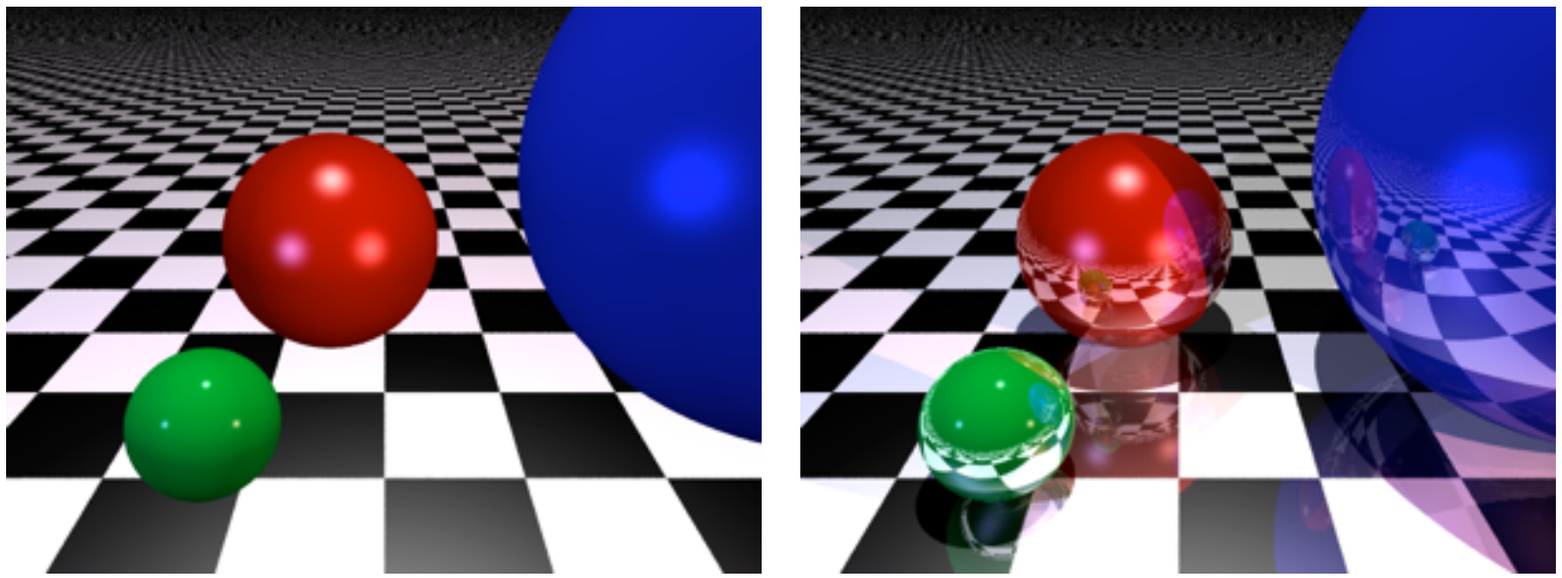
Introduction to Computer Graphics

Implementing a Fast Ray Tracer



Prof. Dr. Mario Botsch

Computer Graphics & Geometry Processing

Universität Bielefeld

CITEC

Outline

- C++ Crash Course
- RayTracer Design
- **Performance optimization**
- Multicore parallelization

Compiler Setting

- `-O`
  - enables compiler optimization
- `-O3`
  - more optimization
  - in particular “function inlining” (later)
- `-march=native -mtune=native`
  - optimize for special architecture
- `funroll-loops`
  - tries to unroll small loops

Compiler Settings

- `-DNDEBUG`
  - defines the preprocessor variable `NDEBUG`
  - removes assertions from the code
  - Important, expensive if-statements otherwise

```
double vec3::operator[](unsigned int _i) const
{
  assert(_i < 3);
  return data_[_i];
}
```

Compiler Settings

- `-DNDEBUG`
  - defines the preprocessor variable `NDEBUG`
  - removes assertions from the code
  - Important, expensive if-statements otherwise

```
double vec3::operator[](unsigned int _i) const
{
#ifndef NDEBUG
  if (!(_i < 3)) print_error_and_abort();
#endif
  return data_[_i];
}
```

Write correct code

- Use constant return values

```
const vec3 add(vec3 _v1, vec3 _v2) { … }
```

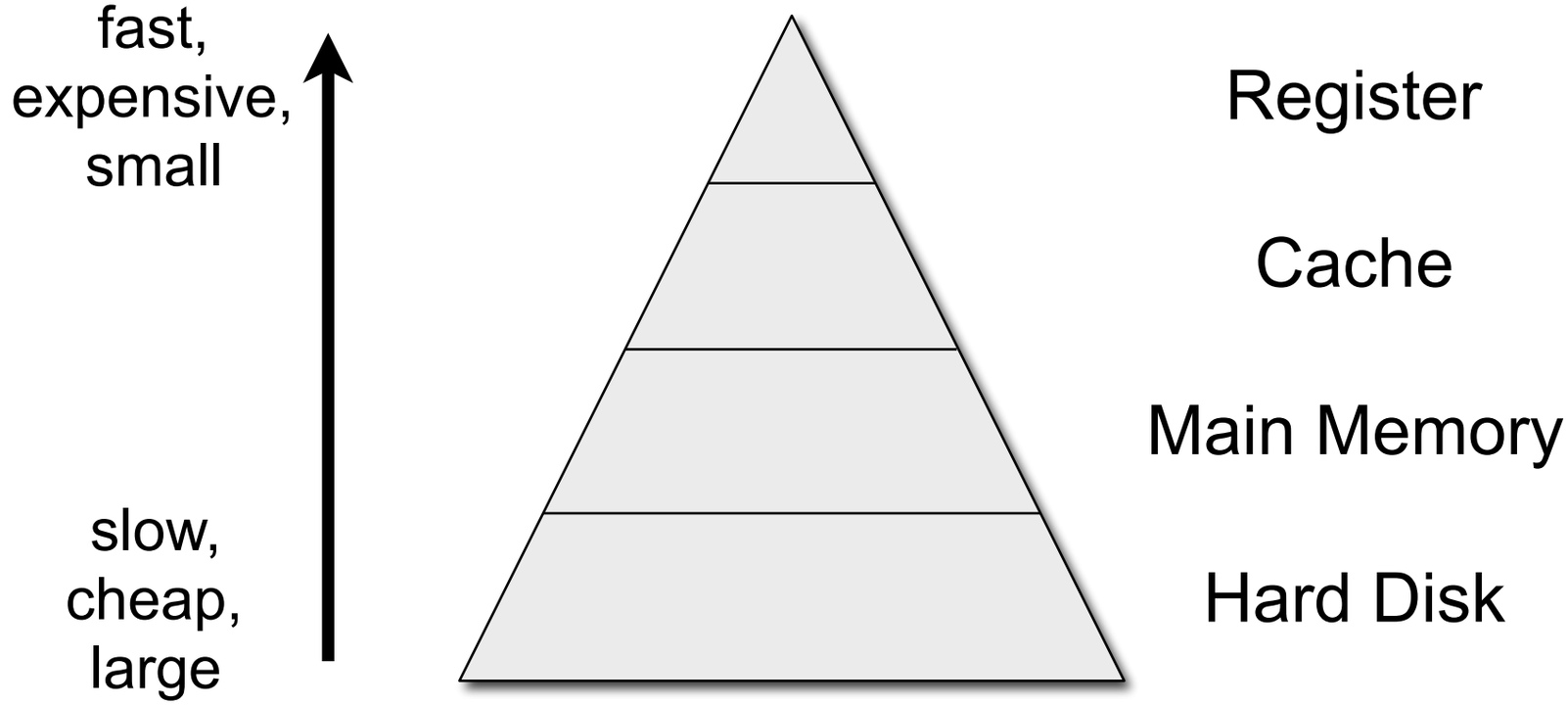
instead of

```
vec3 add(vec3 _v1, vec3 _v2) { … }
```

- Avoids errors like `(v1+v2)=v3;`

Memory Access

- CPUs are highly optimized
  - Memory accesses are typically the bottleneck
- Fast memory is expensive
  - Hierarchical memory organization in caches



Pass by Value?

