High Performance Computing for Weather and Climate (HPC4WC)

Content: Shared Memory Parallelism
Lecturer: Tobias Wicky
Block course 701-1270-00L
Summer 2022

Administrative Things

- git pull in the morning
  - conflicts might occur if you already worked on future notebboks.
  - stash / apply stash will help, otherwise just copy files. We can help

- solutions for the previous day in the solutions folder

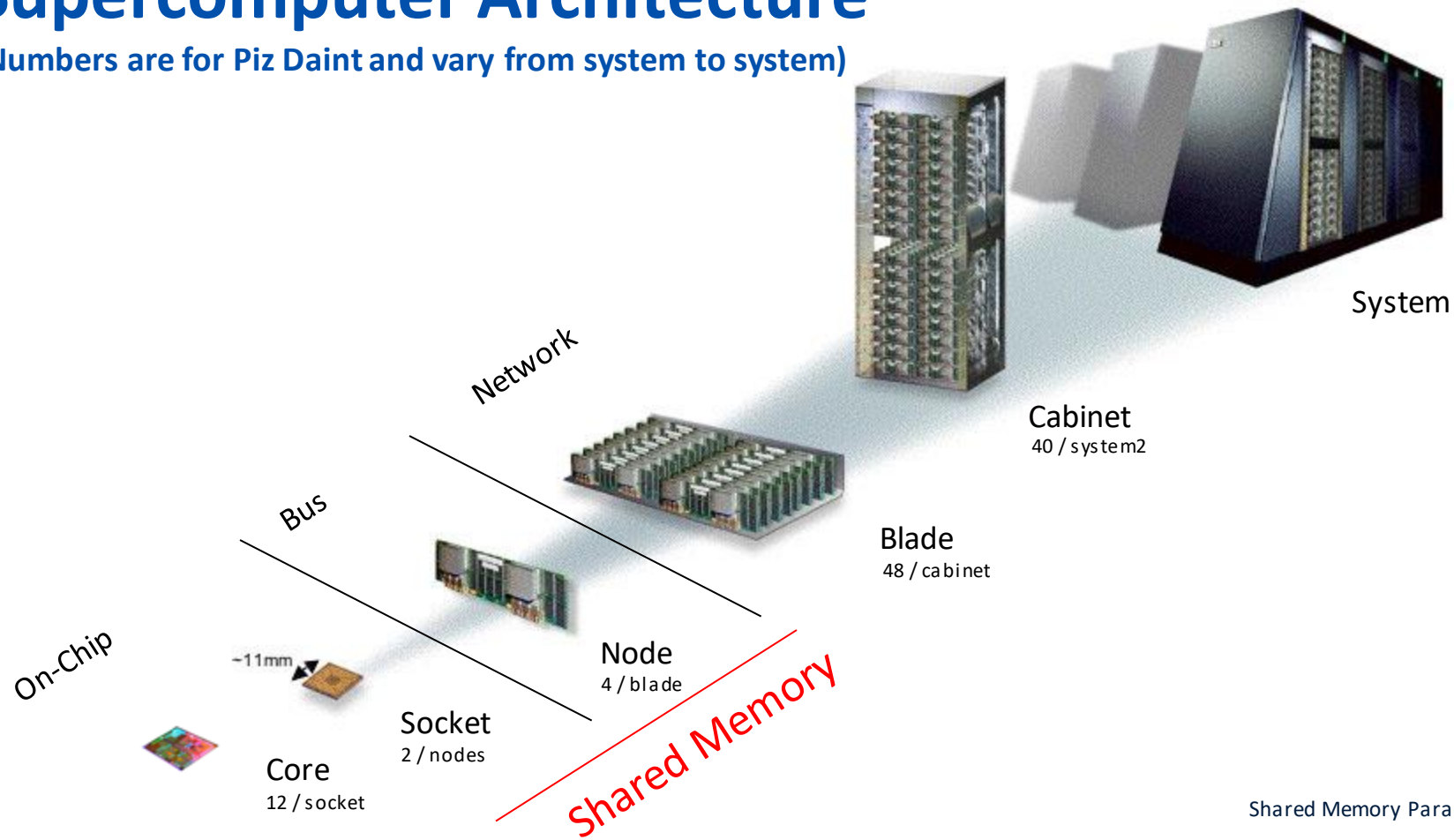
Learning goals

- 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

Natural way of parallelizing

Supercomputer Architecture

(Numbers are for Piz Daint and vary from system to system)



What does that mean for us?

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 5.0 in 2018
- 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

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

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

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

Compiler directives

Pros

- semi automatic parallelisation
- portable across platforms & compilers

Cons

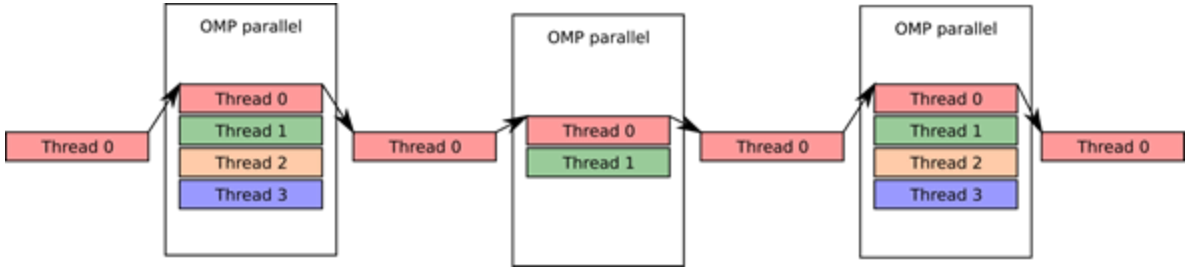
- non optimal code
- not the same for each compiler
- not safe

```
void spawnThreads(int n) {
  std::vector<thread> threads(n);
  for(int i = 0; i < n; i++) {
    threads[i] = thread(doSomething, i + 1);
  }

  for(auto& th : threads) {
    th.join();
  }
}
```

