UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Computing

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Shared memory: timing & profiling

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Timing

Timing

- A simple approach to profile code is to add timers.
- Let us consider C code, using standard library.
- We need to evaluate how much time was needed to execute the sequential and parallel versions of our code.
- In `<time.h>` we find `clock()` and `time()`, and in `<sys/time.h>` we have `gettimeofday()`

clock()

- Returns the approximate **processor time** used by the process since the beginning of an implementation-defined era related to the program's execution. To convert result value to seconds, divide it by CLOCKS_PER_SEC.

- Only the difference between two values returned by different calls to clock is meaningful, as the beginning of the clock era does not have to coincide with the start of the program.

clock()

In a sequential program processor time ≅ program execution time.

- Returns the approximate **processor time** used by the process since the beginning of an implementation-defined era related to the program's execution. To convert result value to seconds, divide it by CLOCKS_PER_SEC.

- Only the difference between two values returned by different calls to clock is meaningful, as the beginning of the clock era does not have to coincide with the start of the program.

clock()

- In a parallel program `clock()` returns the total CPU time spent in all threads. If you run with four threads and each runs for 1/4 of the time of a sequential program, clock() will still return the same value since 4*(1/4) = 1

- Often a thread has an overhead cost, so when using 4 threads the CPU time measured will be >1/4 sequential time and clock will return a higher time difference for the parallel program than the sequential !

Do **NOT** use clock() to compare sequential and parallel execution time !

- In a parallel program `clock()` returns the total CPU time spent in all threads. If you run with four threads and each runs for 1/4 of the time of a sequential program, clock() will still return the same value since 4*(1/4) = 1

Lui misura quanto le cpu sono occupate

- Often a thread has an overhead cost, so when using 4 threads the CPU time measured will be >1/4 sequential time and clock will return a higher time difference for the parallel program than the sequential !

Elapsed real-time

- We need to evaluate how much time has passed, i.e. elapsed real-time, also known as: **wall-clock time**.

- Use `gettimeofday()` or its alternatives to get the wall clock.

- `clock` time may advance faster or slower than the **wall clock**, depending on the execution resources given to the program by the operating system.
  E.g., if the CPU is shared by other processes, `clock` time may advance slower than wall clock. On the other hand, if the current process is multithreaded and more than one execution core is available, clock time may advance faster than wall clock.

gettimeofday()

- The `gettimeofday` system call gets the system's wall-clock time.
  It obtains the current time, expressed as seconds and microseconds; the resolution of the system clock is unspecified.

- `time()` returns the wall-clock time from the OS, with precision in seconds.

According to POSIX standard **`gettimeofday`** is become obsolete and should be substituted with **`clock_gettime`**, that has nanosecond resolution.

gettimeofday()

- The `gettimeofday` system call gets the system's wall-clock time.
  It obtains the current time, expressed as seconds and microseconds; the resolution of the system clock is unspecified.

- `time()` returns the wall-clock time from the OS, with precision in seconds.

According to POSIX standard **`gettimeofday`** is become obsolete and should be substituted with **`clock_gettime`**, that has nanosecond resolution.

Hint: check **`clock_gettime`** if available (e.g. it was introduced in macOS only in 10.12) otherwise fallback to **`gettimeofday`** or use a O.S. specific high res timer

Alternatives

- C++11 has introduced `std::chrono` with `::high_resolution_clock`
- OpenMP has a practical `omp_get_wtime()` which returns a double value of the number of seconds since an arbitrary point in the past.

C'è il wall clock time

UNIVERSITÀ
DEGLI STUDI
FIRENZE

Profiling

Profiling

- Profiling is a technique that helps to identify, among other problems, which parts of our program are slower and perhaps could be improved by parallelization. Non tutte le parti del codice vanno parlallelizzate

- There are several tools to identify how much time is spent in different parts of our programs. Typically they sample periodically the program, recording how many times an instruction has been executed and how much time it took to complete.
  This info is logged to be analyzed at a later stage.