- Pass function parameters by value?

```
const vec3 add(vec3 _v1, vec3 _v2) { … }
```

- Creates unnecessary temporary objects
  - automatically constructed by compiler!
  - constructor, destructor, mem copies…
  - can be quite expensive for “large” objects

Pass by Reference!

- Pass function parameters by reference!

```
const vec3 add(const vec3& _v1, const vec3& _v2);
```

- A reference is basically a pointer
  - No temporary objects for function parameters

Function Inlining

- Inlining “copies” simple functions in their caller’s context
  - Avoids expensive function calls
- Prerequisites
  - Function definition with function declaration

```
const vec3 add(const vec3& a, const vec3& b)
{ return vec3(a[0]+b[0], a[1]+b[1], a[2]+b[2]); }
```

  - Use keyword `inline`
  - Compiler option `-finline-functions` or `-O3`

Outline

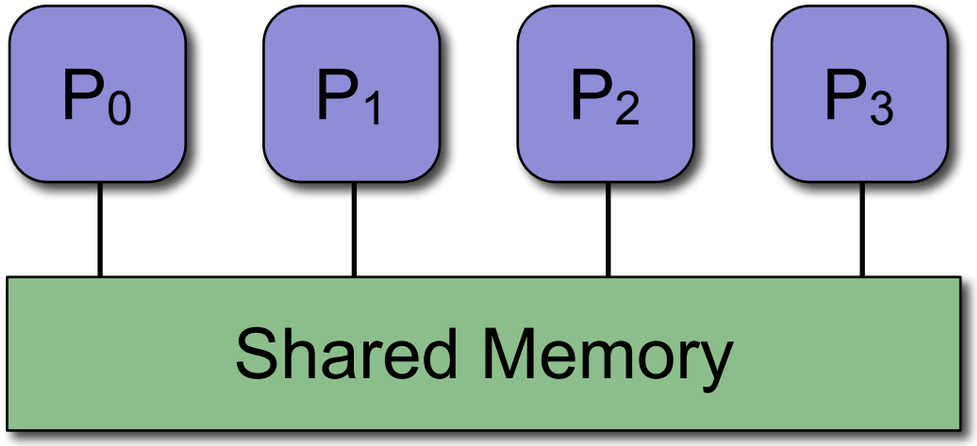
- C++ Crash Course
- RayTracer Design
- Performance optimization
- **Multicore parallelization**

Parallel Computing

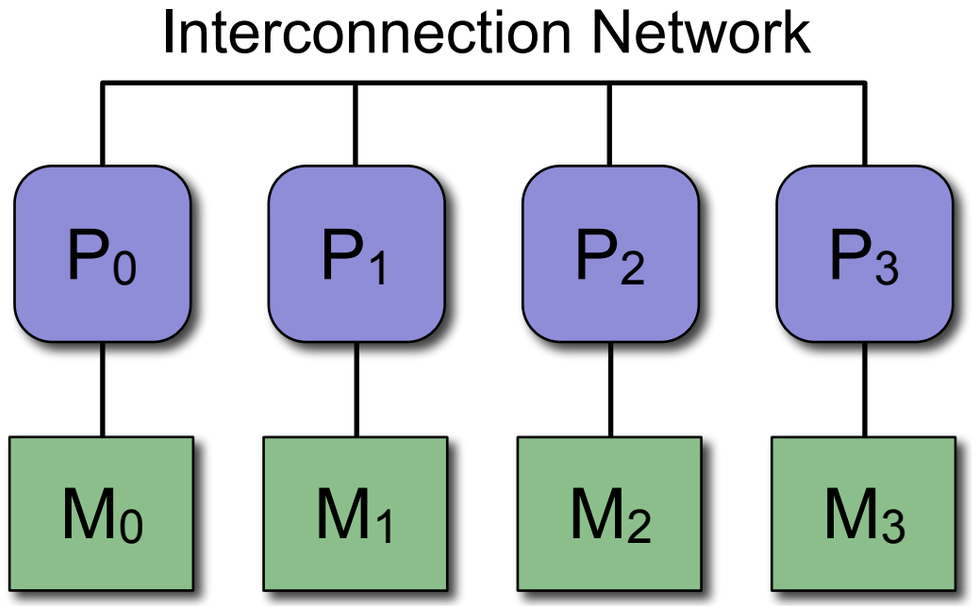
- Why parallel computing?
  - Processors reach physical limits
  - Don’t use faster processors, but more of them!

- Kinds of parallelism
  - instruction level parallelism
  - loop-level parallelism
  - task-level parallelism
  - shared memory vs. distributed computing

