High Performance Computing for Weather and Climate (HPC4WC)

Content: Shared Memory Parallelism
Lecturers: Tobias Wicky
Block course 701-1270-00L
Summer 2020

Administrative Stuff

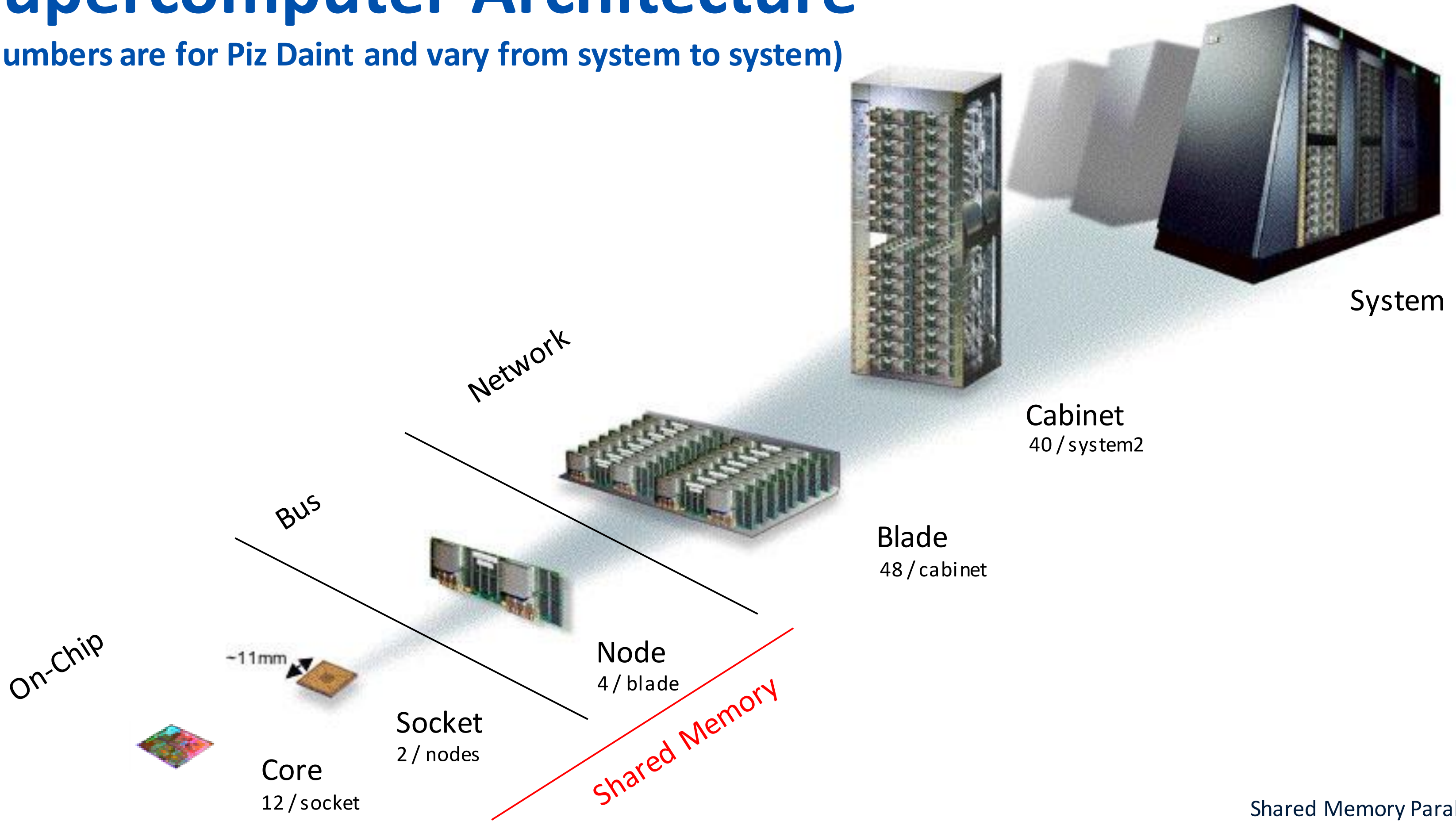
- git pull in the morning for slides
- solutions for the previous day in the solutions folder

Recap

Learning goals

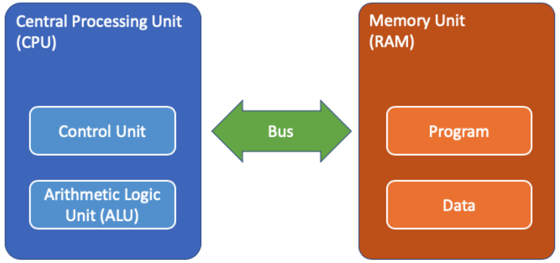
- Understand the shared memory parallelism and the OpenMP programming model
- Understand the limitation of parallelism with Amdahl’s law
- Know about common pitfalls in shared memory computing