Profiling tools

- Depending on the language and framework used in the programs to be profiled we can chose the appropriate profiling tools
  - E.g. VisualVM or Yourkit for Java, Scalasca for OpenMP and MPI, VTune for OpenMP, C++, NVProf for CUDA, Google Perftools and Callgrind for C/C++, Pthreads…
- There are different GUIs to analyze/visualize profiling results
  - E.g. IntelliJ IDEA for VisualVM and Yourkit, VTune itself, NVVP for CUDA, Cachegrind for Callgrind…

Profiling tools

- Depending on the language and framework used in the programs to be profiled we can chose the appropriate profiling tools
  - E.g. VisualVM or Yourkit for Java, Scalasca for OpenMP and MPI, VTune for OpenMP, C++, NVProf for CUDA, Google Perftools and Callgrind for C/C++, Pthreads…

For the sake of simplicity we are going to see only C/C++ profilers

- There are different GUIs to analyze/visualize profiling results
  - E.g. IntelliJ IDEA for VisualVM and Yourkit, VTune itself, NVVP for CUDA, Cachegrind for Callgrind…

Google perftools

- Google `pprof` is a tool available as part of the Google perftools package. It is is used with `libprofiler`, a sampling based profiler that is linked into your binary.
- It can be used in Linux and macOS.
- For Windows use other tools, e.g. Intel VTune.

CPU profiler

- After installing perftools:

1. Link your executable with -lprofiler

2. Run your executable with the CPUPROFILE environment var set:
```
CPUPROFILE=/tmp/prof.out <path/to/binary>
[binary args]
```

3. Run `pprof` to analyze the CPU usage logged in the `/tmp/prof.out` file:
```
pprof --text <path/to/binary> /tmp/
prof.out
```

CPU profiler

- After installing perftools:

1. Link your executable with -lprofiler

2. Run your executable with the CPUPROFILE environment var set:
   `CPUPROFILE=/tmp/prof.out <path/to/binary> [binary args]`

   Set sampling rate (default: 100Hz) with CPUPROFILE_FREQUENCY

3. Run `pprof` to analyze the CPU usage logged in the `/tmp/prof.out` file:
   `pprof --text <path/to/binary> /tmp/prof.out`

CPU profiler

- After installing perftools:

1. Hint: just add the library at execution time (no need to change build!), adding:
   `LD_PRELOAD=/path/to/libprofiler.so.0` (Linux)
   `DYLD_INSERT_LIBRARIES=/path/to/libprofiler.dylib` (macOS)
   to the line used to profile the program
2. Run your executable with the CPUPROFILE environment var set:
   `CPUPROFILE=/tmp/prof.out <path/to/binary> [binary args]`
   Set sampling rate (default: 100Hz) with CPUPROFILE_FREQUENCY
3. Run `pprof` to analyze the CPU usage logged in the `/tmp/prof.out` file:
   `pprof --text <path/to/binary> /tmp/prof.out`

Example

```
pprof --text ./a.out out.prof

... <snip> ...

Total: 311 samples
```

Mi dicono quali funzoni ci mettono di più

```
  144  46.3%  46.3%      144  46.3% bar
   95  30.5%  76.8%       95  30.5% foo
   72  23.2% 100.0%      311 100.0% baz
    0   0.0% 100.0%      311 100.0% __libc_start_main
    0   0.0% 100.0%      311 100.0% _start
    0   0.0% 100.0%      311 100.0% main
```

Example

In macOS the function names are not shown. The program has to be compiled disabling the address space layout randomization (ASLR) introduced in OS X 10.5

Add `-Wl,-no_pie` to the C/CXX flags.

```
Total: 311 samples

   144  46.3%  46.3%      144  46.3% bar
    95  30.5%  76.8%       95  30.5% foo
    72  23.2% 100.0%      311 100.0% baz
     0   0.0% 100.0%      311 100.0% __libc_start_main
     0   0.0% 100.0%      311 100.0% _start
     0   0.0% 100.0%      311 100.0% main
```

