Cross-Layer Performance Analysis of Parallel Programs with Aftermath

Tutorial at HiPEAC 2017

Virtual Machine Image

The virtual machine image based on Debian GNU/Linux includes the source code and a fully working installation of Aftermath, Aftermath-OpenMP, libaftermath-trace and OpenStream. The image is using the *Open Virtualization Format*, compatible with all major virtualization solutions. Once you have imported the VM image and started it, you can log in to the system by using the following user name and password:

User name: *openstream* **Password:** *openstream*

Analysis of OpenMP programs

1. Open a terminal, go to `/home/openstream/example-traces/openmp/shortguide` and open the trace file `prime_naive.ost`:

   ```
   $ cd /home/openstream/example-traces/openmp/shortguide
   $ aftermath prime_naive.ost
   ```

   (a) How much time was spent in total waiting on barriers?

   (b) Set the time line mode to *loop constructs* using the tool bar button labeled *OMP* at the top.

   (c) How were the iterations distributed across the workers?

   (d) How much time did it take for each worker to complete the computation?

2. Open the file `prime_naive.c` in a text editor (*emacs*, *vi* or *gedit* are pre-installed in the VM) and change `schedule(static)` to `schedule(dynamic)`).

   (a) Recompile the executable with the *clang* compiler. This can be done by executing:
   ```
   $ clang -g -o prime_naive prime_naive.c -lm -fopenmp
   ```

   (b) Generate a new trace file named `prime_naive_dynamic.ost` using `aftermath-openmp-trace`:
   ```
   $ aftermath-openmp-trace -o prime_naive_dynamic.ost -f -- ./prime_naive
   ```

   (c) Open the generated trace file with Aftermath. Could the time spent in barriers be reduced? What about the total time of the computation?

3. Change the schedule back to `static` and specify a chunk size of 1000 by changing `schedule(dynamic)` to `schedule(static, 1000)`. Recompile the program with clang and generate a new trace file called `prime_naive_chunked.ost`. Could the total time of the computation be reduced?

4. Go to `/home/openstream/example-traces/openmp/mg` and open the trace file `mg.C.ost`.

   (a) How many parallel loop constructs are there?

   (b) Make all cores visible by clicking on the first button in the tool bar at the top and set the time line to loop construct mode. Choose a loop with a strong imbalance and compare the number of iterations in the iteration set of one of the fastest workers with one of the slowest workers. Assuming a constant amount of work per iteration, does the number of iterations justify the imbalance?

   (c) What is the duration of the main computation phase? How much time was spent on barriers? Compare with `mg-numa.C.ost` and `mg-numa.C-128-threads.ost`.

Analysis of task-parallel OpenStream programs

1. Go to `/home/openstream/example-traces/openstream/seidel/` and open the trace `rnd.ost.bz2` as well as the executable `rnd-stream_df_seidel` with Aftermath by running:

   ```
   $ aftermath rnd.ost.bz2 rnd-stream_df_seidel
   ```

   Select *Color Schemes* from the *Edit* menu, click on *Open* in the popup dialog and select the file `/home/openstream/example-traces/openstream/seidel/seidel-colorscheme.acs`. Next, select the entry *Stream_df_seidel* from the list of color schemes and click on the *Apply* button. This color scheme associates yellow with initialization tasks (writing input data to streams), green with main computation tasks and red with termination tasks (writing output data to global memory).

   (a) Activate the task type map by clicking on the symbol with a capital T from the tool bar. Which phases of the benchmark can be identified? Is there any overlap between the phases?

   (b) Click on the button labeled *Select from graph* at the bottom of the right panel and select the entire execution interval from the time line. What is the average task duration?

   (c) For the next steps, we are only interested in main computation tasks. Deselect all tasks on the left panel by clicking on the button labeled *Select none* below the list of task types. Now select all tasks whose symbol starts with *create_next_iteration_task* and click on the *Apply* button below the list. What does the histogram with distribution of the task duration look like?

   (d) Click anywhere on an interval rendered in green on the time line to select a main computation task. Zoom in until the dashed red rectangle indicating the selected task becomes visible. How much data was read and written by the task? Which NUMA nodes were accessed?

   (e) Click on the button with the yellow circles at the very right of the toolbar. This tells Aftermath to analyze all dependences and to highlight the producers of the selected task and their producers on the time line. On which cores were the producers executed?

   (f) Zoom out and activate the NUMA heat map by clicking on the blue and purple button between the button with a W and the type map button. Given that blue indicates a high fraction of local memory accesses and purple indicates a high fraction of remote memory accesses, how would you characterize this execution?

