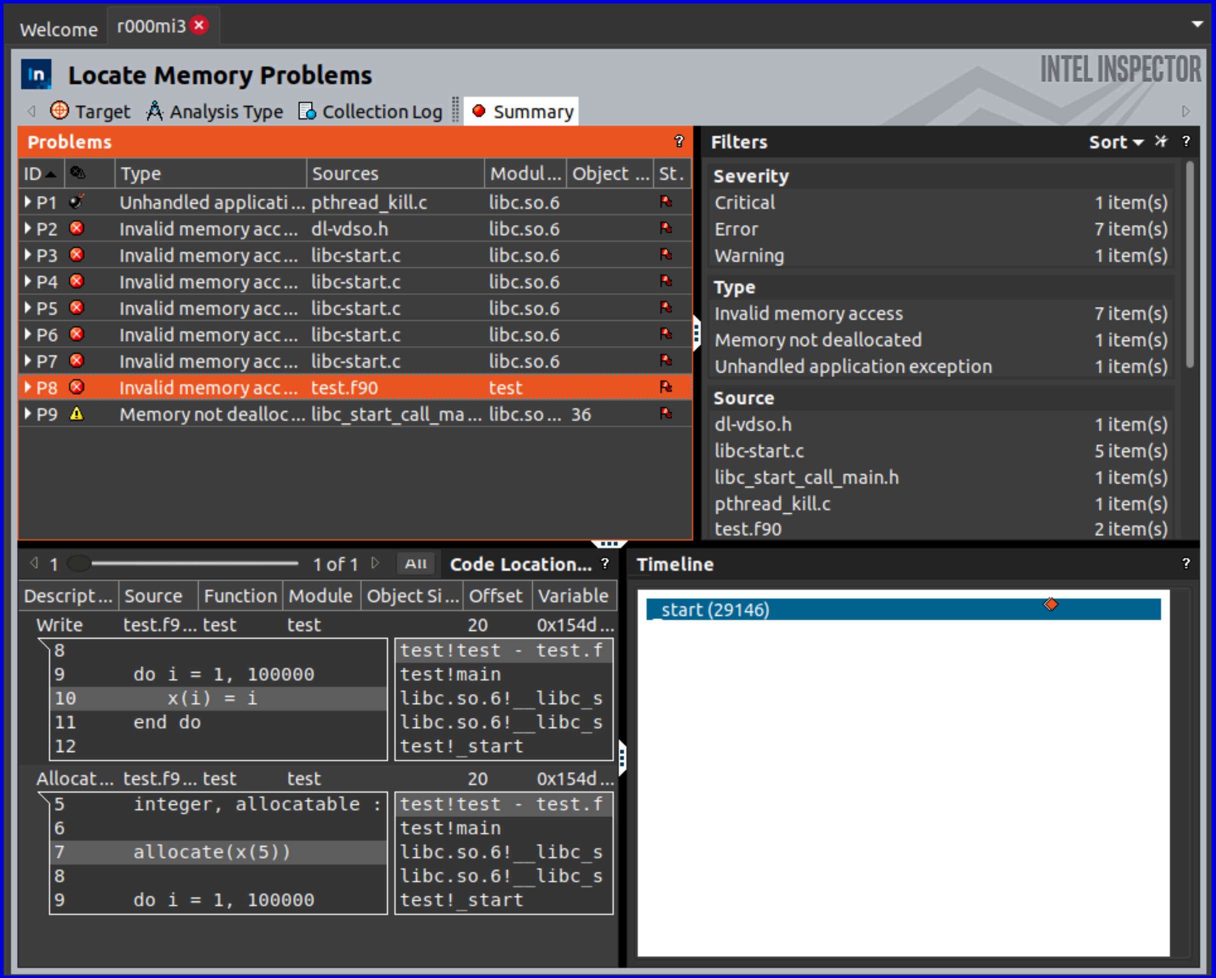
TotalView 8.13.0-0
File Edit View Tools Window Help
ID Rank Host Status Description
1 <local> T mpirun (1 active threads)
10 0 <local> B mpirun<mat_mul_par>.0
11 1 <local> T mpirun<mat_mul_par>.1
mpirun<mat_mul_par>.0
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Rank 0: mpirun<mat_mul_par>.0 (At Breakpoint 5)
Thread 1 (47433249016832): mat_mul_par (At Breakpoint 5)
Stack Frame
mpirun<mat_mul_par>.2
File Edit View Group Process Thread Action Point Debug Tools Window Help
Group (Control)
Go Halt Kill Restart Next Step Out Run To Record GoBack Prev UnStep Caller BackTo Live
Rank 2: mpirun<mat_mul_par>.2 (Stopped)
Thread 1 (47046250679296): mat_mul_par (Stopped) <Stop Signal>
Stack Trace
ompi_coll_tuned_bcast_intra_dec_fixed,
mca_coll_sync_bcast, FP=7fffa4e3be80
PMPI_Bcast, FP=7fffa4e3bec0
mpi_bcast_f, FP=7fffa4e3bf00
mat_mul_par,
main,
__libc_start_main,
_start,
Stack Frame
Function "mat_mul_par":
No arguments.