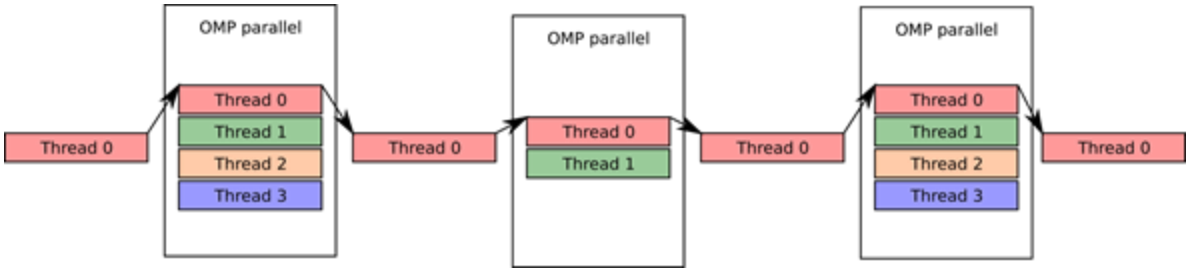
```
omp_set_num_threads(n)
#pragma omp parallel
  { do_something(omp_get_thread_num()); }
```

The fork-join model



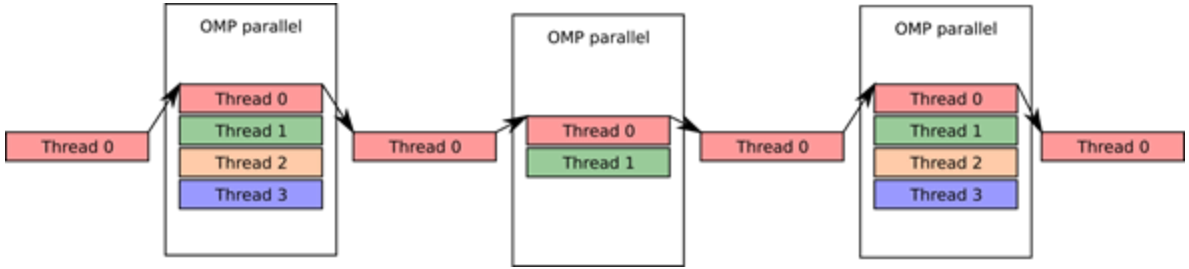
- One master thread that runs through the full programm
- Parallel regions that can fork multiple threads that can execute code in parallel

Infos About the Region



For some codes it is important to know which thread I am and how many are available

Setting the Number of Threads



- Why is it 24 right now?
- What if we want to be flexible?

Parallel Loops

- Core concept for a lot of HPC applications
- Separate pragma that helps handle some loop specific details

Scheduling

- Since performance can be sentitive to which process executes which part of the loop, there is a way to control this

static [, X]	Before running anything, each iteration is assigned to a thread. Each thread gets X consecutive iterations
dynamic [, X]	internal work queue, N/X chunks, first come first serve
guided [, X]	internal work queue, chunks of size of at least X, first come first serve

Variable Scoping

- Who owns the variables?
- How do they look in the parallel region?
- Who can write into which part of the memory

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 **public** 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

pragma omp master	only the thread with number 0 executes this
pragma omp single	only one thread executes this region, it is unknown which one though
pragma omp critical (NAME)	each thread will execute the section, but there is only ever one thread active in each section NAME
pragma omp atomic [type]	a fast version of critical that only allows specific operations

What if parallel regions span multiple tasks

- Tasks need a way to wait for each other
- omp parallel has an implicit barrier in the end

Nowait

- for loops have a barrier in the end
- nowait can parallelize tasks

Reductions

- Common pattern, has a special implementation

Demo: Calculating Pi

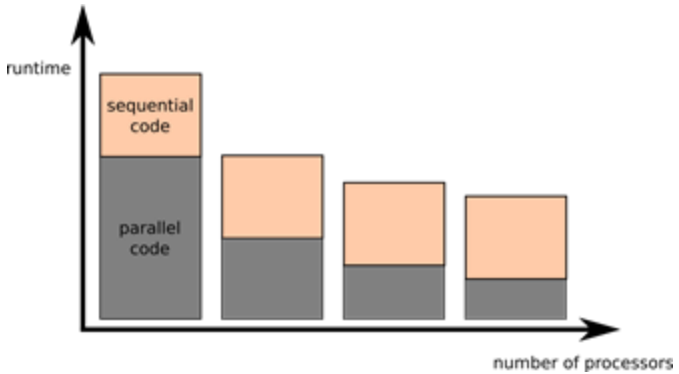
$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \ldots = \frac{\pi}{4}$$

- Assume we have 12 cores, how fast do we expect it to be?

How do we expect speed to work out here?

Performance of a parallel program

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



Sequential part of the code

- Opening the notebook
- Filling in the name of the group

How do we measure speed

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

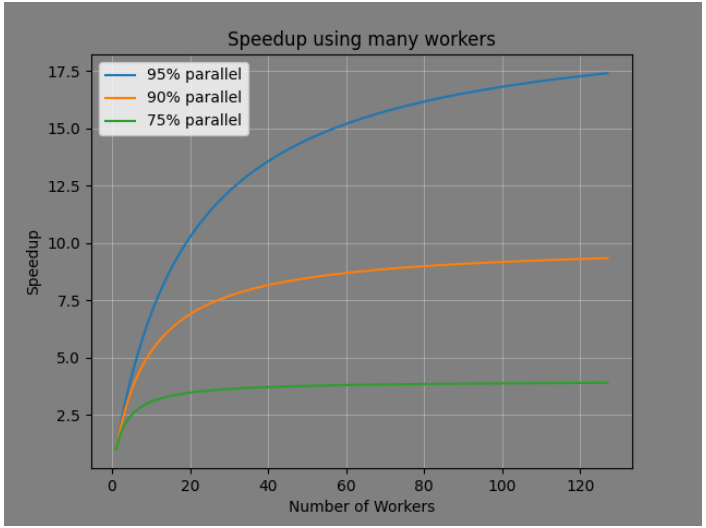
- How good is the ratio of parallel to sequential fraction?
- How big is our overhead?

