Profiling with TAU

Le Yan

User Services

HPC @ LSU
Three Steps of Code Development

• Debugging
  – Make sure the code runs and yields correct results

• Profiling
  – Analyze the code to identify performance bottlenecks

• Optimization
  – Make the code run faster and/or consume less resources
Profiling

• Gather performance statistics during execution
  – Inclusive and exclusive time
  – Number of calls
• Reflects performance behavior of program entities
  – Routines
  – Loops
• Implemented through
  – Sampling: OS interrupts or hardware counters
  – Instrumentation: measurement functions
Profiling with gprof (1)

• gprof is a GNU profiling tool
• How to use
  – Compile the code with “-pg” option
  – Run the program as normal
  – Examine the profile with “gprof <executable>”
    • Or “gprof <executable> <output> | less”
Profiling with gprof (2)

Flat profile:
Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>s/call</td>
</tr>
<tr>
<td>100.00</td>
<td>5.71</td>
<td>5.71</td>
<td>1</td>
<td>5.71</td>
</tr>
<tr>
<td>0.00</td>
<td>5.71</td>
<td>0.00</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>0.00</td>
<td>5.71</td>
<td>0.00</td>
<td>2</td>
<td>0.00</td>
</tr>
</tbody>
</table>

... Call graph (explanation follows)
granularity: each sample hit covers 2 byte(s) for 0.18% of 5.71 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>100.0</td>
<td>0.00</td>
<td>5.71</td>
<td>1/1</td>
<td></td>
<td>main [3]</td>
</tr>
<tr>
<td>[2]</td>
<td>100.0</td>
<td>5.71</td>
<td>0.00</td>
<td>1/1</td>
<td>laplace_ [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>100.0</td>
<td>0.00</td>
<td>0.00</td>
<td>2/2</td>
<td>initialize_ [4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>2/2</td>
<td>set_bcs_ [5]</td>
<td></td>
</tr>
</tbody>
</table>

... Index by function name

What is TAU

• Tuning and Analysis Utilities
  – Developed at University of Oregon

• Scalable and flexible performance analysis toolkit
  – Performance profiling and tracing utilities
  – Performance data management and data mining
  – Automatic instrumentation through Program Database Toolkit (PDT)
  – Provides an instrumentation API
Availability on LONI and LSU HPC resources

• Tezpur and LONI Linux clusters
  – +tau-2.18-intel-11.1-mvapich-1.1
  – +tau-2.18-intel-11.1-mvapich2-1.4

• Philip
  – +tau-2.18-intel-11.1-mpich-1.2.7p1
How to Use

• Add the softenv key to .soft and resoft
• Compile your code with TAU compiler scripts
  – tau_f90.sh for Fortran, tau_cc.sh for C and tau_cxx.sh for C++
  – The code is instrumented automatically
• Execute the generated executable as normal
  – Profile data files: profile.x.x.x
• Analyze/visualize the profiling results with paraprof
Paraprof

- Java-based analysis and visualization tool for performance data
- “pprof” is for text based profile display
- Can work with profile data generated by other profiling tools, e.g. MPIP
- Options
  - -f <file type>: specify type of performance data
  - -m: perform runtime monitoring
  - --pack <file>: pack profile data into one file
Main Data Window
Main Data Window: Unstacked Bars
Function Data Window: Histogram
Function Data Window: Bar Chart
3D View
Individual Thread View

