

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/ldb, 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.
0x0000555555552bc in test () at test.f90:10
10      x(i) = i
(gdb) backtrace
#0  0x0000555555552bc in test () at test.f90:10
(gdb) frame 0
#0  0x0000555555552bc 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

The screenshot displays the TotalView 8.13.0-0 interface. At the top, a menu bar includes File, Edit, View, Tools, Window, and Help. Below it is a table with columns for Rank, Host, Status, and Description. A window titled 'mpirun<mat_mul_par>.0' is open, showing a toolbar with icons for Go, Halt, Kill, Restart, Next Step, Out, Run To, Record, Go Back, Prev, UnStep, Caller, and Back To Live. The main area is divided into several panes: a 'Stack Trace' pane on the left showing a list of functions with their frame pointers (FP), a 'Stack Frame' pane on the right with the message 'Thread must be stopped for frame display.', and a large code editor at the bottom. The code editor shows the source code for 'Function mat_mul_par in mat_mul_par_mpi.f90', with line 29 'call MPI_INIT(ierr)' highlighted in yellow. The code includes comments for matrix dimensions and initialization, and a loop for data initialization. At the bottom of the interface, there are tabs for 'Action Points', 'Processes', and 'Threads', and a set of navigation buttons (P-, P+, Px, T-, T+).

TotalView

The screenshot displays the TotalView 8.13.0-0 interface. At the top, a process list window shows three MPI ranks (0, 1, 2) running the program 'mat_mul_par'. The main window shows the source code of 'mat_mul_par' with a stack trace highlighting the 'call MPI_BCAST(a, n_el, MPI_REAL, MPI_COMM_WORLD, ierr)' at line 76. A variable viewer window shows the value of 'a' as a 9x1 matrix of zeros. A stack frame window shows the function 'mat_mul_par' with local variables 'a', 'b', 'b_local', 'c', and 'c_local'. The bottom panel shows action points for MPI_BCAST and MPI_SCATTER.

Process List:

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 (

Stack Trace:

```
mat_mul_par, FP=7ffa4e3bf00  
main, FP=7ffa4e3bf00  
__libc_start_main, FP=7ffa4e3bf00  
_start, FP=7ffa4e3bf00
```

Function 'mat_mul_par':

```
69 !by a SINGLE process  
70 allocate(b_local(nr_b, n2), STAT=STAT_ALLOC)  
71 b_local = 0.0; c_local = 0.0  
72  
73 !All processes need matrix A  
74 n_el = nr_a * nc_a !Total number of elements in matrix A  
75  
76 call MPI_BCAST(a, n_el, MPI_REAL, MPI_COMM_WORLD, ierr)  
77  
78  
79  
80 !Send n columns each from A to all processes (including MASTER)  
81 n2 = nc_local * nr_b !nc_local number of columns  
82  
83 call MPI_SCATTER(b, n2, MPI_REAL, MPI_COMM_WORLD, ierr)  
84  
85
```

Variable Viewer:

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

Stack Frame:

Function "mat_mul_par":
No arguments.
Local variables:
a: (real(kind=4) (4100,4100))
b: (real(kind=4) (4100,4100))
b_local: (real(kind=4), allocated)
c: (real(kind=4) (4100,4100))
c_local: (real(kind=4), allocated)

Action Points:

STOP	Process	Thread	Address
5	mat_mul_par_mpi.f90#74	mat_mul_par+0x474	
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
- Cli version results can be visualized with GUI later
- Free!

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

The screenshot displays the Intel Inspector interface. The top window is titled "Locate Memory Problems" and shows a table of detected issues. The "Problems" table has columns for ID, Type, Sources, Modul..., Object ..., and St. The "Filters" panel on the right shows severity levels (Critical, Error, Warning) and types (Invalid memory access, Memory not deallocated, Unhandled application exception) with their respective counts. Below the problems table, the "Code Location" panel shows a table with columns for Descript..., Source, Function, Module, Object Si..., Offset, and Variable. It displays two code snippets: one for a loop from line 8 to 12, and another for an allocation from line 5 to 9. The "Timeline" panel at the bottom right shows a single event labeled "start (29146)".