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

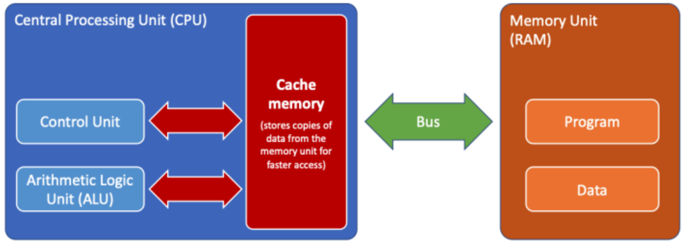


Node Architecture

Von Neumann

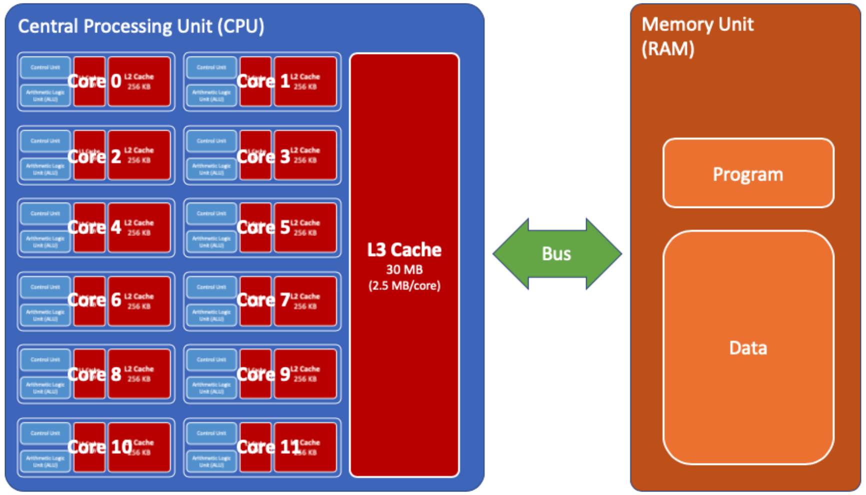


Cache hierarchy



Central Processing Unit (CPU)
Control Unit
Arithmetic Logic Unit (ALU)
Cache memory
(stores copies of data from the memory unit for faster access)
Bus
Memory Unit (RAM)
Program
Data