Local variables:
a: (real(kind=4)(4100,410
b: (real(kind=4)(4100,410
b_local: (real(kind=4),allocata
c: (real(kind=4)(4100,410
c_local: (real(kind=4),allocata
69 !by a SINGLE process
70 allocate(b_local(nr_b,
71 b_local = 0.0; c_local
73 !All processes need matri
n_el = nr_a * nc_a !Tot
call MPI_BCAST(a, n_el,
80 !Send n columns each from
n2 = nc_local * nr_b !#
83 call MPI_SCATTER(b, n2,
84 MPI_COMM_WORLD,
a - mpirun<mat_mul_par>.2 - 12.1
File Edit View Tools Window Help
Expression: a
Address: 0x00602380
Slice: (:,:)
Filter:
Type: real(kind=4)(4100,4100)
Field Value
(1,1) 0
(2,1) 0
(3,1) 0
(4,1) 0
(5,1) 0
(6,1) 0
(7,1) 0
(8,1) 0
(9,1) 0
_par in mat_mul_par_mpi.f90
d the columns needed/calculated
BCAST it
ements in matrix A
ER, MPI_COMM_WORLD, ierr)
L processes (including MASTER)
nc_local number of columns
Action Points Processes Threads
P- P+ Px T- T+
STOP 5 mat_mul_par_mpi.f90#74 mat_mul_par+0x474
STOP 6 mat_mul_par_mpi.f90#81 mat_mul_par+0x4b1

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
- Free!



Language Specific Debuggers

- Bash: Use debug (xtrace) mode with `-x` option
- Python: pdb – add `breakpoint()` in the source (Python3)
  - Interactive source debugger
  - Supports breakpoints and single stepping at the source line level
  - Inspection of stack frames, source code listing
- R: Use RStudio
  - Set breakpoints in RStudio or put `browser()` in the line you want to break
  - Then the IDE will enter debug mode
    - Can check current variable stack, traceback the execution, and more

Tuning

- Tuning involves profiling software to finding room for improvement and making a code run more efficiently
  - Software profiling: dynamic code analysis where a program's behavior is investigated using the data collected as the program runs
    - CPU/memory utilization, frequency of function calls, I/O, MPI library usage, hardware counters, etc.
