

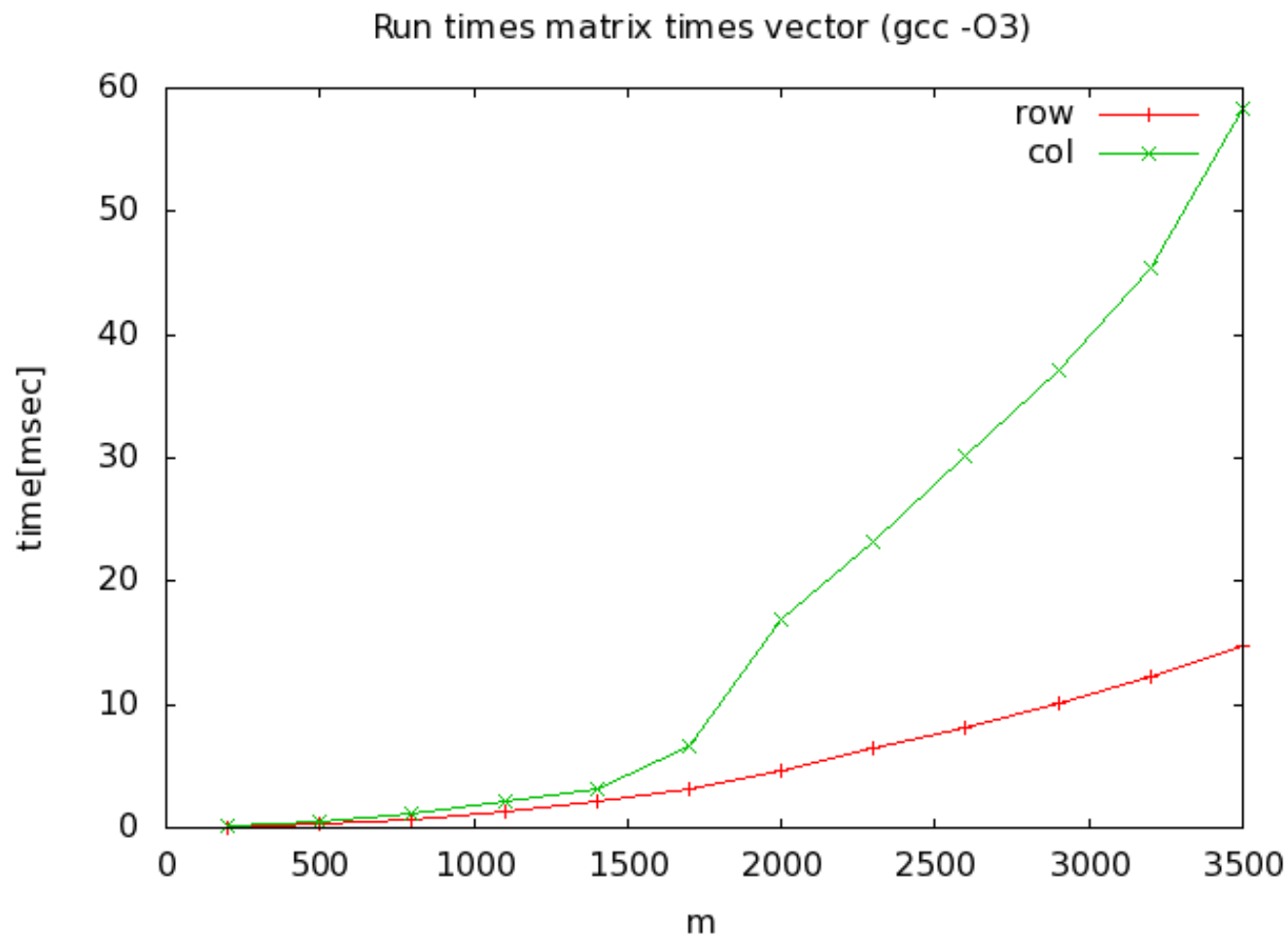
# Mathematical Software Programming (02635)

