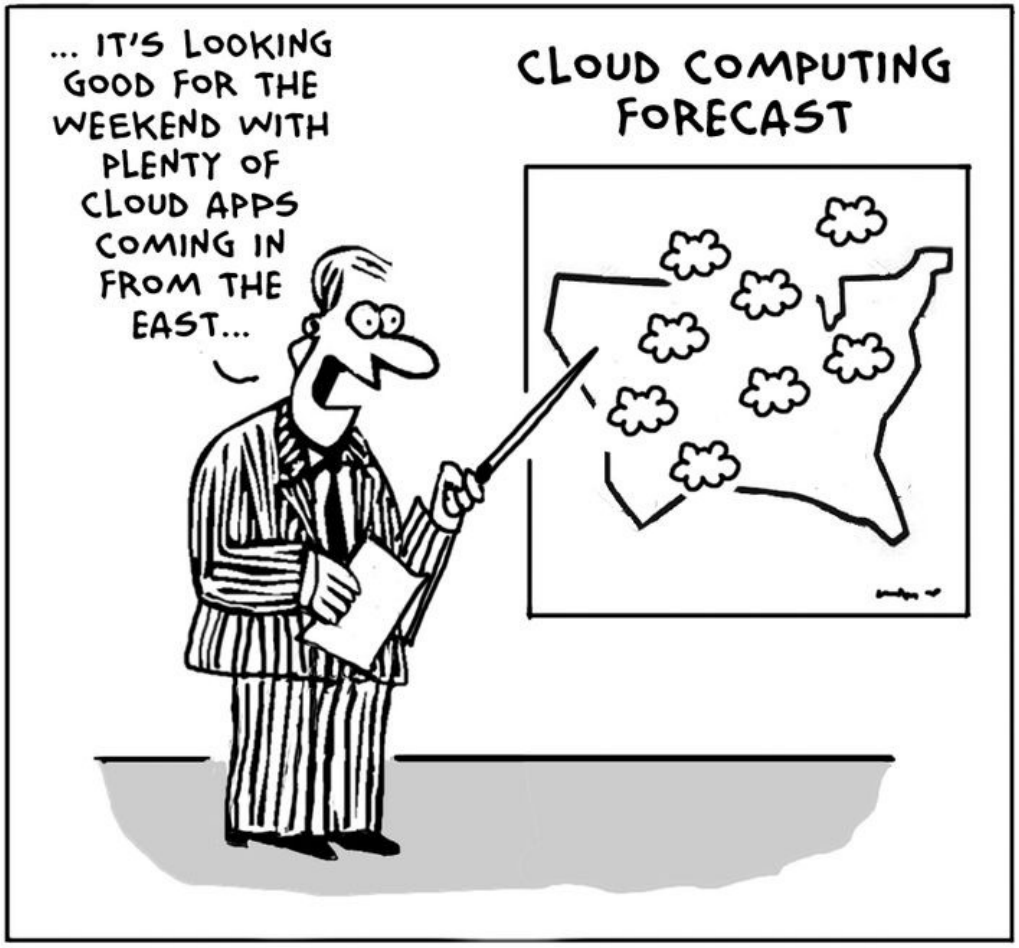
Computer Architectures

Alexander Knebe, *Universidad Autonoma de Madrid*
Violeta González Pérez



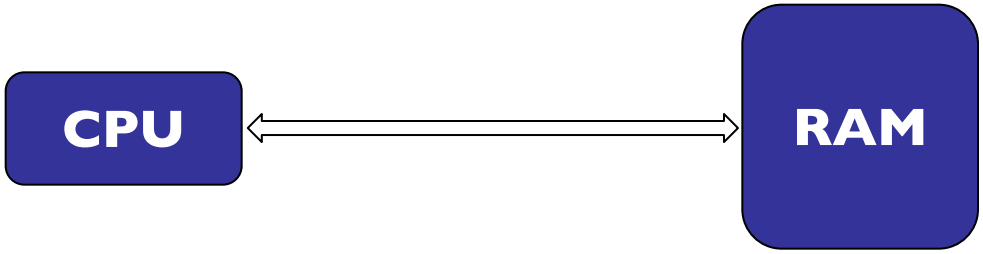
- architectures
- real machines
- computing concepts
- parallel programming

