Motivation - Why parallelize code? (I)

... because now it is possible with our current computing equipment

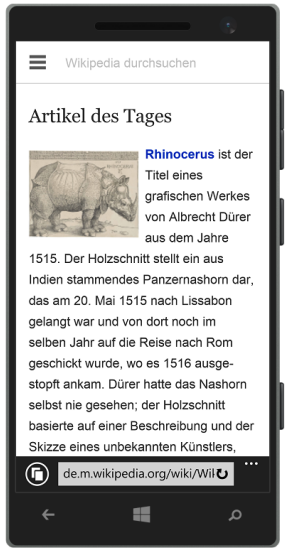
Up to about 2005:

- One core per processor socket
- Dual-socket and multi-socket machines existed on high-end workstations or servers, but these were very expensive.

Now:

- Even very cheap CPUs contain at least two cores
- „Better“ desktops with 4,6, 8 or even 16 Cores
- Large servers with up to 32 cores per socket and several sockets

Motivation - Why parallelize code (II)



up to 8 cores

up to 8 cores

up to 32 cores

304128 cores

Motivation - Why parallelize code (III)

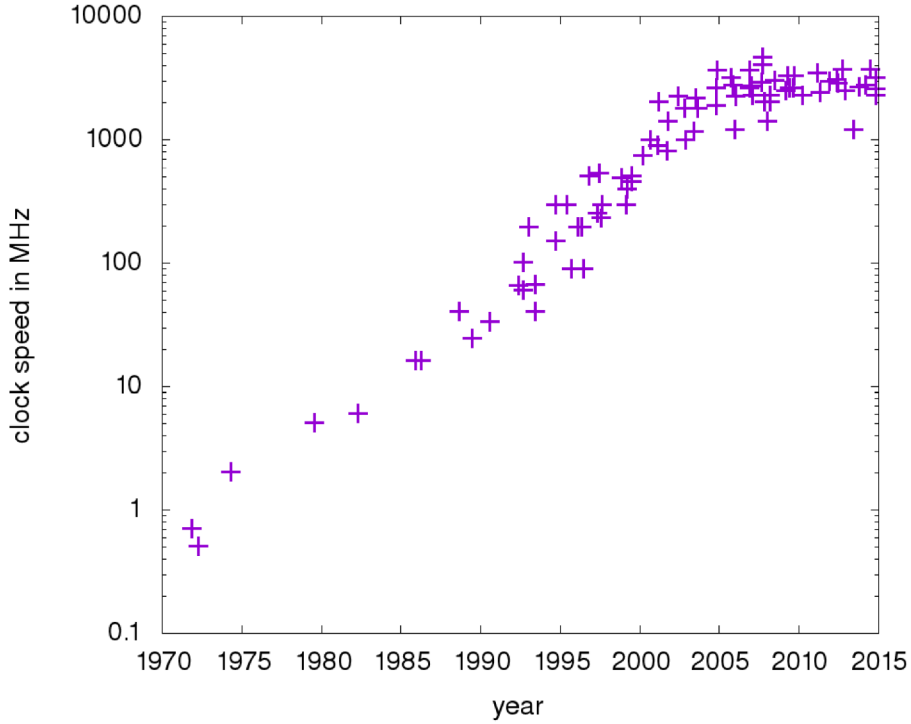
We can, but why should we parallelize code and not wait for CPUs with higher clock frequencies? We more or less have to..

- The increase of performance per CPU is not as fast as it used to be (waiting for a 100 GHz CPU does not make sense)
- Many scientific problems can be (partly) divided into subproblems that can be solved independently of each other

Data up to 2010 collected by:
M. Horowitz, F. Fabonte, O. Shacham, K. Olokotun, L Hammons and C. Batten

Data from 2010-2015 collected by K. Rupp

https://github.com/karlrupp/microprocessor-trend-data/tree/master/40yrs



Problems of Parallelizing Codes (I)

Most problems can not be **entirely** decomposed into independent parts.

For example, in time-dependent simulations, the results for a timestep depend on the results of the previous timestep(s).

Thus, all computations belonging to a timestep have to be completed until the computation of the next timestep can start.

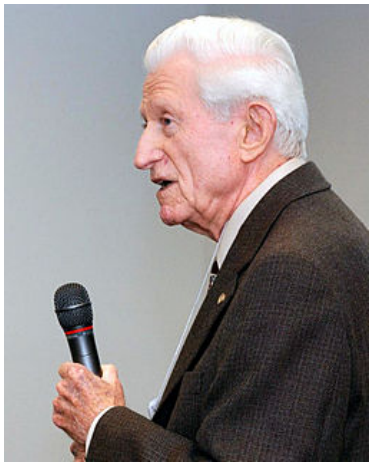
Input/Output to the same file has at to be at least partially serial.

