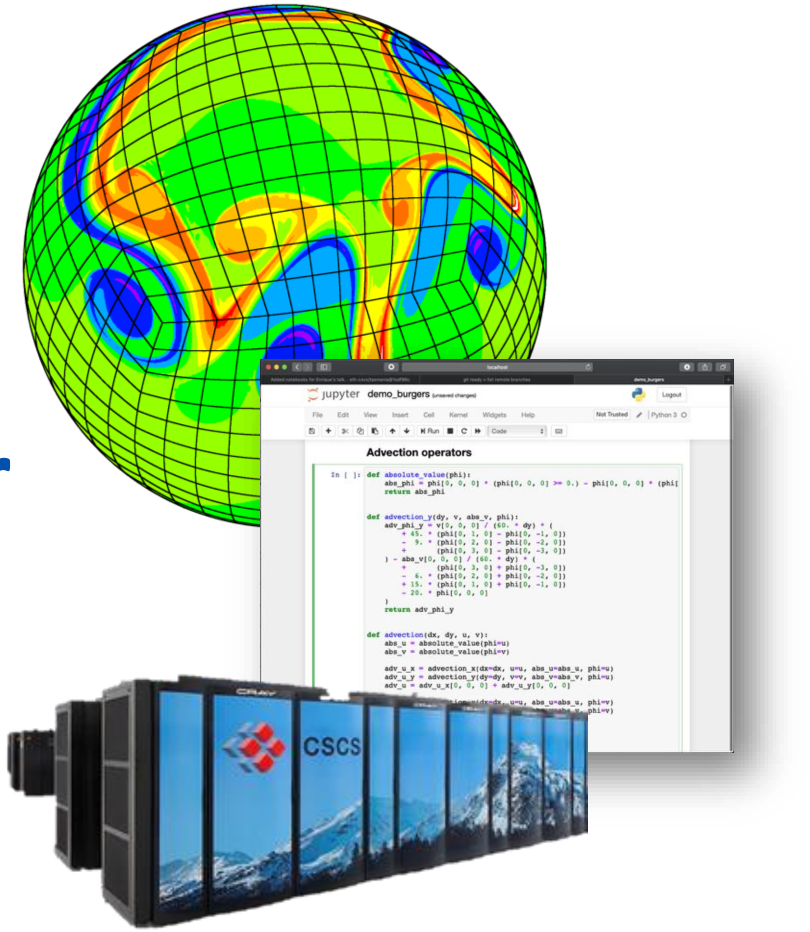
High Performance Computing for Weather and Climate (HPC4WC)

