Performance Measurement and Analysis with \textit{gprof}

Tri-Lab Tools Workshop

Mahesh Rajan, SNL

Wednesday, March 24, 2010 @ SNL
Thursday March 25, 2010 @ LANL
Thursday July 29, 2010 @ LLNL
Objectives

- A quick overview of gprof
- Information to get started on TLCC system like SNL’s glory
- Information to get started on dawndev
- A simple example to illustrate the use of the tool and analysis of an application
**gprof**: What can it do?

A simple to use performance profiler that provides: function flat profile, call-graph profiles and the number of function calls

- Program Instrumentation through compiler flag – No source modification
- Timings are collected by statistical sampling
- Could be used with serial or parallel program
- Measurement impacts run time. Typically the program counter is looked at around 100 times per second of run time
- Mostly used as a command line tool without a GUI, but GUIs are available: [http://kprof.sourceforge.net/](http://kprof.sourceforge.net/)
Basic Information for getting started - TLCC

<table>
<thead>
<tr>
<th>Platform</th>
<th>Version</th>
<th>Setup</th>
<th>Example Dir</th>
<th>POC</th>
<th>Help-line</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLCC glory</td>
<td>2.17.50.0.6-12.el5</td>
<td>none</td>
<td>/project/tools_workshop/gprof</td>
<td>Rajan</td>
<td><a href="mailto:glory-help@sandia.gov">glory-help@sandia.gov</a></td>
</tr>
</tbody>
</table>


Limitations:
1) Beware that compiling with `-pg' adds a significant overhead to function calls – specially for small functions with very large number of calls compared to others
2) output from `gprof` gives no indication of parts of your program that are limited by I/O. i.e. it says nothing about the time the program was not running
3) No easy way to get load balance related performance information for parallel runs
4) Hardware counter information typically not available
5) May need a filter for c++ demangling for convenient viewing of the results
Basic Information for getting started - dawndev

<table>
<thead>
<tr>
<th>Platform</th>
<th>Version</th>
<th>Setup</th>
<th>Example Dir</th>
<th>POC</th>
<th>Help-line</th>
</tr>
</thead>
<tbody>
<tr>
<td>dawndev</td>
<td>TBD</td>
<td>none</td>
<td>/home/mrajan/tools_workshop</td>
<td>Rajan</td>
<td><a href="mailto:mrajan@sandia.gov">mrajan@sandia.gov</a></td>
</tr>
</tbody>
</table>

IBM Differences:

1) GMON_OUT_PREFIX environment variable does not seem to work in renaming profile output files
2) IBM Red Book suggests just linking with `-pg` to reduce overhead of function calls – needs further investigation on dawndev
3) Profile data goes to gmon.out.<mpi_rank_id> for parallel runs
4) `gprof a.out gmon.out.*` DOES NOT produce an ASCII profile output properly interpreting wild-card `*`; Behavior different from clusters running Linux (like TLCC) where the profile has the sum of the run times from all the processors
Using `gprof`

Steps to use `gprof` with your application:

1. Compile and **Link** complete program with `-pg` (gnu, Intel)
2. Run instrumented application; information about function calls and time spent is kept in memory. At program exit, this information is written to a file called `gmon.out`.
3. For parallel runs by default each process writes to same file - not good! The work-around is to set the environment variable GMON_OUT_PREFIX. Then the `gmon.out` files from each process with process ID, pid, is named `${GMON_OUT_PREFIX}.<pid>`
4. Run `gprof`. Output of `gprof` goes to stdout.
**gprof** Example

SNL Mantevo HPCCG (Conjugate Gradient Mini-app)

---

STEP 1: Edit the Makefile to add `–pg` to the compile and **link** lines

STEP 2: Build the application with `make`

STEP 3: Set the environment variable:

```bash
export GMON_OUT_PREFIX=profile_data
```

STEP 4: Run the application:

```bash
mpirun –np 16 ./test_HPCCG 100 100 100
```

you should see sixteen `profile_data.*` raw profile data files

STEP 5: run **gprof** to get profile report:

```bash
gprof ./test_HPCCG profile_data.*
```

( the output from **gprof** goes to stdout; redirect to a file)

STEP 6: View the profile report and analyze the data gathered. Be aware that step 5 produces **sum** of the run times. Both user and library functions appear in the report. There are usually quite a number of library functions of little interest.
Sample *gprof* report

