Allinea MAP and Perf-report

New User Training 2017

Woo-Sun Yang
User Engagement Group, NERSC

February 23, 2017
MAP

- Allinea’s parallel profiling tool with GUI
- Based on sampling
  - Source lines are annotated with performance data
  - Time series of performance metrics for all (MPI) processes are displayed
- 4,096 MAP license tokens (MPI tasks)
  - Shared by other users and among all machines
- Use NX or Allinea remote client
- Reverse connect also works for MAP
- For info:
  - [https://www.allinea.com/products/map](https://www.allinea.com/products/map)
  - [https://www.nersc.gov/users/software/debugging-and-profiling/MAP/](https://www.nersc.gov/users/software/debugging-and-profiling/MAP/)
Using Allinea remote client

(1) Select ‘Configure’ to create a configuration for a NERSC machine

2nd entry for a MOM node
Cori: cmom02 or cmom06
Edison: edimom01, ..., or edimom06

(2) Create a configuration

Note that the paths will change for future versions
Using Allinea remote client works, too

(3) Select a machine

(4) Enter the NIM password
(5) Submit a batch job on a NERSC machine and start DDT

```bash
$ salloc -N 1 -t 30:00 -p debug -C knl
...
$ module load allineatools
$ map --connect ./jacobi_mpiomp
```

(6) Accept the request

![Request Acceptance Dialogue]

(7) Set parameters and run

![Parameter Setting Dialogue]
How to profile with MAP

• Build a statically-linked executable

$ module load allinea-tools
$ make-profiler-libraries --lib-type=static
$ ftn -c -g -O3 -qopenmp jacobi_mpiomp.f90
$ make-profiler-libraries --lib-type=static
$ ftn -O3 -qopenmp jacobi_mpiomp.o -Wl,@./allinea-profiler.1d -o jacobi_mpiomp

• Build a dynamically-linked executable

$ ftn -c -g -O3 -qopenmp jacobi_mpiomp.f90
$ ftn -dynamic -O3 -qopenmp jacobi_mpiomp.o -Wl,--eh-frame-hdr

• Run

$ salloc -N 1 -t 30:00 -p debug -C knl
$ module load allinea-tools
$ map ./.jacobi_mpiomp
$ map --profile srun -n 8 -c 32 ./.jacobi_mpiomp

The module name will change to ‘forge’ for future versions

GUI mode
Command-line mode

Profiling results saved in a file
To view it
Using numactl on KNL

- Avoid using numactl with MAP/7.0, as in
  
  $ map srun -n ... numactl ... ./a.out

- Use ‘--mem-bind=map_mem:1’ etc., with srun, instead of using ‘numactl --mem_bind=1’, etc.

- No equivalent for --preferred=1 for now.

- This problem will be resolved with a new Slurm.
MAP results
MAP results (Cont’d)

For the selected lines
Adding more metrics to the display panel

- You can add more performance metrics

![Image of metrics display panel]
Profiling only part of a program

- **C**
  - Include
    - `#include "mapsampler_api.h"`
    - `allinea_start_sampling()`
    - `allinea_stop_sampling()`
    - `-I${ALLINEA_TOOLS_DIR}/${ALLINEA_TOOLS_VERSION}/map/wrapper`
    - `-L${ALLINEA_TOOLS_DIR}/${ALLLINEA_TOOLS_VERSION}/lib/64 -lmap-sampler`

- **Fortran**
  - Include
    - `CALL ALLINEA_START_SAMPLING()`
    - `CALL ALLINEA_STOP_SAMPLING()`
    - `-L${ALLINEA_TOOLS_DIR}/${ALLINEA_TOOLS_VERSION}/lib/64 -lmap-sampler`

- **Before starting your program:**
  
  ```bash
  export ALLINEA_SAMPLER_DELAY_START=1
  ```
Perf-report

• Newly installed on Cori and Edison
• Allinea’s tool for a quick characterization of parallel code performance
  – Based on times spent in 3 areas
  – “Compute-bound,” “MPI-bound” or “I/O-bound”
  – Results in html and plain-text files

• Build exactly the same way as you do with MAP
• 4,096 license tokens (MPI tasks)
• For info:
How to use perf-report

• Run under perf-report

$ salloc -N 1 -t 30:00 -p debug -C knl
$ module load allineatools/7.0-rep
$ export OMP_NUM_THREADS=8
$ perf-report srun -n 32 -c 8 ./jacobi_mpiomp
$ ls
miniGhost_flat_96p_6n_2t_2017-02-22_12-26.html
miniGhost_flat_96p_6n_2t_2017-02-22_12-26.txt