Analyzing Text Output

- Text mode has lines of output that look like this:
- `14    2.1%  17.2%        58    8.7% foo_bar`
- The columns have the following meaning:
  - Number of profiling samples in this function
  - Percentage of profiling samples in this function
  - Percentage of profiling samples in the functions printed so far
  - Number of profiling samples in this function and its callees
  - Percentage of profiling samples in this function and its callees
  - Function name

Single procedure / lines

- It is possible to analyze a single function or get line by line analysis with:
  - `–list=function_name`
  - `–lines` (default report is at function level)

Example

```
pprof --list=thread_scan a.out a.out.prof

 ... <snip> ...

   .      .   60: void* thread_scan(void* void_arg) {

   .      .   61:    // TODO(awreece) Copy locally so don't interfere with each other.

   .      .   62:  thread_arg_t* args = (thread_arg_t*) void_arg;

   .      .   63:  size_t i;

   .      .   64:

 303    323   65:  for (i = 0; i < arg->size; i++) {

   6     10   66:     uint32_t val = arg->input[i];

   6     15   67:   if (val % 2 == 0) {

   9    300   68:     __sync_fetch_and_add(args->evens, 1);

   .      .   69:   }

   .      .   70:  }

   .      .   71: }
```

Valgrind (Callgrind)

- Alternatively it is possible to use the Callgrind component of Valgrind
  - It is (very much) slower than Google Perftools, but it is associated with nice GUIs to inspect the results (xCachegrind for QT/KDE)
  - It is very accurate
  - Like GPerftools does not require to recompile/link libraries, it instruments the executable

Running Callgrind

- `valgrind --tool=callgrind <path/to/binary>`

- It will create a `callgrind.out.<pid>` containing the sampling data, reporting how many “instructions read” events have been collected

- Analyze it using a xCachegrind GUI or

- Use `callgrind_annotate` command line tool to inspect it

Running Callgrind

- `valgrind --tool=callgrind <path/to/binary>`

- It will create a `callgrind.out.<pid>` containing the sampling data, reporting how many “instructions read” events have been collected