Content: Shared Memory Parallelism

Lecturer: Oliver Fuhrer

Block course 701-1270-00L

Summer 2025

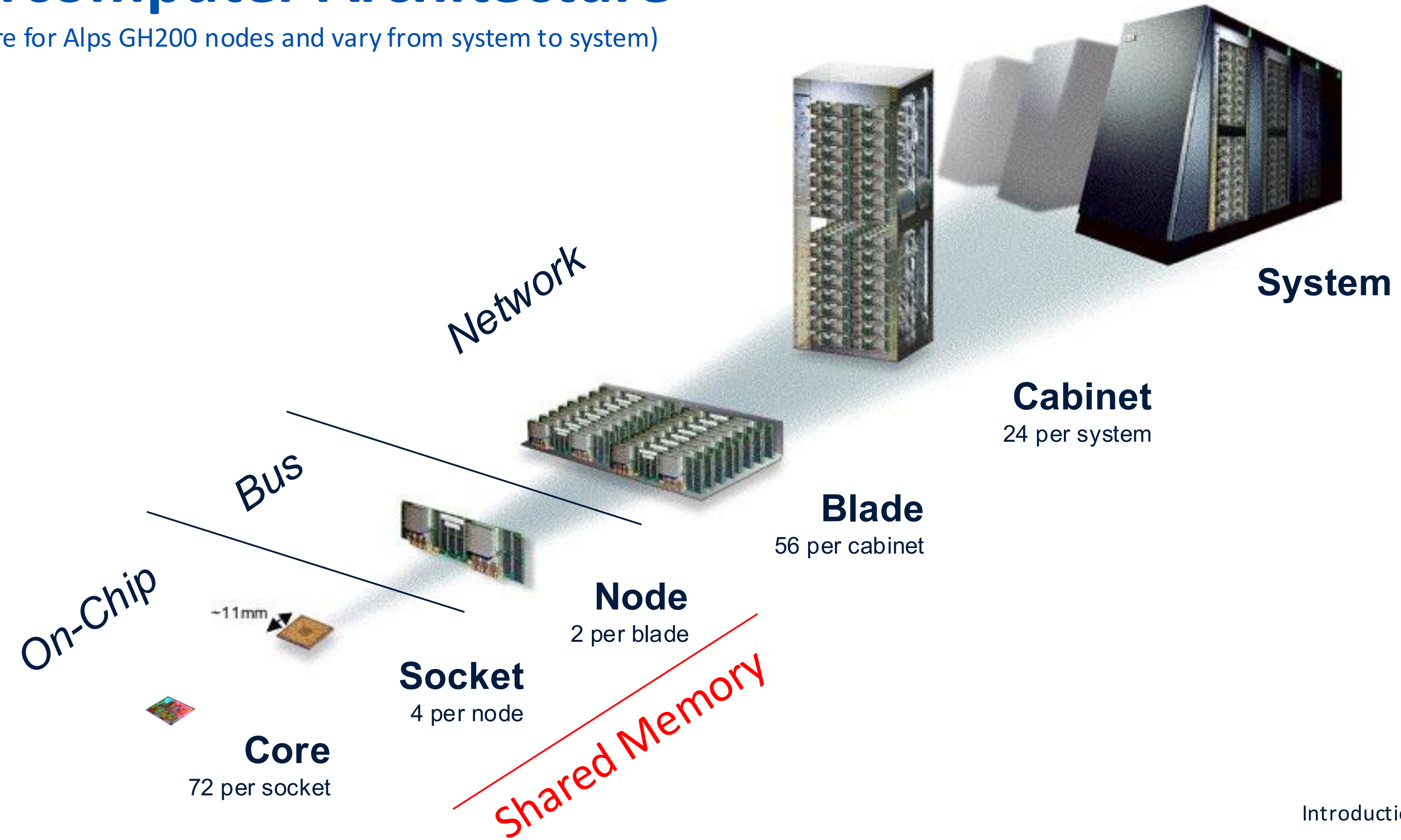


Learning goals

- Understand shared memory parallelism and the OpenMP programming model
- Understand some limitations of parallelism with Amdahl's law
- Know about common pitfalls in shared memory computing

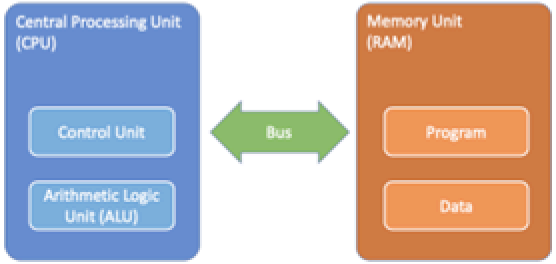
Supercomputer Architecture

(Numbers are for Alps GH200 nodes and vary from system to system)

