Performance Analysis and Debugging Tools

Performance analysis and debugging intimately connected since they both involve monitoring of the software execution. Just different goals:

- Debugging -- achieve correct code behaviour
- Performance analysis -- achieve performance criteria

Both are prey to the pragmatic pitfalls of experimental science:

- They must measure but not unduly perturb the measured system (e.g. debugging threads can perturb the scheduling of events; saved counters can shift memory alignment)
- However, data collected must be sufficiently detailed to capture the phenomenon of interest.

Part of the problem is that the application developer is "once-removed" from the REAL code: i.e. developer sees the machine hardware architecture through the lens of the programming model. Compilers may dramatically change the programming model code to fit the architecture. Developers can only fiddle with the model. This is particularly noticeable with the higher level programming systems (OpenMP vs. MPI)

Therefore, runtime measures from data collection must relate the measurements to the model code to be useful, i.e. the "inverse transformation" must be applied.

Tools must be simple and easy to use despite this, or else people won't take the time to learn them and use them.

Performance Analysis

Object is to minimize wall time in general. Hopefully, did performance analysis earlier to remove major problems and therefore just looking for:

- Hotspots -- places where the code spends a lot of time necessarily -- make these as efficient as possible
- Bottlenecks -- places where the code spends a lot of time unnecessarily -- remove these

Four basic performance data collection techniques:

- Timers : simple command line timing utilities, Fortran/C/MPI timing utilities
- Profilers record the amount of time spent in different parts of a program. Profiles typically are gathered automatically.
- Counters record either frequencies of events or cumulative times. The insertion of counters may require some programmer intervention (extrinsic or intrinsic)
- Event tracers record each occurrence of various specified events, thus typically producing a large amount of data. Tracers can be produced either automatically or with programmer intervention (extrinsic and intrinsic)

Then often require data transformation -- extract useful information from vast amount of data collected. And then data visualisation to interpret the distilled results.

Generally, an iterative process!

Performance Analysis

When selecting a tool for a particular task, the following issues should be considered:

1. Accuracy. In general, performance data obtained using sampling techniques are less accurate than data obtained by using counters or timers. In the case of timers, the accuracy of the clock must be taken into account (low clock accuracy, high clock overhead).

2. Simplicity. The best tools in many circumstances are those that collect data automatically, with little or no programmer intervention, and that provide convenient analysis capabilities.

3. Intrusiveness. Unless a computer provides hardware support, performance data collection inevitably introduces some overhead. We need to be aware of this overhead and account for it when analyzing data. Not just time shift: e.g. setting breakpoint in MPI can change the order in which receives gets sent messages. "Uncertainty principle".

Performance Analysis

The most important goal of performance tuning is to reduce a program's wall clock execution time. Reducing resource usage in other areas, such as memory or disk requirements, may also be a tuning goal.

Performance tuning is an iterative process used to optimize the efficiency of a program. It usually involves finding your program's hot spots and eliminating the bottlenecks in them.

* Hot Spot: An area of code within the program that uses a disproportionately high amount of processor time.

* Bottleneck: An area of code within the program that uses processor resources inefficiently and therefore causes unnecessary delays.

Performance tuning usually involves profiling - using software tools to measure a program's run-time characteristics and resource utilization.

Use profiling tools and techniques to learn which areas of your code offer the greatest potential performance increase BEFORE you start the tuning process. Then, target the most time consuming and frequently executed portions of a program for optimization.

Performance Analysis

Consider optimizing your underlying algorithm: an extremely fine-tuned O(N * N) sorting algorithm may perform significantly worse than a untuned O(N log N) algorithm.

Finally, know when to stop - there are diminishing returns in successive optimizations. Consider a program with the following breakdown of execution time percentages for the associated parts of the program:

Procedure	% CPU Time
main()	13%
procedure1()	17%
procedure2()	20%
procedure3()	50%

A 20% increase in the performance of procedure3() results in a 10% performance increase overall.

A 20% increase in the performance of main() results in only a 2.6% performance increase overall.

Timers

Shell:

```
time ./code_exec
```

C/C++:

```
clock()
current_time()
gettimeofday()
```

Fortran:

```
(call timer(t1); call timer(t2); elapsed=t2-t1)
Call etime(t1) (Old:UNIX systems)
call system_clock(t1)
call cpu_time(t1)
```

MPI:

```
MPI_WTIME
```

OpenMP:

```
Seconds = get_omp_wtime()
```

Profilers

Prof

Basically, percentage of CPU used by each procedure

Compile with the "-p" option

Produces files "mon.out.*"

Type "prof -m mon.out.*" (-m = merge) or "prof mon.out.n" for individual stats on each processor

Gprof

Same thing except profiles according to call graphs i.e. it gives percentages for procedure and all child procedures -- timing includes timing for embedded calls further down the call tree.

