Outline

• What is MPIProf and Why
• Basic usage of the mpi prof tool
• Profile results explained (Overflow as an example)
• Used-defined profiling via the mprof API
• Accuracy and overhead study
Performance Analysis

• Understanding performance characteristics of applications
  - Important for
    • Optimizing application performance to reduce compute time
    • Improving computing resource utilization

• Performance analysis tools
  - Often required due to
    • Sophistication in modern high performance computing systems
      - Hierarchical architecture with multicore CPUs and accelerators
      - Sophisticated memory system and network
    • Complicated application structure
  - Commercial tools
    • Intel Vtune, Allinea MAP, ITAC, SGI MPInside, IOT, op_scope, etc.
  - Open-source, research tools
    • TAU, OpenSpeedshop, PerfSuite, etc.
What is MPIProf?

- A profile-based application performance analysis tool
  - Gathers statistics in a counting mode
  - Reports aggregated and per-rank profiling information
  - Supports user-defined profiling
  - Works with many MPI implementations
    - including SGI MPT, Intel MPI, MPICH, MVAPICH, and OpenMP

- Reporting profiling information about
  - Point-to-point and collective MPI functions called by an application
    - time spent, number of calls, message size
  - MPI I/O and POSIX I/O statistics
  - Memory used by processes on each node
  - Call-path based information
Why MPIProf?

• Simple interface
  – A command-line tool without the need of modifying or recompiling applications
  – Auto-detection of MPI environment from different implementations
  – Text output with tabulated results for easy post processing
  – User-defined profiling only if needed

• Lightweight approach
  – Counting based, small amount of data
  – Low overhead (in both data collection and memory usage)
MPIProf Basics
Instrumentation Infrastructure

- Instrumenting MPI or I/O functions
  - By the PMPI interface and dlsym for dynamic shared library
  - Accessing instrumented functions via LD_PRELOAD or linking with the instrumented library
  - Call-path information provided by the libunwind interface
What’re Monitored?

- **MPI Functions (MPI 3)**
  - **Point-to-point calls (blocking and nonblocking)**
    - MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, MPI_Wait, etc.
  - **Collective calls (blocking)**
    - MPI_Bcast, MPI_Gather, MPI_Reduce, MPI_Allgather, MPI_Allreduce, etc.
  - **Collective calls (nonblocking)**
    - MPI_Ibarrier, MPI_Ibcast, MPI_Igather, MPI_Ireduce, MPI_Wait, etc.
  - **One-sided communication calls**
    - MPI_Put, MPI_Get, MPI_Accumulate, MPI_Win_complete, etc.
  - **MPI I/O calls**
    - MPI_File_open, MPI_File_read, MPI_File_write, etc.
What’re Monitored?

• POSIX I/O calls
  - open
    • open | fopen | creat | open64 | creat64
  - close
    • close | fclose
  - read
    • read | fread | pread | pread64
  - write
    • write | fwrite | pwrite | pwrite64
  - fsync
    • sync | fsync | fdatasync
Reported Profiling Results

- **Summary section**
  - Timing for communication, blocking, I/O, and computation
  - Message size and rates
  - I/O size and rates
  - Memory usage of processes on each node
  - Per-function summary

- **Break-down results in each rank**
  - Timing, number of calls, message size, I/O size

- **Map of messages communicated among ranks**
  - Rank, timing, message size

- **Call-path information**
  - Timing along call-path for the instrumented functions
Two Types of Usage

• The `mpiprof` profiling tool
  - Whole program analysis
  - No change or recompilation of application
    
    \[ \text{mpiexec -np } \langle n \rangle \text{ mpiprof [-options] a.out [args]} \]
  
• The `mprof` API routines
  - Selective profiling for selected code segments
  - Requires modification of application (instrumentation)
    
    \[ \text{Link with the mprof library, run as normal} \]

• Control of the amount of profiling information
  - Via `mpiprof` options
  - Via environment variables
  - See the user guide for details
Accessing MPIProf

• Load the proper modules
  
  module load comp-intel/2016.2.181  
  module load mpi-sgi/mpt.2.12r26  
  module load /u/scicon/tools/modulefiles/mpiprof-module  
  - The latest mpiprof version is 1.8.2