ID	Type	Sources	Modul...	Object ...	St.
P1	Unhandled applicati...	pthread_kill.c	libc.so.6		
P2	Invalid memory acc...	dl-vdso.h	libc.so.6		
P3	Invalid memory acc...	libc-start.c	libc.so.6		
P4	Invalid memory acc...	libc-start.c	libc.so.6		
P5	Invalid memory acc...	libc-start.c	libc.so.6		
P6	Invalid memory acc...	libc-start.c	libc.so.6		
P7	Invalid memory acc...	libc-start.c	libc.so.6		
P8	Invalid memory acc...	test.f90	test		
P9	Memory not dealloc...	libc_start_call_ma...	libc.so ...	36	

Descript...	Source	Function	Module	Object Si...	Offset	Variable
Write	test.f9... test	test	test		20	0x154d...
8						test!test - test.f
9	do i = 1, 100000					test!main
10	x(i) = i					libc.so.6!_libc_s
11	end do					libc.so.6!_libc_s
12						test!_start
Allocat...	test.f9... test	test	test		20	0x154d...
5	integer, allocatable :					test!test - test.f
6						test!main
7	allocate(x(5))					libc.so.6!_libc_s
8						libc.so.6!_libc_s
9	do i = 1, 100000					test!_start

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
```

Call graph (explanation follows)

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_

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

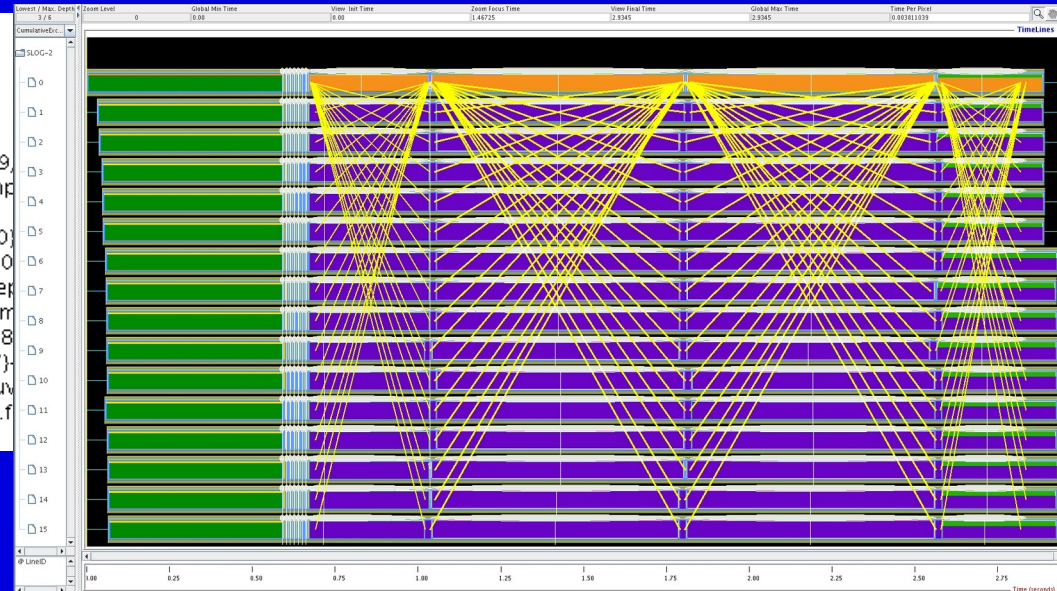
