Outline

› Performance Analysis
  › Background
  › PAPI
  › KOJAK
  › TAU
  › IPM
  › Conclusions
Performance Analysis

It is typically much more difficult to debug and tune parallel programs

Programmers often have no idea where to begin searching for possible bottlenecks

A tool that allows the programmer to get a quick overview of the program’s execution can aid the programmer in beginning this search.

Basic Tuning Process

Select “best” compiler flags
Select/interface with “best” libraries
Measure
Validate
Hand-tune (routine/loop-level tuning)
… iterate

Observation: The best way to improve parallel performance is often still to simply improve sequential performance!
Performance Analysis in Practice

Observation: many application developers don’t use performance tools at all (or rarely)

Why?
Learning curve can be steep
Results can be difficult to understand
Investment (time) can be substantial
Maturity/availability of various tools
Not everyone is a computer scientist
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
No temporal context

Profiling

There are different levels of profiling data
  › Whole program data
    • Summary information for the entire program run
  › Routine level
    • Different profiles for each routine called
    • Data can be inclusive or exclusive
  › Basic block level/loop level
    • Data is broken up for each program basic block
  › User directed block level
    • User defines a block using an available API
  › Statement level
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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

```
1 master
2 slave
3 ...
```

```
58 A ENTER 1
60 B ENTER 2
62 A SEND B
64 A EXIT 1
68 B RECV A
69 B EXIT 2
...```

![Diagram showing master and slave interactions](image-url)
PAPI

› Performance API
› Provides a consistent interface to performance hardware counters
› Generally provides profile data
› Available for most processors and OSs
› Can be used by both end users and higher level tools developers
› Not specifically aimed at parallel applications
› http://icl.cs.utk.edu/papi/

The Library Interface

PAPI provides two APIs to access the underlying counter hardware:
The low level interface manages hardware events in user defined groups called EventSets.
The high level interface simply provides the ability to start, stop and read the counters for a specified list of events.
**PAPI Implementation**

- **Portable Layer**
  - PAPI Low Level
  - PAPI High Level

- **Machine Specific Layer**
  - PAPI Machine Dependent Substrate
  - Kernel Extension
  - Operating System
  - Hardware Performance Counters

**Tools**

**High-level Interface**

- Meant for application programmers wanting coarse-grained measurements
- Calls the lower level API
- Allows only PAPI preset events
- Easier to use and less setup (additional code) than low-level
Low-level Interface

Increased efficiency and functionality over the high level PAPI interface
Approximately 56 functions Supports both presets and native events

PAPI in a Nutshell

› Provides standard interface to hardware counters
› High level interface appropriate for users
› Low level interface more appropriate for higher level tools developers
› Runs under MS Windows, but provides limited functionality
KOJAK

› Kit for Objective Judgment and Knowledge-based detection of Performance Bottlenecks
› Aiming at the development of a generic automatic performance analysis environment for parallel programs
› Can generate profile and trace data for parallel executions

KOJAK
Joint open-source project between
Forschungszentrum Jülich, Germany
University of Tennessee, USA
Automatic performance analysis of parallel applications
MPI and/or OpenMP
New: MPI-2 and SHMEM one-sided communication
URL
http://www.fz-juelich.de/zam/kojak/
http://icl.cs.utk.edu/kojak/
Contact
kojak@cs.utk.edu
Automatic trace analysis
Purely manual trace analysis
Limited coverage
Even for small node counts

Idea
Automatic pattern search
Data distillation

Analysis process
**Additional features**

Performance algebra to compare, integrate, and summarize performance experiments

Holistic hardware-counter analysis
- **Collect** event traces with (related) sets of hardware counter measurements (using PAPI)
- **Structure** analyses of hardware counter metrics in configurable hierarchies
- **Integrate** multiple experiments to complete comprehensive metric hierarchies

**KOJAK in a Nutshell**

- A suite of tools for generating and analyzing both profile and trace data