– Don’t run with numactl with version 7.0

• Generate a perf-report output from a map file

$ module load allineatools/7.0-rep
$ perf-report miniGhost_flat_96p_6n_2t_2017-02-22_09-16.map
$ ls -l
miniGhost_flat_96p_6n_2t_2017-02-22_09-16.html
miniGhost_flat_96p_6n_2t_2017-02-22_09-16.map
miniGhost_flat_96p_6n_2t_2017-02-22_09-16.txt

The module name will change to ‘forge’ for future versions
Perf-report (cont’d)

Command:
srun -n 96 --ntasks-per-node=16 -c 16
/tmp/miniGhost_cache.x --scaling 1 --nx 336 --ny 336 --nz 336 --num_vars 40 --num_spikes 1 --error_tol 8
debug_grid 1 --report_diffusion 21 --percent_sum 100 --num_tsteps 2 --stencil 24 --comm_method
10 --report_perf 1 --npx 4 --npy 4 --npz 6

Resources: 6 nodes (68 physical, 272 logical cores per node)
Memory: 94 GiB per node
Tasks: 96 processes, OMP_NUM_THREADS was 2
Machine: mid06709
Start time: Wed Feb 22 2017 10:42:48 (UTC-08)
Total time: 1102 seconds (about 18 minutes)
Full path: /tmp

Summary: miniGhost_cache.x is Compute-bound in this configuration

Compute 60.1%  
Time spent running application code. High values are usually good.
This is average; check the CPU performance section for advice

MPI 39.9%  
Time spent in MPI calls. High values are usually bad.
This is average; check the MPI breakdown for advice on reducing it

I/O 0.0%  
Time spent in filesystem I/O. High values are usually bad.
This is negligible; there's no need to investigate I/O performance

This application run was Compute-bound. A breakdown of this time and advice for investigating further is in the CPU section below.
**CPU**

A breakdown of the 60.1% CPU time:

- Single-core code: 2.0%
- OpenMP regions: 98.0%
- Scalar numeric ops: 4.9%
- Vector numeric ops: <0.1%
- Memory accesses: 64.8%

The per-core performance is memory-bound. Use a profiler to identify time-consuming loops and check their cache performance.

No time is spent in vectorized instructions. Check the compiler's vectorization advice to see why key loops could not be vectorized.

**MPI**

A breakdown of the 39.9% MPI time:

- Time in collective calls: 97.1%
- Time in point-to-point calls: 2.9%
- Effective process collective rate: 5.24 bytes/s
- Effective process point-to-point rate: 2.88 MB/s

Most of the time is spent in collective calls with a very low transfer rate. This suggests load imbalance is causing synchronization overhead; use an MPI profiler to investigate.

**I/O**

A breakdown of the 0.0% I/O time:

- Time in reads: 0.0%
- Time in writes: 0.0%
- Effective process read rate: 0.00 bytes/s
- Effective process write rate: 0.00 bytes/s

No time is spent in I/O operations. There's nothing to optimize here!

**OpenMP**

A breakdown of the 98.0% time in OpenMP regions:

- Computation: 89.4%
- Synchronization: 10.6%
- Physical core utilization: 46.8%
- System load: 47.3%

**Physical core utilization** is low and some cores may be unused. Try increasing OMP_NUM_THREADS to improve performance.

**Memory**

Per-process memory usage may also affect scaling:

- Mean process memory usage: 318 MB
- Peak process memory usage: 524 MB
- Peak node memory usage: 12.0%

The peak node memory usage is very low. Running with fewer MPI processes and more data on each process may be more efficient.
About sample size

• 1,000 samples per (MPI) task by default
• With multithreading, the sample size per thread is reduced further
• Are they enough for a long running application?
  – A 30-minute run: ~1 sample per every 2 sec (0.56 Hz vs. GHz of clockcycle)
  – CrayPat’s sampling rate: 100 Hz by default
  – If this were for a binomial distribution (i.e., 2 outcomes only), 1,000 samples would be OK (3% of margin of error under 95% confidence level). But it’s a multinomial distribution, instead.

• Can increase the sample size via the ALLINEA_SAMPLER_NUM_SAMPLES environment variable
• Can profile only an interesting portion (to have smaller margins of error without having to use a large sample size)