  - Profiling can help find ways to increase efficiency of a program
  - Identify bottlenecks
  - Improve scaling of parallel programs
- Profilers
  - gprof, TAU (Tuning and Analysis Utilities), Intel tools

gprof

- Terminal based serial profiler
- Produces flat profile and a call graph
  - Flat profile: A breakdown of time spent on each subroutine / function call
  - Call graph: In what order each subroutine / function was called
- Comes with gcc and already exist in most systems

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

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  s/call   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

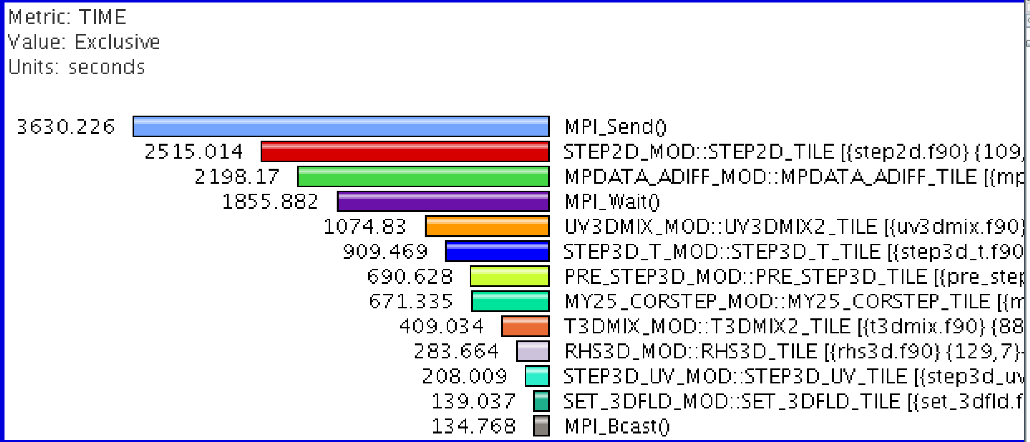
index % time    self  children    called     name
                0.03   20.45       1/1           main [2]
[1]     99.9    0.03   20.45       1         MAIN__ [1]
                0.13   20.08  771780/771780      move_ [3]
                0.16    0.00  771897/771897      locate_ [7]
```

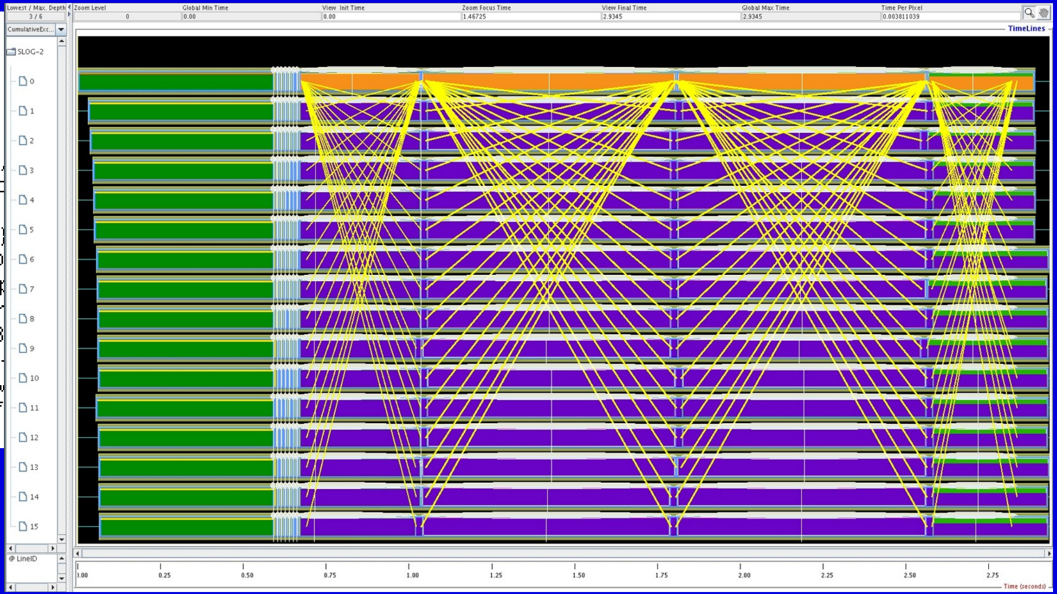
TAU

- Tuning and Analysis Utilities (20+ year project actively developed by Univ. of Oregon, LANL, Julich)
- Integrated performance toolkit
  - Instrumentation, measurement, analysis, visualization
  - Performance data management and data mining
  - Open source and free
- Works with or without recompiling code
  - Dynamic instrumentation (without recompile) provides limited information
- Use PAPI to measure hardware counters (cache, FLOPS, ...)
