Systems Software HS15

Lab Exercises

Claudio Mura
claudio@ifi.uzh.ch

Practical notes on parallel programming using OpenMP

OpenMP

- High-level API for parallelizing C/C++ and Fortran code
  - based on `#pragma` compiler directives
  - includes a run-time library
- How does OpenMP work?
  - place directives before blocks to be parallelized
  - compiler processes these directives and generates multi-thread code
- Easy to parallelize existing code!

OpenMP Directives

- An OpenMP directive has the following structure:

```
#pragma omp directive [ clause list ]
```

- Example:

```
int a[ SIZE ];
int i;
#pragma omp parallel for shared( a ) private( i )
for( i=0; i<SIZE; i++ )
{
  a[ i ] = pow( a[ i ], 2.0f );
}
```

`parallel` Directive

- The most important directive is `parallel` :

```
#pragma omp parallel [ clause list ]
{
  // structured block
}
```

  - when the main thread encounters this directives, it becomes the *master* and generates a group of threads
- You will mostly need it to parallelize `for` loops
  - syntax shortcut: `#pragma omp parallel for`

`parallel` Directive: Clauses

- The `parallel` directive supports a number of clauses:
  - conditional parallelization:

    `#pragma omp parallel for if( scalar expression )`
  - degree of concurrency

    `#pragma omp parallel for num_threads( integer )`
  - data sharing attribute clauses
    - `#pragma omp parallel for private( variable list )`

      variables are local to each thread (each thread has a copy)
    - `#pragma omp parallel for shared( variable list )`

      variables are shared among all the threads (only one copy!)
    - `#pragma omp parallel for firstprivate( variable list )`

      each thread has a copy + all the copies are initialized
    - many other

Scheduling Policies for `omp parallel for`

- The clause **`schedule`** controls how to split iterations and assign them to threads
  - **`#pragma omp parallel for schedule( static, chunk_size )`**

    split iteration space in fixed blocks of `chunk_size`, assign them to threads in round-robin fashion
  - **`#pragma omp parallel for schedule( dynamic, chunk_size )`**

    split iteration space in fixed blocks of `chunk_size`, assign a block to a thread as it becomes idle
  - **`#pragma omp parallel for schedule( guided, chunk_size )`**

    same as `dynamic`, but exponentially reduces the size of the chunk to reduce idling

OpenMP Runtime Library

- A set of routines that deal with:
  - getting info about environment (e.g. `omp_get_num_procs()`)
  - controlling thread creation
  - controlling mutual exclusion
  - timing portions of code
    - `omp_get_wtime()` : elapsed wall-clock time (in seconds) since arbitrary reference time

Compilation

- You must add special flags to your compilation commands
  - `-fopenmp` for the compilation
  - `-lgomp` for the linker

Note for Mac OS X users

- The version of clang provided does not support OpenMP!
  - http://openmp.org/wp/openmp-compilers/
- Solutions:
  - get gcc on Mac OS :-)
    - use homebrew
    - https://solarianprogrammer.com/2013/06/11/compiling-gcc-mac-os-x/
    - http://hpc.sourceforge.net/
  - use Linux (maybe virtualized, using VirtualBox)
  - ... (but make sure it works!)
- Contact me if you need assistance

Exercise 4

Parallel Median Filtering of Image Sets using OpenMP

Median Filtering on Image Sets

- We are already familiar with median filtering of images…

.02	.04	.05	.03	.01
.02	.03	.03	.04	.03
.01	.02	.99	.03	.00
.00	.03	.03	.00	.03
.00	.01	.01	.04	.03