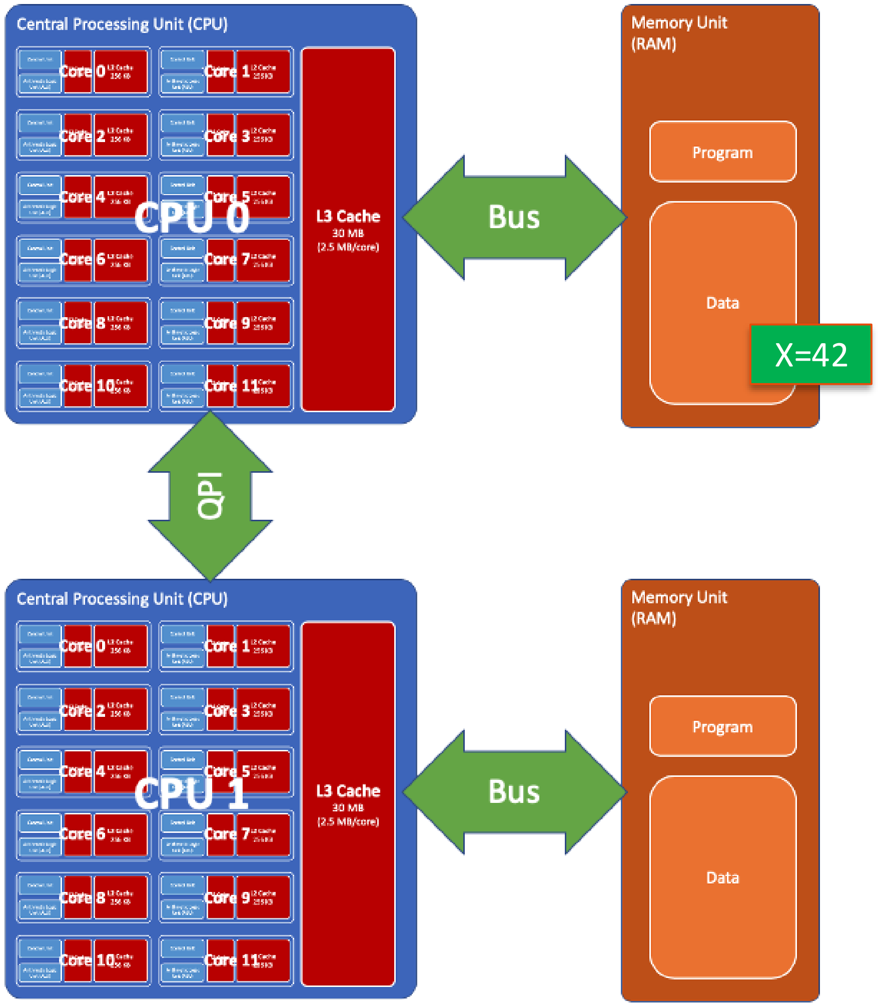
Multicore CPU



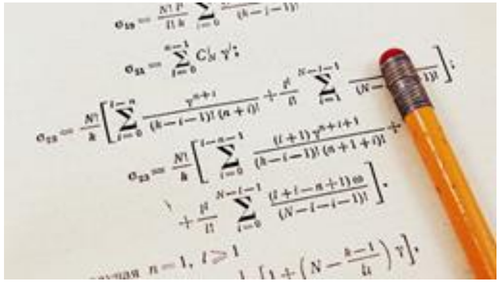
Node Architecture

Share memory node
- Multiple CPUs (1, 2, 4, …)
- Connected via Bus (QPI)
- Many cores (12, 24, 36, …)
- Multiple memorys
- 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.



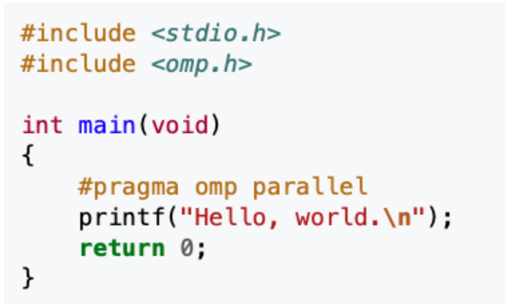
Parallel Computing



Who has already written a parallel program?

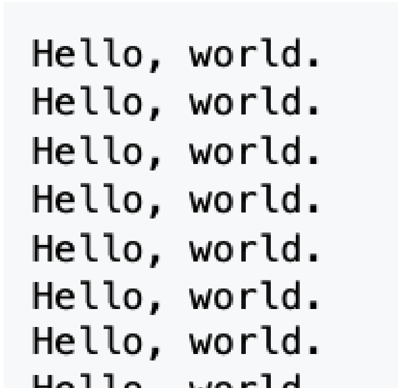
OpenMP

- 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



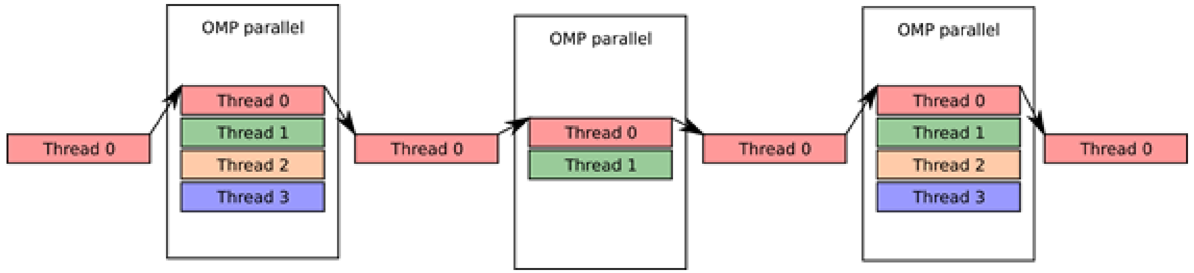
```c
#include <stdio.h>
#include <omp.h>

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.
```