- Serial and MPI profiling capability
- Complicated / steep learning curve

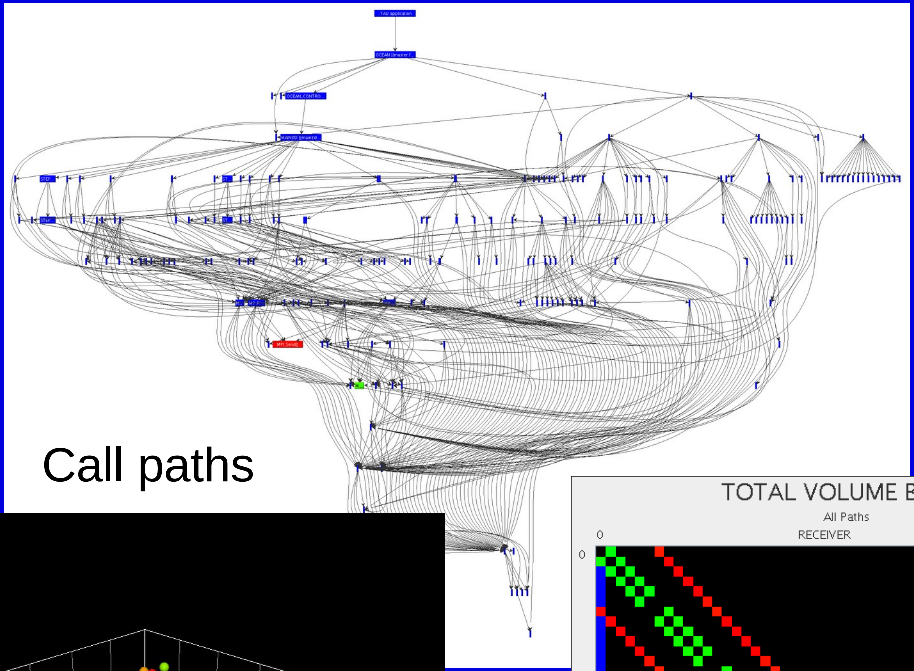
TAU

- Profile: statistical summary of all metrics measured
  - Shows how much total time & resources each call utilized
- Trace: timeline of runtime events took place
  - Shows when each event happened and where