Parallel Computing

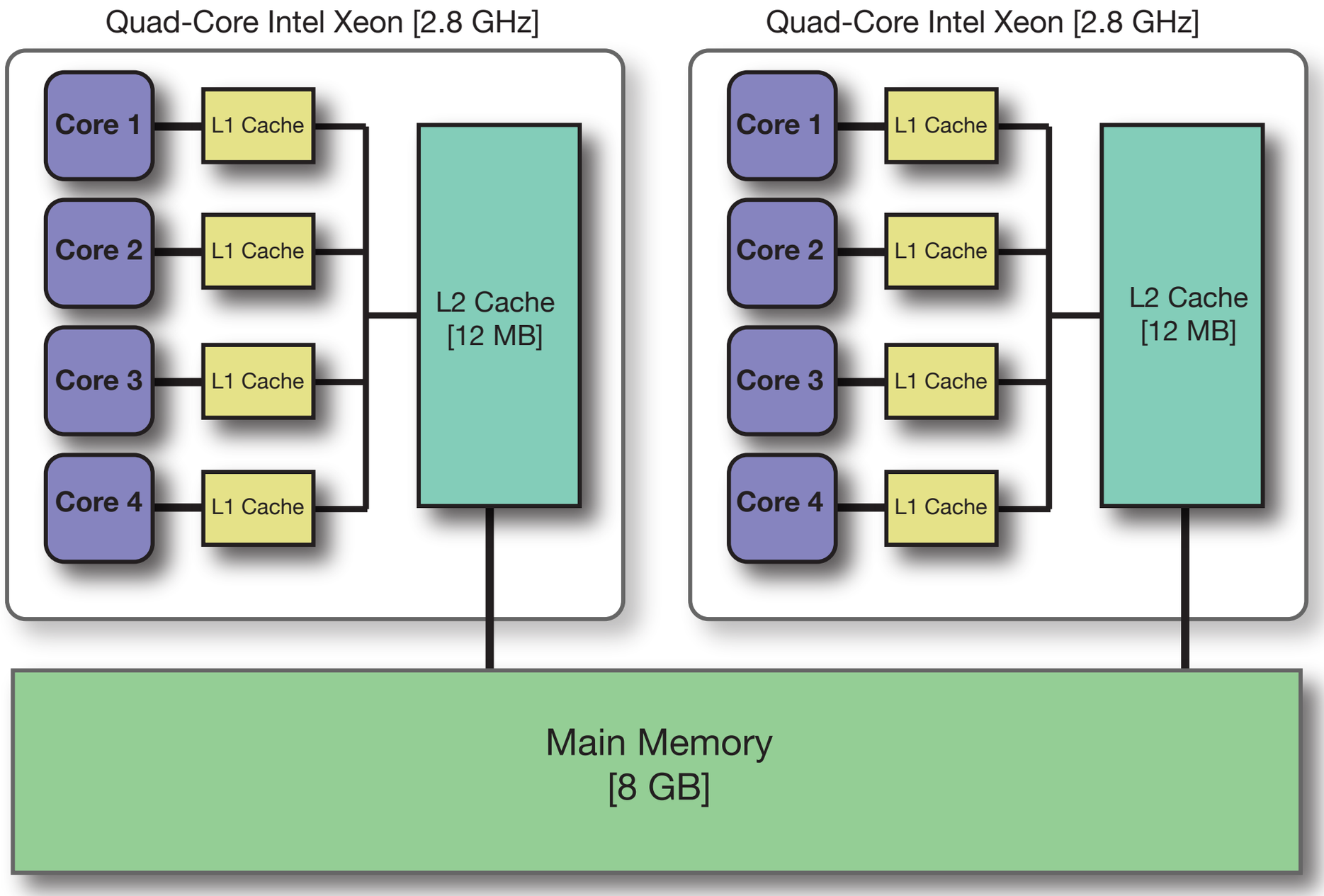


Shared Memory
Parallelization



Distributed
Computing

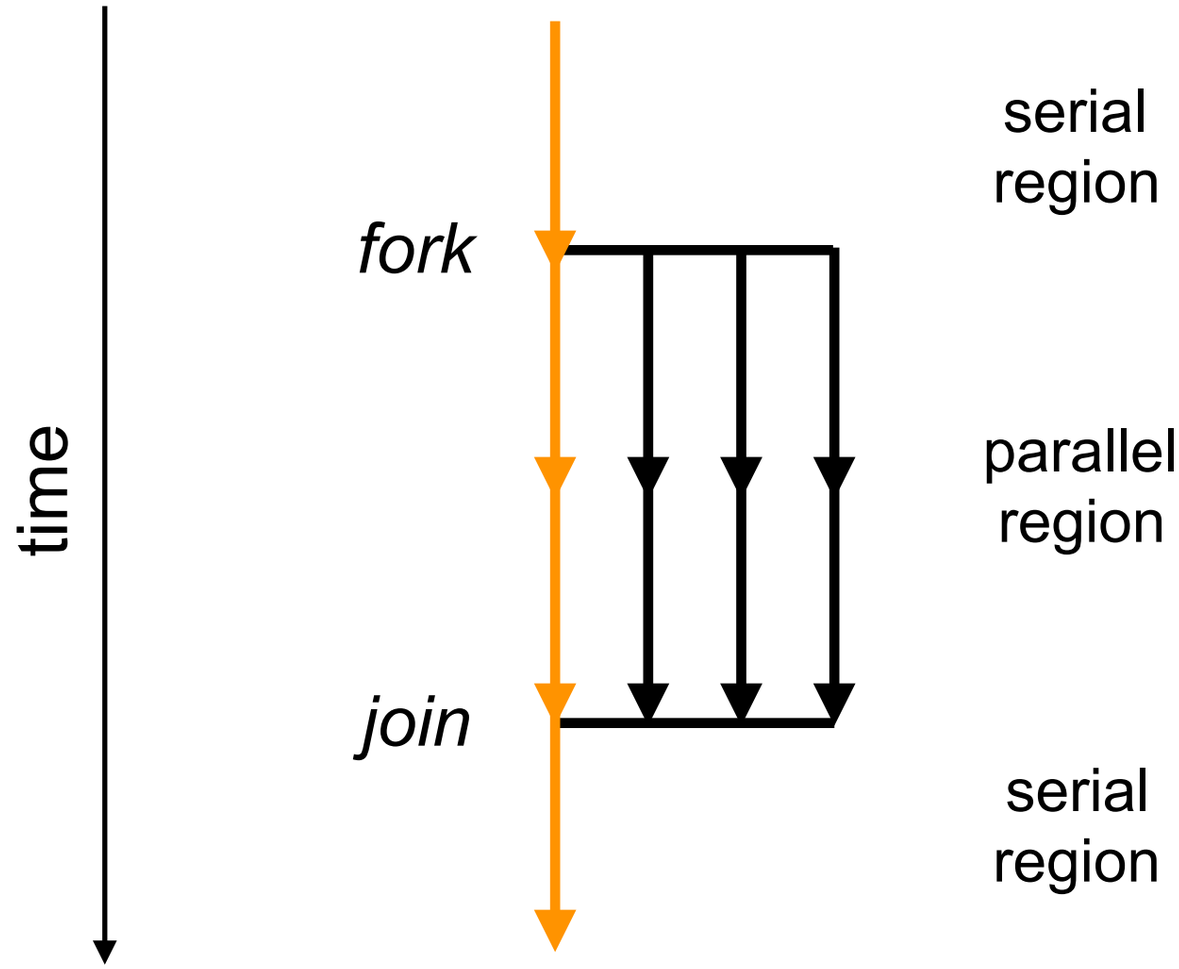
Example: 2007 8-Core MacPro



Multi-Threading

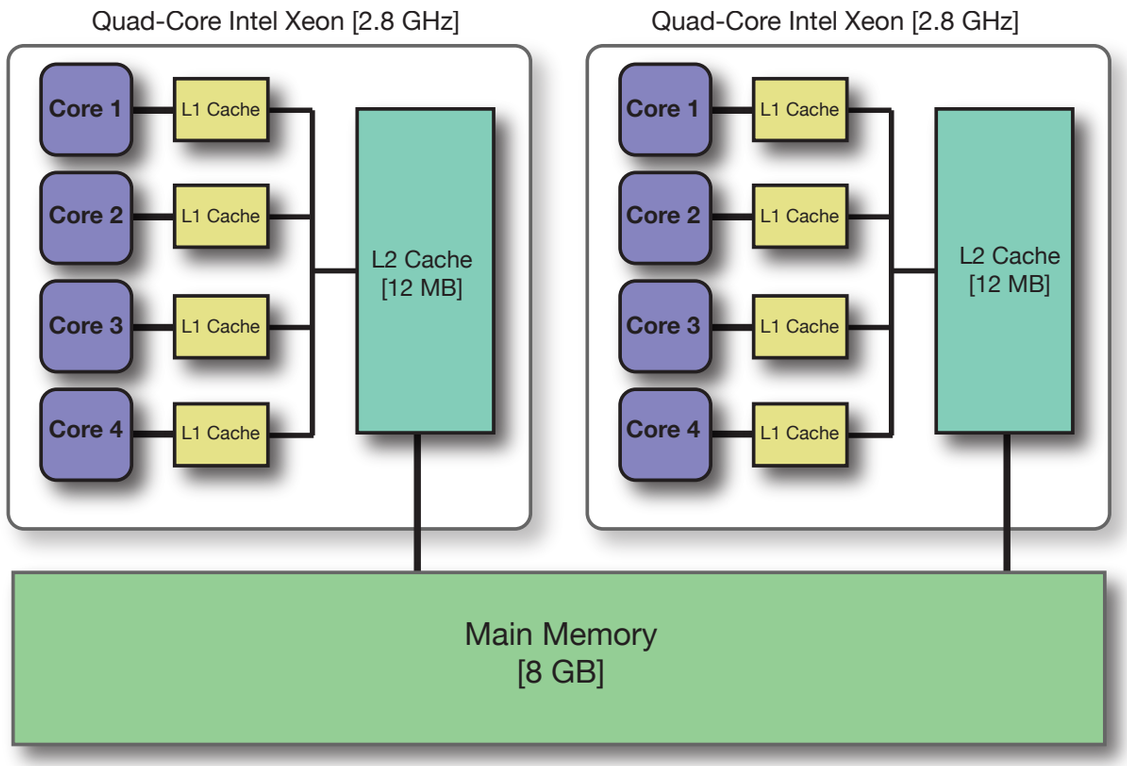
- Parallelization with multiple threads
  - *thread* = lightweight process
- Master thread starts worker threads
- Worker threads process problem
  - run in parallel / simultaneously
