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 \( O(N^2 / P) > O(N \log(N) / \frac{1}{2}P) \)

- Beware of the *hidden constant*
  - If \( N \) is small, \( O(N^2) \sim O(N \log(N)) \)
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)

```bash
$ time ./bin/dc.W.x
...
real    0m9,745s
user    0m31,930s
sys     0m3,509s
```
Coarse grain performance analysis
Profiling tools (eg perf)

- Show which functions take 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>Samples: 152K of event 'cycles', Event count (approx.): 13369733175</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>33.34%</td>
</tr>
<tr>
<td>27.92%</td>
</tr>
<tr>
<td>6.45%</td>
</tr>
<tr>
<td>6.24%</td>
</tr>
<tr>
<td>4.96%</td>
</tr>
<tr>
<td>3.13%</td>
</tr>
<tr>
<td>1.51%</td>
</tr>
<tr>
<td>1.47%</td>
</tr>
<tr>
<td>1.41%</td>
</tr>
<tr>
<td>0.94%</td>
</tr>
<tr>
<td>0.78%</td>
</tr>
<tr>
<td>0.43%</td>
</tr>
<tr>
<td>0.35%</td>
</tr>
<tr>
<td>0.34%</td>
</tr>
<tr>
<td>0.25%</td>
</tr>
</tbody>
</table>

12
Coarse grain performance analysis
Performance counters (eg `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 perf list

```
$ perf stat ./bin/dc.W.x
...
```

```
Performance counter stats for `./bin/dc.W.x`:

35 077.00 msec task-clock
6 632 context-switches
63 cpu-migrations
9 455 page-faults
125 532 103 063 cycles
165 745 603 282 instructions
39 906 372 015 branches
423 268 662 branch-misses

10,550839085 seconds time elapsed
31,532447080 seconds user
3,549348080 seconds sys
```

perf stat -e c1,c2,c3,... cmd
Fine grain performance analysis

clock_gettimee

- 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; )
Fine grain performance analysis
tracing tools

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

```
ENTRY 4294967308 5974522527014944 Region: "read" <78>
LEAVE 4294967308 5974522527016448 Region: "read" <78>
ENTRY 4294967308 5974522527021184 Region: "fflush" <95>
IO_OPERATION_BEGIN 4294967308 5974522527023616 Handle: "stdout" <5>, Mode: FLUSH, Operation Flags: NONE, Bytes Request: 1
METRIC 4294967308 5974522527025980 Metric: 0, 1 Value: ("failed_operations" <1>; U32: 0)
MPI_RECV_REQUEST 14 5974522527026021 Request: 6
IO_OPERATION_COMPLETE 4294967308 5974522527027612 Handle: "stdout" <5>,Bytes Result: 18446744073709551615, Matching Id: 5
LEAVE 4294967308 5974522527028476 Region: "fflush" <95>
ENTRY 4294967308 5974522527029058 Region: "fsync" <86>
LEAVE 14 5974522527029801 Region: "MPI_Irecv" <270>
IO_OPERATION_BEGIN 4294967308 5974522527031152 Handle: "STDOUT_FILENO" <1>, Mode: FLUSH, Operation Flags: NONE, Bytes Request: 1
METRIC 4294967308 5974522527033128 Metric: 0, 1 Value: ("failed_operations" <1>; U32: 1)
IO_OPERATION_COMPLETE 4294967308 5974522527035360 Handle: "STDOUT_FILENO" <1>,Bytes Result: 18446744073709551615, Matching Id: 5
LEAVE 4294967308 5974522527038386 Region: "fsync" <86>
ENTRY 14 5974522527038481 Region: "MPI_Isend" <273>
ENTER 4294967298 5974522527039666 Region: "read" <78>
MPI_RECV_REQUEST 5 5974522527039978 Request: 1
MPI_RECV_REQUEST 6 5974522527043102 Request: 1
LEAVE 4294967298 5974522527043102 Region: "read" <78>
```
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
  
  ```bash
  $ eztrace_avail
  3 stdio Module for stdio functions (read, write, select, poll, etc.)
  2 pthread Module for PThread synchronization functions (mutex, semaphore, spinlock, etc.)
  6 papi Module for PAPI Performance counters
  1 omp Module for OpenMP parallel regions
  4 mpi Module for MPI functions
  5 memory Module for memory functions (malloc, free, etc.)
  7 cuda Module for cuda functions (cuMemAlloc, cuMemcpy, etc.)
  ```

  ```bash
  $ eztrace -t ''module1 module2'' ./mon_programme
  $ mpirun -np 2 eztrace -t ''module1 module2'' ./mon_programme
  $ 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
    ```

    - [EZTrace] The buffer for recording events is full. Stop recording. The trace will be truncated

    ```
    $ export EZTRACE_BUFFER_SIZE=1073741824
    or
    $ export EZTRACE_FLUSH=1
    ```

    - Online tutorials: http://eztrace.gforge.inria.fr/tutorials/index.html