Amdahl‘s Law

- Fixed Problem Size
- How much faster does our program get?

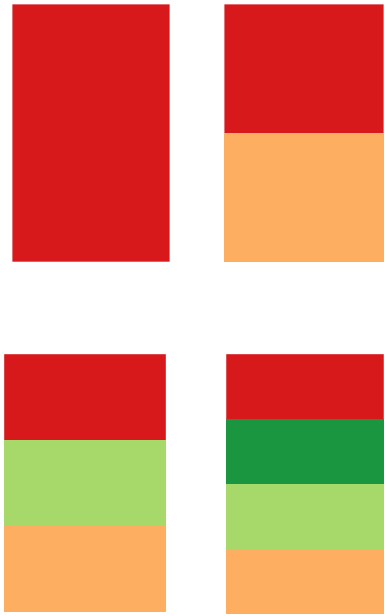
$$S(s,p) = \frac{T(s,1)}{T(s,p)}$$

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

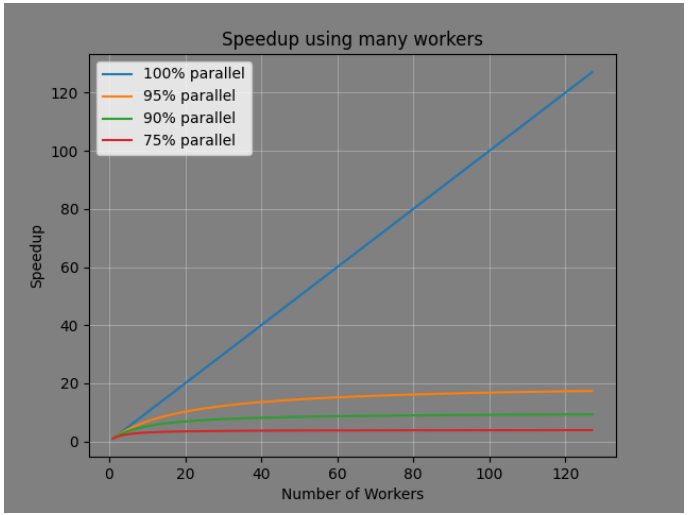


How do we measure speed

Strong Scaling: Keep the problem size the same and see how much faster we get



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



Why do we build large machines then?

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?

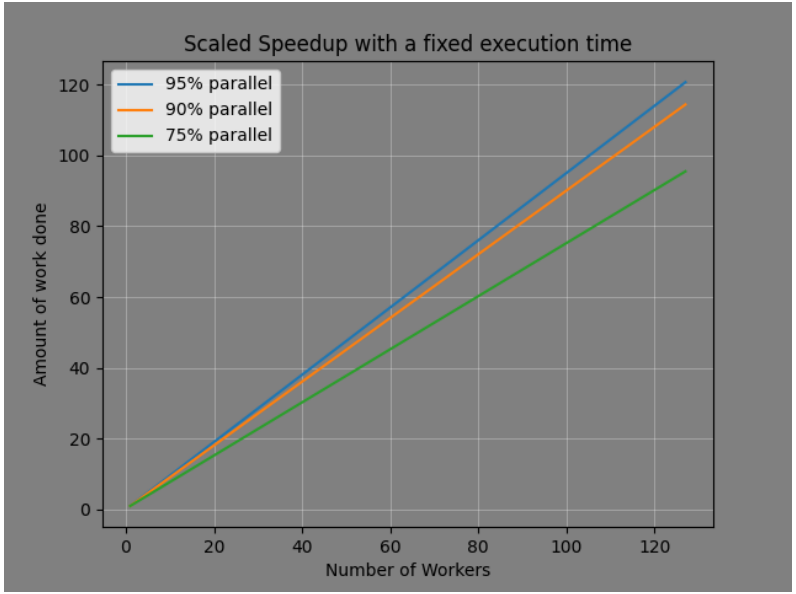
Given my target time, how big can I make my problem by increasing the number of workers?

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

Gustafson‘s Law

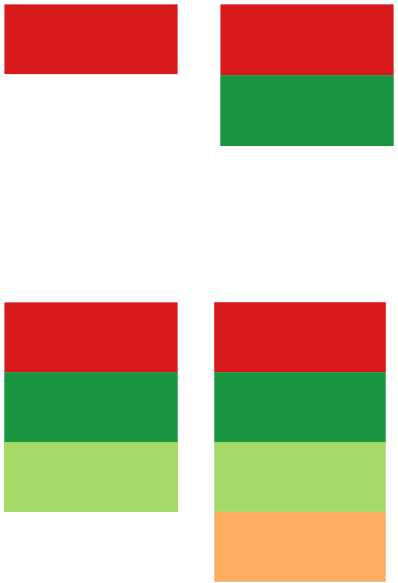
$$W(s,p) = (1-p)W + spW$$

- Fixed Execution Time
- How much bigger can our program become?