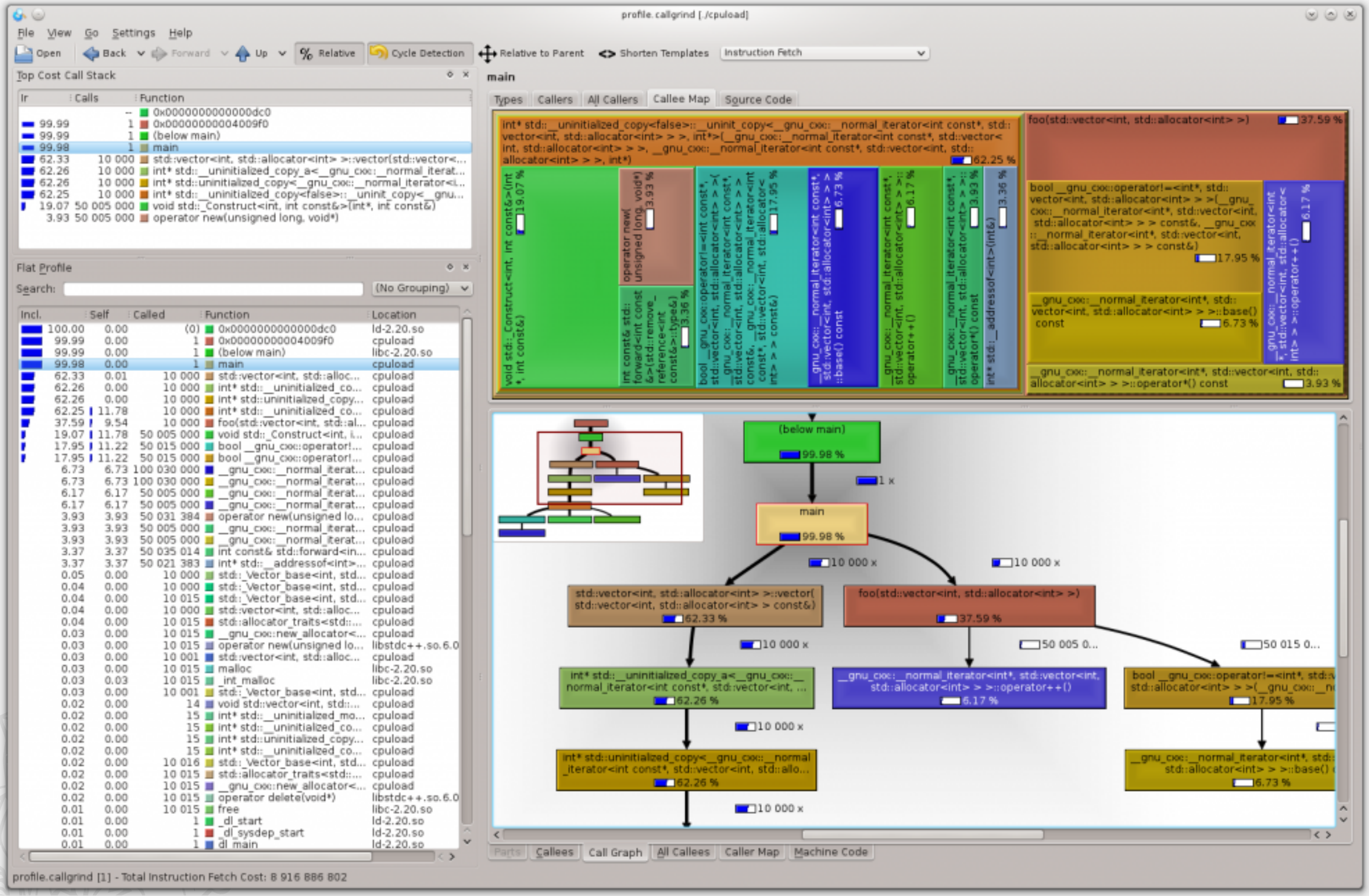
- Analyze it using a xCachegrind GUI or

- Use `callgrind_annotate` command line tool to inspect it

Google Perftools provide Callgrind annotations format. Use `–callgrind` output specifier

