Tutorial 3: Multicore and parallelism

Before you start, get set up on a Linux machine. The instructions assume you are using a departmental machine, where everything will work except `perf`; for that, you will need to use your own Linux installation.

Get to a Linux system, with a directory that has some GB of free space. Then execute the following commands to prepare for the tutorial:

```
$ wget -qO- https://www.doc.ic.ac.uk/~lvilanov/teaching/comp60017/05-tutorial.tgz
    | tar xvz
$ make -C 05-tutorial parsec-prepare
```

Note: Try to use your own machine for this, so you have all permissions to run `perf`. If you still need to use the department's machines, run the commands above on the `bitbucket` directory, where you have plenty of space:

```
$ mkdir -p /vol/bitbucket/$(whoami)
$ cd /vol/bitbucket/$(whoami)
```

1 Synchronization

We are developing a user-level version of a barrier (exclusively using the `std::atomic` type in C++), with the following interface, which you can find in file `barrier.hpp`:

```
class barrier {
public:
  barrier(size_t expected); // { ... }
  void arrive_and_wait(); // { ... }
private:
  // ...
};
```

1.1 The good

Write the full declaration of the `barrier` class and implement its methods. Discuss the simplest alternatives with your peers.

Test your implementation by modifying `05-tutorial/barrier.hpp` and `05-tutorial/barrier.cpp`, compile with `make -C 05-tutorial program`, and run it with `./05-tutorial/program Nthreads`. For now, we assume a simple scenario where each thread executes in parallel, and the number of threads is lower or equal than the number of cores.

1.2 The bad

Remove the `usleep` line from `05-tutorial/program.cpp`, and re-run with a number of threads higher than the number of cores. You should see that the program does not finish execution. Discuss why.

You can force the number of CPUs to be smaller with `taskset -c 0-Ncpus ./05-tutorial/program Nthreads` (see `man taskset` for additional information).

1.3 The ugly

Devise a new solution that solves the problem above, implement and test it.

2 Optimizing a multi-threaded streaming application

We will be optimizing a single application (ferret) from a larger benchmark suite called PARSEC, which contains a set of parallel benchmarks that use various approaches for parallelism. Ferret is an image similarity search benchmark that is written as a parallel pipeline (i.e., using a streaming approach).

Download and build the parsec benchmarks by running the preparation commands above. They will download, compile and generate the necessary input files to run ferret.

You can use the following command to execute ferret manually, but the instructions below will use the simple Makefile you already downloaded to run all the necessary commands:

```
$ cd parsec-3.0
$ source env.sh
$ parsecmgmt -a run -p ferret -i simlarge -k
```

2.1 Finding a problem

By passing an additional "-n" argument to `parsecmgmt`, we can control the level of parallelism in the application. PARSEC always prepends `time` to all its benchmarks by default, so it will print the time they take to execute (the `time` command is embedded in `bash`, see `man bash` for additional information).

We will now run ferret with various parallelism values, and use `grep` to only print the `user` and `real` values printed by `time`:

```
$ make -C 05-tutorial parsec-run
parsecmgmt -a run -p ferret -i simlarge -k -n 1 | grep -E real\|user
...
parsecmgmt -a run -p ferret -i simlarge -k -n 2 | grep -E real\|user
...
parsecmgmt -a run -p ferret -i simlarge -k -n 3 | grep -E real\|user
...
parsecmgmt -a run -p ferret -i simlarge -k -n 4 | grep -E real\|user
...
parsecmgmt -a run -p ferret -i simlarge -k -n 5 | grep -E real\|user
...
```

Now we will also use `perf` to get additional profiling information from the benchmark:

```
$ make -C 05-tutorial parsec-profile
parsecmgmt -a run -p ferret -i simlarge -k -n 1 -s "perf stat" | tail -28
...
parsecmgmt -a run -p ferret -i simlarge -k -n 2 -s "perf stat" | tail -28
...
parsecmgmt -a run -p ferret -i simlarge -k -n 3 -s "perf stat" | tail -28
...
parsecmgmt -a run -p ferret -i simlarge -k -n 4 -s "perf stat" | tail -28
...
parsecmgmt -a run -p ferret -i simlarge -k -n 5 -s "perf stat" | tail -28
...
```

Note: If you are using the department machines, you may not have the necessary permissions to run `perf`. If that's the case, please raise your hand and I will show the results of these commands.

What do these results tell us? How do we nail down the source of the problem? Discuss and come up with a theory.

2.2 Nailing the reason

Reading about this application, we find out that it is designed as a data streaming pipeline across threads. Let's search for `pthread_create` (it's a C program) and find out where are threads created. Track the relationship between thread creation and the "-n" argument in `main`[1].

How does their relation and the streaming-based architecture explain the inefficiency problem we observed? Discuss and reach a thesis that we can later check.

2.3 Finding the source and fixing it

We will now do a typical profile of the program to find out which functions spend the most execution time. To make things simpler to analyze, we can tell perf to also collect callgraph information with "-g" (visible by pressing enter on lines with a "+" on the TUI):

```
$ make -C 05-tutorial parsec-profile2
cd pkgs/apps/ferret/run
perf record -g ../inst/amd64-linux.gcc/bin/ferret \
  corel lsh queries 10 20 1 output.txt
perf report
```

Discuss how this information can lead us to pinpointing the source of the problem. Discuss a surgical improvement based on those observations, and analyze its impact.

How would our analysis change if the application used purely user-level implementations for synchronization?

You can test your changes by re-building and then re-running ferret:

```
$ make -C 05-tutorial parsec-rebuild
$ make -C 05-tutorial parsec-run
$ make -C 05-tutorial parsec-profile
```

Note: The tutorial's Makefile modifies ferret to compile its files with the flag `-fno-omit-frame-pointer`. This flag is necessary to collect accurate call graphs with `perf`, and also to get meaningful backtraces when debugging.

[1]The sources are in directory `pkgs/apps/ferret`, and you can do rough searches for file names and contents using `find` and `rgrep`, respectively. The thread count argument corresponds to `argv[6]`