- Primary focus is automatic performance analysis
- Works for both MPI and OpenMP codes
- Aimed towards tuning parallel codes
- Interoperable with other tools
**TAU Performance System**

Tuning and Analysis Utilities (12+ year project effort)

Performance system framework for scalable parallel and distributed high-performance computing

Targets a general complex system computation model

- nodes / contexts / threads
- Multi-level: system / software / parallelism
- Measurement and analysis abstraction

Integrated toolkit for performance instrumentation, measurement, analysis, and visualization

- Portable performance profiling and tracing facility
- Open software approach with technology integration

University of Oregon, Forschungszentrum Jülich, LANL

---

**TAU Instrumentation**

Flexible instrumentation mechanisms at multiple levels

**Source code**

- manual (TAU API, TAU Component API)
- automatic
  - C, C++, F77/90/95 (Program Database Toolkit (PDT))
  - OpenMP (directive rewriting (Opari), POMP spec)

**Object code**

- pre-instrumented libraries (e.g., MPI using PMPI)
  - statically-linked and dynamically-linked

**Executable code**

- dynamic instrumentation (pre-execution) (DynInstAPI)
- virtual machine instrumentation (e.g., Java using JVMPI)

Proxy Components
Program Database Toolkit (PDT)

Program code analysis framework
  develop 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
Multiple source languages
Implement automatic performance instrumentation tools
  tau_instrumentor

tau_reduce: Rule-Based Overhead Analysis

Analyze the performance data to determine events with high (relative) overhead performance measurements
Create a select list for excluding those events
Rule grammar (used in tau_reduce tool)

[GroupName:] Field Operator Number
GroupName indicates rule applies to events in group
Field is a event metric attribute (from profile statistics)
  numcalls, numsubs, percent, usec, cumusec, count [PAPI], totalcount, stdev, usecs/call, counts/call
Operator is one of >, <, or =
Number is any number
Compound rules possible using & between simple rules
Example Rules

#Exclude all events that are members of TAU_USER
#and use less than 1000 microseconds
   TAU_USER:usec < 1000

#Exclude all events that have less than 100
#microseconds and are called only once
   usec < 1000 & numcalls = 1

#Exclude all events that have less than 1000 usecs per
#call OR have a (total inclusive) percent less than 5
   usecs/call < 1000
   percent < 5

Scientific notation can be used
   usec>1000 & numcalls>400000 & usecs/call<30 & percent>25

Compensation of Instrumentation Overhead

Runtime estimation of a single timer overhead
Evaluation of number of timer calls along a calling path
Compensation by subtracting timer overhead
Recalculations of performance metrics to improve the accuracy of measurements
Configure TAU with –COMPENSATE configuration option
TAU Analysis

Parallel profile analysis

*Pprof*
parallel profiler with text-based display

*ParaProf*
Graphical, scalable, parallel profile analysis and display

Trace analysis and visualization
Trace merging and clock adjustment (if necessary)
Trace format conversion (ALOG, SDDF, VTF, Paraver)
Trace visualization using *Vampir* (Pallas/Intel)

---

**Pprof Output (NAS Parallel Benchmark – LU)**

Intel Quad
P11 Xeon
F90 +
MPICH

Profile
- **Node**
- **Context**
- **Thread**

Events
- **code**
- **MPI**
ParaProf (NAS Parallel Benchmark – LU)

node, context, thread

Global profiles

Routine profile across all nodes

Event legend

Individual profile

Intel Trace Analyzer/Vampir
Trace Visualize

Visualization and Analysis of MPI Programs

Originally developed by Forschungszentrum Jülich

Current development by Technical University Dresden, Germany

Distributed by Intel®

http://www.intel.com/software/products/cluster/tanalyzer
TAU + Vampir (NAS Parallel Benchmark – LU)

Timeline display
Callgraph display
Parallelism display
Communications display

PETSc ex19 (Tracing)

Commonly seen communication behavior
TAU’s EVH1 Execution Trace in "Vampir"

Conclusions

› Performance analysis and tuning is very important for getting the most out of HPC systems
› This is an iterative process
› Easy to use tools are required
› Collecting data is easy, analyzing data is hard
Debugging

Debugging serial programs is hard
Debugging parallel programs is harder
  › Race conditions/non-determinism
  › Different processes doing different things
  › Print statements don’t work as expected