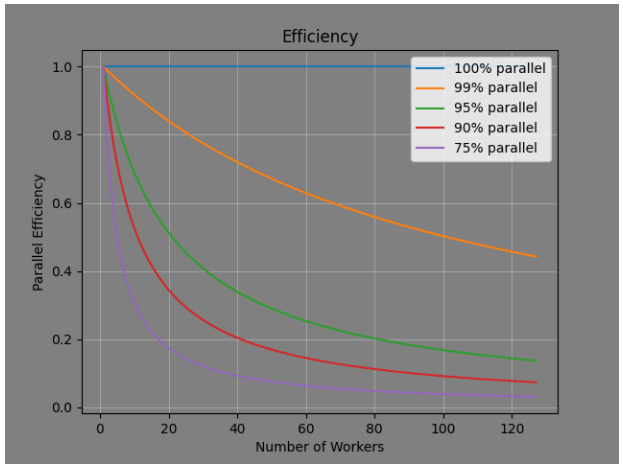


How do we measure speed

Weak Scaling: Keep the problem size *per processor* the same and see how much we lose



$$E(p) = \frac{T(1)}{T(p)}$$



What might lead to bad performance

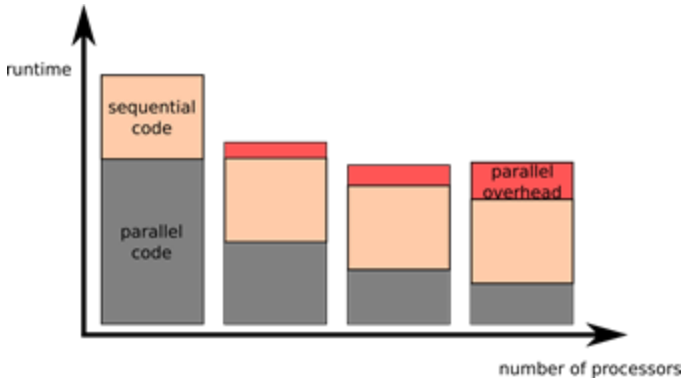
- Sequential parts of the code
  - Barriers
  - Reductions

What might lead to bad performance

- Sequential parts of the code
- Parallel overhead
- Load imbalance

Performance of a parallel program

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

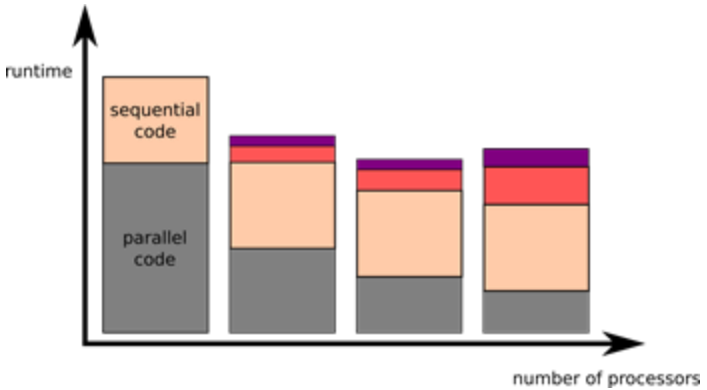


Parallel overhead

- making sure the order is correct
- synchronizing who does what

Performance of a parallel program

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



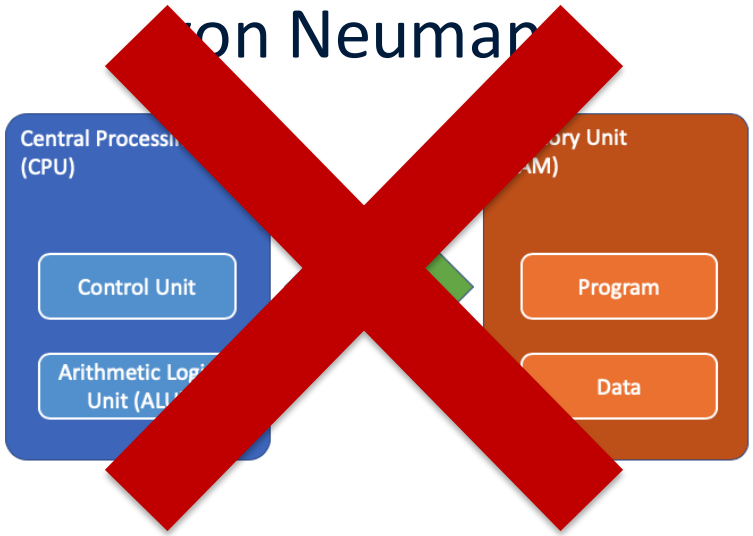
Load imbalance

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

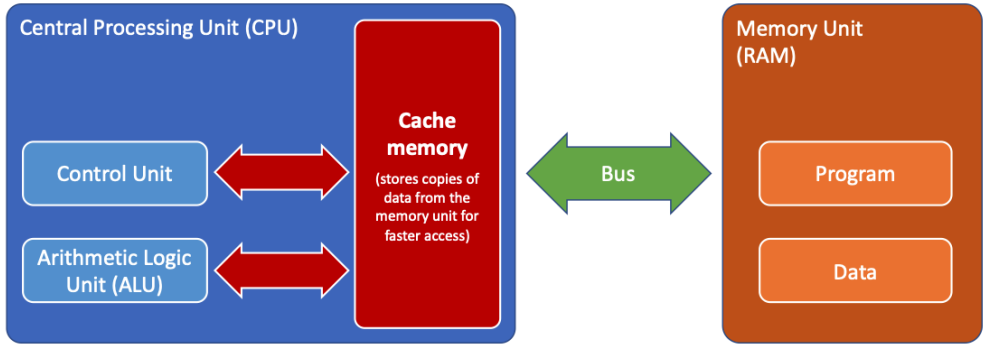
What might lead to bad performance

- Sequential parts of the code
- Parallel overhead
- Load imbalance
- Caching issues