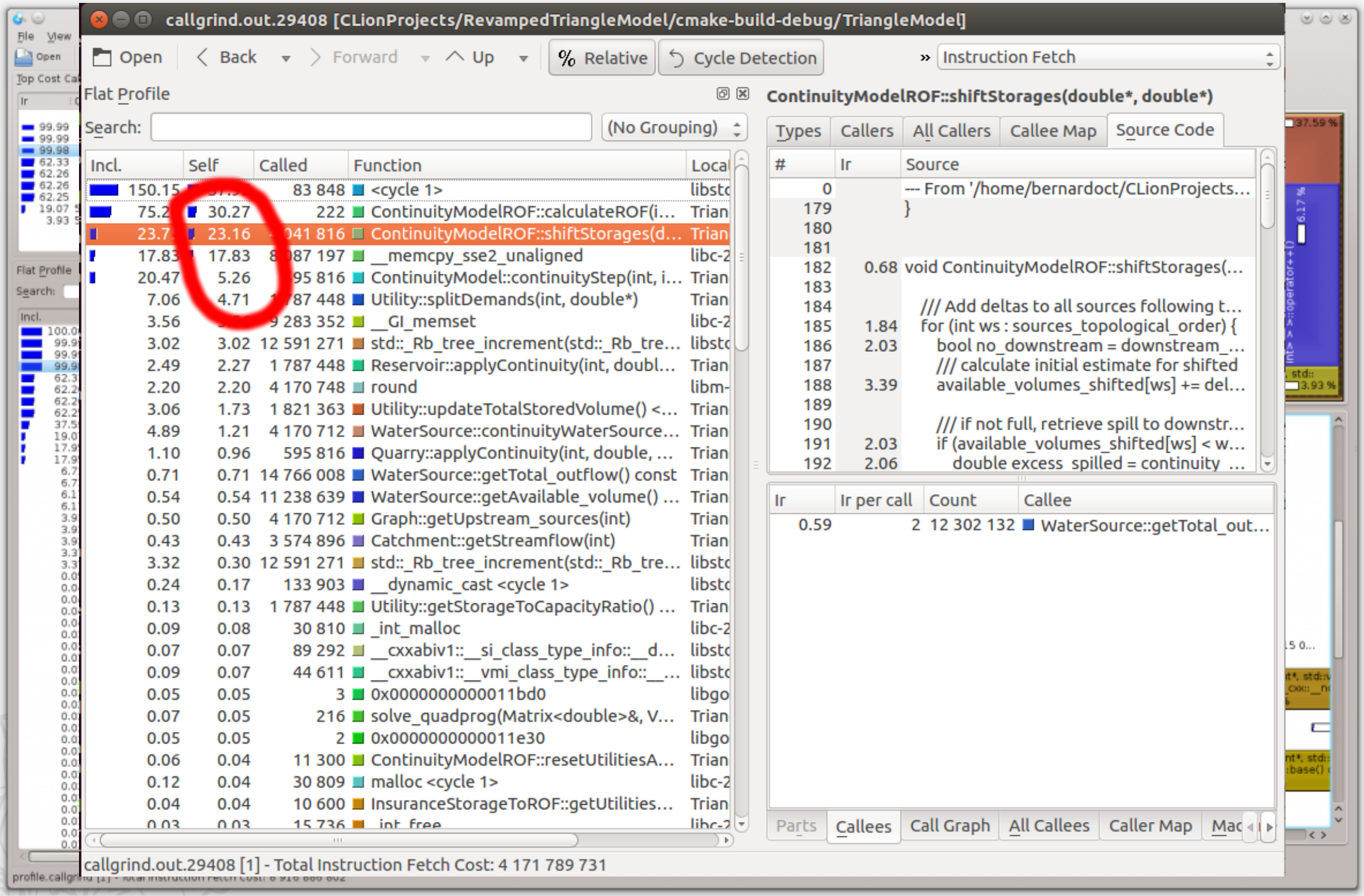
UNIVERSITÀ DEGLI STUDI FIRENZE

Example: KCachegrind



UNIVERSITÀ DEGLI STUDI FIRENZE

Example: KCachegrind



Example: command line

- `callgrind_annotate --auto=yes callgrind.out.<pid>`

- `–auto=yes` provides a line-based analysis, otherwise only functions are analyzed

- The number reported next to each line is how many times the line was executed

- A summary of most frequently called functions is reported at the end

Profiling example

```
. void swap(int *a, int *b)
       3,000  {
       3,000      int tmp = *a;
       4,000      *a = *b;
       3,000      *b = tmp;
       2,000  }
           .
           . int find_min(int arr[], int start, int stop)
       3,000  {
       2,000      int min = start;
   2,005,000      for(int i = start+1; i <= stop; i++)
   4,995,000          if (arr[i] < arr[min])
       6,178              min = i;
       1,000      return min;
       2,000  }
           . void selection_sort(int arr[], int n)
           3  {
       4,005      for (int i = 0; i < n; i++) {
       9,000          int min = find_min(arr, i, n-1);
   7,014,178  => sorts.c:find_min (1000x)
      10,000          swap(&arr[i], &arr[min]);
      15,000  => sorts.c:swap (1000x)
           .    }
           2  }
```

Cache profiling

- Callgrind can profile cache usage (as Cachegrind does).
  - Add `--simulate-cache=yes`
    - It is a simulation of L1/L2 caches, counting hits/ misses
- It will report a summary printed at the end of the execution