• Run your code
  
  mpiexec -np 64 mpiprof a.out  
  - Results will be written to “a.out_64_mpiprof_stats.out” at the end of a run  
  
  mpiprof a.out  
  - For serial (non-MPI) codes

• To get a quick help on mpiprof options, use
  
  mpiprof -help

pfe:/u/scicon/tools/opt/mpiprof/doc/mpiprof_userguide.pdf
The mpiprof Tool and Options
The mpiprof Profiling Tool

• Functionality
  – Whole program analysis
  – No change or recompilation of an application
  – The [-g] compilation flag recommended if collecting call-path information

• Basic usage
  – For MPI codes
    mpiexec -np <n> mpiprof [-options] a.out [args]
  – For non-MPI codes
    mpiprof [-options] a.out [args]

• Control of profiling information
  – Via command options or environment variables
# mpiprof Options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-lib &lt;mproflib&gt;</td>
<td>selects a runtime profiling library &lt;mproflib&gt;</td>
</tr>
<tr>
<td>-o &lt;outfile&gt;</td>
<td>writes profiling results to &lt;outfile&gt;</td>
</tr>
<tr>
<td>-[c,p]blk</td>
<td>estimates blocking time for collective and/or point-to-point communication calls</td>
</tr>
<tr>
<td>-msgm</td>
<td>collects rank-based message size and count maps</td>
</tr>
<tr>
<td>-byte</td>
<td>prints message size in bytes</td>
</tr>
<tr>
<td>-pflag &lt;value&gt;</td>
<td>sets MPROF_PFLAG to &lt;value&gt;</td>
</tr>
<tr>
<td>-mfunc <a href="">func:n</a></td>
<td>specifies a function to be monitored</td>
</tr>
<tr>
<td>-csig[=&lt;signo&gt;]</td>
<td>writes output stats when a signal is caught</td>
</tr>
<tr>
<td>-mem</td>
<td>reports memory usage only</td>
</tr>
<tr>
<td>-ios</td>
<td>reports I/O statistics and memory usage only</td>
</tr>
<tr>
<td>-cpath[=&lt;depth&gt;]</td>
<td>collects call-path information</td>
</tr>
<tr>
<td>-expr=&lt;exp&gt;</td>
<td>performs cpu+comm scaling experiments (experimental)</td>
</tr>
<tr>
<td>-v</td>
<td>sets verbose flag</td>
</tr>
</tbody>
</table>
**Env Variable MPROF_PFLAG**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>disable</td>
<td>disables profiling environment</td>
</tr>
<tr>
<td>off</td>
<td>switches off profiling</td>
</tr>
<tr>
<td>on</td>
<td>switches on profiling</td>
</tr>
<tr>
<td>cblk</td>
<td>estimates collective blocking time</td>
</tr>
<tr>
<td>pblk</td>
<td>estimates point-to-point blocking time</td>
</tr>
<tr>
<td>blk</td>
<td>is equivalent to &quot;cblk+pblk&quot;</td>
</tr>
<tr>
<td>msgm</td>
<td>collects message size and count maps</td>
</tr>
<tr>
<td>msgmx</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>prints message size in bytes</td>
</tr>
<tr>
<td>mem</td>
<td>reports memory usage only</td>
</tr>
<tr>
<td>ios</td>
<td>reports I/O statistics and memory usage only</td>
</tr>
<tr>
<td>cpath</td>
<td>collects call-path information</td>
</tr>
<tr>
<td>cpathx</td>
<td></td>
</tr>
</tbody>
</table>

*Note: mpiprof options override the value of MPROF_PFLAG*
Use of \texttt{mpiprof} Options

• Profiling in default setting (without other options)
  – Included
    • MPI functions, POSIX I/O functions
    • Memory usage
  – Not included
    • Blocking time measurement for MPI calls
    • Rank-based message maps
    • Call-path information

