Introduction

- Modern computers have **high-performance parallel processors**
  - multicore CPU
  - GPU
- How to use them *efficiently* in practice?
Introduction

• Not just for high-end servers but also for everyday programming tasks
  • laptops, desktops, mobile devices…

• Sometimes you can easily improve running times from minutes to seconds
Key challenges

- Know the tools
- Understand memory hierarchy
  - main memory, caches
- Exploit parallel processing units
  - multicore CPU
An example
Image processing

• 2D array, 16000 x 16000 values, 32-bit ints
  • approx. 1 GB of data

• Median filter:
  • new value = median of pixel and its 4 neighbours
static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
Baseline

```c
static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
```

running time: 68 s
Sanity checking

- Classroom computers:
  3.3 GHz CPU, 4 cores, hyperthreading

- We are using > 800 clock cycles per pixel
  - median of 5 elements, should not be that hard?

- We are only using 1 thread on 1 core
Know the tools

```c
static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                              x[i][ja], x[i][jb]);
        }
    }
}
```

g++-4.8 -march=native -O3

running time: **25 s**
static void median(const array_t x, array_t y) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
            x[i][ja], x[i][jb]);
        }
    }
}
Understand memory hierarchy

```c
static void median(const array_t x, array_t y) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                              x[i][ja], x[i][jb]);
        }
    }
}
```

```
x[0][0]  x[0][1]  x[0][2]  x[0][3]  x[0][4]  x[0][5]  ...  x[1][0]  x[1][1]  ...
x[0][1]  x[0][2]  x[0][3]  x[0][4]  x[0][5]  ...  x[1][0]  x[1][1]  ...
x[0][2]  x[0][3]  x[0][4]  x[0][5]  ...  x[1][0]  x[1][1]  ...
x[0][3]  x[0][4]  x[0][5]  ...  x[1][0]  x[1][1]  ...
x[0][4]  x[0][5]  ...  x[1][0]  x[1][1]  ...
x[0][5]  ...  x[1][0]  x[1][1]  ...
...  ...  ...  ...  ...  ...
```

static void median(const array_t x, array_t y) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
Exploit parallel processing units

```c
static void median(const array_t x, array_t y) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                             x[i][ja], x[i][jb]);
        }
    }
}
```

g++-4.8  -fopenmp

running time: 2 s
## Running times

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Compiler</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>68 s</td>
<td>25 s</td>
<td>9 s</td>
</tr>
<tr>
<td>Parallel</td>
<td>12 s</td>
<td>5 s</td>
<td>2 s</td>
</tr>
</tbody>
</table>
It can be this easy!

- Significant improvements in running times
  - from over a minute to a few seconds
- No algorithmic changes needed this time
- Memory layout: change array indexing
- Parallelisation: just add one #pragma
Are we done now?

- **We are using** \( \approx 20 \text{ clock cycles per pixel} \)
  - sounds reasonable, but…

- **Reading & writing memory** \( \approx 1 \text{ GB/second} \)
  - not a bottleneck yet, "memcpy" achieves > 18 GB/second
  - can we make the "median" function faster?
Better algorithms?

```c++
static int median(int v1, int v2, int v3, int v4, int v5) {
    int a[] = {v1, v2, v3, v4, v5};
    std::nth_element(a+0, a+2, a+5);
    return a[2];
}
```
Better algorithms?

```java
static int median(int v1, int v2, int v3, int v4, int v5) {
    int a[] = {v1, v2, v3, v4, v5};
    for (int i = 0; i < 4; ++i) {
        int b = 0;
        for (int j = 0; j < 5; ++j) {
            b += (a[j] < a[i] || (a[i] == a[j] && i < j));
        }
        if (b == 2) {
            return a[i];
        }
    }
    return a[4];
}
```
Better algorithms?

static int median(int v1, int v2, int v3, int v4, int v5) {
    int a[] = {v1, v2, v3, v4, v5};
    for (int i = 0; i < 4; ++i) {
        int b = 0;
        for (int j = 0; j < 5; ++j) {
            b += (a[j] < a[i] || (a[i] == a[j] && i < j));
        }
        if (b == 2) {
            return a[i];
        }
    }
    return a[4];
}

Wait, what, $O(n^2)$ time??
Better algorithms?

- Implement a better “median” function:
  - $\approx 0.6 \text{ s in total}$
  - $\approx 7 \text{ clock cycles per pixel}$
  - $\approx 4 \text{ GB/s}$

- Are we happy now?
Know when to stop!