Cache profiling example

```
==16409== Events    : Ir Dr Dw I1mr D1mr D1mw I2mr D2mr D2mw
==16409== Collected : 7163066 4062243 537262 591 610 182 16 103 94
==16409==
==16409== I   refs:      7,163,066
==16409== I1  misses:            591
==16409== L2i misses:             16
==16409== I1  miss rate:         0.0%
==16409== L2i miss rate:         0.0%
==16409==
==16409== D   refs:      4,599,505  (4,062,243 rd + 537,262 wr)
==16409== D1  misses:            792  (      610 rd +     182 wr)
==16409== L2d misses:            197  (      103 rd +      94 wr)
==16409== D1  miss rate:         0.0% (      0.0%   +     0.0%  )
==16409== L2d miss rate:         0.0% (      0.0%   +     0.0%  )
==16409==
==16409== L2 refs:             1,383  (    1,201 rd +     182 wr)
==16409== L2 misses:             213  (      119 rd +      94 wr)
==16409== L2 miss rate:          0.0% (      0.0%   +     0.0%  )
```

Cache profiling: output

- The cache simulator models a machine with a split L1 cache (separate instruction I1 and data D1), backed by a unified second-level cache (L2). This matches the general cache design of most modern CPUs.
- Ir: I cache reads (instructions executed)
- I1mr: I1 cache read misses (instruction wasn't in I1 cache but was in L2)
- I2mr: L2 cache instruction read misses (instruction wasn't in I1 or L2 cache, had to be fetched from memory)
- Dr: D cache reads (memory reads)
- D1mr: D1 cache read misses (data location not in D1 cache, but in L2)
- D2mr: L2 cache data read misses (location not in D1 or L2)
- Dw: D cache writes (memory writes)
- D1mw: D1 cache write misses (location not in D1 cache, but in L2)
- D2mw: L2 cache data write misses (location not in D1 or L2)

It is possible to get cache profiling at instruction line level

Tips and tricks

- When debugging we are used to compile in debug mode with no optimizations, but when profiling it is OK to test the optimized program

- These profilers record the count of instructions, not the actual time spent in a function. If you have a program where the bottleneck is file I/O, the costs associated with reading and writing files won't show up in the profile, as those are not CPU-intensive tasks.

- Evaluate using different input sizes.

- You can even profile at assembly level (with Callgrind)

Tips and tricks

- L2 misses are much more expensive than L1 misses, so pay attention to passages with high D2mr or D2mw counts.
  You can use `callgrind_annotate` show/sort options to focus on key events:
  - `callgrind_annotate --show=D2mr --sort=D2mr`