Module 9 – Fall 2018

Instructor: Bernd Dammann

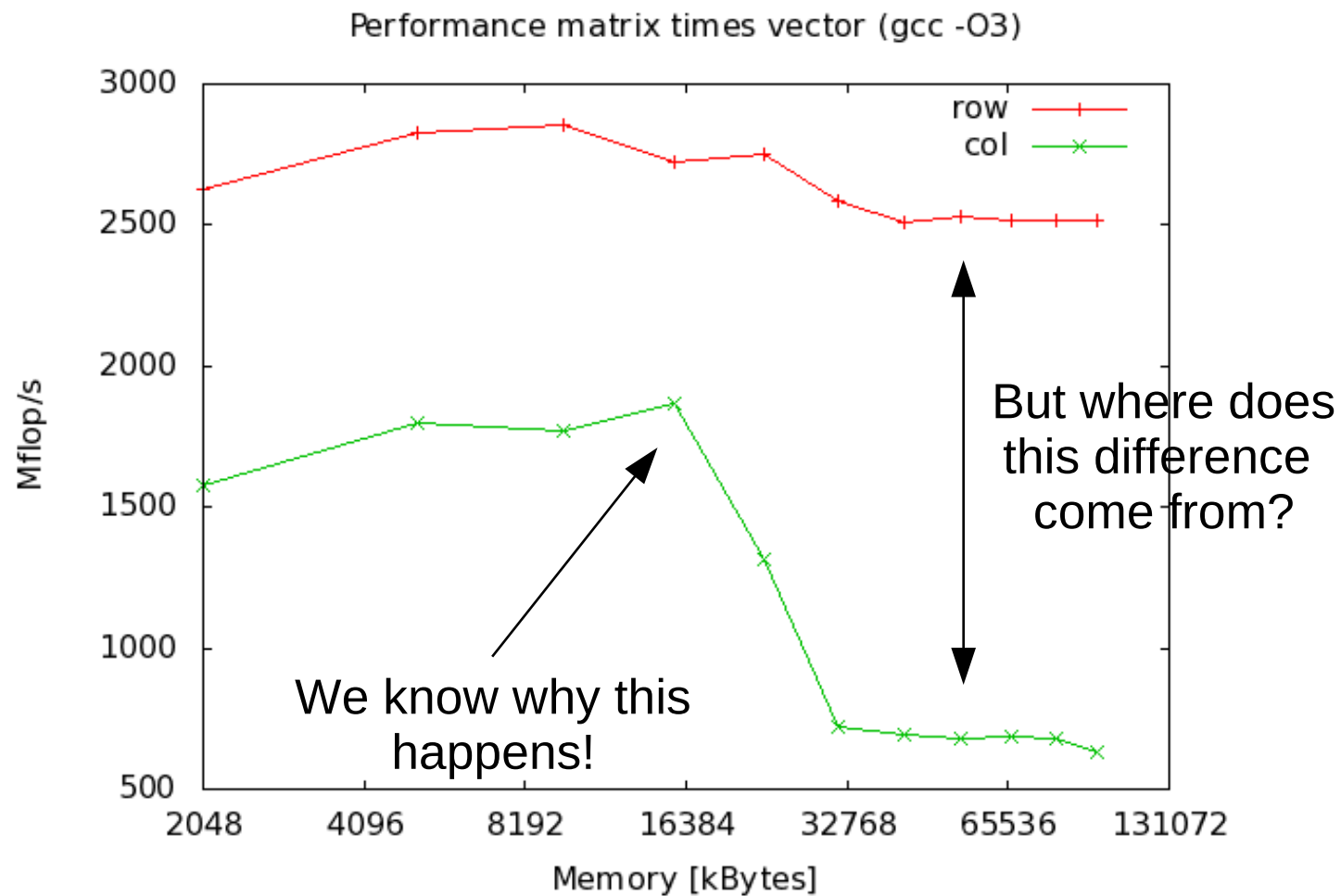
# Re-cap from last weeks

Matrix times vector (runtimes):



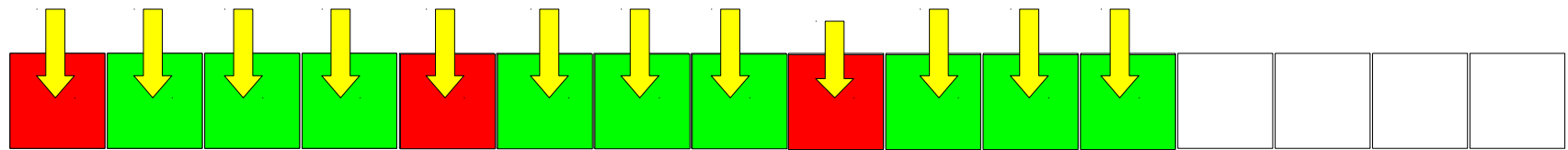
# Re-cap from last weeks

## Matrix times vector (performance):



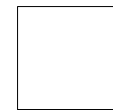
# Re-cap from last weeks

## Accessing vector elements in C:

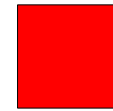


size of a  
cache line

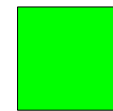
Legend:



vector element



cache miss



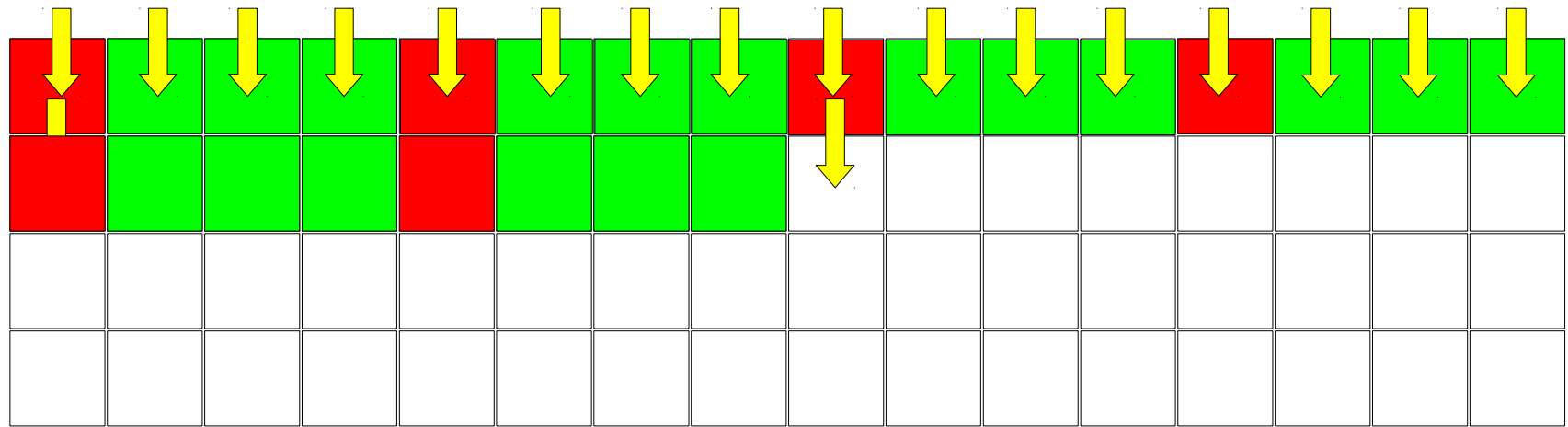
cache hit







memory access

# Re-cap from last weeks

Accessing 2d arrays in C – row wise:

