Debugging and Optimization Tools

Richard Gerber
NERSC User Services
David Skinner
NERSC Outreach, Software & Programming Group
UCB CS267
February 15, 2011
Outline

- Introduction
- Debugging
- Performance / Optimization
Introduction

• Scope of Today’s Talks
  – Debugging and optimization tools
  – Some basic strategies

• Take Aways
  – Common problems and strategies
  – How tools work in general
  – A few specific tools you can try
• **Types of problems**
  - “Serial”
    • Invalid memory references
    • Array reference out of bounds
    • Divide by zero
    • Uninitialized variables
  - Parallel
    • Unmatched sends/receives
    • Blocking receive before corresponding send
    • Out of order collectives
• `printf()`, `print`, `write`
  – Versatile, sometimes useful
  – Doesn’t scale well, have to recompile

• **Compilers**
  – Turn on bounds checking, exception handling
  – Check dereferencing of NULL pointers

• **Serial gdb**
  – GNU debugger, serial, command-line interface
  – See “man gdb”

• **Parallel GUI debuggers**
  – DDT
  – Totalview
Out of bounds reference in source code for program “flip”

allocate(put_seed(random_size))

bad_index = random_size+1
put_seed(bad_index) = 67

Compiler runtime bounds checking

```
ftn -c -g -Ktrap=fp -Mbounds flip.f90
ftn -c -g -Ktrap=fp -Mbounds printit.f90
ftn -o flip flip.o printit.o -g

% qsub -I -qdebug -lmppwidth=48
% cd $PBS_O_WORKDIR
%
% aprun -n 48 ./flip

0: Subscript out of range for array put_seed (flip.f90: 50)
   subscript=35, lower bound=1, upper bound=34, dimension=1

0: Subscript out of range for array put_seed (flip.f90: 50)
   subscript=35, lower bound=1, upper bound=34, dimension=1
```
Ddt video
Performance Questions

• How can we tell if a program is performing well?

• Or isn’t?

• If performance is not “good,” how can we pinpoint why?

• How can we identify the causes?

• What can we do about it?
• **Primary metric: application time**
  - but gives little indication of efficiency

• **Derived measures:**
  - rate (Ex.: messages per unit time, Flops per Second, clocks per instruction), cache utilization

• **Indirect measures:**
  - speedup, parallel efficiency, scalability
• Serial
  – Leverage ILP on the processor
  – Feed the pipelines
  – Exploit data locality
  – Reuse data in cache

• Parallel
  – Minimizing latency
  – Maximizing work vs. communication
Identifying Targets for Optimization

• **Sampling**
  – Regularly interrupt the program and record where it is
  – Build up a statistical profile

• **Tracing / Instrumenting**
  – Insert hooks into program to time events

• **Use Hardware Event Counters**
  – Special registers count events on processor
  – E.g. floating point instructions
  – Many possible events
  – Only a few (~4 counters)
Performance Instrumentation

• Use a tool to “instrument” the code
  1. Transform a binary executable before executing
  2. Include “hooks” for important events
  3. Run the instrumented executable to capture those events, write out raw data file
  4. Use some tool(s) to interpret the data
Performance Tools @ NERSC

• **IPM**: Integrated Performance Monitor

• **Vendor Tools:**
  - CrayPat

• **Community Tools (Not all fully supported):**
  - TAU (U. Oregon via ACTS)
  - OpenSpeedShop (DOE/Krell)
  - HPCToolKit (Rice U)
  - PAPI (Performance Application Programming Interface)
Types of Counters

- Cycles
- Instruction count
- Memory references, cache hits/misses
- Floating-point instructions
- Resource utilization
• PAPI (Performance API) provides a standard interface for use of the performance counters in major microprocessors

• Predefined actual and derived counters supported on the system
  – To see the list, run ‘papi_avail’ on compute node via aprun:
    module load perftools
    aprun -n 1 papi_avail

• AMD native events also provided; use ‘papi_native_avail’:
  aprun -n 1 papi_native_avail
Introduction to CrayPat

• Suite of tools to provide a wide range of performance-related information

• Can be used for both sampling and tracing user codes
  – with or without hardware or network performance counters
  – Built on PAPI

• Supports Fortran, C, C++, UPC, MPI, Coarray Fortran, OpenMP, Pthreads, SHMEM

• intro_craypat(1), intro_app2(1), intro_papi(1)
1. **Access the tools**
   - module load perftools

2. **Build your application; keep .o files**
   - make clean
   - make

3. **Instrument application**
   - `pat_build ... a.out`
   - Result is a new file, `a.out+pat`

4. **Run instrumented application to get top time consuming routines**
   - `aprun ... a.out+pat`
   - Result is a new file `XXXXX.xf` (or a directory containing `.xf` files)

5. **Run `pat_report` on that new file; view results**
   - `pat_report XXXXX.xf > my_profile`
   - `vi my_profile`
   - Result is also a new file: `XXXXX.ap2`

**Adjust script for +pat**
Using Apprentice

• Optional visualization tool for Cray’s perftools data
• Use it in a X Windows environment
• Uses a data file as input (xxx.ap2) that is prepared by `pat_report`

1. module load perftools
2. ftn -c mpptest.f
3. ftn -o mpptest mpptest.o
4. pat_build -u -g mpi mpptest
5. aprun -n 16 mpptest+pat
6. pat_report mpptest+pat+PID.xf > my_report
7. app2 [--limit_per_pe tags] [XXX.ap2]
Apprentice Basic View

- Can select new (additional) data file and do a screen dump
- Worthless
- Useful
- Can select other views of the data
- Can drag the "calipers" to focus the view on portions of the data
National Energy Research Scientific Computing Center