Problems of Parallelizing Codes (II)

The maximal performance gain as a function of the fraction of serial code and the number of parallel processes is described by *Amdahl's law*.

$$\eta_s = \frac{1}{t_s + \frac{t_p}{n_p}}$$

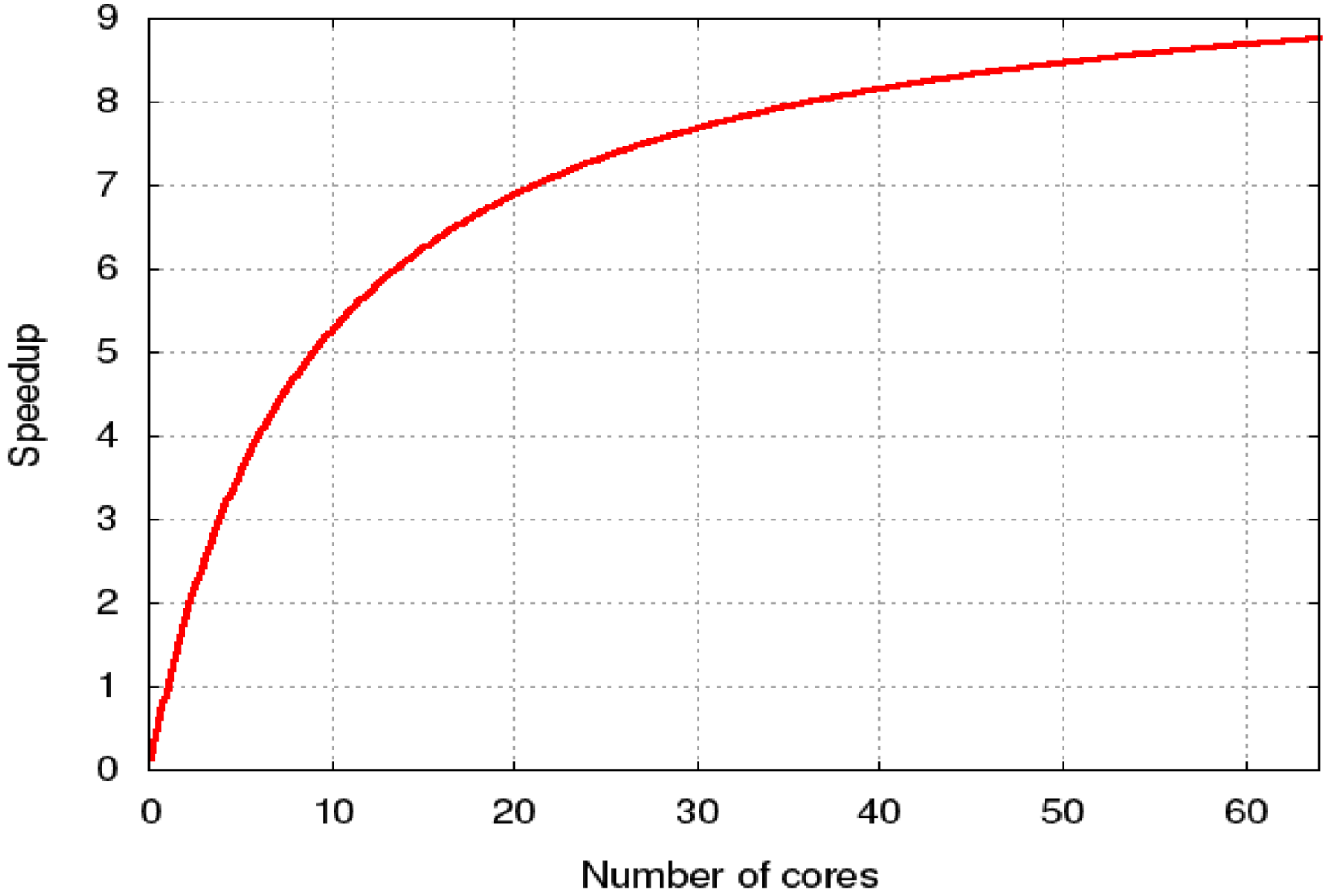
η_s : Speedup of the parallel code compared to the un-parallelized code
t_s : Fraction of code that can not be parallelized
t_p : Fraction of code that can be parallelized
n_p : Number of parallel processes



Gene Amdahl (1922-2015)

Problems of Parallelizing Codes (III)

Example: Amdahl law for a serial fraction of 10%



Problems of Parallelizing Codes (IV)

Amdahl's law is a raw approximation - this is both bad and good news:

- It does not consider the often rising communication overhead for a large number of parallel processes
- It assume the serial fraction to be constant.

  In reality it depends on the size of the computed system, I/O performance, ...