  - typically 1 thread per processor core
- Worker threads are joined into master at the end

Multi-Threading



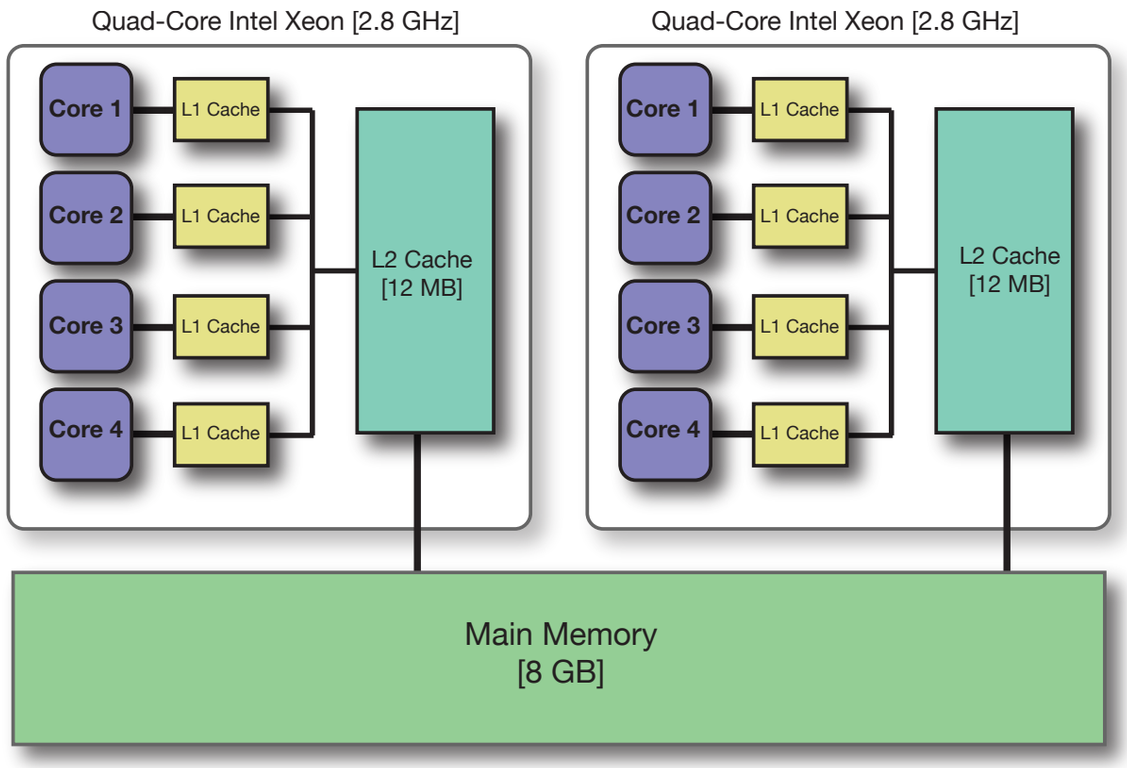
Memory Access

- **Read access**
  - Simultaneous read is ok
  - Shared L2/L3 cache
  - Own L1 cache per core



Memory Access

- **Write access**
  - One thread writes a datum
  - Cache line containing datum gets invalidated
    - in its own cache
    - in other caches as well