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

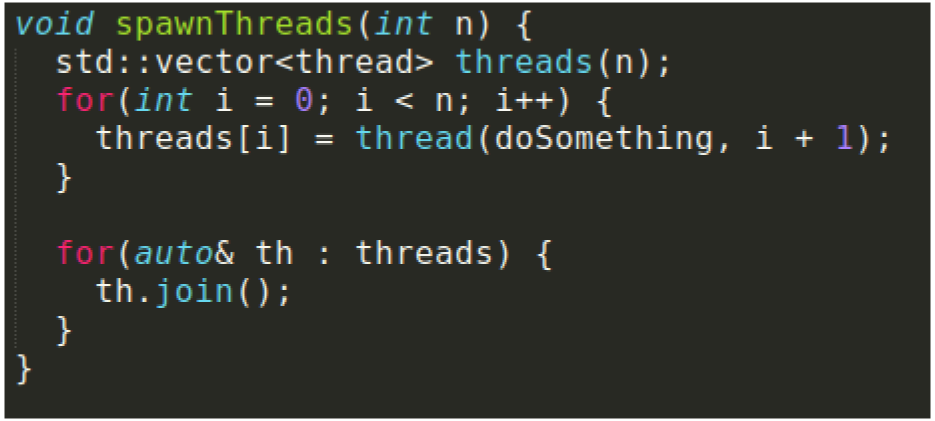
Compiler directives

Pros

- semi automatic parallelisation
- portable across platforms & compilers

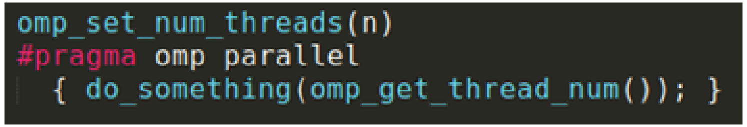
Cons

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



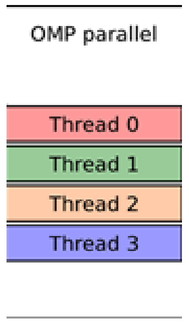
```
void spawnThreads(int n) {
  std::vector<thread> threads(n);
  for(int i = 0; i < n; i++) {
    threads[i] = thread(doSomething, i + 1);
  }

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



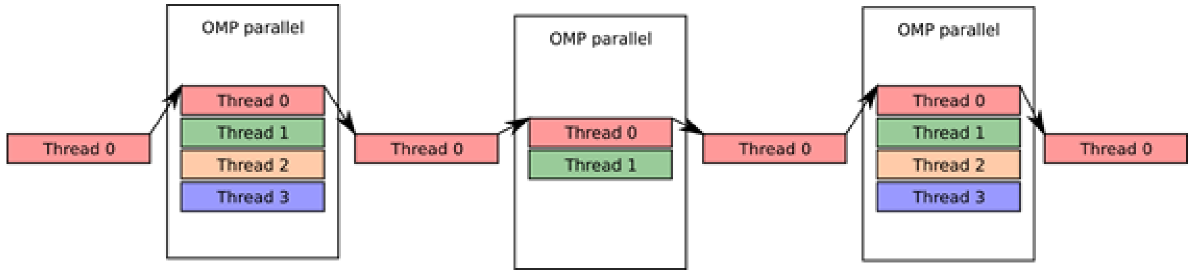
```
omp_set_num_threads(n)
#pragma omp parallel
  { do_something(omp_get_thread_num()); }
```

Region Information



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

Control the Nummer of threads



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

Parallelizing Loops

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

Scheduling of Loops

- 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 **firstprivate** variable is copied in from the sequential code
- 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)

Special parallel sections

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

Barrier and nowait

- Tasks need a way to wait for each other
- omp parallel has an implicit barrier in the end
- for loops have a barrier in the end
- nowait can parallelize tasks

Reductions

- Common pattern, has a special implementation

Are all computer programs parallelizable?

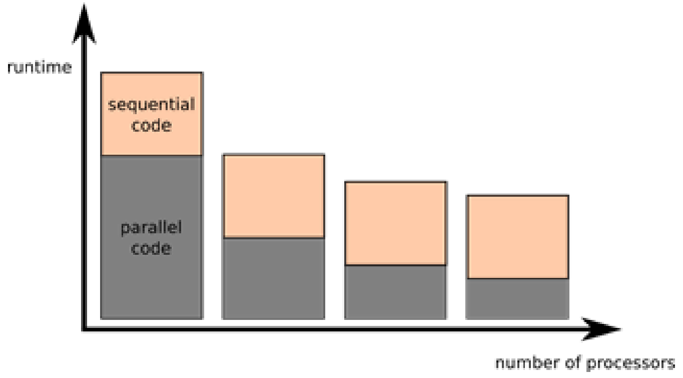
Demo: Calculating Pi

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

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

Amdahl's law

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

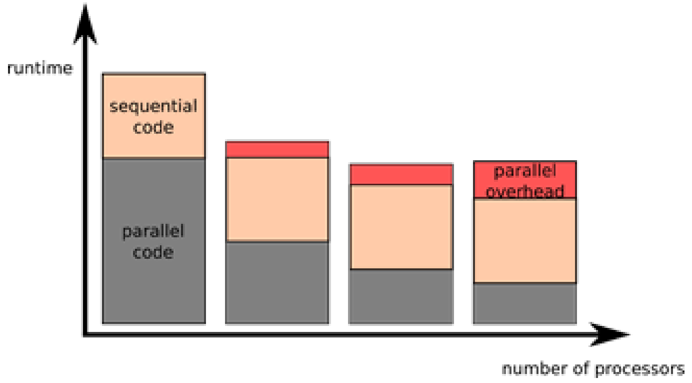


Sequential part of the code

- Opening the notebook
- Filling in the name

Amdahl's law

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