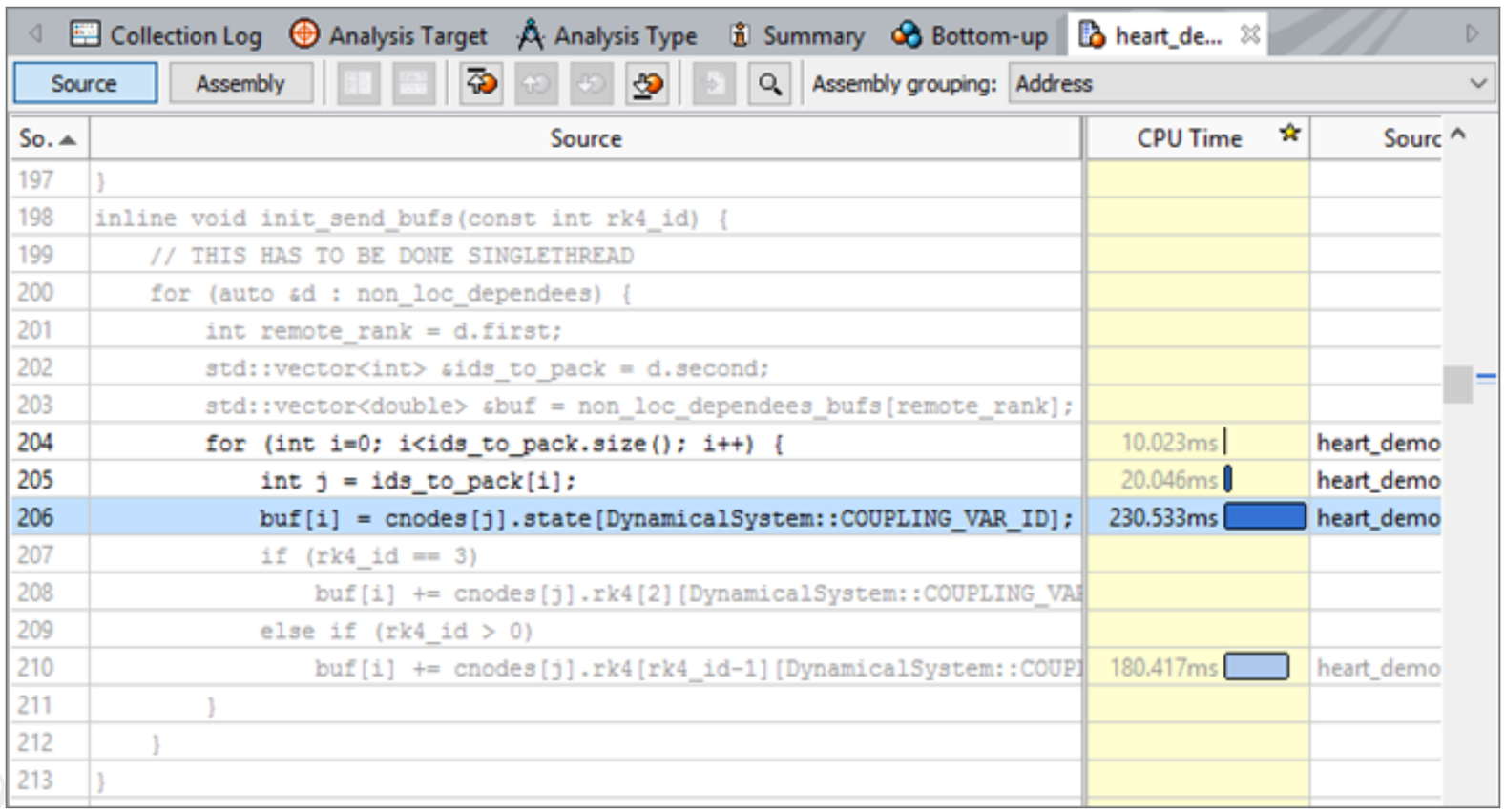
    will highlight D2mr counts.

- These tools (Perftools and Valgrind) have many parameters: check their manuals !

VTune

- Intel VTune is a general-purpose optimization tool that helps to identify bottlenecks and potential improvements.

- It is a proprietary tool but it is freely available for Windows and Linux. Part of oneAPI Toolkit.



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Bottlenecks

Bottlenecks

- When dealing with threads and parallel processing remember these considerations:
  - I/O is a bottleneck.
  - Memory is usually a bottleneck.
  - Managing threads requires an overhead.

I/O Bottleneck

- Accessing I/O, whether it be a hard drive, data from a video camera, or sending through Bluetooth, will cause a thread to wait.

- Unless you program has something to do while waiting for I/O, parallel threads won't help here.

Memory Bottleneck / contention

- There may be contention when one core wants to access memory at the same time another core accesses memory. One thread (core) will have to wait for the other to finish (or interweave the requests). This contention may cause you to loose efficiency.

Overhead of Managing threads

- There is an overhead associated with the creation, maintenance and destruction of threads.
- A thread should perform enough execution to make the thread creation and maintenance worthwhile.
- Use a thread pool to reduce the cost of thread creation: instantiate them at the beginning and then re-use them during the execution of the parallel program.

Links

- Differences between C/C++ clocks: https://stackoverflow.com/questions/12392278/measure-time-in-linux-time-vs-clock-vs-getrusage-vs-clock-gettime-vs-gettimeof/12480485#12480485

- Intel VTune: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

Books

- Parallel and High Performance Computing, R. Robey, Y. Zamora, Manning - Chapt. 3