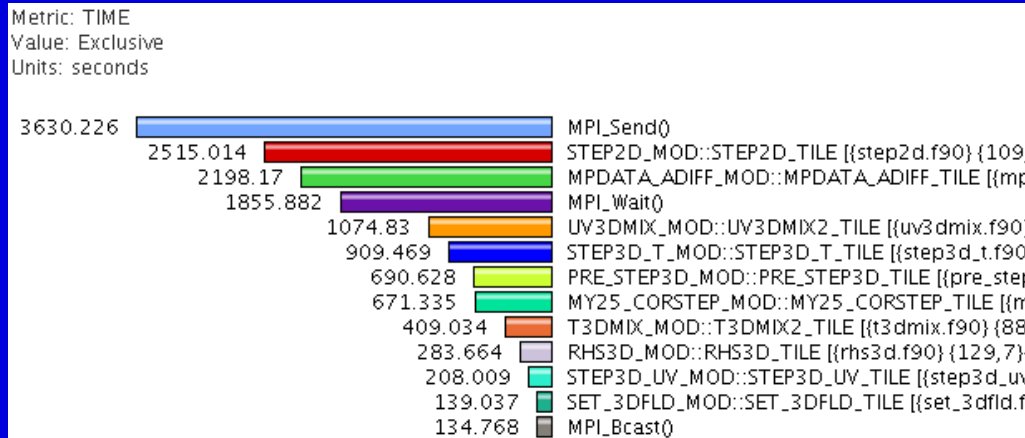
index	% time	self	children	called	name
		0.03	20.45	1/1	main [2]
[1]	99.9	0.03	20.45	1	MAIN__ [1]
		0.13	20.08	771780/771780	move_ [3]
		0.16	0.00	771897/771897	locate_ [7]

TAU

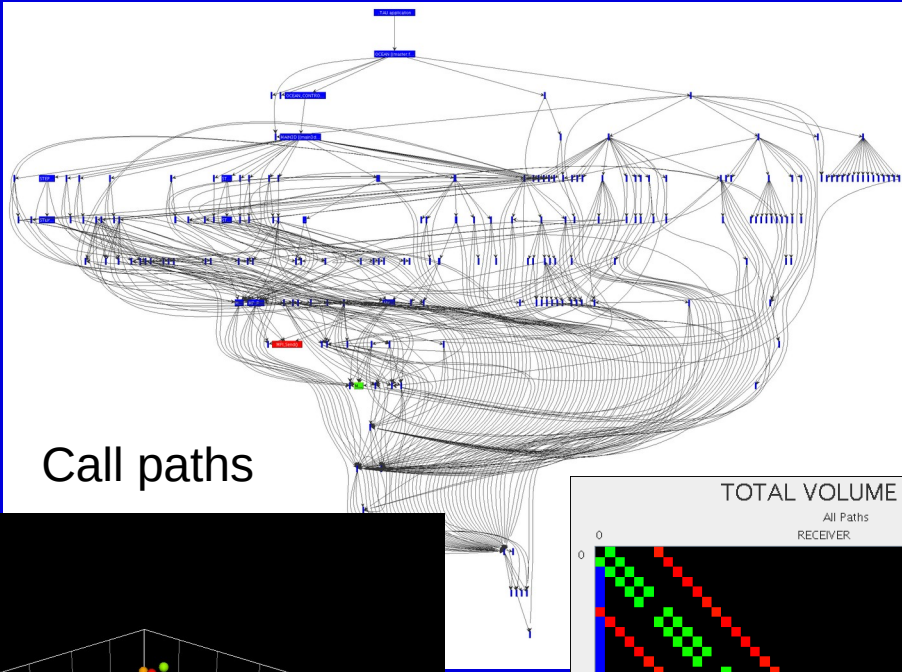
- Tuning and Analysis Utilities (20+ year project actively developed by Univ. of Oregon, LANL, Julich)
- Integrated performance toolkit
 - Instrumentation, measurement, analysis, visualization
 - Performance data management and data mining
 - Open source and free
- Works with or without recompiling code
 - 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

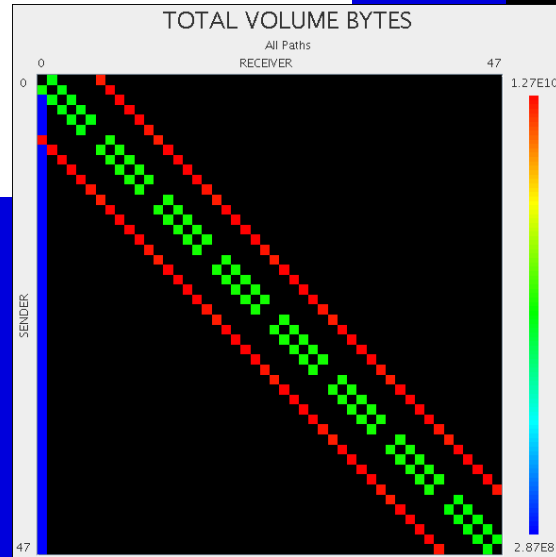
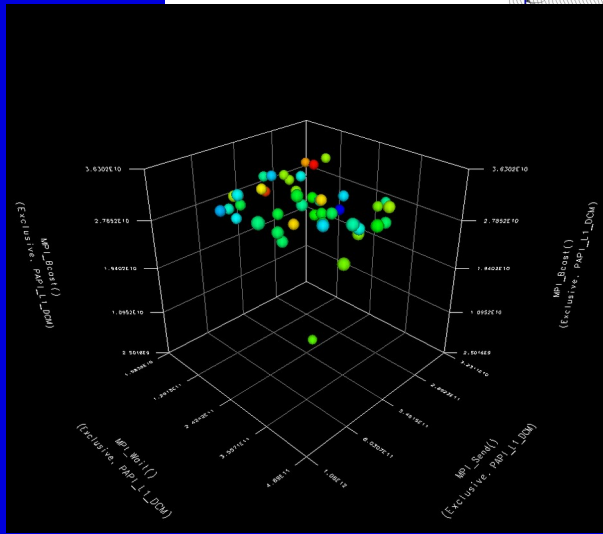
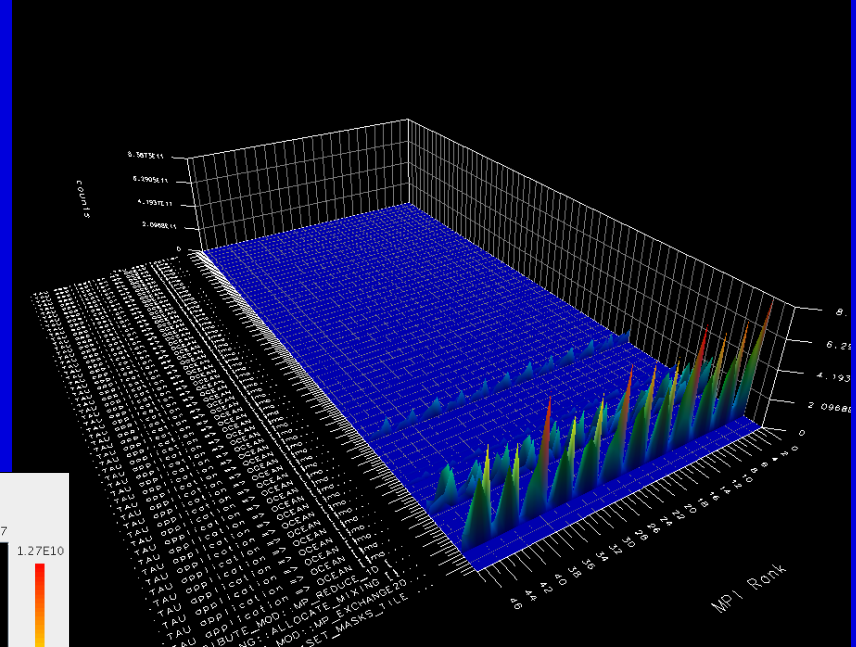
- Profile: statistical summary of all metrics measured
 - Shows how much total time & resources each call utilized
- Trace: timeline of runtime events took place
 - Shows when each event happened and where



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

The screenshot displays the Intel Advisor interface. At the top, it shows 'Elapsed time: 218.99s' and 'Vectorized' status. Below this, there are filter options: 'All Modules', 'All Sources', 