Node Architecture

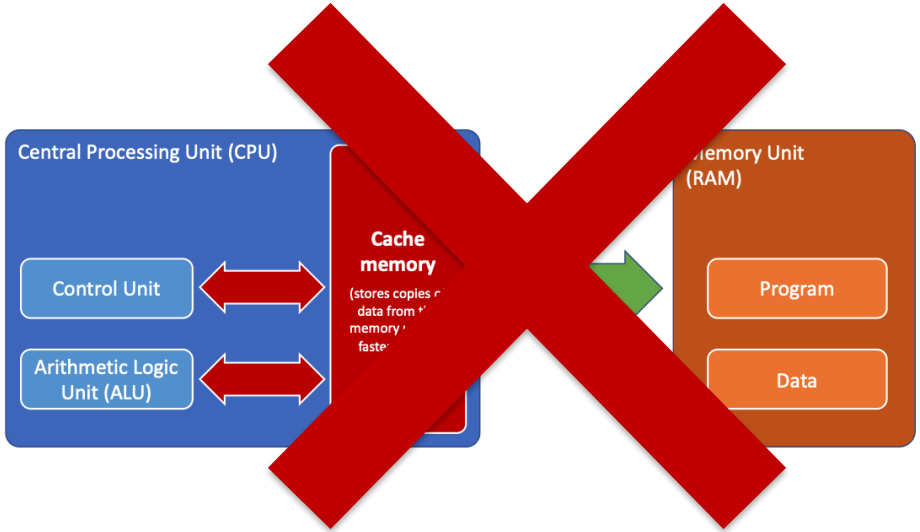


cache hierarchy

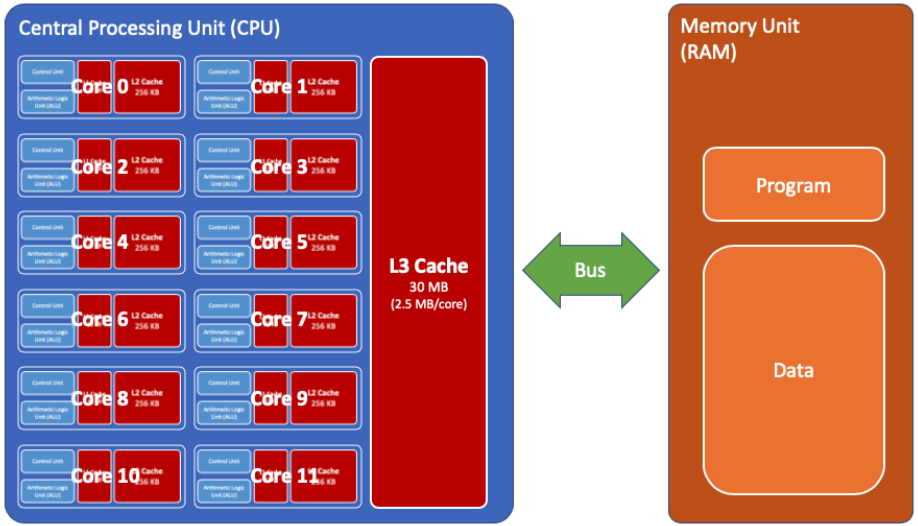


Node Architecture

cache hierarchy



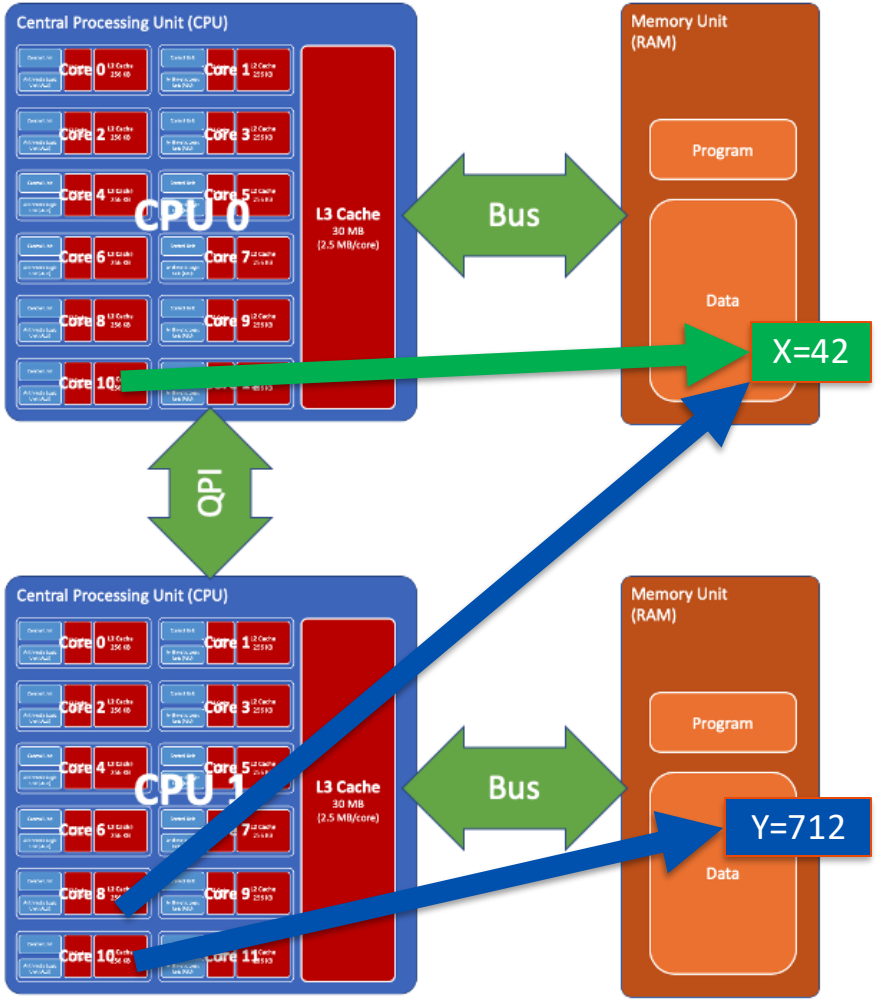
multicore CPU



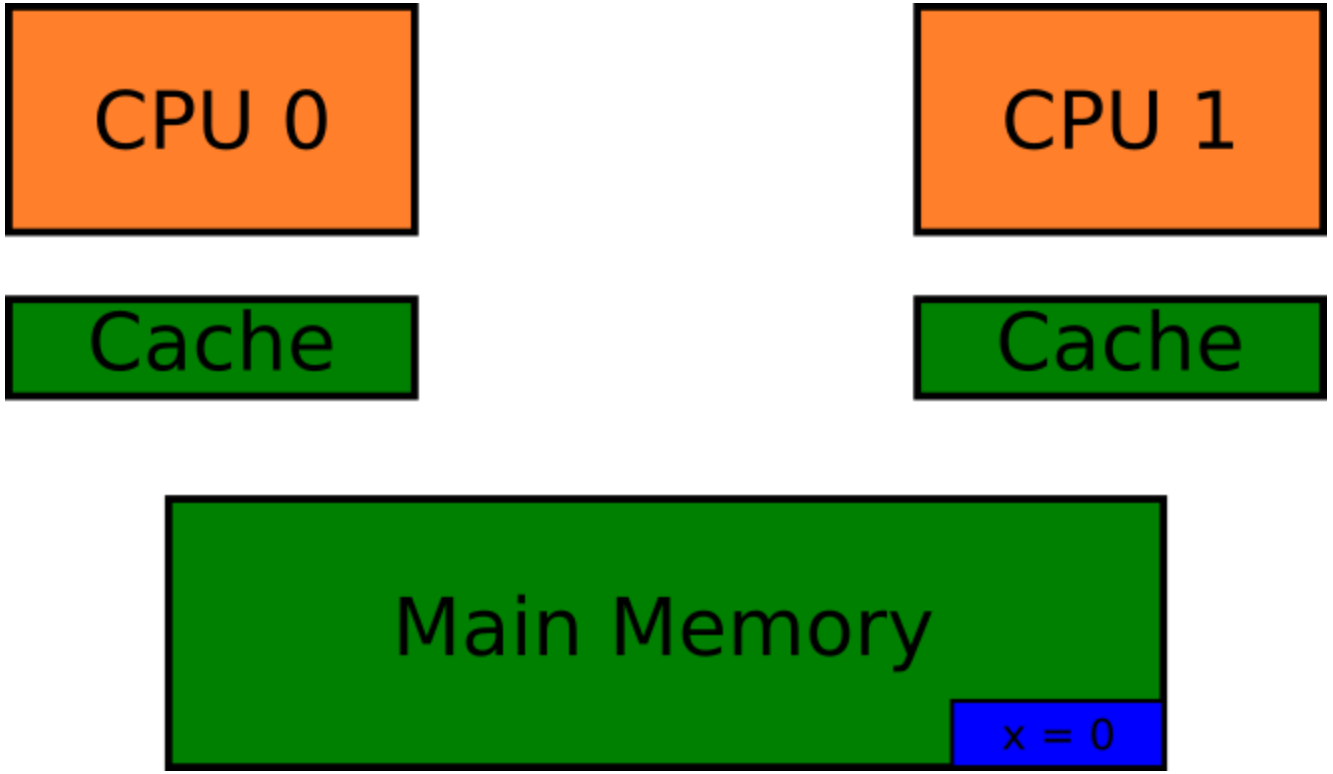