→

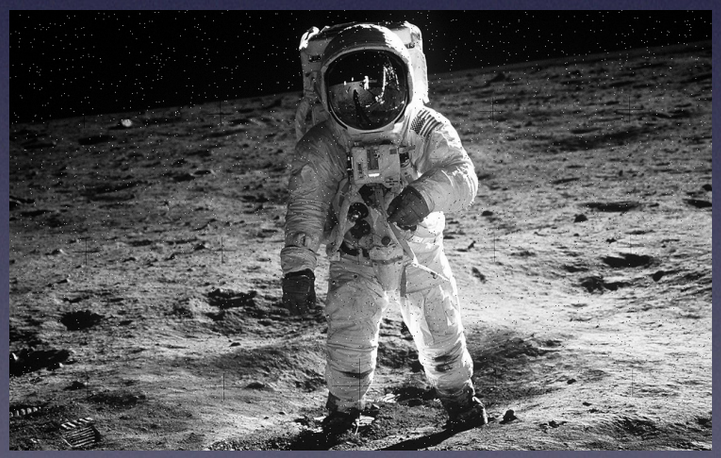
.02	.04	.05	.03	.01
.02	.03	.03	.04	.03
.01	.02	.03	.03	.00
.00	.03	.03	.00	.03
.00	.01	.01	.04	.03

.00	.02	.03	.03	.03	.03	.03	.04	.99

↑

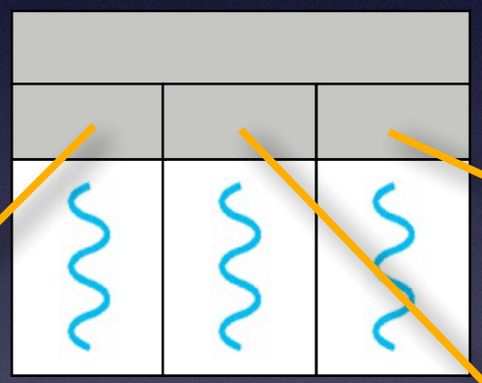
Parallel Median Filtering of Images

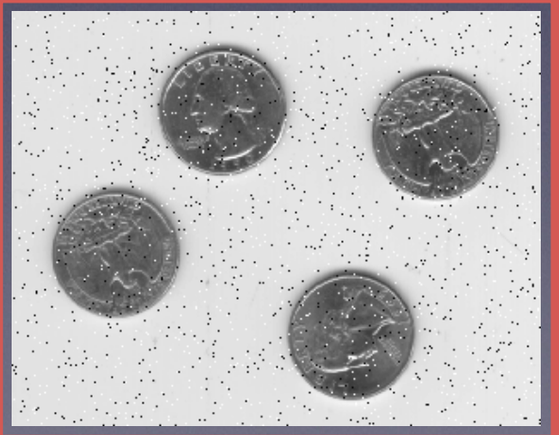
- In this exercise we'll consider *multiple input image*
  - each to be filtered with a filter of the same size
- Focus of the exercise is on parallelisation of the computation
  - we'll consider two strategies