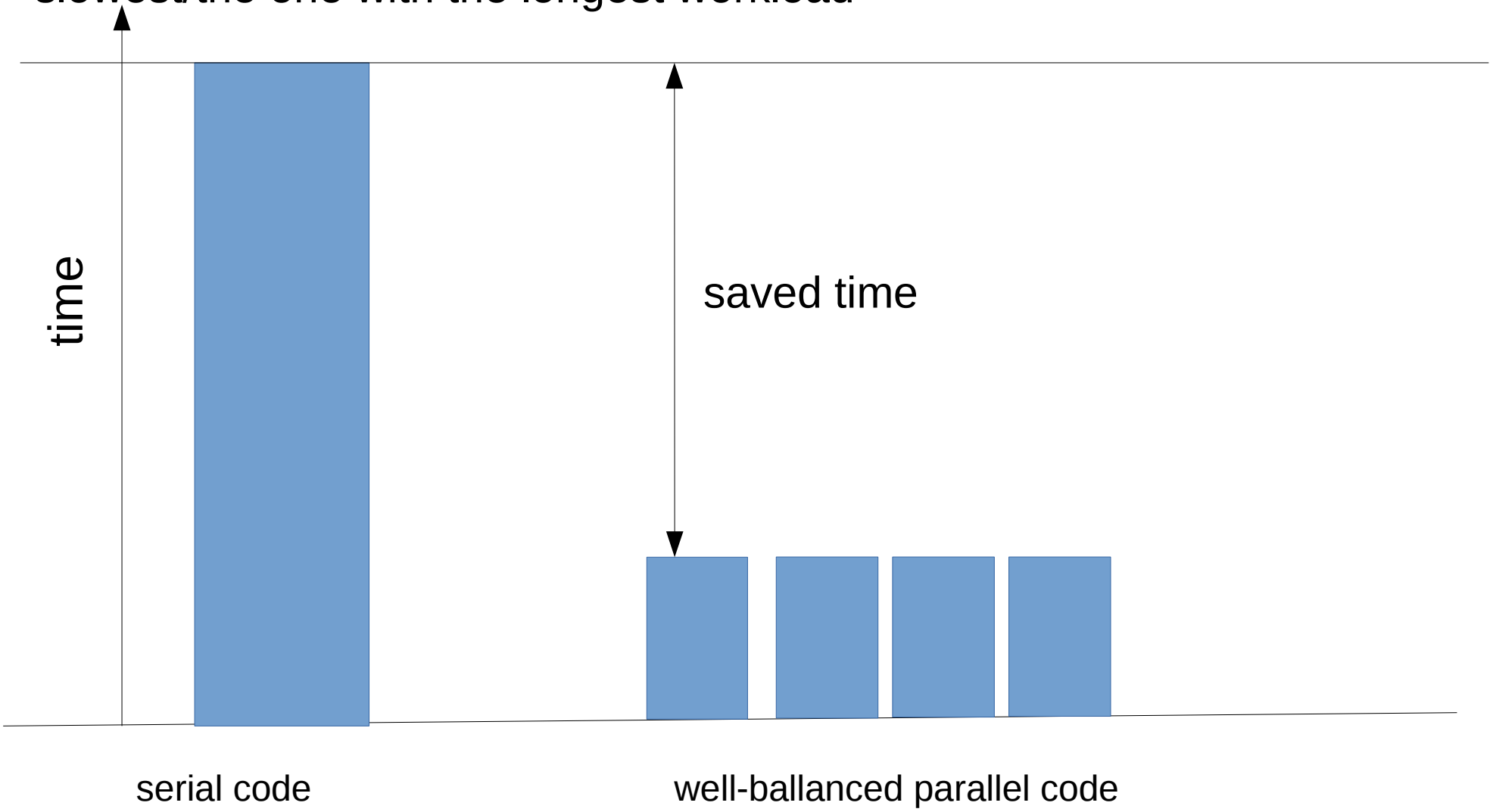
  Fortunately, the serial fractions tends to decrease for larger data sets.

Problems of Parallelizing Codes (V) - Load Balancing

If there are several parallel processes, the system is defined by the slowest / the one with the longest workload

Problems of Parallelizing Codes (VII) - Load Balancing

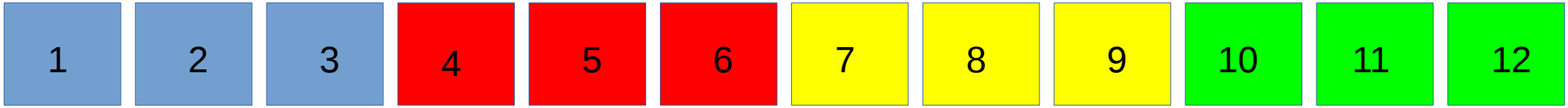
If there are several parallel processes, the system is defined by the slowest/the one with the longest workload



