Revision 2

C2115 Practical introduction to supercomputing

Lesson 13

Petr Kulhanek

kulhanek@chemi.muni.cz

National Center for Biomolecular Research, Faculty of Science,
Masaryk University, Kotlářská 2, CZ-61137 Brno

Content

- **Increasing performance of supercomputers**

  SMP, multicore CPU, NUMA

- **Parallelization of programs**

  numerical integration

  - **OpenMP**
  - **MPI**

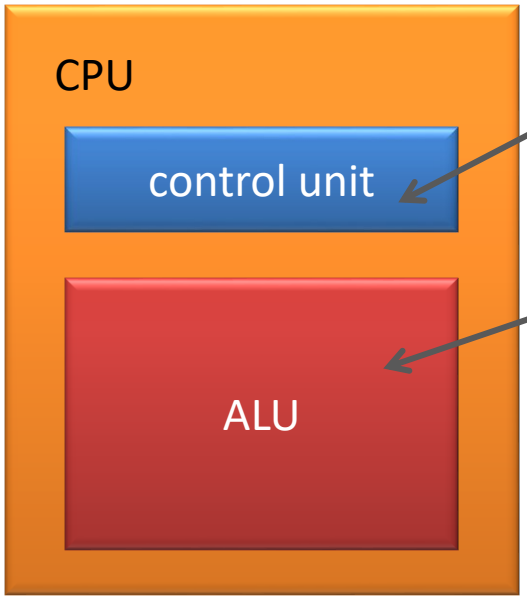
  parallelization pitfalls, Amdahl's law

Architecture of computer
Increasing computational power

CPU

Processor or **CPU** (**Central Processing Unit**) is an essential part of the computer; it is a very complex circuit that (sequentially) **executes the machine code** stored in the computer's memory. The machine code is composed of the instructions, which are loaded into operation memory.

www.wikipedia.org



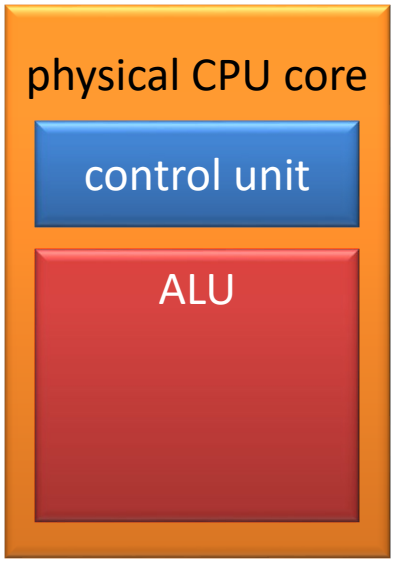
Control unit reads machine code (instructions) and data and prepares them for execution on ALU.

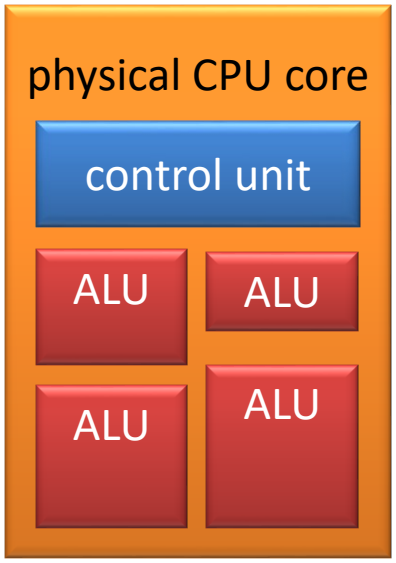
ALU (arithmetic and logic unit) executes arithmetic operations and evaluates logical conditions.

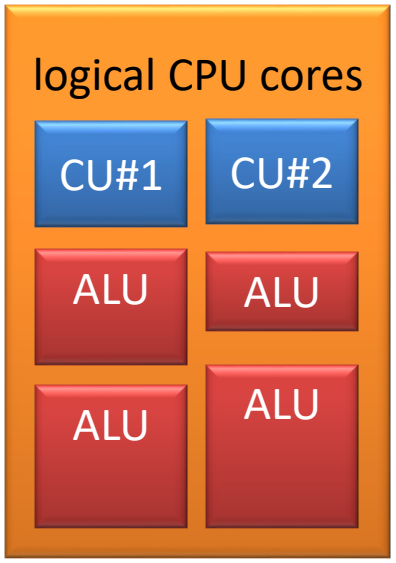
(Sequential) execution of machine code is controlled by internal clock.

How does CPU (ALU) work with numerical values?

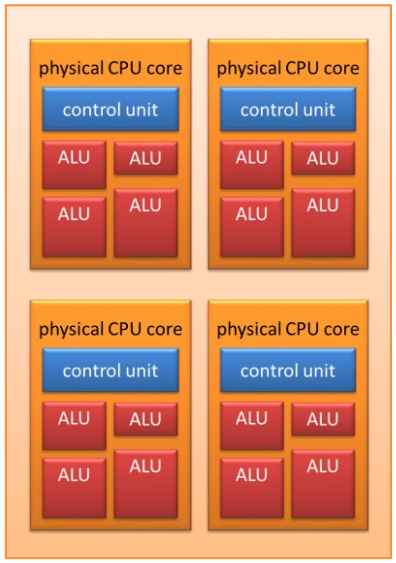
Increasing Computing Power







hyperthreading



multi-core processor

Strategies:

- **increasing clock frequency**
  - – physical limitations (miniaturization, lowering voltage)
- **increasing number of ALUs** and their specializations (out-of-order execution, speculative execution, vector instructions)
  - – efficiency limited by executed code
- **sharing ALUs among control units** (hyperthreading)
  - – efficiency limited by executed code
- **multi-core processor**
  - – efficiency limited by executed code

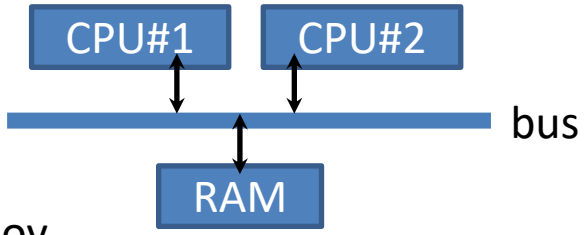
software optimization
or
new algorithms
are necessary to benefit from these features

