Funded by the European Union

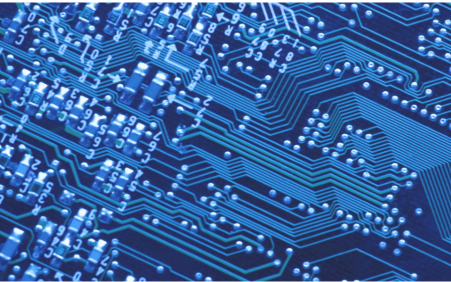
Co-ordinated by

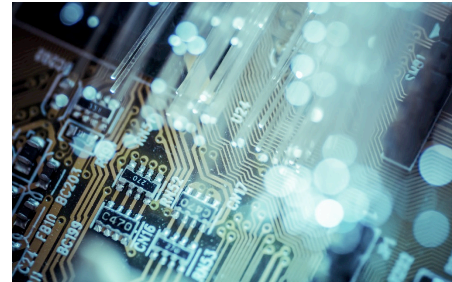
ESCAPE

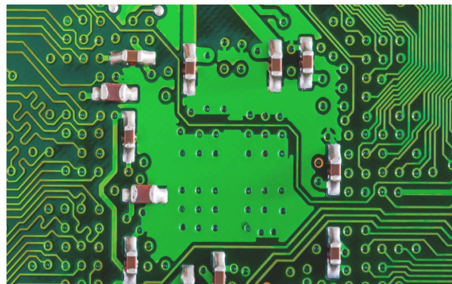
This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671627

Funded by the European Union

Co-ordinated by



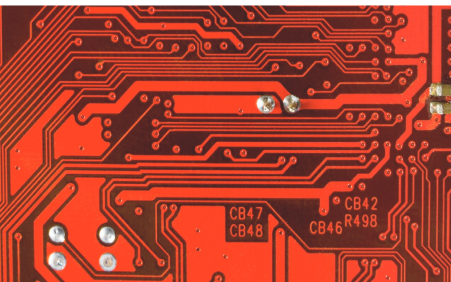




Massively Parallel Computing for NWP and climate

Andreas Mueller

17.03.2017, Reading, UK



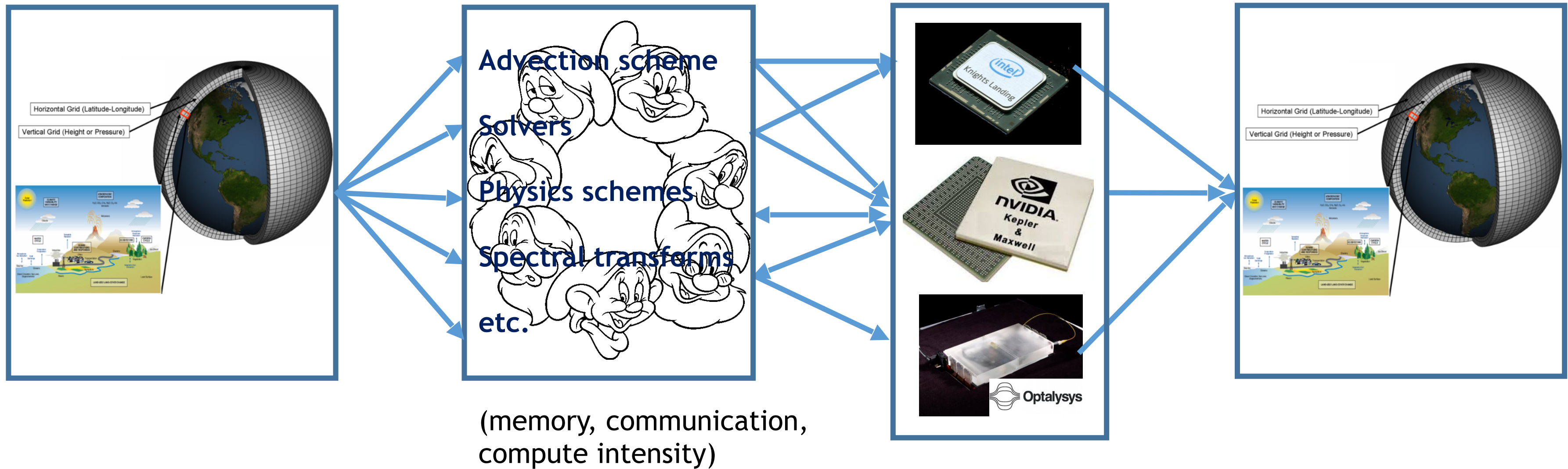
Energy-efficient Scalable Algorithms for Weather Prediction at Exascale

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671627

Funded by the European Union

ESCAPE

ESCAPE

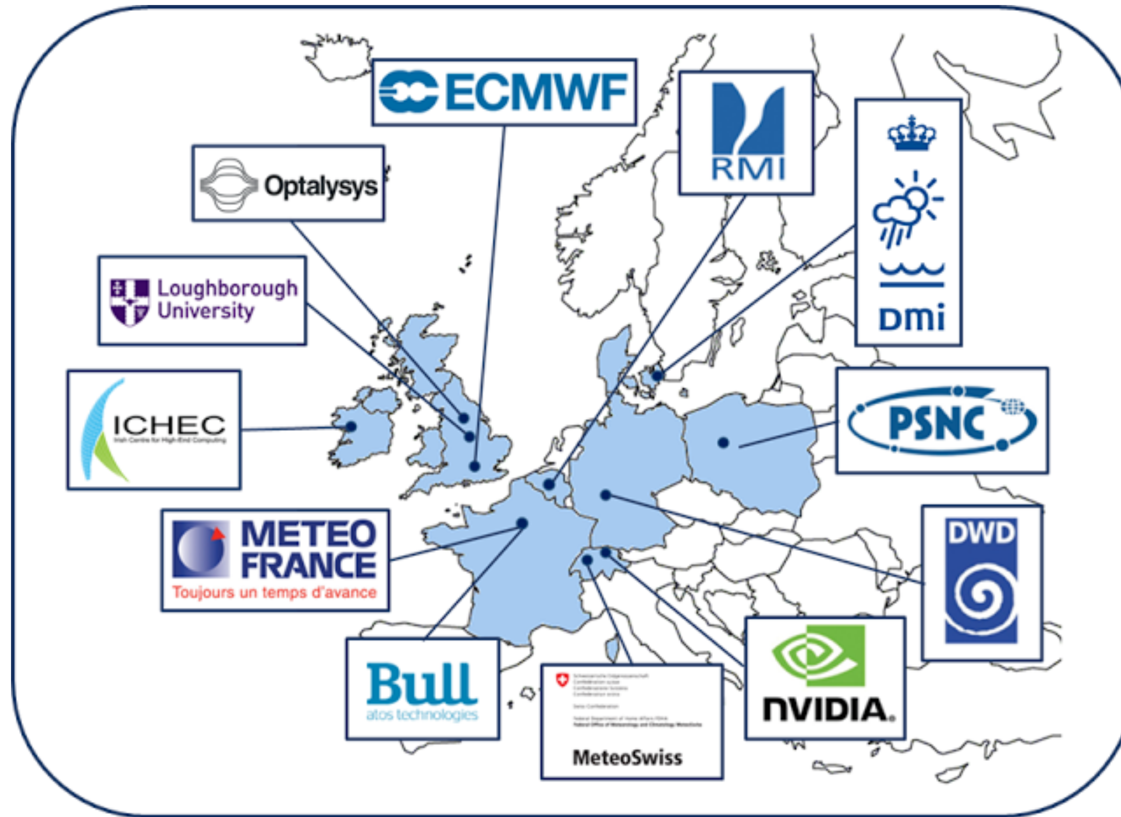


MeteoSwiss DMI RMI NVIDIA METEO FRANCE Toujours un temps d'avance ICHEC Deutscher Wetterdienst Wetter und Klima aus einer Hand DWD Optalysys Bull PSNC Loughborough University

Funded by the
European Union

ESCAPE

ESCAPE partners



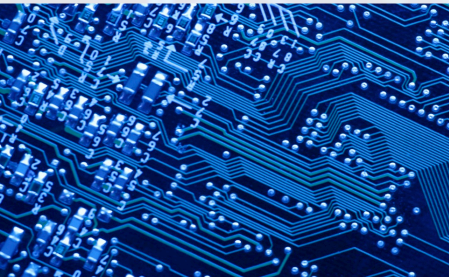
Overview

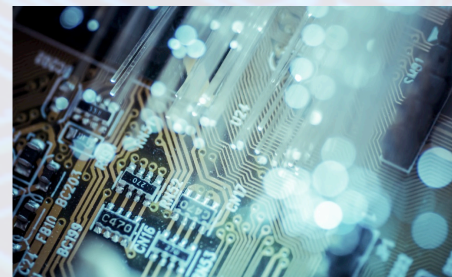
- Why do we as scientists need to know so much about computer science?
- What do we need to be aware of to write efficient code?
- How good are we?

Funded by the European Union

ESCAPE

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





Funded by the
European Union

ESCAPE

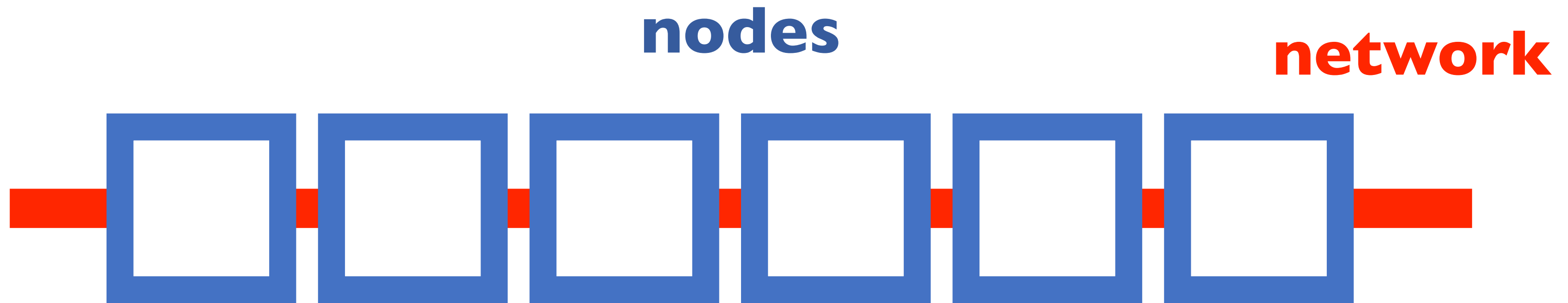
Why do we as scientists need to know so much about computer science?

- Excuse 1: let the computer scientists take care of it
- Response: computer scientists 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!

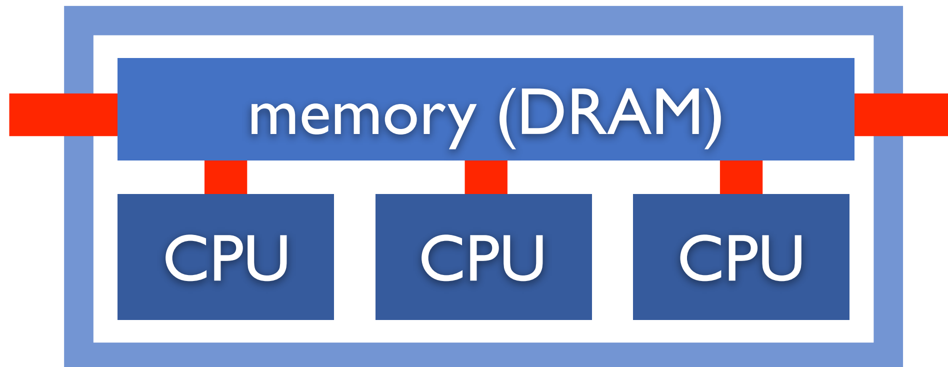
computer	electricity cost per year
ECMWF	~3 million £
fastest current supercomputer	~15 million $
next generation (exascale)	~20 million $

Funded by the
European Union

Supercomputer/Cluster



Node



CPU

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

CPU clock rate over time

10
1 GHz
0.1
0.01
0.001
0.0001
1971 1976 1981 1986 1991 1996 2001 2006 2011 2016

source: James Reinders, Intel Xeon Phi

Number of cores per chip over time

512
256
128
64
32
16
8
4
2
1

Threads
Cores

1971 1976 1981 1986 1991 1996 2001 2006 2011 2016

source: James Reinders, Intel Xeon Phi

Funded by the European Union

ESCAPE

top500.org

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
4	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
5	DOE/SC/LBNL/NERSC United States	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	622,336	14,014.7	[illegible]	[illegible]
6	Joint Center for Advanced High Performance Computing Japan	**Oakforest-PACS** - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path Fujitsu	556,104	[illegible]	[illegible]	[illegible]
7	RIKEN Advanced Institute	K computer, SPARC64 VIIIfx 2.0GHz, Tofu	705,024	10,510.0	11,280.4	12,660

Funded by the
European Union

What next?

finer resolution

computing/energy resources

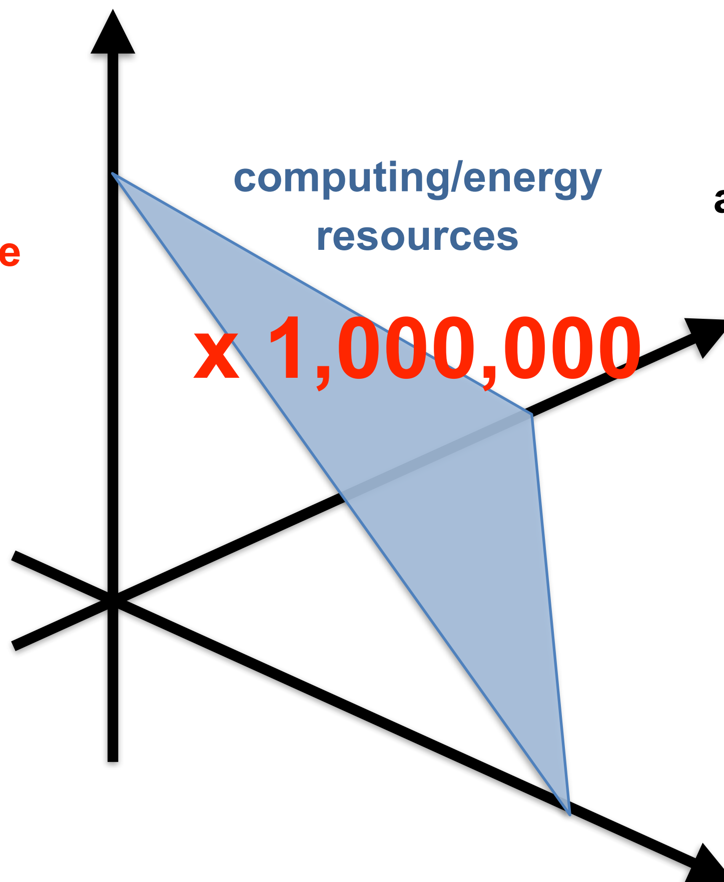
add more processes (e.g. chemistry)

more ensemble members

What next?

finer resolution

x 10 in each direction and time = 10,000



add more processes (e.g. chemistry)

x 10

more ensemble members

x 10

ESCAPE

Computing at ECMWF

Historic Growth

Sustained Teraflops

1000
100
10
1
0.1
0.01
0.001
0.0001
1E-05

1978 1983 1988 1993 1998 2003 2008 2013 2018

Cray 1A
Cray X-MP/22
Cray X-MP/48
Cray Y-MP/8
Cray C90/12
Cray C90/16
Fujitsu VPP700/46
Fujitsu VPP700/116
Fujitsu VPP5000/38
Fujitsu VPP5000/100
IBM Power4 690
IBM Power4+ 690
IBM Power5+ 575
IBM Power6 575
IBM Power7 775
Cray XC30

Sustained Exaflop in 2033 ?

Sustained Teraflops

1000000
100000
10000
1000
100
10
1
0.1
0.01
0.001
0.0001
1E-05

1978 1983 1988 1993 1998 2003 2008 2013 2018 2023 2028 2033 2038

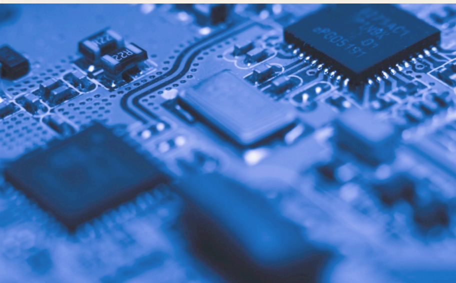
Cray 1A
Cray X-MP/22
Cray X-MP/48
Cray Y-MP/8
Cray C90/12
Cray C90/16
Fujitsu VPP700/46
Fujitsu VPP700/116
Fujitsu VPP5000/38
Fujitsu VPP5000/100
IBM Power4 690
IBM Power4+ 690
IBM Power5+ 575
IBM Power6 575
IBM Power7 775
Cray XC30

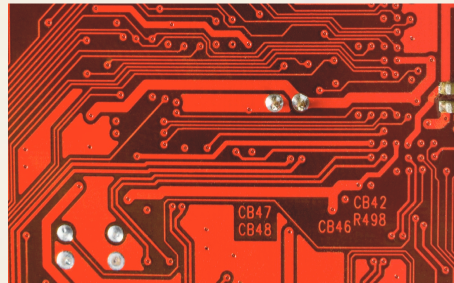
Funded by the
European Union

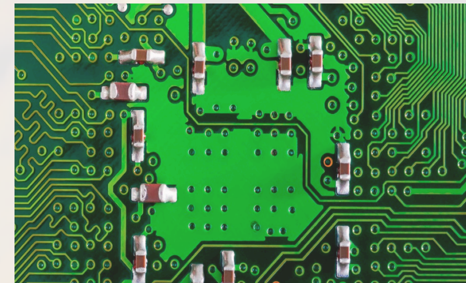
ESCAPE

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

not covered: readability, portability, maintainability





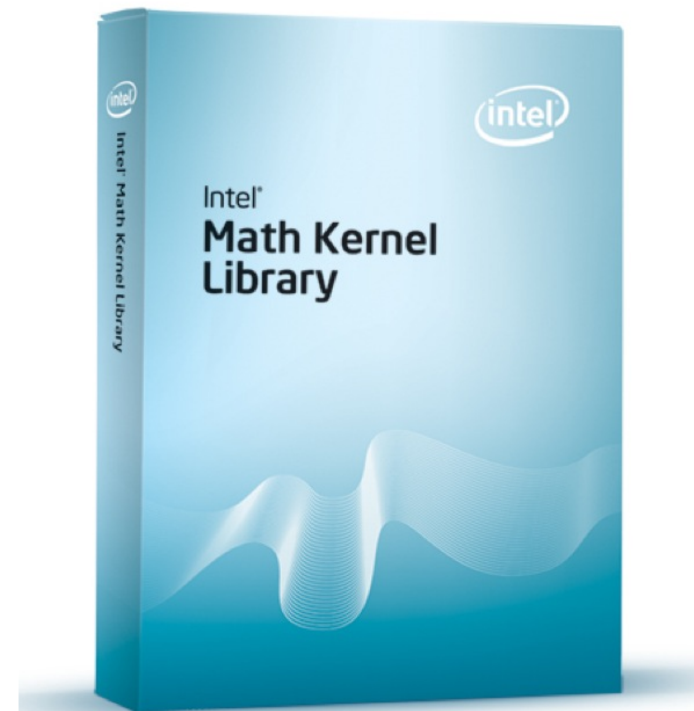


Funded by the
European Union

List of 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)
- 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)



Funded by the
European Union

List of recommendations

- **try if using libraries is fast enough**

Funded by the
European Union

ESCAPE

Compiler optimisation

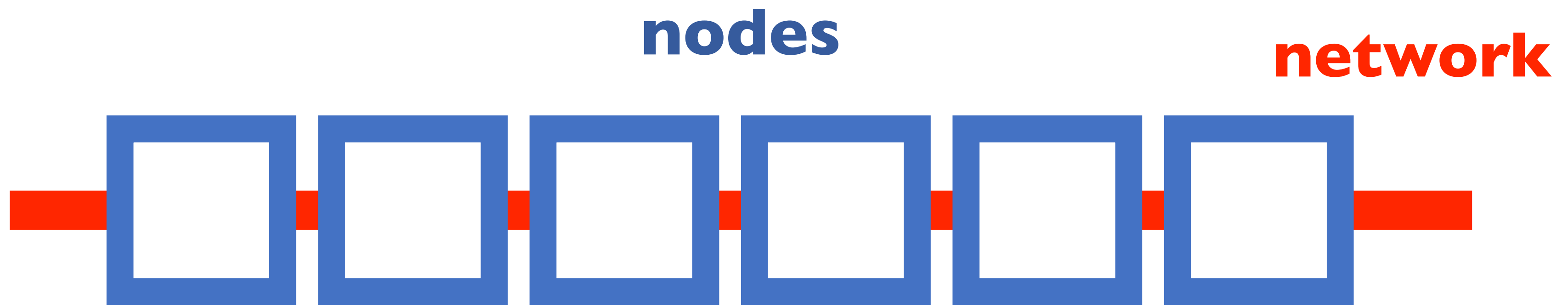
- compilers have optimisation flag -On (O0: no optimisation, O3: strong compiler optimisation)
- O3 is usually much faster than O2, but it can also be slower than O2
- O3 can produce completely wrong results!
- you can use different compiler flags for different files
- different compiler versions can 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

Funded by the
European Union

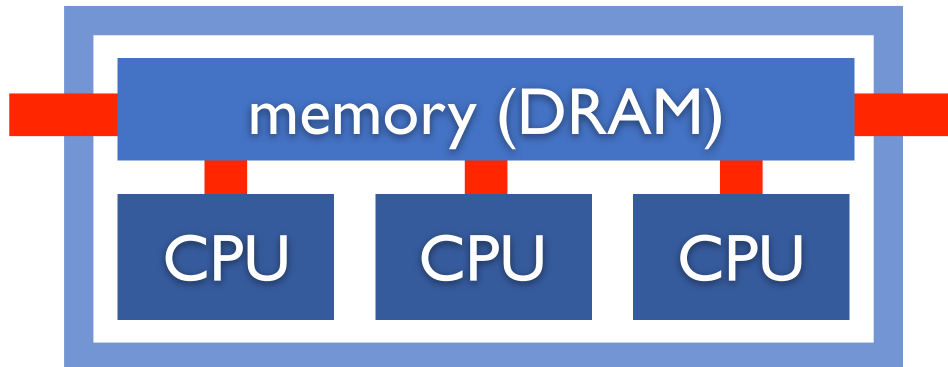
List of recommendations

- try if using libraries is fast enough
- **try to use compiler optimisation (be careful!)**

Supercomputer/Cluster



Node



Bottlenecks

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

Funded by the
European Union

List of recommendations

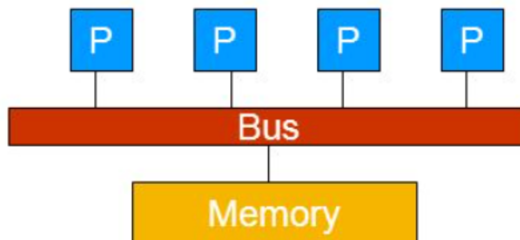
- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- **avoid unnecessary computation and communication**

Funded by the
European Union

ESCAPE

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
```

Funded by the European Union

ESCAPE

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
```

List of recommendations

- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- **decide yourself how to parallelise your code**

Funded by the
European Union

Shared memory: OpenMP

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
```

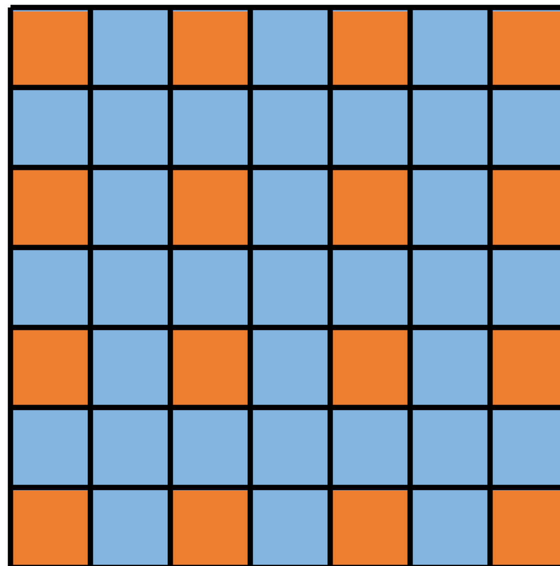
Funded by the
European Union

ESCAPE

Shared memory: OpenMP

Example 2: race conditions

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



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

List of recommendations

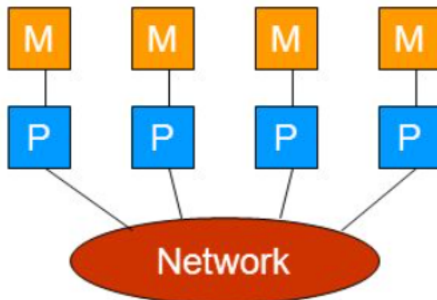
- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- decide yourself how to parallelise your code
- **let the threads do work that does not affect others**

Funded by the
European Union

ESCAPE

Distributed memory: MPI

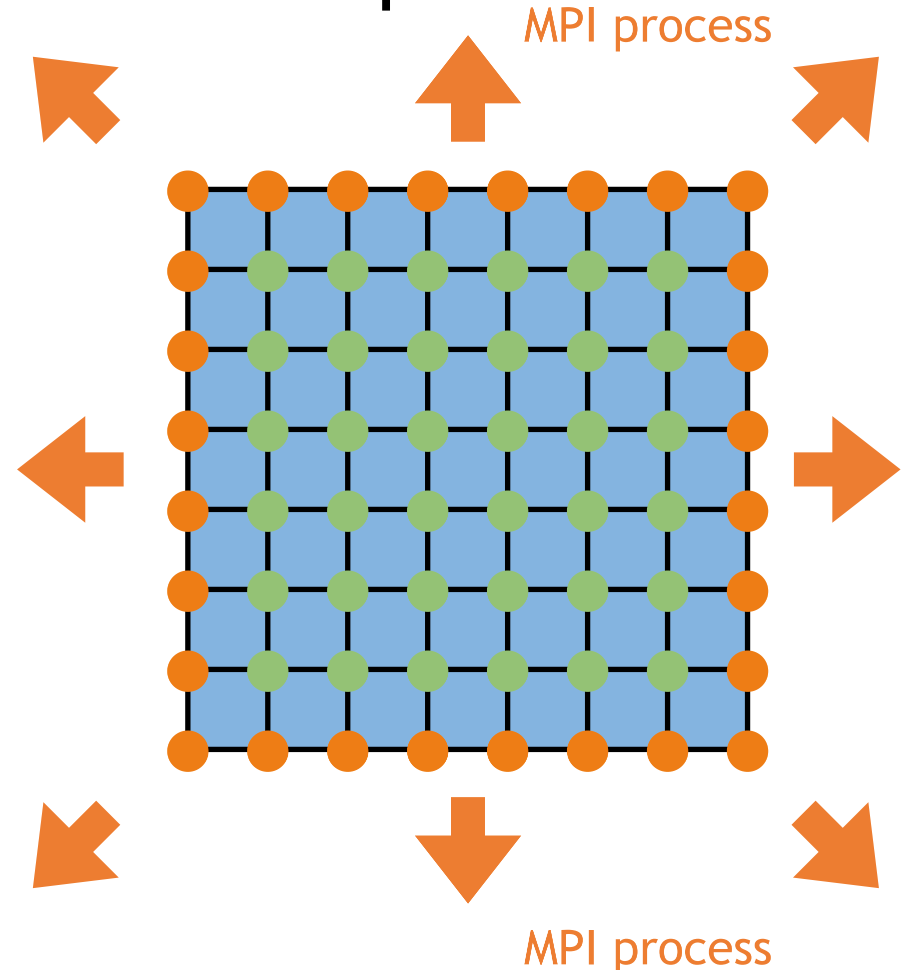
- many processes run on multiple nodes
- process can access only data on the node it is running
- use communication library MPI (Message Passing Interface) to access data on other nodes




```
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)
```

Overlap communication and computation

- Example: grid point method with only next neighbour communication:
    - compute values along processor boundaries first (orange) and send result to neighbours
    - compute interior points while the data is on its way (green)
- try to reduce the physical distance that data needs to travel (difficult)



List of recommendations

- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- decide yourself how to parallelise your code
- 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
```

Funded by the
European Union

ESCAPE

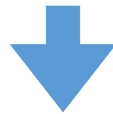
List of recommendations

- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- decide yourself how to parallelise your code
- let the threads do work that does not affect others
- overlap computation and communication
- **use data only once per time-step**

Contiguous memory access

double precision
floating point number (64bit)

memory



cache line
(often 128 Bytes)

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

Fortran (column major order):

```
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):

```
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
  }
}
```

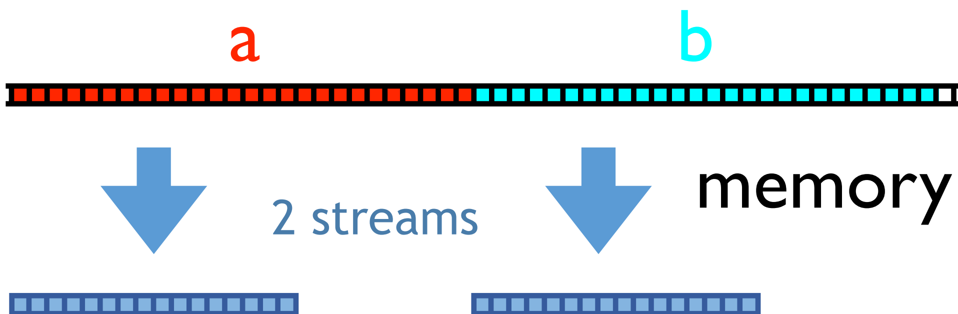
Contiguous memory access

best: one stream of data (good for prefetching)

example:

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

} 2 streams



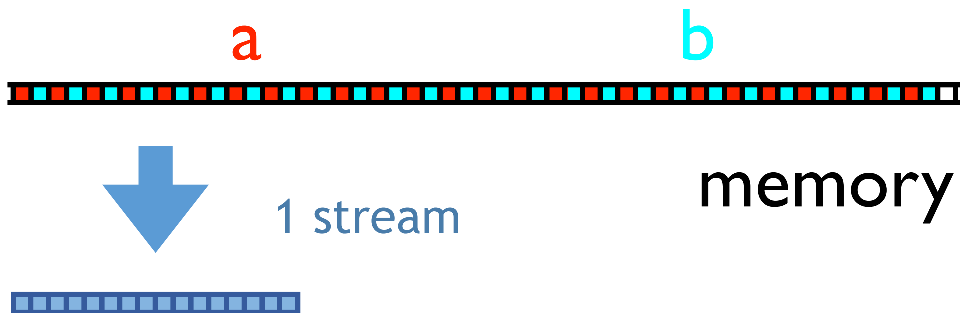
Contiguous memory access

best: one stream of data (good for prefetching)

example:

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

2 streams



better for prefetching:

```
real, dimension(N) :: a,b
real, dimension(2*N) :: c
integer :: i,N
do i=1,N
   c(2*i) = a(i)
   c(2*i-1) = b(i)
end do
do i=1,2*N,2
   c(i) = c(i) + c(i-1)
end do
do i=1,N
   a(i) = c(2*i)
end do
```

1 stream

efficient, if data needs to be rearranged only at beginning and end of simulation!

Funded by the
European Union

ESCAPE

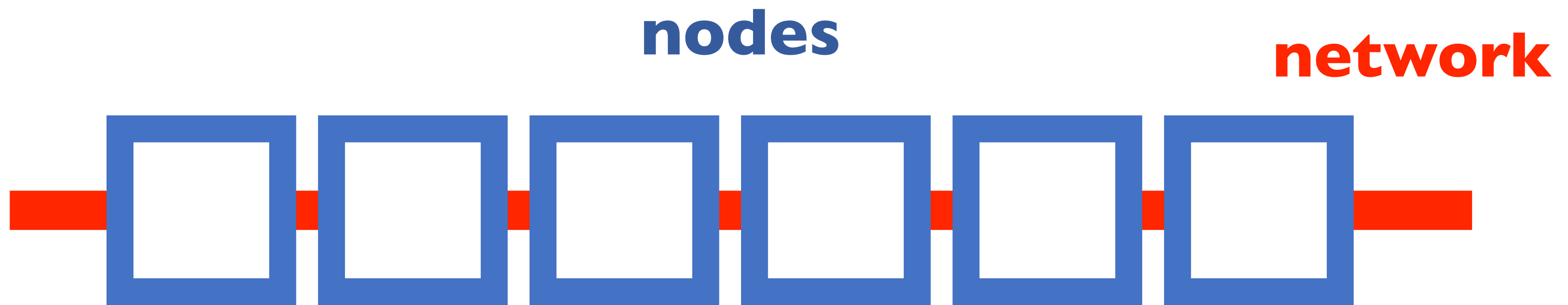
List of recommendations

- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- decide yourself how to parallelise your code
- let the threads do work that does not affect others
- overlap computation and communication
- use data only once per time-step
- **contiguous memory access**

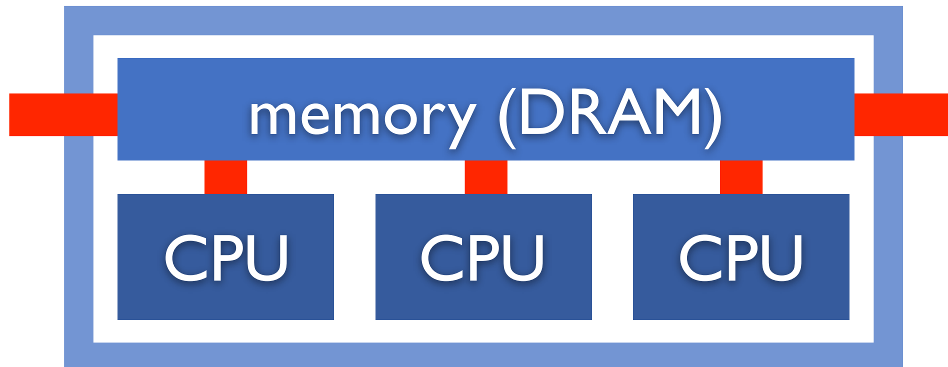
Funded by the
European Union

ESCAPE

Supercomputer/Cluster



Node



CPU

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

Funded by the
European Union

ESCAPE

Memory hierarchy inside one node

Socket 1

C 1 … C n

Load Buffer

Store Buffer

L1 … L1

L2 … L2

L3

QPI

MC

DRAM

DRAM

DRAM

DRAM

Registers/Buffers
~1 cycle <1ns

~3 cycles ~1ns

~12 cycles ~3ns

~38 cycles ~12ns

QPI ~40ns

~65ns

Socket 2

C 1 … C n

L1 … L1

L2 … L2

L3

QPI

MC

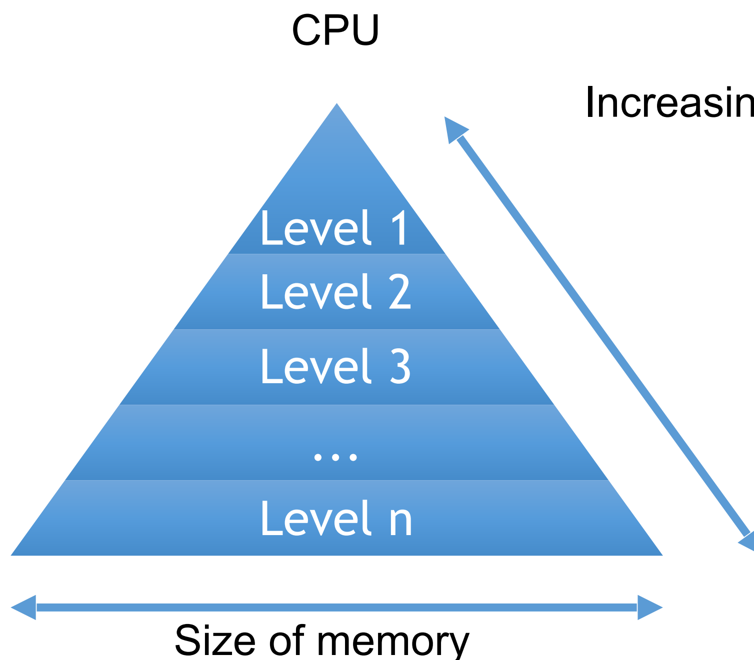
DRAM

DRAM

DRAM

DRAM

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

Cache hit – data found in cache
Cache miss – data not found in cache, thus must be copied from lower memory level
Capacity miss – cache runs out of space for new data
Conflict miss – more that one item is mapped to the same location in cache

Funded by the
European Union

ESCAPE

IFS: divide work into blocks with length NPROMA

Funded by the European Union

ESCAPE

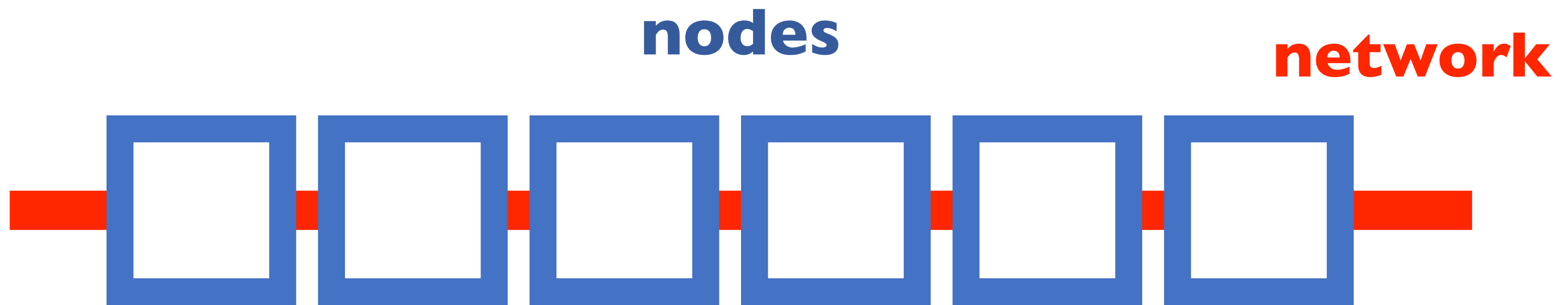
List of recommendations

- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- decide yourself how to parallelise your code
- 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**

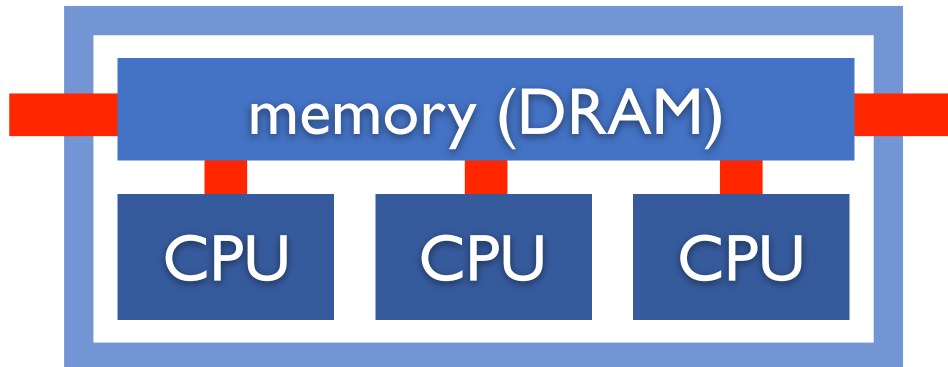
Funded by the European Union

ESCAPE

Supercomputer/Cluster



Node



Bottlenecks

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

Fast and slow operations

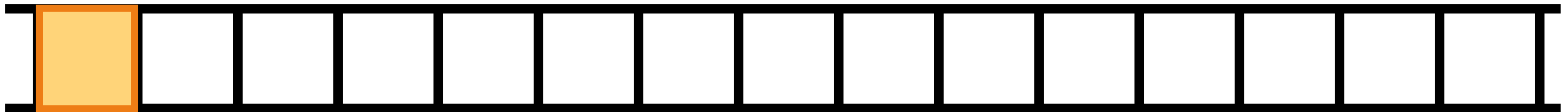
- In terms of cost
- Fast and inexpensive: add, multiply, sub, fma (fused multiply add)
- Medium: divide, modulus, sqrt
- Slow: power, trigonometric functions

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

Vectorisation

double precision
floating point number (64bit)

memory



Vectorisation

double precision floating point number (64bit)

memory

+
•

256bit register (BG/Q)

Vectorisation: initial version:

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

optimised for compiler vectorisation:

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

Vectorisation: initial version:

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

9.4s
14.4% vector operations

optimised for compiler vectorisation:

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

2.1s
73.9% vector operations

measurements: spectral element model NUMA, NPS

Funded by the European Union

ESCAPE

vector intrinsics (here for BG/Q):

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

```
11        do j=1,4 ! loop over points in y-direction
12           ... ! compute derivatives rho_x, ...
13           ! load always four floating point numbers:
14           vct_u = vec_ld(0, u(1,j,k))
15           vct_v = vec_ld(0, v(1,j,k))
16           vct_w = vec_ld(0, w(1,j,k))
17           vct_rhox = vec_ld(0, rho_x(1,j,k))
18           vct_rhoy = vec_ld(0, rho_y(1,j,k))
19           vct_rhoz = vec_ld(0, rho_z(1,j,k))
20           ! rhs = u*rho_x
21           vct_rhs = vec_mul(vct_u,vct_rhox)
22           ! rhs = rhs + v*rho_y
23           vct_rhs = vec_madd(vct_v,vct_rhoy,vct_rhs)
24           ! rhs = rhs + w*rho_z
25           vct_rhs = vec_madd(vct_w,vct_rhoz,vct_rhs)
26           ! write result from register into cache:
27           call vec_st(vct_rhs, 0, rhs(1,j,k))
28           ...
29        end do !j
30     end do !k
31  end do !e
```

1.0s
98.6% vector operations

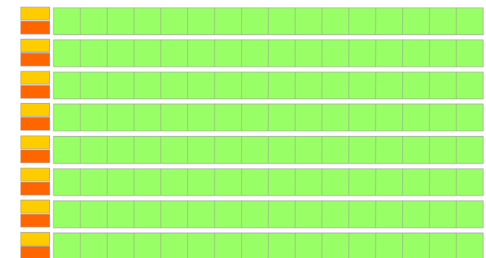
measurements: spectral element model NUMA, NPS

Funded by the
European Union

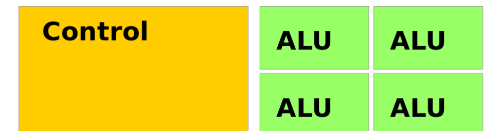
ESCAPE

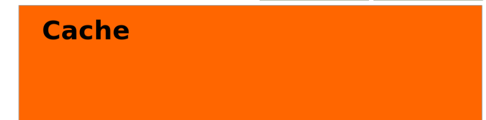
Accelerators (2 main types)

- GPU – Graphics Processing Unit
  - small number of instructions => requires host CPU
  - GPU/CPU interface (PCIx up to 16GB/sec today, in future NVLINK up to 80GB/sec between GPUs in same node)
  - more energy efficient than CPUs
  - high performance GPUs today mainly supplied by NVIDIA
  - lots of cores share one control unit
  - very little memory inside the GPU
- **INTEL (Xeon Phi, aka "MIC")**
  - "Knights Corner" from 2012 requires CPU host (via PCIx)
  - "Knights Landing" from 2016, does not require CPU host (64-72 cores), 512bit register (8 double, 16 single precision numbers)
  - "Knights Hill" from 2018



GPU



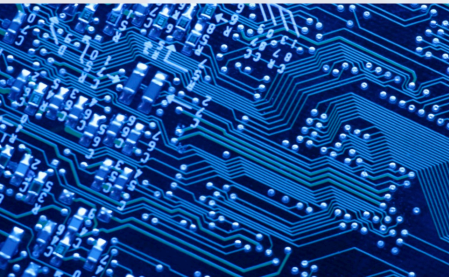


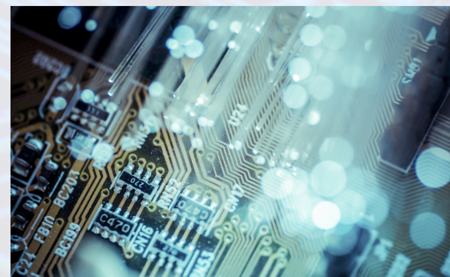
CPU

List of recommendations

- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- decide yourself how to parallelise your code
- 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?



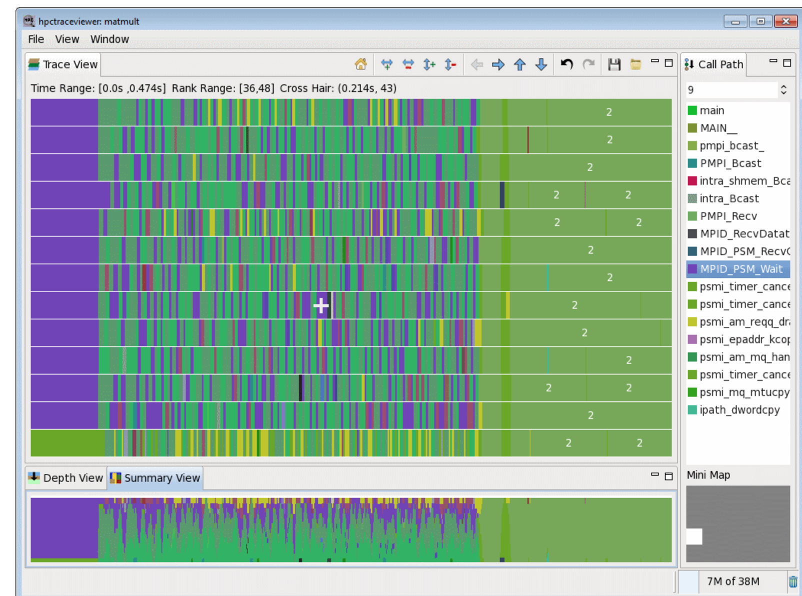


Funded by the
European Union

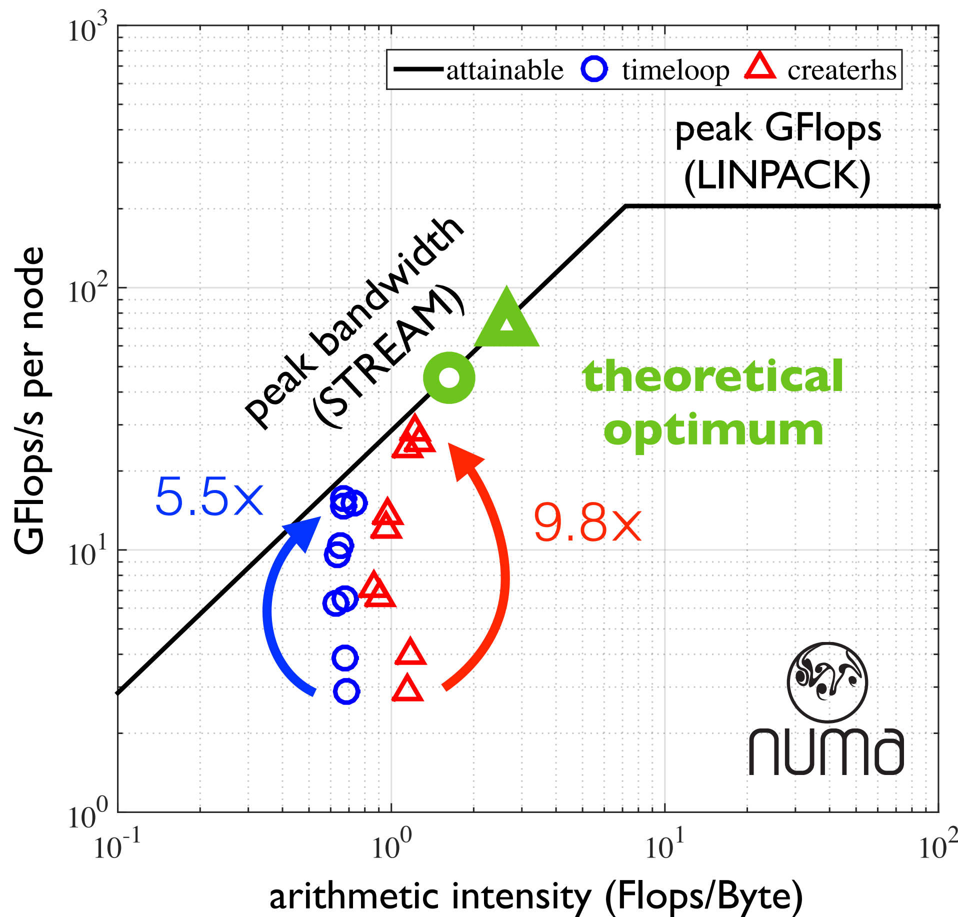
ESCAPE

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, ...



Roofline plot



blue: entire timeloop

red: main computational kernel

data points: different optimization stages

measurements: spectral element model NUMA, NPS

ESCAPE

Strong scaling efficiency

baroclinic instability, p=3, 3.0km horizontal resolution

strong scaling: fixed total amount of work
weak scaling: fixed amount of work per processor

99.1% strong scaling efficiency

3.14M threads

model days per wallclock day

number of threads

$\times 10^6$

NUMA

measurements: spectral element model NUMA, NPS

Funded by the
European Union

ESCAPE

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

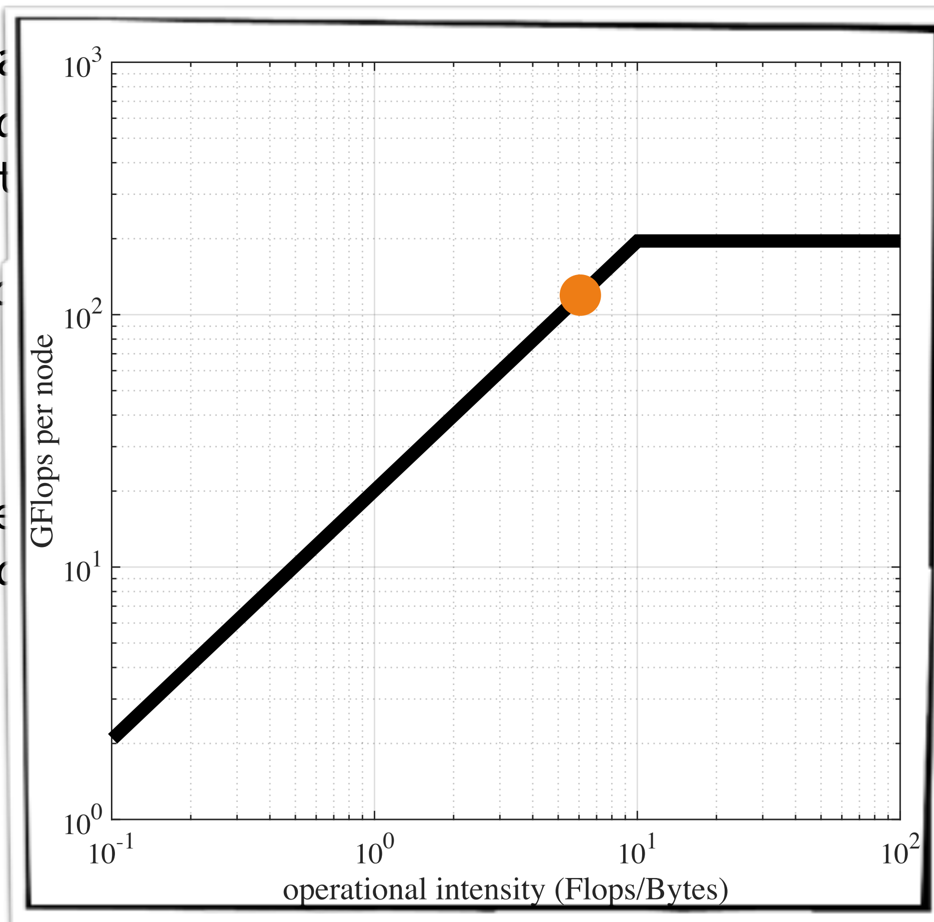
memory:

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

Funded by the European Union

Create performance model

exa
rec
int
do
(
e
end



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

		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
					1.92E+13	6.4E+12
					2400	800
intensity	6.25			runtime in seconds		160.0

ESCAPE

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

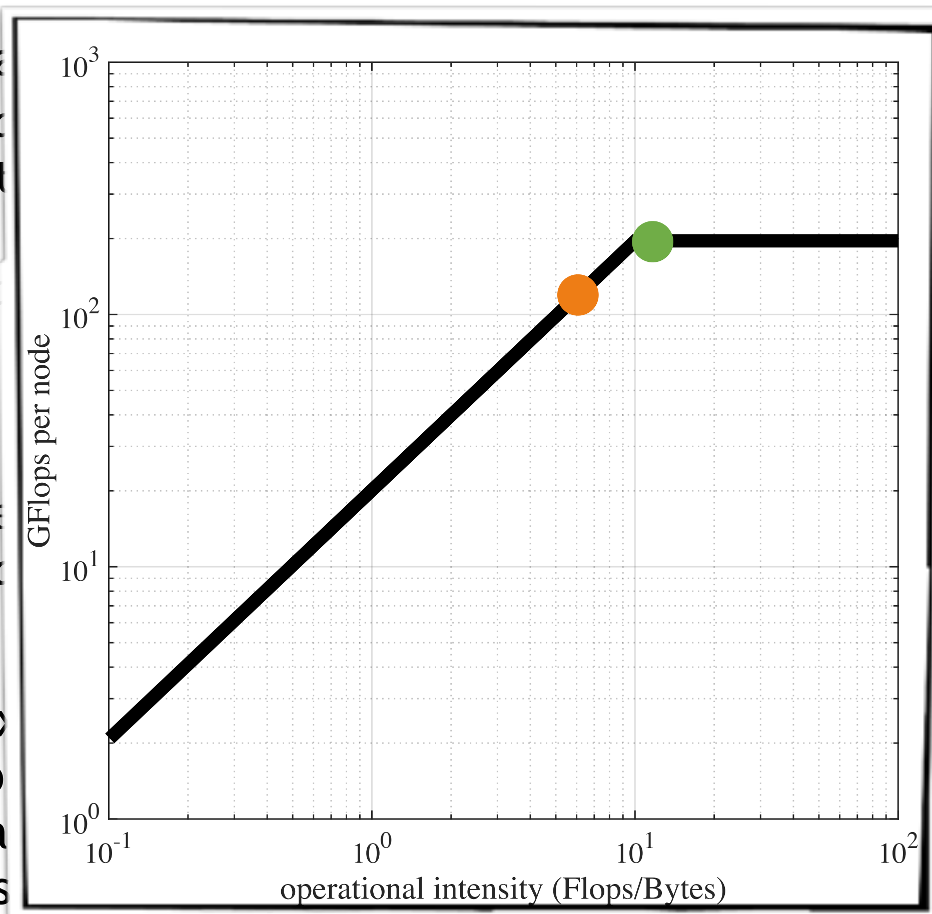
next step: distinguish between worst case (all data has to be loaded from memory) and best case (previously used data is still in cache)

Funded by the European Union

Create performance model

exa
rec
int
do
c
e
enc

nex
wo
loa
cas
still in cache)



parameters:

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

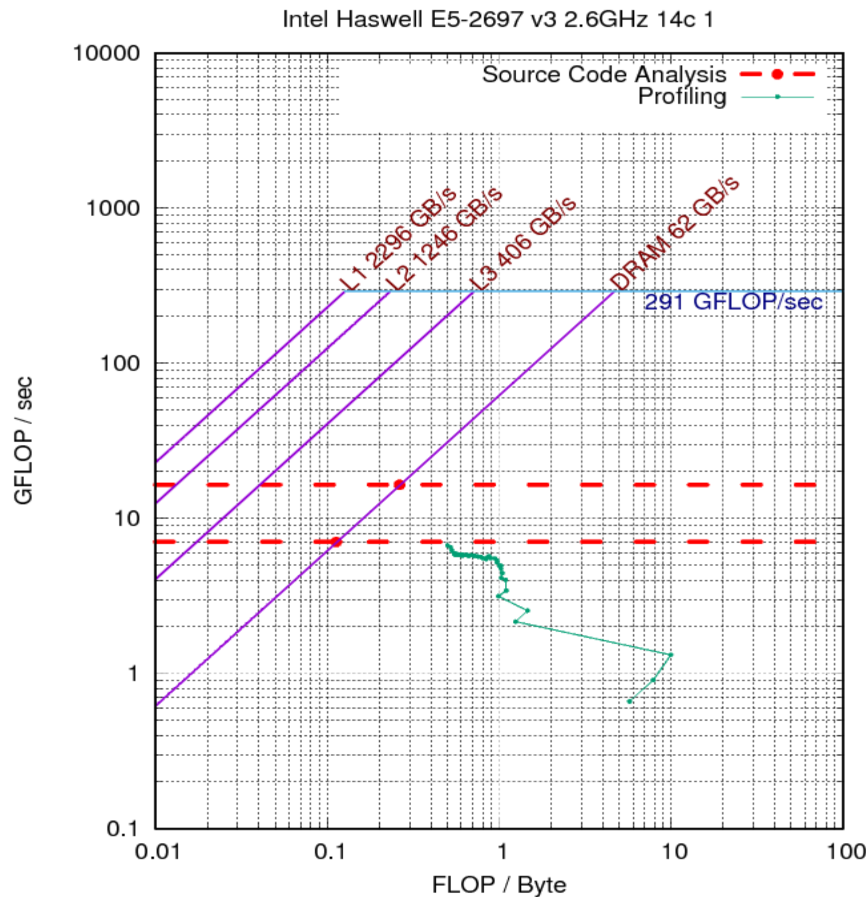
floating point operations:

function	operations per step	
main	2*N*M	2E+11
total GFlops for all steps		20000
runtime		100.0

		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
					6.4E+12	6.4E+12
					800	800
intensity	12.5			runtime in seconds		80.0

Funded by the
European Union

Roofline model



measurements and analysis performed by PSNC for spectral transform dwarf

List of recommendations

- try if using libraries is fast enough
- try to use compiler optimisation (be careful!)
- avoid unnecessary computation and communication
- decide yourself how to parallelise your c
- let the threads do work that does not affe
- overlap computation and communicatior
- use data only once per time-step
- contiguous memory access
- try to fit data into cache
- make good use of vectorisation
- **measure performance and compare with expectations**

open question

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