Node Architecture

Share memory node

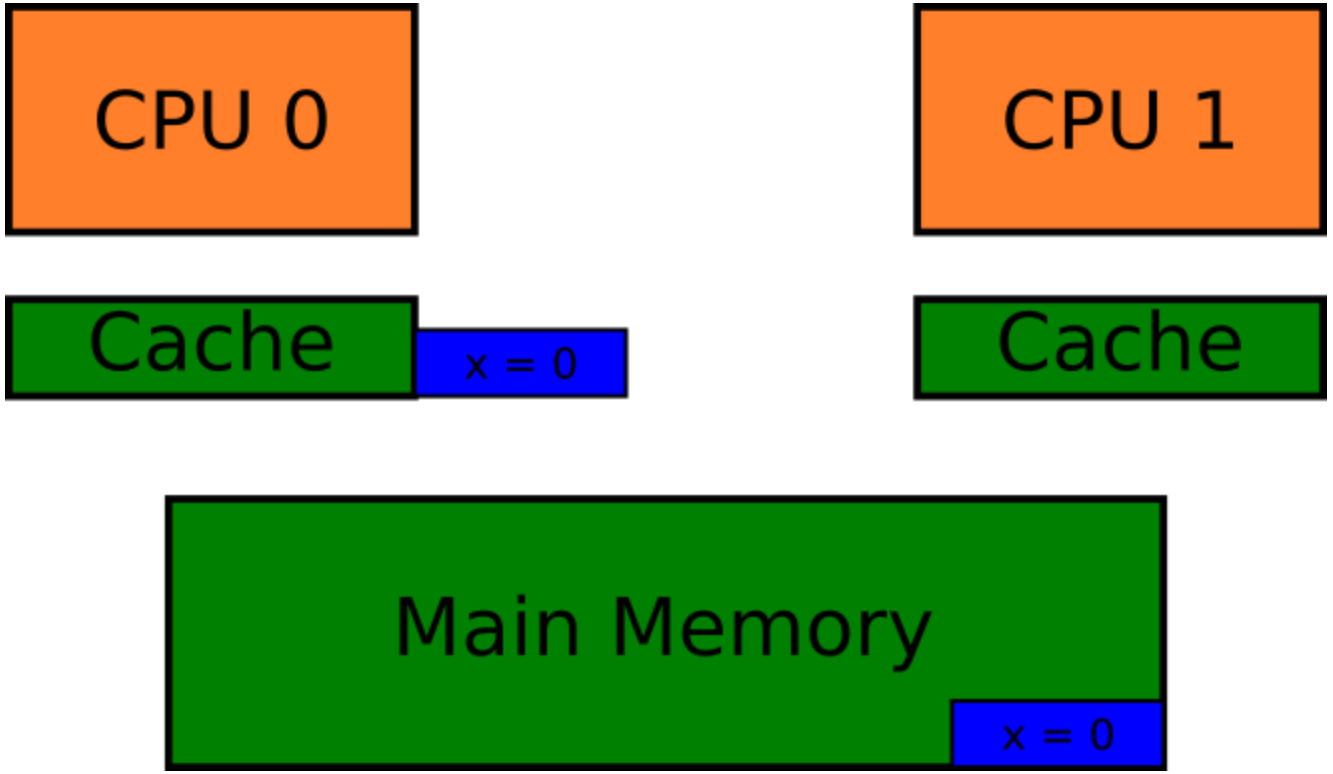
- Multiple CPUs (1, 2, 4, …)
- Connected via Bus (QPI)
- Many cores (12, 24, 36, …)
- Multiple memories
- Shared address space (data in any memory is accessible to any core)



