Allinea MAP and Perf-report

New User Training 2017

Woo-Sun Yang
User Engagement Group, NERSC

February 23, 2017
• 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.nersc.gov/users/software/debugging-and-profiling/MAP/
(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
(3) Select a machine

- **PROFILE**
  Profile a program.

- **LOAD PROFILE DATA FILE**
  Load a profile data file from a previous run.

- **OPTIONS**

  **Remote Launch:**
  - **On**
  - **Off**
  - **Configure...**

  **cori**
  **edison**

(4) Enter the NIM password

![Screen capture of connecting to remote host](image)
(5) Submit a batch job on a NERSC machine and start DDT

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

(6) Accept the request

A new Reverse Connect request is available from nid06246 for Allinea MAP.
Command Line: --connect ./jacobi_mpiomp
Do you want to accept this request?

[Help] [Accept] [Reject]

(7) Set parameters and run
How to profile with MAP

• **Build a statically-linked executable**

```bash
$ module load allineatools
$ make-profiler-libraries --lib-type=static
$ ftn -c -g -O3 -qopenmp jacobi_mpiomp.f90
$ ftn -O3 -qopenmp jacobi_mpiomp.o -Wl,@./allinea-profiler.ld -o jacobi_mpiomp
```

Build 2 static libs that MAP needs

Use the MAP-provided option file

• **Build a dynamically-linked executable**

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

• **Run**

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

GUI mode

Command-line mode

Profiling results saved in a file

```bash
$ ls -l
-rw-------- 1 wyang wyang 1253277 Feb 22 09:36 miniGhost_flat_96p_6n_2t_2017-02-22_09-16.map
$ map miniGhost_flat_96p_6n_2t_2017-02-22_09-16.map
```

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.
- A workaround will be available with a new Slurm.
MAP results
MAP results (Cont’d)

For the selected lines

rhol=zi0n(6,m)^smu_inv

\[ r = \sqrt{2 \cdot |\text{psitmp}|} \]
\[ ip = \max \left(0, \min \left(\text{mpi.i_int} \cdot (r-a0)^* \text{delt} + 0.5\right)\right) \]
\[ j = \max \left(0, \min \left(\text{mtheta(ip).int} \cdot \text{thetaatmp} \cdot \text{pi2}_\text{int}^4 \cdot \text{delt} \cdot (ip) + 0.5)\right)\right) \]
\[ ipj = (\text{grid(id)} + jt) \]
\[ wz1 = (\text{zetatmp} \cdot \text{zeta} \cdot \text{delz}) \]
\[ kk = \max \left(0, \min \left(\text{nzeta_int} \cdot (wz1)\right)\right) \]
Adding more metrics to the display panel

- You can add more performance metrics
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${ALLINEA_TOOLS_DIR}/${ALLINEA_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:**
  
  `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 ‘reports’ for future versions
Summary: miniGhost_cache.x is **Compute-bound** in this configuration

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>60.1%</td>
<td>Time spent running application code. High values are usually good. This is <strong>average</strong>; check the CPU performance section for advice.</td>
</tr>
<tr>
<td>MPI</td>
<td>39.9%</td>
<td>Time spent in MPI calls. High values are usually bad. This is <strong>average</strong>; check the MPI breakdown for advice on reducing it.</td>
</tr>
<tr>
<td>I/O</td>
<td>0.0%</td>
<td>Time spent in filesystem I/O. High values are usually bad. This is <strong>negligible</strong>; there's no need to investigate I/O performance.</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-core code</td>
<td>2.0%</td>
</tr>
<tr>
<td>OpenMP regions</td>
<td>98.0%</td>
</tr>
<tr>
<td>Scalar numeric ops</td>
<td>4.9%</td>
</tr>
<tr>
<td>Vector numeric ops</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>Memory accesses</td>
<td>64.8%</td>
</tr>
</tbody>
</table>

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 39.8% MPI time:

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in collective calls</td>
<td>97.1%</td>
</tr>
<tr>
<td>Time in point-to-point calls</td>
<td>2.9%</td>
</tr>
<tr>
<td>Effective process collective rate</td>
<td>5.24 bytes/s</td>
</tr>
<tr>
<td>Effective process point-to-point rate</td>
<td>2.88 MB/s</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time in reads</td>
<td>0.0%</td>
</tr>
<tr>
<td>Time in writes</td>
<td>0.0%</td>
</tr>
<tr>
<td>Effective process read rate</td>
<td>0.00 bytes/s</td>
</tr>
<tr>
<td>Effective process write rate</td>
<td>0.00 bytes/s</td>
</tr>
</tbody>
</table>

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

## Memory
Per-process memory usage may also affect scaling:

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean process memory usage</td>
<td>518 MiB</td>
</tr>
<tr>
<td>Peak process memory usage</td>
<td>524 MiB</td>
</tr>
<tr>
<td>Peak node memory usage</td>
<td>12.0%</td>
</tr>
</tbody>
</table>

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

## OpenMP
A breakdown of the 98.0% time in OpenMP regions:

<table>
<thead>
<tr>
<th>Component</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation</td>
<td>89.4%</td>
</tr>
<tr>
<td>Synchronization</td>
<td>10.6%</td>
</tr>
<tr>
<td>Physical core utilization</td>
<td>46.8%</td>
</tr>
<tr>
<td>System load</td>
<td>47.3%</td>
</tr>
</tbody>
</table>

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