- Implement a better “median” function:  
  \( \approx 0.6 \text{ s in total} \)

- Just copying data with “memcpy”:  
  \( \approx 0.1 \text{ s} \) (even after warm-up)
## Running times

<table>
<thead>
<tr>
<th></th>
<th>Baseline</th>
<th>Compiler</th>
<th>Memory</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>68 s</td>
<td>25 s</td>
<td>9 s</td>
<td>3 s</td>
</tr>
<tr>
<td>Parallel</td>
<td>12 s</td>
<td>5 s</td>
<td>2 s</td>
<td>0.6 s</td>
</tr>
</tbody>
</table>
What about GPUs?

cudaHostGetDevicePointer((void**)&outputGPU, outputCPU, 0);
cudaMalloc((void**)&inputGPU, size * sizeof(int));
cudaMemcpy(inputGPU, inputCPU, size * sizeof(int), cudaMemcpyHostToDevice);

dim3 dimBlock(64, 1);
dim3 dimGrid((width + dimBlock.x - 1) / dimBlock.x, (height + dimBlock.y - 1) / dimBlock.y);
medianKernel<<<dimGrid, dimBlock>>>(
    outputGPU, inputGPU, width, height, size
);

cudaFree(inputGPU);

(some boring details omitted...)
__global__ void medianKernel(int* output, const int* input,
              const int width, const int height, const int size)
{
    int x = threadIdx.x + blockIdx.x * blockDim.x;
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    if (x >= width || y >= height) return;

    int p0 = x + width * y;
    int p1 = p0 - 1; int p2 = p0 + 1;
    int p3 = p0 - width; int p4 = p0 + width;

    if (x == 0) p1 += width;
    if (x == width - 1) p2 -= width;
    if (y == 0) p3 += size;
    if (y == height - 1) p4 -= size;

    int a0 = input[p0]; int a1 = input[p1]; int a2 = input[p2];
    int a3 = input[p3]; int a4 = input[p4];

    int b0 = min(a0, a1); int b1 = max(a0, a1); int b2 = min(a2, a3);
    int b3 = max(a2, a3); int c0 = min(b0, b2); int c2 = max(b0, b2);
    int c1 = min(b1, b3); int d1 = min(c1, c2); int d2 = max(c1, c2);
    int e4 = max(c0, a4); int f2 = min(d2, e4); int g2 = max(d1, f2);

    output[p0] = g2;
}

running time: 0.3 s
Know when to stop!

• Median filtering with GPU:
  \( \approx 0.3 \text{ s} \) in total

• Just moving data to GPU and back:
  \( \approx 0.3 \text{ s} \)
About this course
Course overview

• Practical hands-on course
• Non-trivial algorithmic problems
• Everything happens on a single machine
  • no networking, no distributed computing
• Only wall-clock time matters
Only wall-clock time matters

- How many seconds does it take for this machine to solve this problem?
- Parallelism not a goal in itself, just one way to get more performance
- Benchmark everything! Do not assume, try it out and see yourself!
Assignments

• Programming tasks, 1 exercise / week

• Always two components:
  • implementation
  • report on experiments

• See the course web page for details
Workload

• 5 credits in 6 weeks ≈ 22 working hours per week
  • more than a half-time job!

• Lecture + exercises only 6 hours per week
  • you are expected to spend lot of time programming on your own
Tools

- C or C++ with OpenMP
- C or C++ with CUDA
- Linux
- Classroom: Maari-A
Cool things

- GPU programming
  - Nvidia GPUs, CUDA

- Access to high-performance cloud servers at CSC
  - 16-core CPUs
Why parallelism?
Why parallelism?

<table>
<thead>
<tr>
<th>Year</th>
<th>Transistors</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>3,000</td>
<td>6502</td>
</tr>
<tr>
<td>1979</td>
<td>30,000</td>
<td>8088</td>
</tr>
<tr>
<td>1985</td>
<td>300,000</td>
<td>386</td>
</tr>
<tr>
<td>1989</td>
<td>1,000,000</td>
<td>486</td>
</tr>
<tr>
<td>1995</td>
<td>6,000,000</td>
<td>Pentium Pro</td>
</tr>
<tr>
<td>2000</td>
<td>40,000,000</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>Year</td>
<td>Transistors</td>
<td>Clock Speed</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1975</td>
<td>3,000</td>
<td>1,000,000</td>
</tr>
<tr>
<td>1979</td>
<td>30,000</td>
<td>5,000,000</td>
</tr>
<tr>
<td>1985</td>
<td>300,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>1989</td>
<td>1,000,000</td>
<td>20,000,000</td>
</tr>
<tr>
<td>1995</td>
<td>6,000,000</td>
<td>200,000,000</td>
</tr>
<tr>
<td>2000</td>
<td>40,000,000</td>
<td>2,000,000,000</td>
</tr>
<tr>
<td>Year</td>
<td>Clock Cycles</td>
<td>Processor</td>
</tr>
<tr>
<td>------</td>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>1980</td>
<td>100</td>
<td>8087</td>
</tr>
<tr>
<td>1987</td>
<td>50</td>
<td>387</td>
</tr>
<tr>
<td>1993</td>
<td>3</td>
<td>Pentium</td>
</tr>
</tbody>
</table>