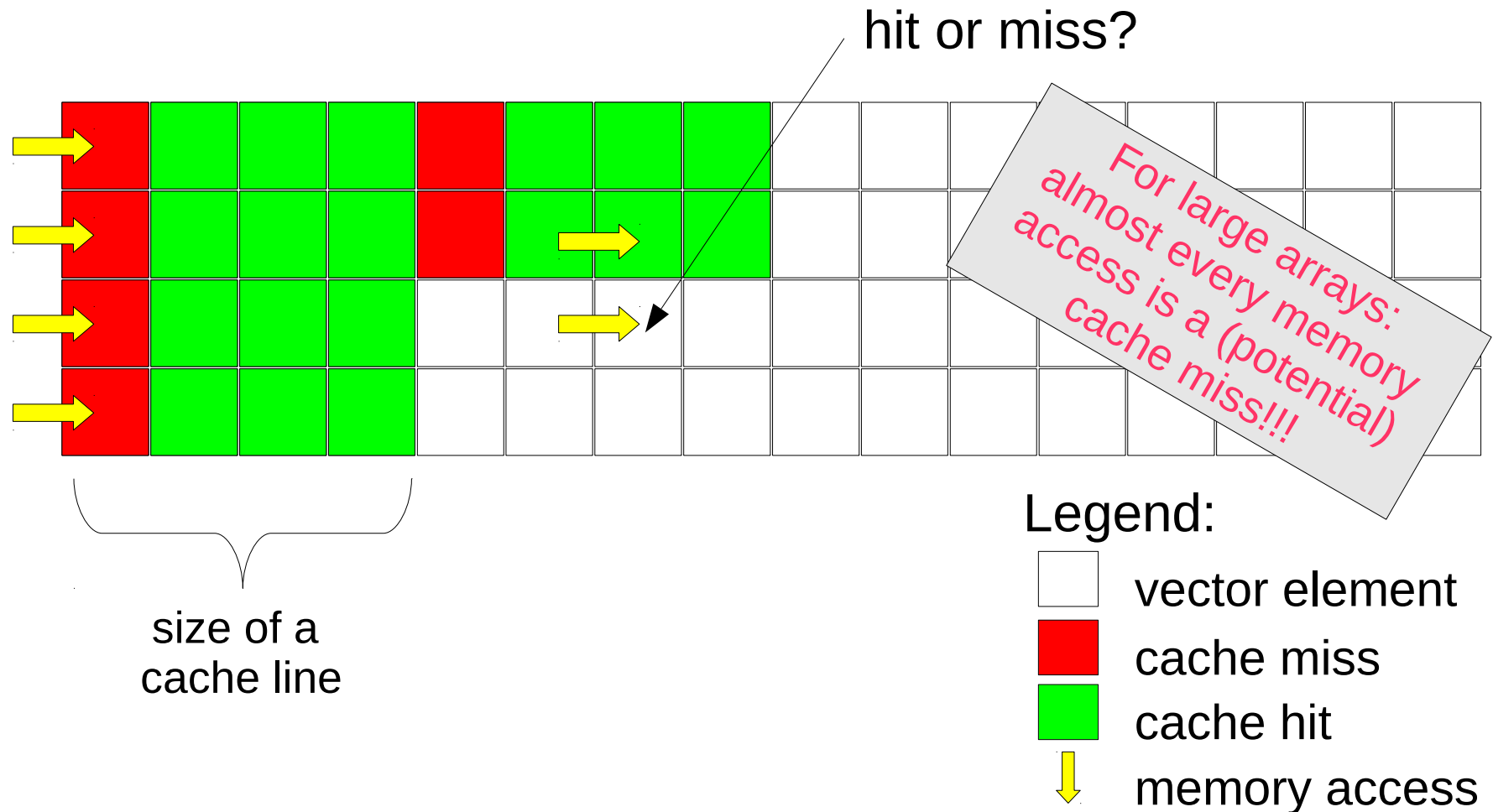


Legend:

-  vector element
-  cache miss
-  cache hit
-  memory access

# Re-cap from last weeks

Accessing 2d arrays in C – column wise:



# Today's topics

- ❑ Parallelism – what is that?
- ❑ Parallel execution models
- ❑ Parallel speed-up – what is that, and what can we expect?
- ❑ Exploiting parallelism with OpenMP

# Today's goal

- ❑ Basic understanding of parallel computations.
- ❑ Implement a parallel version of last week's code matrix times vector, using OpenMP.



# A quick poll

*A given application, Prog.exe, takes 60 seconds to execute on a single-core of a CPU. What is the execution time on all 4 cores of a quad-core CPU?*

- A) 15 seconds
- B) less than 15 seconds
- C) between 15 and 30 seconds
- D) between 30 and 60 secs
- E) more than 60 seconds
- F) I need more information about Prog.exe

# What is Parallelization?

An attempt of a definition:

*“Something”* is parallel, if there is a certain level of independence in the order of operations

*“Something”* can be:

- ▶ A collection of program statements
- ▶ An algorithm
- ▶ A part of your program
- ▶ The problem you are trying to solve