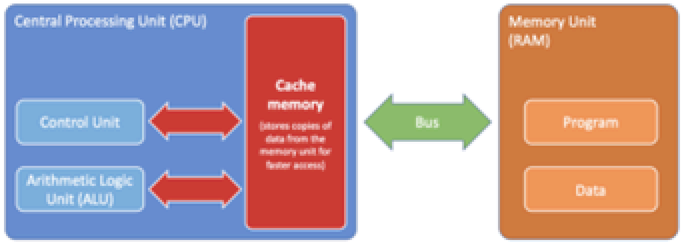


Node Architecture

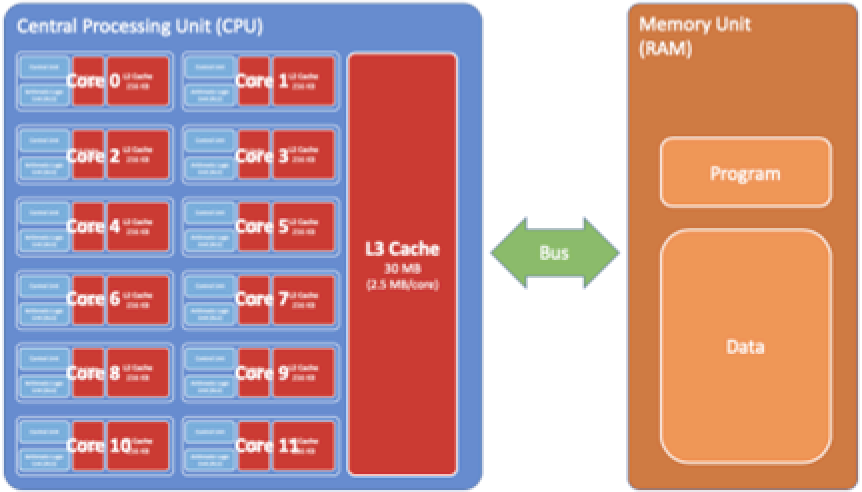
Von Neumann



Cache hierarchy



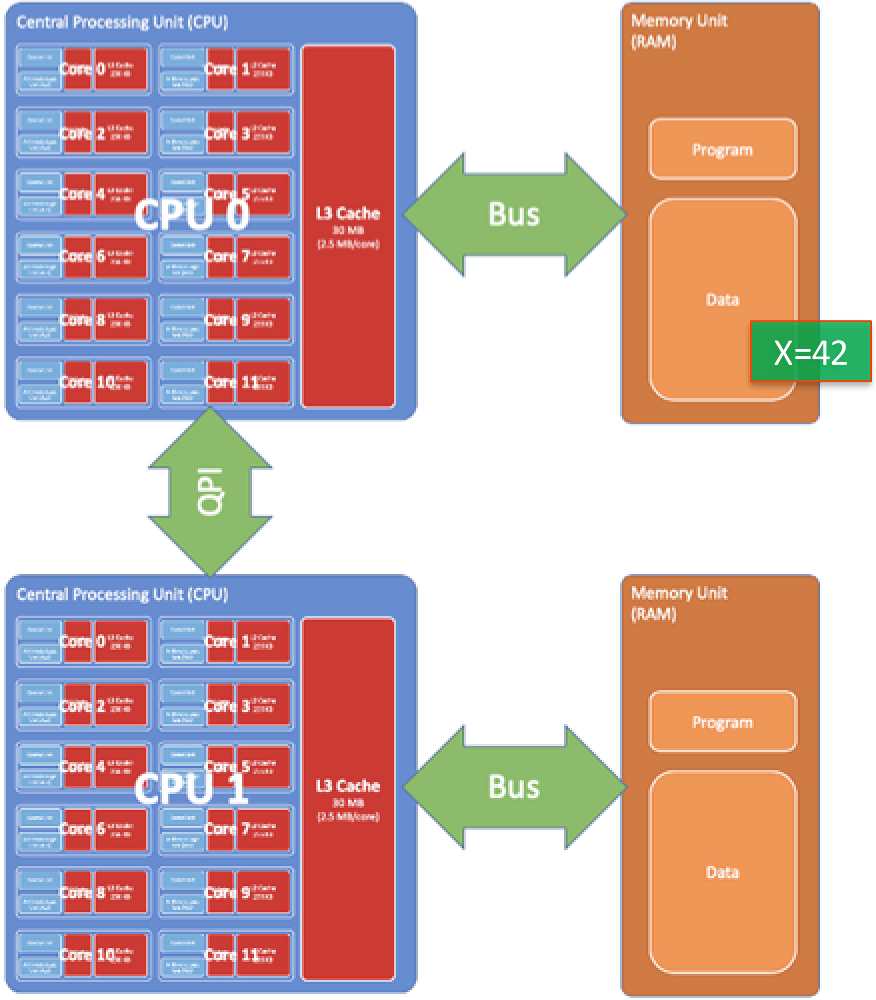
Multicore CPU



Node Architecture

Share memory node
- Multiple CPUs (1, 2, 4, …)
- Connected via Bus (NVLink, QPI)
- Many cores (12, 24, 36, …)
- Multiple memories
- Shared address space (data in any memory is accessible to any core)

To make efficient use of the resources, a program must run in parallel on multiple cores.