Flat & Call graph profile

**Flat profile:**

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>self total</th>
<th>time</th>
<th>seconds</th>
<th>seconds calls</th>
<th>s/call</th>
<th>s/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>82.48</td>
<td>688.07</td>
<td>688.07</td>
<td>2400</td>
<td>0.29</td>
<td>0.29</td>
<td>HPC_sparsemv(HPC_Sparse_Matrix_STRUCT*, double const*, double*)</td>
<td></td>
</tr>
<tr>
<td>11.10</td>
<td>780.68</td>
<td>92.60</td>
<td>7184</td>
<td>0.01</td>
<td>0.01</td>
<td>waxpby(int, double, double const*, double, double const*, double*)</td>
<td></td>
</tr>
<tr>
<td>4.95</td>
<td>821.99</td>
<td>41.32</td>
<td>4768</td>
<td>0.01</td>
<td>0.01</td>
<td>ddot(int, double const*, double const*, double*, double&amp;)</td>
<td></td>
</tr>
</tbody>
</table>

**Call graph (explanation follows):**

granularity: each sample hit covers 2 byte(s) for 0.00% of 834.33 seconds

<table>
<thead>
<tr>
<th>index</th>
<th>% time</th>
<th>self</th>
<th>children</th>
<th>called</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00</td>
<td>822.56</td>
<td>16/16</td>
<td>main [1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td>98.6</td>
<td>0.00</td>
<td>822.56</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>688.07</td>
<td>0.00</td>
<td>2400/2400</td>
<td>HPCCG(HPC_Sparse_Matrix_STRUCT*, double const*, double*, int, double, int&amp;, double&amp;, double*) [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>92.60</td>
<td>0.00</td>
<td>7184/7184</td>
<td>HPC_sparsemv(HPC_Sparse_Matrix_STRUCT*, double const*, double*) [3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>41.32</td>
<td>0.00</td>
<td>4768/4768</td>
<td>waxpby(int, double, double const*, double, double const*, double*) [4]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.57</td>
<td>0.00</td>
<td>2400/2400</td>
<td>ddot(int, double const*, double const*, double*, double&amp;) [5]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>28768/38368</td>
<td>exchangeExternals(HPC_Sparse_Matrix_STRUCT*, double const*) [8]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>0.00</td>
<td>28768/38368</td>
<td>mytimer() [19]</td>
<td></td>
</tr>
</tbody>
</table>

**CODE Timer Output:**

********** Performance Summary (times in sec) **********

<table>
<thead>
<tr>
<th>Function</th>
<th>gprof flat</th>
<th>code time</th>
<th>code_t 16X</th>
<th>Difference</th>
<th>gprof % flat</th>
<th>code % calculated</th>
</tr>
</thead>
<tbody>
<tr>
<td>HPC_sparsemv</td>
<td>688.8</td>
<td>47.53</td>
<td>760.5</td>
<td>71.64</td>
<td>82.48</td>
<td>82.83%</td>
</tr>
<tr>
<td>waxpby</td>
<td>92.44</td>
<td>6.366</td>
<td>101.8</td>
<td>9.422</td>
<td>11.10</td>
<td>11.06%</td>
</tr>
<tr>
<td>ddot</td>
<td>41.68</td>
<td>3.142</td>
<td>50.28</td>
<td>8.604</td>
<td>4.95</td>
<td>5.80%</td>
</tr>
</tbody>
</table>

Observe that *gprof* profile times have small discrepancies to the times reported by the code. However percentages are close.
Performance Measurement and Analysis with CrayPat and Apprentice

Tri-Lab Tools Workshop

Mahesh Rajan, SNL

Wednesday, March 24, 2010 @ SNL
Thursday March 25, 2010 @ LANL
Thursday July 29, 2010 @ LLNL
Objectives

• A quick overview of Cray’s CrayPat and Apprentice tools
• Information to get started on SNL’s Red Storm, and Cray XT5 and ACES Cielo/Cray XE6
• A simple example to illustrate the use of the tool and analysis of an application
• A preview of what is covered in the afternoon session -- experience with use of CrayPat with a few applications
CrayPat: What can it do?

Helps you identify and characterize performance issues

- Automatic Program Instrumentation – No source or Makefile modification
- Can be used to measure MPI, I/O, heap, hardware counter data
- Basic infrastructure consists of:
  - pat_build, pat_hwpc/lib_hwpc, pat_report, pat_help
- Single Processor Optimization: call graph, hardware counter data, API for low level instrumentation
- Message Passing Optimization: MPI profiler, load balance information, message statistics, MPI tracing
## Basic Information for getting started – Red Storm