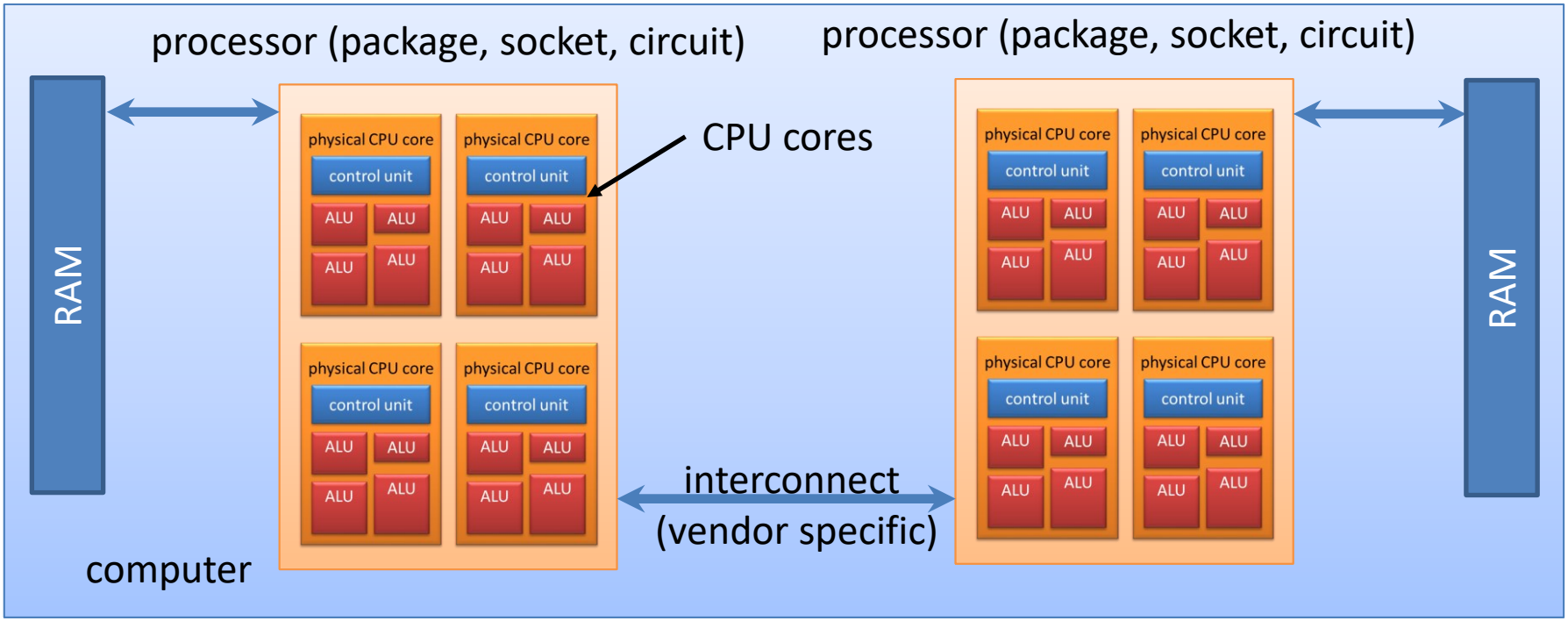
Symmetric Multiprocessing (SMP)

Symmetric multiprocessing represents a system containing **identical** CPUs accessing shared memory.

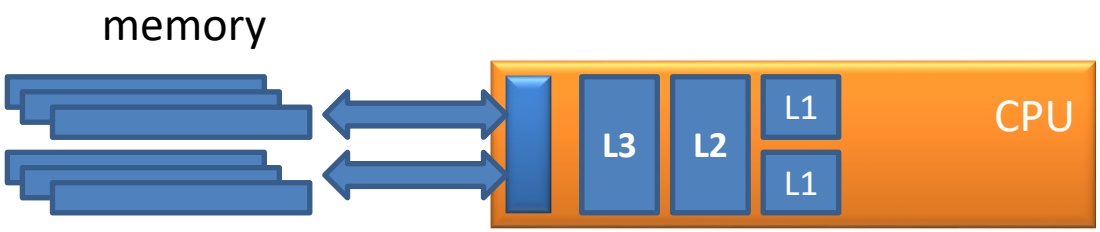
Utilizations of more CPUs **increases computing power** of the system.

new algorithms are required to employ the computing power

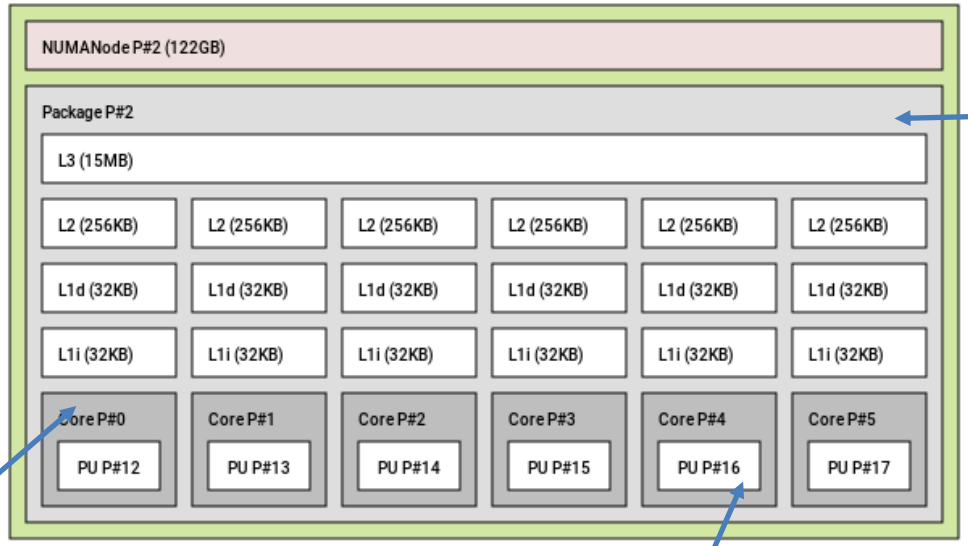




Processor Caches



Processor cache improves efficiency of CPU access into central memory (latency and bandwidth).

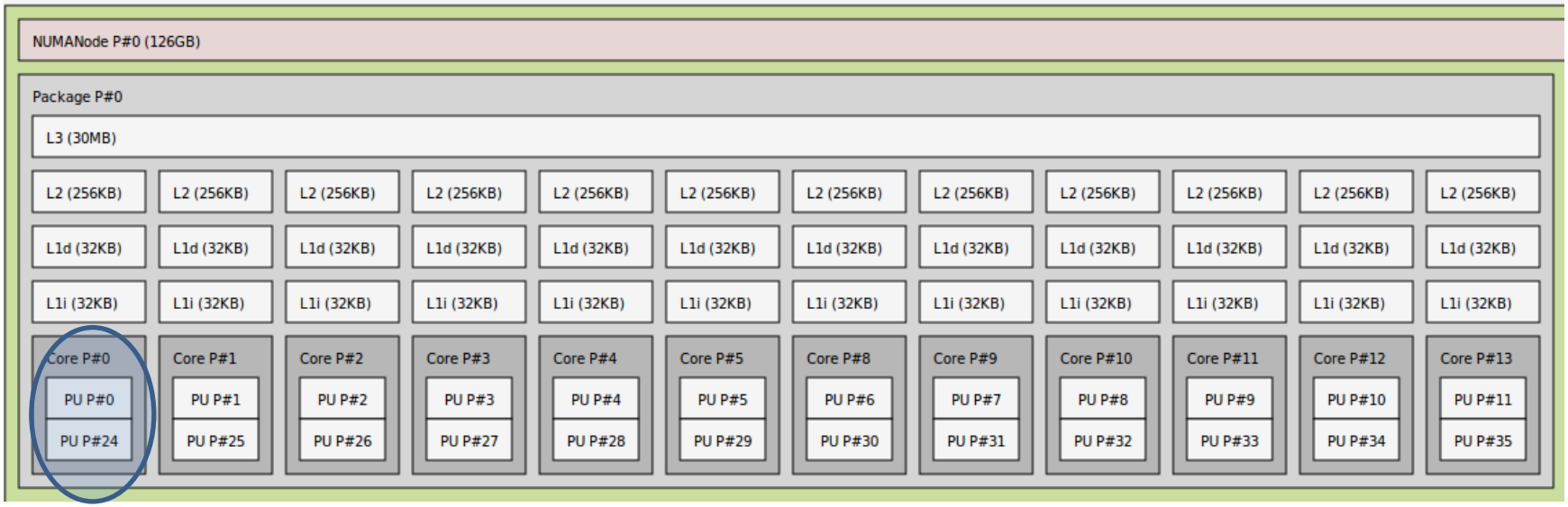


processor

CPU core

PU = processing unit (=control unit)

