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 reasonnable 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 developper time
    - Should I spend 6 month 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

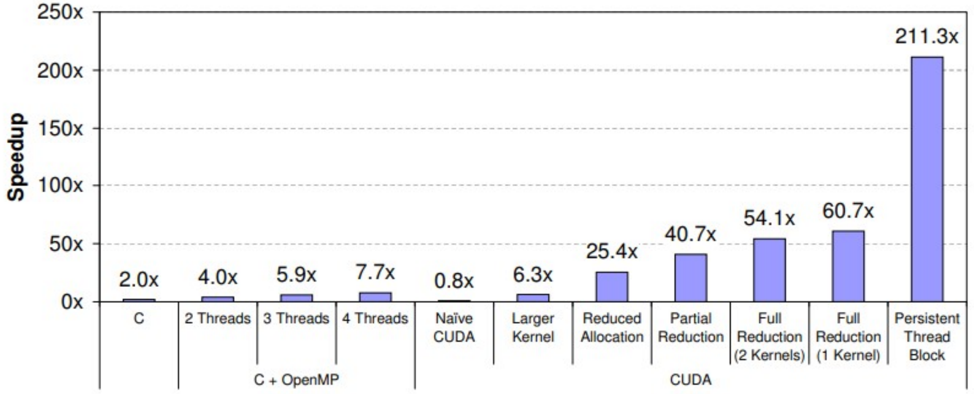


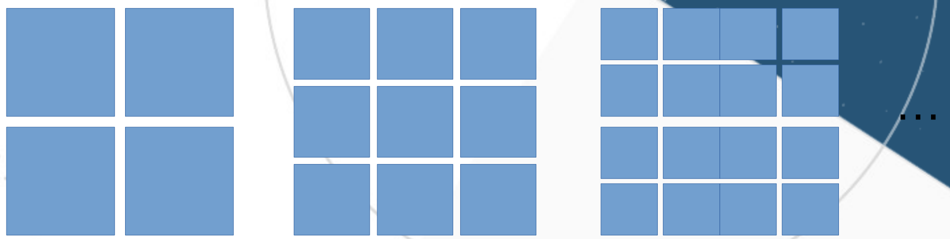
Figure 3. Speedup of the different implementations of the tracking stage over the original MATLAB implementation

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

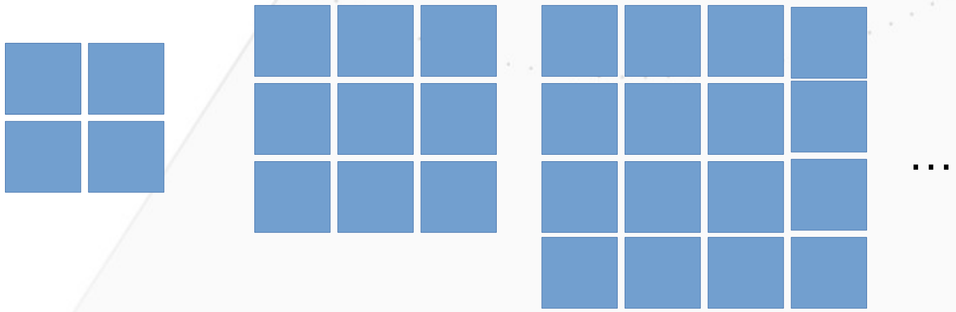
Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In IPDPS 2009

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

```
$ time ./bin/dc.W.x
…
real     0m9,745s
user     0m31,930s
sys      0m3,509s
```