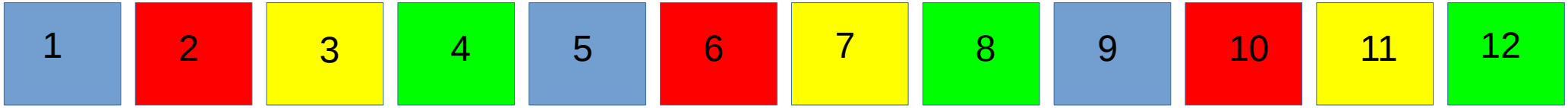
Problems of Parallelizing Codes (VIII)

In some cases it can be important how the different subtasks are distributed among the threads:

one thread

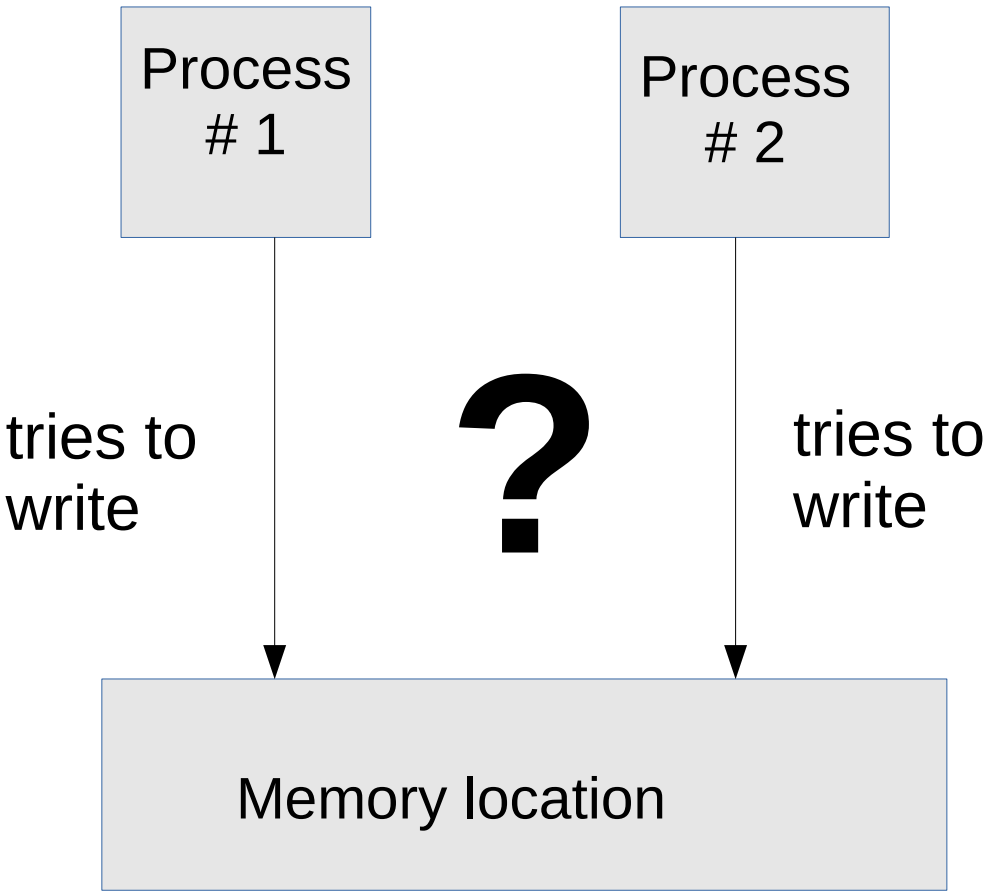


4 threads, chunksize=3



4 threads, chunksize=1

Problems of Parallelizing Codes (IX)
Race conditions



If two processes try to write to the same memory location at the same time, the behavior is unpredictible (but probably wrong) !

And it is likely that your program *silently* gives you wrong results without a warning !

Parallelization Techniques (I)

This list does by no means claim any completeness.

- **Automatic parallelization**
  - \+ In principle „just a compiler switch“
  - – Works only for simple loops without side effects

- **Threading libraries of the operating systems**
  - \+ Very flexible
  - – Communication between processes has to be implemented more or less „manually“
  - – Portability issues and issues with non-C-like programming languages (may require additional work not directly related to the physics problem one wants to solve).

Parallelization Techniques (II)

- **Threading functionality of the standard library of your programming language (C++-11 / Python / Java / ...)**

  + portable among different operating systems (as long as the corresponding programming language is available...)
  + flexible, ...

  – **...** but no semi-automatism, thus more work has to be done by yourself
  – usually limited to shared-memory systems

Parallelization techniques (III) - OpenMP

- **OpenMP**

  OpenMP is a semi-automatic approach. The basic idea is that the programmer declares which control structures should be parallelized and what dependencies exist between the data, i.e. must there a copy of each variable for each computing thread (~process) or has the same variable (= location in the memory) to be accessed by each thread.