• A few useful options
  -\texttt{cblk} to enable blocking time measurement for collective calls
  -\texttt{msgm} to enable report of rank-based message maps
  -\texttt{cpath} to enable call-path information collection
  -\texttt{byte} to report message size and I/O size in bytes
  -\texttt{mem} to report memory usage only without profiling
  -\texttt{ios} to report I/O stats only (no MPI functions)
Profile Results Explained
The Overflow Test Case

• The NTR benchmark test case
  – DLRF6, 36 million grid points
  – 128 MPI processes on 8 Intel Sandy Bridge nodes

• Two run setups
  – Using the default setting
    `mpiexec -np 128 mpiprof ./overflowmpi`
  – Measuring the blocking time from MPI collectives
    `mpiexec -np 128 mpiprof -cblk ./overflowmpi`
Sample Outputs

MPIPROF v1.8.2, built 06/30/16, collected 06/30/16 09:35:17

==> List of environment variables
  MPROF_LIB   = sgimpt
  MPROF_EXEC  = ./overflowmpi

===> Summary of this run
  Number of nodes   = 8
  Number of MPI ranks = 128
  Number of inst'd functions = 17

  Total wall clock time   = 1027.71 secs
  Average computation time = 858.698 secs (83.55%)
  MPIProf overhead time   = 0.09571 secs (0.01%)

  Average communication time = 168.665 secs (16.41%)
    collective           = 96.6505 secs (9.40% or 57.30%Comm)
    point-to-point       = 72.0143 secs (7.01% or 42.70%Comm)
  Total message bytes sent = 1.5020T
    collective           = 40.252G
    point-to-point       = 1.4617T
  Total message bytes received = 1.5020T
    collective           = 40.252G
    point-to-point       = 1.4617T
  Gross communication rate = 10.0448 Gbytes/sec
  Communication rate per rank = 78.4749 Mbytes/sec

  Average I/O time (% L H) = 0.25426 secs (0.02%, 0.00000, 32.5237)
    write time           = 0.21419 secs (0.02%, 0.00000, 27.3985)
    read time            = 0.04007 secs (0.00%, 0.00000, 5.12879)

  . . . . . . .
Summary of Profile Results

- Statistics about a run
  - Number of nodes, ranks, and instrumented functions
  - Overall timing and rate information
- Meanings of a few key entries

<table>
<thead>
<tr>
<th>Entry</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total wall clock time</td>
<td>$T(\text{wallclock})$</td>
<td>Time spent from MPI_Init (inclusive) up to MPI_Finalize, or from mprof_init to mprof_end</td>
</tr>
<tr>
<td>Average computation time</td>
<td>$T(\text{comp})$</td>
<td>$= T(\text{wallclock}) - T(\text{comm}) - T(\text{i/o}) - T(\text{overhead})$</td>
</tr>
<tr>
<td>MPIProf overhead time</td>
<td>$T(\text{overhead})$</td>
<td>Average time used by MPIProf for gathering data, including mprof_init but excluding mprof_end</td>
</tr>
<tr>
<td>Average communication time</td>
<td>$T(\text{comm})$</td>
<td>Average time spent in MPI calls, excluding MPI-IO</td>
</tr>
<tr>
<td>Average I/O time</td>
<td>$T(\text{i/o})$</td>
<td>Average time spent in MPI-IO and Posix I/O</td>
</tr>
<tr>
<td>Effective I/O time</td>
<td>$T(\text{eff_i/o})$</td>
<td>Time estimated from I/O rates for each rank</td>
</tr>
<tr>
<td>Communication rate</td>
<td>$r(\text{comm})$</td>
<td>$= \text{Message size} / t(\text{comm})$ for each rank</td>
</tr>
<tr>
<td>I/O rate</td>
<td>$r(\text{i/o})$</td>
<td>$= \text{Data size} / t(\text{i/o})$ for each rank</td>
</tr>
</tbody>
</table>
Reported Information

- In summary sections
  - Average time across all ranks for communication, I/O, computation
  - Percentage of time relative to the total wall clock time
  - Communication and I/O rates
    - Calculated for each rank
    - Aggregated for all ranks
  - Memory usage

- Per-function summary
  - List of instrumented functions
  - Break-down timing, counts, message/data size

- Message/data size histograms

- Per-rank profiling data
  - Break-down timing, counts, message/data size
Per-Function Timing