- 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 (1

$ perf report
```

```
Samples: 152K of event 'cycles', Event count (approx.): 133697331175
Overhead  Command   Shared Object        Symbol
  33,34%  dc.W.x    dc.W.x               [.] KeyComp
  27,02%  dc.W.x    dc.W.x               [.] TreeInsert
   6,45%  dc.W.x    dc.W.x               [.] WriteViewToDiskCS
   6,24%  dc.W.x    libc-2.31.so         [.] _IO_fread
   4,96%  dc.W.x    libc-2.31.so         [.] _IO_fwrite
   3,13%  dc.W.x    libc-2.31.so         [.] __memmove_avx_unaligned_erms
   1,84%  dc.W.x    [kernel.kallsyms]    [k] copy_user_enhanced_fast_string
   1,78%  dc.W.x    dc.W.x               [.] SelectToView
   1,07%  dc.W.x    [kernel.kallsyms]    [k] do_syscall_64
   1,01%  dc.W.x    libc-2.31.so         [.] _IO_file_xsgetn
   0,94%  dc.W.x    libc-2.31.so         [.] _IO_file_xsputn@@GLIBC_2.2.5
   0,78%  dc.W.x    dc.W.x               [.] RunFormation
   0,43%  dc.W.x    [kernel.kallsyms]    [k] clear_page_erms
   0,35%  dc.W.x    [kernel.kallsyms]    [k] syscall_return_via_sysret
   0,34%  dc.W.x    [kernel.kallsyms]    [k] entry_SYSCALL_64
   0,25%  dc.W.x    [kernel.kallsyms]    [k] __list_del_entry_valid
```

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 -e c1,c2,c3,... cmd
```

```
$ perf stat ./bin/dc.W.x
…
```

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

      35 077,08 msec task-clock                #    3,322 CPUs utilized
          6 652      context-switches          #    0,190 K/sec
             63      cpu-migrations            #    0,002 K/sec
          9 455      page-faults               #    0,270 K/sec
125 532 103 063      cycles                    #    3,579 GHz
165 745 603 282      instructions              #    1,32  insn per cycle
 39 986 372 815      branches                  # 1139,957 M/sec
    423 268 662      branch-misses             #    1,06% of all branches

   10,558830905 seconds time elapsed

   31,532447000 seconds user
    3,549348000 seconds sys
```

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;`)

```
/* 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);
```

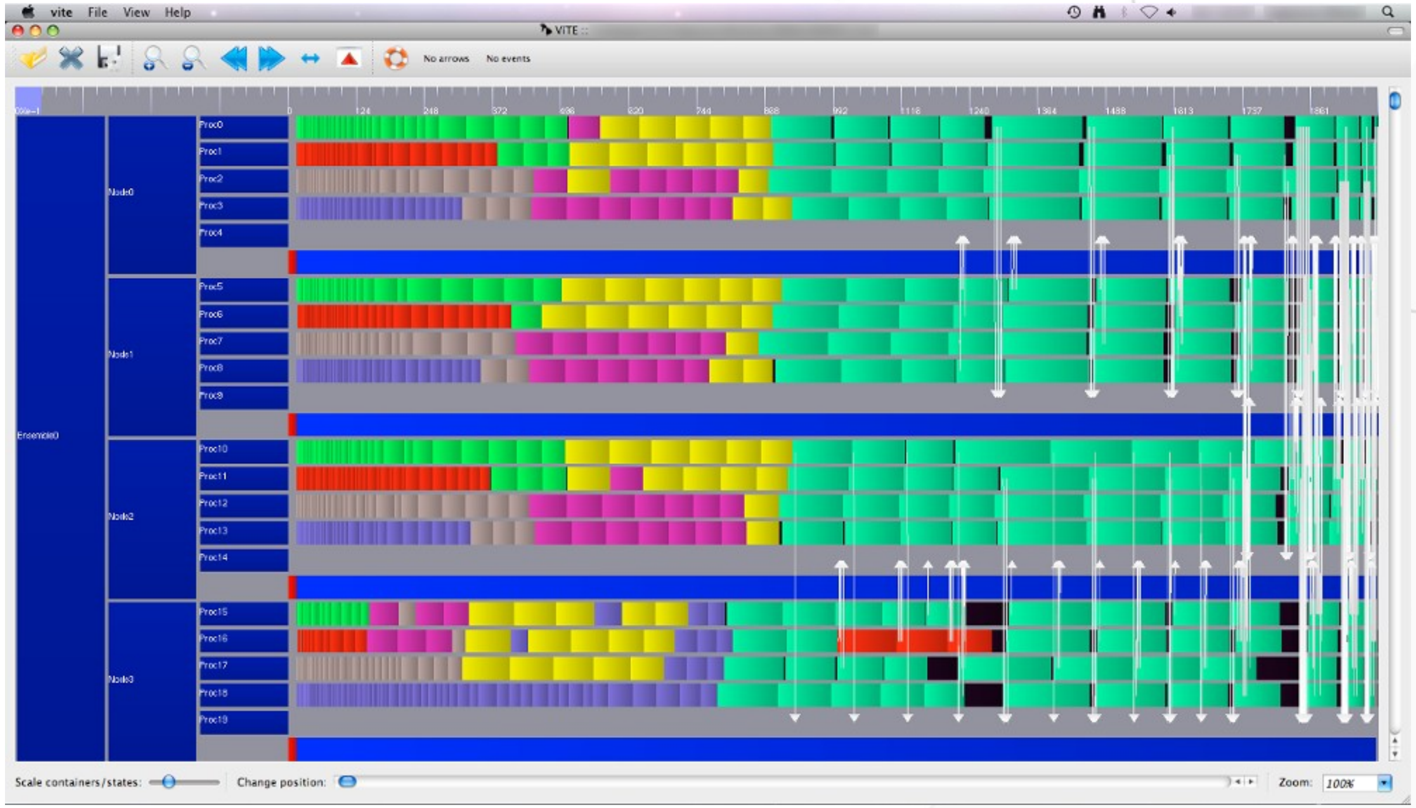
Fine grain performance analysis
tracing tools

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