  + Supported by basically all Fortran and C ++ Compilers (==> portable)
  + Programs can be written such that they use but not require OpenMP
  + rather easy to use (most difficult task: knowing the dependencies, which can not be avoided no matter what programming technique is used)
  + efficient for shared memory machines

Parallelization techniques (IV) - OpenMP

- **OpenMP (continued)**
  - not suitable for distributed memory systems (clusters)
  - not available for languages other than C/C++/Fortran

  ==> OpenMP-only programs will not work properly on „real“ supercomputers (OpenMP can however be combined with other techniques for parallelizing code)

Parallelization techniques (V) - MPI

- **MPI (Message Passing Interface)**
  Message passing system (data are explicitly sent between separate processes, i.e. not just threads of the same program [1])

  + flexible: can be used both on a single computer and distributed over several nodes of a cluster (appropriate solution for supercomputers)

  + available for each relevant system

  – complex programming (and usage)

1) „Threads“ usually means subprocesses of the same program. The program will appear as a single program (with a CPU usage of possibly more than 100 %) when displayed with `top` *or the windows task manager. Threads can directly access the same part of the memory, while different OpenMP processes usually can not do that.*

Parallelization techniques (VI): Coarray Fortran

- **Coarray Fortran**

  + Serveral tasks on a single computer or tasks distributed over a cluster
  + Transparent access to parts of the arrays that are primarily used by other tasks (Partitioned Global Address Space; PGAS)
  + Appears to be more easily used than MPI
  + Supported by the Fortran 2008 standard ==> likely to be future-proof (however, actual compiler support rather worse than OpenMP)
  – Fortran-only (A similar approach, UPC, exists for C, but it is not standardized)
  – Supported by `gfortran` only in the newest versions (For clusters, additional libraries are required)
  – For some implementations, the MPI libraries and infrastructure (`mpirun` ect.) have to be used
  – Programs will not work at all with compilers not supporting CAF

Parallelization Techniques (VII) - GPUs

- **GPU programming (or: accelerator cards)**

  + *In principle* considerable speedups can be reached with relatively inexpensive hardware (graphics cards/specialized GPU cards)
  − The performance gain crucially depends on the organization of the data (SIMD: *S*ingle *I*nstruction *M*ultiple *D*ata)
  − **Two** platforms have to be considered: host computer and accelerator card

  However: New approaches like OpenAAC (similar to OpenMP) try to semi-automatize the communication such that only a single program has to be written.

More details about OpenMP (I)

- OpenMP is available for C, C++ and Fortran (90)
- Although OpenMP is supported by most of the “major” compilers (Microsoft VC++, gcc, clang (==> Apple !), Intel, NAG, Pathscale, ect.), it is neither a part of the any C, C++, nor any FORTRAN standard. Therefore has to be explicitly activated.

```
gcc/g++/gfortran -fopenmp

ifort/icc -qopenmp     (Linux/OS X)

ifort/icc /Qopenmp     (Windows)

cl /openmp             (MS C/C++)
```

Thus, compilers that do not support or where it is switched off OpenMP, will ignore the directives and generate serial code.

More details about OpenMP (II)

- As mentioned above, OpenMP is available for C, C++ and Fortran (90).
- In Fortran, OpenMP directives are implemented by specially formatted comments.

```
...
!$OMP PARALLEL &
!$OMP PRIVATE (i)
!$OMP DO
 DO i=min, max
 ...
 END DO
!$OMP END DO
!$OMP END PARALLEL
...
```

Thus, compilers that do not support OpenMP, will ignore the directives and generate serial code.

More details about OpenMP (II)

- In C and C++ the `#pragma omp` preprocessor directive is used

```
...
#pragma omp parallel for  private (i)
 for (i=imin; i<=imax; i++)
 {
   ...
 }
...
```

Like in the Fortran case, these directives are also ignored by compilers that do not support the OpenMP extension or compilers where the OpenMP extension is switched of.

More details about OpenMP (III)

Additionally, there exist the Fortran 90 module **`OMP_lib`** and the C(++) include file **`omp.h`** that provides some additional soubroutines and functions. While it is not necessary for the basic functionality of OpenMP, they allow for a more flexible usage of OpenMP (e.g., querying the number of parallel threads or the id of the current thread, adjusting the number of threads or the scheduling behavior,...)

Example (Fortran):

```
...
use OMP_lib
...
write(*,*) OMP_get_thread_num()
...
```

Example (C++):

```
...
#include <omp.h>
...
cout<<omp_get_thread_num()<<endl;
...
```

More details about OpenMP (IV)

The run-time behavior of OpenMP can also be controlled by environment variables:

`OMP_NUM_THREADS`	:	Number of parallel processes
`OMP_SCHEDULE`	:	Controls the balancing
`OMP_DYNAMIC`	:	Can the number of threads be dynamically adjusted ?
`OMP_NESTED`	:	Are nested !$OMP directives legal ?