- Sorted by timing from the run with default setting
Message Size Distribution

Message Size Histogram

Counts

SendCnt
RecvCnt

Message Size

NASA High End Computing Capability

Question? Use the Webex chat facility to ask the Host
Function Profiling on Each Rank

Overflow MPI Timing from "mpiprof"

Large amount of time spent in MPI_Gatherv in the first 16 ranks
Blocking Time in MPI Calls

• Two parts of time in MPI calls
  - Time waiting for the post of a message from a remote rank
  - Actual time spent in transmitting the message

• A large waiting (or blocking) time
  - Usually an indication of load imbalance

• Measurement of blocking time
  - No direct measurement without knowing the MPI implementation details
  - Estimation of blocking time by MPIProf (via the -cb1k option)
    • Inserting a barrier in front of each collective call
    • Measuring time spent in the barrier
    • Reporting the effective communication time (excluding the blocking time)
Function Profiling with “Blocking”

Overflow MPI Timing from "mpiprof -cblk"

Large amount of time spent in blocking when calling MPI_Gatherv in the first 16 ranks
I/O Stats Summary

- **Averaged I/O stats**
  - Calculated across all ranks

- **Effective I/O stats**
  - Estimated from rates
    
    \[
    T_{\text{effective}} = \frac{\text{total}_{\text{IO}}_{\text{data}}}{\text{aggregated}_{\text{IO}}_{\text{rate}}}
    \]
  - Useful to show when I/O is unbalanced

- **Example**

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average I/O time (%, L, H)</td>
<td>0.25426 secs (0.02%, 0.00000, 32.5237)</td>
</tr>
<tr>
<td>Write time</td>
<td>0.21419 secs (0.02%, 0.00000, 27.3985)</td>
</tr>
<tr>
<td>Read time</td>
<td>0.04007 secs (0.00%, 0.00000, 5.12879)</td>
</tr>
<tr>
<td>Effective I/O time (%, iF)</td>
<td>32.3911 secs (3.15%, 126.39)</td>
</tr>
<tr>
<td>Effective write time</td>
<td>27.2623 secs (2.65%, 126.28)</td>
</tr>
<tr>
<td>Effective read time</td>
<td>5.12879 secs (0.50%, 127.00)</td>
</tr>
</tbody>
</table>

** imbalance factor (iF) = (T_{\text{effective}} – T_{\text{average}}) / T_{\text{average}}**
More on I/O Stats

- I/O stats from option [-ios]
  - Do not include MPI I/O calls, but rather from low level I/O calls
  - May show different behavior than without the option
  - Reflect more about details of an MPI implementation

- Example: the FLASH IO benchmark with collective MPI I/O
  - 120-rank run across 5 nodes with 24 ranks/node
  - SGI MPT with Lustre filesystem support
  - I/O stats from [-ios]
    - With a stripe count of 1, only rank 0 does the writes
    - With a stripe count of 12, 4 ranks (0,24,48,72) do the writes
  - Related to the optimization made by MPT for I/O collective buffering

```bash
if (#nodes >= #stripes)
    #io_ranks = #stripes
else
    #io_ranks = largest number < #nodes that evenly divides #stripes
```
User-Defined Profiling Interface
(mprof Routines)
The mprof API Routines

- For profiling selected code segments
  - Instrumentation manually
  - Code recompilation required

- Four mprof API routines
  - `mprof_init(pflag)` - initializes the profiling environment, all ranks
  - `mprof_start()` - switches on data collection
  - `mprof_stop()` - switches off data collection
  - `mprof_end()` - finalizes and writes stats to output, all ranks

- Header include files
  - For C: `include "mprof_lib.h"
  - For Fortran: `include "mprof_flib.h"` or use `mprof_flib`
API Routine Calling Sequence

**Fortran Example**

```fortran
include "mprof_flib.h" (or use mprof_flib)

! initialize and turn on profiling
call mprof_init(MPF_ON)

... 1st profiled code segment
call mprof_stop() ! stop profiling

... Code segments without profiling

call mprof_start() ! restart profiling

... 2nd profiled code segment

call mprof_end() ! Finish and write results
```

MPF_ON implies mprof_start

Repeat as needed

Implied mprof_stop
Compilation and Execution

• Prerequisite: Load proper modules
  
  module load mpi prof-module

• Flags for compilation
  – At compiling time
    -I${MPROF_DIR}/include (or -I${MPROF_INC})
  – At linking time
    -L${MPROF_DIR}/lib -lmprof <mproflib> [-lmprof_flib]
    <mproflib> is one of the supported libraries (sgimpt, intelmpi, ...)