Processor Caches

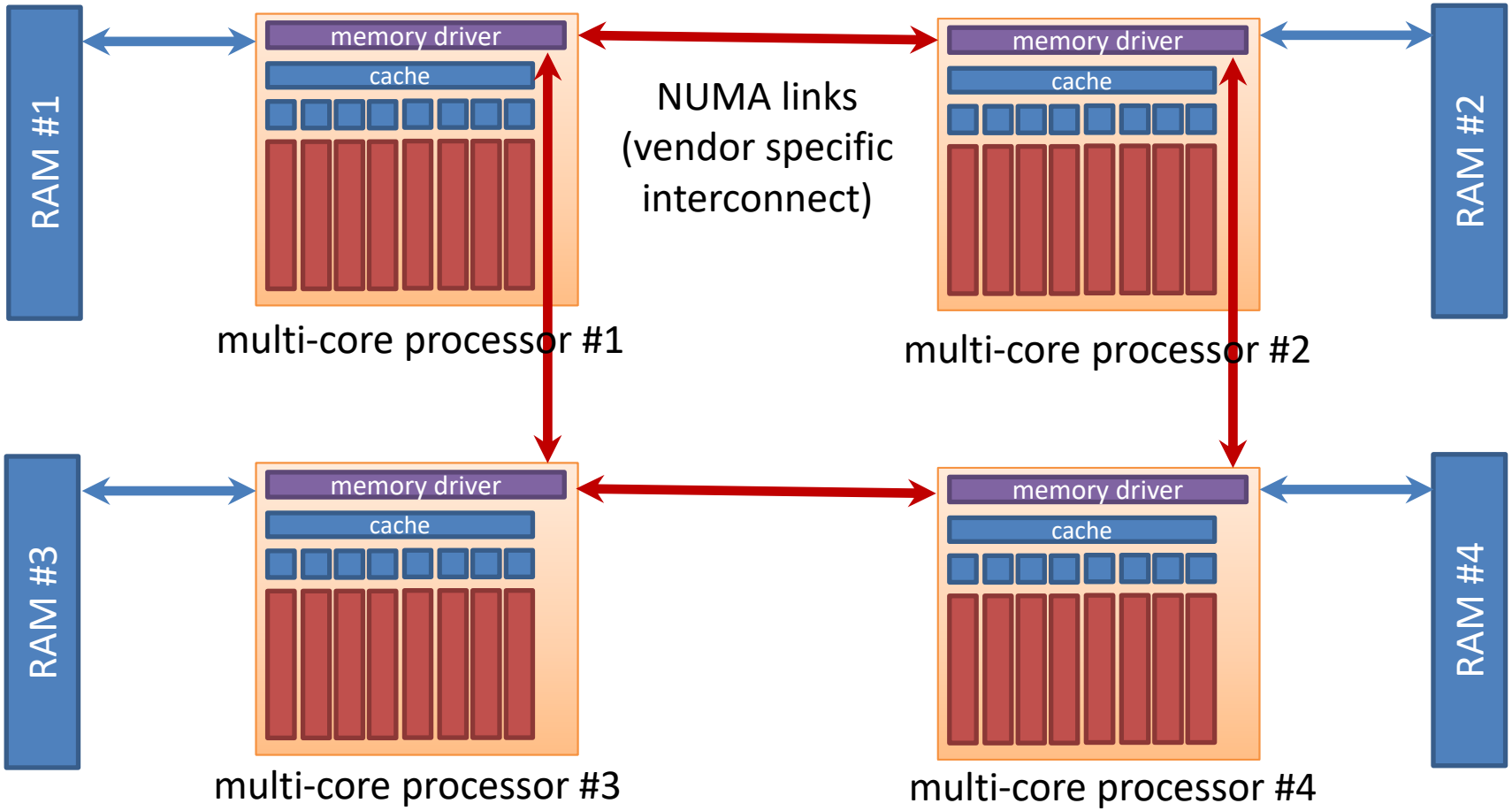


two processing units per CPU core (hyperthreading)

L1i – instruction cache,
L1d – data cache
L2, L3 – other caches

Speed:
L1d,L2d >> L2 > L3

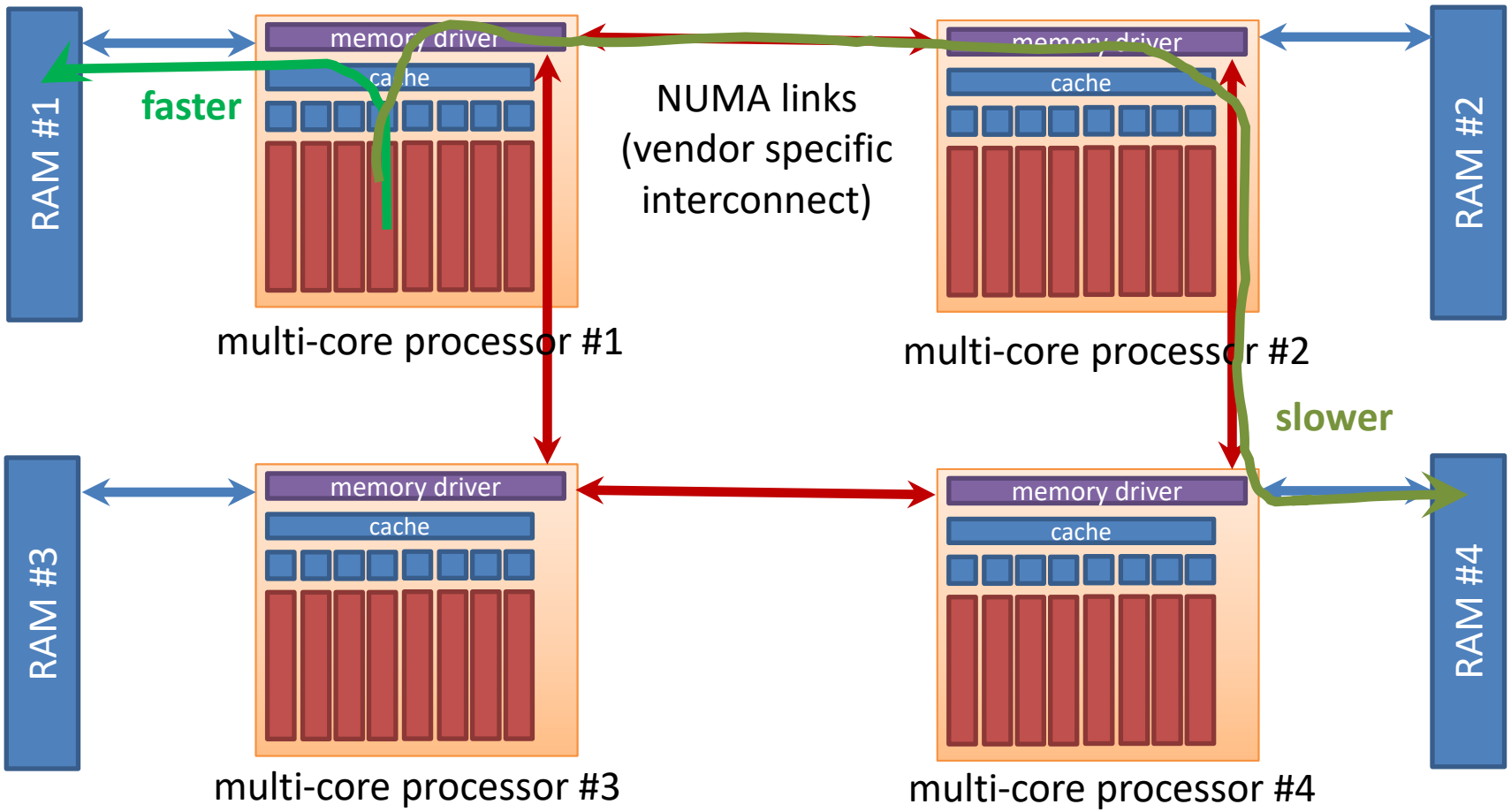
NUMA (Nonuniform Memory Access)