'Loops And Functions', and 'All Threads'. The main view is a 'Roofline' table with columns for 'Function Call Sites and Loops', 'Performance Issues', 'CPU Time' (Total Time and Self Time), 'Type', 'Why No Vectorization?', and 'Vectorized' (Vecto... and G).

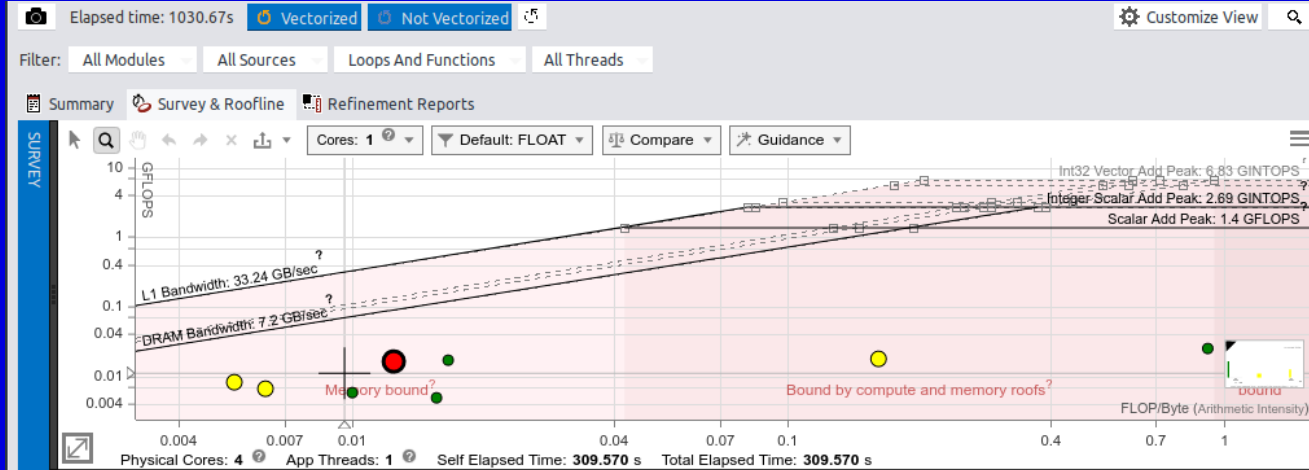
Function Call Sites and Loops	Performance Issues	CPU Time		Type	Why No Vectorization?	Vectorized
		Total Time	Self Time			
[loop in sumit at thermal2lat.f:371]	1 Data type co...	71.491s	71.491s	Scalar		
energy		61.743s	61.743s	Function		
[loop in update at thermal2lat.f:762]		134.594s	33.935s	Scalar		
_libm_expf_e7	1 Data type con...	45.533s	32.126s	Function		
_libm_error_support		13.407s	7.601s	Function		
update		144.688s	3.024s	Function		
move		217.427s	1.132s	Function		
[loop in locate at thermal2lat.f:391]	1 Data type con...	0.721s	0.721s	Vectorized (Body)		SSE2
[loop in _libm_error_support]	1 Misaligned lo...	0.672s	0.672s	Scalar		
expf		0.453s	0.453s	Function		
[loop in _unnamed_main\$ at thermal2lat.f:148]	1 Data type con...	218.943s	0.344s	Scalar		

Below the roofline table, there is a 'Source' view showing the code for the subroutine 'sumit' at line 371. The code is as follows:

```
362 c Subroutine to sum tsum vector of total rates
363 c
364   subroutine sumit(nsites9,rate,tsum)
365     integer*4 nsites9
366     real*8 tsum(nsites9+1)
367     real*4 rate(nsites9)
368     integer*4 i,j
369   c
370     tsum(1)=0.
371     do 10 i=2,nsites9+1
372       [loop in sumit at thermal2lat.f:371]
373         Scalar loop
374       tsum(i)=tsum(i-1)+rate(i-1)
375     continue
376   return
377   end
```

The 'Source' view also includes columns for 'Total Time', '%', 'Loop/Function Time', '%', and 'Traits'. The selected loop at line 371 has a total time of 12.204s and a loop/function time of 71.491s. The 'Traits' column for the selected loop is 'Type Conv...'. At the bottom right, it shows 'Selected (Total Time): 12.204s'.

Intel Advisor



Source Top Down Code Analytics Assembly Recommendations Why No Vectorization?

File: thermal2lat.f:371:sumit

Line	Source	Total Time	%	Loop/Function Time	%	Traits
362	c Subroutine to sum tsum vector of total rates					
363	c					
364	subroutine sumit(nsites9,rate,tsum)					
365	integer*4 nsites9					
366	real*8 tsum(nsites9+1)					
367	real*4 rate(nsites9)					
368	integer*4 i,j					
369	c					
370	tsum(1)=0.					
371	do 10 i=2,nsites9+1 [loop in sumit at thermal2lat.f:371] Scalar loop	43.215s		309.570s		
372	tsum(i)=tsum(i-1)+rate(i-1)	262.589s				Type Conv...
373	10 continue	3.766s				
374	c					
375	return					
376	end					
377	c-----					
Selected (Total Time):		43.215s				

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
```