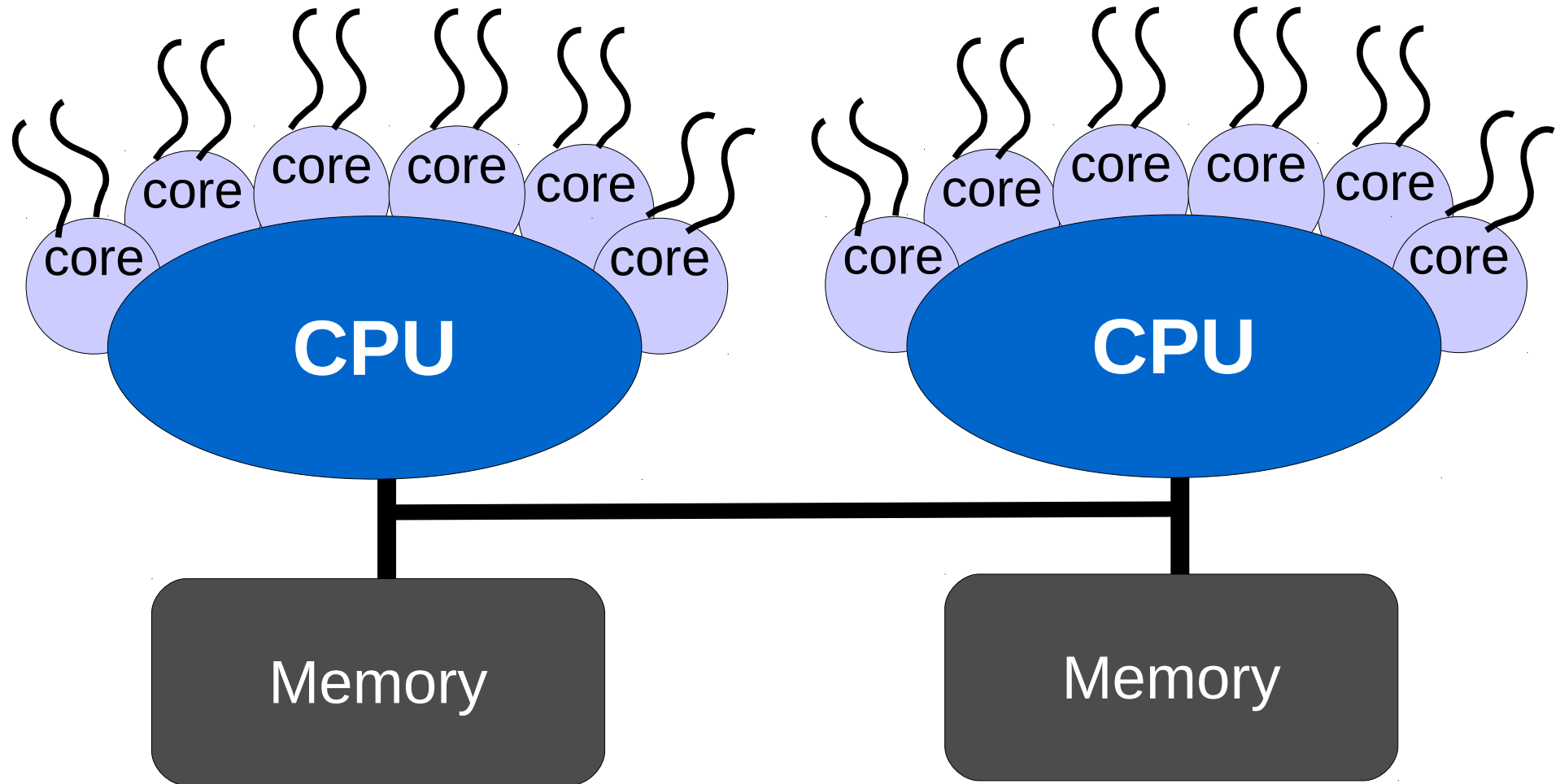


# Parallelism is everywhere

In today's computer installations one has many levels of parallelism:

- ❑ Instruction level (ILP)
- ❑ Chip level (multi-core, multi-threading)
- ❑ System level (multi-socket, i.e. multi-CPU)
- ❑ accelerators: GPU, Intel Xeon Phi, FPGA
- ❑ Cluster: “network of compute nodes”
- ❑ ...