TAU: ParaProf: /work/lyan1/ClusterTest

Metric: Time
Value: Exclusive
Units: seconds

15.622

TAU: ParaProf: n,c,t 3,0,0 - /work/lyan1/ClusterTest

File Options Windows Help

2.267 void LAMMPS_NS::PairJCut::compute(int, int) {{pairJCut->computePair...}}
2.051 void LAMMPS_NS::Neighbor::half_bir_newton(LAMMPS_NS::...}
1.244 void LAMMPS_NS::Comm::communicate() {{comm.cpp...}}
1.013 void LAMMPS_NS::Comm::reverse_communicate() {{comm...}}
0.735 void LAMMPS_NS::FixNVE::initial_integrate(int) {{fix_nve_init...}}
0.824 MPI_Wtio
0.465 void LAMMPS_NS::FixNVE::final_integrate() {{fix_nve_call...}}
0.367 MPI_Init
0.214 MPI_Recv
0.207 int LAMMPS_NS::Neighbor::check_distance() {{neighbor...}}
0.189 MPI_Cart_create
0.189 void LAMMPS_NS::Verlet::iterate(int) {{verlet.cpp...}}
0.17 void LAMMPS_NS::Verlet::force_clear() {{verlet.cpp...}}
0.134 MPI_Send
0.101 MPI_Sendrecv
0.094 void LAMMPS_NS::FixSetForce::post_force(int) {{fix_set...}}
0.084 void LAMMPS_NS::Neighbor::bin_atoms0() {{neighbor...}}
0.084 void LAMMPS_NS::Comm::borders() {{comm.cpp...}}
0.058 void LAMMPS_NS::Comm::exchange() {{comm.cpp...}}
0.051 void LAMMPS_NS::AtomVecAtomic::unpack_reverse(int,...
0.046 int LAMMPS_NS::AtomVecAtomic::pack_comm(int, int,...
0.043 void LAMMPS_NS::Timer::stamo(int) {{timer.cpp...}}
0.043 void LAMMPS_NS::Timer::stamo() {{timer.cpp...}}
0.037 MPI_Irecv
0.025 void LAMMPS_NS::Integrate::ev_set(int) {{integrate...}}
0.025 void LAMMPS_NS::Modify::initial_integrate(int) {{modif...}}
Comparing Multiple Threads
Callpath Profile
Options for TAU Compiler Scripts

• Display available options with “tau_xxx.sh –help”

• Options
  – -optVerbose: display verbose debugging information
  – -optKeepFiles: keep intermediate files (instrumented source files)
  – -optDetectMemory: trace malloc/free calls
Keeping Intermediate Files (1)

```bash
[lyan1@poseidon2 single_file]$ 11
total 16
-rwrxr-x 1 lyan1 loniadmin 2163 Apr 17 09:23 mat_trans_alt.f90
-rw-r--r-- 1 lyan1 loniadmin 10300 Apr 17 09:50 mat_trans_alt.o

[lyan1@poseidon2 single_file]$ tau_f90.sh -optKeepFiles mat_trans_alt.f90
...
[lyan1@poseidon2 single_file]$ 11
total 1032
-rwrxr-x 1 lyan1 loniadmin 1578296 Apr 17 10:18 a.out
-rwrxr-x 1 lyan1 loniadmin 2163 Apr 17 09:23 mat_trans_alt.f90
-rw-r--r-- 1 lyan1 loniadmin 2493 Apr 17 10:18 mat_trans_alt.inst.f90
-rw-r--r-- 1 lyan1 loniadmin 10300 Apr 17 10:18 mat_trans_alt.o
-rw-r--r-- 1 lyan1 loniadmin 2019 Apr 17 10:18 mat_trans_alt.pdb
```
Keeping Intermediate Files (2)

```fortran
[lyan1@poseidon2 single_file]$ cat mat_trans_alt.inst.f90
...
! Matrix dimension
data ndim /16,12/
character(len=*), parameter :: FMT1="(12(1x,i4))"
character(len=*), parameter :: FMT2="(16(1x,i4))"

integer profiler(2) / 0, 0 /
save profiler

call TAU_PROFILE_INIT()
call TAU_PROFILE_TIMER(profiler, '
&
 &MATRIXTRANS_ALT1 [{mat_trans_alt.f90} {1,1}-{90,28}]'
call TAU_PROFILE_START(profiler)
call mpi_init(ierr)

call mpi_comm_size(mpi_comm_world,nprocs,ierr)
call mpi_comm_rank(mpi_comm_world,myrank,ierr)
...
```
Notes for Fortran Programmers

• Use `include 'mpif.h'` instead of `use mpi`

• If free format is used with `.f` files, use the `'-optPdtF95Opts=-R free'` option