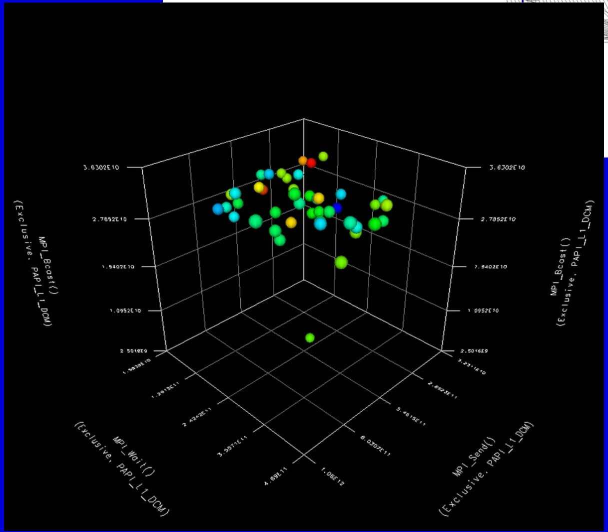


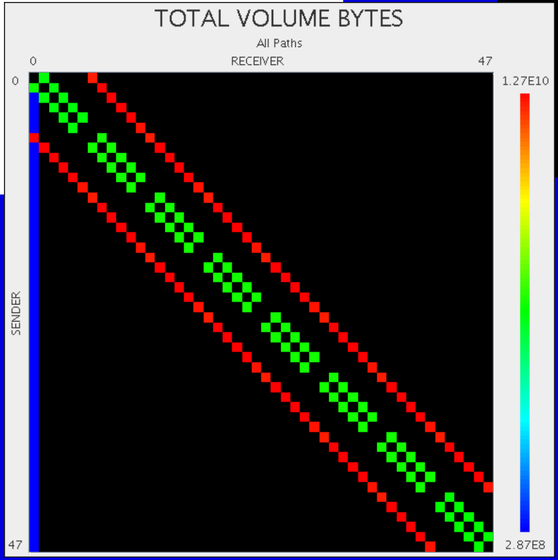


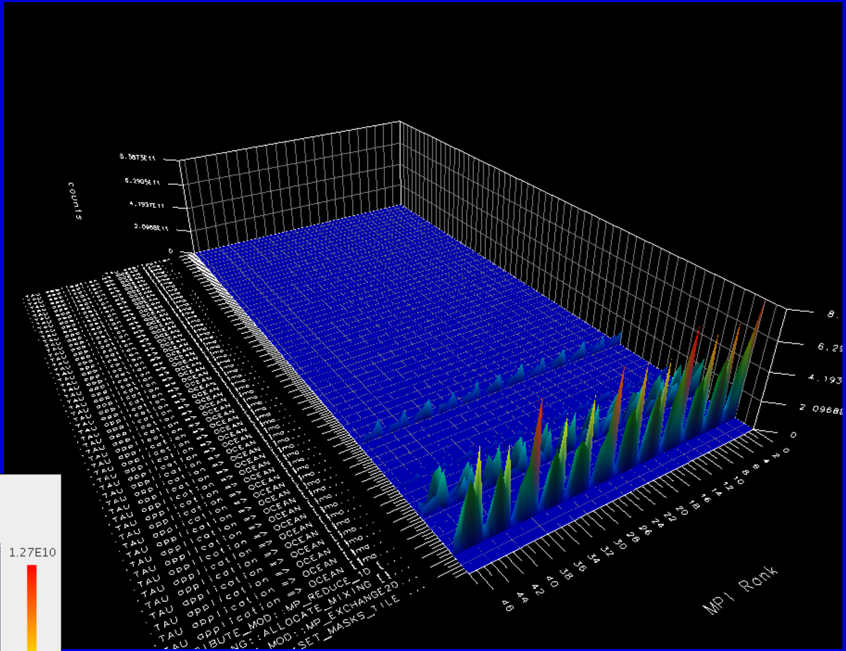
TAU



Call paths



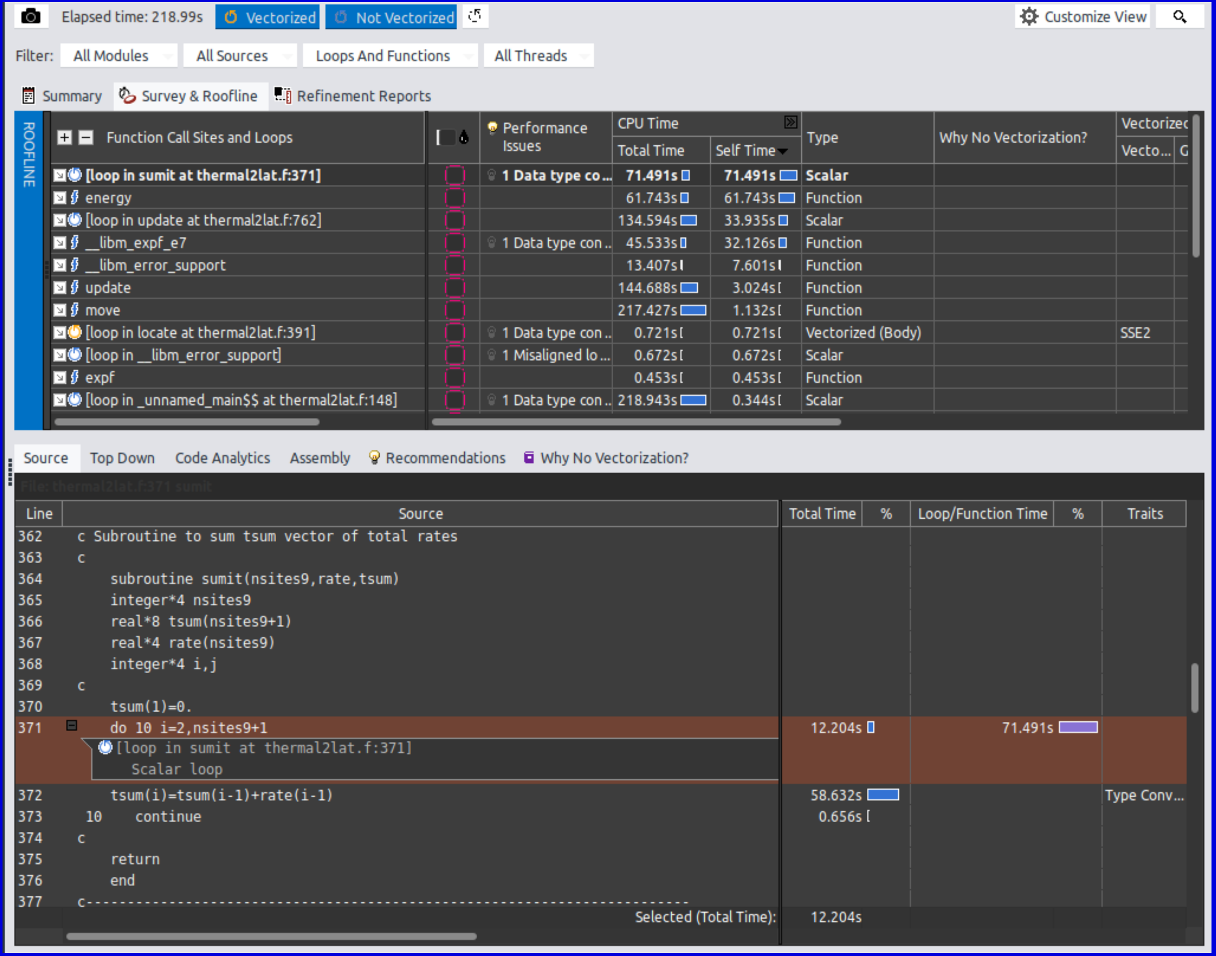




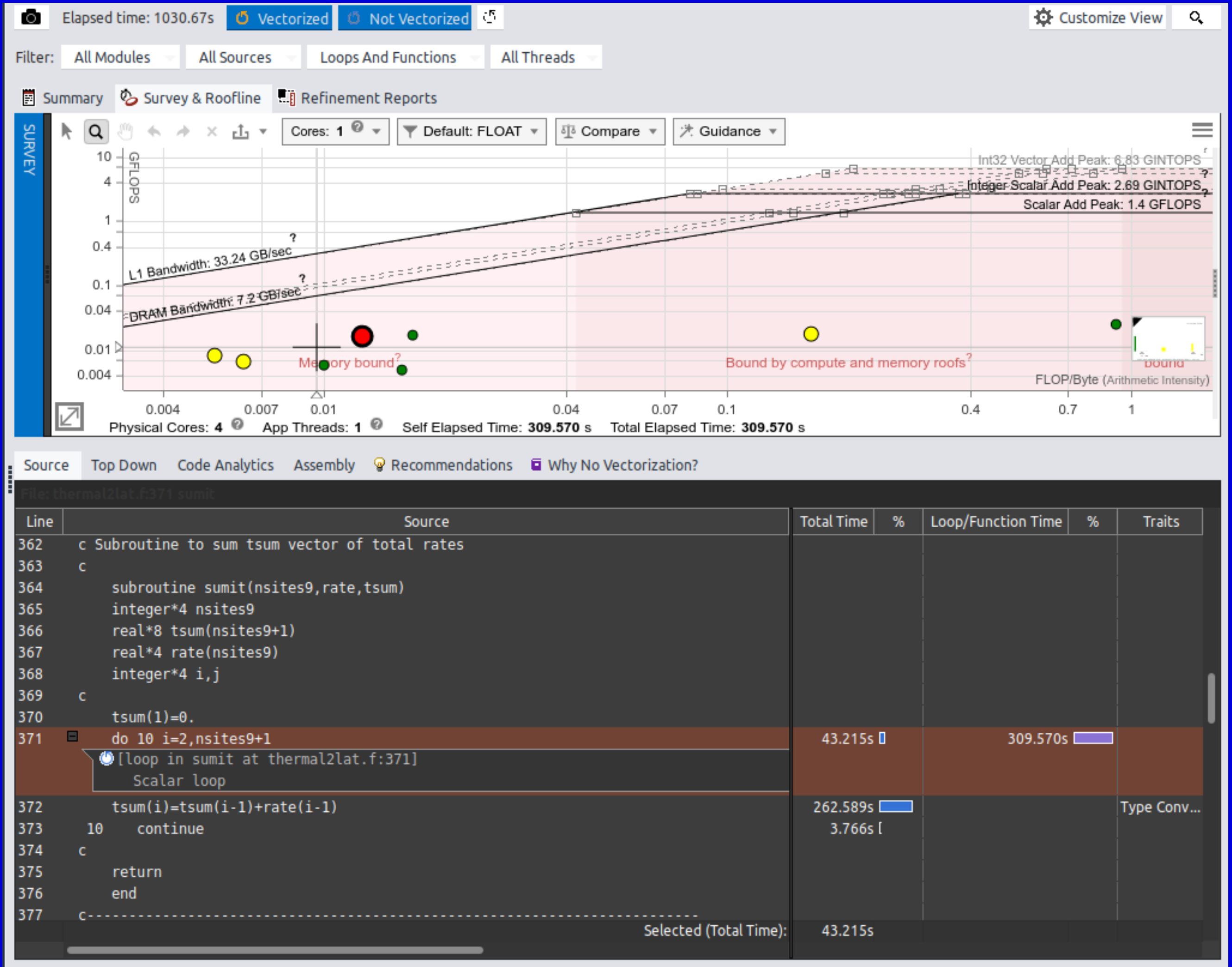
Communication matrix

Intel Advisor

- Roofline Analysis for CPU/GPU
- Vectorization Optimization
- Offload Modeling