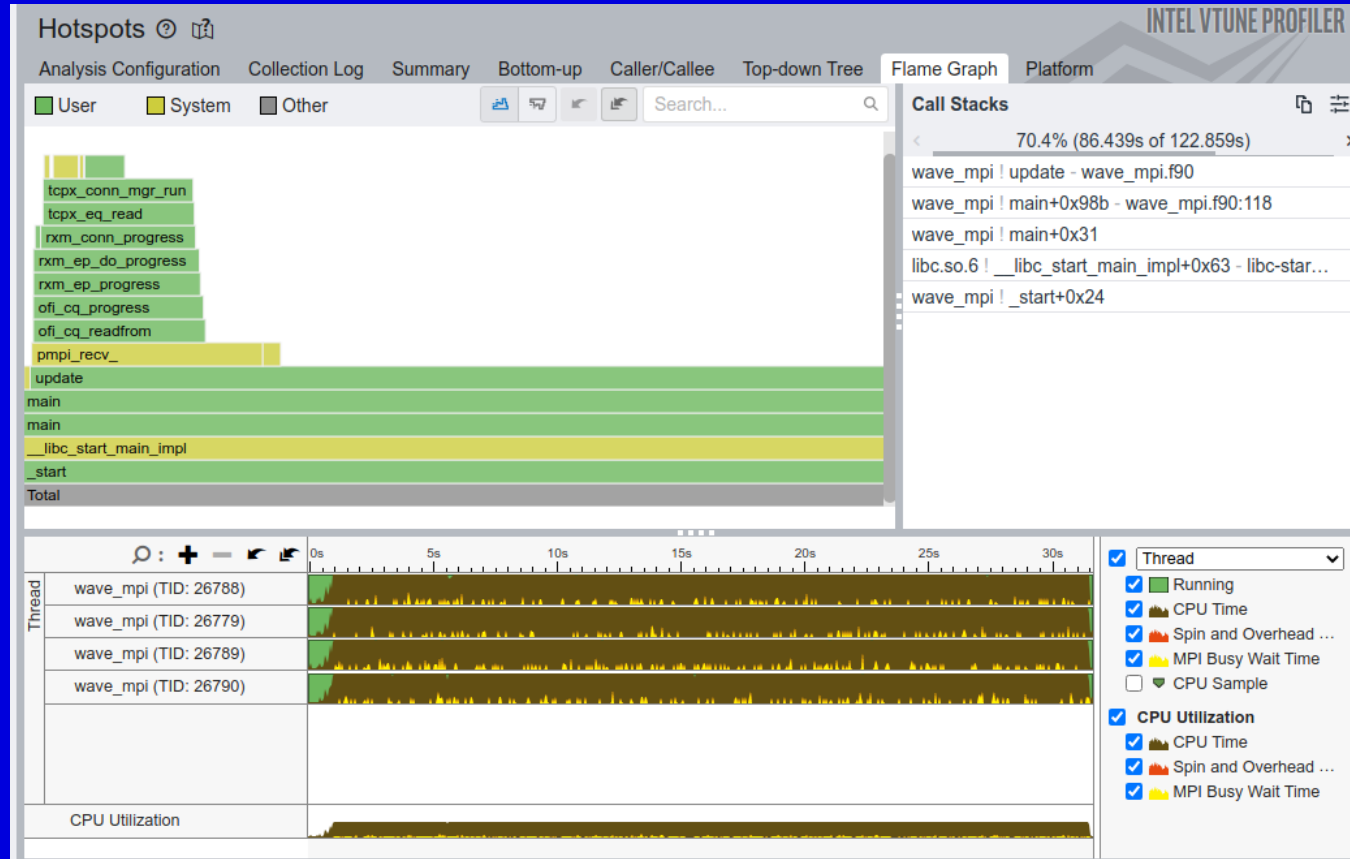
Flat Profile Load Balance Call Tree Call Graph

Children of All_Processes

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

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