OpenMP

- Open Multi-Processing is an API that supports shared-memory multiprocessing (https://www.openmp.org/)
- Version 1.0 in 1997, latest Version 6.0 in 2025
- Support for Fortran, C, C++
- Common programming model used in HPC
- Section of code that should run in parallel is marked with a compiler directive (if ignore, legal sequential code)
- Reference sheet of Fortran API v4.0

```fortran
program hello_world
    use omp_lib
    implicit none

    !$omp parallel
    write(*,*) 'Hello, world.'
    !$omp end parallel

end program hello_world
```

```
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
Hello, world.
```

Compiler directives (`!$omp`)

Pros

- ease of use
- incremental adoption
- portable across platforms & compilers

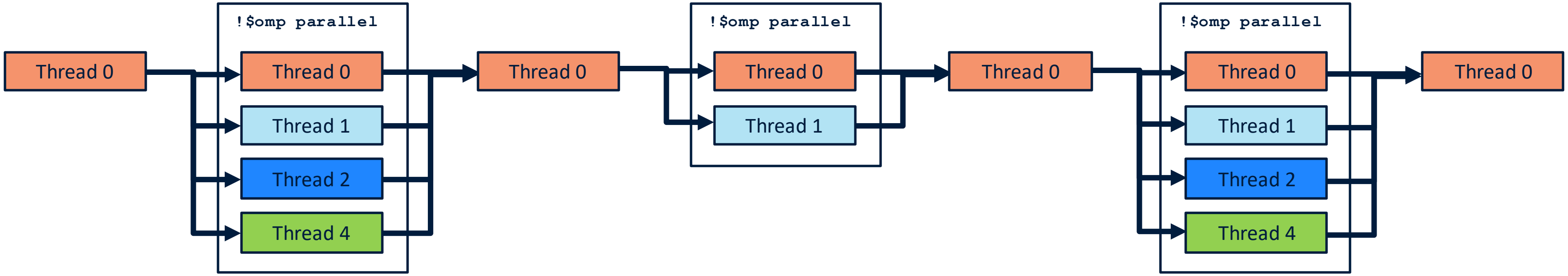
Cons

- maintenance overhead
- not safe
- hard to debug
- varying compiler support
- scalability limits

```fortran
program main
    use omp_lib
    implicit none

    integer :: i, size, rank

    !$omp parallel num_threads(3) private(size, rank, i)
    size = omp_get_num_threads()
    rank = omp_get_thread_num()

    !$omp do
    do i = 0, 5
        write(*,*) 'loop 1, iteration ', i
    end do
    !$omp end do

    !$omp do
    do i = 0, 5
        write(*,*) 'loop 2, iteration ', i
    end do
    !$omp end do

    !$omp end parallel

end program main
```

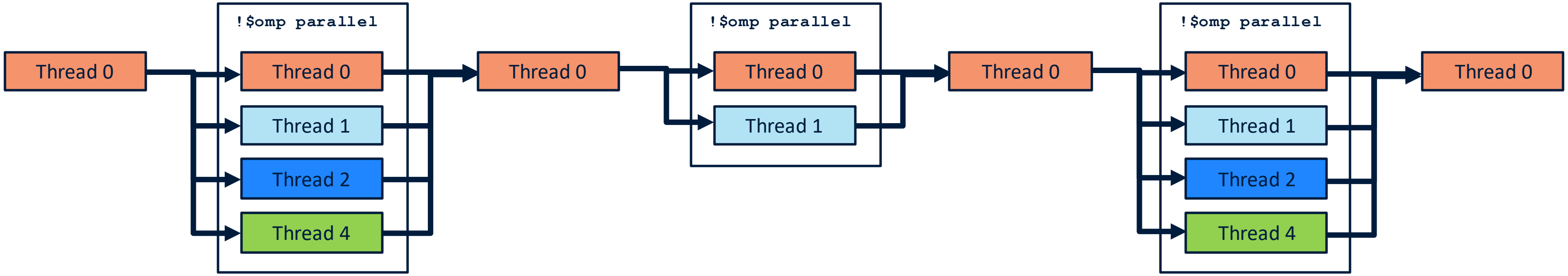
The fork-join model

- One main thread that runs through the full programm
- Parallel regions that can fork multiple threads that can execute code in parallel



Who am I? How many are there?