Open MP - Demos (I)

The `openmp_demos.zip` archive contains several example programs that demonstrate key aspects of OpenMP and are available both in a C++ and a Fortran 2003 version.

- `hello_parallel_c.cxx / hello_parallel_f.f90`

  demostrate the concept of the “omp prallel” statement and how to avoid race conditions

- `parallel_primes_c.cxx / parallel_primes_f.f90`

  compute the number of primes up to a certain upper limit
  The performance will be shown in the next slides.

- `energy density_c.cxx / energy_density_f.f90`

  demonstrate the effects of different parallelization strategies in the case of nested loops

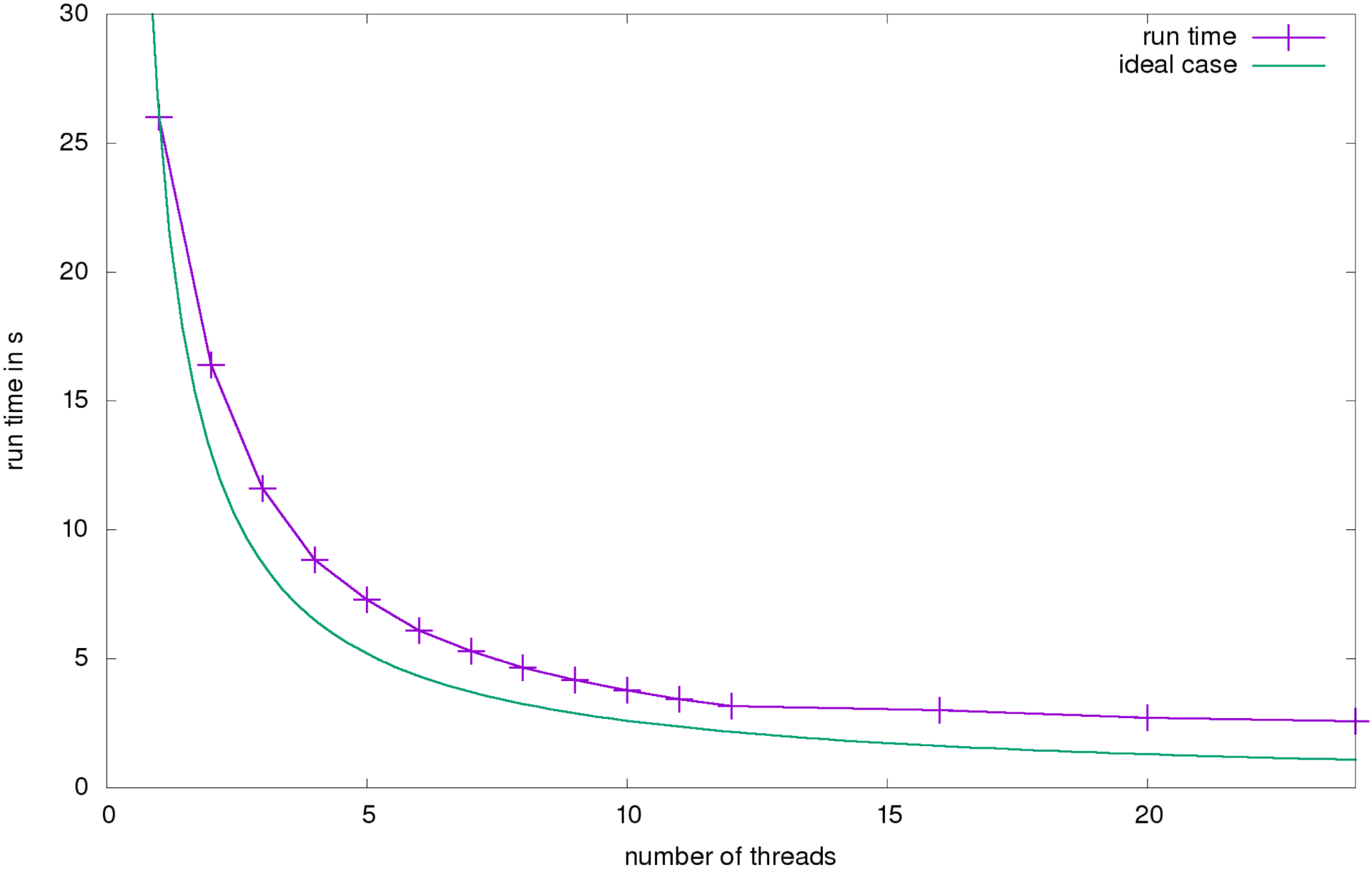
Open MP - Demos (II)

Unzip the file and enter the unzipped directory.
Then use `make` to build the program. By default, the make file uses the GCC compilers. If you want to use other compilers, modify the variables in the `Makefile` accordingly.