Tools are needed to assist in the debugging process
Debugging

A lot of errors in parallel programs have nothing to do with the parallelism
› Logic errors in the algorithm
Still, introducing parallelism can lead to additional errors
› Deadlock in MPI programs
› Data corruption in OpenMP programs
Memory errors make up for the large majority of bugs

Common bugs

Common bugs in MPI programs:
› Deadlock – No progress is possible
  • Cycles of blocking sends
  • Mixing ptp and collective communication
  • Mismatched tags
  • Mismatched source/destination pairs
› Overwriting buffers/trashing memory
  • Mismatched datatypes
› Non-determinism
› Thread support level
Common OpenMP bugs

Often cross subroutine boundaries
Are often timing dependent
Can be intermittent
Are sometimes nearly imperceptible
Are difficult to reproduce
Can be masked by debugging probes

Common OpenMP bugs

Not sharing variables that should be
Sharing variables that shouldn’t be
Not synchronizing updates of shared variables
Data dependencies (e.g. loop iterations)
Race conditions
Deadlocks
TotalView

Etnus TotalView

http://www.etnus.com

Version 7.1

Platforms: IBM, SGI, Sun, HP, AMD, Intel, Mac, Cray XT3

Languages/Libraries: Pthreads, C, C++, Fortran 77/90, MPI, OpenMP, and mixed language

Features

› Graphical User Interface
› Command-Line Interface
› Handles multiprocess and/or multithreaded programs
› Automatic process acquisition
› Capability of attaching to running processes
› Can debug code not compiled with –g (but with reduced capabilities)
Features (cont)

› Load and examine core files
› Breakpoint and evaluation points
› Examine and change data
› Signal handling
› Message state display
› Memory debugging

Special Features for Multiprocess Programs

› Separate window for each process
› Sharing of breakpoints among processes
› Control of process groups for SPMD or MPMD programs
› Multiprocess barrier breakpoints
› Single-stepping of process groups
› Handles multiple symbol tables if more than one executable
› Groups of threads can be controlled independently of other threads (even within the same process)
**Controlling Processes and Threads**

› Automatically attach to processes: If your program creates processes and/or threads, TotalView automatically attaches to them, making their symbol tables available and allowing you to manipulate their execution.

› Start and stop execution: You can start, stop, resume, delete restart, and even reload recompiled versions of your program.

› Attach to existing processes: You can examine processes not yet under TotalView’s control.

› Examine core files: You can load a core file and examine it the same way as any other executable. Or you can load a core file dynamically.

---

**Controlling Processes and Threads (cont)**

› Single step your program: You can step through your program one instruction at a time or step over function calls. You can tell TotalView to execute up to a certain source code line or through a function and then resume single-stepping. TotalView supports process, process group, and thread-level single stepping.

› Signal handling: You can change the way a program handles signals. You can also indicate how TotalView handles signals.
**Action Points**

You can set, delete, suppress, unsuppress, enable, and disable action points at the source level

- **Breakpoints:** Stop execution when a statement or instruction executes
- **Barrier Breakpoints:** Hold other threads until all threads have reached a particular statement or instruction
- **Conditional Breakpoints:** Only perform an action if a code fragment is satisfied
- **Evaluation points:** Execute code you create at a statement or instruction
- **Watchpoints:** Monitor when changes occur to a variable’s value

---

**Examining and manipulating data**

- **Diving:** You can obtain additional information about almost everything displayed by TotalView by *diving-in* on it
- **You can change a variables type or value:** May be useful in controlling the way data is displayed
- **Laminated variables:** Display the value of a variable across multiple processes and multiple threads
- **Examine array data:** You can filter array data, sort array data, or even tell TotalView to display statistics about an array’s contents
Visualization

› Array data: You can graphically view array data
› Call trees: You can dynamically view the sequence of calls associated with any current routine
› Message queue graph: Visually display the processes that are linked together by sending and receiving messages

