Why parallelization?

Top 500 computers in the world: www.top500.org

Kilo 10^3
Meta 10^6
Giga 10^9
Tera 10^{12}
Peta 10^{15}
Exa 10^{18}

Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371

Fast computers have several million cores, which need to be used efficiently & simultaneously

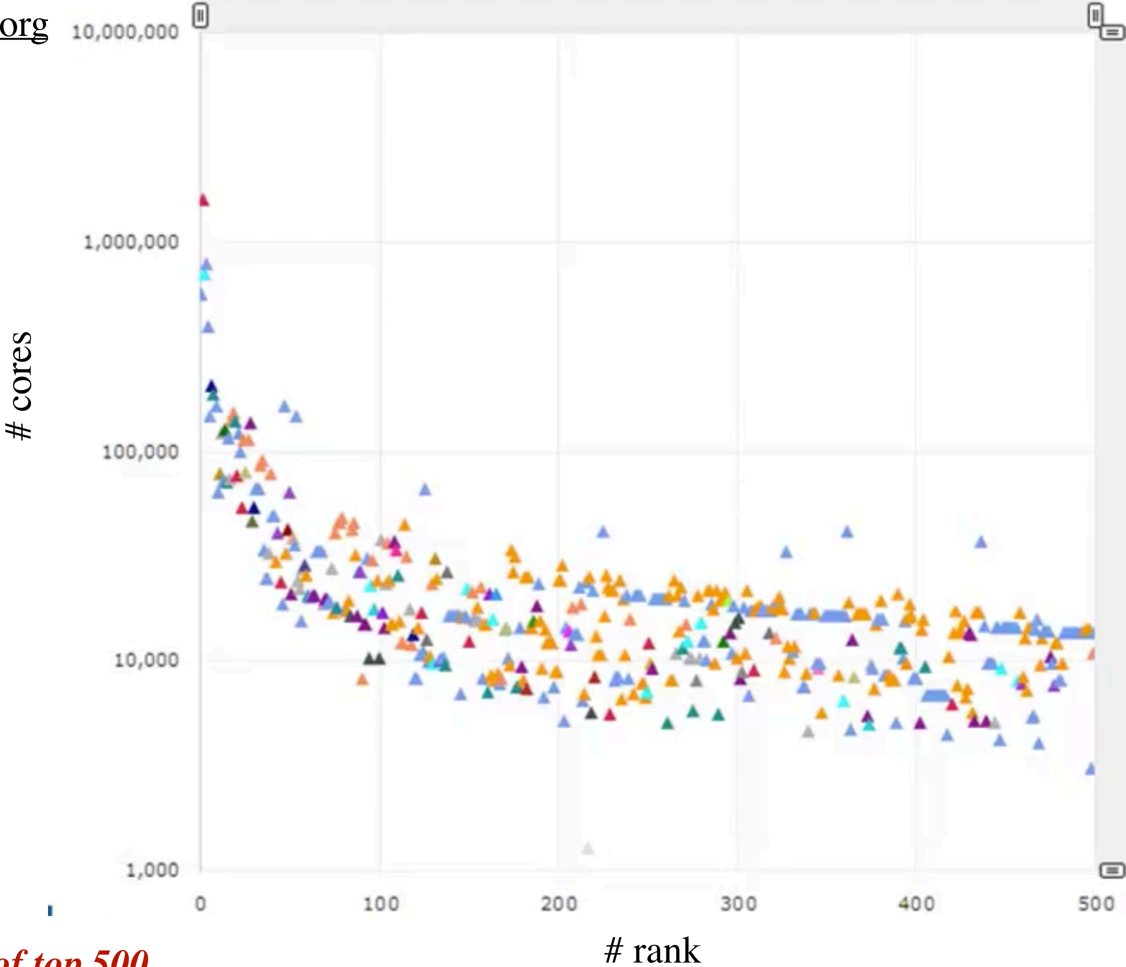
my laptop: 8 cores, 2.4 GHz with 8 single-precision FLOPS's per second
hence theoretical performance = 8*2.4GHz*8 = 38.4GFLOPS/s=0.0384TFLOPS/s

This is theoretical not actual speed, the list contains actual TFLOPS by running LINPACK benchmark

Why parallelization?

Top 500 computers in the world: www.top500.org

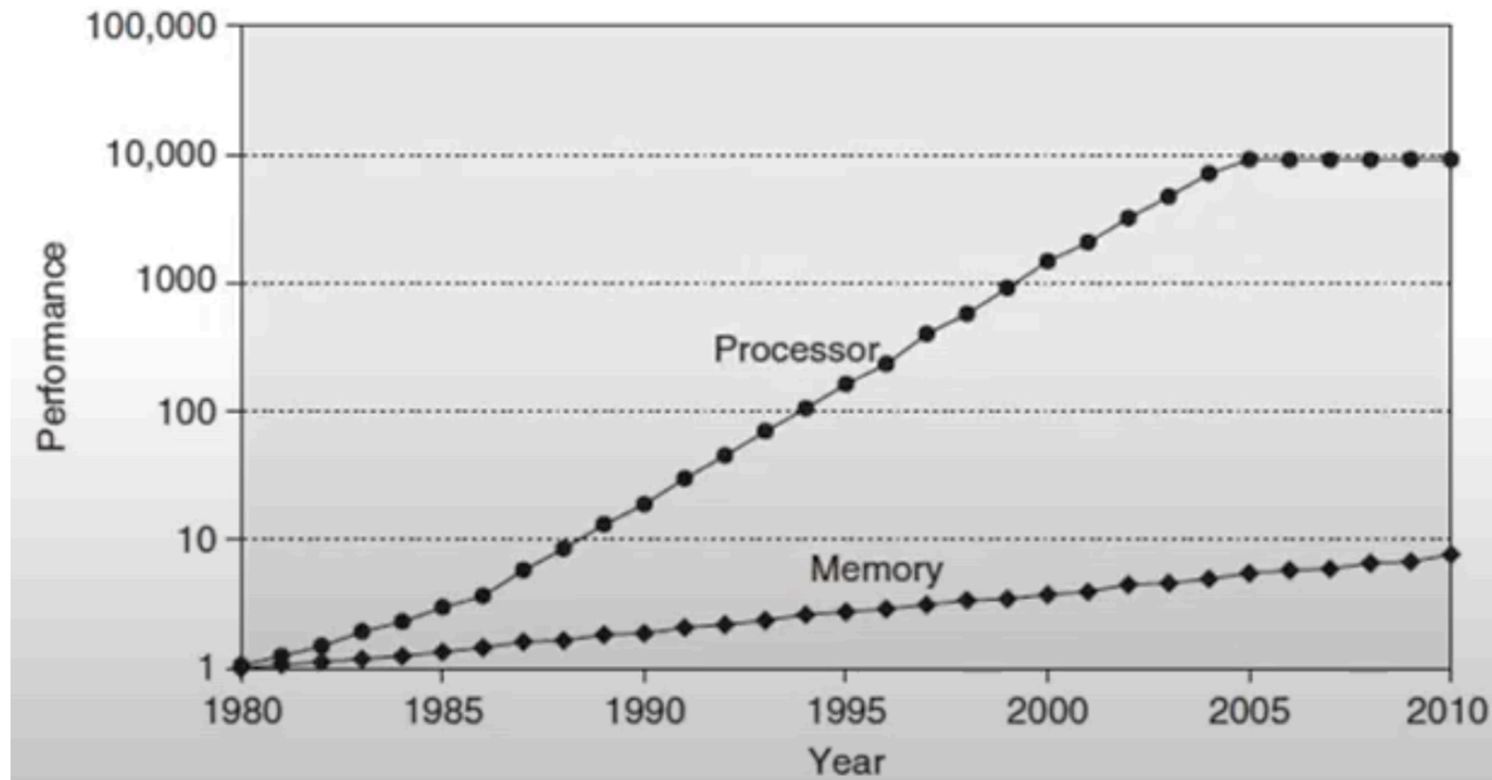
Kilo 10^3
Mega 10^6
Giga 10^9
Tera 10^{12}
Peta 10^{15}
Exa 10^{18}



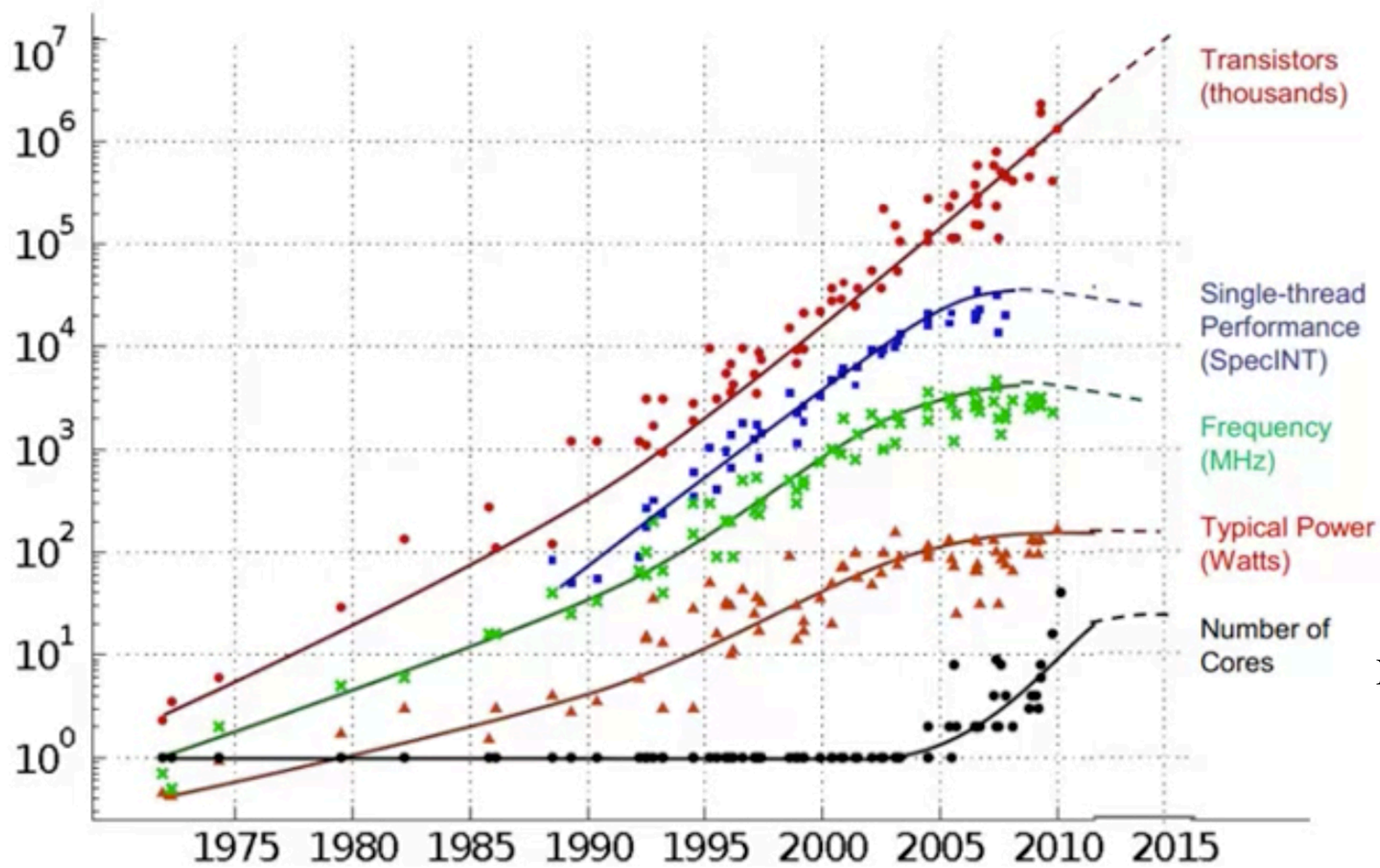
The number of cores is exploding in the list of top 500

Why parallelization?

Processor's speed increased linearly with small slope between 1980-1985 (1.25/year), and larger slope between 1985-2000 (1.52/year)
Processor's speed plateaued in 2005 (people were predicting Moor's law to break).
Instead of increasing the speed of single processor, number of processors and cores is now increasing exponentially



Moore's law still works!



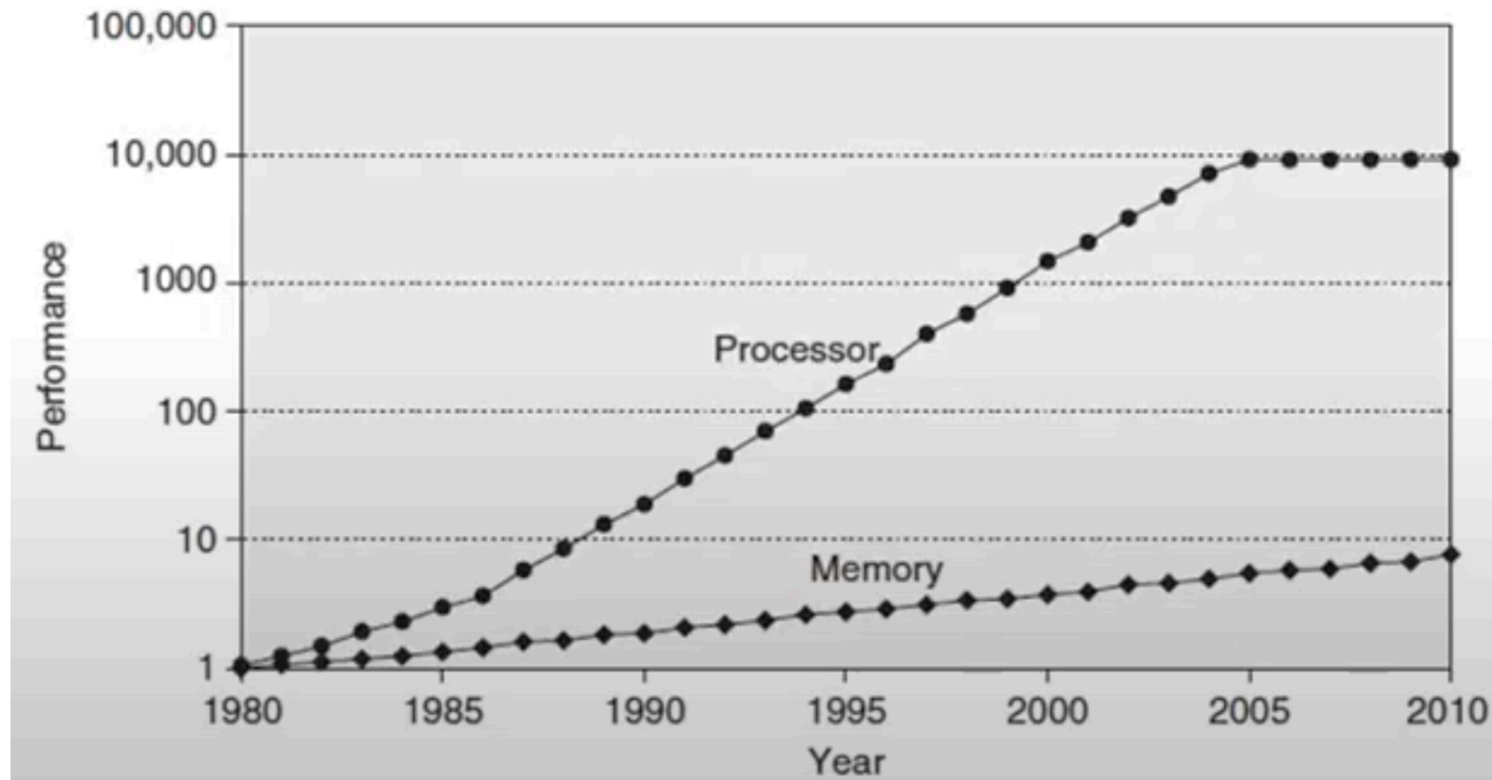
Number of transistors is still exploding, which defines Moore's law.

Quantum limit for single core, it can not be too small, because it stops behaving classically

Number of cores is exploding

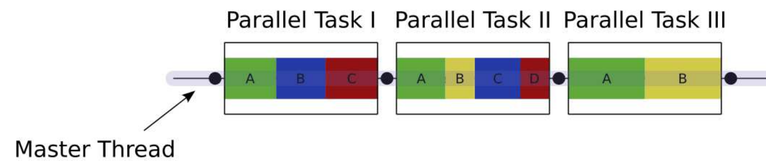
Why parallelization?

Also important is memory latency, which is improving slowly with 1.07/year. Hence memory speed is substantially slower than processor speed, and it will remain so for foreseeable future.

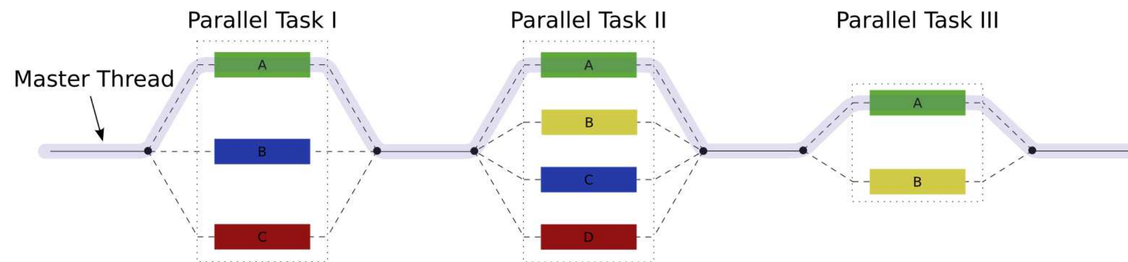


openMP and multicore execution

Usual serial execution



openMP multicore execution



- OpenMP is designed for multi-processor/core to run a program on several cores (using several ”threads”)
- OpenMP programs accomplish parallelism exclusively through the use of threads. Typically, the number of threads match the number of machine processors/cores. However, the actual use of threads is up to the application.
- OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default.
- But sometimes private variables are necessary to avoid race conditions
- OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- Parallelization can be as simple as taking a serial program and inserting compiler directives.... Or as complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

openMP and multicore execution

The simplest case of parallel mandelbrot calculation:

```
#pragma omp parallel for
for (int i=0; i<Nx; i++){
  for (int j=0; j<Ny; j++){
    double x = ext[0] + (ext[1]-ext[0])*i/(Nx-1.);
    double y = ext[2] + (ext[3]-ext[2])*j/(Ny-1.);
    mand[i*Ny+j] = Mandelb(complex<double>(x,y), max_steps);
  }
}
```

The loop over `i` is parallelized. Each core is calculating different `i` term.

Note that `mand` array is shared across all cores, because all cores have access to the entire array, but each core is changing only its own slice of the array.

Note that `x` and `y` must be different on each core. As they are declared inside the loop, compiler makes them private to each core.

In more general case, the `omp parallel` statement is

```
#pragma omp parallel shared(mand,ax,ay) private(beta,pi)
```

By default all variables are shared, hence `shared` statement is not really needed.

openMP and multicore execution

The same loop in fortran is:

```
!$OMP PARALLEL DO  PRIVATE(j,x,y,z0)
do i=1,Nx
   do j=1,Ny
      x = ext(1) + (ext(2)-ext(1))*(i-1.)/(Nx-1.)
      y = ext(3) + (ext(4)-ext(3))*(j-1.)/(Ny-1.)
      z0 = dcmplx(x,y)
      mande(i,j) = Mandelb(z0, max_steps)
   enddo
enddo
!$OMP END PARALLEL DO
```

Note that in fortran all variables are declared at the top of the program, hence `x, y, z0, j` need to be declared private. Also `i` is private, but the first loop counter does not need to be added to the private list, as compiler will add it automatically.

The code is compiled by adding a flag `-fopenmp`:

```
g++ -fopenmp -O3 -o mandc mandc.cc
```

or

```
gfortran -fopenmp -O3 -o mandf mandf.f90
```

Also the environment variable `OMP_NUM_THREADS` should be set to the number of cores (threads) we want to use. We can issue a command

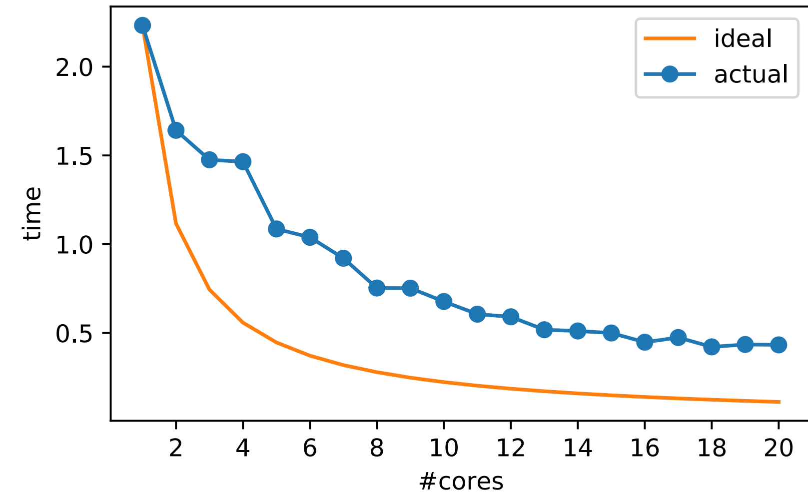
```
export OMP_NUM_THREAS=4
```

openMP and multicore execution

Example of time for mandelbrot set on multiple cores for Intel Core i9 processor:

speed improves, but not close to theoretical (1/core) estimate. Why?

speed improves even beyond 8 threads, even though we have 8 cores. Why?



One more openMP example

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

```
#include <iostream>
#include <ctime>
#include <cmath>
#include <omp.h>
using namespace std;

double f(double x){
  return 4.0/(1.0+x*x);
}
double calcPi(int n)
{
  const double dx = 1.0/n;
  double fSum = 0.0;
  #pragma omp parallel for reduction(+:fSum)
  for (int i=0; i<n; ++i){
    double x = (i+0.5)*dx;
    fSum += f(x);
  }
  return fSum*dx;
}
```

1/n is spacing for trapezoid rule

reduction: We not only make the loop parallel, but we need to tell the compiler that fSum is neither private not shared, but variable to be reduced.

reduction operators are:

`+, -, *, min, max, &, |, ^, &&, ||`

One more openMP example

The alternative, but worse implementation:
We do not specify that fSum is obtained by reduction, but we specify that a particular line “fSum+=df” should be done without parallelization.

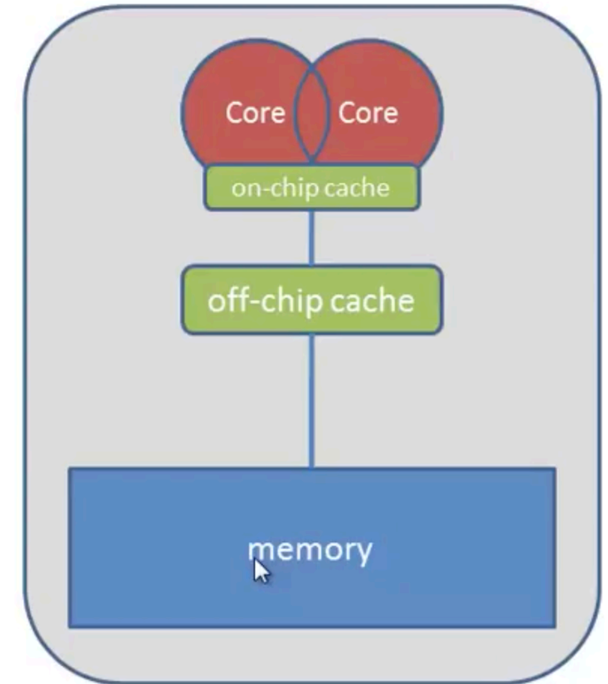
omp critical can be used for any line that can not be parallelized.

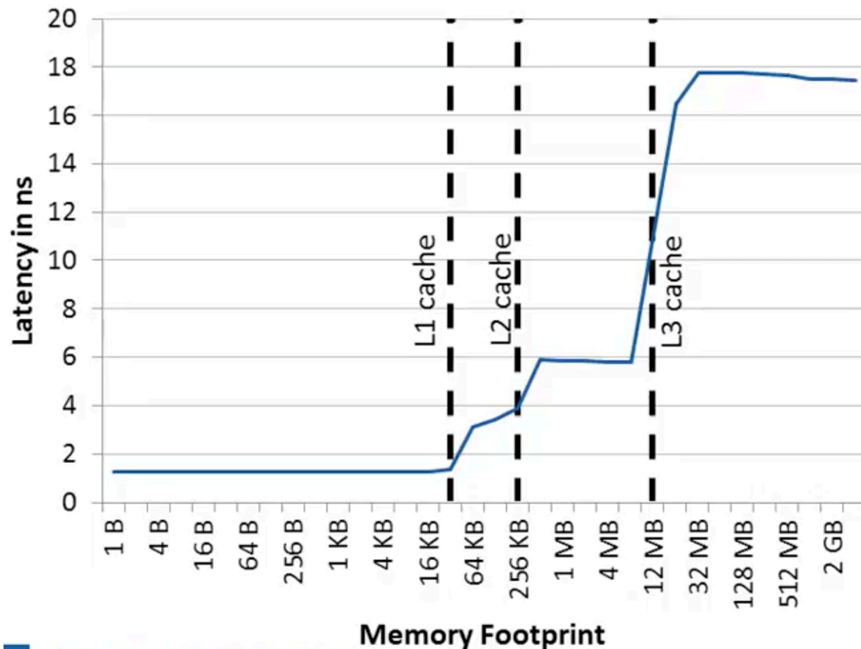
```
#include <iostream>
#include <ctime>
#include <cmath>
#include <omp.h>
using namespace std;

double f(double x){
  return 4.0/(1.0+x*x);
}

double calcPi_bad(int n)
{
  const double dx = 1.0/n;
  double fSum = 0.0;
  #pragma omp parallel for
  for (int i=0; i<n; ++i){
    double x = (i+0.5)*dx;
    double df = f(x);
    #pragma omp critical
    fSum += df;
  }
  return fSum*dx;
}
```

openMP and multicore execution

Memory access is slow. When several cores need to manipulate few MB of data, several cores compete for the bandwidth/access to RAM and L3 cache.

CPU: ~3GHz ~ 0.3ns per tick ~ 0.04ns for floating point operation (8FP per tick)
L1 cache: latency~ 1ns, size ~16KB
L2 cache: latency~ 3ns, size ~256KB
L3 cache: latency~ 6ns, size ~2MB
RAM : latency~20ns, size ~GB, bandwidth~0.3GHz, corresponding to 3.3ns



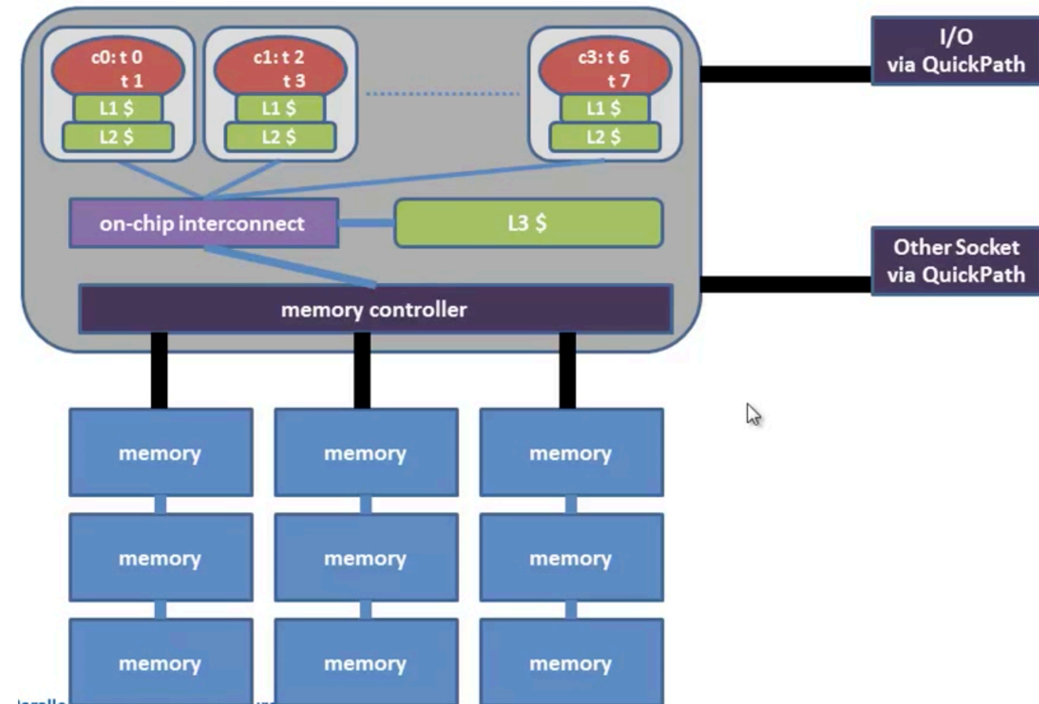


Latency: Delay incurred when a processor accesses data inside the memory (even when reading just one number)

Bandwidth: Rate at which data can be read from or stored into memory by a processor

More realistic multicore architecture

~32KB L1 cache per core
~256KB L2 cache per core
~2MB L3 cache per core, but shared by all cores
several GB RAM

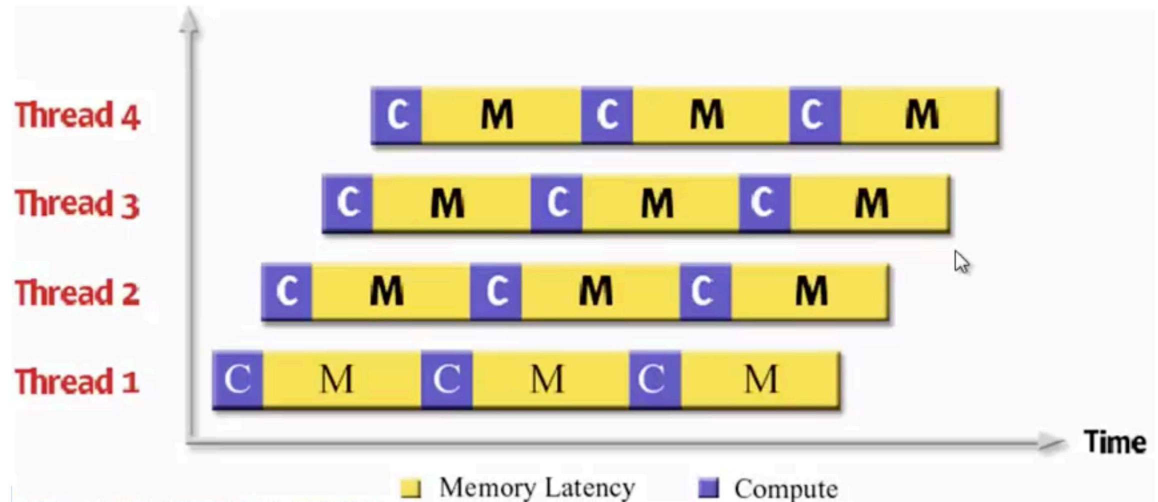


Since we write data into common variable, speed is limited by memory access and not computation, hence we do not get theoretical performance.

Why do we get speedup when using more threads than cores?

Design of modern CPU

Access to memory is arranged to be staggered: some threads are doing computation and some are writing, so we can squeeze out a bit of performance by floading CPU with threads. Notice that this is not necessary the case. Sometimes the execution is slowed down when number of threads exceeeds number of cores.



If you want to learn more about `openMP,` consult these resources

https://www.openmp.org
https://www.openmp.org/resources/tutorials-articles/
https://www.youtube.com/channel/UCtdrEoe46tD2IvJJRs_JH1A/videos

How to improve memory management?

To squeeze out best performance can be a very hard software engineering problem, which is handled by compiler, and user does not have complete overview how memory access is handled.

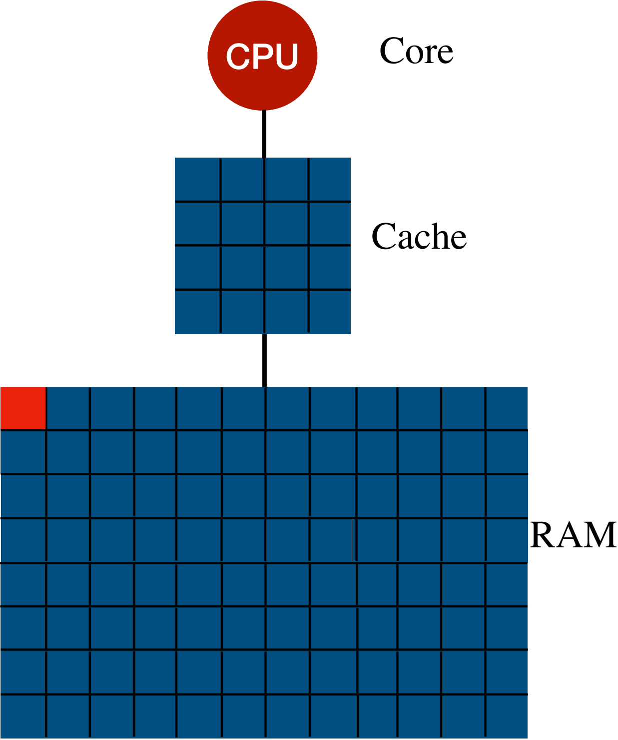
However, there are some general ideas tips of how to access memory to allow compiler well optimize the code.

- Do not use hard-disc for data manipulation if possible. Keep data in RAM. If you need a lot of RAM, estimate whether it fits into RAM. Rethink your algorithm before you start writing data to hard-disc.
- Try avoiding random access of data in RAM to reduce cache misses.
- The data which you need in the innermost loop should be stored in a way that the access is maximally continuous.

Why should we access memory continuously?
Because CPU does not load a single number, but a page, which is 64 byte (8 double's).
We can use data already present.

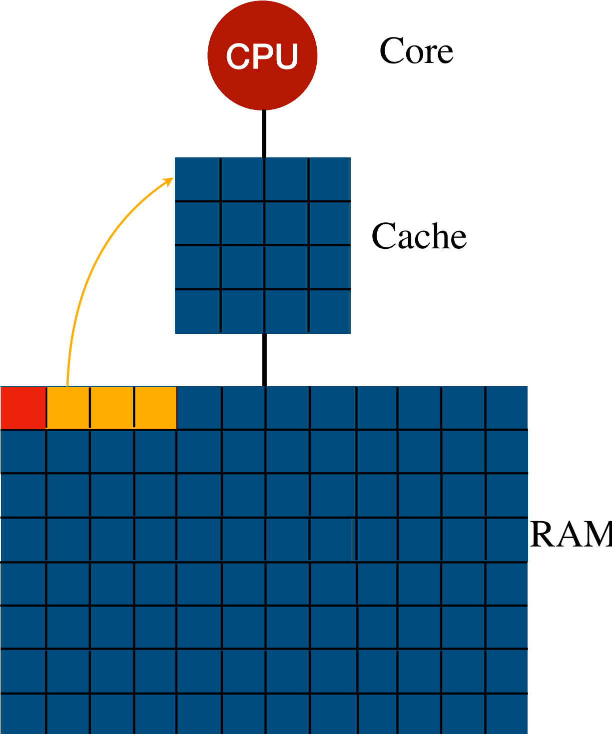
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache



How does memory work?

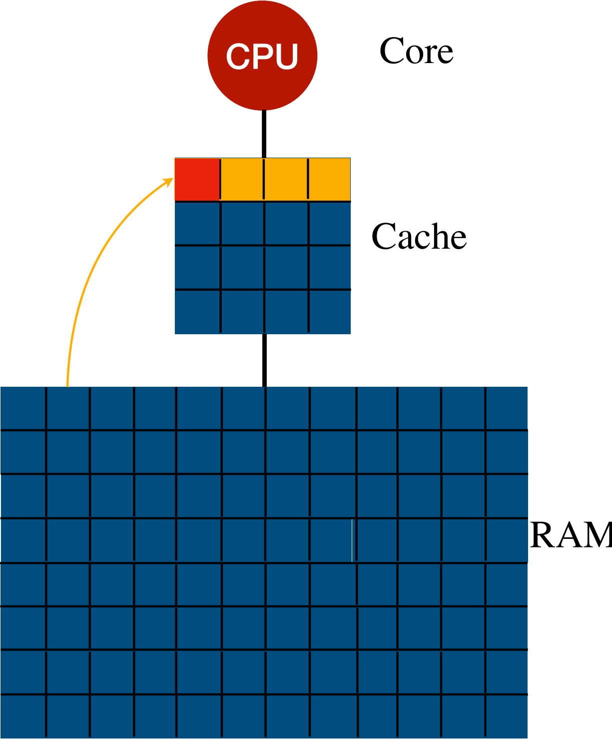
For reading or writing one element in the memory, a complete page of memory has to be loaded into cache



How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache

Now the processor can read and write the elements

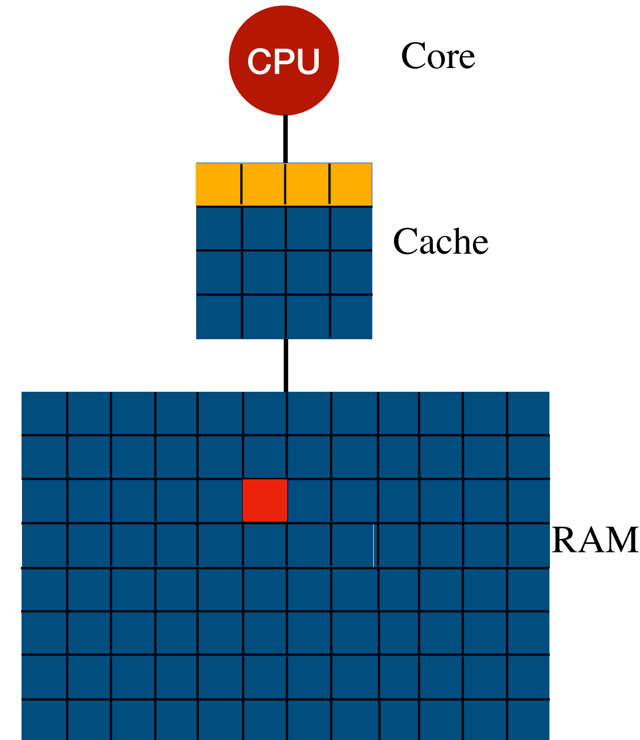


How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache

Now the processor can read and write the elements

If the next element is outside the loaded cache pages, another page needs to be loaded.

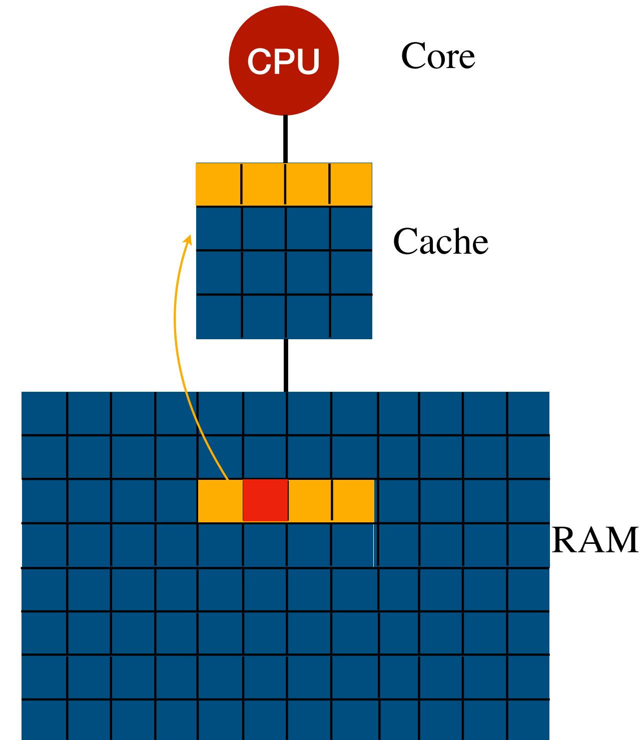


How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache

Now the processor can read and write the elements

If the next element is outside the loaded cache pages, another page needs to be loaded.

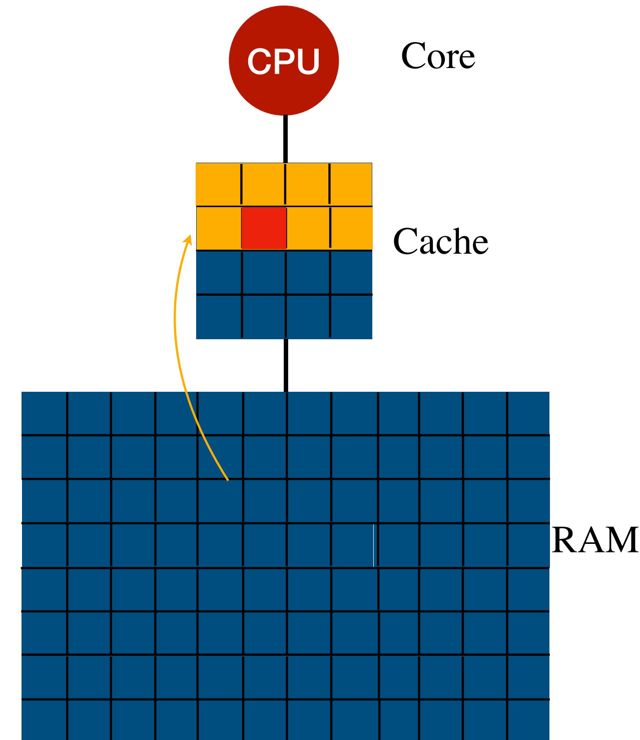


How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache

Now the processor can read and write the elements

If the next element is outside the loaded cache pages, another page needs to be loaded.



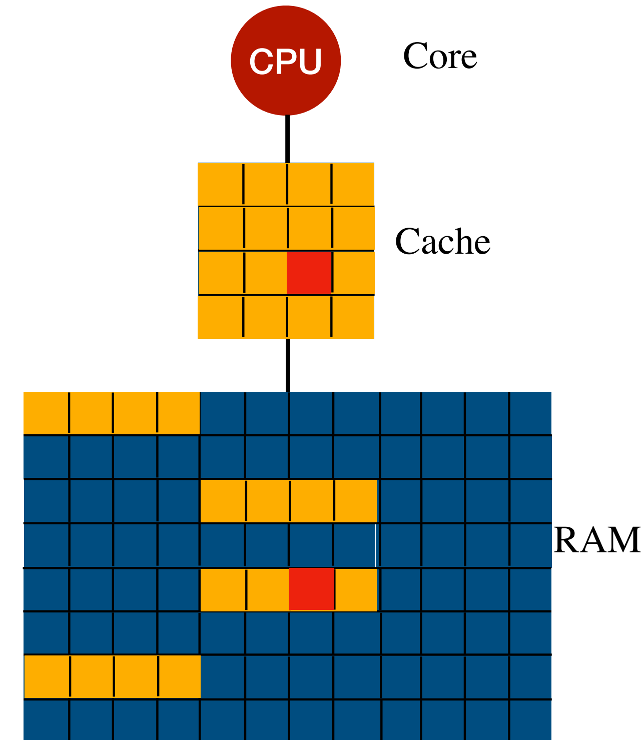
How does memory work?

For reading or writing one element in the memory, a complete page of memory has to be loaded into cache

Now the processor can read and write the elements

If the next element is outside the loaded cache pages, another page needs to be loaded.

Accessing an element already loaded in cache is very fast and does not cost extra cycles.



How does memory work?

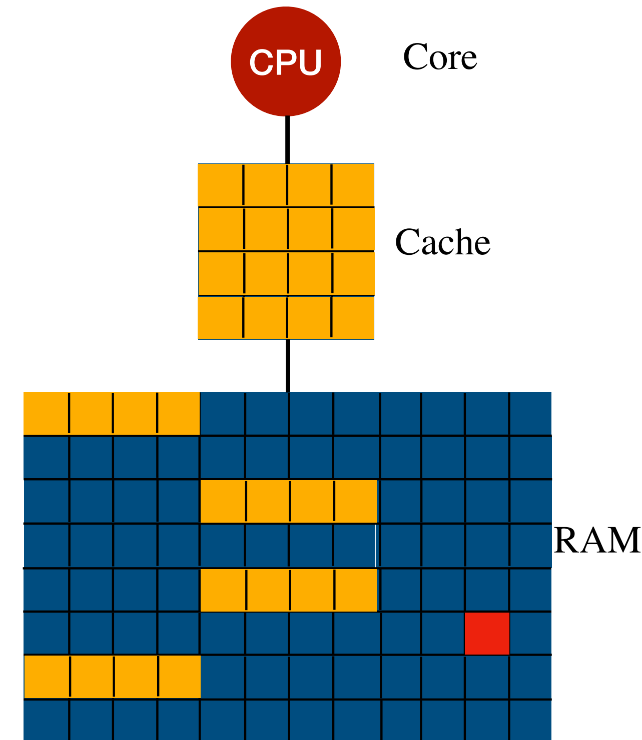
For reading or writing one element in the memory, a complete page of memory has to be loaded into cache

Now the processor can read and write the elements

If the next element is outside the loaded cache pages, another page needs to be loaded.

Accessing an element already loaded in cache is very fast and does not cost extra cycles.

If the cache is full and a new cache page should be loaded, an old one must be dropped, which costs several hundred cycles, and is called cache miss.



How to improve memory management?

Typical example is a matrix manipulation.
In C or C++, one needs to access multidimensional arrays in the following order since the data is stored in a row major order.

```
for (int i=0; i<size; i++)
    for (int j=0; j<size; j++)
            A[i][j] = .....
```

In Fortran, the same loop should be written in the following way

```
do i=1, size
    do j=1, size
            A(j,i) = .....
    enddo
enddo
```

This is because Fortran (C) uses column (row) major storage. The figure explains it all.

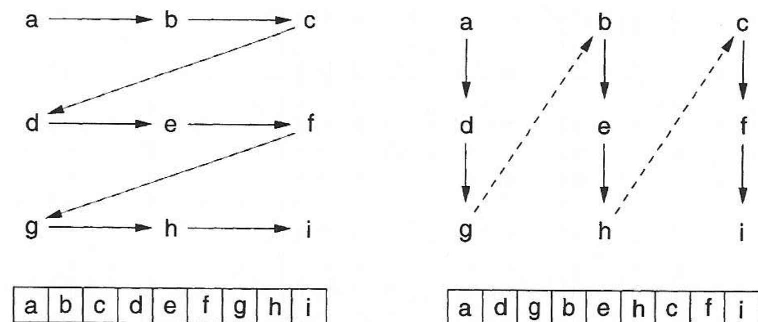


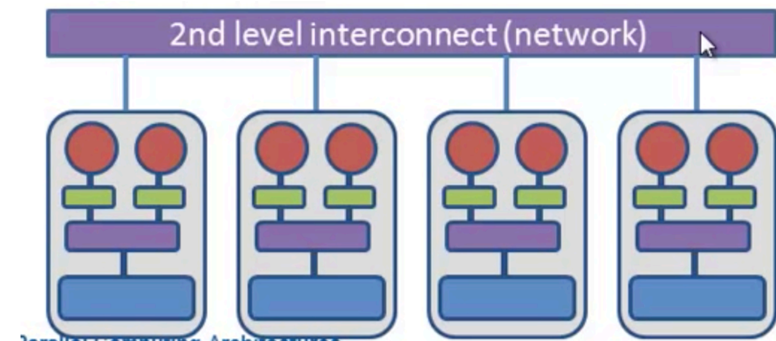
Fig. 15.2 (*Left*) Row-major order used for matrix storage in C and Pascal; (*right*) column-major order used for matrix storage in Fortran. On the bottom is shown how successive matrix elements are stored in a linear fashion in memory.

Multi-node parallelization : MPI

When parallel execution uses several nodes (not just several cores on a single node), we need to use MPI parallelization.
MPI requires one to call specialized MPI routines to communicate and exchange data. This is more technically involved programing.

Inter-node (2nd level interconnect) speed:

- InfiniBand: latency ~5μs, bandwidth ~1Gb/s
- GigaBit Ethernet: latency 60μs, bandwidth ~0.1Gb/s



Latency: Time required to send a message of size zero (time to set up communication)

Bandwidth: Rate at which large messages (>=2Mb) are transfered

Virtual box from past years (which should work if other installations fail):

If you do not want/succeed to install the necessary software, you should download the file :
http://hauleweb.rutgers.edu/downloads/509/509.ova
(warning: 4.8GB file, it might take a while)

Then you should install VirtualBox to run the provided virtual machine:
https://www.virtualbox.org

Finally, start the VirtualBox and navigate to *File/Import Appliance*, and choose the downloaded 509.ova file.

Then click *Start* and wait for the linux to start. Once linux is running, you can start a terminal *Konsole* and start *emacs* in the terminal. You can navigate to

```
cd ~/ComputationalPhysics/mandelbroat
```

and examine the files we will discuss in the first lecture. If you need username, use *student*, and passwd *student123*.

Learning Python

Next learning python from the following lectures:

https://github.com/jrjohansson/scientific-python-lectures

If you prefer video, this might be very good one:

https://www.youtube.com/watch?v=xCKfR80E8ZA