   (g) What does the NUMA communication incidence matrix in the right panel look like? How much data was accessed locally? How much data was accessed remotely?

2. Open `opt.ost.bz2` and `opt-stream_df_seidel`, load and apply the same color scheme as above. Make the entire execution and all cores visible by clicking on the first two buttons in the tool bar.

   (a) Activate the NUMA read map by clicking on the tool bar button with a capital R. This view associates a color with each NUMA node, determines the NUMA node with the highest amount of read accesses for each pixel's interval and renders the pixel in the appropriate color. What does the appearing pattern look like?

   (b) Activate the NUMA write map by clicking on the tool bar button with a capital W. Is the pattern similar to the pattern shown by the NUMA read map?

   (c) When comparing the NUMA read and write maps there is a large "gap" in the read map at the beginning. Why?

   (d) Select the entire execution and have a look at the NUMA communication incidence matrix. How would you characterize this execution?

3. Go to `/home/openstream/openstream/examples`, create a copy of the `skeleton` folder named `prodcons` and change to that directory. Next, compile the program by invoking `make` and run it:

   ```
   $ cd /home/openstream/openstream/examples
   $ cp -r skeleton prodcons
   $ cd prodcons
   $ make
   $ ./skeleton
   ```

(a) Open the file `skeleton.c` with a text editor. What does this program do?

(b) Modify the program such that the first task produces an integer that is consumed by the second task. To this end, modify the main function as in the following listing:

```
int main(int argc, char** argv)
{
  int s __attribute__((stream));

  #pragma omp task output(s)
  {
    s = 1234;
    printf("Producer writes %d\n", s);
  }

  #pragma omp task input(s)
  {
    printf("Consumer reads %d\n", s);
  }

  #pragma omp taskwait
  return 0;
}
```

Recompile and execute the program.

(c) Change the program, such that multiple data elements are exchanged between the producer and consumer:

```
int main(int argc, char** argv)
{
  int s __attribute__((stream));
  int view[5];

  #pragma omp task output(s << view[5])
  {
    for(int i = 0; i < 5; i++) {
      view[i] = i*i;
      printf("Producer writes %d\n", view[i]);
    }
  }

  #pragma omp task input(s >> view[5])
  {
    for(int i = 0; i < 5; i++) {
      printf("Consumer reads %d\n", view[i]);
  }

  #pragma omp taskwait
  return 0;
}
```

Recompile and execute the program. The run-time has been configured to generate a trace file named `events.ost` at each execution of an OpenStream program. Open the trace file from the last execution, locate the producer and the consumer task and verify how much data was exchanged between them.

(d) Modify the last program, such that it generates 100 pairs of producers and consumers. Execute the program and open the trace file. Activate rendering of work-stealing events by clicking on the tool bar button with the orange dots and lines. On the time line, each work-stealing event is now rendered with a dot on the worker from whom the task was stolen and a line ending at the worker that stole the task. Is there a pattern for work-stealing events in this program?

(e) Set the communication incidence matrix to CPU mode by clicking on the *CPUs* radio button below the matrix. Uncheck the two check boxes labeled *R* and *W* and check the *S* box in order to visualize work-stealing events. To count the number of events, also check the box labeled *# events only*. Are there any work-stealing events for which the worker on core 0 is not the victim?

Tool bar buttons

Draw annotations	Draw performance counters
Draw measurement intervals	Draw states
Show all cores	Limit view to measurement interval

OpenMP

Activate OpenMP mode (select specific mode from drop down menu)

OpenStream

Draw task creation / destruction / execution events	Indicate producers of the selected task on the time line
Draw work-pushing events	Draw read accesses to streams
Draw write accesses to streams	Indicate the size of stream accesses
Draw work-stealing events	Activate heat map mode
Activate NUMA read map mode	Activate NUMA write map mode
Activate NUMA heat map mode	Activate task type mode

References

All papers, documentation and the virtual machine image can be found online at the main website `https://www.aftermath-tracing.com/`

[1] Andi Drebes, Jean-Baptiste Bréjon, Antoniu Pop, Karine Heydemann, and Albert Cohen. Language-Centric Performance Analysis of OpenMP Programs with Aftermath. In *IWOMP '16*, 2016.

[2] Andi Drebes, Antoniu Pop, Karine Heydemann, and Albert Cohen. Interactive Visualization of Cross-Layer Performance Anomalies in Dynamic Task-Parallel Applications and Systems. In *ISPASS '16*, April 2016.

[3] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach-Temam. Aftermath: A Graphical Tool for Performance Analysis and Debugging of Fine-Grained Task-Parallel Programs and Run-time Systems. In *MULTIPROG '14*, 2014.