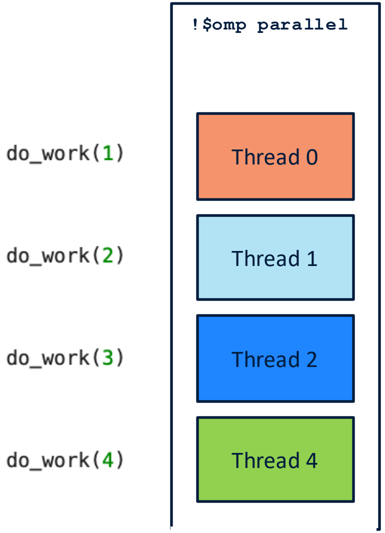
- Each thread has a unique number (thead ID)
- The master thread is always number 0
- Possible to query thread ID and number of threads



Parallel loops

- Typically where the work is happening and potential parallelism is present

```fortran
program main
    use omp_lib
    implicit none

    integer :: i, myvar
    myvar = -1

    do i = 0, 9
        call do_work(i)
    end do

end program main
```

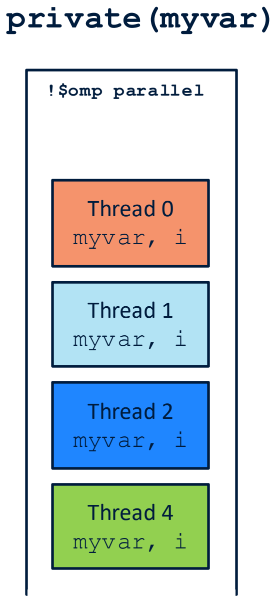


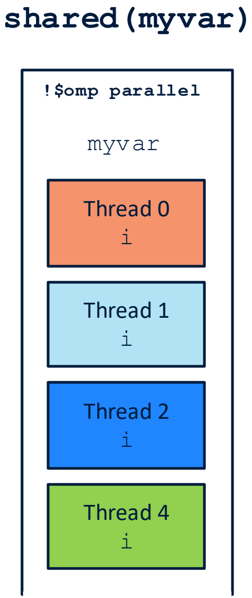

```fortran
program main
    use omp_lib
    implicit none

    integer :: i, myvar
    myvar = -1

    !$omp parallel do
    do i = 0, 9
        call do_work(i)
    end do
    !$omp end parallel do

end program main
```

Variable scoping

- Defines whether variables are shared among threads or private to threads

```
program main
    use omp_lib
    implicit none

    integer :: i, myvar
    myvar = -1

    !$omp parallel do
    do i = 0, 9
        myvar = i
    end do
    !$omp end parallel do

end program main
```





Variable Scoping

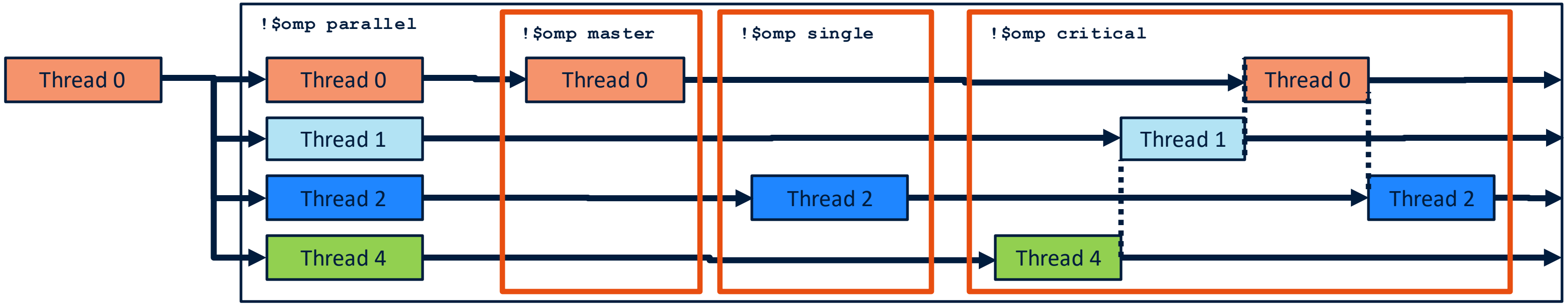
- Each **private** variable is not initialized at the start of the parallel region
- Each thread owns it's own copy of the private variable

