Definitions – Profiling

- Profiling
  - Recording of summary information during execution
  - inclusive, exclusive time, # calls, hardware statistics, …
  - Reflects performance behavior of program entities
  - functions, loops, basic blocks
  - user-defined “semantic” entities
  - Very good for low-cost performance assessment
  - Helps to expose performance bottlenecks and hotspots
  - Implemented through sampling: periodic OS interrupts or hardware counter traps
  - instrumentation: direct insertion of measurement code

Definitions – Tracing

- Tracing
  - Recording of information about significant points (events) during program execution
  - entering/exiting code region (function, loop, block, …)
  - thread/process interactions (e.g., send/receive message)
  - Save information in event record
    - timestamp
    - CPU identifier, thread identifier
    - Event type and event-specific information
  - Event trace is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation

Available Profiling Tools

- prof, gprof
- PAPI (profiling based on timers or on any PAPI hardware counter metric)
- Dynaprof (requires dyninst or DPCL)
- GuideView (OpenMP) (being phased out)
- Vampir (MPI)
- TAU (OpenMP, MPI, MPI/OpenMP)
- SvPablo
- HPCView
- Vprof
- Vendor specific tools (e.g., SGI IRIX perfex and ssrun, IBM tprof and trace, Cray PAT)

prof Profile of FSPX Benchmark

<table>
<thead>
<tr>
<th>time</th>
<th>cumulative</th>
<th>self</th>
<th>calls</th>
<th>mu/call</th>
<th>tot/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.71</td>
<td>18.93</td>
<td>18.93</td>
<td>6080</td>
<td>3.11</td>
<td>flux_</td>
</tr>
<tr>
<td>19.99</td>
<td>36.36</td>
<td>17.43</td>
<td>9224</td>
<td>1.91</td>
<td>prof_</td>
</tr>
<tr>
<td>7.96</td>
<td>43.56</td>
<td>7.20</td>
<td>6080</td>
<td>1.18</td>
<td>proflux_</td>
</tr>
<tr>
<td>8.26</td>
<td>50.63</td>
<td>7.43</td>
<td>6080</td>
<td>1.16</td>
<td>flux_</td>
</tr>
<tr>
<td>7.46</td>
<td>57.57</td>
<td>6.94</td>
<td>100661388</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>6.05</td>
<td>64.08</td>
<td>6.51</td>
<td>100661388</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>5.60</td>
<td>71.24</td>
<td>4.88</td>
<td>49807362</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>4.07</td>
<td>77.79</td>
<td>3.55</td>
<td>62202877</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>2.44</td>
<td>79.92</td>
<td>2.13</td>
<td>37371904</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>1.47</td>
<td>81.38</td>
<td>1.44</td>
<td>37371904</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>1.43</td>
<td>82.63</td>
<td>1.13</td>
<td>37371904</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>1.07</td>
<td>83.56</td>
<td>0.93</td>
<td>24903681</td>
<td>0.00</td>
<td>1.18</td>
</tr>
<tr>
<td>0.89</td>
<td>84.34</td>
<td>0.78</td>
<td>37371904</td>
<td>0.00</td>
<td>1.18</td>
</tr>
</tbody>
</table>

prof, gprof

- Available on many Unix platforms (e.g., IBM AIX, Sun Solaris, HP/Compaq Tru64)
- Produces “flat” profile (prof) or “call graph” profile (gprof)
- Compile with special flag or set environment variable or rewrite executable to enable instrumentation and produce profile data file
- Collect profile data by periodically sampling the program counter during execution and/or calling monitoring routines
- See
  - man prof
  - man gprof
IBM Profiling Utilities

- In addition to man pages, see the AIX 5 Performance Management Guide
- prof
- gprof
- tprof

SGI Profiling Utilities

- ssrun
  - Collects Speedshop and Workshop performance data
  - Types of experiments: totaltime, usertime, pcsamp, ideal, hardware counter
- prof
  - Analyzes and displays SpeedShop performance data generated by ssrun
- perfex
  - Runs program and collects hardware counter data

HP/Compaq Profiling Utilities

- prof
  - Displays profiling data from -p or pixie
- pixie
  - Instruction-counting profiler
- gprof
  - Displays profiling data from -pg or hiprof
- hiprof
  - Creates instrumented version of program for call-graph profiling
- uprofile
  - Profiles a program with Alpha on-chip performance counters