<table>
<thead>
<tr>
<th>Platform</th>
<th>Version</th>
<th>Setup</th>
<th>Example Dir</th>
<th>POC</th>
<th>Help-line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Storm Qual</td>
<td>3.2/4.3.2 (CrayPat /apprentice2)</td>
<td>Module load craypat/3.2 Module load apprentice2/4.3.2</td>
<td>/home/mrajan/tools_workshop</td>
<td>Rajan, Davis, Dinge</td>
<td><a href="mailto:redstorm-help@sandia.gov">redstorm-help@sandia.gov</a></td>
</tr>
<tr>
<td>Red Storm - classified</td>
<td>3.2/4.3.2 (CrayPat /apprentice2)</td>
<td>Module load craypat/3.2 Module load apprentice2/4.3.2</td>
<td>/home/mrajan_s/tools_workshop</td>
<td>Rajan, Davis, Dinge</td>
<td><a href="mailto:redstorm-help@sandia.gov">redstorm-help@sandia.gov</a></td>
</tr>
</tbody>
</table>

### Documentation: [http://docs.cray.com](http://docs.cray.com)  -- click on ‘platforms’, choose ‘XT3’, ‘XT4’

### Limitations:
1) `pat_build -g mpi -u <executable>` can bloat run time; sampling available in later release of CrayPat
2) `Pat_build -g mpi` for complex codes like Sierra can make the executable > 2GB
3) `lib_hwpc` on Red Storm may not give correct counts for quad-core Opterons
4) `Pat_build -g mpi -u < executable>` may break for complex codes like Sierra – may need to selectively instrument

### Tutorials Offered by Cray: At every SC conference, at CUG meetings and at LCI conference
Basic Information for getting started – Cray XT5, Cielo/Cray XE6

<table>
<thead>
<tr>
<th>Platform</th>
<th>Version</th>
<th>Setup</th>
<th>Example Dir</th>
<th>POC</th>
<th>Help-line</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cray XT5 (xtplogin01)</td>
<td>5.0.2.2 (xt-crayPat/apprentice2)</td>
<td>Module load xt-craypat/5.0.2.2&lt;br&gt;Module load apprentice2/5.0.2.2</td>
<td>/home/mrajan/tools_workshop</td>
<td>Rajan, Davis, Dinge</td>
<td><a href="mailto:xtp-users@sandia.gov">xtp-users@sandia.gov</a></td>
</tr>
<tr>
<td>Cray XE6 (mzlogin01)</td>
<td>5.0.2/5.1.0.14 (CrayPat/apprentice2)</td>
<td>Module load craypat/5.0.2.&lt;br&gt;Module load apprentice2/5.1.0.14</td>
<td>/home/mrajan/tools_workshop</td>
<td>Rajan, Davis, Dinge</td>
<td>TBD</td>
</tr>
</tbody>
</table>

Documentation: [http://docs.cray.com](http://docs.cray.com) -- click on ‘platforms’, choose ‘XT3’, ‘XT4’

Enhancements:
1) Sampling is available - helps lower instrumentation overhead and run time bloat
Using CrayPat

Six steps to use of CrayPat with your application:

1. Load CrayPat module
2. Build application - No makefile modification needed
3. Instrument application with pat_build
   
   % pat_build [-g group] [-u] [options] a.out

   Analysis Groups: mpi, io, heap, user function (-u) ...
4. Run instrumented application
5. Generate performance file (.ap2) with pat_report
   
   % pat_report –f ap2 [options] <.xf file>
6. Performance analysis and visualization with CrayPat and Cray Apprentice2

Apprentice2 GUI could be run on LINUX or Windows desktop
CrayPat & Apprentice Example
SNL Mantevo HPCCG (Conjugate Gradient Mini-app)

STEP 1: Load craypat Module; *module load craypat/3.2*
STEP 2: Build the application: ‘make’ gives executable ‘test_HPCCG’
STEP 3: Instrument the application:
   \[ \text{pat\_build} \ -u \ -g \text{mpi} \ \text{test\_HPCCG} \]
   (gives an instrumented executable test_HPCCG+pat)
STEP 4: Run the application:
   \[ \text{yod} \ -sz 4 \ \text{test\_HPCCG+pat} \ 100 \ 100 \ 100 \]
STEP 5: Generate ap2 or ASCI performance file
   \[ \text{pat\_report} \ -f \ ap2 \ <\text{the .xf file produced in STEP 4}> \]
   \[ \text{pat\_report} \ <\text{the .xf file produced in STEP 4}> \]
   pat_report has options: -b, -d, -s to customize the report; get from manpage
STEP 6: Visualize the performance data; *module load apprentice2/4.3.2*
   \[ \text{app2} \ <\text{.ap2 file from STEP 5}> \]