• If more than one module files are used, use the `'-optPdtGnuFortranParser'` option

• If C preprocessor directive are used, use the `'-optPreProcess'` option
TAU Environment Variables

• TAU provides many environment variables
  – TAU_MAKEFILE
  – TAU_THROTTLE
  – TAU_OPTIONS
  – PROFILEDIR
  – TRACEDIR
  – ...
  – ...
  – ...
  – ...
TAU_MAKEFILE

- Different TAU makefiles corresponds to different configurations
- The default is `icpc-mpi-pdt-openmp-opari`
- There are quite a few others

```bash
[lyan1@philip1 lib]$ ls Makefile.tau-intel-11.1-mpich-1.2.7p1-*
Makefile.tau-intel-11.1-mpich-1.2.7p1-callpath-icpc-mpi-compensate-pdt-openmp
Makefile.tau-intel-11.1-mpich-1.2.7p1-callpath-icpc-mpi-pdt-openmp
Makefile.tau-intel-11.1-mpich-1.2.7p1-depthlimit-icpc-mpi-pdt-openmp
Makefile.tau-intel-11.1-mpich-1.2.7p1-icpc-mpi-compensate-pdt-openmp
Makefile.tau-intel-11.1-mpich-1.2.7p1-icpc-mpi-pdt-openmp
Makefile.tau-intel-11.1-mpich-1.2.7p1-icpc-mpi-pdt-openmp-opari
Makefile.tau-intel-11.1-mpich-1.2.7p1-icpc-pdt-openmp
Makefile.tau-intel-11.1-mpich-1.2.7p1-icpc-pdt-openmp-opari
Makefile.tau-intel-11.1-mpich-1.2.7p1-icpc-pthread-pdt-openmp
Makefile.tau-intel-11.1-mpich-1.2.7p1-param-icpc-mpi-pdt-openmp
```
TAU_CALLPATH

• Enables callpath profiling
  – Recorded callpath for each event
  – Need to set TAU_MAKEFILE to one of those with callpath in their names
• TAU_CALLPATH_DEPTH
  – Level to which callpath is recorded
  – Default is 2
  – Overhead increases with the depth of callpath
Other Environment Variables

- **TAU_THROTTLE**
  - Enable event throttling
  - Purpose: reduce profiling overhead
  - If a function executes more than $\text{TAU\_THROTTLE\_NUMCALLS}$ times and has an inclusive time per call of less than $\text{TAU\_THROTTLE\_PERCALLS}$ microseconds, then profiling of that function will be disabled after the threshold is reached.

- **PROFILEDIR**
  - Controls where the profile files are written to (the default is current directory)

- **TAU_OPTIONS**
  - Override the default instrumentation options
Selective Profiling (1)

• Instruct TAU
  – Which part(s) of the code to profile
  – How they are profiled
• -optTauSelectFile=<file>
  – The select profiling file specifies files, functions and sections that will be included or excluded in the profiling
  – Wildcards can be used
Selective Profiling (2)

[lvan1@poseidon2 src]$ echo $TAU_OPTIONS
-optVerbose -optTauSelectFile=/work/lvan1/ClusterTest/tauTests/lammps-mpi-only/src/select.tau
[lvan1@poseidon2 src]$ cat select.tau
BEGIN_INCLUDE_LIST

MPI#
mpi#
Mpi#

END_INCLUDE_LIST
Tracing (1)

- Recording of information about events during execution
  - Entering/exiting code region (function, loop, block...)
  - Thread/process interactions (send/receive message...)
- Save information in event record
  - Timestamp
  - CPU identifier
  - Event type and event-specific information
- Event trace is a time-sequenced stream of event records
Tracing (2)

• Pick the correct makefile using TAU_MAKEFILE (those with “trace” in the file name”)
• Compile with TAU compiler scripts and run the program
• Use external utilities to analyze the trace files
  – JUMPSHOT
  – VAMPIR
• Be careful: trace files can grow very big!
Not Covered

- Database management
- Phase based profiles
- Track memory and I/O
- Instrumentation API
Questions?