Compare communication between Processor#1 <> RAM#1 and Processor#1 <> RAM#4.

NUMA links can have various topologies to speedup CPU access to memory.

NUMA (Nonuniform Memory Access)



Compare communication between Processor#1 <> RAM#1 and Processor#1 <> RAM#4.

NUMA links can have various topologies to speedup CPU access to memory.

Exercise M1.1

1. Examine type and parameters of processor on your workstation (command lscpu, file /proc/cpus).
2. Examine NUMA topology on your workstation (command lstopo, module hwloc).
3. Does your CPU support hyperthreading?
4. What is a process?
5. What is difference between CPU intensive and data intensive tasks?
6. A parallel task is data intensive. Each its process works with different data sets. What is better for speeding up the calculation?
    1. To double number of CPU cores.
    2. To double number of processors (sockets).

Užitečné příkazy:

```
$ lscpu
$ lstopo              # module add hwloc
$ cat /proc/cpuinfo
$ ams-host            # Infinity
```

Numerical integration

parallelization using

OpenMP

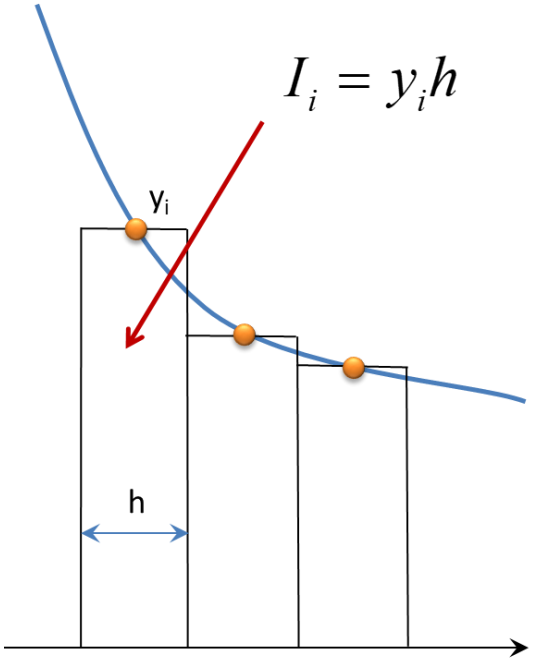
Sequential implementation

```
program integral

  implicit none
  integer(8)                  :: i
  integer(8)                  :: n
  double precision            :: rl,rr,h,v,y,x
  !-----------------------------------------------
  rl= 0.0d0
  rr= 1.0d0
  n = 2000000000
  h = (rr-rl)/n
  v = 0.0d0

  do i=1,n
    x = (i-0.5d0)*h + rl
    y = 4.0d0 / (1.0d0 + x**2)
    v = v + y*h
  end do

  write(*,*) 'integral = ',v

end program integral
```

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

rectangular method

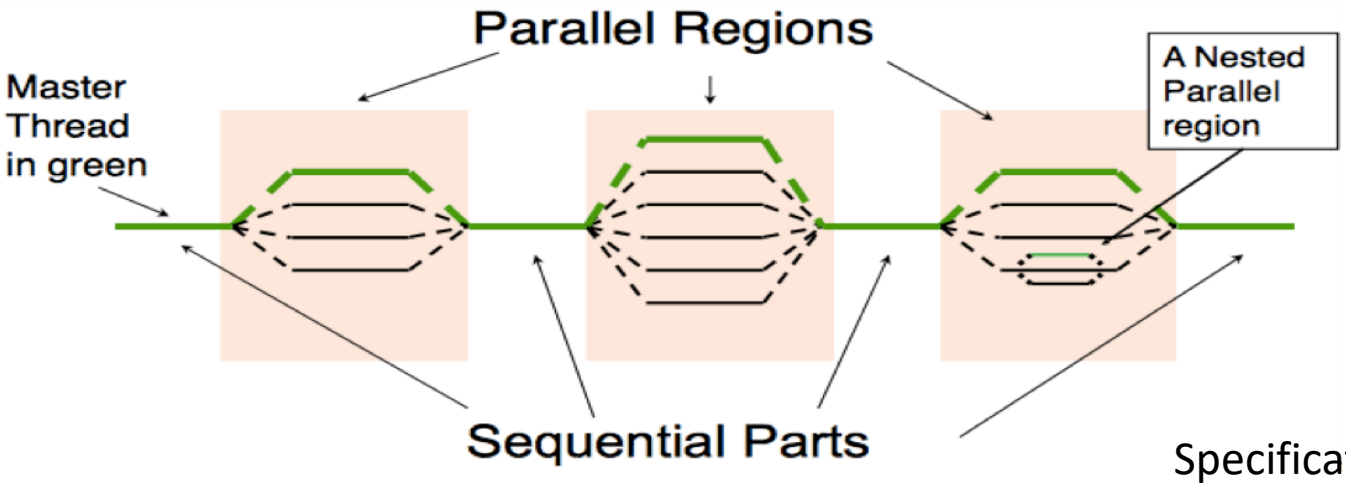
$$I_i = y_i h$$



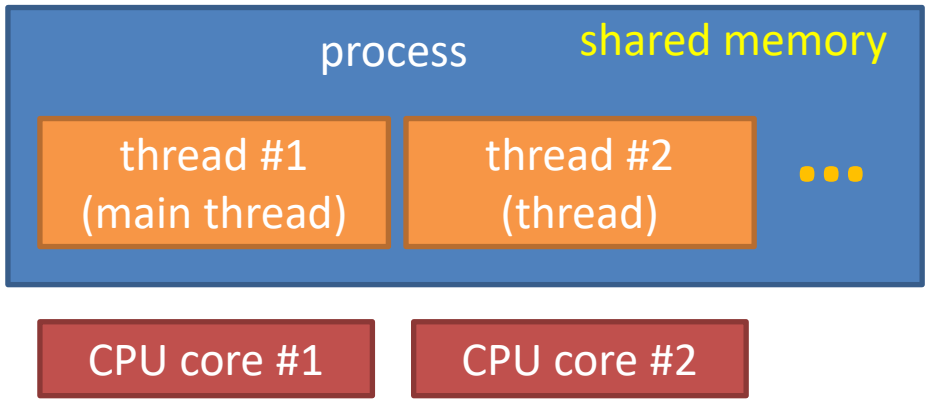
Parallelization - OpenMP

OpenMP is a system of **directives** for compiler and library procedures for parallel programming. It is a standard for programming computers with shared memory. OpenMP makes it easy to create multiple threaded programs in Fortran, C, and C++ programming languages.

www.wikipedia.org



Specifications: **www.openmp.org**



OpenMP is limited to SMP computing node and multicore CPU, alternatively their combination

OpenMP implementation

```
ncpu = 1
 !$ ncpu = omp_get_max_threads()
 write(*,*) 'Number of threads = ',ncpu

 !$omp parallel

 !$omp do private(i,x,y),reduction(+:v)
 do i=1,n
   x = (i-0.5d0)*h + rl
   y = 4.0d0/(1.0d0+x**2)
   v = v + y*d
 end do
 !$omp end do

 !$omp end parallel
 write(*,*) 'integral = ',v
```

OpenMP compilation

```
$ gfortran -O3  integral.f90 -o integral
$ ldd ./integral
        linux-vdso.so.1 =>
        libgfortran.so.3 => /usr/lib/x86_64-linux-gnu/libgfortran.so.3
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
        libquadmath.so.0 => /usr/lib/x86_64-linux-gnu/libquadmath.so.0
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
        /lib64/ld-linux-x86-64.so.2

$ gfortran -O3 -fopenmp integral.f90 -o integral
$ ldd ./integral
        linux-vdso.so.1 =>  (0x00007fff593ff000)
        libgfortran.so.3 => /usr/lib/x86_64-linux-gnu/libgfortran.so.3
        libgomp.so.1 => /usr/lib/x86_64-linux-gnu/libgomp.so.1
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6
        libquadmath.so.0 => /usr/lib/x86_64-linux-gnu/libquadmath.so.0
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
        /lib64/ld-linux-x86-64.so.2
```

OpenMP launch

```
$ export OMP_NUM_THREADS=4
$ ./integral
 Number of threads =                    4
 integral =    3.141592596529 5672
```

number of threads that the application can use

Note: if OMP_NUM_THREADS variable is not set, the maximum number of available CPU cores will be used
(however, on the WOLF cluster, the default value of OMP_NUM_THREADS variable is explicitly set to 1)

OpenMP and PBSPro batch system:

- based on configuration, batch system can set value of variable OMP_NUM_THREADS automatically (according to values of ncpus and mpiprocs in definition of block (chunk))
- value of variable OMP_NUM_THREADS can be set explicitly:

```
export OMP_NUM_THREADS=$PBS_NCPUS
```

number of assigned CPUs, jib must request only one computing node

Exercise M2.1

Source codes:
/home/kulhanek/Documents/C2115/code/integral/openmp

1. Compile the program **integral.f90** with optimization **-O3** and without OpenMP support.
2. Measure the application run time required for integration. Use **/usr/bin/time** program to measure the time.
3. Compile the program **integral.f90** with optimization **-O3** and OpenMP support turned on.
4. Determine the number of CPU cores on your computer (lscpu).
5. Run the program sequentially for 1, 2, 3, up to N threads, where N is the maximum available number of CPU cores. Measure the run time for each run. Write the obtained data in the following table and evaluate it.
6. Does the number of CPU cores affect the resulting value of integral? Why is that so?

measured time

N	T_{real} [s]	Speedup	CPU effectivity [%]
1	27.8	1.0	100.0
2	14.7	1.9	94.8
3	11.0	2.5	84.1
4	8.2	3.4	84.7

$$Speedup = \frac{T_{real}(N=1)}{T_{real}} \qquad CPUeffectivity = \frac{Speedup}{N}100$$

Numerical integration

parallelization using

MPI

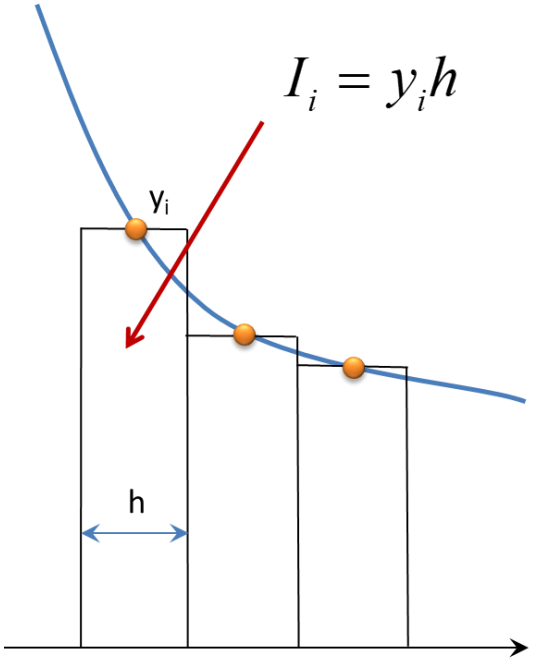
Sequential implementation

```
program integral

  implicit none
  integer(8)                  :: i
  integer(8)                  :: n
  double precision            :: rl,rr,h,v,y,x
  !-----------------------------------------------

  rl= 0.0d0
  rr= 1.0d0
  n = 2000000000
  h = (rr-rl)/n
  v = 0.0d0

  do i=1,n
    x = (i-0.5d0)*h + rl
    y = 4.0d0 / (1.0d0 + x**2)
    v = v + y*h
  end do

  write(*,*) 'integral = ',v

end program integral
```

