Performance analysis
CSC5001 – Systèmes Hautes Performances
Summary

• Why / when to analyze performances ?
• How to evaluate the performances of an application ?
• Tools for performance analysis
Why/when to analyze performance?

- **Why?**
  - In order to reduce the application execution time and/or memory consumption
  - Supercomputers are expensive to operate
    - Before buying a more powerful one you’d better use the current one efficiently
  - To solve a problem in a reasonable amount of time

- **When?**
  - Once the application works
Why NOT to optimize performance?

"Premature optimization is the root of all evil" – Knuth, D. E. The art of computer programming

- Drawbacks of optimizing applications
  - It consumes lots of developer time
    - Should I spend 6 months optimizing an application in order to improve its completion time by 3%?
  - The source code becomes hard to maintain
  - The optimization targets one hardware platform
    - It may degrade performance on other platforms
How to evaluate the performance?
Algorithm complexity

• Parallel complexity depends on
  − N: the problem size
  − P: the number of processors

• Estimate the asymptotic complexity of the algorithm
  − If N >> P, improving the algorithm is more important than improving the parallelization
  − eg O(N^2 / P) > O(N log (N) / ½P)

• Beware of the hidden constant
  − If N is small, O(N^2) ~= O(N log(N))
Measuring the application scalability

- Find a performance metric that suits the application
  - Application whole execution time
  - Application run time (without the initialization)
  - Throughput / response time

- Fairly compare the sequential and parallel codes
  - Compare source codes with similar level of optimization
  - "On the Limits of GPU Acceleration", Richard Vuduc et al. HotPar 2010

Example of (possible) unfair comparison:
Comparing a matlab implementation with a highly tuned CUDA implementation

"Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In IPDPS 2009"
Strong scaling vs weak scaling

- **Strong scaling study**
  - Study how performance scales for a fixed problem size
  - How to solve problems faster?
  - Ultimately, the computation becomes too small, and performance degrades

- **Weak scaling study**
  - Study how performance scales with a constant problem size per processor
  - How to solve bigger problems?
Sources of performance issues

- Problem size is too small
  - cf. strong scaling study
- The application lacks parallelism
  - eg. only a part of the application is parallel, workload imbalance, ...
- Bottleneck on a shared resource
  - eg. IO on a disk, concurrent access to the network, shared lock, ...
- Bad memory usage
  - eg. lots of cache misses, memory accesses on remote NUMA nodes, false sharing, ...
- ...
Tools for performance analysis
Very coarse grain performance analysis

**time**

- Outputs timing statistics for executing a command.
  - **Real**: time difference between the start date and the end date
  - **User**: total CPU time consumed by thread in user space
  - **Sys**: total CPU time consumed by thread in kernel space

- Can be used for:
  - Computing speedup
  - Detecting I/O intensive applications (if **sys** is high)
  - Detecting a lack of parallelism (**user** should be roughly **real**\***nprocs**)
Coarse grain performance analysis

Profiling tools (eg perf)

- Show which functions takes most of the CPU time
- Collecting samples
  - Use the CPU sampling mechanism to know which instruction is being executed
  - Can record the callgraph (see -g )
- Many other cpu profilers exist
  - gprof, oprofile, valgrind, ...

```
$ perf record ./bin/dc.W.x
...
[ perf record: Woken up 21 times to write data ]
[ perf record: Captured and wrote 5,637 MB perf.data (147114 samples) ]
$ perf report
```