Memory Debugging

› Detect memory leaks
› See if you exceed array boundaries
› Compare data during different points during execution
› Track allocation/deallocation
   › Dangling pointers
   › Deallocate wrong address
MPI_CHECK

Tool developed at Iowa State University for debugging MPI programs written in free or fixed format Fortran 90 and Fortran 77
You can download your own free copy of the software and license at [http://andrew.ait.iastate.edu/HPC/MPI-CHECK.htm](http://andrew.ait.iastate.edu/HPC/MPI-CHECK.htm)
MPI-CHECK does both compile-time and runtime error checking
Does not appear to be an active project

Compile Time Error Checking

Checks for consistency in the data type of each argument
Checks the number of arguments
Checks the little used intent of each argument
Run-Time Error Checking

Buffer data type inconsistency
This error is flagged if the Fortran data type of the send or receive buffer of an MPI send or receive call is inconsistent with the declared datatype in the MPI call

Buffer out of bounds
This error is flagged if either the starting or ending address of a send or receive buffer is outside the declared bounds of the buffer

Improper placement of MPI_Init or MPI_Finalize

Run-Time Error Checking

Illegal message length
Invalid MPI Rank
Actual or potential deadlock
Any cycle of blocking send calls creates a potential for deadlock. While this deadlock may not be manifest on all machines, MPI-CHECK will detect if the potential for deadlock exists.
Using MPI-CHECK

Programs are compiled the same way as normal, except mpicheck is the first command on the command line:

\[ \text{f90} \quad -o \quad \text{a.out} \quad -O3 \quad \text{main.f90} \quad \text{sub1.f90} \quad \text{sub2.f90} \quad -lmpi \]

Becomes

\[ \text{mpicheck f90} \quad -o \quad \text{a.out} \quad -O3 \quad \text{main.f90} \quad \text{sub1.f90} \quad \text{sub2.f90} \quad -lmpi \]

Source files are required, rather than object files

Programs are ran just as without MPI-CHECK

Remarks

While MPI-CHECK does not flag all possible MPI errors, and it may flag some instances of correct usage as potential errors, it has been shown to be very useful in discovering many subtle, yet common, MPI programming errors. It is easy to use and adds little overhead to the execution times of programs.

More information on MPI-CHECK and MPI-CHECK2 (deadlock detection) can be found at:

http://andrew.ai.iastate.edu/HPC/Papers/mpicheck/mpicheck1.htm

and

http://andrew.ai.iastate.edu/HPC/Papers/mpicheck2/mpicheck2.htm
Valgrind

http://valgrind.org

Tool suite consisting of two memory error detectors, a thread error detector, and a cache profiler.

Types of errors detected:

- Use of uninitialized memory
- Reading/writing memory after it has been freed
- Reading/writing off the end of malloc’d blocks
- Reading/writing inappropriate areas on the stack
- Memory leaks
- Passing of uninitialized and/or unaddressable memory to system calls
- Overlapping src and dst pointers in memcpy()-type fxns
- Some misuses of the POSIX pthreads API

Works only on x86 machines running Linux.

Memory Errors Identified with Valgrind

Memory errors manifested differently on different operating systems

- E.g. IRIX seems more forgiving than AIX
- Valgrind helpful because it instruments source code
  - Does not depend on runtime systems but focuses on logic and coding errors
  - Problems identified and corrected on Linux will assist on other operating systems

Valgrind good at performing program slicing

- Identifies uses of un-initialized variables
- Often associated with character-to-value conversions
  - `atoi` and `atof`
    - Possibly due to large character strings to hold small converted strings
Memory Errors Identified with Valgrind

Dangling pointers
Subtle errors that can be common in object-oriented programming
Can appear due to a lack of creating adequate copy constructors for objects using dynamic memory
Default copy constructors only perform shallow copies
Multiple objects all pointing to the same memory reference can result
Once one object frees the memory, the rest point to returned space
Valgrind will catch this if the destructors are properly coded
Multiple calls to free already returned memory will be flagged

Conclusions

Debugging is HARD
There are several tools available to assist you in debugging your code
Just like debugging, using these tools is not necessarily easy
Usually best to start simple and move to advanced techniques as required