Overview of PAPI

- Performance Application Programming Interface
- The purpose of the PAPI project is to design, standardize and implement a portable and efficient API to access the hardware performance monitor counters found on most modern microprocessors.
- Parallel Tools Consortium project
- Being installed and supported at the DoD HPC Centers as part of PET CE004

PAPI Counter Interfaces

- PAPI provides three interfaces to the underlying counter hardware:
  1. The low level interface manages hardware events in user defined groups called EventSets.
  2. The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.
  3. Graphical tools to visualize information.

PAPI Implementation
PAPI Preset Events

• Proposed standard set of events deemed most relevant for application performance tuning
• Defined in papiStdEventDefs.h
• Mapped to native events on a given platform
  – Run tests/avail to see list of PAPI preset events available on a platform

High-level Interface

• Meant for application programmers wanting coarse-grained measurements
• Not thread safe
• Calls the lower level API
• Allows only PAPI preset events
• Easier to use and less setup (additional code) than low-level

High-level API

• C interface
  PAPI_start_counters
  PAPI_read_counters
  PAPI_stop_counters
  PAPI_accum_conters
  PAPI_num_conters
  PAPI_flops

• Fortran interface
  PAPIF_start_counters
  PAPIF_read_counters
  PAPIF_stop_counters
  PAPIF_accum_conters
  PAPIF_num_conters
  PAPIF_flops

PAPI_flops

• int PAPI_flops(float *real_time, float *proc_time,
  long_long *flpins, float *mflops)
  – Only two calls needed, PAPI_flops before and after the code you want to monitor
  – real_time is the wall-clocktime between the two calls
  – proc_time is the “virtual” time or time the process was actually executing between the two calls (not as fine grained as real_time but better for longer measurements)
  – flpins is the total floating point instructions executed between the two calls
  – mflops is the Mflop/s rating between the two calls

Low-level Interface

• Increased efficiency and functionality over the high level PAPI interface
• About 40 functions
• Obtain information about the executable and the hardware
• Thread-safe
• Fully programmable
• Callbacks on counter overflow

Callbacks on Counter Overflow

• PAPI provides the ability to call user-defined handlers when a specified event exceeds a specified threshold.
• For systems that do not support counter overflow at the OS level, PAPI sets up a high resolution interval timer and installs a timer interrupt handler.
Statistical Profiling

- PAPI provides support for execution profiling based on any counter event.
- PAPI_profil() creates a histogram of overflow counts for a specified region of the application code.

PAPI_profil

```c
int PAPI_profil(unsigned short * buf, unsigned int bufsiz, unsigned long offset, unsigned scale, int EventSet, int EventCode, int threshold, int flags)
```

- `buf` – buffer of `bufsiz` bytes in which the histogram counts are stored
- `offset` – start address of the region to be profiled
- `scale` – contraction factor that indicates how much smaller the histogram buffer is than the region to be profiled

What is DynaProf?

- A portable tool to instrument a running executable with Probes that monitor application performance.
- Simple command line interface.
- Open Source Software
- A work in progress…

DynaProf Methodology

- Make collection of run-time performance data easy by:
  - Avoiding instrumentation and recompilation
  - Using the same tool with different probes
  - Providing useful and meaningful probe data
  - Providing different kinds of probes
  - Allowing custom probes

Why the “Dyna”?

- Instrumentation is selectively inserted directly into the program’s address space.
- Why is this a better way?
  - No perturbation of compiler optimizations
  - Complete language independence
  - Multiple Insert/Remove instrumentation cycles

DynaProf Design

- GUI, command line & script driven user interface
- Uses GNU readline for command line editing and command completion.
- Instrumentation is done using:
  - Dyninst on Linux, Solaris and IRIX
  - DPCL on AIX
DynaProf Commands

load <executable>
list [module pattern]
use <probe> [probe args]
instr module <module> [probe args]
instr function <module> <function> [probe args]
stop
continue
run [args]
info
unload