<table>
<thead>
<tr>
<th>Overhead</th>
<th>Command</th>
<th>Shared</th>
<th>Project</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>33.34%</td>
<td>dc.W.x</td>
<td>dc.W.x</td>
<td>KeyComp</td>
<td></td>
</tr>
<tr>
<td>27.02%</td>
<td>dc.W.x</td>
<td>dc.W.x</td>
<td>TreeInsert</td>
<td></td>
</tr>
<tr>
<td>6.45%</td>
<td>dc.W.x</td>
<td>dc.W.x</td>
<td>WriteViewToToolsKCS</td>
<td></td>
</tr>
<tr>
<td>6.24%</td>
<td>dc.W.x</td>
<td>llibc-2.31.so</td>
<td>_IO_fread</td>
<td></td>
</tr>
<tr>
<td>4.96%</td>
<td>dc.W.x</td>
<td>llibc-2.31.so</td>
<td>_IO_fwrite</td>
<td></td>
</tr>
<tr>
<td>3.23%</td>
<td>dc.W.x</td>
<td>llibc-2.31.so</td>
<td>_memmove_avx_unaligned_erns</td>
<td></td>
</tr>
<tr>
<td>1.84%</td>
<td>dc.W.x</td>
<td>[kernel.kallsyms]</td>
<td>_IO_readtext</td>
<td></td>
</tr>
<tr>
<td>1.58%</td>
<td>dc.W.x</td>
<td>dc.W.x</td>
<td>SelectToView</td>
<td></td>
</tr>
<tr>
<td>1.47%</td>
<td>dc.W.x</td>
<td>[kernel.kallsyms]</td>
<td>do_syscall_64</td>
<td></td>
</tr>
<tr>
<td>1.41%</td>
<td>dc.W.x</td>
<td>llibc-2.31.so</td>
<td>_IO_file_xattr</td>
<td></td>
</tr>
<tr>
<td>0.94%</td>
<td>dc.W.x</td>
<td>llibc-2.31.so</td>
<td>_IO_file_xputstr@GLIBC_2.2.5</td>
<td></td>
</tr>
<tr>
<td>0.70%</td>
<td>dc.W.x</td>
<td>dc.W.x</td>
<td>RunFormation</td>
<td></td>
</tr>
<tr>
<td>0.43%</td>
<td>dc.W.x</td>
<td>[kernel.kallsyms]</td>
<td>clear_page_erns</td>
<td></td>
</tr>
<tr>
<td>0.35%</td>
<td>dc.W.x</td>
<td>[kernel.kallsyms]</td>
<td>syscall_return_via_sysret</td>
<td></td>
</tr>
<tr>
<td>0.34%</td>
<td>dc.W.x</td>
<td>[kernel.kallsyms]</td>
<td>entry_SYSCALL_64</td>
<td></td>
</tr>
<tr>
<td>0.20%</td>
<td>dc.W.x</td>
<td>[kernel.kallsyms]</td>
<td>_list_del_entry_valid</td>
<td></td>
</tr>
</tbody>
</table>
Coarse grain performance analysis

Performance counters (eg \texttt{perf stat})

- Performance counters are collected during the execution
  - Hardware events (eg branch-misses, cpu-cycle, ...)
  - Software events (eg context-switches, page-faults, ...)
  - Low level counters (eg LLC-load-misses, power/energy-pkg/,...)
    
    → see \texttt{perf list}

\texttt{perf stat -e c1,c2,c3,... cmd}
Fine grain performance analysis

**clock_gettime**

- Manual timing of parts of the code
  - Precise timing/variation measurement
- Need a clock
  - gettimeofday()
    - Precision: 1µs, overhead: 20 ns
  - clock_gettime()
    - Precision: 1 ns, overhead: 10-200 ns
  - RDTSC assembly instruction
    - Precision: 1 cycle, overhead: 6-7 ns
  - Logical clock (eg. _Atomic int clock=0; )

```c
/* collect samples for all the threads */
for(int i=0; i<nthreads; i++) {
    size_t read_size, write_size;
    copied_size = 0;
    get_tick(&t1);
    numap_sampling_read_stop(thread_ranks[i].sm);
    get_tick(&t2);
    __process_samples(thread_ranks[i].sm, ACCESS_READ);
    get_tick(&t3);
    read_size = copied_size;
    numap_sampling_resume(thread_ranks[i].sm);
    get_tick(&t4);

    read_stop_duration += time_diff(t1, t2);
    process_samples_duration += time_diff(t2, t3);
    sampling_resume += time_diff(t3, t4);
}
```
**Fine grain performance analysis**