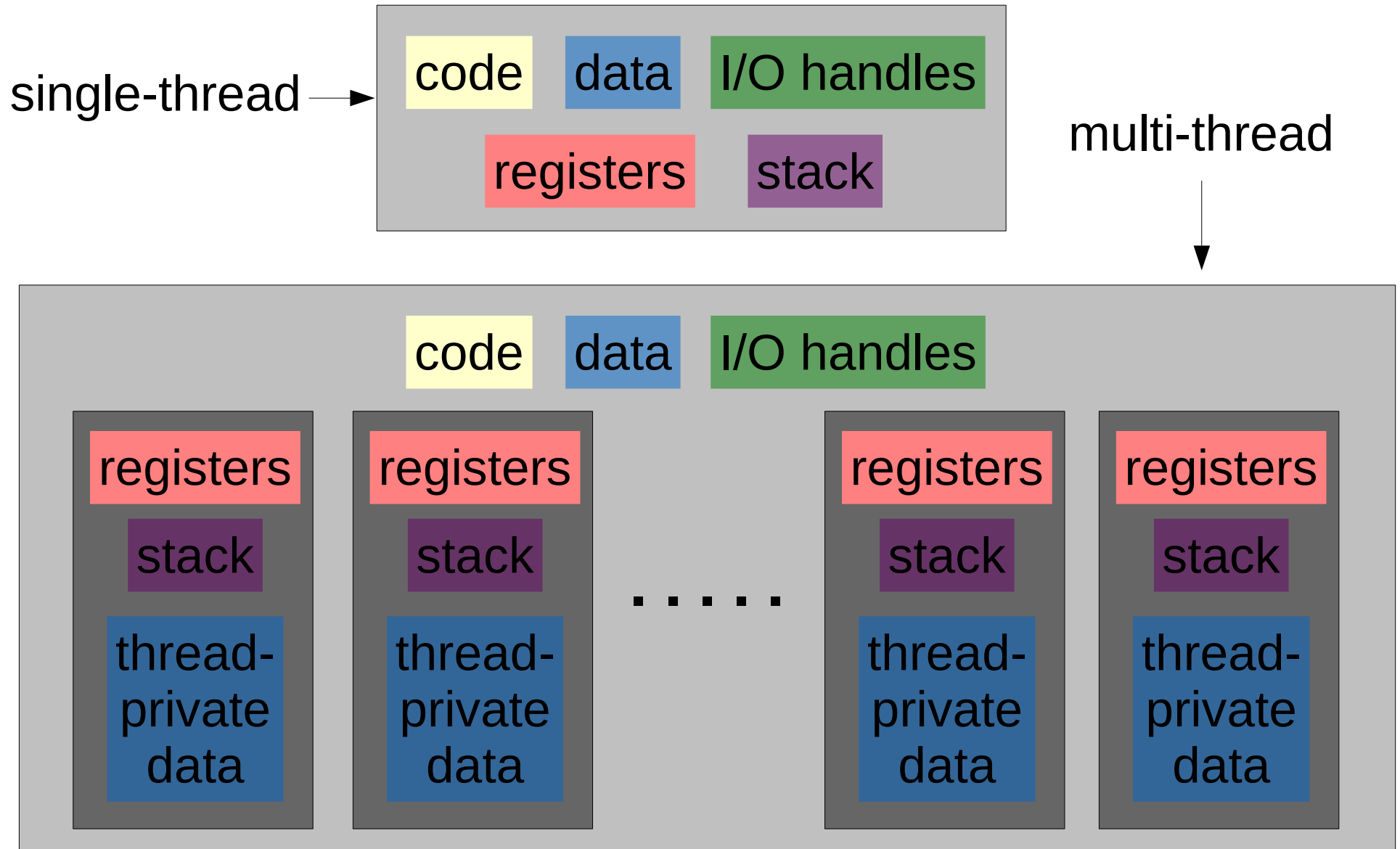
# A typical multi-core setup



a 2-socket, 12-core, 24-(hyper-)threads server  $\Rightarrow$  24 logical CPUs

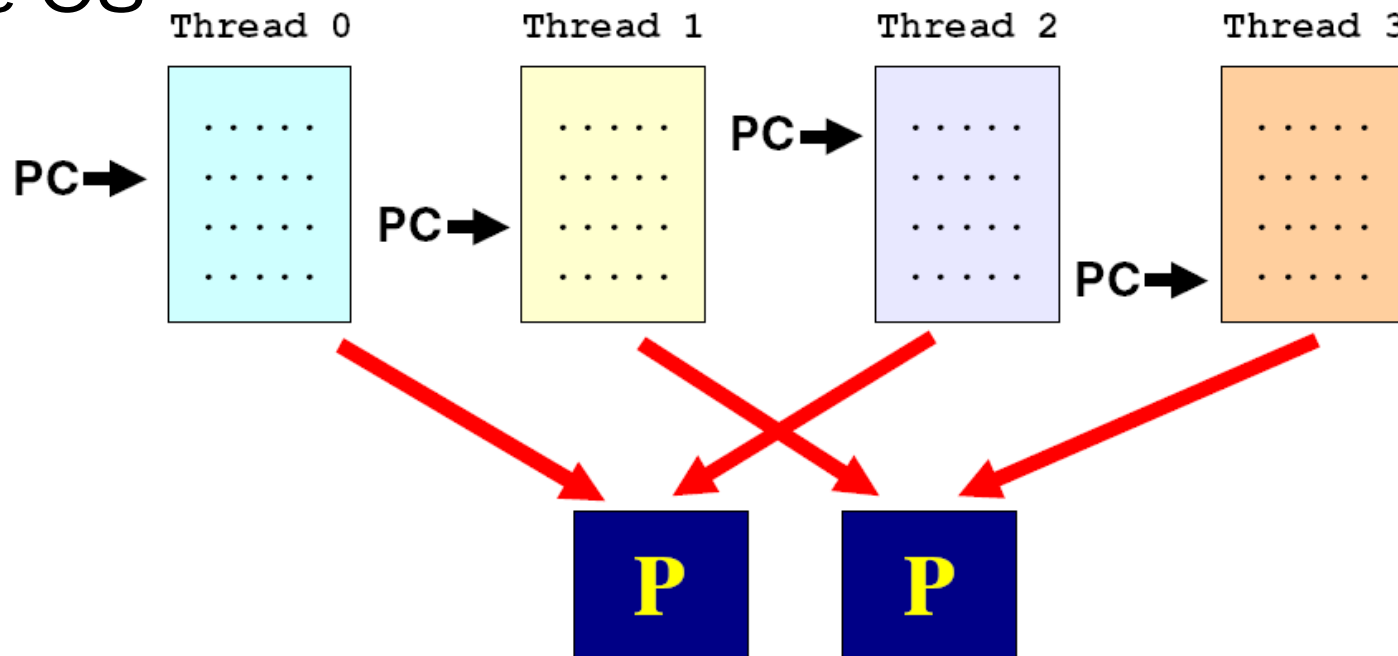
Note: we do not use hyperthreading in our setup!

# Single- vs. multi-threaded



# What is a thread?

- ❑ Loosely said, a thread consists of a series of instructions with it's own program counter ("PC") and state
- ❑ A parallel program will execute threads in parallel
- ❑ These threads are then scheduled onto processors by the OS



# Parallel execution models

## ❑ Multi-threaded:

- ❑ one process
- ❑ multiple threads
- ❑ “communication” (implicit) via shared-memory (shm)
- ❑ limited to one node (computer)

## ❑ Multi-process:

- ❑ multiple processes (usually single threaded)
- ❑ communication via interconnect (network or shm)
- ❑ can run on “any” number of nodes

## ❑ Hybrid: multiple multi-threaded processes

# Timings in parallel programs

- ❑ So far, we have used `clock()` to time the speed of our programs, i.e. the CPU time
- ❑ In parallel programs:
  - ❑ the CPU time will very likely go up (parallel overhead)
  - ❑ `clock()` measures the accumulated time of all threads(!)
  - ❑ we need another measure: wallclock time, i.e. the time the user has to wait to get the result
- ❑ All parallel programming models provide a function to get the wallclock time.
- ❑ On the next slides: wall-time = wallclock time