• At runtime
  – Run the instrumented code with or without the mpi prof tool
  – For the latter case, use environment variables to control profiling
    • Such as setting MPROF_LIB=sgimpt
MPIProf Accuracy and Overhead
Accuracy and Overhead

• Runtime overhead
  – In initial setup
  – During data collection for each function
  – In final writing of output data

• Memory overhead
  – From MPIProf internal buffers

• Accuracy and overhead study
  – Use NPB3.3.1-MPI, compare with benchmark timers
  – Three experiments
    • Without mpiprof
    • With mpiprof in default setting
    • With mpiprof –cpath option
Accuracy and Overhead Study

• Timing in seconds for seven NPBs
  - Class C problem
  - 64 ranks on Pleiades SandyBridge nodes

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>no mpiprof</th>
<th>with mpiprof</th>
<th>with mpiprof -cpath</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bmk-Time</td>
<td>Bmk-Comm</td>
<td>Bmk-Time</td>
</tr>
<tr>
<td>bt.C.64</td>
<td>18.9623</td>
<td>1.8871</td>
<td>18.8956</td>
</tr>
<tr>
<td>cg.C.64</td>
<td>4.8216</td>
<td>1.5901</td>
<td>4.8298</td>
</tr>
<tr>
<td>ft.C.64</td>
<td>6.3641</td>
<td>2.4831</td>
<td>6.4016</td>
</tr>
<tr>
<td>is.C.64</td>
<td>0.5781</td>
<td>0.3517</td>
<td>0.5966</td>
</tr>
<tr>
<td>lu.C.64</td>
<td>16.2089</td>
<td>2.8413</td>
<td>16.3658</td>
</tr>
<tr>
<td>mg.C.64</td>
<td>1.4375</td>
<td>0.1477</td>
<td>1.4569</td>
</tr>
<tr>
<td>sp.C.64</td>
<td>18.9689</td>
<td>2.6778</td>
<td>18.4860</td>
</tr>
</tbody>
</table>

• A few observations
  - Difference of benchmark time with/without mpiprof is less than 3%
  - The –cpath option has slightly larger overhead
  - Measured communication times agree in general with those reported by benchmarks except for LU where “Bmk-Comm” includes time for data packing and unpacking
Memory Overhead

• Memory usage of MPIProf internal buffers
  – Dependent on the number of ranks ($N$) and the number of instrumented functions ($M$)
  – An estimate of the buffer memory usage (in bytes)
    • For rank=0: $(656+32 \times M) \times N + 5984$
    • For rank>0: $32 \times (M+N) + 5488$

• Examples
  – For a case of $N=4096$, $M=12$
    • mem(rank=0) = 4.266 MB
    • mem(rank>0) = 0.137 MB
  – For a case of $N=10K$, $M=15$
    • mem(rank=0) = 11.366 MB
    • mem(rank>0) = 0.326 MB
Disclaimer
Limitations

• In data collection
  – No detailed trace information (due to counting mode)
  – For multi-threaded MPI, only the stats from the master thread of each rank are reported
  – When [-ios] is used, MPI I/O information is reported as level-low I/O
  – I/O support still in progress

• Presentation
  – Text-based tool, no GUI support

• Implementation
  – No call-path support for MVAPICH
Acknowledgment

• Members of the NAS APP group
  – For constantly testing and sending feedbacks
  – In particular Sherry Chang for constantly requesting new features and patiently reviewing the user guide

• Michael Raymond of SGI
  – For sharing some of the insight of SGI MPT data

• Some of the original idea was motivated by SGI MPInside

• Contact information
  Henry Jin <hjin@nas.nasa.gov>