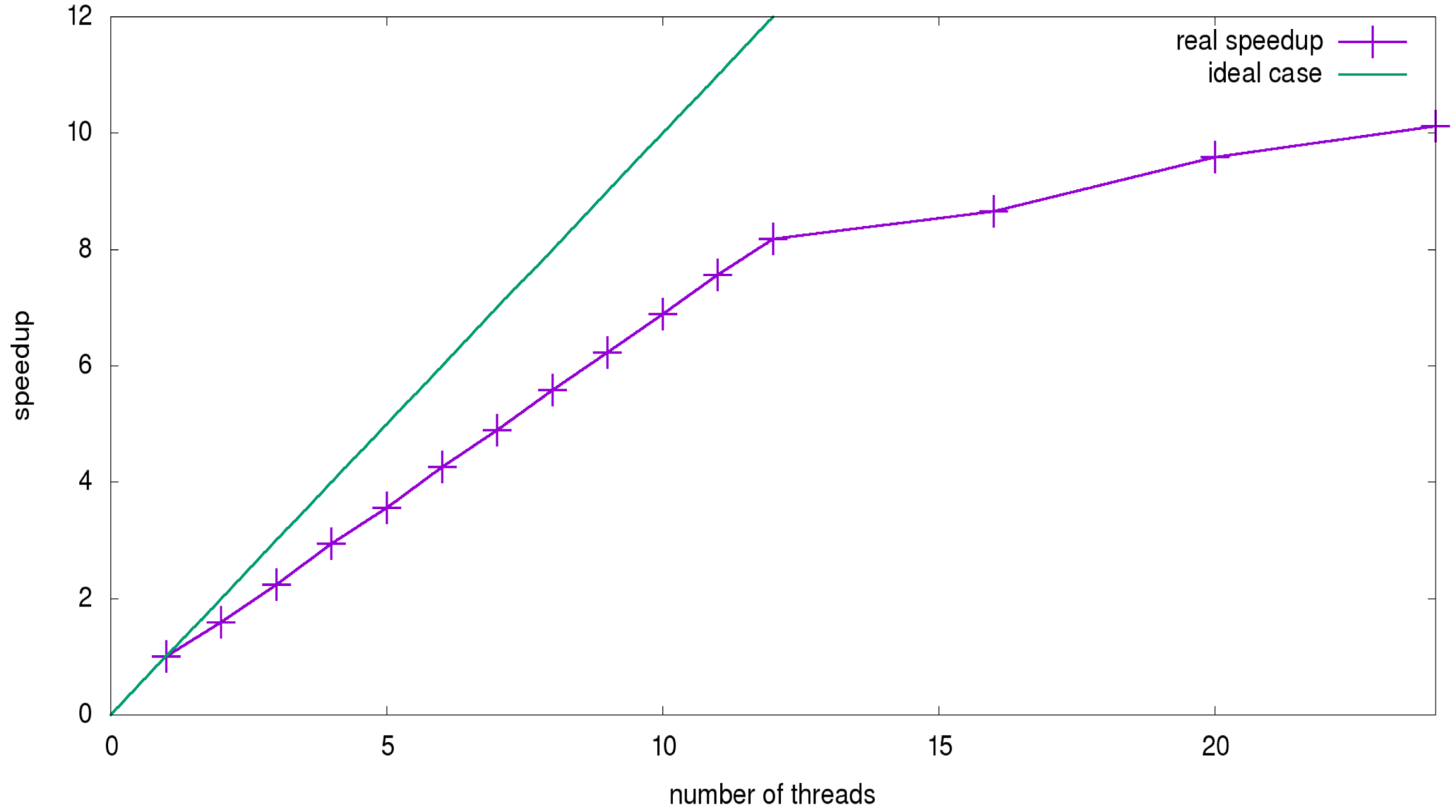
The following make tragets are available:

- `make` ==> only C++ files are compiled (default case)
- `make cxx` ==> only C++ files are compiled
- `make fortran` ==> only Fortran files are compiled
- `make all` ==> both C++ and Fortran files are made
- `make clean` ==> deletes the executables (and Fortran module files)