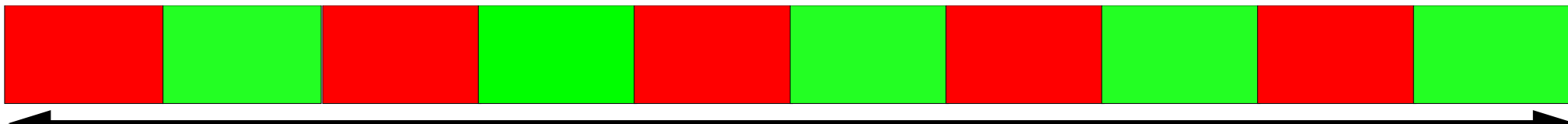


# Parallelism: speed-up

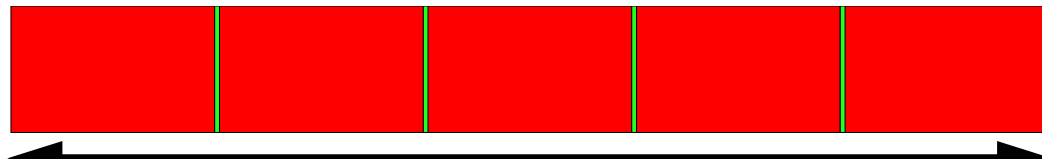
- ❑ What is this “speed-up”?
  - ❑  $S(p) = (\text{wall-time on 1 core}) / (\text{wall-time on } p \text{ cores})$   
 $= T(1) / T(p)$
- ❑ ideal case: linear speed-up, e.g.  $S(p) = p$ 
  - ❑ but: the world is not ideal
  - ❑ parallel overhead: extra instructions, communication, synchronization, etc
  - ❑ not all parts of your code can run in parallel – there will always be sequential code
  - ❑ in general: wall-time goes down – but CPU time goes up!

# Parallelism: speed-up

- let  $f$  be the **parallel fraction** of your code, and  $(1-f)$  the **sequential part**, e.g.  $f = 0.5$

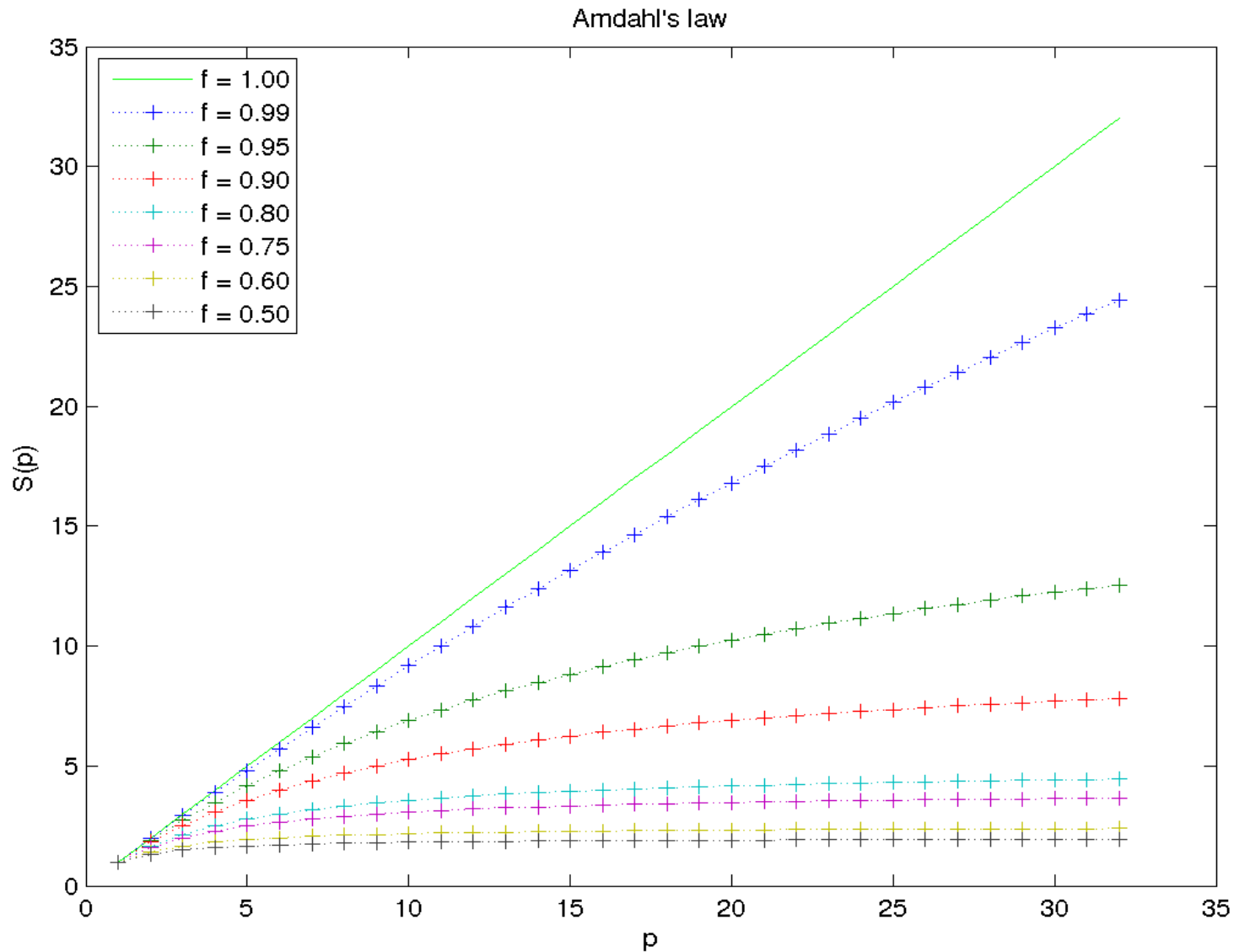

 $T(1)$ 

- What is the max. speed-up, if we had an infinite number of cores ( $p = \infty$ ), and no communication costs, etc?