How many clock cycles does it take to do a floating-point multiplication (FMUL)?
Progress!

- **Increasing:**
  
  *clock cycles / second*

- **Decreasing:**

  *clock cycles / operation*

- **Increasing rapidly:**

  *operations / second*
<table>
<thead>
<tr>
<th>Year</th>
<th>Transistors</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1975</td>
<td>3 000</td>
<td>6502</td>
</tr>
<tr>
<td>1979</td>
<td>30 000</td>
<td>8088</td>
</tr>
<tr>
<td>1985</td>
<td>300 000</td>
<td>386</td>
</tr>
<tr>
<td>1989</td>
<td>1 000 000</td>
<td>486</td>
</tr>
<tr>
<td>1995</td>
<td>6 000 000</td>
<td>Pentium Pro</td>
</tr>
<tr>
<td>2000</td>
<td>40 000 000</td>
<td>Pentium 4</td>
</tr>
<tr>
<td>2005</td>
<td>100 000 000</td>
<td>2-core Pentium D</td>
</tr>
<tr>
<td>2008</td>
<td>700 000 000</td>
<td>8-core Nahelem</td>
</tr>
<tr>
<td>2014</td>
<td>6000 000 000</td>
<td>18-core Haswell</td>
</tr>
<tr>
<td>Year</td>
<td>Transistors</td>
<td>Clock speed</td>
</tr>
<tr>
<td>------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>1975</td>
<td>3 000</td>
<td>1 000 000</td>
</tr>
<tr>
<td>1979</td>
<td>30 000</td>
<td>5 000 000</td>
</tr>
<tr>
<td>1985</td>
<td>300 000</td>
<td>20 000 000</td>
</tr>
<tr>
<td>1989</td>
<td>1 000 000</td>
<td>20 000 000</td>
</tr>
<tr>
<td>1995</td>
<td>6 000 000</td>
<td>200 000 000</td>
</tr>
<tr>
<td>2000</td>
<td>40 000 000</td>
<td>2000 000 000</td>
</tr>
<tr>
<td>2005</td>
<td>100 000 000</td>
<td>3000 000 000</td>
</tr>
<tr>
<td>2008</td>
<td>700 000 000</td>
<td>3000 000 000</td>
</tr>
<tr>
<td>2014</td>
<td>6000 000 000</td>
<td>2000 000 000</td>
</tr>
<tr>
<td>Year</td>
<td>Clock Cycles</td>
<td>Processor</td>
</tr>
<tr>
<td>------</td>
<td>--------------</td>
<td>-----------</td>
</tr>
<tr>
<td>1980</td>
<td>100</td>
<td>8087</td>
</tr>
<tr>
<td>1987</td>
<td>50</td>
<td>387</td>
</tr>
<tr>
<td>1993</td>
<td>3</td>
<td>Pentium</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2013</td>
<td>5</td>
<td>Haswell</td>
</tr>
</tbody>
</table>

How many clock cycles does it take to do a floating-point multiplication (FMUL)?
Progress???

- Not increasing: clock cycles / second
- Not decreasing: clock cycles / operation
- Not increasing: operations / second
<table>
<thead>
<tr>
<th>Dependent operations</th>
<th>Independent operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1 *= a0;</td>
<td>b1 *= a1;</td>
</tr>
<tr>
<td>a2 *= a1;</td>
<td>b2 *= a2;</td>
</tr>
<tr>
<td>a3 *= a2;</td>
<td>b3 *= a3;</td>
</tr>
<tr>
<td>a4 *= a3;</td>
<td>b4 *= a4;</td>
</tr>
</tbody>
</table>

Inherently sequential

Bottleneck: *latency*

Opportunities for parallelism

Bottleneck: *throughput*
New kind of progress

• Difficult: designing CPUs with faster multiplication units
  • latency not improving

• “Easy”: designing CPUs with a large number of parallel multiplication units
  • throughput improving
New kind of progress

• 1993 (Pentium):
  • 0.3 dependent multiplications / cycle
  • 0.5 independent multiplications / cycle

• 2014 (12-core Haswell):
  • 0.2 dependent multiplications / cycle
  • 100 independent multiplications / cycle
New kind of progress

- Not increasing:
  dependent operations / second

- Increasing rapidly:
  independent operations / second

- *All new performance comes from parallelism*
What kind of parallelism is there?