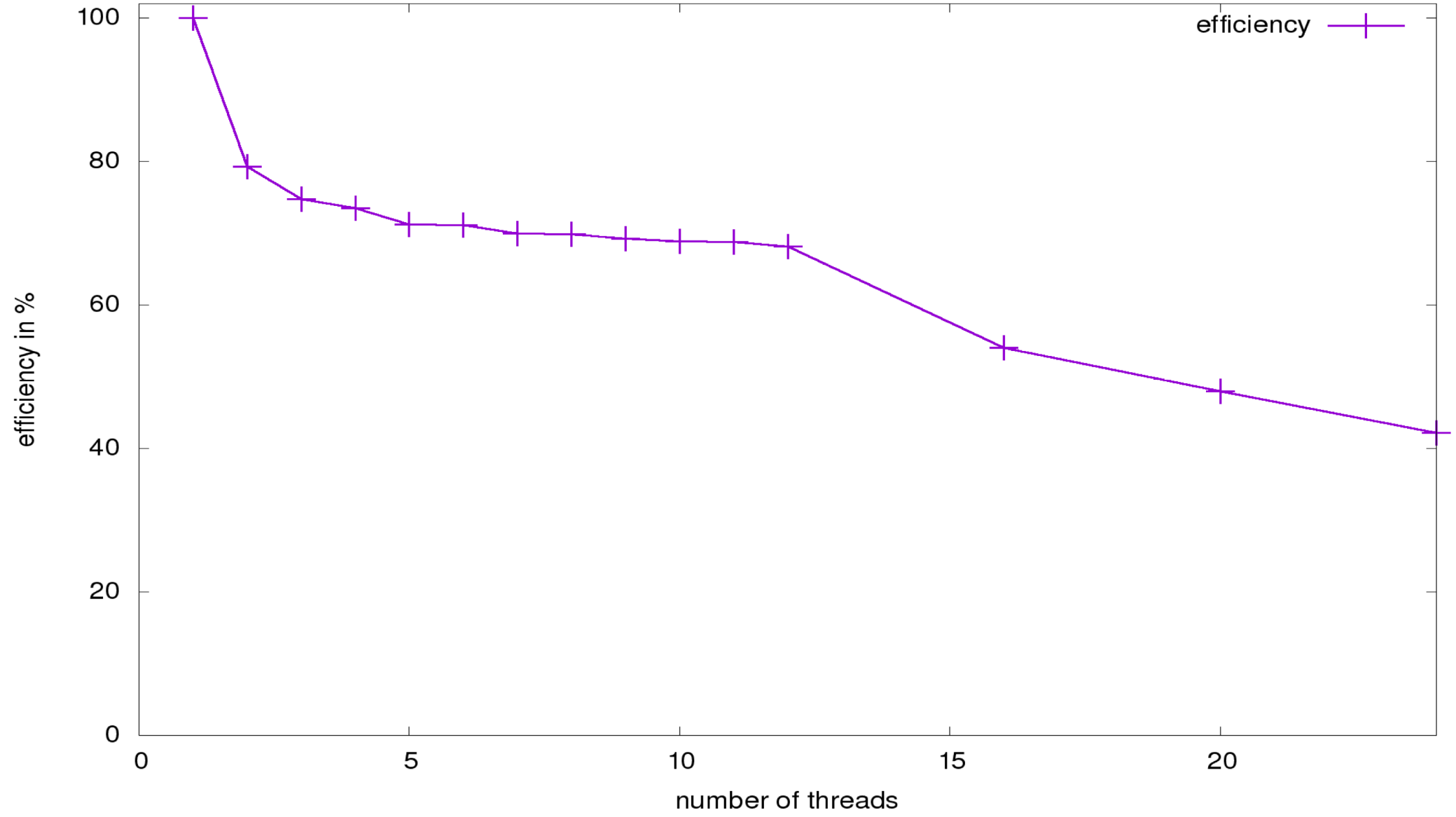
Results - Execution Time (Determine Number of Primes below 25 000 000)



Results - Speedup (Determine Number of Primes below 25 000 000)



Results - Efficiency (Determine Number of Primes below 25 000 000)



Literature

- https://www.openmp.org
  Homepage of the OpenMP ARB (Architecture Review Board), the group that realeases the OpenMP standards
  The website links to formal description of the standards as well as to more basic introductotry material

- The OpenMP ARB also provides a youtube channel with talks:
  https://www.youtube.com/user/OPENMPARB

- Simon Hoffmann, Rainer Lienhart: *OpenMP (Informatik im Fokus)*
  A compactGerman introduction to OpenMP (C++ only)

Used images

- Page 2 (upper left)
  https://de.wikipedia.org/wiki/Smartphone#/media/File:WinWikiStart.PNG
  created by Wikimedia user D. Reust (public domain)
- page 2 (upper right)
  https://de.wikipedia.org/wiki/Laptop#/media/File:IBM_Thinkpad_R51.jpg
  © Wikimedia user André Karwith; license CC BY_SA 3.0
- page 2 (lower left)
  https://en.wikipedia.org/wiki/Workstation#/media/File:HP_Z820_Workstation.jpg
  © Wikimedia user Vernon Chan; license CC BY 2.0
- page 2 (lower right)
  https://de.wikipedia.org/wiki/SuperMUC#/media/File:SuperMUC.jpg
  © Wikimedia user Mdw77; license CC BY-SA 4.0
- page 7
  https://de.wikipedia.org/wiki/Gene_Amdahl#/media/File:Amdahl_march_13_2008.jpg
- © Wikimedia user Pkivolowitz; license CC BY 3.0

  Links to Creative Commons Licenses: