ECMWF – DESTINATION EARTH

MASSIVELY PARALLEL COMPUTING FOR NWP AND CLIMATE

Andreas Müller, ECMWF

Funded by the European Union
Destination Earth
implemented by ECMWF esa EUMETSAT

Announcement

Virtual machines (traininglab**.ecmwf.europeanweather.cloud) will be deleted on Monday morning!

Option 1:

- inside Jupyterlab start X11 Desktop connection
- open Web Browser (Earth icon in the dock)
- log in to some online account that you own (Google drive, Dropbox, webmail, ...)
- drag and drop files into your online account

Option 2:

- log in to Jupyterlab from your personal laptop
- drag and drop from the file browser in Jupyterlab (left panel) to your laptop

Overview

- Why do scientists need to know so much about computer hardware?
- What do we need to be aware of to write efficient code?
- How good are we?

Why do we as scientists need to know so much about computer hardware?

Why do we as scientists need to know so much about computer hardware?

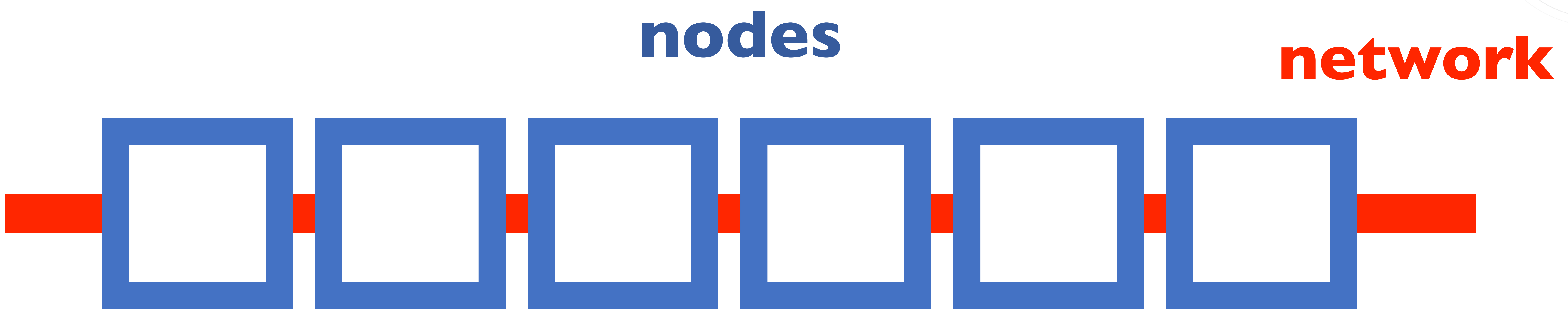
- Excuse 1: let the software engineers take care of it
- Response: software engineers cannot do everything because they do not know about different numerical methods

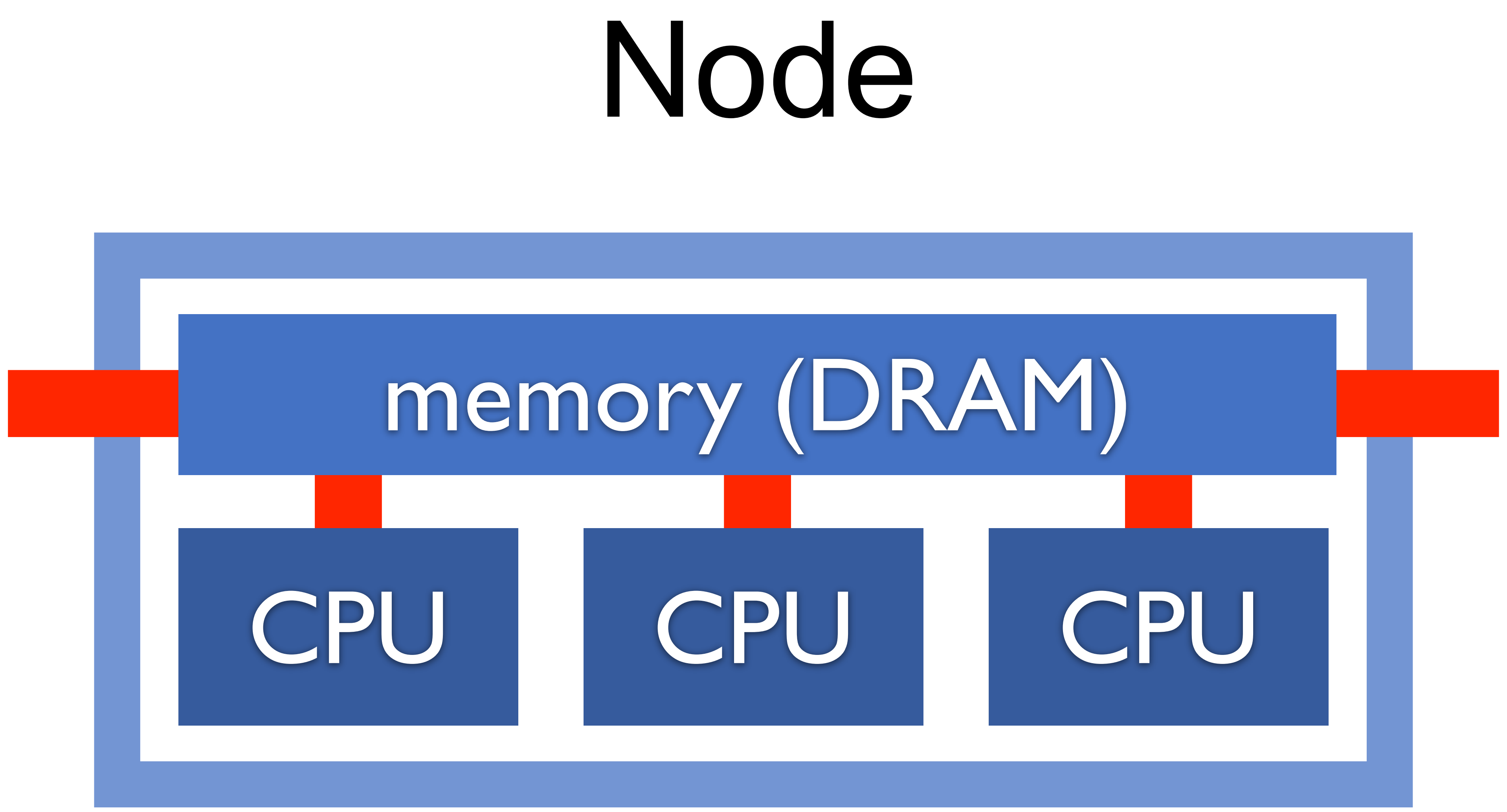
Why do we as scientists need to know so much about computer hardware?

- Excuse 1: let the software engineers take care of it
- Response: software engineers cannot do everything because they do not know about different numerical methods
- Excuse 2: just buy a faster computer if the code is not fast enough
- Response: we (and the environment) cannot afford wasting that much energy!

computer	electricity cost per year
ECMWF	~5 million £
fastest current supercomputers	~20 million $

Supercomputer/Cluster

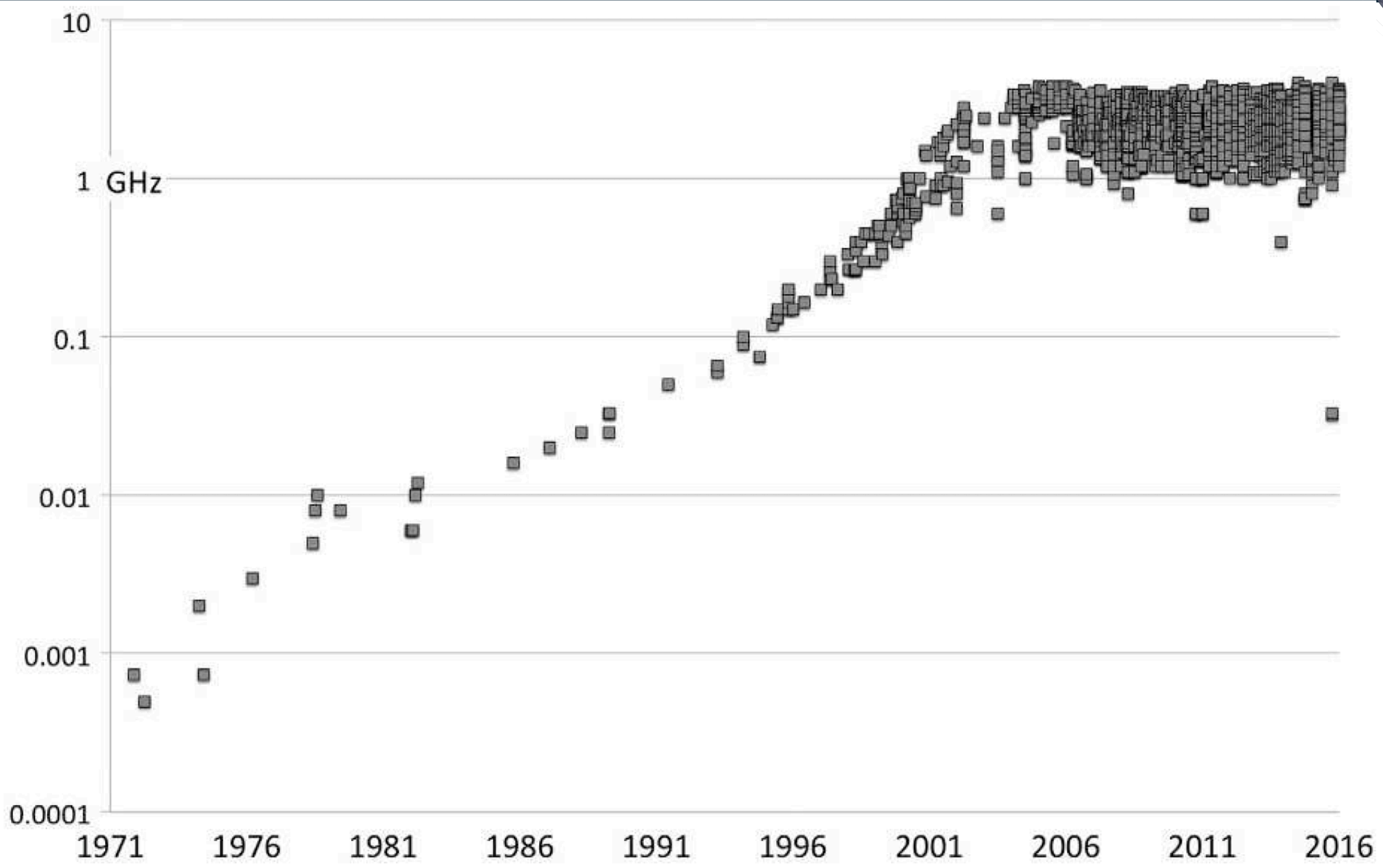




CPU

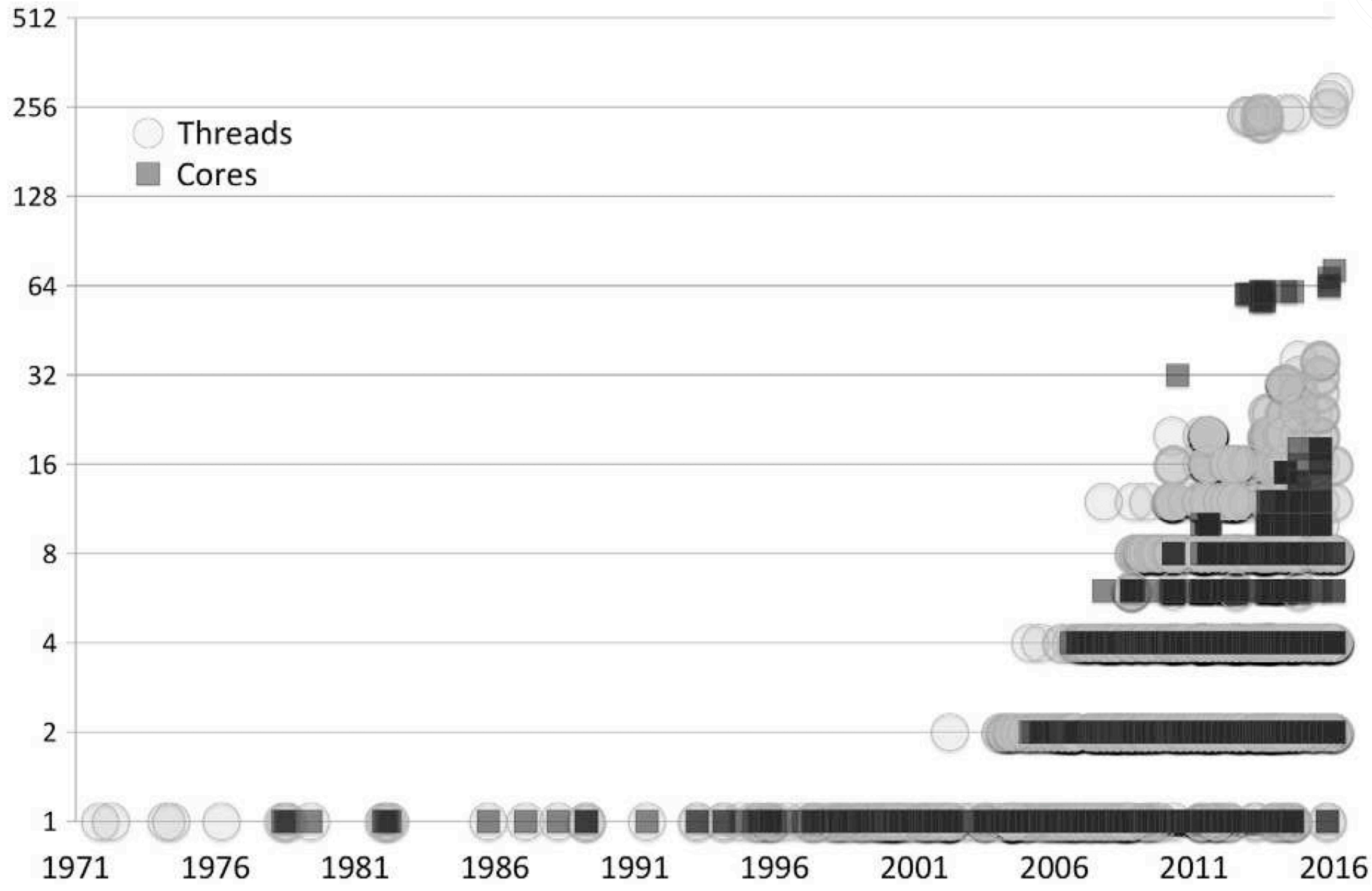
central processing unit; does one instruction like c=a+b per clock cycle