... and how to exploit it?
Bit-level parallelism

```c
bool a[64];
bool b[64];
for (int i = 0; i < 64; ++i) {
    a[i] = a[i] || b[i];
}

uint64_t a;
uint64_t b;
a |= b;
```
Instruction-level parallelism

Pipelining:
can start to process B before finished with A
(if independent)
Instruction-level parallelism

Superscalar execution:
multiple parallel units,
process A and B simultaneously
(if independent)

code:
Instruction-level parallelism

Out-of-order execution: run whatever you can

B depends on A,
A and C can be pipelined,
A and D use different units

code:  A  B  C  D
Instruction-level parallelism

How to exploit: make sure as much is independent as possible

Then (and only then) the CPU will figure out an efficient way to run your code
Bad

a1 *= a0;
a2 *= a1;
a3 *= a2;
a4 *= a3;
a5 *= a4;
a6 *= a5;
a7 *= a6;
a8 *= a7;

Good

b1 *= a1;
b2 *= a2;
b3 *= a3;
b4 *= a4;
b5 *= a5;
b6 *= a6;
b7 *= a7;
b8 *= a8;
Bad

\[ a_1 = v[a_0] ; \]
\[ a_2 = v[a_1] ; \]
\[ a_3 = v[a_2] ; \]
\[ a_4 = v[a_3] ; \]
\[ a_5 = v[a_4] ; \]
\[ a_6 = v[a_5] ; \]
\[ a_7 = v[a_6] ; \]
\[ a_8 = v[a_7] ; \]

Good

\[ b_1 = v[a_1] ; \]
\[ b_2 = v[a_2] ; \]
\[ b_3 = v[a_3] ; \]
\[ b_4 = v[a_4] ; \]
\[ b_5 = v[a_5] ; \]
\[ b_6 = v[a_6] ; \]
\[ b_7 = v[a_7] ; \]
\[ b_8 = v[a_8] ; \]
Vector instructions

• 256-bit wide “AVX” registers
  • YMM0, YMM1, …, YMM15

• Can be interpreted e.g. as:
  • a vector with $8 \times 32$-bit floats
  • a vector with $4 \times 64$-bit doubles
Vector instructions

- **SIMD** = single instruction, multiple data
- Same operation for each vector element
  - \( a[i] = b[i] + c[i] \) for each \( i = 0, 1, \ldots, 7 \)
  - \( a[i] = b[i] / c[i] \) for each \( i = 0, 1, \ldots, 7 \)
- Special functional units, special instructions
Vector instructions

```c
float a[8]; float b[8]; float c[8];
for (int i = 0; i < 8; ++i) {
    a[i] = b[i] * c[i];
}

typedef float float8_t __attribute__((__vector_size__(8*sizeof(float))));
float8_t a; float8_t b; float8_t c;
a = b * c;
```
Vector instructions

```c
float a[8]; float b[8]; float c[8];
for (int i = 0; i < 8; ++i) {
    a[i] = b[i] * c[i];
}

typedef float float8_t __attribute__((__vector_size__ (8*sizeof(float))));
float8_t a; float8_t b; float8_t c;
a = b * c;
vmulps %ymm0, %ymm1, %ymm2
```
Vector instructions

```c
float a[8]; float b[8]; float c[8];
for (int i = 0; i < 8; ++i) {
    a[i] = b[i] * c[i];
}
typedef float float8_t __attribute__((__vector_size__(8*sizeof(float))));
float8_t a; float8_t b; float8_t c;
a = b * c;
```

Some care needed with **memory alignment**! More about this later…
Multiple threads

• Multiple processors
  • independent processors
  • shared main memory
Multiple threads

- Multiple processors
- Multiple cores per processor
  - multiple processors in single package
  - often some shared components, e.g. caches
Multiple threads

• Multiple processors

• Multiple cores per processor

• Multiple threads per core
  • “Hyper-threading”
  • better utilisation of CPU resources
Multiple threads

- **My phone:** 1 processor × 4 cores × 1 thread
- **My laptop:** 1 processor × 2 cores × 2 threads
- **Our classroom:** 1 processor × 4 cores × 2 threads
- **CSC servers:** 2 processors × 12 cores × 1 thread
Multiple threads

• How to exploit:
  • run multiple processes simultaneously (e.g.: same program, different parameters)
  • OpenMP: #pragma omp
  • pthread_create(), fork(), etc.
GPU

• **GPGPU**: general-purpose computing on graphics processing units
  • massively parallel hardware

• **How to exploit**: CUDA, OpenCL
  • can also run code on both CPU and GPU simultaneously in parallel
Bit-level parallelism

Instruction-level parallelism

SIMD: vector instructions

Multiple threads

GPU

GPU + CPU in parallel

long words

automatic

vector types

OpenMP

CUDA

CUDA
Bit-level parallelism

Instruction-level parallelism

SIMD: vector instructions

Multiple threads

GPU

GPU + CPU in parallel

long words

automatic

vector types

OpenMP

CUDA

CUDA
Parallel processing with OpenMP

#pragma omp
OpenMP

- Extension of C, C++, Fortran
- Standardised, widely supported
- Just compile and link your code with:
  - gcc -fopenmp
  - g++ -fopenmp
OpenMP

• You add `#pragma omp` directives in your code to tell what to parallelise and how

• Compiler & operating system takes care of everything else

• You can often write your code so that it works fine even if you ignore all `#pragmas`
OpenMP

- Shared memory multiprocessing
- Multiple simultaneous *threads* of execution
- All threads have access to the same *shared memory*
a();
#pragma omp parallel
{
    b();
}
c();
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
}
d();
a();
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
e();
a();
#pragma omp parallel
{
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
e();
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
}

= 

#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
a();
#pragma omp parallel
{
    b();
    #pragma omp critical
    {
        c();
    }
}
d();
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
e();
global_initialisation();
#pragma omp parallel
{
    local_initialisation();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        do_some_work(i);
    }
    #pragma omp critical
    {
        update_global_data();
    }
}
report_result();
// shared variable
int sum_shared = 0;
#pragma omp parallel
{
    // private variables (one for each thread)
    int sum_local = 0;
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        sum_local += i;
    }
    #pragma omp critical
    {
        sum_shared += sum_local;
    }
}
print(sum_shared);
OpenMP memory model

• Contract between programmer & system

• Local "temporary view", global "memory"
  • threads read & write temporary view
  • may or may not be consistent with memory

• Consistency guaranteed only after a "flush"
OpenMP memory model

- Implicit “flush” e.g.:
  - when entering/leaving “parallel” regions
  - when entering/leaving “critical” regions

- Mutual exclusion:
  - for “critical” regions
```c
int a = 0;
#pragma omp parallel
{
    #pragma omp critical
    {
        a += 1;
    }
}
```

Simple rules

• Permitted (without explicit synchronisation):
  • multiple threads reading, no thread writing
  • one thread writing, same thread reading

• Forbidden (without explicit synchronisation):
  • multiple threads writing
  • one thread writing, another thread reading
Simple rules

- Safe:
  - thread 1: \( p[0] = q[0] + q[1] \)
Simple rules

• Safe:
  • thread 1: \( p[0] = p[0] + q[1] \)
Simple rules

- Not permitted without synchronisation:
  - thread 1: \( p[0] = q[0] + p[1] \)

- “Data race”, unspecified behaviour
Simple rules

Not permitted without synchronisation:

- thread 1: \( p[0] = q[0] + q[1] \)

“Data race”, unspecified behaviour
Simple rules

- Not permitted without synchronisation:
  - thread 1: \( p[0] = 1 \)
  - thread 2: \( p[0] = 1 \)
  - thread 3: \( p[0] = 1 \)

- “Data race”, unspecified behaviour
Filtering is very easy

```c
void filter(const int* data, int* result) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        result[i] = compute(data[i]);
    }
}
```
Filtering is very easy

```c
static void median(const array_t x, array_t y) {
    #pragma omp parallel for
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int ia = (i + n - 1) % n; int ib = (i + 1) % n;
            int ja = (j + n - 1) % n; int jb = (j + 1) % n;
            y[i][j] = median(x[i][j], x[ia][j], x[ib][j],
                              x[i][ja], x[i][jb]);
        }
    }
}
```
OpenMP: summary

• These are sufficient for now:
  • #pragma omp parallel
  • #pragma omp for
  • #pragma omp critical

• Your turn: use these in this week’s exercises!