DynaProf Probes

- papiprobe
- wallclockprobe
- perfometerprobe

DynaProf Probe Design

- Can be written in any compiled language
- Probes export 3 functions with a standardized interface.
- Easy to roll your own (<1 day)
- Supports separate probes for MPI/OpenMP/Pthreads

Future development

- GUI development
- Additional probes
  - Perfex probe
  - Vprof probe
  - TAU probe
- Better support for parallel applications

Perfometer

- Application is instrumented with PAPI
  - callperfometer()
  - call_mark_perfometer(int color, char *label)
- Application is started. At the call to perfometer, a signal handler and a timer are set up to collect and send the information to a Java applet containing the graphical view.
- Sections of code that are of interest can be designated with specific colors.
- Real-time display or trace file
Perfometer Display

Machine info

Process & Real time

Flop/s Rate

Flop/s Min/Max

Perfometer Parallel Interface

KAP/Pro Toolset

- http://www.kai.com/
- Set of tools for developing parallel scientific software using OpenMP
- Components
  - Guide OpenMP compiler
  - Assure OpenMP debugger
  - GuideView performance analyzer
  - Utilities for translating older directives to OpenMP
- Licensed at ERDC MSRC and ARL MSRC
- Being phased out by Intel/KSL

GuideView

- Intuitive, color-coded display of parallel performance bottlenecks
- Regions of code are identified where improving local performance will have the greatest impact on overall performance.
- Configuration file Gvproperties.txt controls GUI features (fonts, colors, window sizes and locations, etc.)
- Online help system under Help menu
- Use kmp_set_parallel_name to name parallel regions so that name will be displayed by GuideView

Guide Instrumentation Options

-WGnoopenmp
  - Enable profiling but no OpenMP
-WGprof
  - Activates profiling for Vampir and GuideView; implies -WGstats
-WGprof_leafprune<integer>
  - Sets the minimum size of procedures to retain in Vampir or GuideView profiles to <integer> lines
  - Use to reduce instrumentation overhead and tracefile size

GuideView – OpenMP Profile

SWEEP3D run on 16 MPI process by 4 OpenMP Threads
- Unsorted view shows different performance for each process
- Sorted by overhead to find important processes
Vampir

- Primarily a tracing tool but also generates and displays profiling statistics
- Version 3 will support OpenMP and mixed MPI/OpenMP
- Version 3 will use PAPI to access hardware counter data
- Licensed at ERDC MSRC, ARL MSRC, and ARSC

Vampir Statistics Display (also available in text form)

Vampir: Timeline Diagram

- Functions organized into groups
- Coloring by group
- Message lines can be colored by tag or size

- Information about states, messages, collective, and I/O operations available by clicking on the representation

Vampir: Profile Statistics Displays

- Aggregated profiling information: execution time, # calls, inclusive/exclusive
- Available for all/any group (activity)
- Available for all routines (symbols)
- Available for any trace part (select in timeline diagram)
Vampir: Communication Statistics Displays

- Bytes sent/received for collective operations
- Message length statistics
  - Available for any trace part
- Byte and message count, min/max/avg message length and min/max/avg bandwidth for each process pair

Vampir: Other Features

- Dynamic global call graph tree
- Parallelism display
- Powerful filtering and trace comparison features
- All diagrams highly customizable (through context menus)

Vampir: Process Displays

- Activity chart
- Call tree
- Timeline
- For all selected processes in the global displays

Vampir (NAS Parallel Benchmark – LU)

- Timeline display
- Callgraph display
- Parallelism display
- Communications display

Vampir v3.x: Hardware Counter Data

- Counter Timeline Display

TAU

- Tuning and Analysis Utilities
  - http://www.cs.uoregon.edu/research/paracomp/tau/
- Portable profiling and tracing toolkit for performance analysis of parallel programs
  - Fortran 77/90, C, C++, Java
  - OpenMP, Pthreads, MPI, mixed mode
- In use at DOE ASCI Labs and at NCSA
- Being installed and supported at DoD HPC Centers as part of PET CE002
TAU Instrumentation

- Manually using TAU instrumentation API
- Automatically using
  - Program Database Toolkit (PDT)
  - MPI profiling library
  - Opari OpenMP rewriting tool
- Uses PAPI to access hardware counter data