(example batch script)
```bash
#!/bin/bash
#PBS -lselect=16:dual
#PBS -lwalltime=0:10:00
#PBS -A FY092581
export PAT_RT_EXPFILE_PER_PROCESS=1
export PAT_RT_HWPC=1
export PAT_RT_SUMMARY=1
cd $PBS_O_WORKDIR
yod -VN2 -sz 16 test_HPCCG+pat 100 100 100
```

CrayPat runtime environment variables:
1) PAT_RT_SUMMARY=1 (summary); PAT_RT_SUMMARY=0 (for trace)
2) PAT_RT_EXPFILE_PER_PROCESS=1 (one file per process if running from
   NFS, Do not need if running from /scratch Lustre)
3) PAT_RT_HWPC=1 (hardware counter event set 1)
EVENT_SET 1 gives: Float Ops, Load Stores, L1 Misses, & TLB Misses
more info: man hwpc
Sample pat_report ASCI Output

Notes for table 1:

Table option:
- O profile
Options implied by table option:
- d ti%@0.05,ti,imb_ti,imb_ti%,tr -b ex,gr,fu,pe=HIDE
Table 1: Profile by Function Group and Function

<table>
<thead>
<tr>
<th>Time %</th>
<th>Time</th>
<th>Imb. Time</th>
<th>Imb.</th>
<th>Calls</th>
<th>Experiment=1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Group</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PE='HIDE'</td>
</tr>
</tbody>
</table>

100.0% | 35.307257 | -- | -- | 123578 |
|
| 99.2% | 35.008960 | -- | -- | 55216 | USER |
|
| 78.7% | 27.567933 | 0.054516 | 2.3% | 2400 | HPC_sparsemv(HPC_Sparse_Matrix_STRUCT *, const double *, double *) |
| 14.5% | 5.080704  | 0.024785 | 0.5% | 7184 | waxpby(int, double, const double *, double, const double *, double *) |
| 4.9%  | 1.711066  | 0.037586 | 2.1% | 4768 | ddot(int, const double *, const double *, double *, double &)
| 1.0%  | 0.359326  | 0.000920 | 0.3% | 16 | generate_matrix(int, int, int, HPC_Sparse_Matrix_STRUCT **, double **, double **, double **) |
| 0.7%  | 0.240166  | 0.003094 | 1.4% | 16 | make_local_matrix(HPC_Sparse_Matrix_STRUCT *) |
| 0.1%  | 0.026610  | 0.003094 | 1.4% | 16 | exchange_externals(HPC_Sparse_Matrix_STRUCT *, const double *) |

CODE Timer Output:

********** Performance Summary (times in sec) ***********

Total Time/FLOPS/MFLOPS = 34.626/1.52576e+11/4406.41.
DDOT Time/FLOPS/MFLOPS = 2.18476/9.536e+09/4364.77.
Minimum DDOT MPI_Allreduce time (over all processors) = 0.115714
Maximum DDOT MPI_Allreduce time (over all processors) = 0.433244
Average DDOT MPI_Allreduce time (over all processors) = 0.17458
WAXPBY Time/FLOPS/MFLOPS = 5.06025/1.4304e+10/2826.74.
SPARSEMV Time/FLOPS/MFLOPS = 27.3116/1.28736e+11/4713.6.

Observe: CrayPat profile times match the times reported by the code (similar colors)

Run time of instrumented code can bloat; be aware!
Table 1: Profile by Function Group and Function

| Time% | 77.6%  |
| Time  | 25.835989 |
| Imb.Time | 0.324971 |
| Imb.Time% | 1.3% |
| Calls | 2400 |
| PAPI_TLB_DM | 8.557M/sec | 221065210 misses |
| PAPI_L1_DCA | 8357.318M/sec | 215916426481 ops |
| PAPI_FP_OPS | 5571.610M/sec | 143945966653 ops |
| DC_MISS | 559.023M/sec | 14442703857 ops |
| User time | 25.836 secs | 62005472433 cycles |
| Utilization rate | 100.0% |
| HW FP Ops / Cycles | 2.32 ops/cycle |
| HW FP Ops / User time | 5571.610M/sec | 143945966653 ops | 7.3%peak |
| HW FP Ops / WCT | 5571.529M/sec |
| Computation intensity | 0.67 ops/ref |
| LD & ST per TLB miss | 976.71 ops/miss |
| LD & ST per D1 miss | 14.95 ops/miss |
| D1 cache hit ratio | 93.3% |
| % TLB misses / cycle | 0.0% |
Sample apprentice screen-shots
Afternoon Session Possible Topics

- Live demo of CrayPat and Apprentice with couple of applications
- Analysis of an application to understand performance issues
- Limitations of CrayPat and possible work-around
- Answer user specific questions