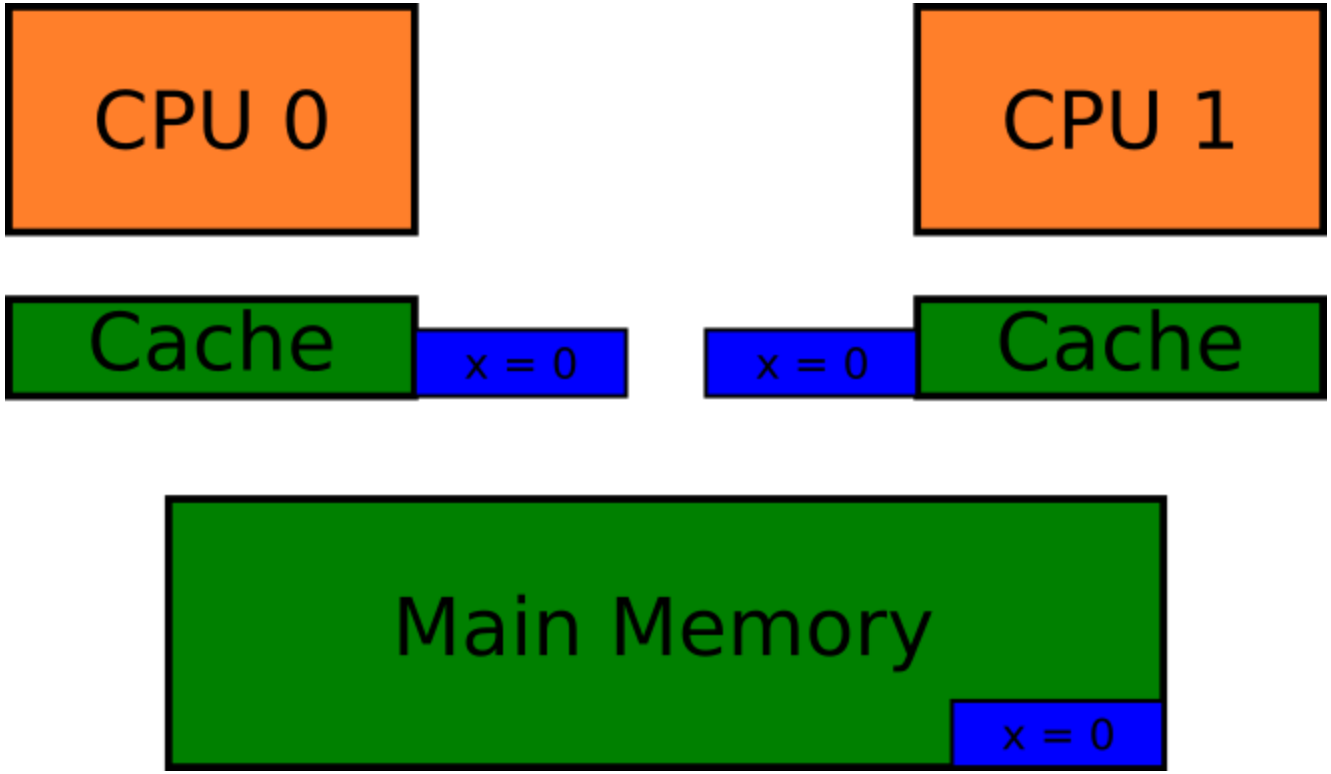
Cache invalidation



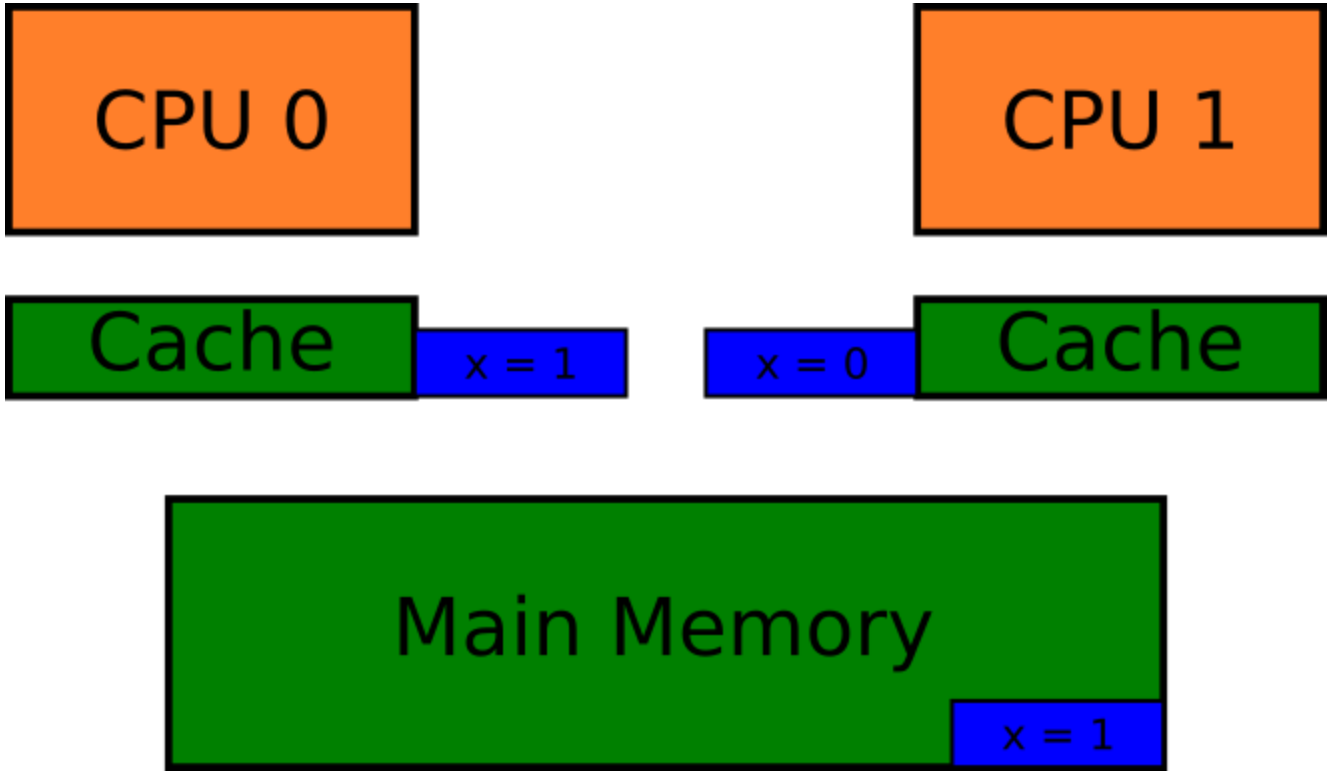
Cache invalidation



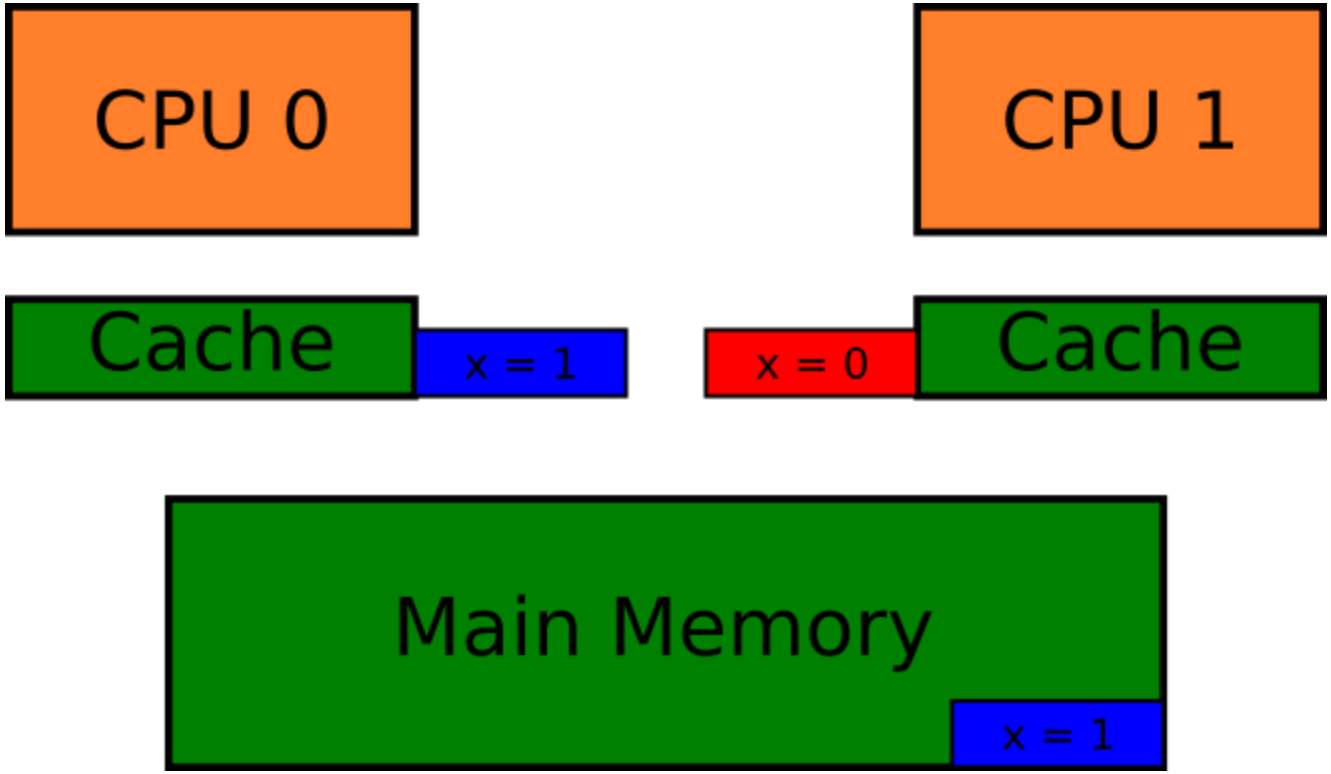
Cache invalidation



Cache invalidation



Cache invalidation



Implications for us

- Do not worry about correctness
  - caching protocols exist for that exact reason

- Do worry about speed of your code
  - protocols will make sure they maximize what they can do
  - We need to write code that maximizes caching for us

Lab Exercises

01-OpenMP-introduction.ipynb

- Learn the basic OpenMP concepts (in C++, 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!

Let’s go!