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

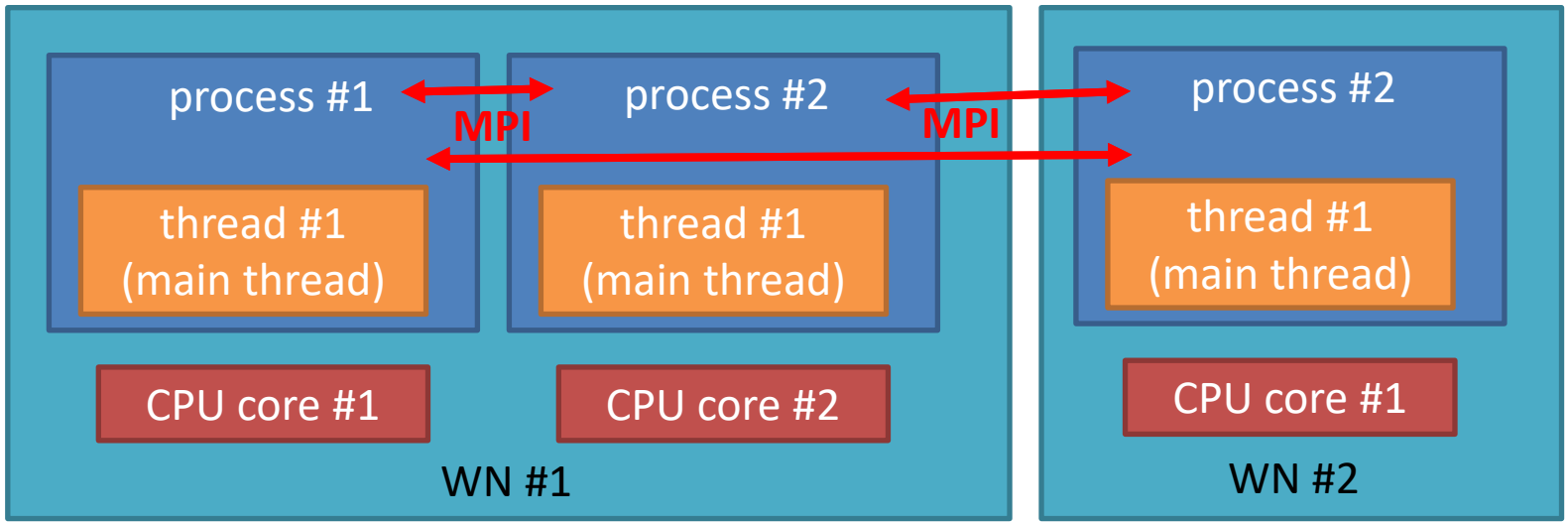
rectangular method

$$I_i = y_i h$$



Parallelization - MPI

Message Passing Interface (hereinafter referred to as MPI) is a library implementing the specification (protocol) of the same name to support parallel solving of computational problems in computer clusters. Specifically, it is an application development interface (API) based on messaging between individual nodes. These are both point-to-point messages and global operations. The library supports both shared and distributed memory architectures.



- MPI jobs can be run on one or more WN
- MPI can be combined with OpenMP.

MPI messages

MPI implementation

```
do i=1,n
   x = (i-0.5d0)*h + rl
   y = 4.0d0 / (1.0d0 + x**2)
   v = v + y*h
 end do
```

- Problem must be divided so that each process calculates part of loop.
- Processes do not share memory, so information about division of the task and the transfer of partial results must be solved by **passing messages**.
- One of the processes has the role of an organizer who manages the other processes and communicates with the environment.

Source codes:
/home/kulhanek/Documents/C2115/code/integral/mpi

Recorded comment of the file **integral.f90**

MPI compilation

activation of MPI development environment (OpenMPI)

```
$ module add openmpi:3.1.5-gcc
$ mpif90 -O3  integral.f90 -o integral
```

Fortran compiler with MPI support (compiled internally using gfortan)

```
[kulhanek@wolf mpi]$ ldd integral
        linux-vdso.so.1 (0x00007ffe24944000)
        libmpi_mpifh.so.40 => /software/ncbr/softrepo/devel/openmpi/3.1.5-gcc/x86_64/para/lib/libmpi_mpifh.so.40 (0x0000149914f34000)
        libgfortran.so.4 => /usr/lib/x86_64-linux-gnu/libgfortran.so.4 (0x0000149914b55000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x0000149914764000)
        libmpi.so.40 => /software/ncbr/softrepo/devel/openmpi/3.1.5-gcc/x86_64/para/lib/libmpi.so.40 (0x0000149914453000)
        libopen-rte.so.40 => /software/ncbr/softrepo/devel/openmpi/3.1.5-gcc/x86_64/para/lib/libopen-rte.so.40 (0x000014991419e000)
        libopen-pal.so.40 => /software/ncbr/softrepo/devel/openmpi/3.1.5-gcc/x86_64/para/lib/libopen-pal.so.40 (0x0000149913ed9000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x0000149913cd5000)
        librt.so.1 => /lib/x86_64-linux-gnu/librt.so.1 (0x0000149913acd000)
        libutil.so.1 => /lib/x86_64-linux-gnu/libutil.so.1 (0x00001499138ca000)
        libz.so.1 => /lib/x86_64-linux-gnu/libz.so.1 (0x00001499136ad000)
        libhwloc.so.5 => /software/ncbr/softrepo/devel/hwloc/1.11.13/x86_64/single/lib/libhwloc.so.5 (0x0000149913470000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00001499130d2000)
        libnuma.so.1 => /usr/lib/x86_64-linux-gnu/libnuma.so.1 (0x0000149912ec7000)
        libxml2.so.2 => /usr/lib/x86_64-linux-gnu/libxml2.so.2 (0x0000149912b06000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00001499128e7000)
        libquadmath.so.0 => /usr/lib/x86_64-linux-gnu/libquadmath.so.0 (0x00001499126a7000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x000014991248f000)
        /lib64/ld-linux-x86-64.so.2 (0x000014991538e000)
        libicuuc.so.60 => /usr/lib/x86_64-linux-gnu/libicuuc.so.60 (0x00001499120d7000)
        liblzma.so.5 => /lib/x86_64-linux-gnu/liblzma.so.5 (0x0000149911eb1000)
        libicudata.so.60 => /usr/lib/x86_64-linux-gnu/libicudata.so.60 (0x0000149910308000)
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x000014990ff7f000)
```

MPI launch

number of process which the application uses for the calculation

```
$ mpirun –np 2 ./integral
```

file that contains a list of nodes on which processes run

```
$ mpirun –np 2 –machinefile nodes ./integral
```

```
wolf01 slots=2
wolf02 slots=3
```

CPU count

name of computational node

Requirements:

- ssh without password
- Application must be in the same path on all nodes, where the processes are run

MPI and PBSPro batch system:

- Correctly configured MPI is able to load assigned resources automatically from the batch system
- In manual mode, it is possible to use file with computing nodes (variable PBS_NODEFILE) and number of assigned CPUS in variable PBS_NCPU

Exercise M3.1

Source codes:
/home/kulhanek/Documents/C2115/code/integral/mpi

1. Compile the program **integral.f90** with optimization **-O3** and MPI support.
2. Run the program **integral** sequentially for 1, 2, 3, to N processes, where N is the maximum available number of CPU cores. Measure execution time for each run. Write the obtained data in a table and evaluate it as in exercise M2.1.
3. Run the program **integral** sequentially for 1, 2, 4, 8 to N processes (multiples of 2), where N is the maximum available number of CPUs on the two nodes of the WOLF cluster (select unoccupied nodes). Measure execution time for each run. Write the obtained data in table and evaluate it as in exercise M2.1. In another terminal, monitor running processes on both compute nodes with the top command.
4. Does the number of CPU processes affect the resulting value of integral? Why is that so?

