Assignment 4a: Acceleration using OpenMP

Martijn Koedam

2016

Organization

In Assignment 3 we tried to optimize the application by hand and later by using the compiler. These optimizations are however limited to a single core. The current trend, now that we seem to have hit a frequency limit, is to add more and more cores/threads to processors. Even mobile phones today come with quad or octa core processor. One obvious way to speedup modern applications is to parallelize over the available cores using threading. There are several ways of doing this. In this assignment we are going to use OpenMP[1] for parallelizing the code over multiple cores.

1 Compiling the program with OpenMP support

We start assignment 4a with the version you got after doing assignment 3. If you do not have this, or it is not working, start with a clean checkout[2]. To compile the application with OpenMP support there are several steps we need to take. First we need to include the `OpenMP` header file:

```
#include <omp.h>
```

Next we are going to modify the `Makefile` to link against and use `OpenMP`.

```
CFLAGS:=-I. -O3 -g3 -Wall -Wextra -std=c99 -D_POSIX_C_SOURCE=200000L -fopenmp
LIBS:=-lpng -lm -lrt -lomp
```

Here `-lomp` tells the linker to link against `libomp.so` and `-fopenmp` tells clang to use OpenMP. Now we can compile and run the application[3].

```
make && ./filter in.png out.png
```

This however results in the following error:

```
./filter: error while loading shared libraries: libomp.so: cannot open shared object
    file: No such file or directory
```

It seems that the `OpenMP` library cannot be found at run-time. This library is part of LLVM; Because we ship the latest LLVM in */home/pcp16/opt/* which is not in the default library paths, so to make it known we need to set the following environment variable.

```
export LD_LIBRARY_PATH=/home/pcp16/opt/lib/
```

[1] `http://openmp.org/wp/`
[2] See assignment 3 how to obtain a clean checkout
[3] && tells the shell to execute filter if make was successful

This tells the runtime linker to look in the right directory. Now we should be able to run the program.

Exercise 1

Annotate your program with the `#pragma omp parallel for` before each for loop to use OpenMP.[a]

[a] Take special care of for loops that update a single variable. See the OpenMP website for hints on how to solve race-conditions.

You will notice this won't give a big speed improvement. One possible reason is that a lot of variables are considered shared, and writing to these variables is protected. To improve this we need to make sure there are not to many unneeded shared variables. For example; variables like loop iterators should be private in the loop.

A good way to fix this is to manually set the shared variables.

Exercise 2

Lets see if we can improve performance by reducing the amount of shared variables. Start by setting the default to not allow any shared variable:

```
#pragma omp parallel for default(none)
```

If you compile the code, the compiler will warn you about the variables used inside the for loop. Then add all the variables you think need to be shared

```
#pragma omp parallel for default(none) shared(Kernel, sum)
```

Variables like loop iterators or hold intermediate values should be private to the loop. Constants that are only read, should be marked as constants. When the number of shared variables is minimized it should greatly improve the obtained speedup.

We can improve the performance by doing manual tiling. With tiling we devide the input data up into large blocks an have each thread process one block.

Exercise 3

By doing a manual tiling we can improve performance further, below is the start of tiling a loop:

```
#pragma omp parallel default(none) shared(output, input, ysize,hsize)
{
    int step = ( ysize ) / omp_get_num_threads ();
    int cpu  = omp_get_thread_num ();
    int stop = min ( ( cpu + 1 ) * step, wh );
    int start = cpu * step;
```

You can then use the calculated start/stop as the loop bounds. Why does this help?

Note

To check that multiple threads are created, you can 'trace' the program. The following command will list the amount of threads created:

```
1 strace -f -c ./filter in.png out.png 2>&1 | grep -c "Process .* attached"
```

The `strace` tool traces system calls and signals. Strace will print to stderr a *Process pid attached* for each thread or fork. The `grep` command counts these.

When done right the `CONTRAST` and `END FILTER` total execution time can be reduced to 1 second.

In total we should have obtained a total reduction of execution time of a factor 8 in assignment 3 and 4a.

Exercise 4

Can you think of more ways (think of the method explained during the lectures) how you can improve the speed of the parallelization using OpenMP. The most gain is obtained by removing/reducing access to shared variables.