PDT Components

- Language front end
  - Edison Design Group (EDG): C, C++
  - Mutek Solutions Ltd.: F77, F90
  - creates an intermediate-language (IL) tree
- IL Analyzer
  - processes the intermediate language (IL) tree
  - creates “program database” (PDB) formatted file
- DUCTAPE (Bernd Mohr, ZAM, Germany)
  - C++ program Database Utilities and Conversion Tools Application Environment
  - processes and merges PDB files
  - C++ library to access the PDB for PDT applications

OPARI: Basic Usage (f90)

- Reset OPARI state information
  - `rm -f opari.rc`
- Call OPARI for each input source file
  - `opari file1.f90 ...
  - `opari fileN.f90`
- Generate OPARI runtime table, compile it with ANSI C
  - `opari -table opari.tab.c
  - cc -c opari.tab.c`
- Compile modified files `*.mod.f90 using
  - OpenMP
- Link the resulting object files, the OPARI runtime table `opari.tab.o` and the TAU POMP RTL

Program Database Toolkit (PDT)

- Program code analysis framework for developing source-based tools
- High-level interface to source code information
- Integrated toolkit for source code parsing, database creation, and database query
  - commercial grade front end parsers
  - portable IL analyzer, database format, and access API
  - open software approach for tool development
- Target and integrate multiple source languages
- Use in TAU to build automated performance instrumentation tools

PDT Status

- Program Database Toolkit (Version 2.2, web download)
  - EDG C++ front end (Version 2.45.2)
  - Mutek Fortran 90 front end (Version 2.4.1)
  - C++ and Fortran 90 IL Analyzer
  - DUCTAPE library
  - Standard C++ system header files (KCC Version 4.0f)
- PDT-constructed tools
  - TAU instrumentor (C/C++/F90)
  - Program analysis support for SILOON and CHASM
- Platforms
  - SGI, IBM, Compaq, SUN, HP, Linus (IA32/IA64), Apple, Windows, Cray T3E, Hitachi
**TAU Analysis**

- Profile analysis
  - **pprof**
    - parallel profiler with text-based display
  - **racy**
    - graphical interface to pprof (Tcl/Tk)
  - **jracy**
    - Java implementation of Racy

- Trace analysis and visualization
  - Trace merging and clock adjustment (if necessary)
  - Trace format conversion (ALOG, SDDF, Vampir)
  - **Vampir (Pallas)** trace visualization
  - **Paraver (CEPBA)** trace visualization

**TAU Pprof Display**

![TAU Pprof Display](image)

**jracy (NAS Parallel Benchmark – LU)**

![jracy (NAS Parallel Benchmark – LU)](image)

**1-Level Callpath Implementation in TAU**

- TAU maintains a performance event (routine) callstack
- Profiled routine (child) looks in callstack for parent
  - Previous profiled performance event is the parent
  - A **callpath profile structure** created first time parent calls
  - TAU records parent in a **callgraphmap** for child
  - String representing 1-level callpath used as its key
    - "a( )=>b( )": name for time spent in "b" when called by "a"
  - Map returns pointer to callpath profile structure
  - 1-level callpath is profiled using this profiling data
- Build upon TAU’s performance mapping technology
- Measurement is independent of instrumentation
- Use **PROFILECALLPATH** to configure TAU

**SvPablo**

- **Sourceview Pablo**
- **http://www-pablo.cs.uiuc.edu/Project/SVPablo/SVPabloOverview.htm**
  - Collects profile data and maps it to constructs in the original source code
  - Automatic and interactive instrumentation of Fortran 77/90, C, MPI, OpenMP
  - Uses PAPI to access hardware counter data
  - In use at DOE sites and at NCSA
**SvPablo**

![SvPablo](Image)

**HPCView**

- Combines multiple sets of program profile data
- Correlates profile data with source code
- Computes derived performance metrics
- Generates browsable database
- Uses ssrun, uprofile, and PAPI to collect profile data
- In use at DOE ASCI Labs and at NCSA

**VProf**

- Visual Profiler
- Generate profiling data sorted by file, function, line
- Display using vprof GUI or cprof command-line utility
- Ported to Linux/IA-32/IA-64, AIX/RS6000, Linux/Alpha
- In use at DOE ASCI sites and at NCSA

**vprof**

![vprof](Image)