Tools of the Trade – Part III: Memory Debugger and Profiler

Computational Science II (CAAM 520)

Christopher Thiele

Rice University, Spring 2020

Memory debugging

Recall that improper memory management can cause leaks (in languages like C and C++):

```
void foo(const void *data, size_t size)
{
  void *copy = malloc(size);

  if (!data) {
    return; // Memory leak!
  }

  // ...

  free(copy);
}
```

Memory debugging

In the example, the memory leak is obvious.

→ How to find leaks in a large, complex code base?

We can use a memory debugger like Valgrind:

```
valgrind --tool=memcheck ./myapp
```

Valgrind will point out definite and possible memory leaks.

Note: Running your code in Valgrid will slow down execution ***significantly***!

Valgrind

Besides memory debugging, Valgrind has additional features:

- Identify (some types) of performance bottlenecks.
- Find errors in parallel applications.

These features will not be covered in class, but you may want to try them at some point.

Profiling

Before we attempt to write parallel and performance-conscious code, we should understand how to assess the performance of an application.

- Where in the code does the application spend most time?
- What are the performance hotspots and bottlenecks?
- Where should we start optimizing?

Profiling

Profilers can help us to collect this information.

The most common approaches are:

- ***Instrumentation:*** The compiler automatically adds timers, counters, etc. to each function in our code.
- ***Statistical profiling:*** The code remains unchanged. At specific intervals, the profiler checks which part of the code is currently executed.

Profiling

Instrumentation:

- Pros: Easy to use, "exact" measurements
- Cons: Intrusive, slows down execution, can cause "heisenbugs"

Statistical profiling

- Pros: Not intrusive, does not slow down execution
- Cons: Require support from the OS (drivers)

$\rightarrow$ We will use instrumentation with gprof.

Profiling with instrumentation

Adding timers and counters manually is the most basic form of profiling with instrumentation.

```
void my_function()
{
  const double start = start_timer();

  // ...

  const double delta = stop_timer() - start;
}
```

Profiling with instrumentation

Do not use `time()`!

→ It has a low resolution of one second.

```
void my_function()
{
  const time_t start = time();

  // ...

  const time_t delta = time() - start;
}
```

Profiling with instrumentation

Do not use `clock()`!

- It measures ***CPU time***, not ***elapsed time***.
- `CLOCKS_PER_SEC` is ***not*** the number of clock ticks per second.

```
void my_function()
{
  const clock_t start = clock();

  // ...

  const clock_t end = clock();
  const double delta = (end - start)/CLOCKS_PER_SEC;
}
```

Profiling with instrumentation

On Linux, use `clock_gettime()`, which provides a high-resolution timer that measures real time.

```
void my_function()
{
  struct timespec start, end;
  // Use CLOCK_MONOTONIC, not CLOCK_REALTIME!
  clock_gettime(CLOCK_MONOTONIC, &start);

  // ...

  clock_gettime(CLOCK_MONOTONIC, &end);
  const double delta = time_diff(end, start);
}
```

Profiling with instrumentation

On Linux, use `clock_gettime()`, which provides a high-resolution timer that measures real time.

```
double time_diff(const struct timespec *end,
                 const struct timespec *start)
{
  const double start_double
    = start->tv_sec + 1.0e-9*start->tv_nsec;
  const double end_double
    = end->tv_sec + 1.0e-9*end->tv_nsec;

  return end_double - start_double;
}
```

Using gprof

Step 1: Compile with `-pg` flag.

```
gcc -pg -o myapp myapp.c
```

This tells the compiler to add gprof instrumentation.

Step 2: Run your application as usual. Doing so will produce a file named `gmon.out`.

Using gprof

Step 3: Use gprof to convert `gmon.out` to a human-readable performance profile.

```
gprof myapp gmon.out > profile.txt
```