```
ENTER                              4294967308     5974522527014944  Region: "read" <78>
LEAVE                              4294967308     5974522527016448  Region: "read" <78>
ENTER                              4294967308     5974522527021184  Region: "fflush" <95>
IO_OPERATION_BEGIN                 4294967308     5974522527023616  Handle: "stdout" <5>, Mode: FLUSH, Operation Flags: NONE, Bytes Request: 1
METRIC                             4294967308     5974522527025988  Metric: 0, 1 Value: ("failed_operations" <10>; UINT64; 0)
MPI_IRECV_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>
ENTER                              4294967308     5974522527029540  Region: "fsync" <66>
LEAVE                                      14     5974522527029801  Region: "MPI_Irecv" <270>
IO_OPERATION_BEGIN                 4294967308     5974522527031152  Handle: "STDOUT_FILENO" <1>, Mode: FLUSH, Operation Flags: NONE, Bytes Req
METRIC                             4294967308     5974522527033120  Metric: 0, 1 Value: ("failed_operations" <10>; UINT64; 1)
IO_OPERATION_COMPLETE              4294967308     5974522527033560  Handle: "STDOUT_FILENO" <1>, Bytes Result: 18446744073709551615, Matching
LEAVE                              4294967308     5974522527033816  Region: "fsync" <66>
ENTER                                      14     5974522527038481  Region: "MPI_Isend" <273>
ENTER                              4294967298     5974522527039666  Region: "read" <78>
MPI_IRECV_REQUEST                           5     5974522527039788  Request: 1
MPI_IRECV_REQUEST                           6     5974522527039973  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`

```
$ 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, cuMemcopy, etc.)
```

- Collecting events
  - `eztrace` or `eztrace.preload`
  - Generates an OTF2 trace file (`<PROGRAM>_trace/eztrace_log.otf2`)

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

- Visualizing the trace

```
$ vite program_trace/eztrace_log.otf2
$ otf2-print  program_trace/eztrace_log.otf2
```

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

```
int pthread_mutex_lock(pthread_mutex_t * mutex) {
  FUNCTION_ENTRY;
  EZTRACE_EVENT_PACKED_1(EZTRACE_MUTEX_LOCK_START, (app_ptr)mutex);
  int ret = libpthread_mutex_lock(mutex);
  EZTRACE_EVENT_PACKED_2(EZTRACE_MUTEX_LOCK_STOP, (app_ptr)mutex, ret);
  return ret;
}
```

- Caveats
  - `openmp` plugin: need to recompile the application with `eztrace_cc` :

    ```
    $ eztrace_cc gcc -o my_app my_app.c -fopenmp
    ```
  - `ompt` plugin: only works with OpenMP implementation that implement the OMPT interface (eg. clang)
  - Online tutorials: `https://gitlab.com/eztrace/eztrace-tutorials/`