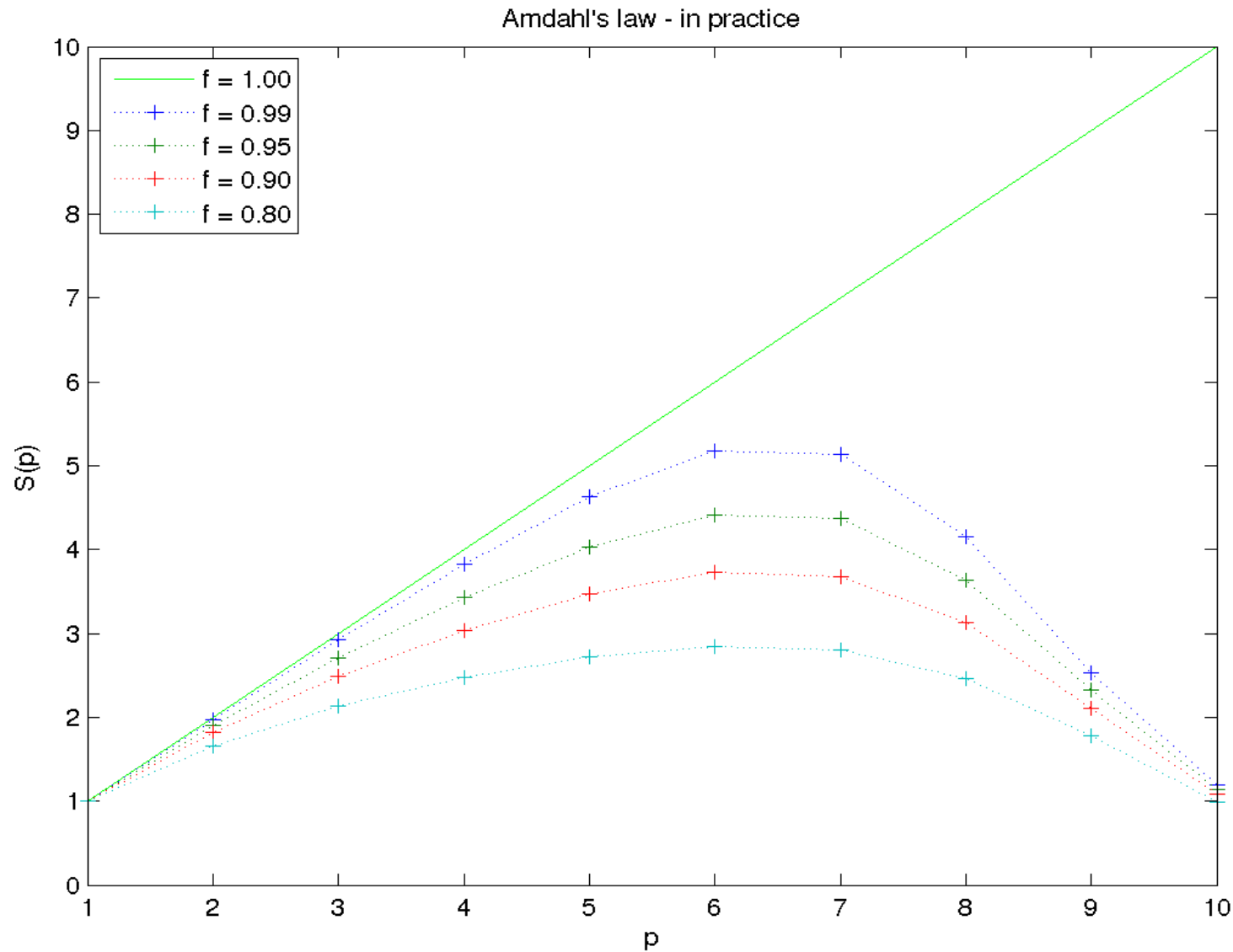

 $T(p=\infty)$ 

$$S = T(1) / T(p=\infty) < 2$$

# Parallelism: Amdahl's law



# Parallelism: Amdahl's law in practice



# Exploiting parallelism using OpenMP

# What is OpenMP?

From [openmp.org](http://openmp.org):

“The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. The OpenMP API defines a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.”

- ❑ OpenMP is a “kind of add-on” to C/C++, Fortran
- ❑ it is not a programming language
- ❑ it requires a compiler that supports OpenMP

# OpenMP components

- ❑ Directives
  - ❑ in your source code
  - ❑ e.g. parallel for-loop
- ❑ Runtime library
  - ❑ support functions
  - ❑ e.g. wallclock timer, etc
- ❑ Environment variables
  - ❑ control program behaviour at runtime
  - ❑ e.g. number of threads to be used

# OpenMP: Hello world

OpenMP version of “Hello world”:

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    #pragma omp parallel
    {
        printf("Hello parallel world!\n");
    } /* end parallel */
    return(0);
}
```



# OpenMP: Hello world

Compile and run ...

```
$ cc -o hello_omp hello_omp.c
```

```
$ ./hello_omp  
Hello parallel world!
```

```
$ OMP_NUM_THREADS=2 ./hello_omp  
Hello parallel world!
```

# OpenMP: Hello world

Compile with OpenMP enabled – and run ...

```
$ cc -fopenmp -o hello_omp hello_omp.c
```

```
$ ./hello_omp  
Hello parallel world!
```

```
$ OMP_NUM_THREADS=2 ./hello_omp  
Hello parallel world!  
Hello parallel world!
```

# OpenMP: Hello world v2

```
#include <stdio.h>
#ifdef _OPENMP
#include <omp.h>
#endif

int main(int argc, char *argv[]) {
    int t_id = 0;
    #pragma omp parallel private(t_id)
    {
        #ifdef _OPENMP
        t_id = omp_get_thread_num();
        #endif
        printf("Hello world from %d!\n", t_id);
    } /* end parallel */
    return(0);
}
```

# OpenMP: Hello World v2

```
$ ./hello_omp2  
Hello world from 0!
```

```
$ OMP_NUM_THREADS=4 ./hello_omp2  
Hello world from 0!  
Hello world from 3!  
Hello world from 1!  
Hello world from 2!
```

- ❑ Note: The order of execution will be different from run to run!
- ❑ The default no. of threads depends on the OpenMP implementation

# OpenMP: Parallel for-loop

## Work-sharing – Loop parallelism:

- ❑ OpenMP implements parallel do/for-loops only!