- Thread Prototyping
- Flow Graph Analyzer
- Much higher overhead compared to TAU

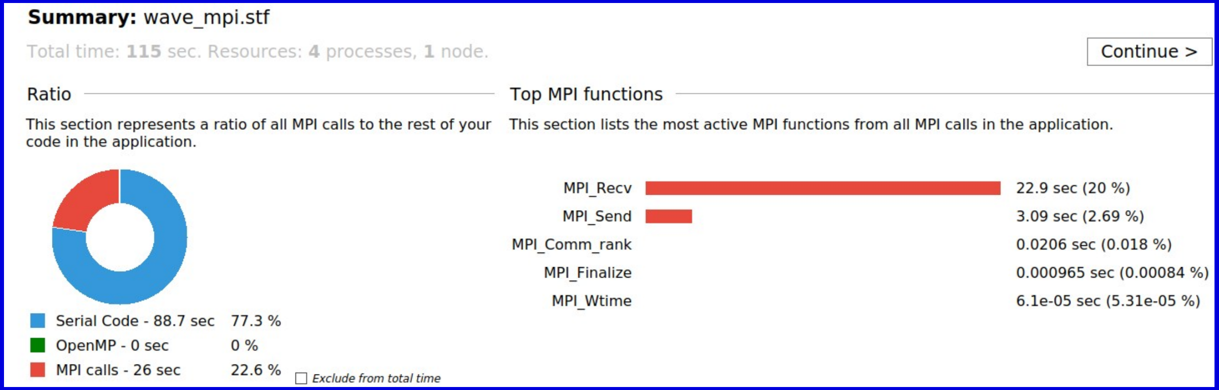


Intel Advisor



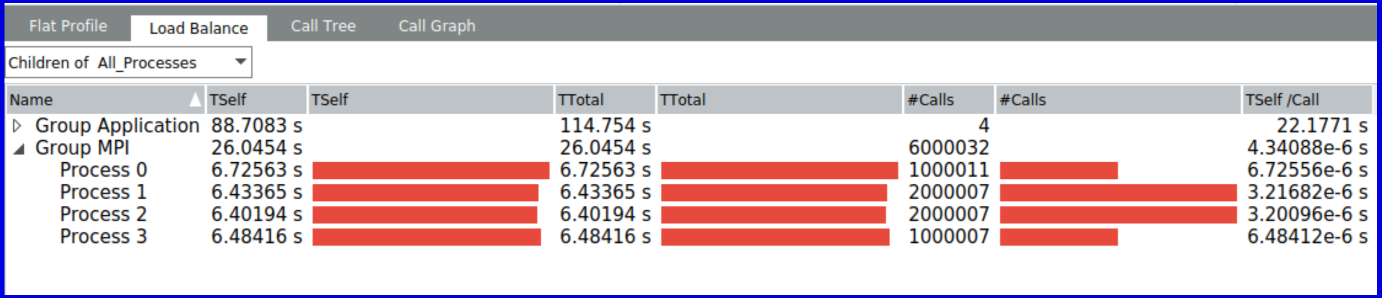
Intel Trace Analyzer

- MPI profiler
- traces MPI code
- identifies communication inefficiencies
- to profile the executable, just append '`-trace`' to mpirun