- Each **shared** variable is shared amongst all threads and is copied in
- Each thread can write to shared variables at any point (no safety)

- Each **firstprivate** variable is copied in from the sequential code
- Each thread owns it's own copy of the private variable

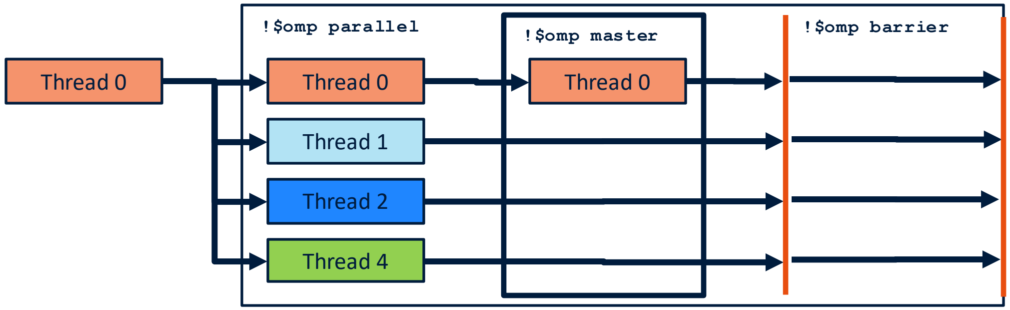
Special regions (master, single, critical)

- Within a parallel region, special constructs control which threads and when threads enter a specific code section



Synchronization

- Tasks wait for each other at the end of a parallel region
- Explicit !$omp barrier to synchronize threads
- Explicit `nowait` at end of parallel region to avoid synchronization



Demo: Calculating Pi

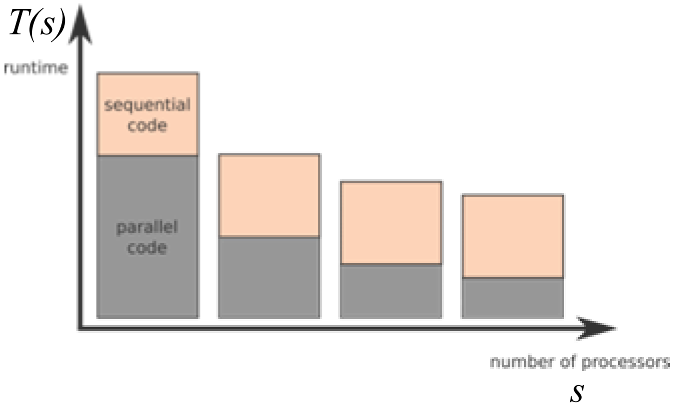
$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots = \frac{\pi}{4}$$

```fortran
program main
    use omp_lib
    implicit none

    integer :: steps = 10000000
    integer :: t
    double precision :: sum

    sum = 0.0

    do t = 0, steps - 1
        sum = sum + (1.0d0 - 2.0d0 * mod(t, 2)) / (2 * t + 1)
    end do

    write(*,*) 'pi = ', sum

end program main
```