```
int i;  
float a[N], b[N], c[N];  
  
for (i=0; i < N; i++)  
    a[i] = b[i] = i * 1.0;
```

```
#pragma omp parallel shared(a,b,c) private(i)  
{  
    #pragma omp for  
    for (i=0; i < N; i++)  
        c[i] = a[i] + b[i];  
} /* end of parallel region */
```

*for has to follow the  
pragma – no {... }!*

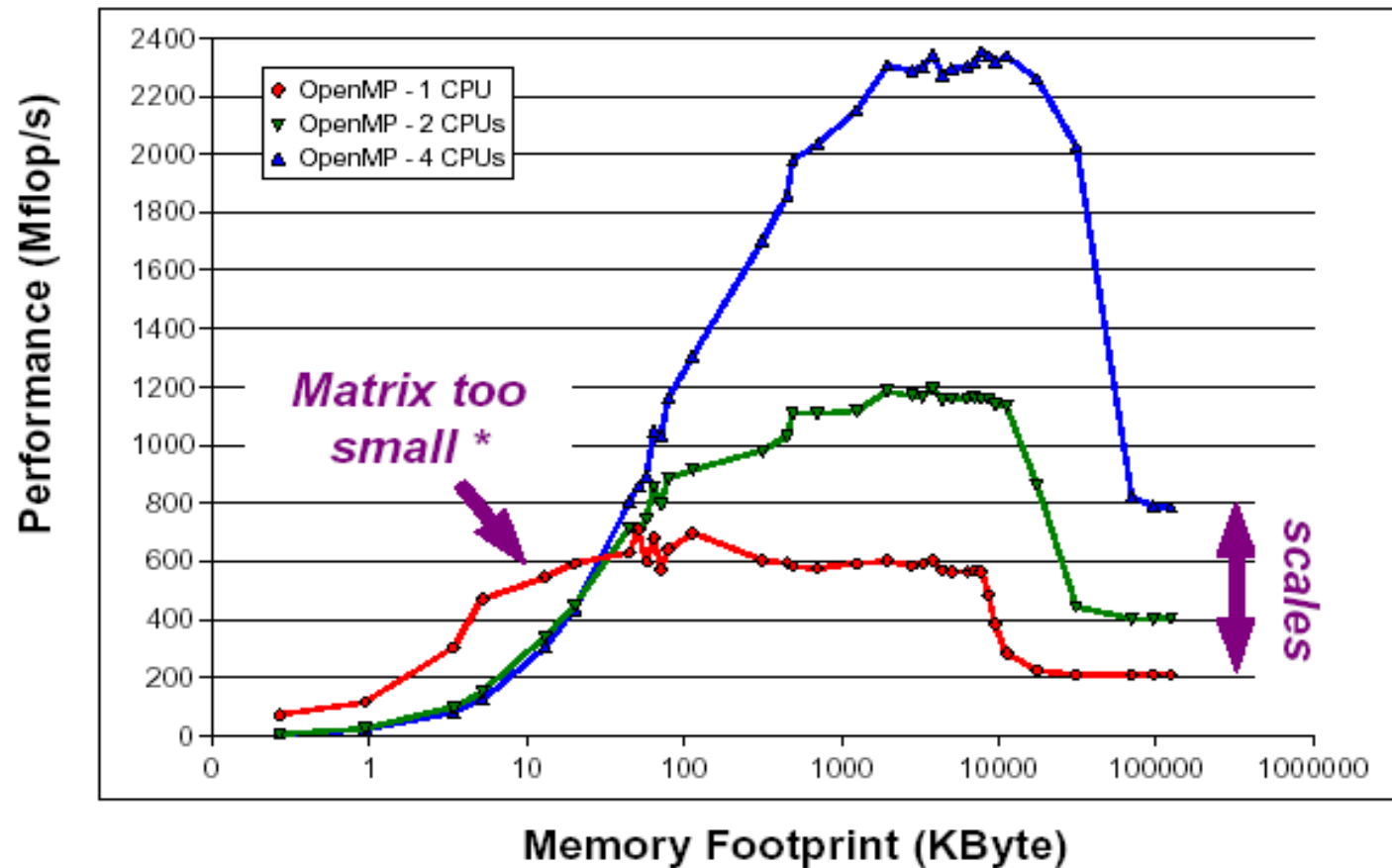
# OpenMP: Parallel for-loop

## Work-sharing – Loop parallelism:

- Another version: combined “parallel for”

```
int i;  
float a[N], b[N], c[N];  
  
for (i=0; i < N; i++)  
    a[i] = b[i] = i * 1.0;  
  
#pragma omp parallel for shared(a,b,c) \  
                        private(i)  
for (i=0; i < N; i++)  
    c[i] = a[i] + b[i];
```

# OpenMP: Matrix times vector



SunFire 6800  
UltraSPARC III Cu @ 900 MHz  
8 MB L2-cache

courtesy: Ruud van der Pas, Oracle

# Summary



# Summary: Parallelism

- ❑ Parallel execution can speed up your code
- ❑ Wallclock time goes down – but the CPU time goes usually up (more resources, parallel overhead!)
- ❑ Don't expect magic ...
  - ❑ remember Amdahl's law!
  - ❑ is your problem too small?
  - ❑ don't use too many threads!
- ❑ Always check your results – compare to serial version!

# Recap: the quick poll

*A given application, Prog.exe, takes 60 seconds to execute on a single-core of a CPU. What is the execution time on all 4 cores of a quad-core CPU?*

- A) 15 seconds
- B) less than 15 seconds
- C) between 15 and 30 seconds
- D) between 30 and 60 secs
- E) more than 60 seconds
- F) I need more information about Prog.exe

*What would you answer now ...*

# Today's exercises

- ❑ Make your first parallel steps:
  - ❑ implement the “Hello World” example from the lecture
  - ❑ this should help you to understand how OpenMP works with your compiler
  
- ❑ Make parallel versions of last week's examples
  - ❑ row-wise version
  - ❑ column-wise version