- `traceanalyzer` gui can use to visualize generated results



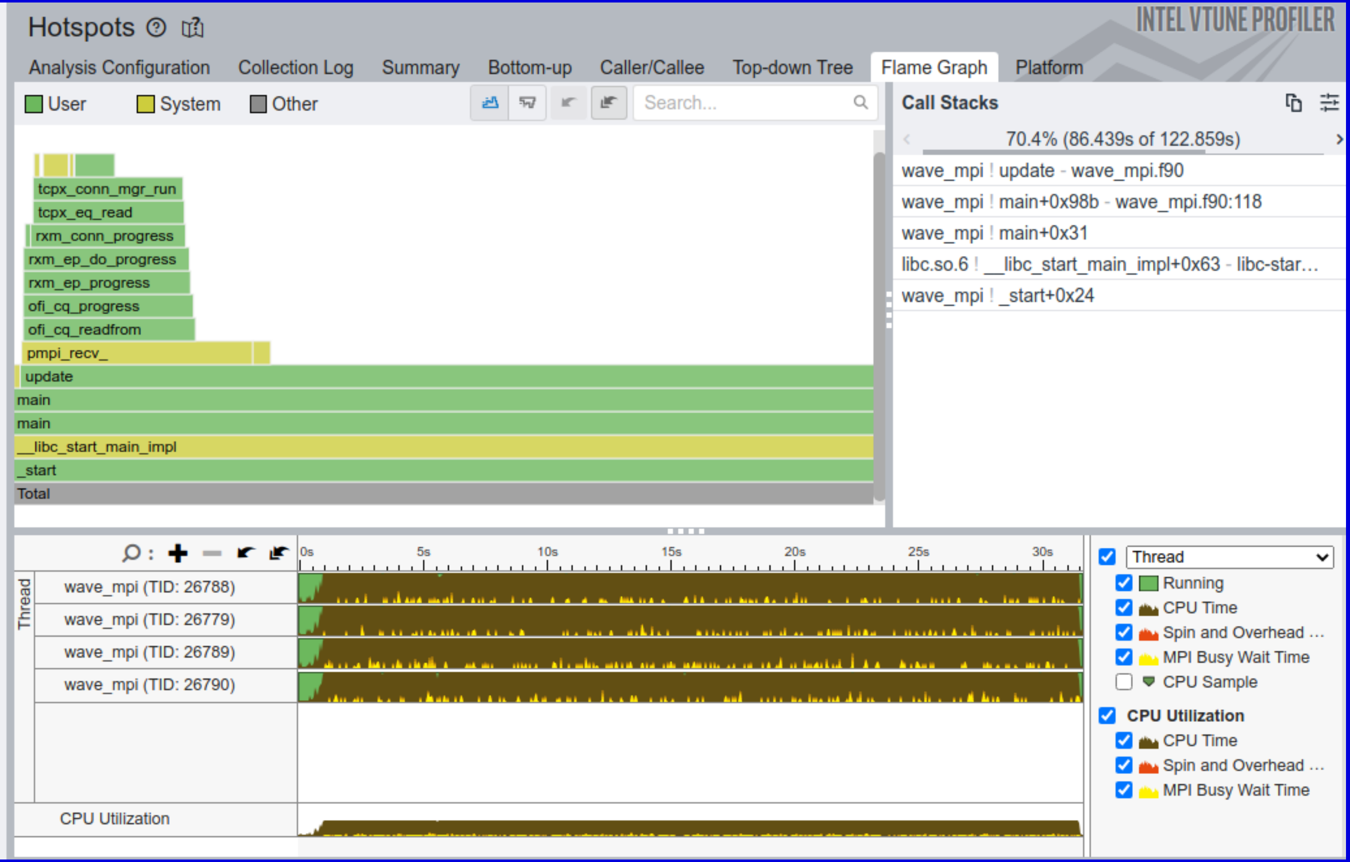

```
mpirun -trace -np 4 ./wave_mpi

traceanalyzer wave_mpi.stf
```



Intel VTune

- 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 or detailed results mapped to source code
- Multi node (MPI) profiling



Profiling Python

```
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}
.
.
.
```

Profiling R

- Use RStudio's built in `Profile > Start Profiling` menu and run the R code
- Or, enclose the function or code with `profvis` function
- Enables a user to:
  - Profile time, memory
  - Find bottlenecks

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

Thank you