Amdahl's law

p = Fraction of the program which is parallel

$$T(s) = (1-p)T_1 + \frac{p}{s}T_1$$



Sequential part of the code

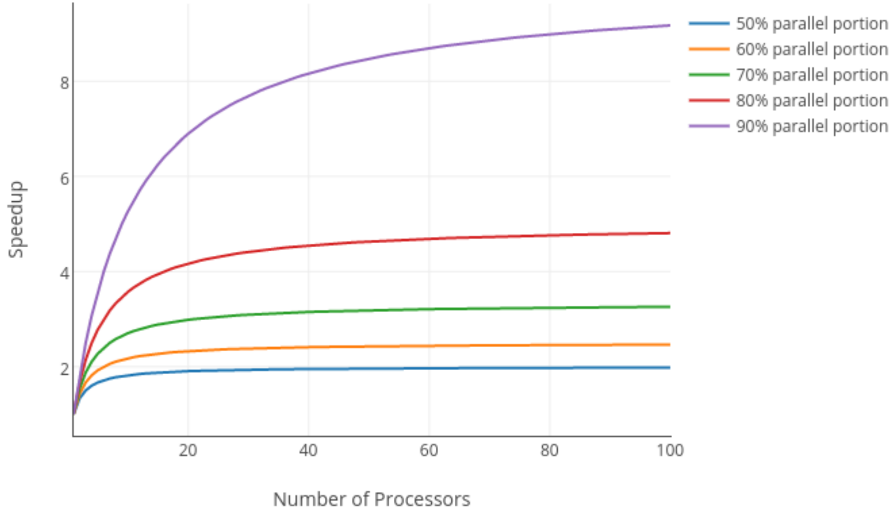
- Startup
- Initializing sum = 0

Amdahl's Law

- What speedup can we expect from parallel execution?

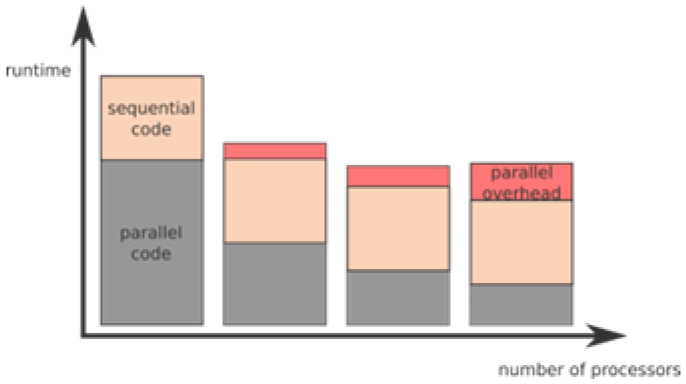
$$S(s,p) = \frac{T(s,1)}{T(s,p)}$$

$$= \frac{1}{(1-p) + \frac{p}{s}}$$



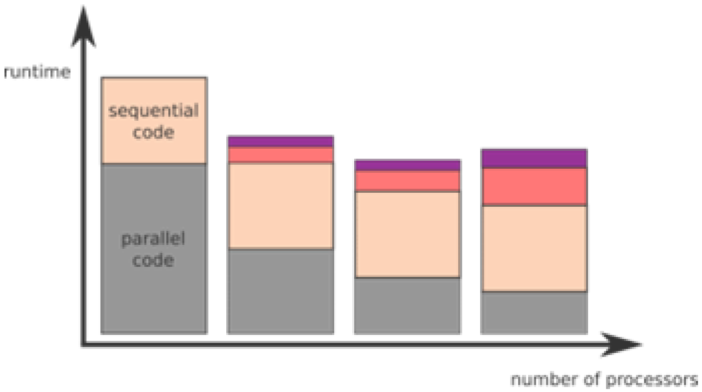
Amdahl's law is an optimistic upper bound on speedup by parallelization

Real world effects degrade speedup…



Parallel overhead
- Generating threads
- Scheduling work to threads

Real world effects degrade speedup…



Load imbalance

- waiting for the other worker to finish writing
- waiting for the other worker to finish their task

How do we measure speed

Given my problem size, how much faster does it get by increasing the number of workers?

- How good is the ratio of parallel to sequential fraction?
- How big is our overhead?

How do we measure speed

Given my problem size, how much faster does it get by increasing the number of workers?

- How good is the ratio of parallel to sequential fraction?
- How big is our overhead?

Given my target time, how big can I make my problem by increasing the number of workers?

- How large can I make my application such that the execution time stays similar?

Lab Exercises

01-OpenMP-introduction.ipynb or **01-OpenMP-introduction-Cpp.ipynb**

- Learn the basic OpenMP concepts (from lecture)

02-OpenMP-exercises.ipynb

- Parallelize the stencil2d program in Fortran using OpenMP
- Perform basic data-locality optimizations (fusion, inlining)
- Use a performance using a profiling tool for analysis and guidance

03-OpenMP-concepts_bonus.ipynb

- Learn more advanced OpenMP concepts (in C++) Bonus

Note: Take a look at the OpenMP-Fortran-Cheatsheet.pdf to get help for how to use OpenMP in Fortran!