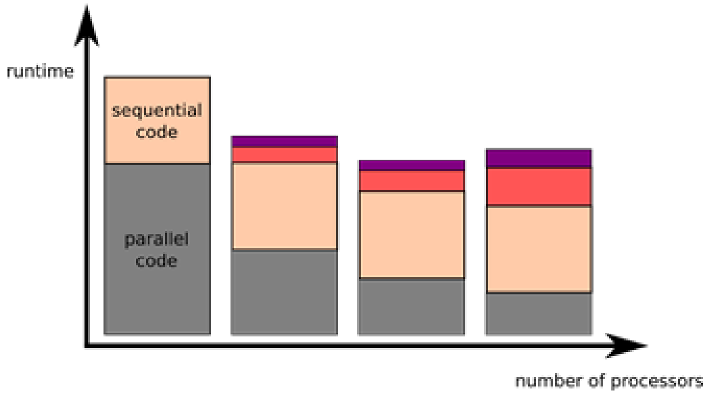


Parallel overhead

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

Amdahl’s law

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



Load imbalance

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

How do we measure speed

- How do we compute how efficient our parallelization is

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

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

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

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

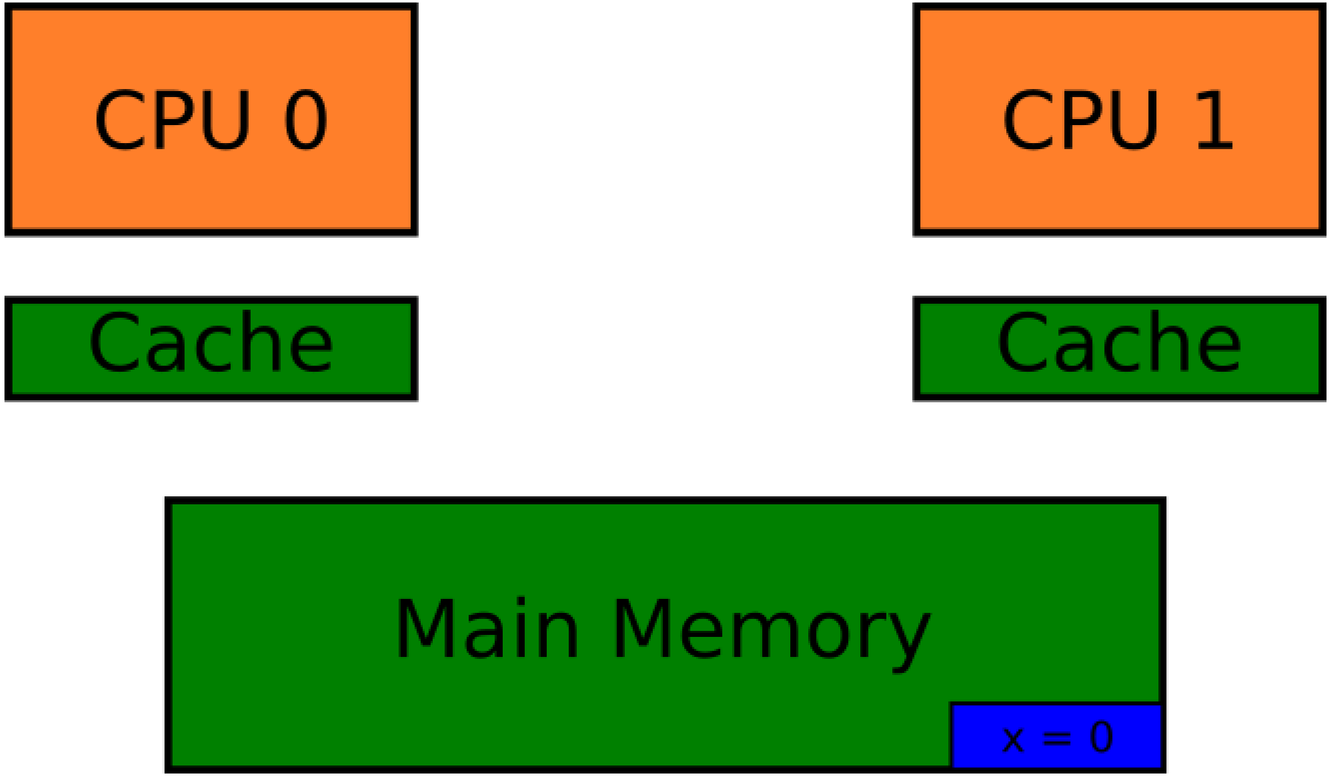
What might lead to bad performance

- Barriers
- Reductions