CPU clock rate over time



source: James Reinders, Intel Xeon Phi

Number of cores per chip over time



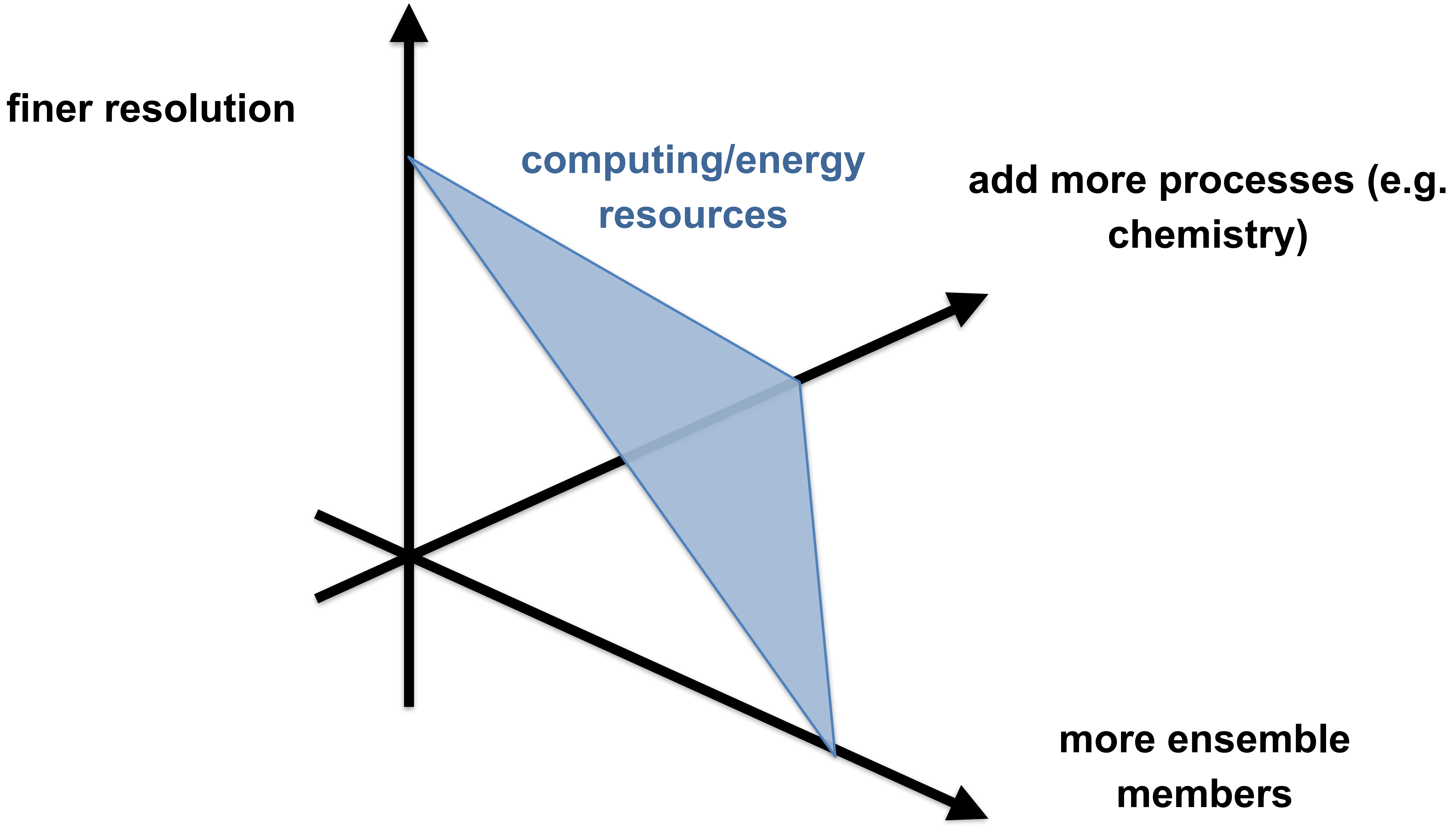
source: James Reinders, Intel Xeon Phi

http://top500.org

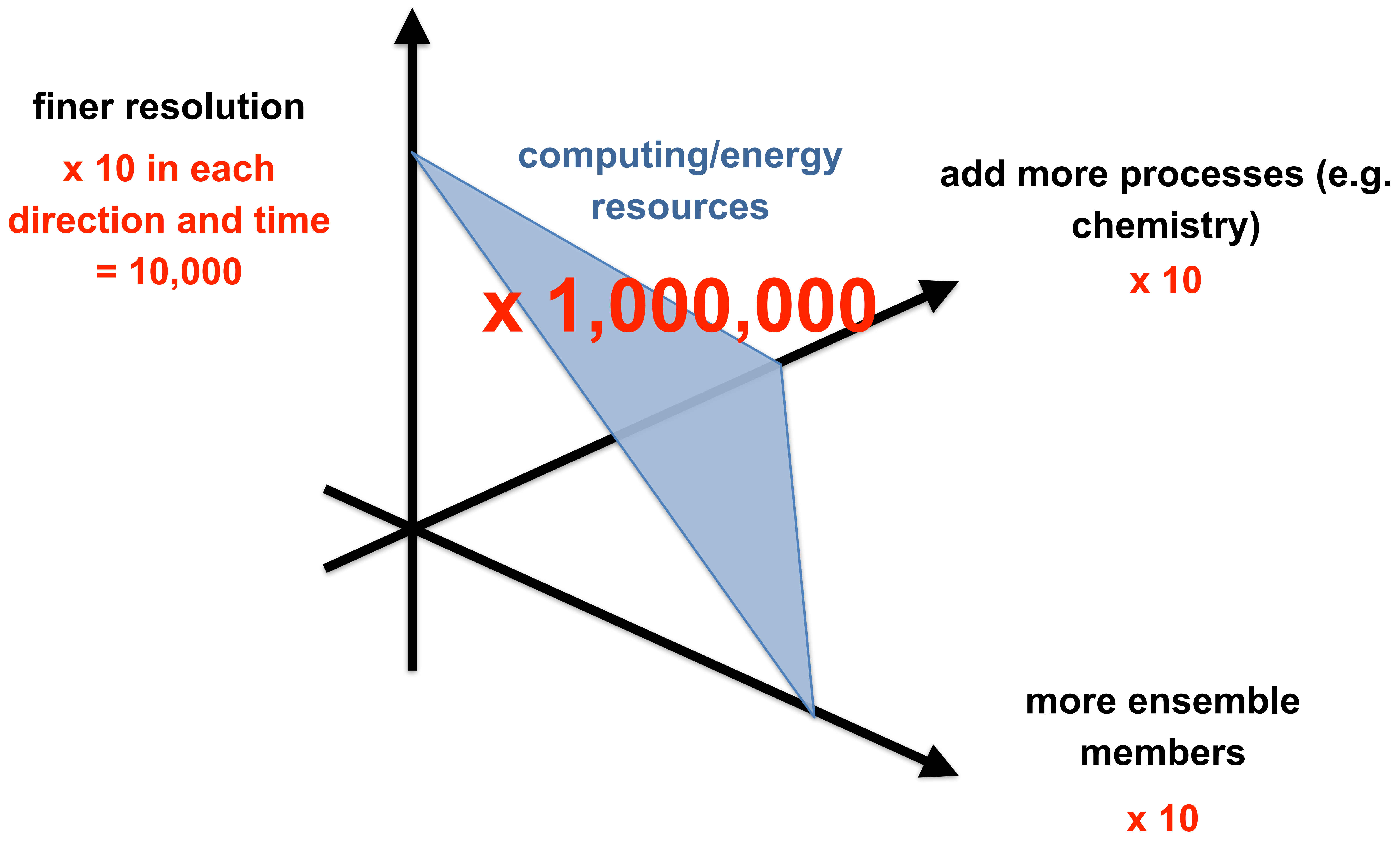
Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,699,904	1,206.00	1,714.81	22,786
2	**Aurora** - HPE Cray EX - Intel Exascale Compute Blade, Xeon CPU Max 9470 52C 2.4GHz, Intel Data Center GPU Max, Slingshot-11, Intel DOE/SC/Argonne National Laboratory United States	9,264,128	1,012.00	1,980.01	38,698
3	**Eagle** - Microsoft NDv5, Xeon Platinum 8480C 48C 2GHz, NVIDIA H100, NVIDIA Infiniband NDR, Microsoft Azure Microsoft Azure United States	2,073,600	561.20	846.84	
4	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	53[illegible]	[illegible]
5	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	2,752,704	379.70	531.51	7,107

TOP500 The List.

What comes next?

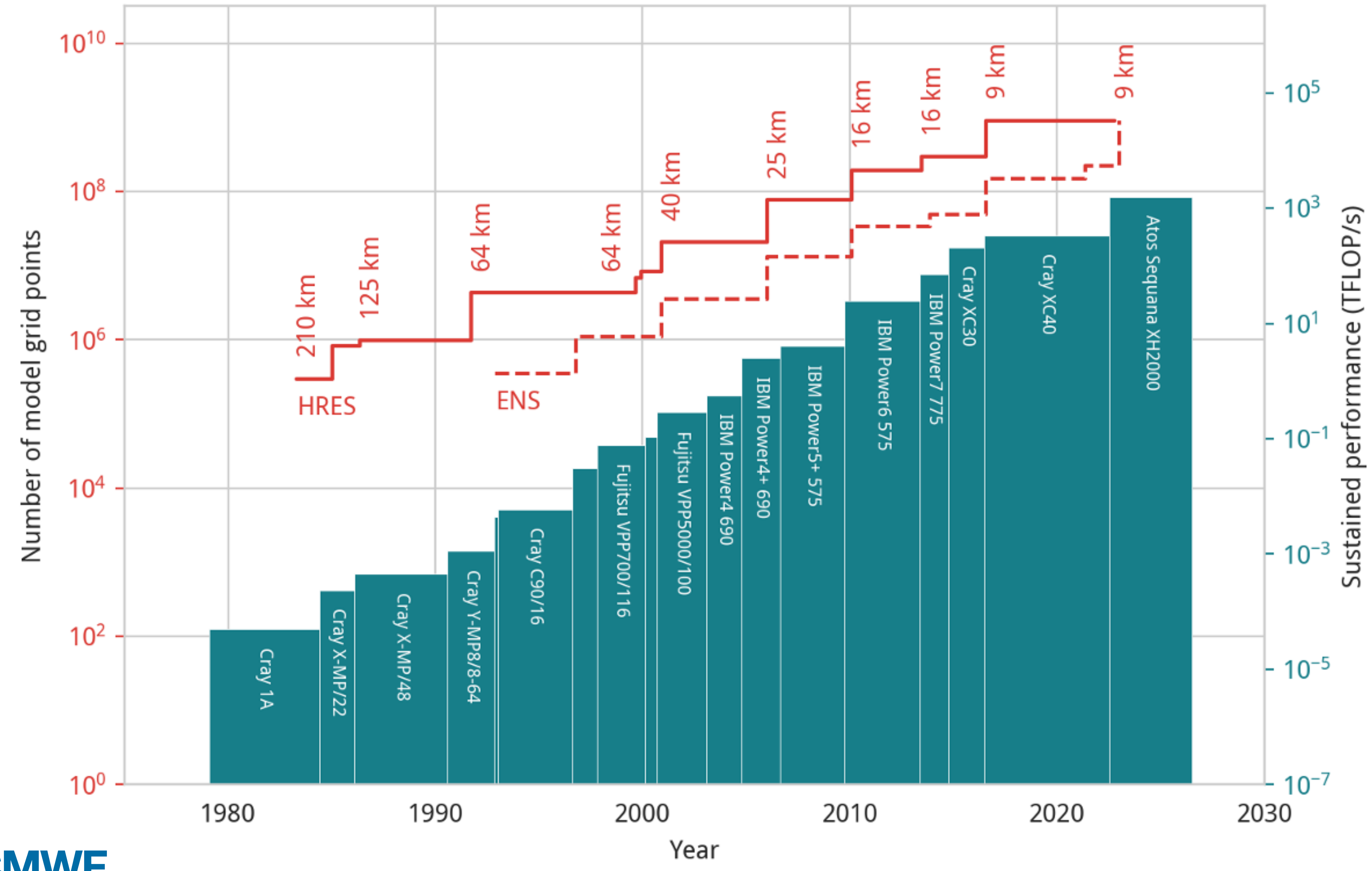


What comes next?



ECMWF

Computing at ECMWF



What do we need to be aware of to write efficient code?

Recommendations

-
-
-

Libraries

- there are well optimised libraries for many tasks
- BLAS for vector-matrix product or matrix-matrix product (if matrices are large)
- Lapack for matrix factorisation (e.g. LU decomposition)
- FFTW for Fast Fourier Transform
- some hardware vendors have special math libraries, e.g. MKL by Intel
- there are some cases in which libraries are fairly slow (e.g. BLAS with very small matrices)

Recommendations

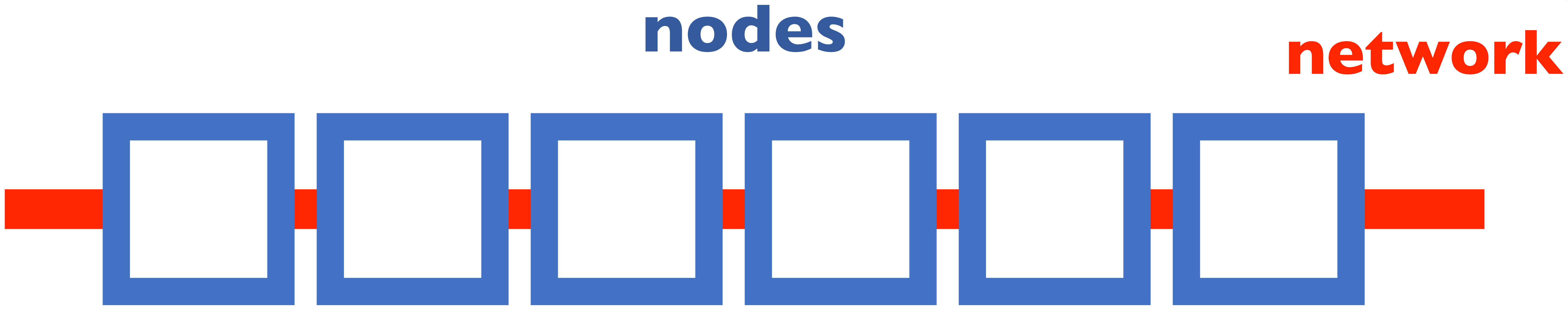
- **try to use well optimized libraries**

Compiler optimisation

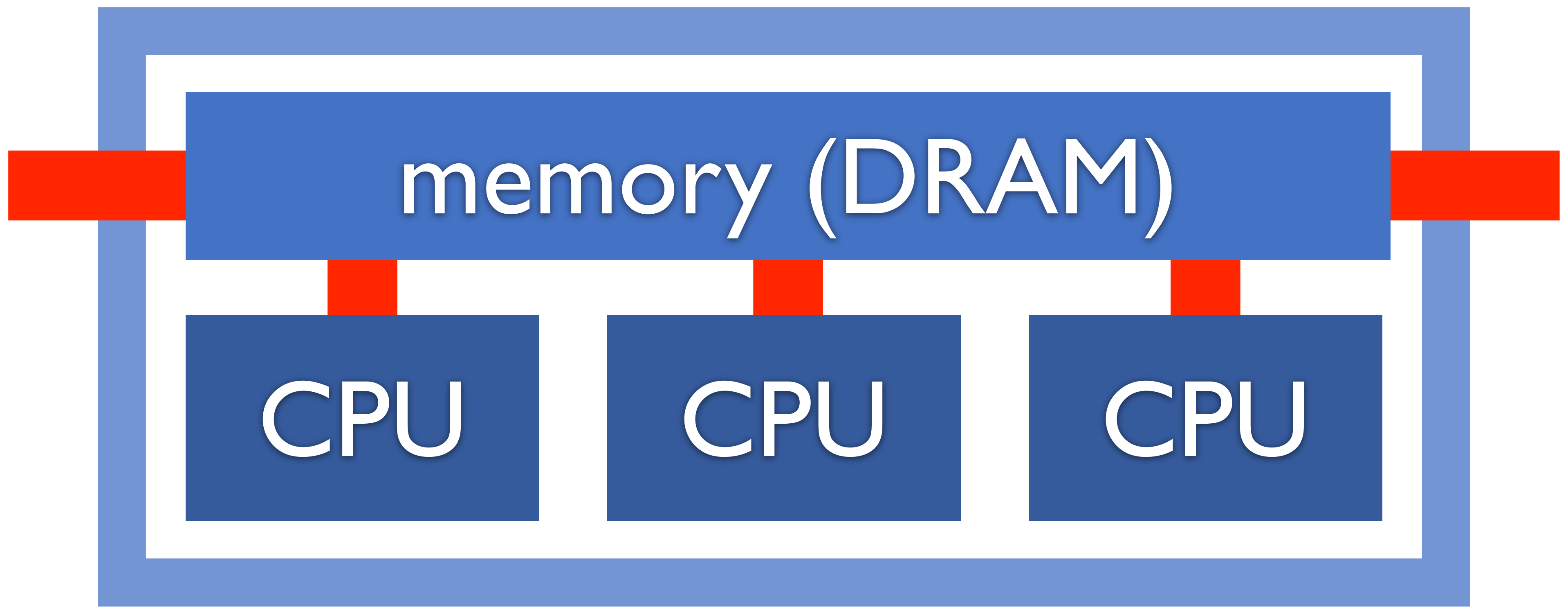
- compilers have optimisation flag -On (O0: no optimisation, O3: strong compiler optimisation)
- O3 is usually much faster than O2, but it can also be slower than O2
- O3 can produce completely wrong results!
- you can use different compiler flags for different files
- different compiler versions can have very different performance
- check compiler messages (Intel: ifort -O2 -qopt-report=2 code.f90 -o program)
- make sure that your code runs correctly with different compilers

Recommendations

- try to use well optimized libraries
- **try to use compiler optimisation (be careful!)**



Node



Bottlenecks

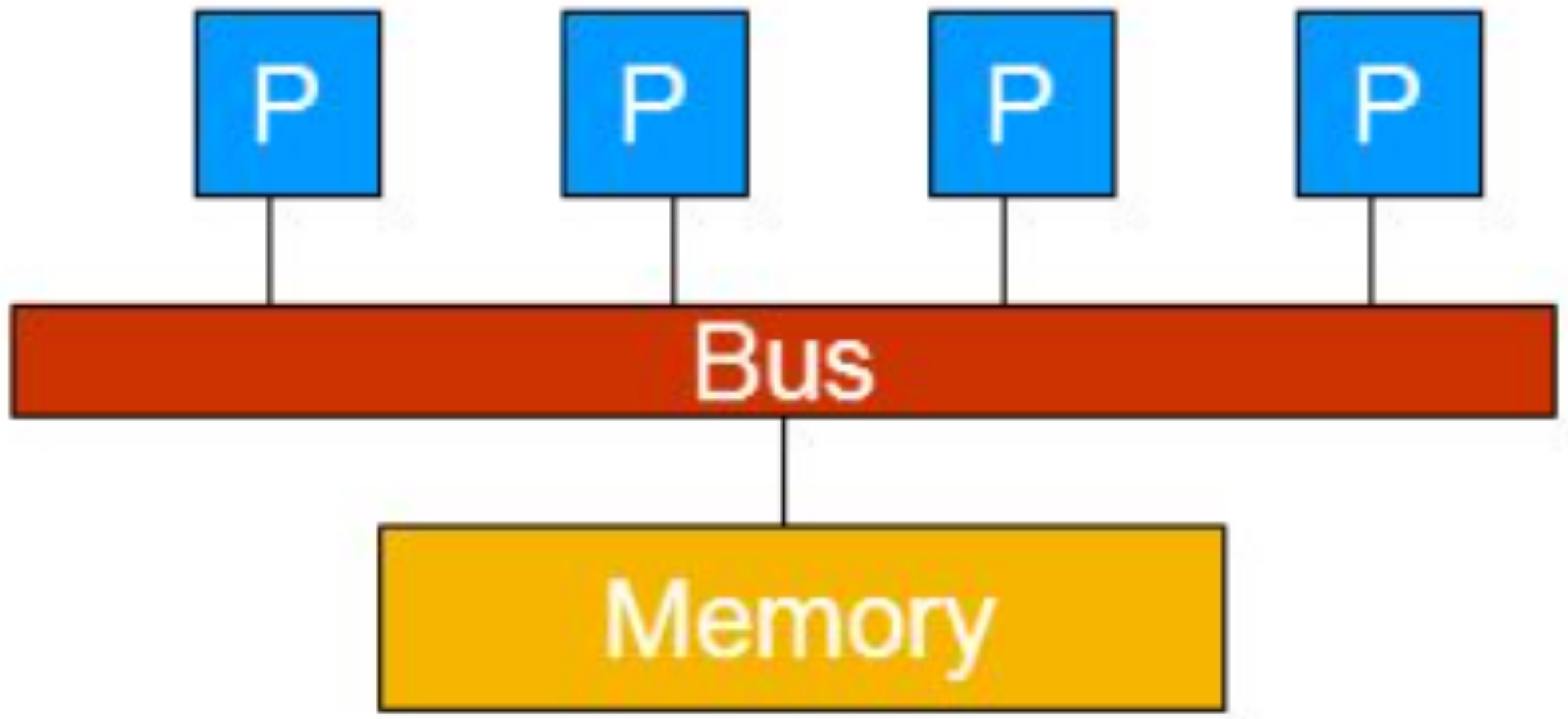
- network (connection between nodes)
- connection between DRAM and processor

Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- **avoid unnecessary computation and communication**

Shared memory: OpenMP

- many threads of a process run on a single node
- all threads can access the same data
- data may be physically distributed, but logically shared



without OpenMP:

```
real, dimension(N) :: a,b
integer :: i,N
do i=1,N
 a(i) = a(i) + b(i)
end do
```

with OpenMP:

```
real, dimension(N) :: a,b
integer :: i,N
!$omp parallel do private(i)
do i=1,N
 a(i) = a(i) + b(i)
end do
!$omp end parallel do
```

Shared memory: OpenMP

faster for bigger codes:

```
real, dimension(N) :: a,b
integer :: i, N, iStart, iEnd,
 myid, numthreads
!$omp parallel private(i,iStart,iEnd)
myid = omp_get_thread_num()
numthreads = omp_get_num_threads()
iStart = ...
iEnd = ...
do i=iStart,iEnd
 a(i) = a(i) + b(i)
end do
!$omp end parallel
```

without OpenMP:

```
real, dimension(N) :: a,b
integer :: i,N
do i=1,N
 a(i) = a(i) + b(i)
end do
```

with OpenMP:

```
real, dimension(N) :: a,b
integer :: i,N
!$omp parallel do private(i)
do i=1,N
 a(i) = a(i) + b(i)
end do
!$omp end parallel do
```

Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- **give each thread as much work as possible**

Example 2: race conditions

without OpenMP:

```
real, dimension(N) :: a
real :: sum
integer :: i,N
do i=1,N
 sum = sum + a(i)
end do
```

with OpenMP (wrong!):

```
real, dimension(N) :: a
real :: sum
integer :: i,N
!$omp parallel do private(i)
do i=1,N
 sum = sum + a(i)
end do
!$omp end parallel do
```

working, but slow:

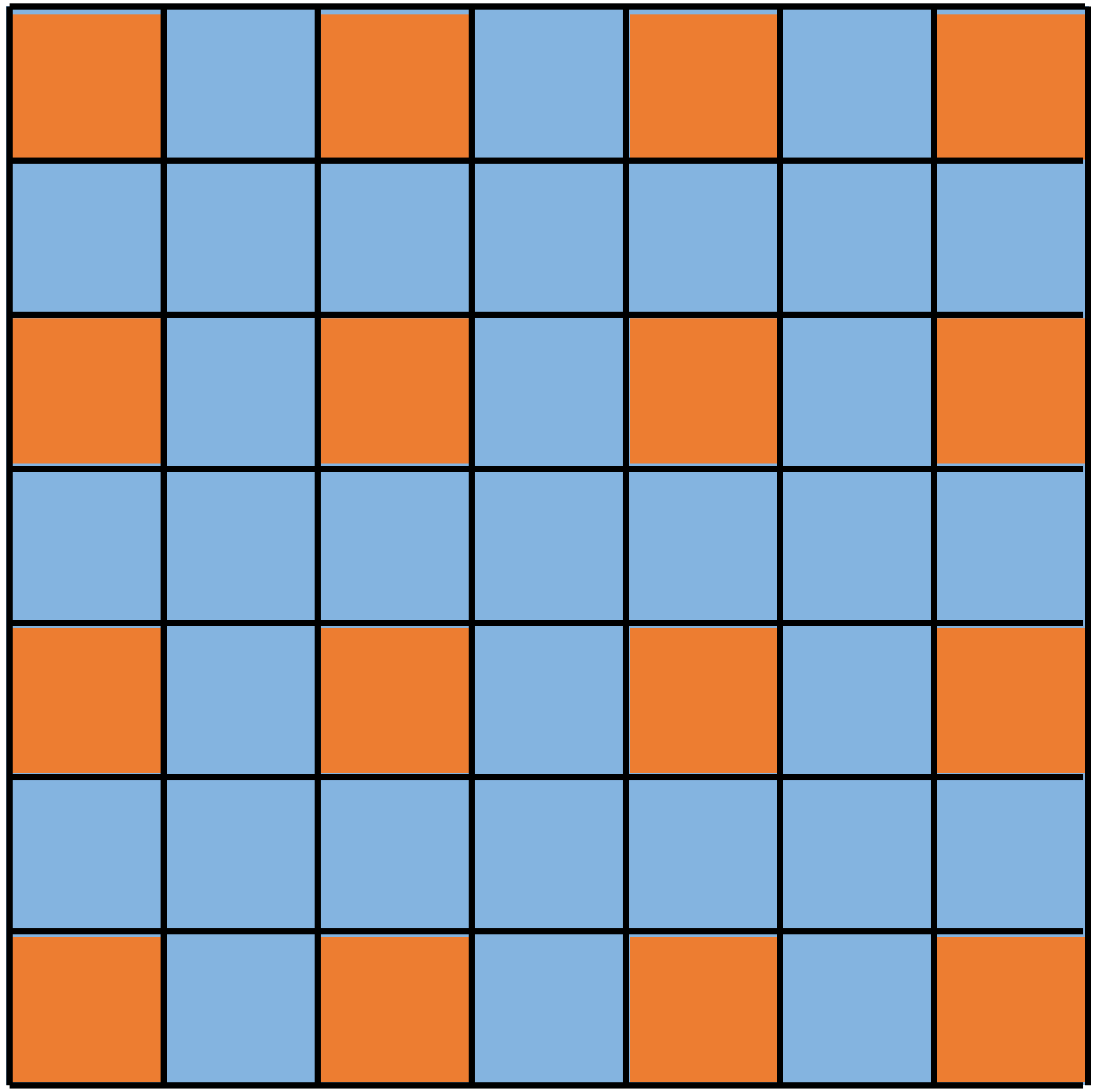
```
real, dimension(N) :: a
real :: sum
!$omp parallel do private(i)
do i=1,N
 !$omp atomic
 sum = sum + a(i)
end do
!$omp end parallel do
```

faster:

```
real, dimension(N) :: a
real :: sum
!$omp parallel do private(i)
 reduction (+: sum )
do i=1,N
 sum = sum + a(i)
end do
!$omp end parallel do
```

Example 2: race conditions

best: arrange work such that different threads work on different data



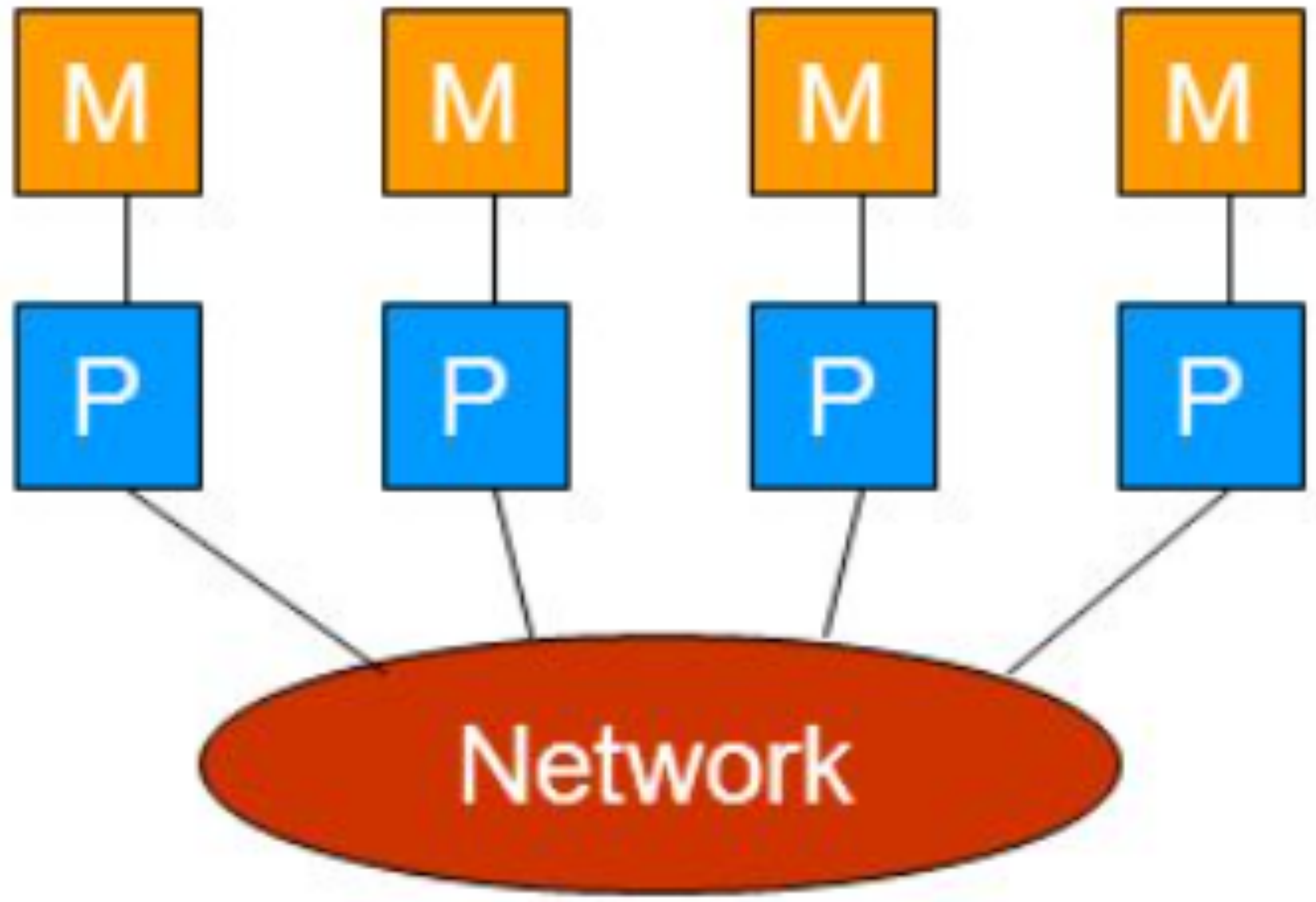
example: spectral element, start with orange (non-adjacent) elements

Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- **let the threads do work that does not affect others**

Distributed memory: MPI

- many processes run on multiple nodes
- process can access only data on the node it is running on
- use communication library MPI (Message Passing Interface) to access data on other nodes



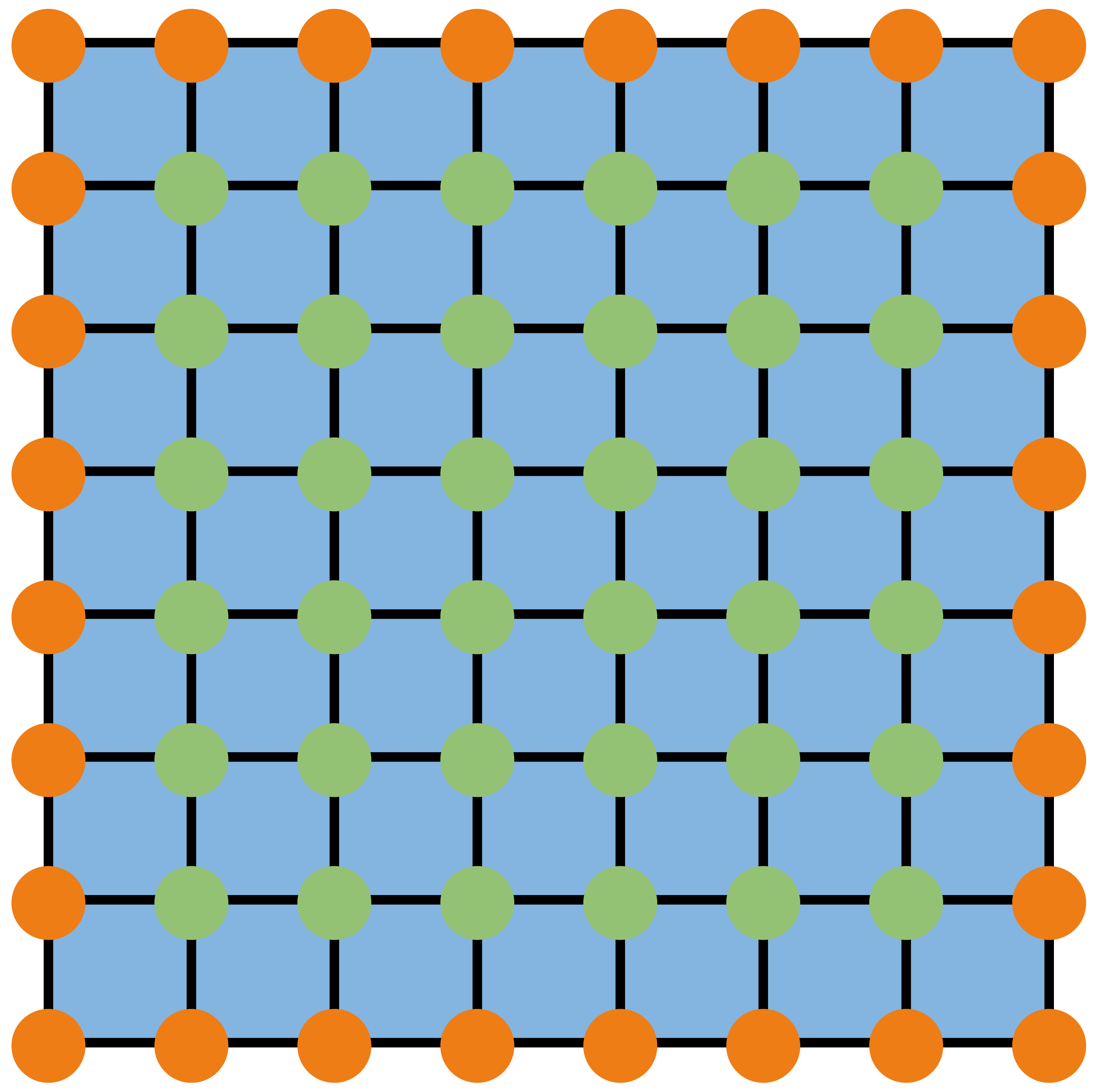

```
integer :: len, destination, tag, nreq
comm = mpi_comm_world
call mpi_init(ierr)
call mpi_comm_rank(comm, myid, ierr)
call mpi_comm_size(comm, numproc, ierr)
nreq = 0
...
do i=1,N ! loop over processors with which we
    want to communicate
 destination = ...
 nreq = nreq + 1
 call mpi_irecv(recvdata, len, mpi_real,
     destination, tag, comm, request(nreq), ierr)
 nreq = nreq + 1
 call mpi_isend(senddata, len, mpi_real,
     destination, tag, comm, request(nreq), ierr)
end do
... do some work ...
call mpi_waitall(nreq, request, status, ierr)
call mpi_finalize(ierr)
```

ECMWF

Overlap communication and computation

- Example: grid point method with only next neighbour communication:
  1. initiate communication to send and receive data for boundary points (orange)
  2. compute interior points while the data is on its way (green)
  3. compute boundary points (orange) once data has arrived

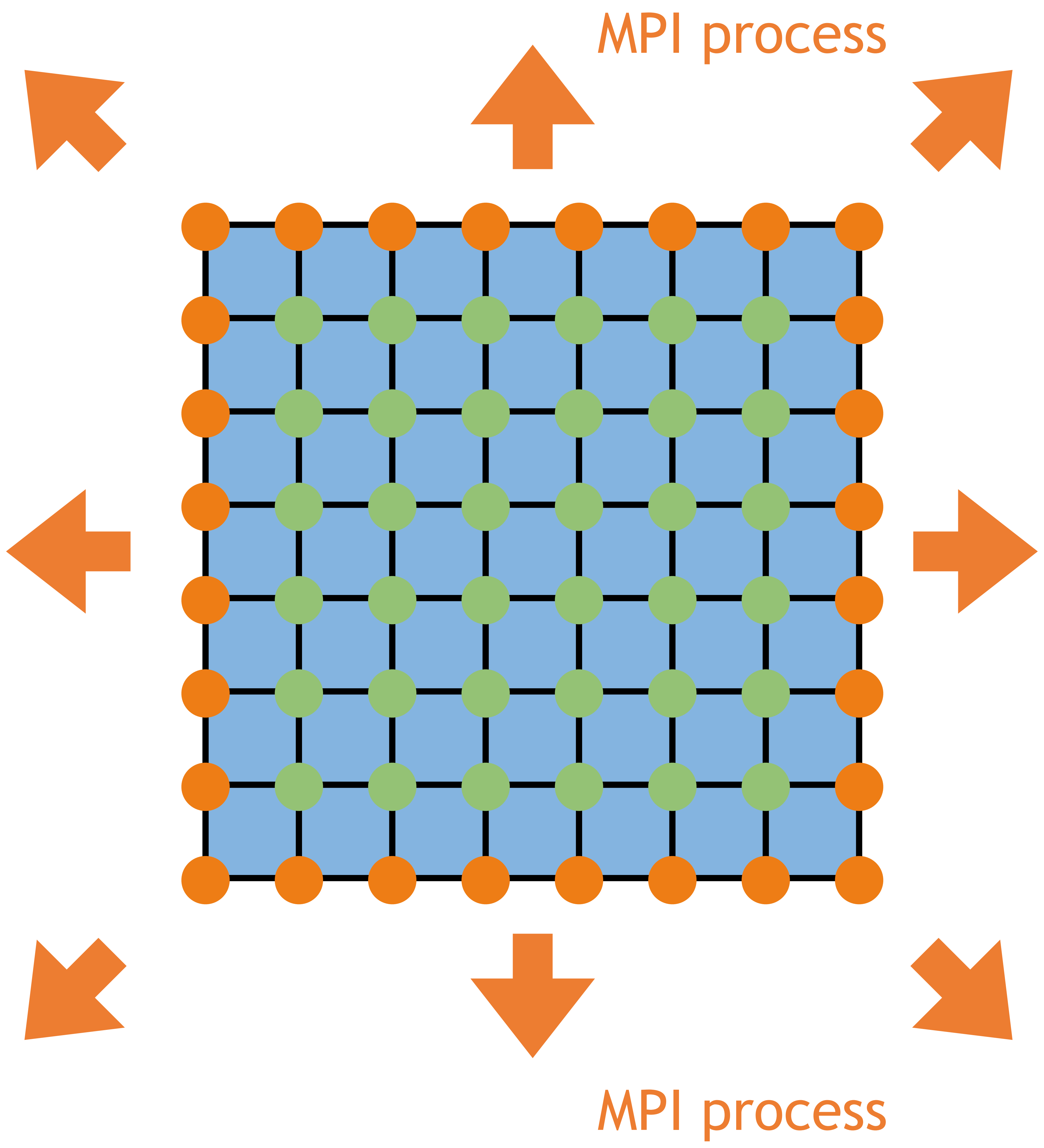
- try to reduce the physical distance that data needs to travel (difficult)



Overlap communication and computation

- Example: grid point method with only next neighbour communication:
  1. initiate communication to send and receive data for boundary points (orange)
  2. compute interior points while the data is on its way (green)
  3. compute boundary points (orange) once data has arrived

- try to reduce the physical distance that data needs to travel (difficult)



Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- let the threads do work that does not affect others
- **overlap computation and communication**

Use data once per time-step

bad example:

```
real, dimension(N) :: a,b
real :: sum
integer :: i,N
sum = 0.0
a = 0.0
b = 0.0
do i=1,N
 b(i) = i
end do
do i=1,N
 a(i) = a(i) + b(i)
end do
do i=1,N
 sum = sum + a(i)
end do
print*,sum
```

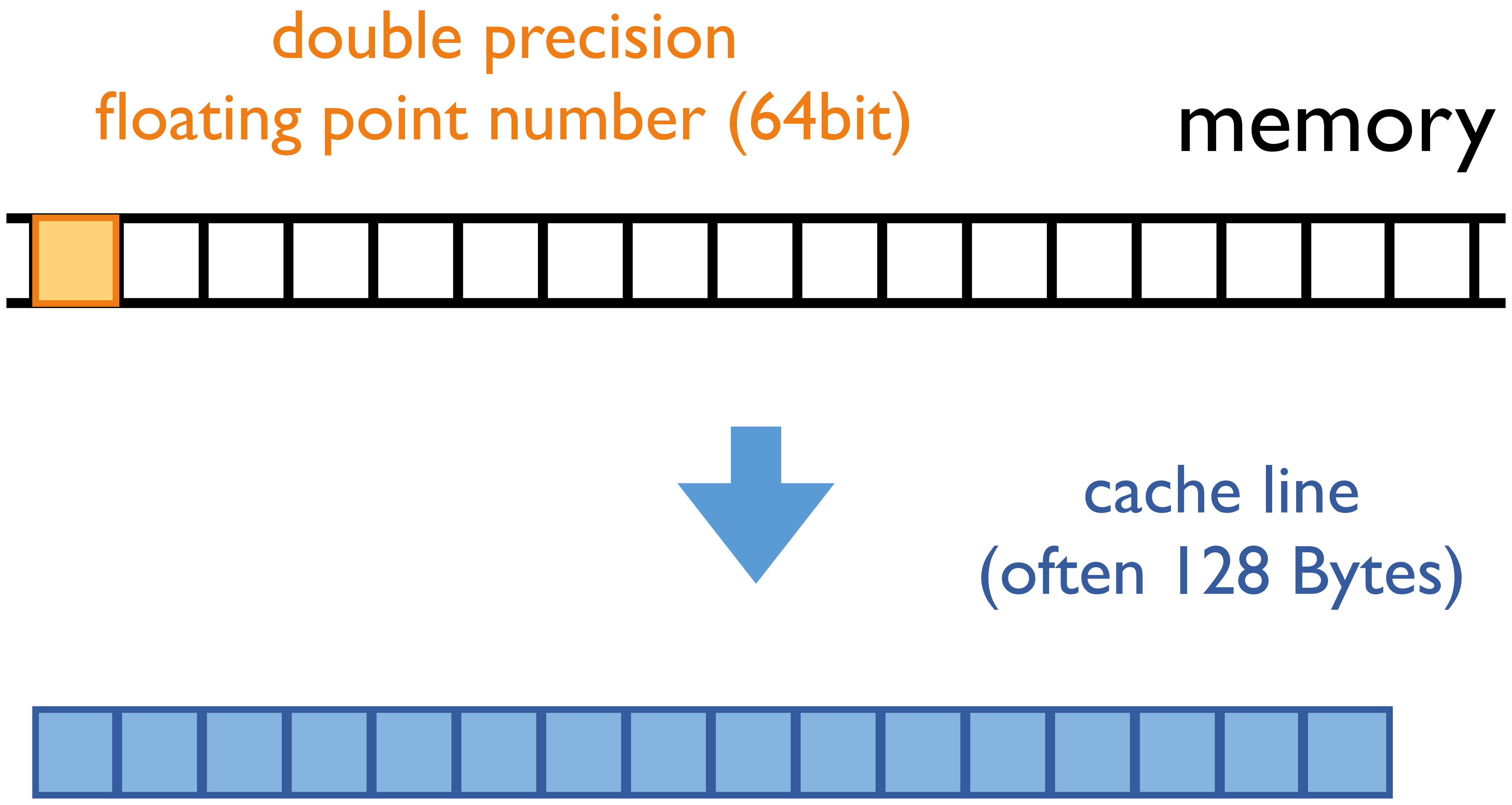
good:

```
real, dimension(N) :: a,b
real :: sum
integer :: i,N
sum = 0.0
do i=1,N
 a(i) = 0.0
 b(i) = i
 a(i) = a(i) + b(i)
 sum = sum + a(i)
end do
print*,sum
```

Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- let the threads do work that does not affect others
- overlap computation and communication
- **use data only once per time-step**

Contiguous memory access



store data in the order in which you need it
and use it in this order!

Fortran (column major order):

```fortran
real, dimension(N,M) :: a,b
integer :: i,j,N,M
do j=1,M
 do i=1,N
  a(i,j) = a(i,j) + b(i,j)
  ! fast index should be i
 end do
end do
```

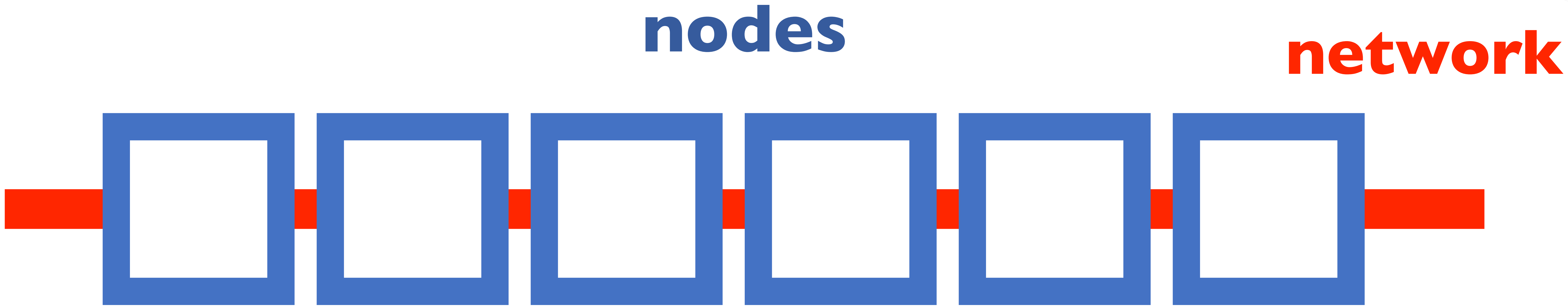
C (row major order):

```c
int i,j,N,M;
for (i=0; i<N; i++) {
 for (j=0; j<M; j++) {
  a[i][j] = a[i][j] + b[i][j]
  // fast index should be j
 }
}
```

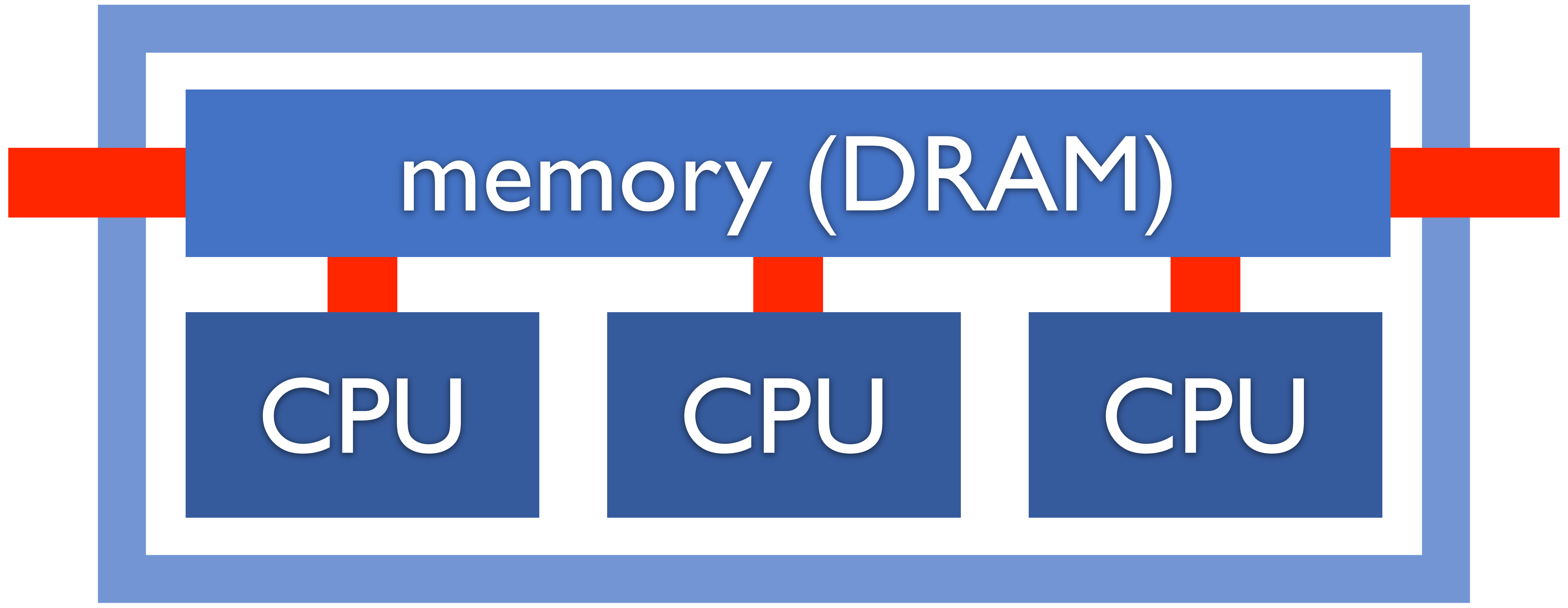
Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- let the threads do work that does not affect others
- overlap computation and communication
- use data only once per time-step
- **contiguous memory access**

Supercomputer/Cluster



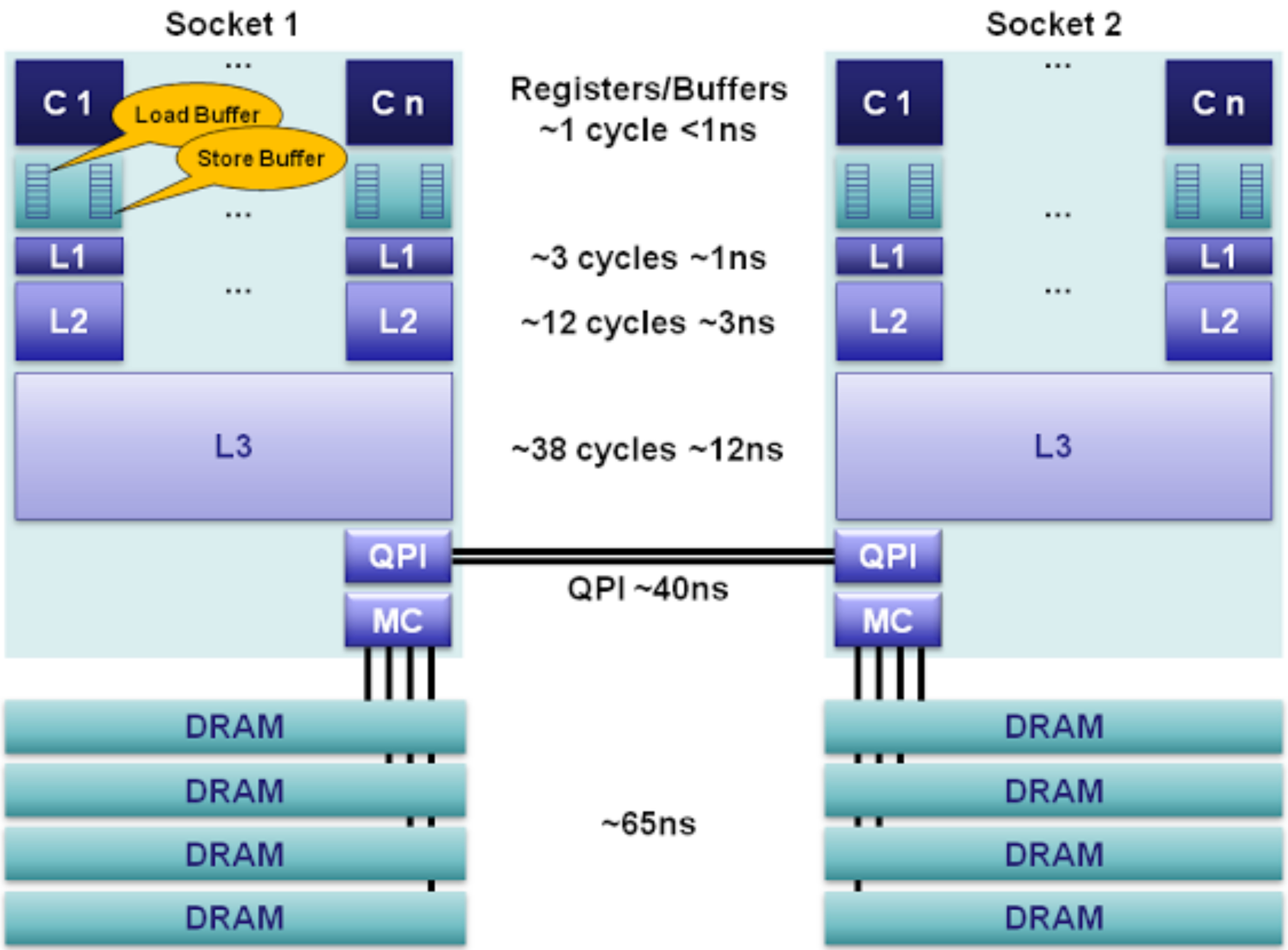
Node



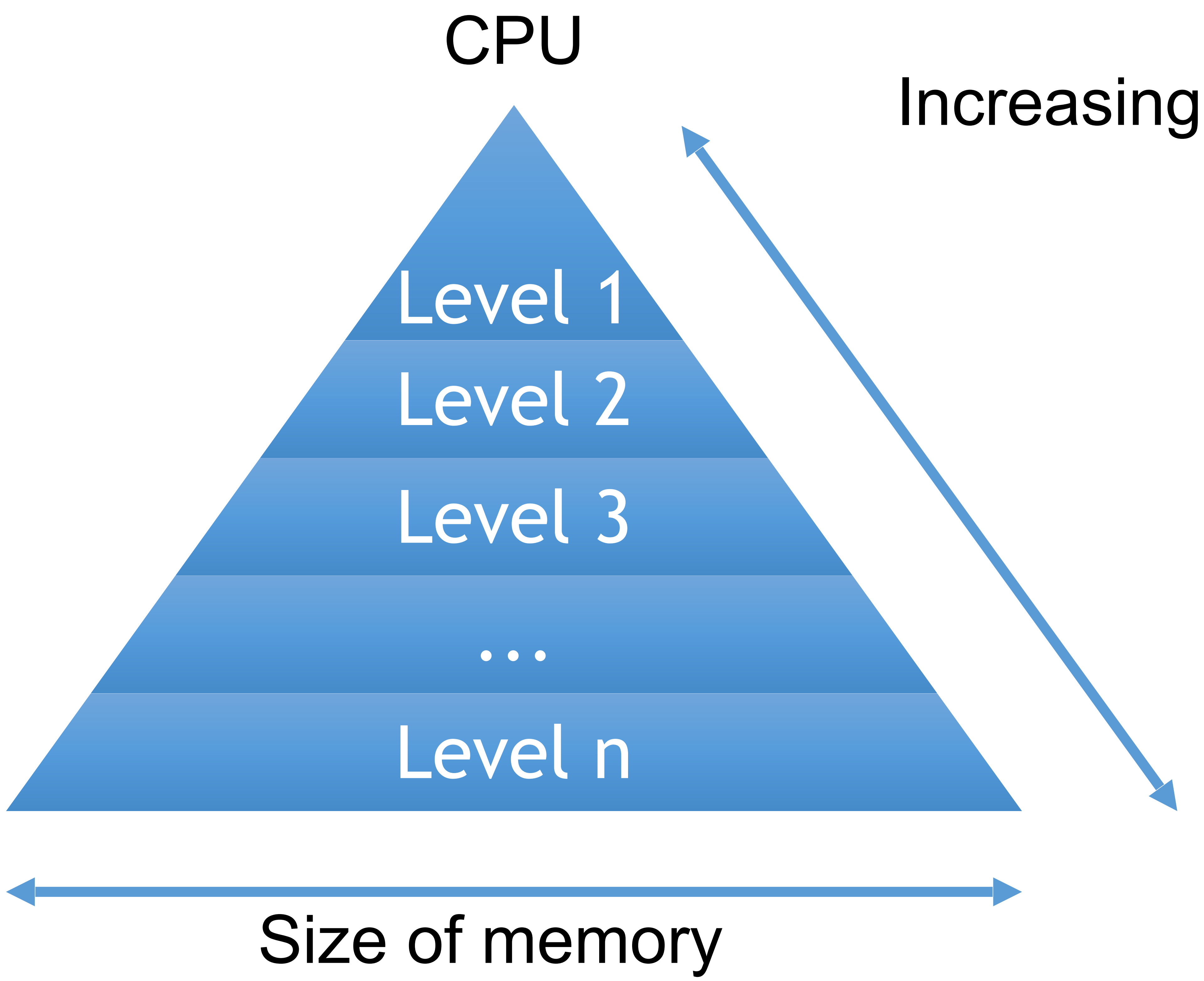
CPU

central processing unit; does one instruction like c=a+b per clock cycle

Memory hierarchy inside one node



Cache



Increasing distance from CPU = larger access time

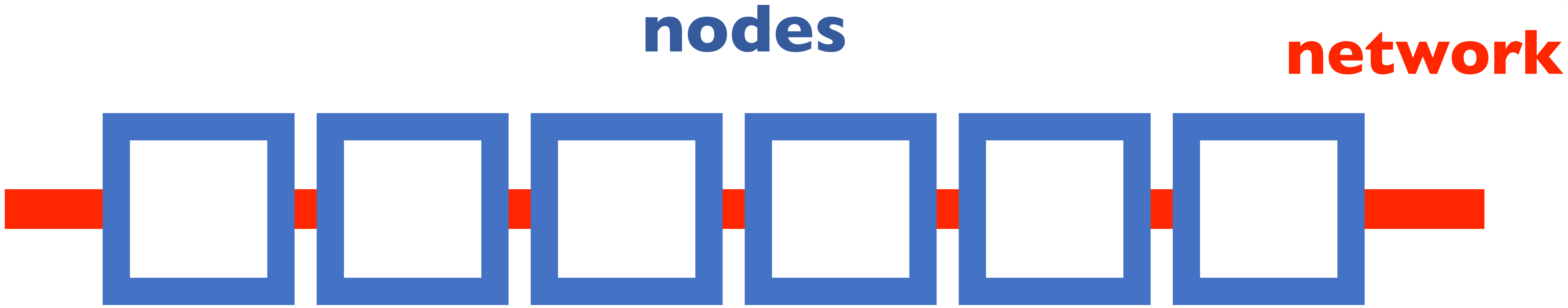
Example:

L1: 32 kB, latency 3 cycles
L2: 256 kB, latency 10 cycles
L3: 8MB, latency 40 cycles
DRAM: 16GB, latency 200 cycles
DISK: 1TB, latency 1.000.000 cycles

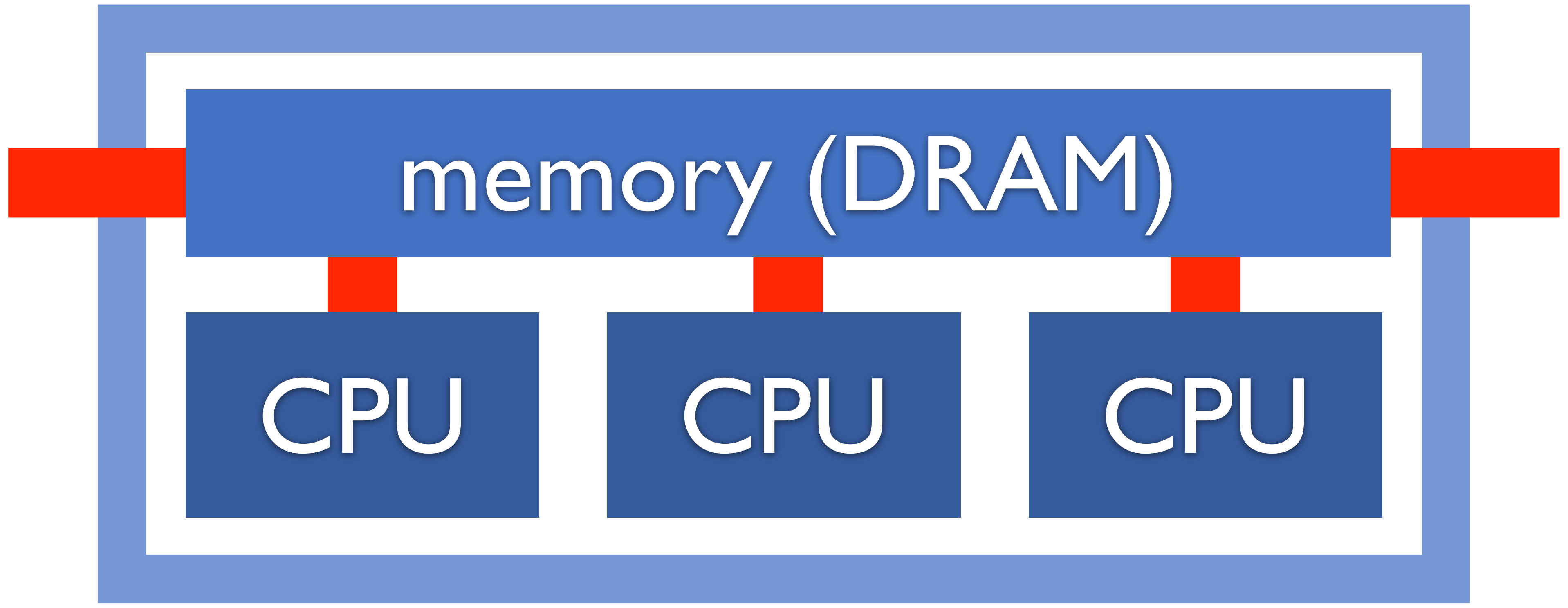
Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- let the threads do work that does not affect others
- overlap computation and communication
- use data only once per time-step
- contiguous memory access
- **try to fit data into cache**

Supercomputer/Cluster



Node



Bottlenecks

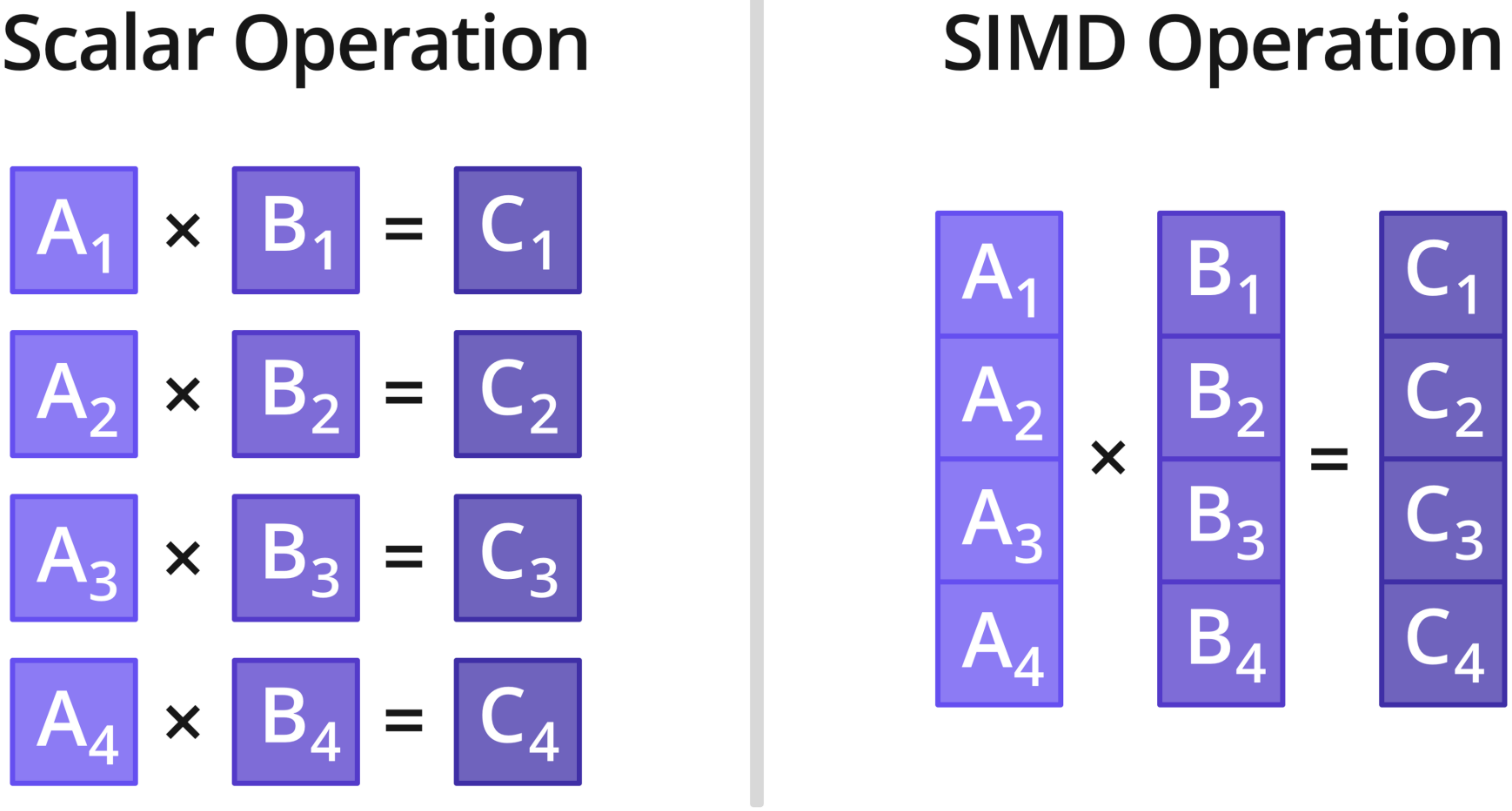
- network (connection between nodes)
- connection between DRAM and processor

Fast and slow operations

- In terms of cost
- Fast and inexpensive: add, multiply, sub, fma (fused multiply add)
- Medium: divide, modulus, sqrt
- Slow: power, trigonometric functions

- try linear algebra (BLAS, LAPACK) and math libraries (Intel MKL)

Vectorisation
SIMD = single instruction multiple data



important: data needs to be contiguous in memory

Vectorisation

initial version:

```
real :: rho, rho_x, rho_y, rho_z, u, v, w, rhs
do e=1,num_elem ! loop through all elements
   do i=1,num_points_e ! loop through all points of
     the element e
      ... ! compute derivatives rho_x, rho_y, rho_z
      rhs = u*rho_x + v*rho_y + w*rho_z + ...
   end do !i
end do !e
```

optimised for compiler vectorisation:

```
real, dimension(num_points_e) :: rho, rho_x, rho_y, &
   rho_z, u, v, w, rhs
do e=1,num_elem ! loop through all elements
   ... ! compute derivatives like rho_x, rho_y, rho_z
   rhs = u*rho_x + v*rho_y + w*rho_z + ...
end do !e
```

Vectorisation

initial version:

```
real :: rho, rho_x, rho_y, rho_z, u, v, w, rhs
do e=1,num_elem ! loop through all elements
   do i=1,num_points_e ! loop through all points of the element e
      ... ! compute derivatives rho_x, rho_y, rho_z
      rhs = u*rho_x + v*rho_y + w*rho_z + ...
   end do !i
end do !e
```

9.4s
14.4% vector operations

optimised for compiler vectorisation:

```
real, dimension(num_points_e) :: rho, rho_x, rho_y, &
   rho_z, u, v, w, rhs
do e=1,num_elem ! loop through all elements
   ... ! compute derivatives like rho_x, rho_y, rho_z
   rhs = u*rho_x + v*rho_y + w*rho_z + ...
end do !e
```

2.1s
73.9% vector operations

measurements: spectral element model NUMA, NPS

vector intrinsics (here for BG/Q)

```
real, dimension(4,4,4) :: rho, rho_x, rho_y, &
   rho_z, u, v, w, u_x, v_y, w_z, rhs
!IBM* align(32, rho, rho_x, rho_y, rho_z, u, v, w,
     u_x, v_y, w_z, rhs)
! declare variables representing registers: (each
     contains four double precision floating point
     numbers)
vector(real(8)) vct_rho, vct_rhox, vct_rhoy, vct_rhoz
vector(real(8)) vct_u, vct_v, vct_w, vct_rhs
if (iand(loc(rho), z'1F') .ne. 0) stop 'rho is not
     aligned'
... ! check alignment of other variables
do e=1,num_elem ! loop through all elements
   do k=1,4 ! loop over points in z-direction
      do j=1,4 ! loop over points in y-direction
         ... ! compute derivatives rho_x, ...
         ! load always four floating point numbers:
         vct_u = vec_ld(0, u(1,j,k))
         vct_v = vec_ld(0, v(1,j,k))
         vct_w = vec_ld(0, w(1,j,k))
         vct_rhox = vec_ld(0, rho_x(1,j,k))
         vct_rhoy = vec_ld(0, rho_y(1,j,k))
         vct_rhoz = vec_ld(0, rho_z(1,j,k))
         ! rhs = u*rho_x
```

vector intrinsics (here for BG/Q)

```
      do j=1,4 ! loop over points in y-direction
         ... ! compute derivatives rho_x, ...
         ! load always four floating point numbers:
         vct_u = vec_ld(0, u(1,j,k))
         vct_v = vec_ld(0, v(1,j,k))
         vct_w = vec_ld(0, w(1,j,k))
         vct_rhox = vec_ld(0, rho_x(1,j,k))
         vct_rhoy = vec_ld(0, rho_y(1,j,k))
         vct_rhoz = vec_ld(0, rho_z(1,j,k))
         ! rhs = u*rho_x
         vct_rhs = vec_mul(vct_u,vct_rhox)
         ! rhs = rhs + v*rho_y
         vct_rhs = vec_madd(vct_v,vct_rhoy,vct_rhs)
         ! rhs = rhs + w*rho_z
         vct_rhs = vec_madd(vct_w,vct_rhoz,vct_rhs)
         ! write result from register into cache:
         call vec_st(vct_rhs, 0, rhs(1,j,k))
         ...
      end do !j
   end do !k
end do !e
```

1.0s
98.6% vector operations

ECMWF

measurements: spectral element model NUMA, NPS

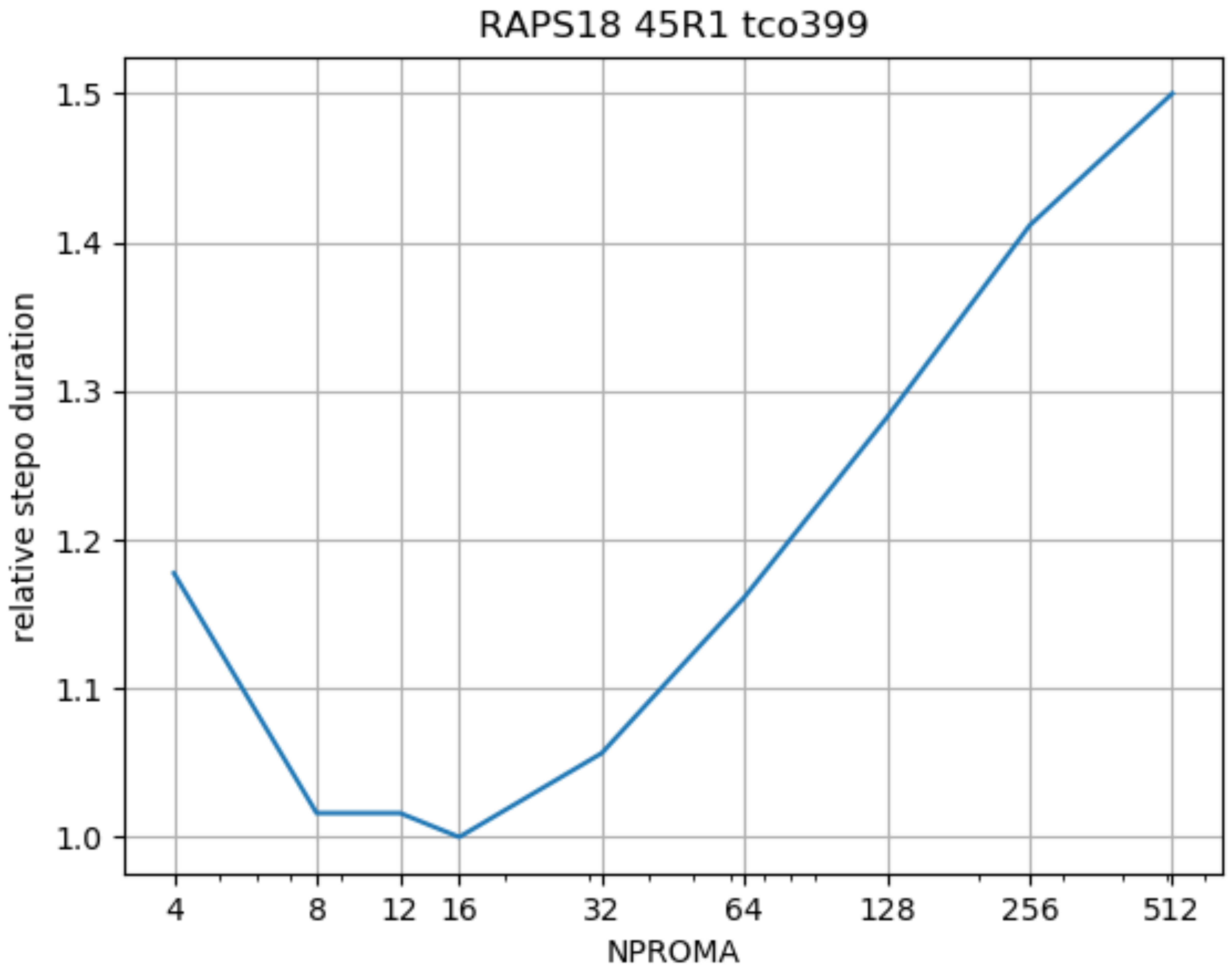
Vectorization in IFS

horizontal grid columns often independent of each other
=> idea: use block of NPROMA columns for vectorization

NPROMA: block size
NBLOCK: block number
total number of grid columns:
NGPTOT = NPROMA * NBLOCK

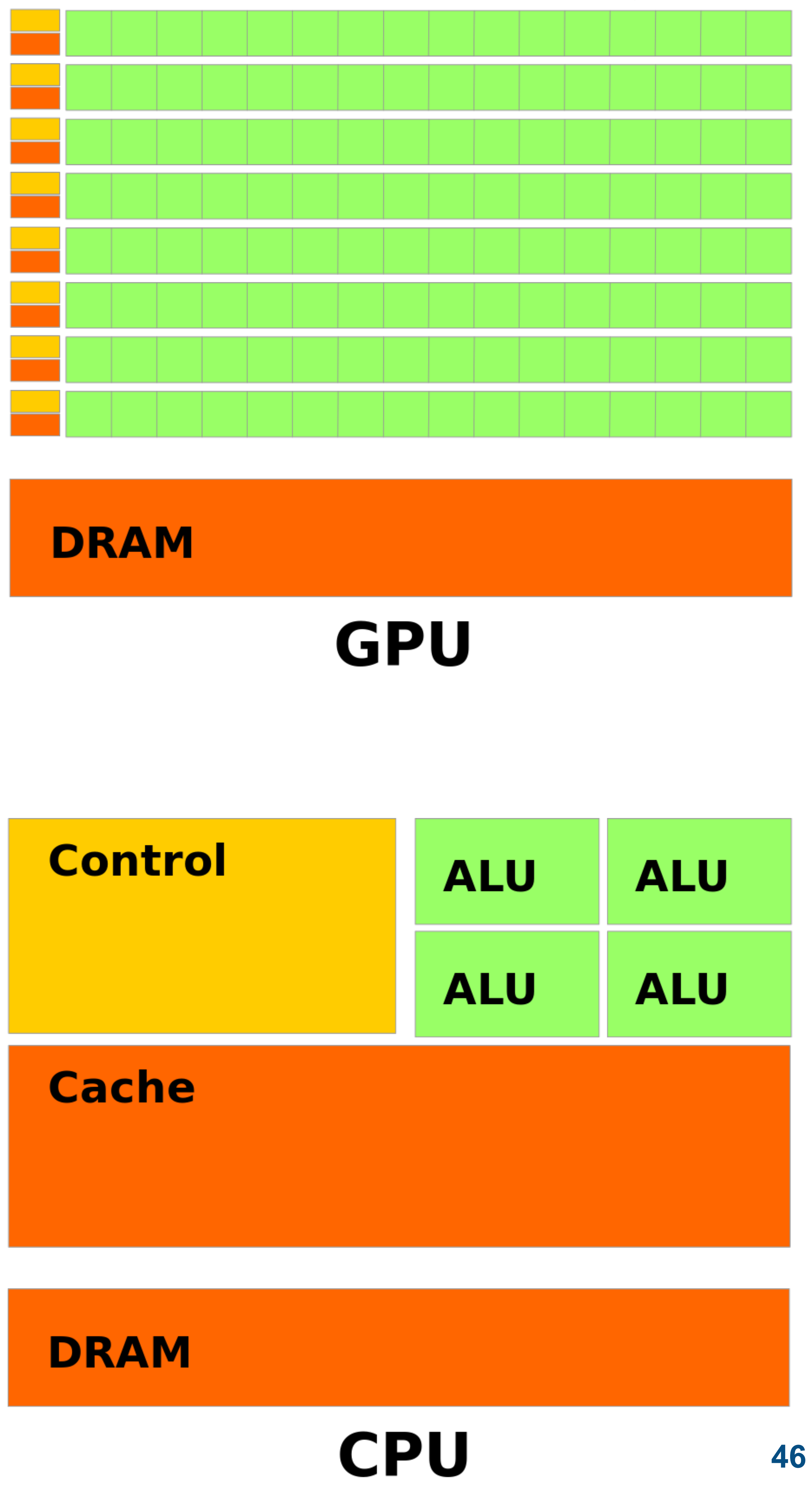
```
real :: array(NPROMA ,NLEVELS ,NBLOCK)

do bl = 1, NBLOCKS
   do lev = 1, NLEVELS
      do jl = 1, NPROMA
         array(jl, lev, bl) = <expression >
         ...
      end do
   end do
end do
```



GPU (Graphics Processing Unit)

- small number of instructions => requires host CPU
- GPU/CPU interface (PCIe up to 16GB/sec, NVLINK up to 300GB/sec between GPUs in same node)
- more energy efficient than CPUs
- high performance GPUs today mainly supplied by NVIDIA but supercomputers based on AMD GPUs are currently built
- lots of cores share one control unit
- very little memory inside the GPU



ECMWF

Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- let the threads do work that does not affect others
- overlap computation and communication
- use data only once per time-step
- contiguous memory access
- try to fit data into cache
- **make good use of vectorisation**

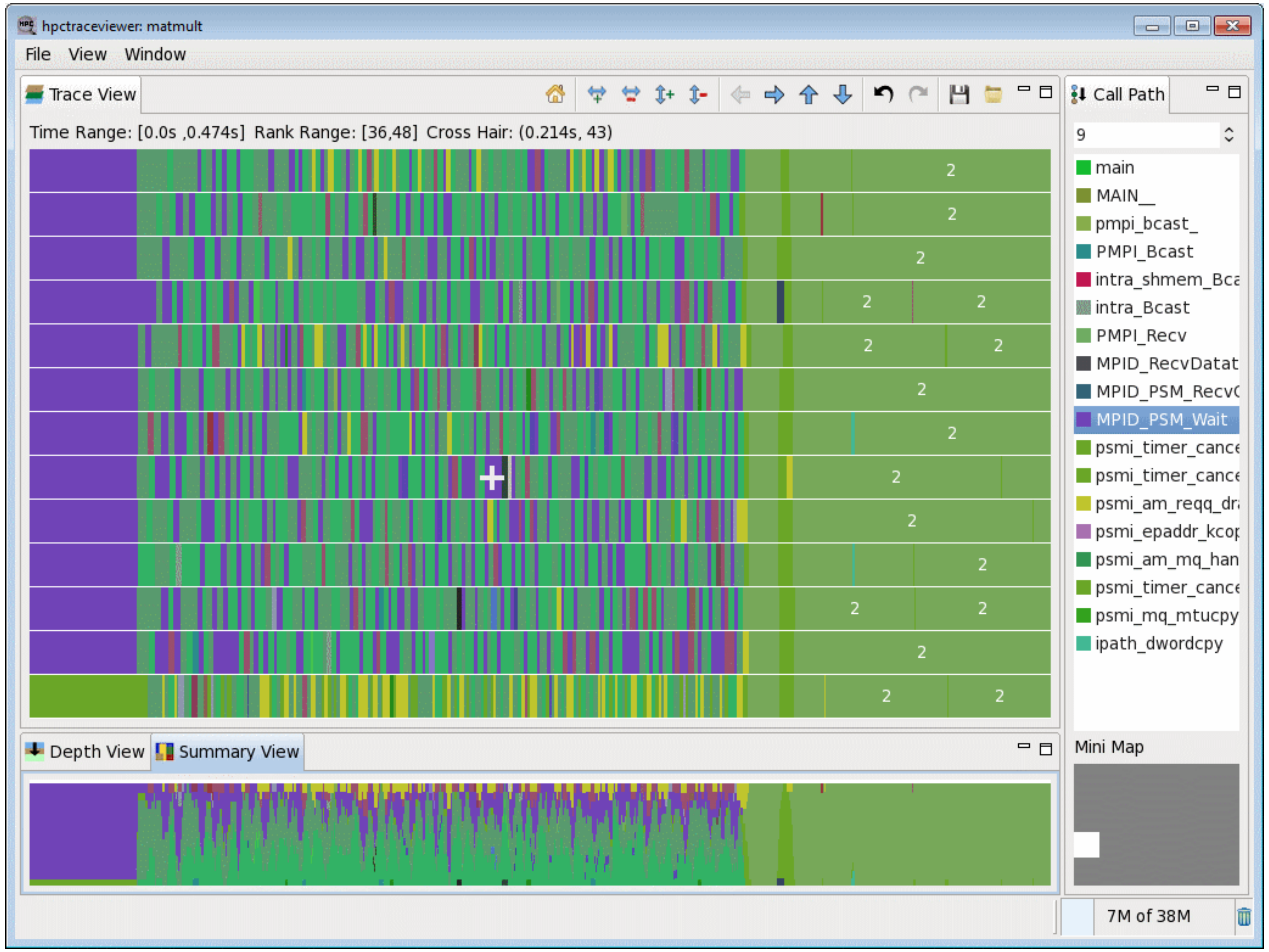
How good are we?

Hardware performance counters

- set of special-purpose hardware registers to store counts of hardware-related activities
- can help in spotting the application bottlenecks
- allow for low-level performance analysis and tuning, though implementation may be somehow difficult
- tools: PAPI, VTUNE, HPCToolkit, Nsight, Rocprof, ...

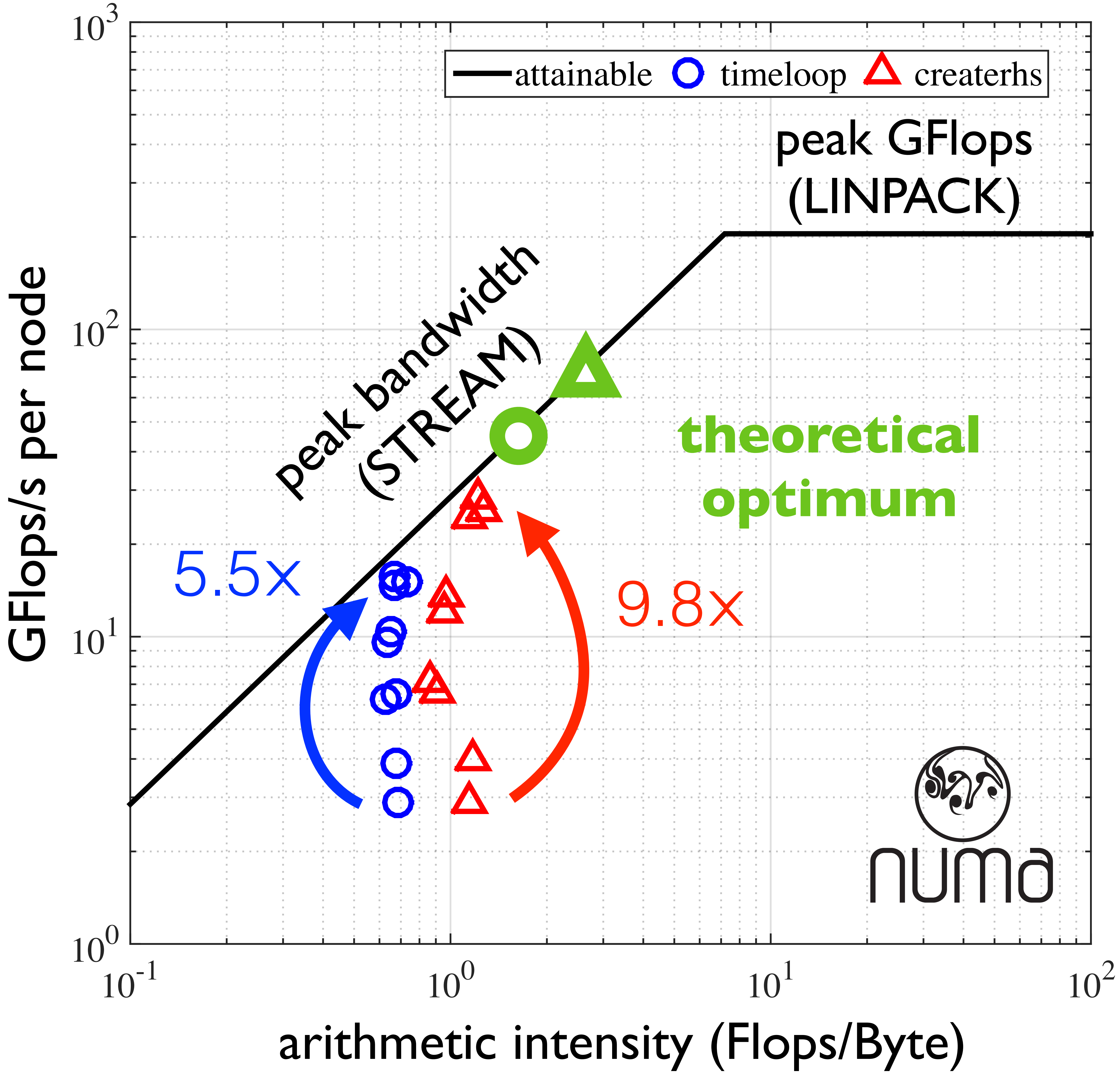
Hardware performance counters

- set of special-purpose hardware registers to store counts of hardware-related activities
- can help in spotting the application bottlenecks
- allow for low-level performance analysis and tuning, though implementation may be somehow difficult
- tools: PAPI, VTUNE, HPCToolkit, Nsight, Rocprof, ...



ECMWF

Roofline plot



blue: entire timeloop

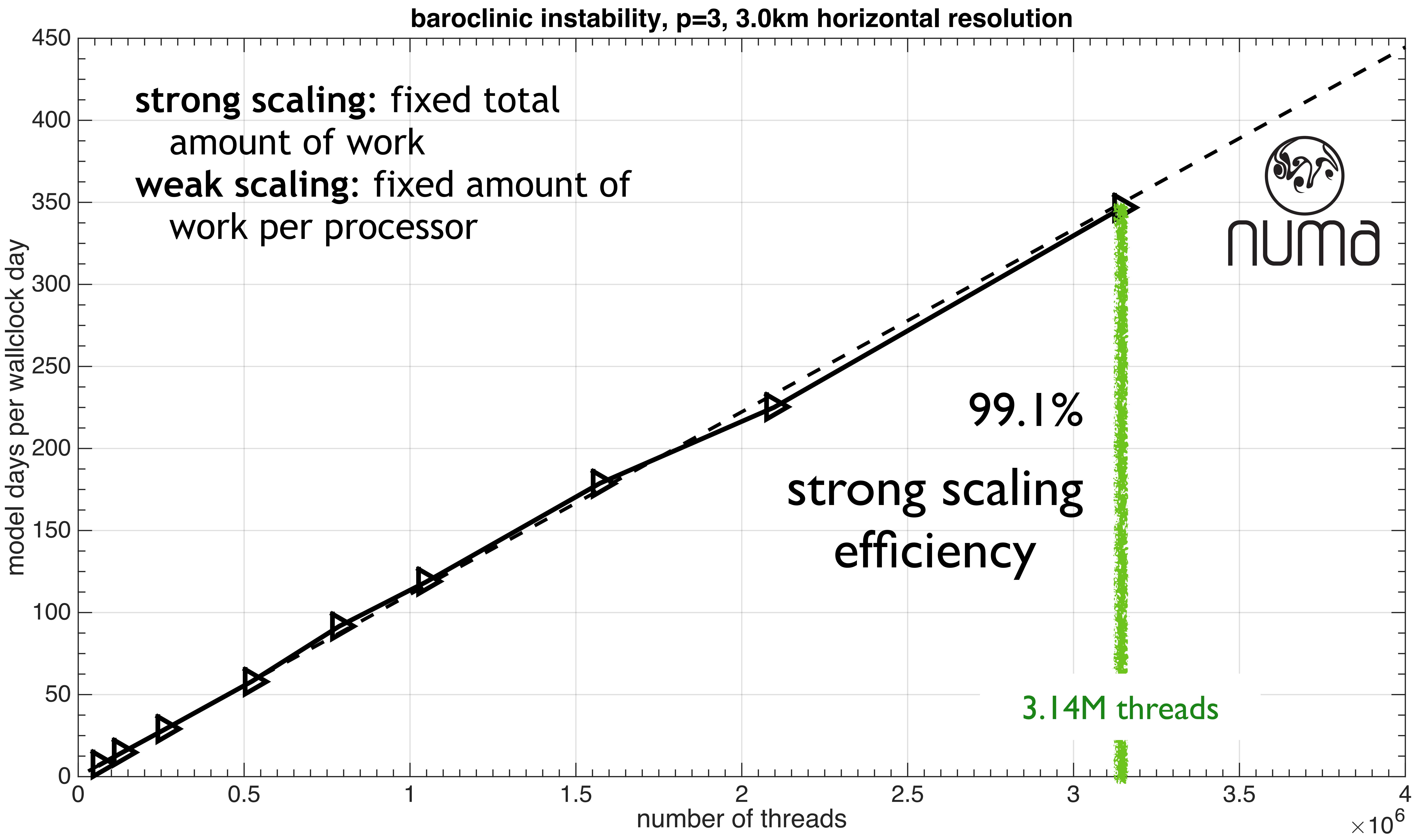
red: main computational kernel

data points: different optimization stages

measurements: spectral element model NUMA, NPS

ECMWF

Strong scaling efficiency



measurements: spectral element model NUMA, NPS

ECMWF

Create performance model

example code:

```
real, dimension(N,M) :: a,b,c
integer :: i,j,N,M
do timestep=1,nstep
 do j=1,M
  do i=1,N
   a(i,j) = a(i,j) + b(i,j) * c(i,j)
  end do
 end do
end do
```

Create performance model

example code:

```
real, dimension(N,M) :: a,b,c
integer :: i,j,N,M
do timestep=1,nstep
 do j=1,M
  do i=1,N
   a(i,j) = a(i,j) + b(i,j) * c(i,j)
  end do
 end do
end do
```

parameters:

parameter	value
N	1E+04
M	1E+05
nstep	100
GB/s	20
GFlops/s	200

Create performance model

example code:

```
real, dimension(N,M) :: a,b,c
integer :: i,j,N,M
do timestep=1,nstep
 do j=1,M
  do i=1,N
   a(i,j) = a(i,j) + b(i,j) * c(i,j)
  end do
 end do
end do
```

parameters:

parameter	value
N	1E+04
M	1E+05
nstep	100
GB/s	20
GFlops/s	200

floating point operations:

function	operations per step	
main	2*N*M	2E+11
total GFlops for all steps		20000
runtime		100,0

Create performance model

example code:

```
real, dimension(N,M) :: a,b,c
integer :: i,j,N,M
do timestep=1,nstep
 do j=1,M
  do i=1,N
   a(i,j) = a(i,j) + b(i,j) * c(i,j)
  end do
 end do
end do
```

parameters:

parameter	value
N	1E+04
M	1E+05
nstep	100
GB/s	20
GFlops/s	200

floating point operations:

function	operations per step	
main	2*N*M	2E+11
total GFlops for all steps		20000
runtime		100,0

memory:

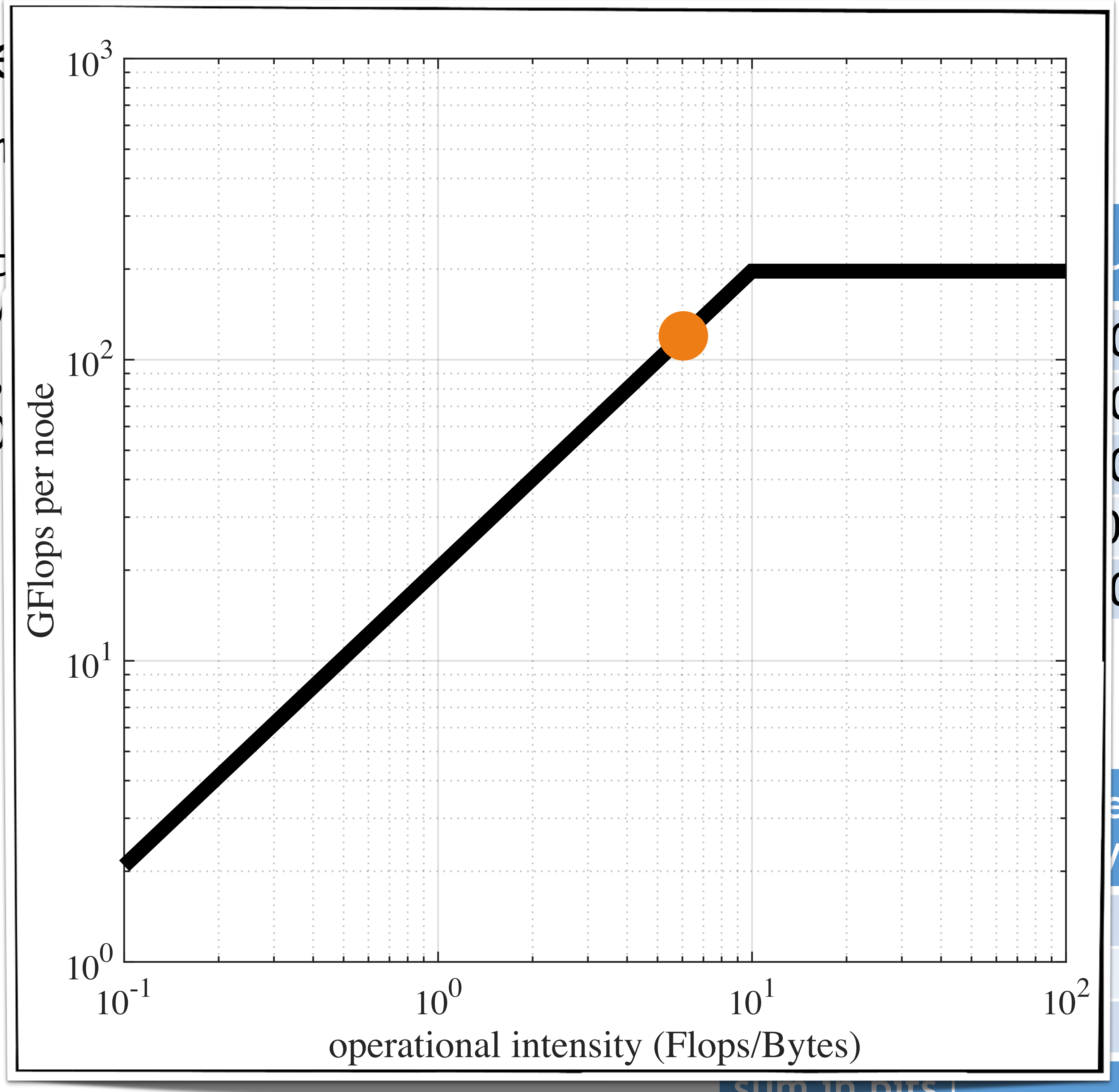
variable	bits per entry	size	#read per step	#write per step	total bits read	total bits written
a	64	N*M	1	1	6,4E+12	6,4E+12
b	64	N*M	1	0	6,4E+12	0E+00
c	64	N*M	1	0	6,4E+12	0E+00
sum in bits					1,92E+13	6,4E+12
sum in GB					2400	800
intensity	6,25			runtime in seconds		160,0

ECMWF

Create performance model

example

```
real, dim
integer :
do timest
 do j=1,M
  do i=1,
   a(i,j)
  end do
 end do
end do
```



ue
04
05
0
)
0

floating point operations:

function	operations per step	
main	2*N*M	2E+11
total GFlops for all steps		20000
runtime		100,0

	er y	size	#read per step	#write per step	total bits read	total bits written
		N*M	1	1	6,4E+12	6,4E+12
		N*M	1	0	6,4E+12	0E+00
		N*M	1	0	6,4E+12	0E+00
sum in bits					1,92E+13	6,4E+12
sum in GB					2400	800
intensity	6,25			runtime in seconds		160,0

ECMWF

Create performance model

example code:

```
real, dimension(N,M) :: a,b,c
integer :: i,j,N,M
do timestep=1,nstep
 do j=1,M
  do i=1,N
   a(i,j) = a(i,j) + b(i,j) * c(i,j)
  end do
 end do
end do
```

parameters:

parameter	value
N	1E+04
M	1E+05
nstep	100
GB/s	20
GFlops/s	200

floating point operations:

function	operations per step	
main	2*N*M	2E+11
total GFlops for all steps		20000
runtime		100,0

memory:

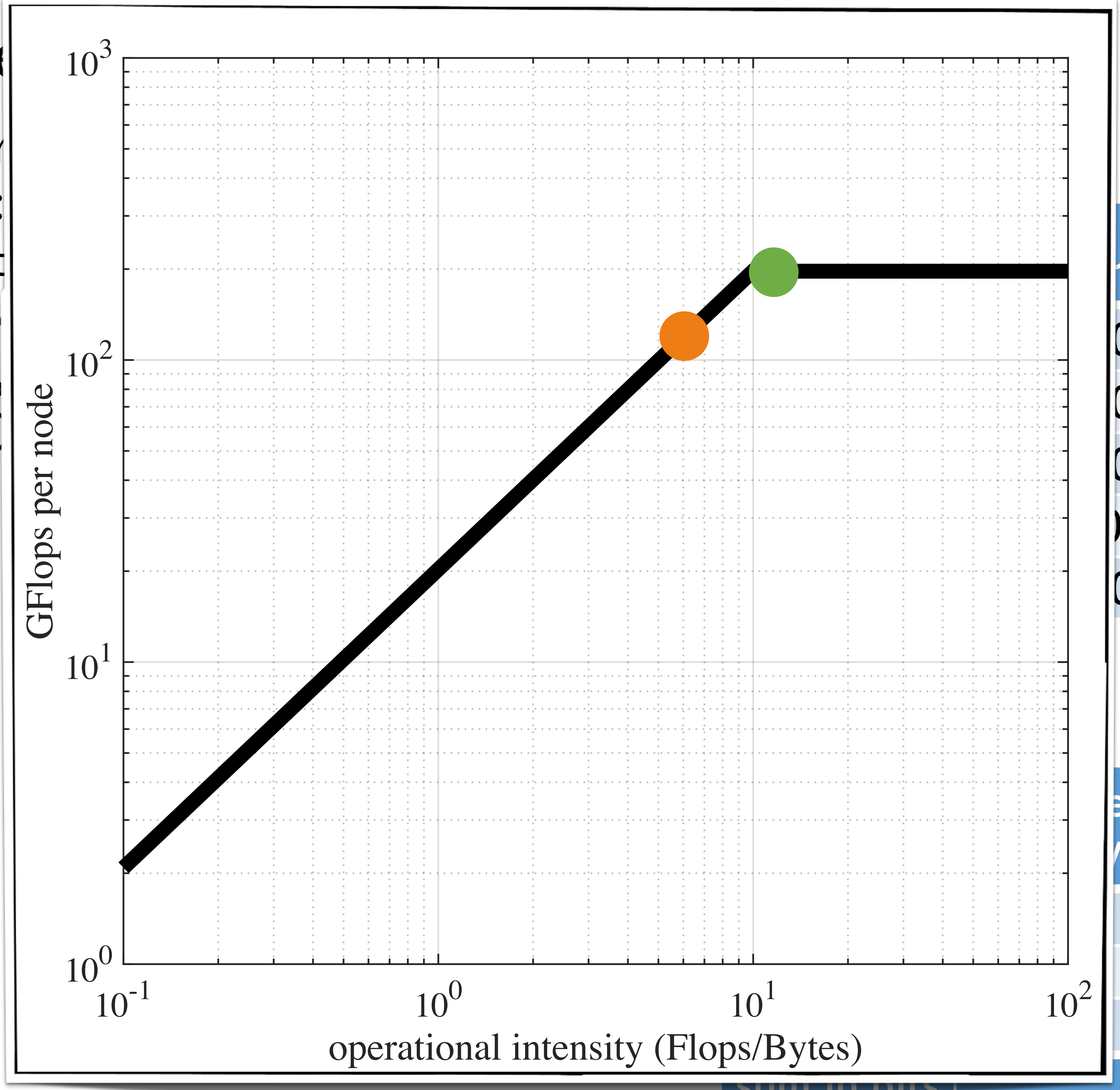
variable	bits per entry	size	#read per step	#write per step	total bits read	total bits written
a	64	N*M	1	1	6,4E+12	6,4E+12
b	64	N*M	0	0	0E+00	0E+00
c	64	N*M	0	0	0E+00	0E+00
sum in bits					6,4E+12	6,4E+12
sum in GB					800	800
intensity	12,5			runtime in seconds		80,0

ECMWF

Create performance model

exampl

```
real, dim
integer :
do timest
 do j=1,M
  do i=1,
   a(i,j)
  end do
 end do
end do
```



ue
04
05
0
)
0

floating point operations:

function	operations per step	
main	2*N*M	2E+11
total GFlops for all steps		20000
runtime		100,0

	er y	size	#read per step	#write per step	total bits read	total bits written
		N*M	1	1	6,4E+12	6,4E+12
		N*M	0	0	0E+00	0E+00
		N*M	0	0	0E+00	0E+00
sum in bits					6,4E+12	6,4E+12
sum in GB					800	800
intensity	12,5			runtime in seconds		80,0

ECMWF

Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- let the threads do work that does not affect others
- overlap computation and communication
- use data only once per time-step
- contiguous memory access
- try to fit data into cache
- make good use of vectorisation
- **compare performance with expectations**

Recommendations

- try to use well optimized libraries
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- give each thread as much work as possible
- let the threads do work that does not affect others
- overlap computation and communication
- use data only once per time-step
- contiguous memory access
- try to fit data into cache
- make good use of vectorisation
- compare performance with expectations

open question

How to find good compromise between performance and readability, portability, maintainability?

Questions?