Strategy 1: Image-level Parallelism

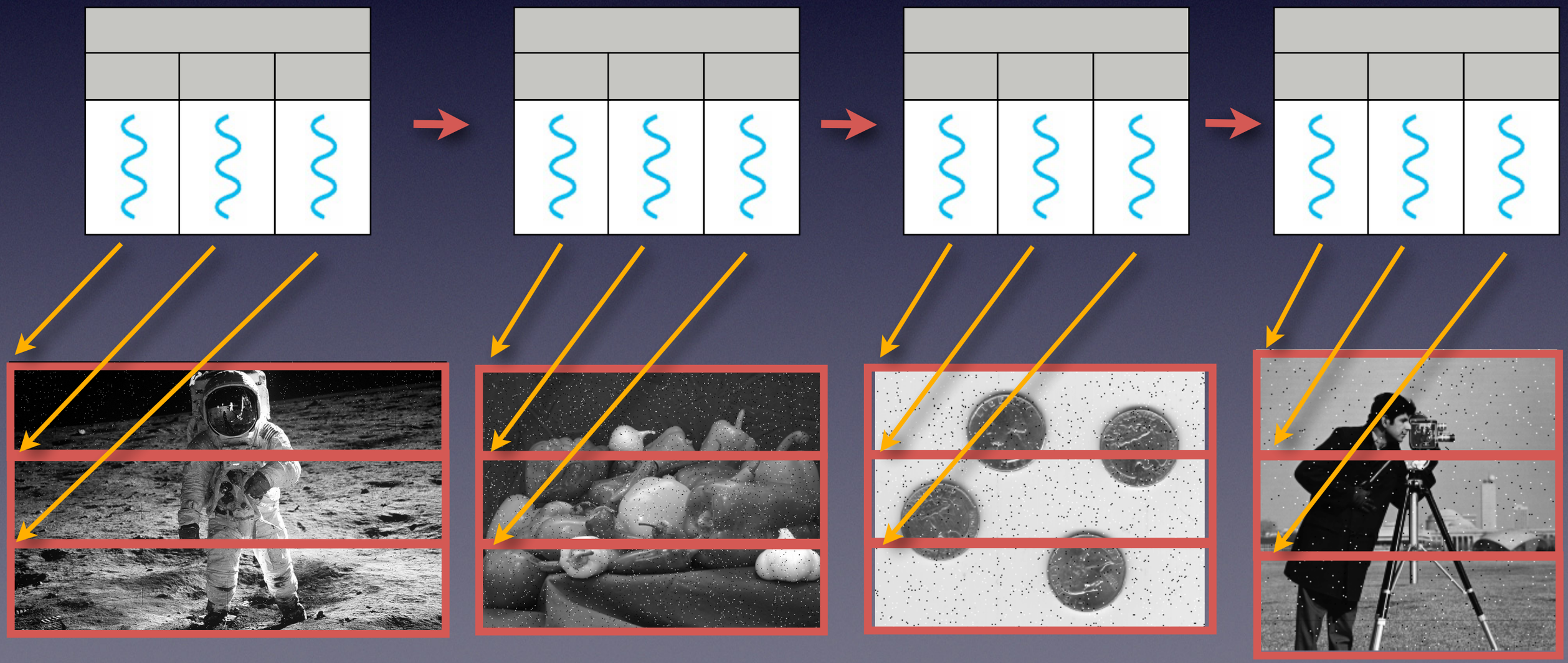
- Create multiple threads, each processes one or more images
  - each image is processed *serially*, one pixel after another
- Set of input images split among available threads





Strategy 2: Pixel-level Parallelism

- Consider each image one after another
- For each image, filter the pixels in parallel, creating multiple threads and assigning each a sub-set of pixels



Comparing the approaches

- How fast is the parallel version w.r.t. to a serial execution? Which parallelisation strategy works better?
- Implement a *benchmark* mode
  - on the same set of images, run sequentially the serial version of the computation, the one parallelised at image-level and the one parallelised at pixel level
  - keep track of the computation time for each mode

Technical Notes

- The parallelisation should be achieved using OpenMP
  - mainly `#pragma omp parallel for` directives
  - use `omp_get_wtime()` to track the processing time
- The number of threads to be created is passed as command-line argument
  - 2 main choices with OpenMP: clause after `parallel for` directive or function in runtime library

Additional Details

- Input images stored in text files
  - same format as for Ex. 3
  - filenames are provided as command-line argument
- Window size of filter provided as command-line argument
- Your application should be able to run in one of 4 modes:
  - serial (no parallelism at all)
  - parallel, using strategy 1 (image-level parallelism)
  - parallel, using strategy 2 (pixel-level parallelism)
  - benchmark
  - a command-line argument to select which mode to run
- Number of threads: also command-line argument

Additional Details - 2

- Unless running benchmark mode, write filtered images to files
  - filename for each filtered image: OUT_ + name of text file containing original image
- In benchmark mode print a summary of the computation time spent in each mode when *all* processing is done (i.e., when the 3 modes have completed)