- Sequential parts of the code
- Load imbalance

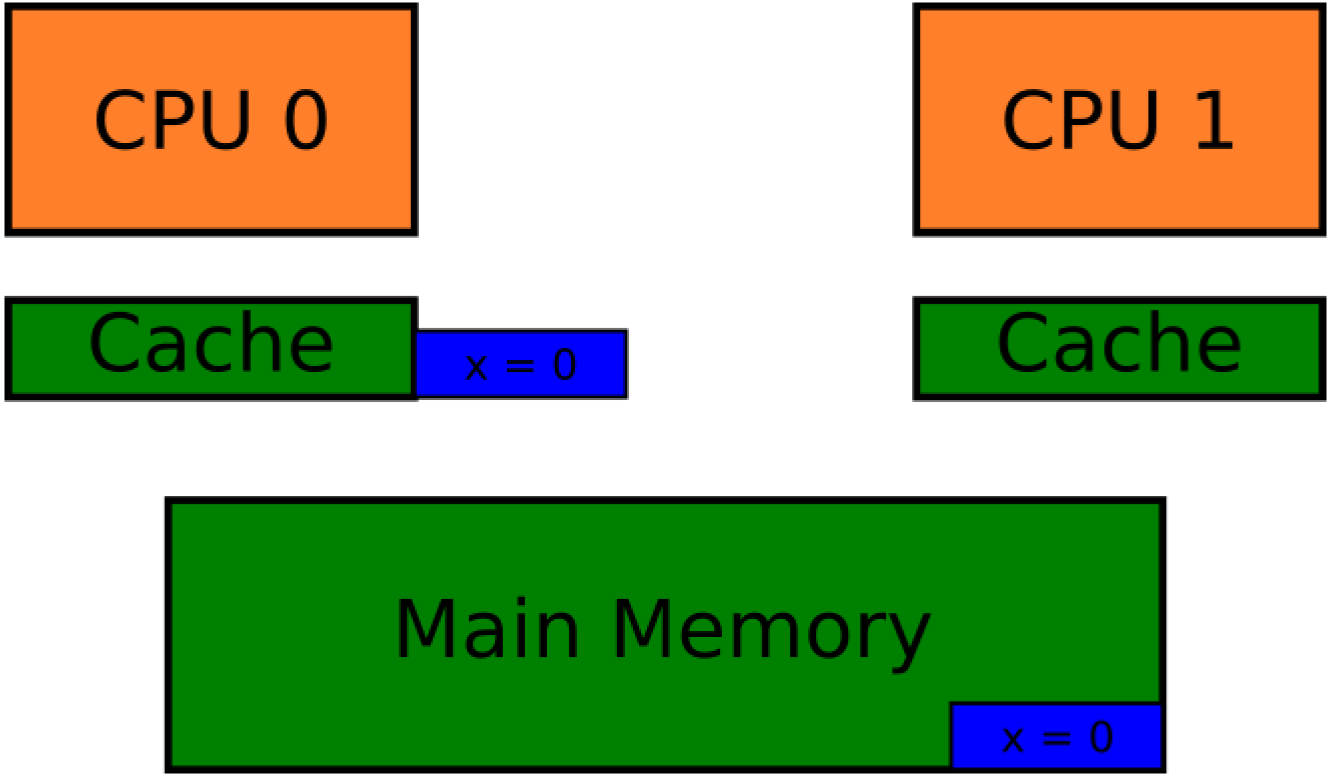
What might lead to bad performance

- Caching issues

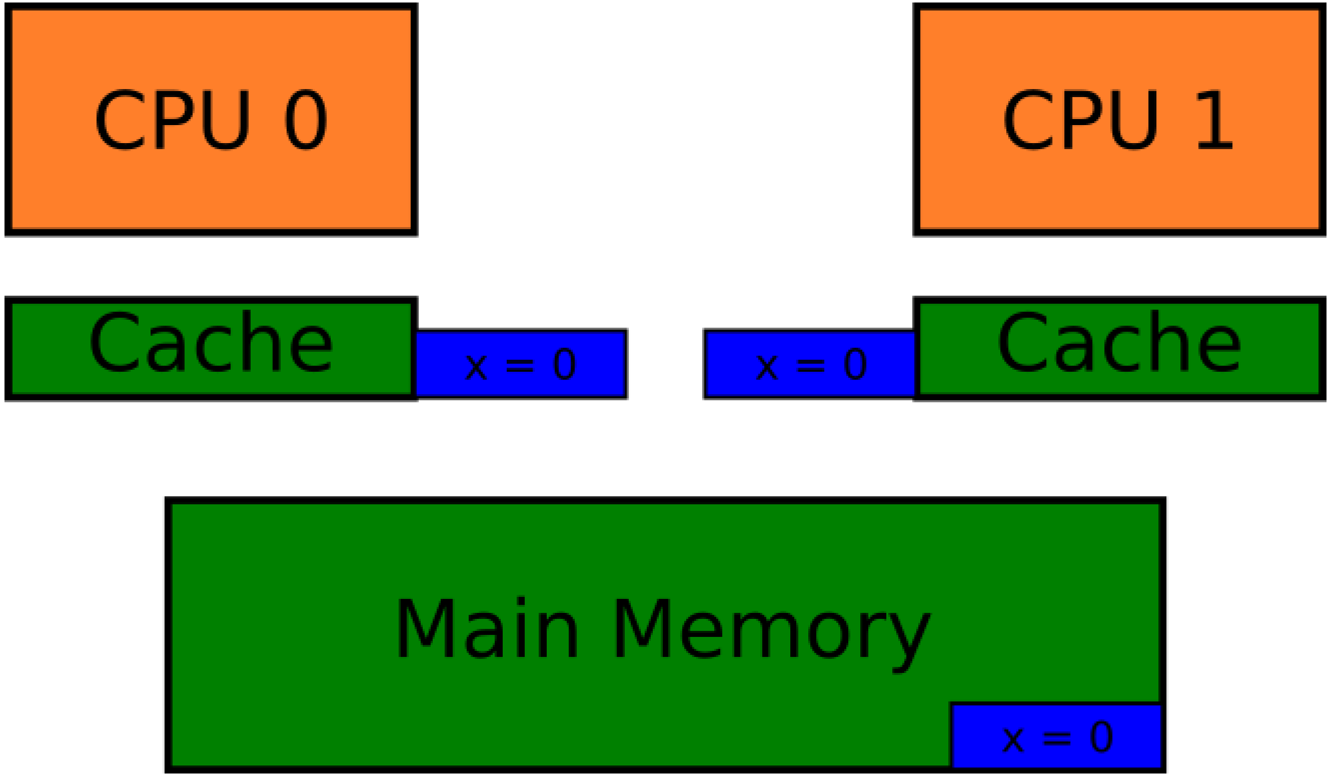
Cache invalidation



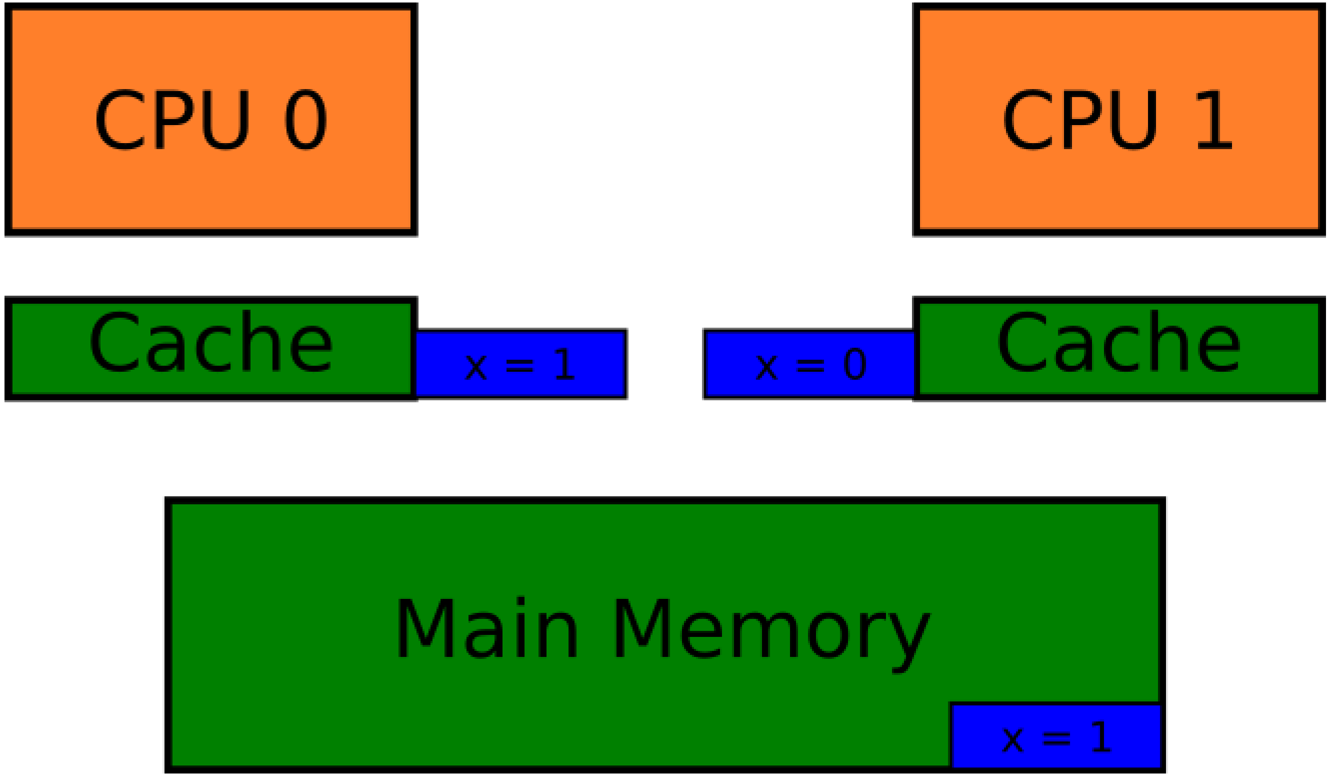
Cache invalidation



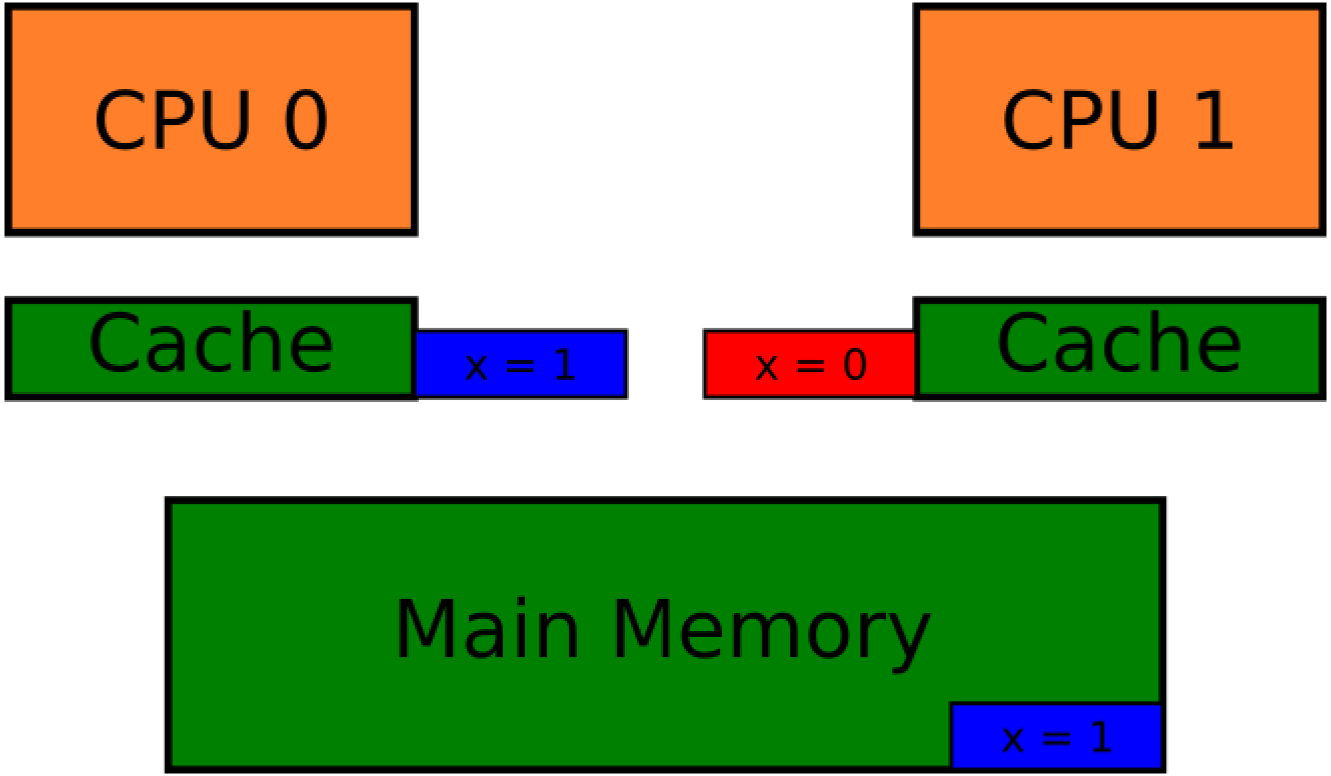
Cache invalidation



Cache invalidation



Cache invalidation



Performance implications

Most of the cache protocols right now take care of correctness for us

We need to ensure performance

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

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

Let’s go!

(see you on Slack)