Parallel jobs

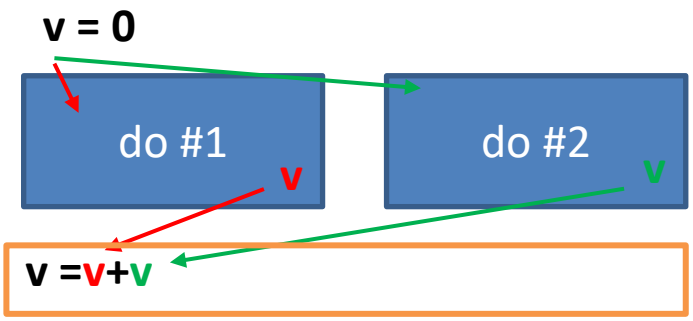
Concurrency of errors, numerical errors, running applications, efficiency of parallelization (Amdahl's law)

Parallelization pitfalls, concurrency errors

Parallel jobs are prone to **concurrency errors** (race condition). Therefore, it is necessary to pay considerable attention to this pitfall when designing parallel application.

```
v = 0.0d0
 !$omp do private(i,x,y),reduction(+:v)
 do i=1,n
   x = (i-0.5d0)*h + rl
   y = 4.0d0/(1.0d0+x**2)
   v = v + y*d
 end do
 !$omp end do
```

!!! key for functionality !!!

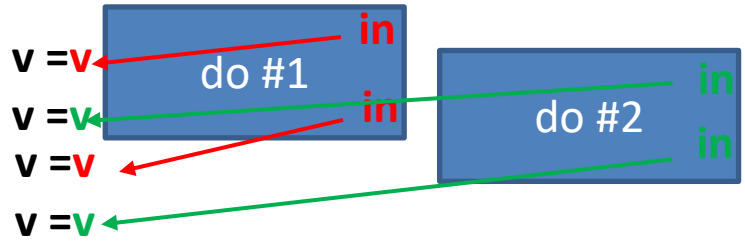


parallel run (two threads)

barrier (first wait for all threads to finish and only then calculate the result)

Without barrier the variable "v" will be used in an undefined order with the possibility of overwriting intermediate results.

Simplified (for illustration):



Parallelization pitfalls, numerical errors

The order in which arithmetic operations are performed may change over time, so you can obtain **different result** using different number of CPUs or repeating the same parallel task (typical for jobs with dynamic load balancing).

Jobs with dynamic load balancing changes the division of the problem into individual CPUs (e.g., atoms which will be processed by the CPU in molecular dynamics) so that the task runs as fast as possible.

```
[kulhanek@wolf openmp]$ OMP_NUM_THREADS=1 ./integral
 Number of threads =                    1
 integral =    3.14159265358 94521
[kulhanek@wolf openmp]$ OMP_NUM_THREADS=2 ./integral
 Number of threads =                    2
 integral =    3.14159265358 88206
[kulhanek@wolf openmp]$ OMP_NUM_THREADS=4 ./integral
 Number of threads =                    4
 integral =    3.14159265358 98944
```

Running parallel tasks

- **For new jobs, always verify that you are actually running a parallel version of the application** (program).
- If the task does not run in parallel, verify that you are running the correct version of the program and in the correct way (OpenMP vs MPI).

Useful commands:

- **Ldd** lists the dynamic libraries that the program uses
- **top** lists the running processes
- **pstree** lists the running processes

Running parallel tasks, cont.

OpenMP

```
bash(17479)──integral(21331)─┬─{integral}(21332)
                             ├─{integral}(21333)
                             └─{integral}(21334)
```

1 process and **4 threads**

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
21331 kulhanek  20   0   29284   1592   1452 R 399.3  0.0   1:40.14 integral
 3469 root      20   0       0      0      0 S   0.3  0.0   4:53.43 afsd
```

MPI

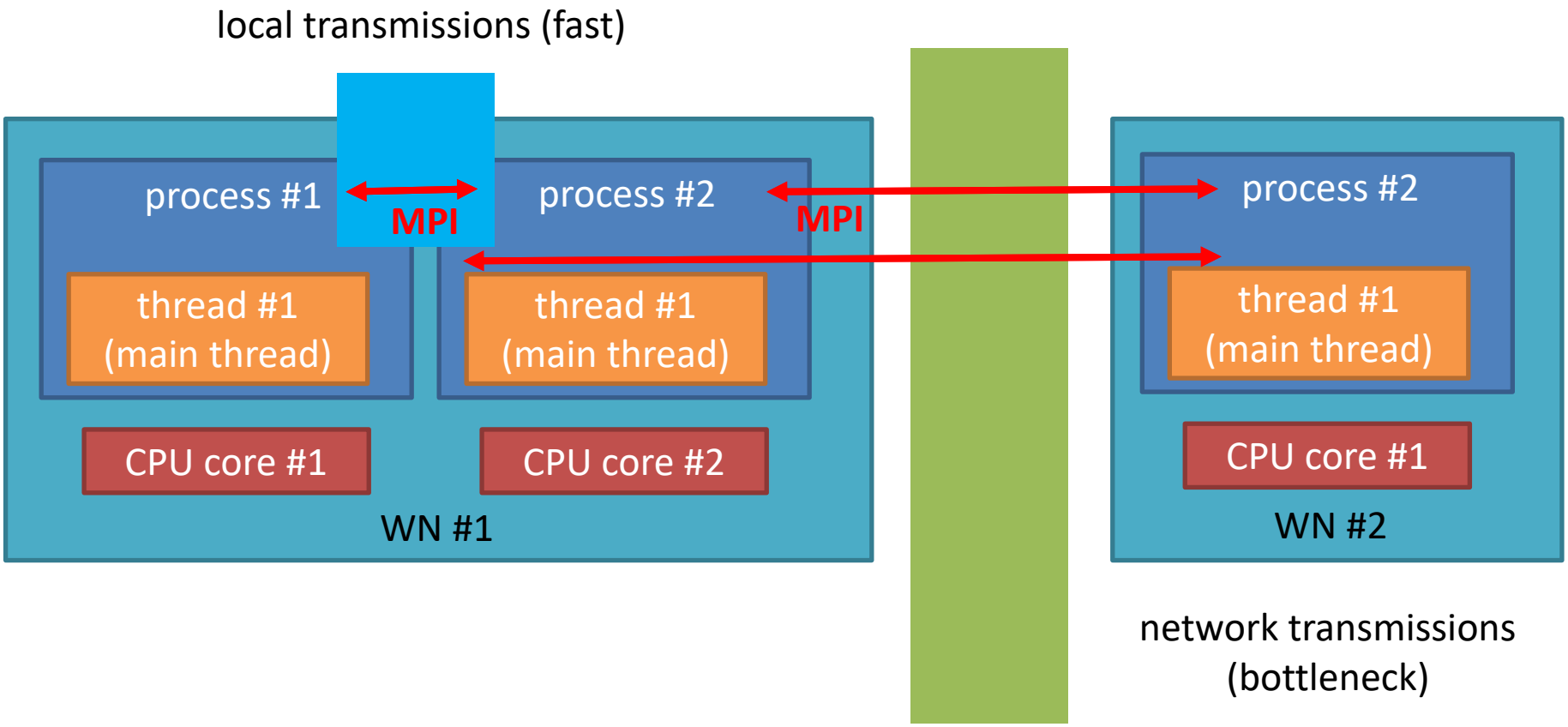
```
bash(17479)──mpirun(23938)─┬─integral(23942)─┬─{integral}(23946)
                           │                 ├─{integral}(23948)
                           │                 └─{integral}(23956)
                           ├─integral(23943)─┬─{integral}(23949)
                           │                 ├─{integral}(23952)
                           │                 └─{integral}(23955)
                           ├─integral(23944)─┬─{integral}(23947)
                           │                 ├─{integral}(23950)
                           │                 └─{integral}(23957)
                           ├─integral(23945)─┬─{integral}(23951)
                           │                 ├─{integral}(23953)
                           │                 └─{integral}(23954)
                           ├─{mpirun}(23939)
                           ├─{mpirun}(23940)
                           └─{mpirun}(23941)
```

4 processes (+ service threads used by MPI)