**tracing tools**

- Dynamic representation of the program behavior
- Execution trace:
  - Timestamped list of events

```plaintext
<table>
<thead>
<tr>
<th>ENTRY</th>
<th>Time</th>
<th>Value</th>
<th>Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEAVE</td>
<td>4294967308</td>
<td>5974522527014944</td>
<td>Read &lt;75&gt;</td>
</tr>
<tr>
<td>ENTRY</td>
<td>4294967308</td>
<td>5974522527016448</td>
<td>&quot;read&quot; &lt;78&gt;</td>
</tr>
<tr>
<td>ENTRY</td>
<td>4294967308</td>
<td>5974522527021184</td>
<td>&quot;ftime&quot; &lt;95&gt;</td>
</tr>
<tr>
<td>IO_OPERATION_BEGIN</td>
<td>4294967308</td>
<td>5974522527023616</td>
<td>Handle: &quot;stdout&quot; &lt;5&gt;, Mode: FLUSH, Operation Flags: NONE, Bytes Request: 1</td>
</tr>
<tr>
<td>METRIC</td>
<td>4294967308</td>
<td>5974522527025988</td>
<td>Metric: 0, 1 Value: (&quot;failed_operations&quot; &lt;10&gt;; UINT64; 0)</td>
</tr>
<tr>
<td>MPI_RECV_REQUEST</td>
<td>14</td>
<td>5974522527026021</td>
<td>Request: 6</td>
</tr>
<tr>
<td>IO_OPERATION_COMPLETE</td>
<td>4294967308</td>
<td>5974522527027612</td>
<td>Handle: &quot;stdout&quot; &lt;5&gt;, Bytes Result: 18446744073709551615, Matching Id: 5</td>
</tr>
<tr>
<td>LEAVE</td>
<td>4294967308</td>
<td>5974522527028476</td>
<td>&quot;ftimeflush&quot; &lt;95&gt;</td>
</tr>
<tr>
<td>ENTRY</td>
<td>4294967308</td>
<td>5974522527029540</td>
<td>&quot;ftimefsync&quot; &lt;66&gt;</td>
</tr>
<tr>
<td>LEAVE</td>
<td>14</td>
<td>5974522527029801</td>
<td>&quot;MPI_RECV_request&quot; &lt;70&gt;</td>
</tr>
<tr>
<td>IO_OPERATION_BEGIN</td>
<td>4294967308</td>
<td>5974522527031152</td>
<td>&quot;STDOUT_FILENO&quot; &lt;1&gt;, Mode: FLUSH, Operation Flags: NONE, Bytes Request: 1</td>
</tr>
<tr>
<td>METRIC</td>
<td>4294967308</td>
<td>5974522527033128</td>
<td>Metric: 0, 1 Value: (&quot;failed_operations&quot; &lt;10&gt;; UINT64; 1)</td>
</tr>
<tr>
<td>IO_OPERATION_COMPLETE</td>
<td>4294967308</td>
<td>5974522527035560</td>
<td>&quot;STDOUT_FILENO&quot; &lt;1&gt;, Bytes Result: 18446744073709551615, Matching Id: 5</td>
</tr>
<tr>
<td>LEAVE</td>
<td>4294967308</td>
<td>5974522527033816</td>
<td>&quot;ftimefsync&quot; &lt;66&gt;</td>
</tr>
<tr>
<td>ENTRY</td>
<td>14</td>
<td>5974522527036481</td>
<td>&quot;MPI_Isend&quot; &lt;273&gt;</td>
</tr>
<tr>
<td>ENTER</td>
<td>4294967298</td>
<td>5974522527039666</td>
<td>&quot;read&quot; &lt;78&gt;</td>
</tr>
<tr>
<td>MPI_RECV_REQUEST</td>
<td>5</td>
<td>5974522527039768</td>
<td>Request: 1</td>
</tr>
<tr>
<td>MPI_RECV_REQUEST</td>
<td>6</td>
<td>5974522527039973</td>
<td>Request: 1</td>
</tr>
<tr>
<td>LEAVE</td>
<td>4294967298</td>
<td>5974522527043182</td>
<td>&quot;read&quot; &lt;78&gt;</td>
</tr>
</tbody>
</table>
```
Fine grain performance analysis
visualizing execution traces

- Graphical representation of the application behavior
Fine grain performance analysis
tracing tools: EZTrace

- List the available modules
  - `eztrace_avail`

- Collecting events
  - `eztrace` or `eztrace.preload`
  - Generates one file per MPI rank (`${USER}_eztrace_log_rank*`)

- Visualizing the trace
  - `eztrace_convert ${USER}_eztrace_log_rank_*`
  - `vite eztrace_output.trace`
Fine grain performance analysis
EZTrace internals

- Functions instrumentation
  - Uses LD_PRELOAD to intercept calls to a set of functions
- Recording events
  - Events are stored in thread-local buffers at runtime
  - Buffers are flushed at the end or when full
- Caveats
  - OpenMP events: need to recompile the application with eztrace_cc:
    ```
    $ eztrace_cc gcc -o my_app my_app.c -fopenmp
    ```
  - Online tutorials: https://eztrace.gitlab.io/eztrace/tutorials/index.html