Compile with "-pg"

Produces files "gmon.out.*"

Type "gprof -m mon.out.*" or "gprof mon.out.n"

Valgrind (http://www.valgrind.org/info/tools.html)

Suit of dynamical analysis tools: memory error detector, call tree generator, various profilers

e.g. valgrind –tool=callgrind ./a.out ; callgrind_annotate –auto=yes callgrind.out<pid>

https://web.stanford.edu/class/archive/cs/cs107/cs107.1174/guide_callgrind.html

valgrind –tool=memcheck ./a.out

MPI/OpenMP profiling tools

Built in -- Switch all MPI_XXX calls to PMPI_XXX; IPM; mpiP; HPCToolkit

OmP, PomP for OpenMP

Counters

Lot of vendors provide "on-chip" hardware counters these days and these can be accessed and read.

e.g. Cray/IBM Power series -- Hardware performance monitor HPM:

Gives clock cycles, instructions completed, load/store cache misses …

Run executable with "hpmcount -o outputfile myprog"

Examine results with "perf outfile"

Portable counters: Performance Application Programming Interface (PAPI)

`http://icl.cs.utk.edu/papi/`

Commercial/vendor tracing tools

Tau – Tuning and Analysis Utilities (ParaProf, PerfExplorer, Jumpshot)
Stampede2 : https://portal.tacc.utexas.edu/software/tau

Vampir -- event-trace visualisation using MPI traces (regular MPI)

Jumpshot -- event-trace visualisation using MPI traces (MPICH/MPE)

MPI_trace

Paragraph

Paraver/Dimemas -- MPI/OpenMP, Linux

Pablo -- Instrumentation and perf analysis of C, F77, F90 and HPF

IBM AIX parallel environment

Paradyn

Pgprof

Peekperf

Debugging Tools

Parallel computing means that software bugs may arise from the complex interactions of the large number of parallel software components.

These bugs may have high latency -- i.e. they don't manifest themselves until sometime after they were caused.

They may also be subtly dependent on timing conditions that are not easily reproducible.

⇒ Debugging can be extremely time consuming and tedious!

Some tips for avoiding having to debug:

- ⇒ Implicit none
- ⇒ Comment/indent/etc -- write readable code
- ⇒ Use version update system: cvs, svn, git

Tools:

Print statements still often the best way to go!

Breakpoint debuggers:

gdb -- ouch

Totalview -- all singing, all dancing commercial software. Pray it is on your system!

GDB

gfortran –g hello.f90 –o hello

gdb ./hello

(gdb)

- List
- Run <arg1 arg2 “arg3”>
- <ctrl C>
- Kill
- Quit
- <ctrl C>
- continue
- Next (step over)
- Step (step into)
- Print var
- Set var <var> = <val>

GDB

Finish (returns from fn)

Backtrace (call stack)

Frame <frame #> (switch to place in stack)

Info frame

Info locals

Info args

Break <line #>

Break file:line

Tbreak (temp break – once only)

Info breakpoints

Disable <breakpt #>

Ignore <breakpt #>

Watch (for vars not lines or fn)

Rwatch (read watch instead of write)

Awatch (both read and write)

Visual versions: ddd gvd?

GDB

It is possible to debug MPI codes with gdb!

Not for the faint of heart? `https://www.open-mpi.org/faq/?category=debugging`

1. Put stall/sleep code in code
2. Mpirun
3. Login to mpi node
4. Gdb –pid to attach to the process id
5. Go up function stack to sleep code
6. Unset variable that is causing sleep
7. Set breakpoint after sleep codew
8. Run
9. Now can continue gdb

Launch xterm from each MPI node with gdb running

1. Add all compute nodes to xhost: For host in `cat my_hostfile`; do xhost +host; done
2. Send displays from gdb execution to master node: mpirun –np 4 --hostfile my_hostfile –x DISPLAY=`hostname`:0 xterm –e gdb my_application

MPI debugging: Totalview

Only good if your system has it!

(We don't, not even on Hyades ☹)

V expensive

But awesome!

Great tutorial:

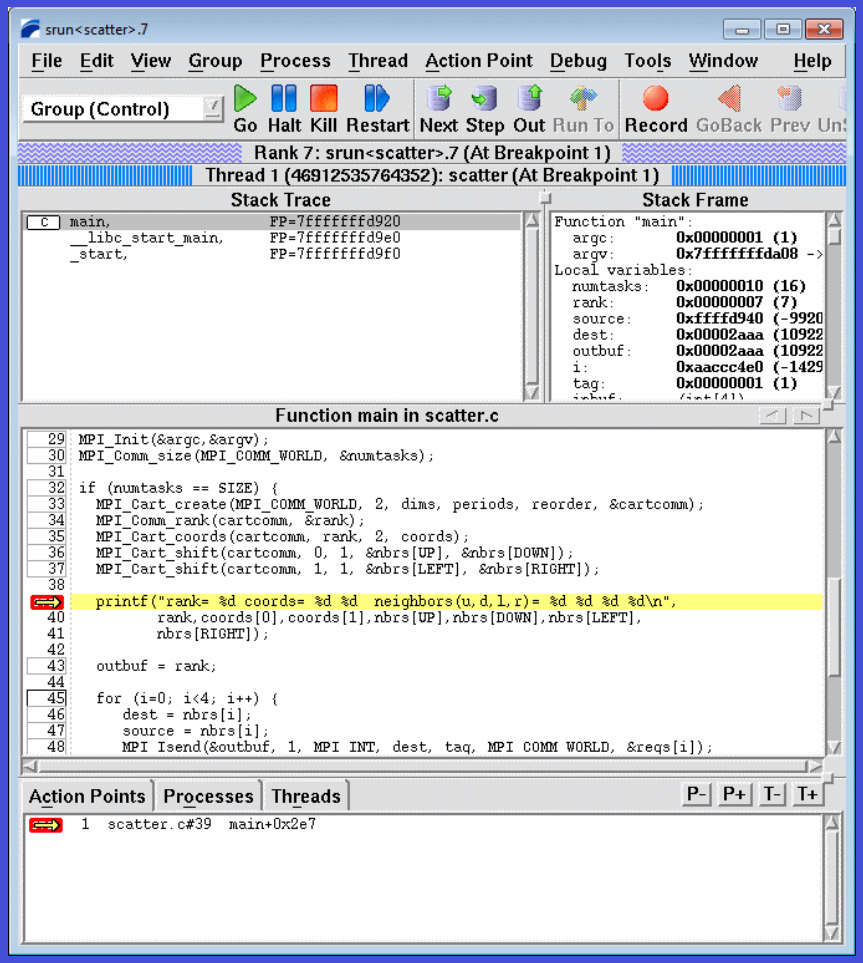
https://computing.llnl.gov/tutorials/totalview/

We do have Allinea Distributed Debugging Tool (DDT) on hyades which is similar ☺

(apparently – I haven't used it)

Basically compile with –g, run interactive with qsub –I, and use DDT to start the executable (ddt ./a.out)

Stampede2: https://portal.tacc.utexas.edu/software/ddt



MPI debugging help

1. setenv <MPI_DEBUG_OPT> 1 (1 is on)
2. mpirun –mca <MPI_DEBUG_OPTION> 1 –np 4 ./my_application

- **mpi_param_check:** If set to true (any positive value), and when Open MPI is compiled with parameter checking enabled (the default), the parameters to each MPI function can be passed through a series of correctness checks. Problems such as passing illegal values (e.g., NULL or MPI_DATATYPE_NULL or other "bad" values) will be discovered at run time and an MPI exception will be invoked (the default of which is to print a short message and abort the entire MPI job). If set to 0, these checks are disabled, slightly increasing performance.
- **mpi_show_handle_leaks:** If set to true (any positive value), OMPI will display lists of any MPI handles that were not freed before MPI_FINALIZE (e.g., communicators, datatypes, requests, etc.).
- **mpi_no_free_handles:** If set to true (any positive value), do not actually free MPI object when their corresponding MPI "free" function (e.g., do not free communicators when MPI_COMM_FREE is invoked). This can be helpful in tracking down applications that accidentally continue to use MPI handles after they have been freed.
- **mpi_show_mca_params:** If set to true (any positive value), show a list of all MCA parameters and their values during MPI_INIT. This can be quite helpful for reproducability of MPI applications.
- **mpi_show_mca_params_file:** If set to a non-empty value, and if the value of **mpi_show_mca_params** is true, then output the list of MCA parameters to the filname value. If this parameter is an empty value, the list is sent to stderr.
- **mpi_keep_peer_hostnames:** If set to a true value (any positive value), send the list of all hostnames involved in the MPI job to every process in the job. This can help the specificity of error messages that Open MPI emits if a problem occurs (i.e., Open MPI can display the name of the peer host that it was trying to communicate with), but it can somewhat slow down the startup of large-scale MPI jobs.
- **mpi_abort_delay:** If nonzero, print out an identifying message when MPI_ABORT is invoked showing the hostname and PID of the process that invoked MPI_ABORT, and then delay that many seconds before exiting. A negative value means to delay indefinitely. This allows a user to manually come in and attach a debugger when an error occurs. Remember that the default MPI error handler -- MPI_ERRORS_ABORT -- invokes MPI_ABORT, so this parameter can be useful to discover problems identified by **mpi_param_check**.
- **mpi_abort_print_stack:** If nonzero, print out a stack trace (on supported systems) when MPI_ABORT is invoked.
- **mpi_ddt_<foo>_debug**, where **<foo>** can be one of **pack**, **unpack**, **position**, or **copy**: These are intenral debugging features that are not intended for end users (but `ompi_info` will report that they exist).