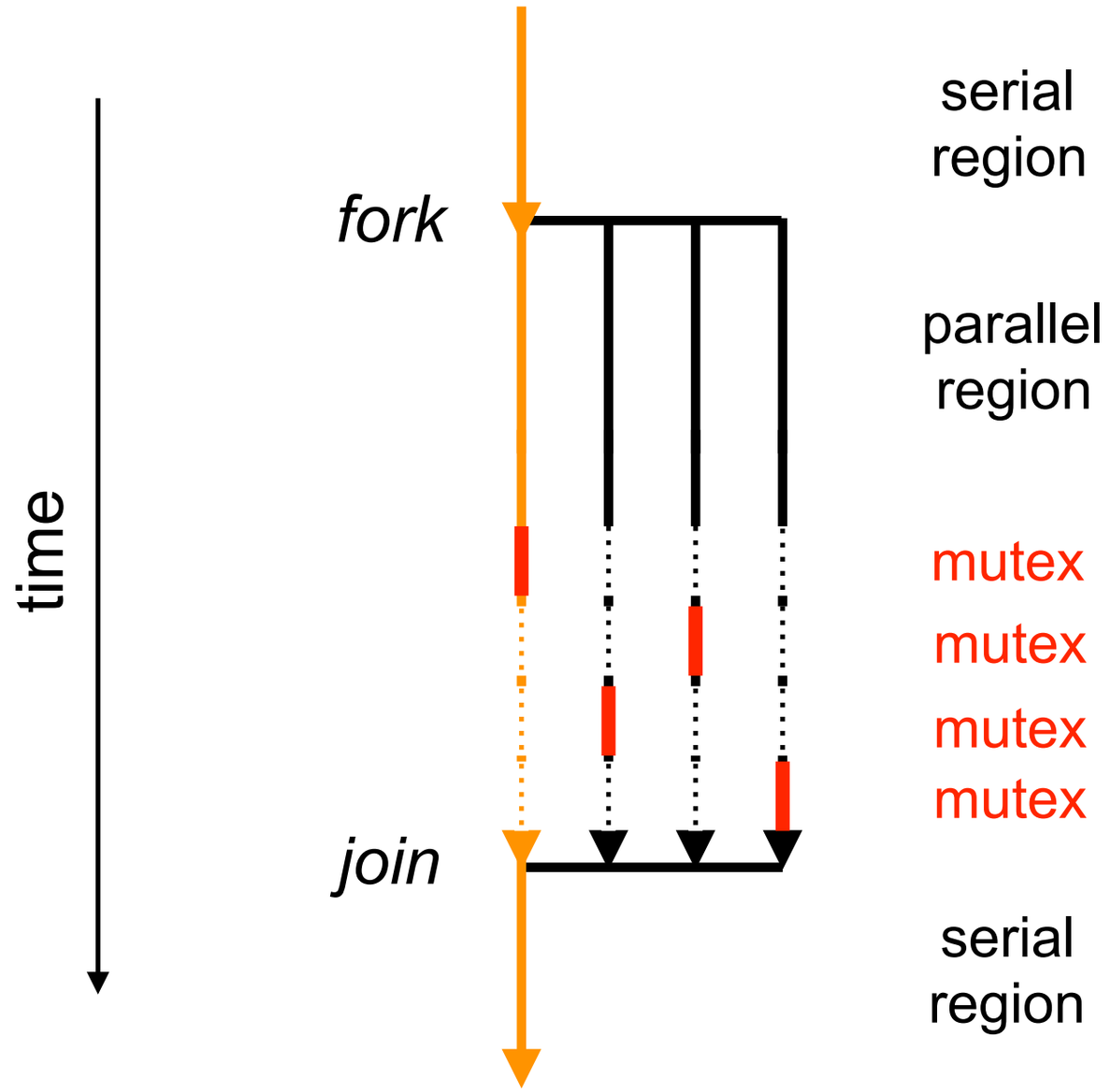


Memory Access

- **Write access**
  - One thread writes a datum
  - Cache line containing datum gets invalidated
    - in its own cache
    - in other caches as well

- Simultaneous writing to same address?
  - result is undefined!
  - prevent by using mutual exclusion (*mutex*)

Multi-Threading



Multi-Threading APIs

- **Pthreads**
  - Portable Operating System Interface (POSIX) threads
  - Library for `fork`, `join`, `mutex`, …
  - Serial code has to be changed quite a bit

- **OpenMP**
  - Open Multi-Processing
  - Compiler directives + library
  - Serial code can stay almost the same
  - Allows for incremental parallelization

OpenMP Hello World

```
#include <stdio.h>


int main(int argc, char** argv)
{

   int id = ...;
   int nthreads = ...;
   printf("hello(%d/%d)\n", id, nthreads);
   printf("world(%d/%d)\n", id, nthreads);

}
```

OpenMP Hello World

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

int main(int argc, char** argv)
{
#pragma omp parallel
 {
   int id = omp_get_thread_num();
   int nthreads = omp_get_num_threads();
   printf("hello(%d/%d)\n", id, nthreads);
   printf("world(%d/%d)\n", id, nthreads);
 }
}
```

OpenMP include

fork

which thread?

how many threads?

join

```
gcc -fopenmp -lgomp -o hello hello.c
```

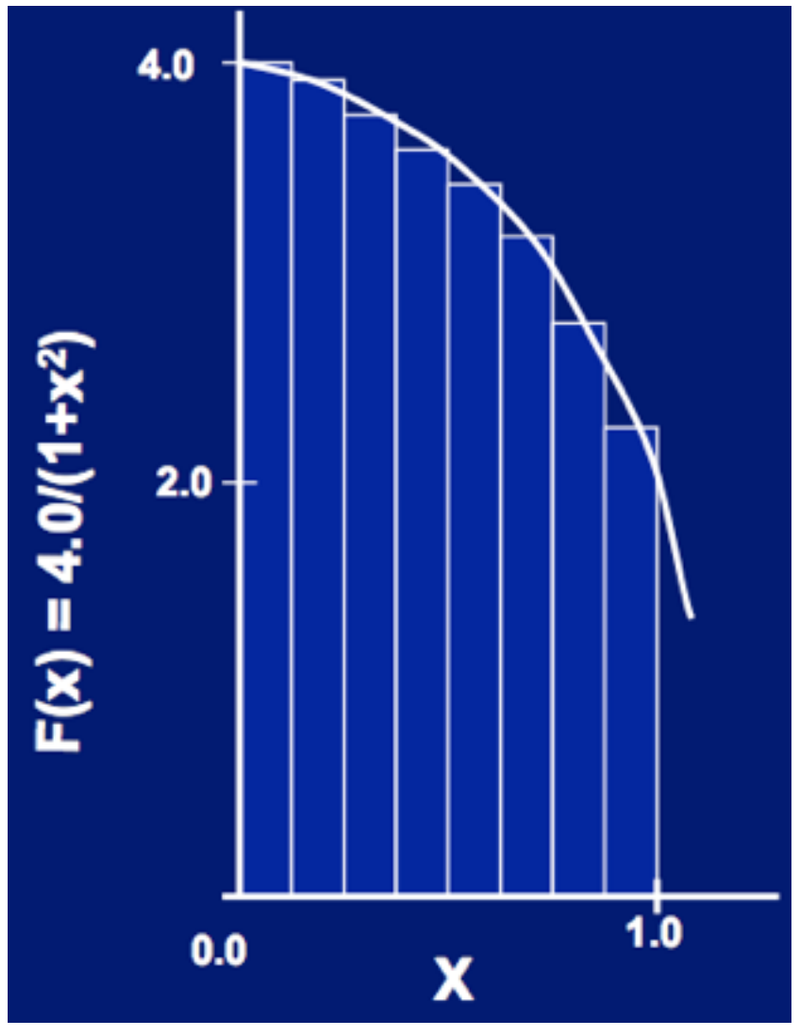
Example: Compute π

- Compute π through numerical integration

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x =: \int_0^1 f(x)\,\mathrm{d}x$$

- Approximate integral by sum
  - Box height $f(x_i)$
  - Box width δx

$$\pi \approx \sum_{i=0}^{N} \delta x \cdot f(x_i)$$



Compute π (serial)

$$\pi \approx \sum_{i=0}^{N} \delta x \cdot f(x_i)$$

```
int      i, N=100000000;
double   x, dx=1.0/(double)N;
double   f, pi=0.0;

for (i=0; i<N; ++i)
{
  x = (i+0.5) * dx;
  f = 4.0 / (1.0 + x*x);
  pi += dx * f;
}

printf("pi = %f\n", pi);
```

Compute π (parallel)

```
  int      i, N=100000000;
  double   x, dx=1.0/(double)N;
  double   f, pi=0.0;

#pragma omp parallel
  {
    int nthreads = omp_get_num_threads();
    int id       = omp_get_thread_num();
    int istart   = id*(N/nthreads);
    int iend     = (id+1)*(N/nthreads);

    for (i=istart; i<iend; ++i)
    {
        x = (i+0.5) * dx;
        f = 4.0 / (1.0 + x*x);
        pi += dx * f;
    }
  }

  printf("pi = %f\n", pi);
```

how many threads?

which thread?

thread's start and end index for the loop

write conflict for `i, x, f, pi`: undefined result!

Compute π (parallel)

```
  int     i, N=100000000;
  double  x, dx=1.0/(double)N;
  double  f, pi=0.0;

#pragma omp parallel private(i,x,f)
  {
    int nthreads = omp_get_num_threads();
    int id       = omp_get_thread_num();
    int istart   = id*(N/nthreads);
    int iend     = (id+1)*(N/nthreads);

    for (i=istart; i<iend; ++i)
    {
        x = (i+0.5) * dx;
        f = 4.0 / (1.0 + x*x);
        pi += dx * f;
    }
  }

  printf("pi = %f\n", pi);
```

private variables: each thread has its own copy

write conflict: undefined result!

Compute π (parallel)

```
  int      i, N=100000000;
  double   x, dx=1.0/(double)N;
  double   f, pi=0.0;

#pragma omp parallel private(i,x,f)
  {
    int nthreads = omp_get_num_threads();
    int id       = omp_get_thread_num();
    int istart   = id*(N/nthreads);
    int iend     = (id+1)*(N/nthreads);

    for (i=istart; i<iend; ++i)
    {
        x = (i+0.5) * dx;
        f = 4.0 / (1.0 + x*x);
#pragma omp critical
        {
            pi += dx * f;
        }
    }
  }

  printf("pi = %f\n", pi);
```

mutex region

correct result, but ***much*** slower than serial version!

Compute π (parallel)

```
  int     i, N=100000000;
  double  x, dx=1.0/(double)N;
  double  f, pi=0.0;
  double  mypi[] = { 0, 0, 0, 0 };

#pragma omp parallel private(i,x,f)
  {
    int nthreads = omp_get_num_threads();
    int id       = omp_get_thread_num();
    int istart   = id*(N/nthreads);
    int iend     = (id+1)*(N/nthreads);

    for (i=istart; i<iend; ++i)
    {
        x = (i+0.5) * dx;
        f = 4.0 / (1.0 + x*x);
        mypi[id] += dx * f;
    }
  }

  pi = mypi[0] + mypi[1] + mypi[2] + mypi[3];

  printf("pi = %f\n", pi);
```

accumulator for each thread

accumulator for each thread

slow because of ***false sharing***:

- `mypi[4]` in <u>one</u> cache line
- writing `mypi[id]` invalidates caches of all threads!

sum the sums

Compute π (parallel)

```
  int      i, N=100000000;
  double   x, dx=1.0/(double)N;
  double   pi=0.0;

#pragma omp parallel private(i,x)
  {
    int nthreads = omp_get_num_threads();
    int id       = omp_get_thread_num();
    int istart   = id*(N/nthreads);
    int iend     = (id+1)*(N/nthreads);
    double mypi  = 0.0;

    for (i=istart; i<iend; ++i)
    {
        x = (i+0.5) * dx;
        mypi += dx * 4.0 / (1.0 + x*x);
    }

#pragma omp critical
    { pi += mypi; }
  }

  printf("pi = %f\n", pi);
```

private accumulator for each thread

only one mutex region for each thread

Compute π (parallel)

```
  int     i, N=100000000;
  double  x, dx=1.0/(double)N;
  double  pi=0.0;

#pragma omp parallel private(i,x) reduction(+:pi)
  {
    int nthreads = omp_get_num_threads();
    int id       = omp_get_thread_num();
    int istart   = id*(N/nthreads);
    int iend     = (id+1)*(N/nthreads);

    for (i=istart; i<iend; ++i)
    {
       x = (i+0.5) * dx;
       pi += dx * 4.0 / (1.0 + x*x);
    }
  }

  printf("pi = %f\n", pi);
```

OpenMP has a specific instruction for such a reduction operation

Compute π (parallel)

```
  int      i, N=100000000;
  double   x, dx=1.0/(double)N;
  double   pi=0.0;

#pragma omp parallel private(i,x) reduction(+:pi)
  {
#pragma omp for
    for (i=0; i<N; ++i)
    {
        x = (i+0.5) * dx;
        pi += dx * 4.0 / (1.0 + x*x);
    }
  }

  printf("pi = %f\n", pi);
```

the `for` directive automatically distributes loops to threads

Compute π (parallel)

```
  int      i, N=100000000;
  double   x, dx=1.0/(double)N;
  double   pi=0.0;

#pragma omp parallel for private(i,x) reduction(+:pi)
  for (i=0; i<N; ++i)
  {
    x = (i+0.5) * dx;
    pi += dx * 4.0 / (1.0 + x*x);
  }

  printf("pi = %f\n", pi);
```

finally combine `parallel` and `for`

Compute π

```
int      i, N=100000000;
double   x, dx=1.0/(double)N;
double   pi=0.0;



for (i=0; i<N; ++i)
{
  x = (i+0.5) * dx;
  pi += dx * 4.0 / (1.0 + x*x);
}

printf("pi = %f\n", pi);
```

serial

```
 int      i, N=100000000;
 double   x, dx=1.0/(double)N;
 double   pi=0.0;

#pragma omp parallel for \
  private(i,x) reduction(+:pi)
 for (i=0; i<N; ++i)
 {
   x = (i+0.5) * dx;
   pi += dx * 4.0 / (1.0 + x*x);
 }

 printf("pi = %f\n", pi);
```

parallel

OpenMP Work Sharing

```
#pragma omp <directive> [options]
```

- `parallel` marks parallel regions
- `for` parallelizes for-loops
- `sections` parallelizes independent blocks
- `single` will be executed by a single thread only
- `master` will be executed by master thread only
- `critical` only one thread at a time (mutex)
- ...

OpenMP Data Sharing

`#pragma omp <directive> [`**`options`**`]`

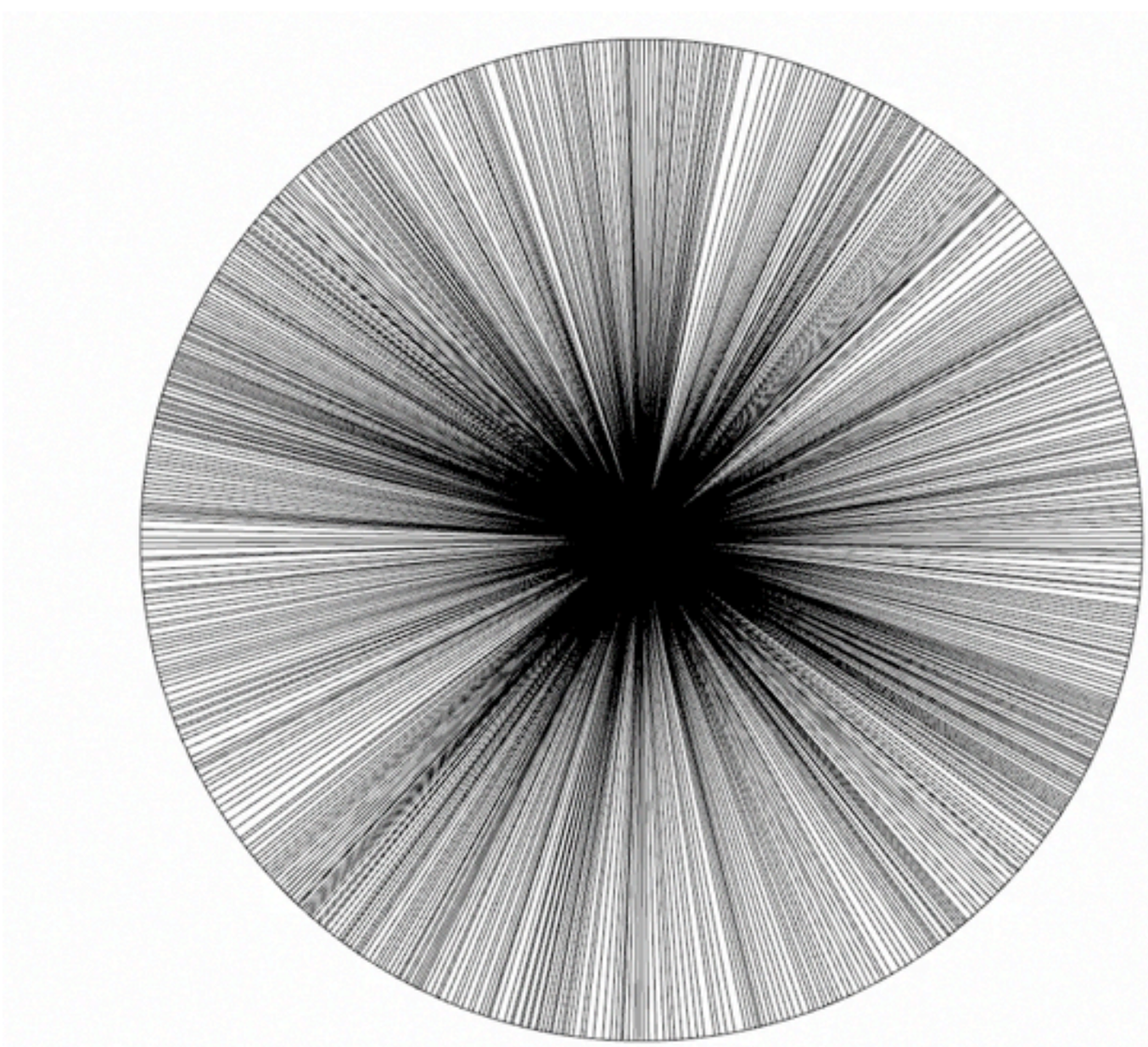
- `shared` — shared variables. be careful with write conflicts!
- `private` — private variables, not initialized
- `firstprivate` — private variables, initialized with the value they have before the parallel region
- `default` — sets default sharing behavior to `none`, `shared`, `private`, ...
  the default is `shared`
- ...

OpenMP Funktionen

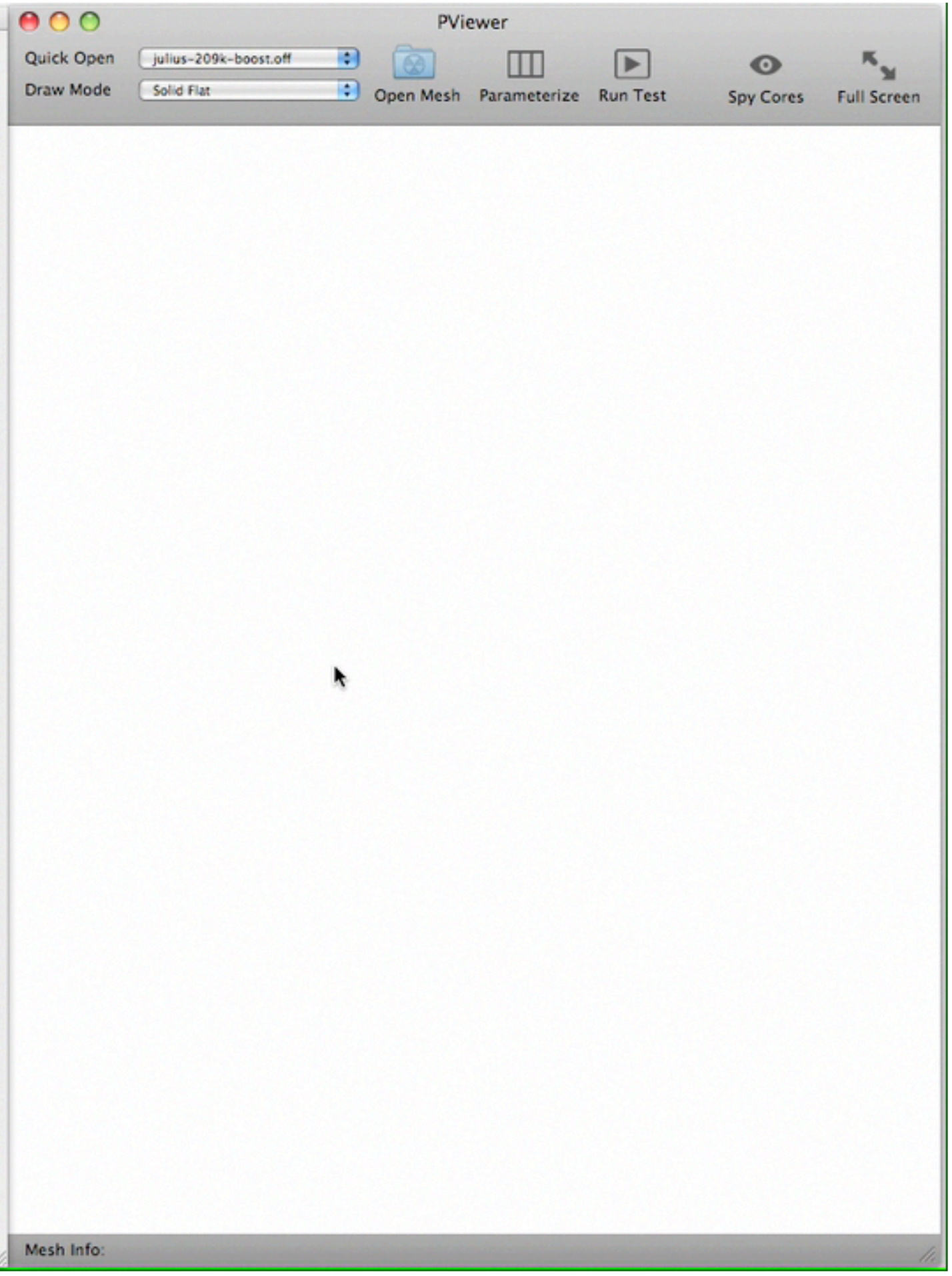
`_omp_<`**`function`**`>`

- `get_num_procs()` number of processor cores
- `get_num_threads()` number of active threads
- `set_num_threads()` set number of wanted threads
- `get_wtime()` get current time, can be used to implement a reliable timer as ($wtime_{end}$ - $wtime_{start}$)
- ...

Example: Mesh Parameterization




Processors
User
System
Nice
CPU 1
CPU 2
CPU 3
CPU 4
CPU 5
CPU 6
CPU 7
CPU 8
+
−


PViewer
Quick Open
julius-209k-boost.off
Draw Mode
Solid Flat
Open Mesh
Parameterize
Run Test
Spy Cores
Full Screen
Mesh Info:

Processors

User System Nice

CPU 1

CPU 2

CPU 3

CPU 4

CPU 5

CPU 6

CPU 7

CPU 8

PViewer

Quick Open julius-209k-boost.off

Draw Mode Solid Flat

Open Mesh Parameterize Run Test Spy Cores Full Screen

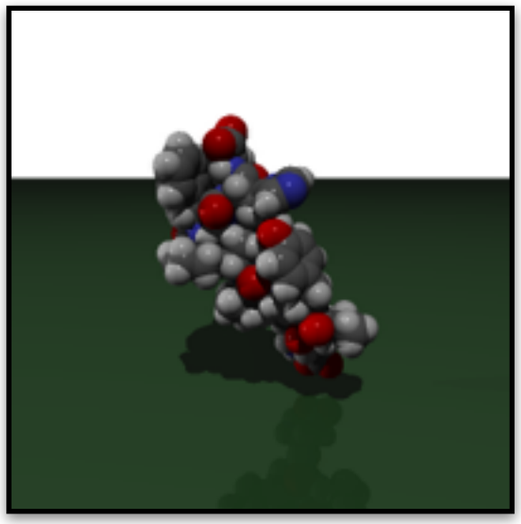
Mesh Info: 209083 Vertices, 416286 Faces, 625368 Edges

Back to Ray Tracing…

```
// the main ray tracing loop
#pragma omp parallel for schedule(dynamic)
for (unsigned int y=0; y<camera.height; ++y)
{
    for (unsigned int x=0; x<camera.width; ++x)
    {
        // generate primary ray for pixel (x,y)
        Ray ray = camera.primary_ray(x,y);

        // trace ray and write color to image
        image(x,y) = trace(ray, 0);
    }
}
```

Benchmark (milliseconds)



molecule2.sce

	Apple llvm MacBookPro	g++ 4.9 MacBookPro	g++ 4.9 Linux 8-core
initial	19240	16130	14927
compiler optimization	4030	2630	2465
pass by reference	1980	1620	1599
function inlining	812	780	801
OpenMP	—	385	194

Analysis of Parallel Programs

- ***Speed-Up***
  - Compare to serial implementation

$$S(n) = \frac{T(1)}{T(n)}$$

speed-up on n processors

time for serial programm

time on n processors

- Linear speed-up $S(n) = n$ is optimal

Analysis of Parallel Programs

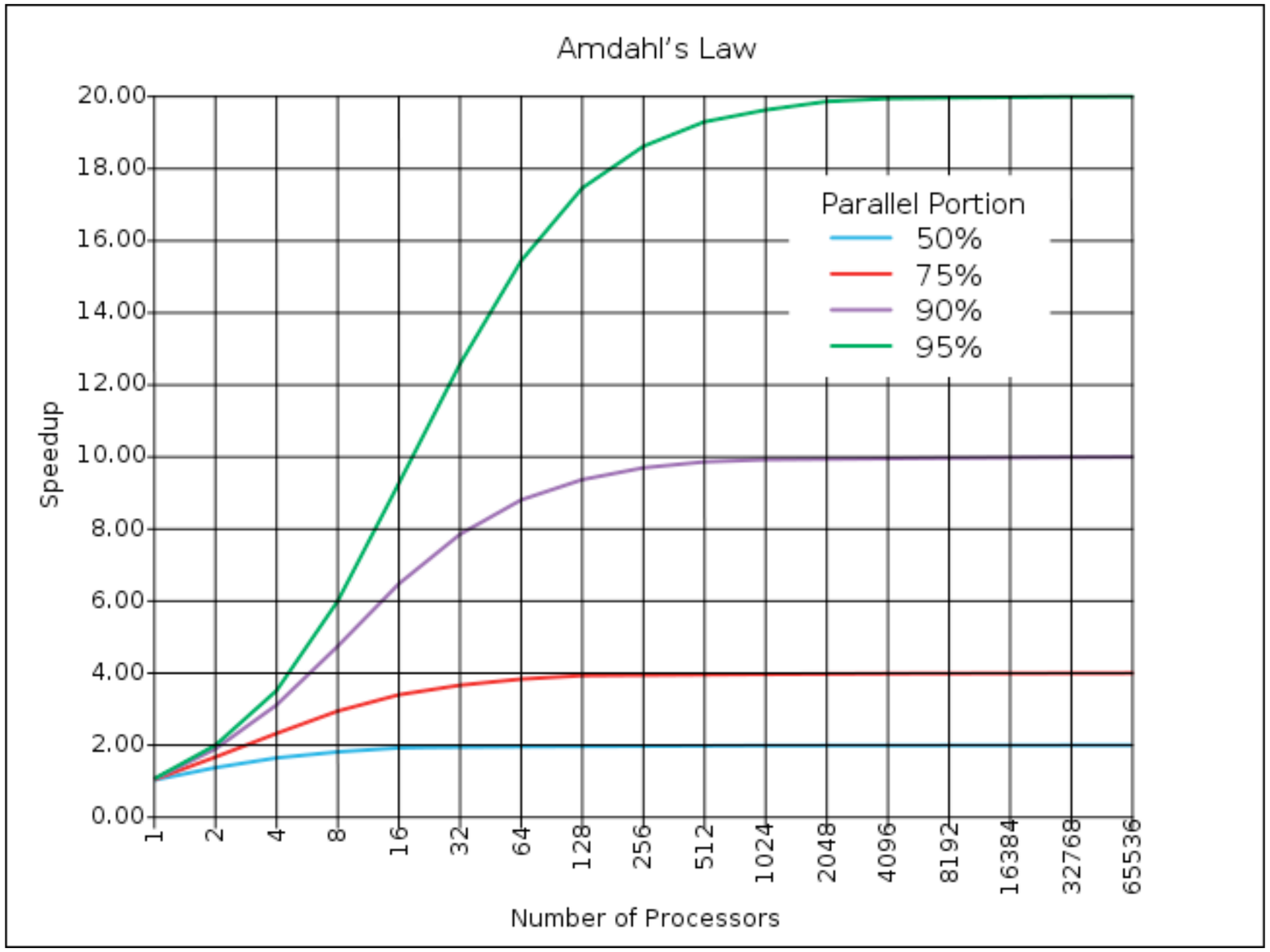
- ***Amdahl's Law***
  - How much can an algorithm be accelerated?
  - Percentage of parallelisable code is *P*
  - Number of processors is *N*

$$S(n) = \frac{T(1)}{T(n)} \leq \frac{1}{(1-P) + \frac{P}{N}} \xrightarrow{N \to \infty} \frac{1}{1-P}$$

serial part does not change with growing *N*

maximum linear speed-up for parallel part

Amdahl's Law



[Wikipedia]

Outline

- ✓ C++ Crash Course
- ✓ RayTracer Design
- ✓ Performance optimization
- ✓ Multicore parallelization

Literature

- Bjarne Stroustrup: ***The C++ Programming Language***, 4th edition, Addison-Wesley, 2013
- Scott Meyers: ***Effective C++***, Addison-Wesley, 2005

- Barbara Chapman, Gabriele Jost, Ruud van der Pas, David J. Kuck: ***Using OpenMP: Portable Shared Memory Parallel Programming***, MIT Press, 2007.