```
  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
23942 kulhanek  20   0  375740  15796  12576 R 100.0  0.1   0:15.48 integral
23943 kulhanek  20   0  375740  15680  12456 R 100.0  0.1   0:15.48 integral
23944 kulhanek  20   0  375740  15724  12508 R 100.0  0.1   0:15.48 integral
23945 kulhanek  20   0  375740  15736  12512 R 100.0  0.1   0:15.48 integral
```

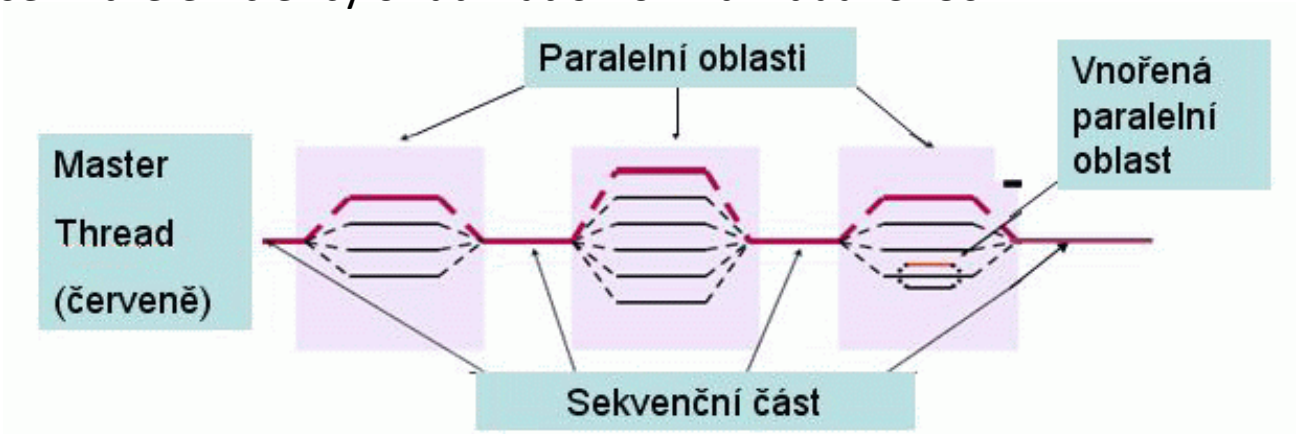
Running parallel tasks, cover.

- Avoid running parallel jobs on multiple nodes (MPI).
- If necessary, use the fastest possible network connection (**Infiniband** vs Ethernet)



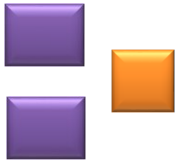
Parallelization efficiency

Sequential parts are an integral (however unwanted) parts of a parallel application. The running of this part is not accelerated with increasing number of CPUs, which in turn leads to a decrease in the efficiency of utilization of individual CPUs.



1 CPU

2 CPU



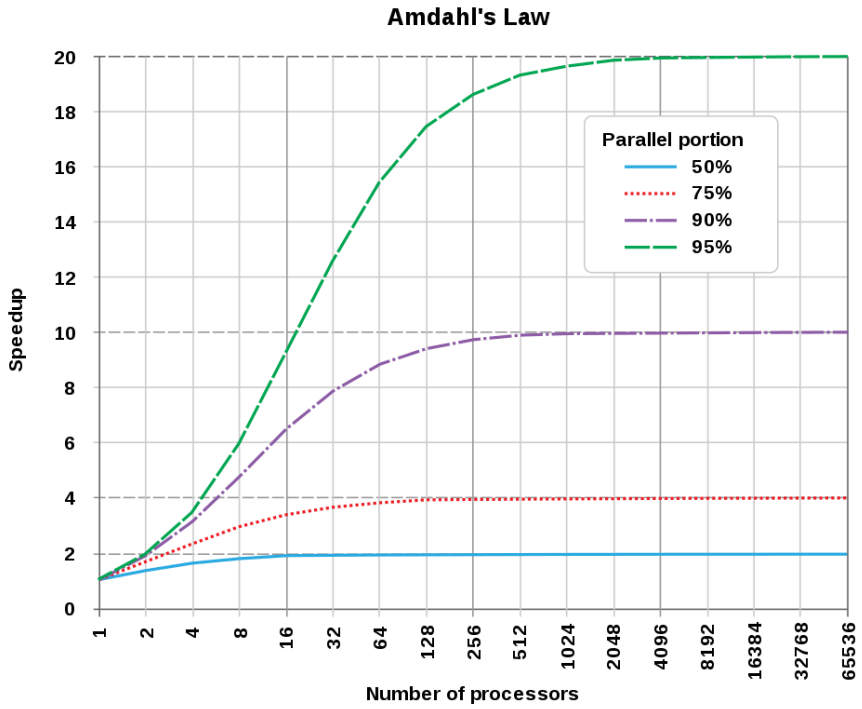
stays the same

shortens

Amdahl's law

Amdahl's law expresses the maximum expected acceleration of the parallel job.

https://en.wikipedia.org/wiki/Amdahl's_law



Theoretical acceleration:

$$S_{speedup} = \frac{1}{(1-p) + \frac{p}{N}}$$

p - parallel part of the program (fraction)
N - number of CPUs

- **ALWAYS verify the efficiency of the parallel application run** (especially for new problems or with change of their size).
- Verification is done according to exercise L13.M2.1, while the acceptable minimum efficiency per CPU is about 80%.

Exercise M4.1

Source codes:
/home/kulhanek/Documents/C2115/code/integral/openmp

1. Compile the program **integral_rc.f90** with optimization **-O3** and support for OpenMP.
2. Successively run the program **integral_rc** for 1, 2, 3, up to N threads, where N is the maximum available number of CPU cores. How does the result change and why?
3. Run the program repeatedly **integral_rc** on 2 threads. How does the result change and why?