- **architectures**
- real machines
- computing concepts
- parallel programming

Computer Architectures

- serial machine

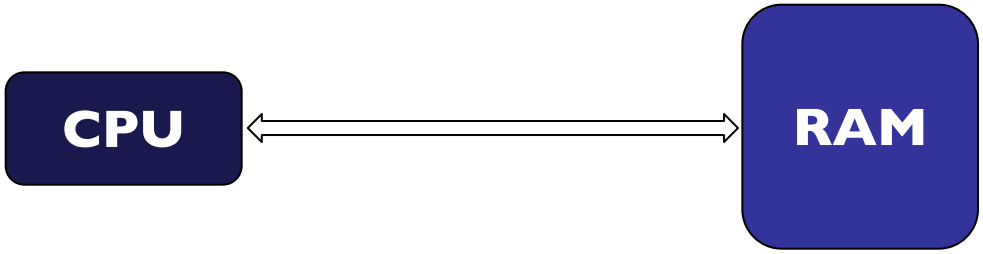
=



Computer Architectures

- serial machine

=



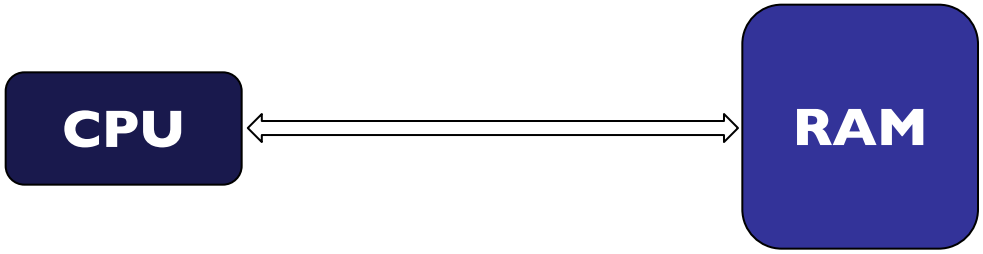
CPU:

- very primitive commands,
  obtained from compilers or interpreters of higher-level languages

Computer Architectures

- serial machine

=



CPU:

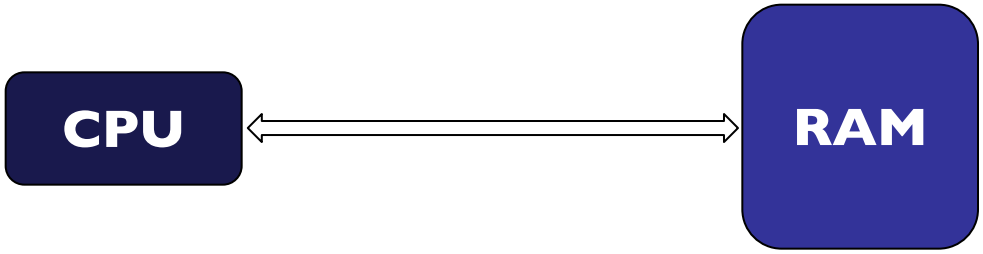
- very primitive commands,
  obtained from compilers or interpreters of higher-level languages

- cycle chain:
  - ***fetch*** – get instruction and/or data from memory
  - ***decode*** – store instruction and/or data in register
  - ***execute*** – perform instruction

Computer Architectures

- serial machine

=



CPU:

- very primitive commands,
  obtained from compilers or interpreters of higher-level languages

- cycle chain:
  - ***fetch*** – get **instruction** and/or data from memory
  - ***decode*** – store **instruction** and/or data in register
  - ***execute*** – perform **instruction**

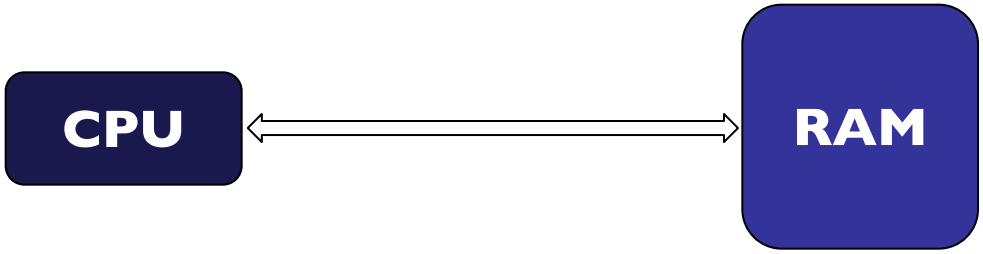
arithmetical/logical instructions: +, -, *, /, bitshift, if

What is a bitshift (>>, <<) good for?
Make a short C program to test this.

Computer Architectures

- serial machine

=



CPU:

- very primitive commands,
  obtained from compilers or interpreters of higher-level languages

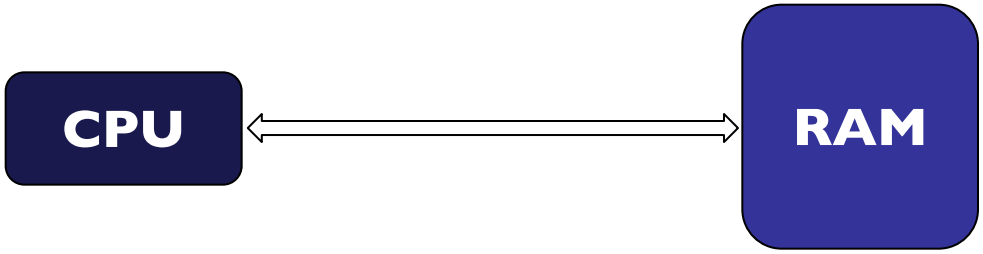
- cycle chain:
  - ***fetch*** – get instruction and/or data from memory
  - ***decode*** – store instruction and/or data in register
  - ***execute*** – perform instruction

- some CPU allow multi-threading,
  i.e. already fetch next instruction while still executing

Computer Architectures

- serial machine

=



CPU:

- execution time: $t = n_i \times \mathrm{CPI} \times t_c$

number of instructions

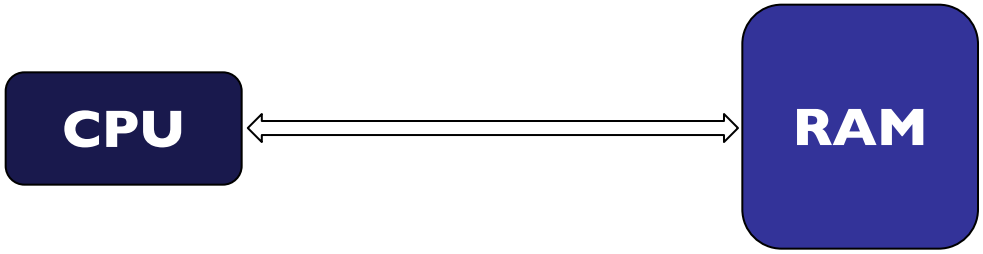
cycles per instruction
(e.g., '+' requires less cycles than '*')

time per cycle

Computer Architectures

- serial machine

=



CPU:

- execution time: $t = n_i \times \mathrm{CPI} \times t_c$

number of instructions

cycles per instruction

time per cycle

speed-ups: **improve your algorithm to require less instructions**

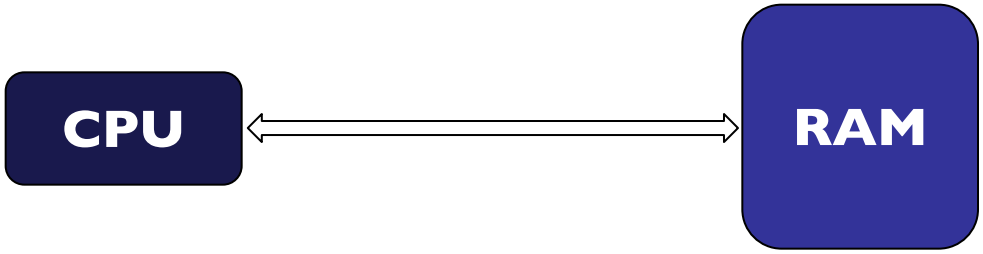
example:
a factor like "3/(8piG)" inside a for-loop should be avoided;
define FAC=3/(8piG) outside the loop and use FAC inside the loop instead…

why?

Computer Architectures

- serial machine

=



CPU:

- execution time: $t = n_i \times \mathrm{CPI} \times t_c$

number of instructions

cycles per instruction (how many cycles does your instruction require)

time per cycle

Complete the code to test this:

```
#include <stdio.h>
#include <math.h>
#include <time.h>

void main(){
  double x = 2.;

  clock_t begin = clock();
  double a = x*x*x;
  clock_t end = clock();
  double time = 1000.*(end -
begin)/CLOCKS_PER_SEC; //ms
  printf("Time x*x*x=%E ms \n",time);
```

speed-ups:

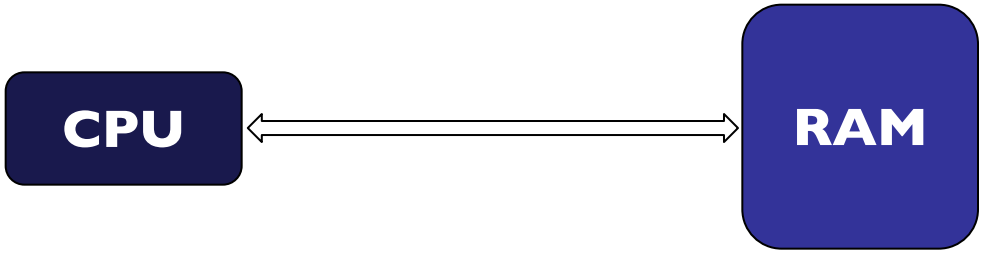
improve your algorithm to use more adequate instructions

example:
avoid at all costs `pow()`, `log()`, etc.,
e.g. `pow(x,2)` should be replaced with `x*x`

Computer Architectures

- serial machine

=



CPU:

- execution time: $t = n_i \times \mathrm{CPI} \times t_c$

number of instructions

cycles per instruction

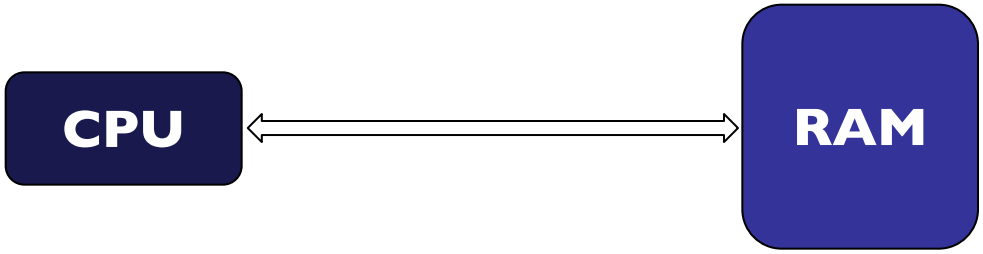
time per cycle

speed-ups: **buy a machine with higher clock-frequency**

Computer Architectures

- serial machine

=



CPU:

- execution time: $t = n_i \times \mathrm{CPI} \times t_c$

number of instructions

cycles per instruction

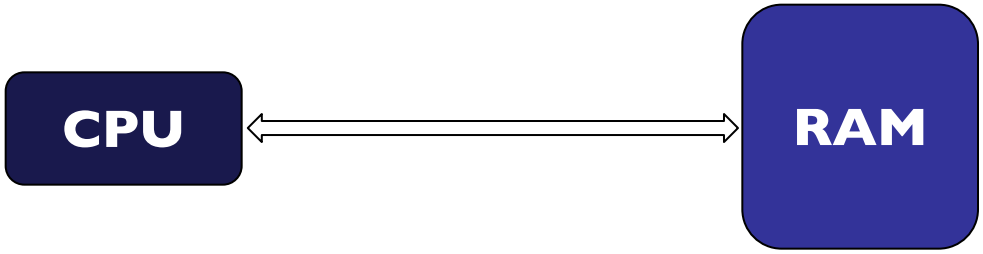
time per cycle

speed-ups: **improve your algorithm!**

Computer Architectures

- serial machine

=



CPU:

- execution time: $t = n_i \times \mathrm{CPI} \times t_c$

number of instructions

cycles per instruction

time per cycle

Check the speed of your laptop!

$ sudo lscpu

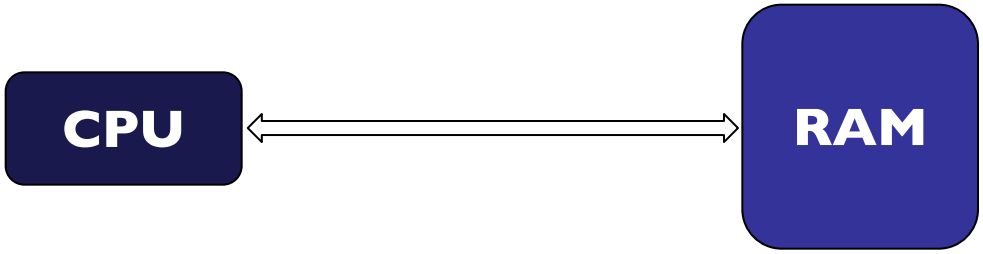
(search CPU MHz)

speed-ups: **...or wait for technology to advance ;-)**

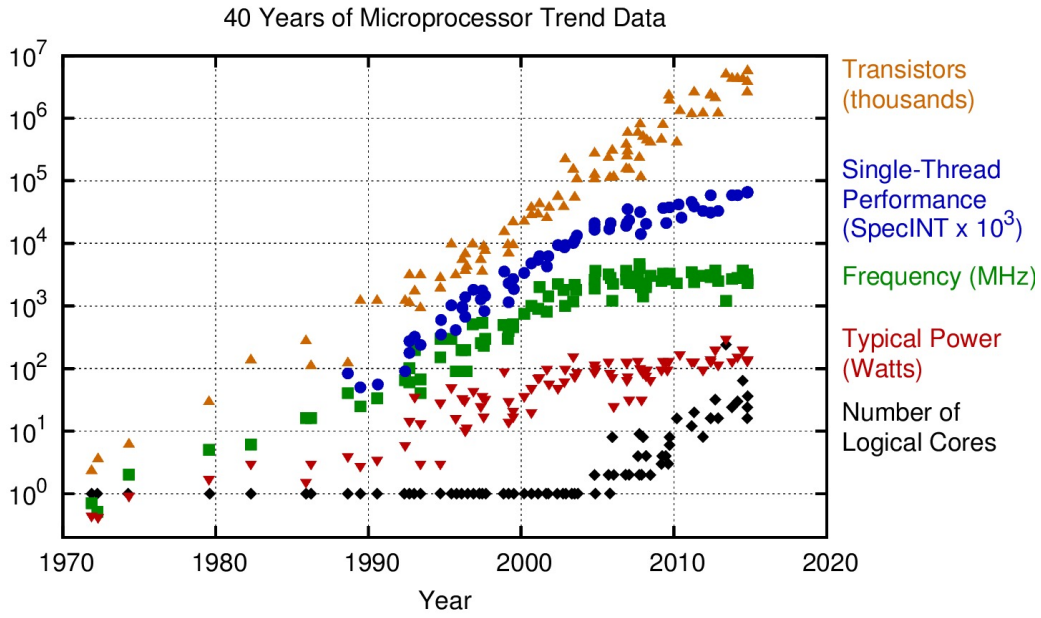
Computer Architectures

- serial machine

=



CPU – evolution during the past years:

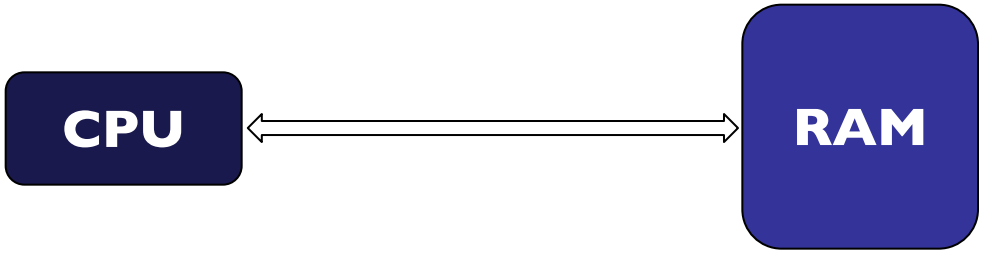


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

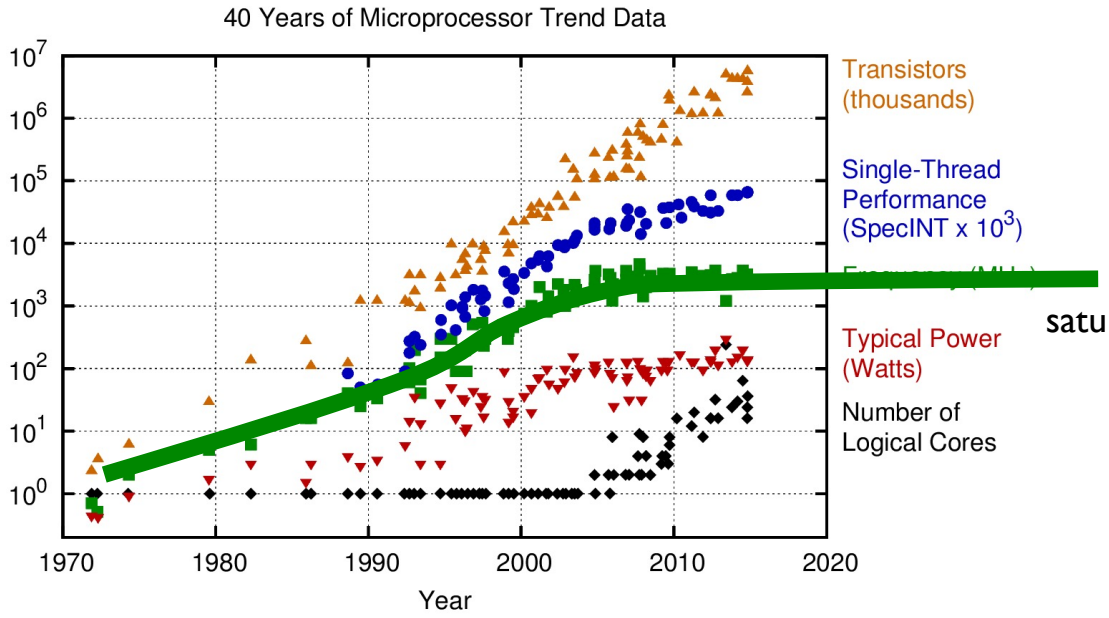
Computer Architectures

- serial machine

=



CPU – evolution during the past years:



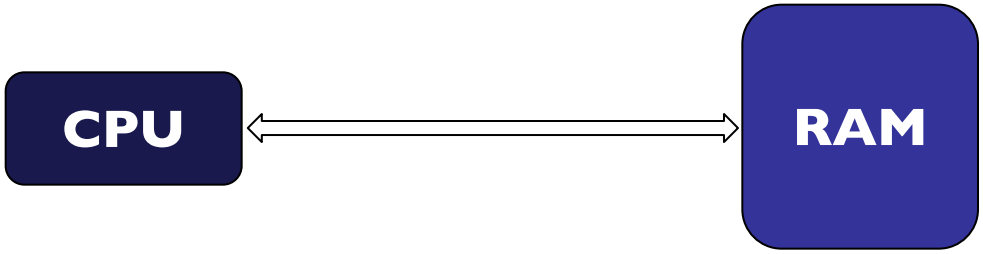
clock speed: saturation level reached!

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

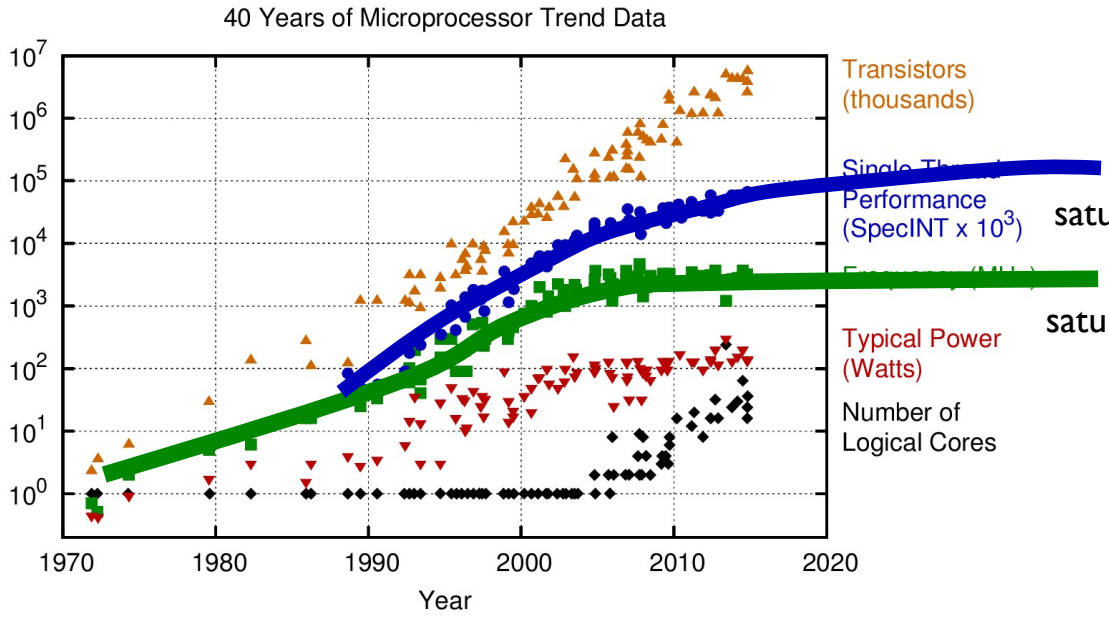
Computer Architectures

- serial machine

=



CPU – evolution during the past years:



single thread: saturation approaching!

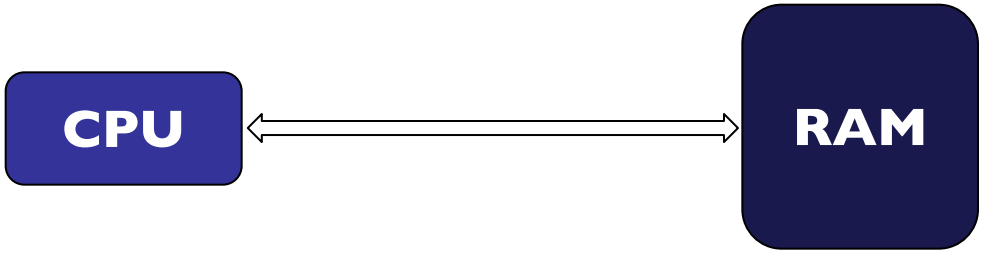
clock speed: saturation level reached!

Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Computer Architectures

- serial machine

=



RAM:

- **R**andom **A**ccess **M**emory, i.e. read and write

- storage in binary system:
  - 1 bit = 0 or 1
  - 8 bits = 1 byte
  - 4 bytes = 1 float (=32 bits, standard for 32-bit architectures)
  - 8 bytes = 1 double (=64 bits, standard for 64-bit architectures)

Check the architecture of your laptop and the number of CPUs and threads you have:
$ sudo lscpu

Computer Architectures

- serial machine

=

RAM:

- **R**andom **A**ccess **M**emory, i.e. read and write

- storage in binary system:
  - 1 bit = 0 or 1
  - 8 bits = 1 byte
  - 4 bytes = 1 float (=32 bits, standard for 32-bit architectures)
  - 8 bytes = 1 double (=64 bits, standard for 64-bit architectures)

- latency = time for memory access (bus width also relevant)

Computer Architectures

- serial machine

=

RAM:

- **R**andom **A**ccess **M**emory, i.e. read and write

- storage in binary system:
  - 1 bit = 0 or 1
  - 8 bits = 1 byte
  - 4 bytes = 1 float (=32 bits, standard for 32-bit architectures)
  - 8 bytes = 1 double (=64 bits, standard for 64-bit architectures)

- latency = time for memory access (bus width also relevant)

- speed-ups:
  - multi-threading CPU's
  - larger bus width

Computer Architectures

- serial machine

=

RAM:

- **R**andom **A**ccess **M**emory, i.e. read and write

- storage in binary system:
  - 1 bit = 0 or 1
  - 8 bits = 1 byte
  - 4 bytes = 1 float (=32 bits, standard for 32-bit architectures)
  - 8 bytes = 1 double (=64 bits, standard for 64-bit architectures)

- latency = time for memory access (bus width also relevant)

- speed-ups:
  - multi-threading CPU's
  - ***larger bus width:***
    - '80s 8-bit wide
    - '90s 16-bit wide
    - '00s 32-bit wide
    - today 64-bit wide

(internal 'highway' for data transfer)

Computer Architectures

- serial machine

=

RAM:

- **R**andom **A**ccess **M**emory, i.e. read and write

- storage in binary system:
  - 1 bit = 0 or 1
  - 8 bits = 1 byte
  - 4 bytes = 1 float (=32 bits, standard for 32-bit architectures)
  - 8 bytes = 1 double (=64 bits, standard for 64-bit architectures)

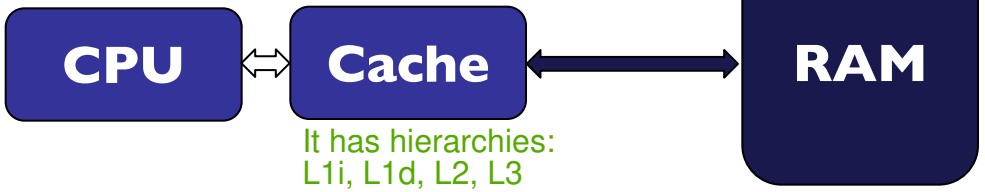
- latency = time for memory access (bus width also relevant)

- speed-ups:
  - multi-threading CPU's
  - larger bus width
  - clever usage of Cache

Computer Architectures

- serial machine

=



RAM:

- **R**andom **A**ccess **M**emory, i.e. read and write
- storage in binary system:
  - 1 bit = 0 or 1
  - 8 bits = 1 byte
  - 4 bytes = 1 float (=32 bits, standard for 32-bit architectures)
  - 8 bytes = 1 double (=64 bits, standard for 64-bit architectures)
- latency = time for memory access (bus width also relevant)
- speed-ups:
  - multi-threading CPU's
  - larger bus width
  - ***clever usage of Cache***

Explore the Cache of your laptop with

$ sudo lscpu

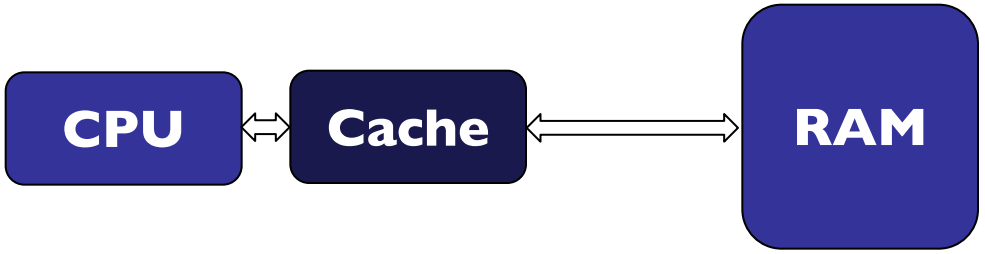
and compare it with your RAM:

$ grep MemTotal /proc/meminfo

Computer Architectures

- serial machine

=



Cache:

- **R**andom **A**ccess **M**emory, i.e. read and write
- built into motherboard next to CPU
- when 'fetch a[i]' also 'fetch a[i+1]' into Cache (in fact, full lines or pages are "cached")
- nowadays multiple Cache levels
- bad programming will lead to "Cache misses":

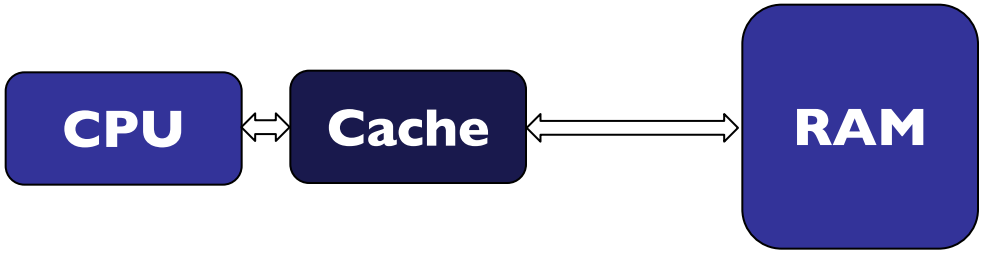
$$t = f \times t_{\text{Cache}} + (1 - f) \times t_{RAM}$$

Cache hit rate **Cache access time** **RAM access time**

Computer Architectures

- serial machine

=



Cache:

- **R**andom **A**ccess **M**emory, i.e. read and write
- built into motherboard next to CPU
- when 'fetch a[i]' also 'fetch a[i+1]' into Cache (in fact, full lines or pages are "cached")
- nowadays multiple Cache levels
- bad programming will lead to "Cache misses":

$$t = f \times t_{\text{Cache}} + (1 - f) \times t_{RAM}$$

Cache hit rate **Cache access time** **RAM access time**

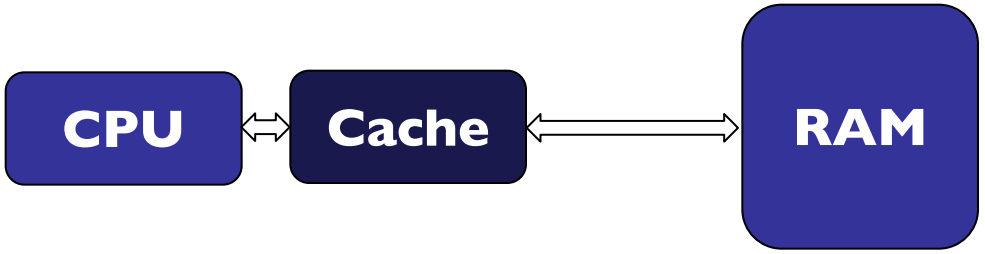
example:

$$t_{Cache} = 10ns, \quad t_{RAM} = 100ns$$
$$t_{Cache} = 10ns, \quad t_{RAM} = 100ns$$

Computer Architectures

- serial machine

=



Cache:

- **R**andom **A**ccess **M**emory, i.e. read and write
- built into motherboard next to CPU
- when 'fetch a[i]' also 'fetch a[i+1]' into Cache (in fact, full lines or pages are "cached")
- nowadays multiple Cache levels
- bad programming will lead to "Cache misses":

- $t = f \times t_{\text{Cache}} + (1 - f) \times t_{RAM}$

Cache hit rate **Cache access time** **RAM access time**

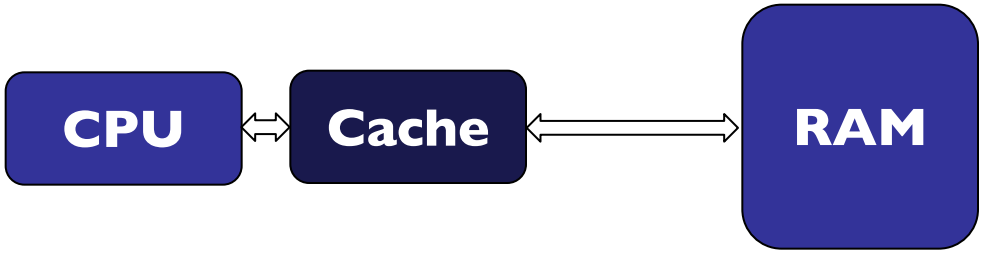
example: $f = 0.1, \quad t_{Cache} = 10ns, \quad t_{RAM} = 100ns$

$f = 0.9, \quad t_{Cache} = 10ns, \quad t_{RAM} = 100ns$

Computer Architectures

- serial machine

=



Cache:

- **R**andom **A**ccess **M**emory, i.e. read and write
- built into motherboard next to CPU
- when 'fetch a[i]' also 'fetch a[i+1]' into Cache (in fact, full lines or pages are "cached")
- nowadays multiple Cache levels
- bad programming will lead to "Cache misses":

$$t = f \times t_{\text{Cache}} + (1 - f) \times t_{RAM}$$

Cache hit rate **Cache access time** **RAM access time**

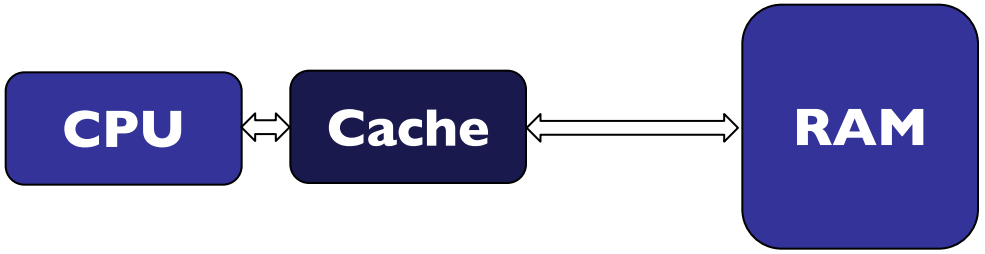
example:

$$f = 0.1, \quad t_{Cache} = 10ns, \quad t_{RAM} = 100ns \quad \Rightarrow 91ns$$
$$f = 0.9, \quad t_{Cache} = 10ns, \quad t_{RAM} = 100ns \quad \Rightarrow 19ns$$

Computer Architectures

- serial machine

=



Cache:

- **R**andom **A**ccess **M**emory, i.e. read and write
- built into motherboard next to CPU
- when 'fetch a[i]' also 'fetch a[i+1]' into Cache (in fact, full lines or pages are "cached")
- nowadays multiple Cache levels
- bad programming will lead to "Cache misses":

- $t = f \times t_{\text{Cache}} + (1 - f) \times t_{RAM}$

Cache hit rate **Cache access time** **RAM access time**

example:

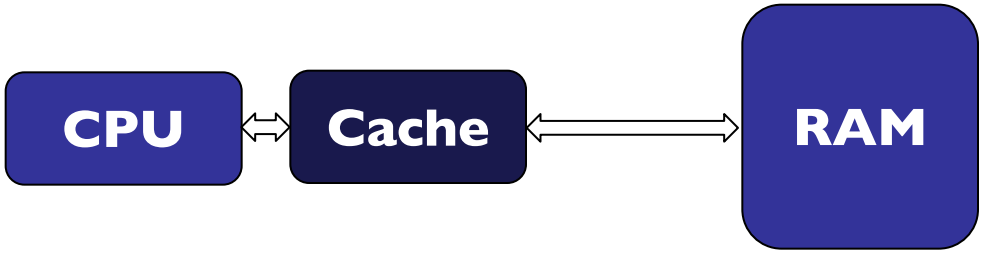
$$f = 0.1, \quad t_{Cache} = 10ns, \quad t_{RAM} = 100ns \quad \Rightarrow 91ns$$
$$f = 0.9, \quad t_{Cache} = 10ns, \quad t_{RAM} = 100ns \quad \Rightarrow 19ns$$

factor of 4.5!

Computer Architectures

- serial machine

=



Cache:

- **R**andom **A**ccess **M**emory, i.e. read and write
- built into motherboard next to CPU
- when 'fetch a[i]' also 'fetch a[i+1]' into Cache (in fact, full lines or pages are "cached")
- nowadays multiple Cache levels
- bad programming will lead to "Cache misses":

$$t = f \times t_{\text{Cache}} + (1 - f) \times t_{RAM}$$

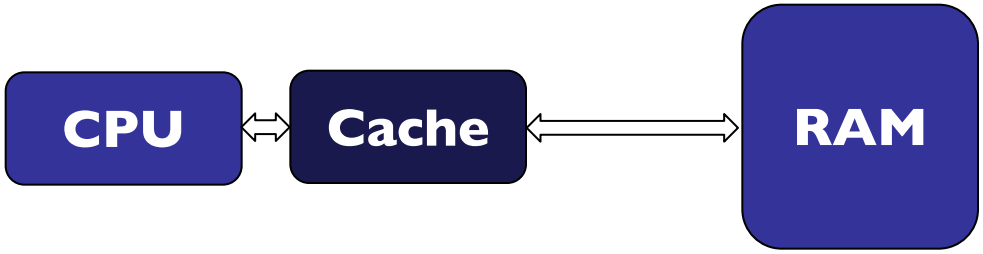
Cache hit rate — Cache access time — RAM access time

design your code in order to access contiguous memory blocks!

Computer Architectures

- serial machine

=

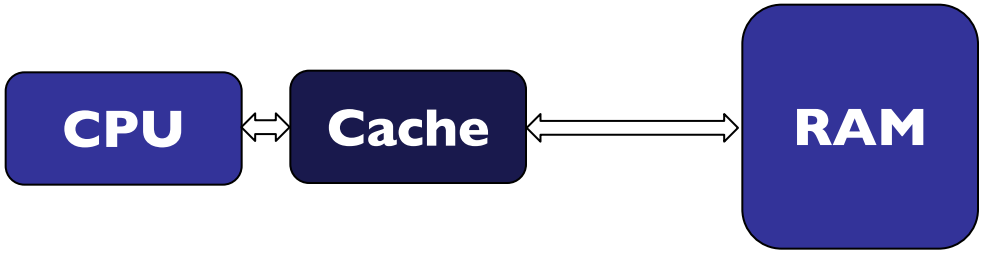


- Cache hits and misses:
  - 2D array in *C*: `density[2][3]`

Computer Architectures

- serial machine

=

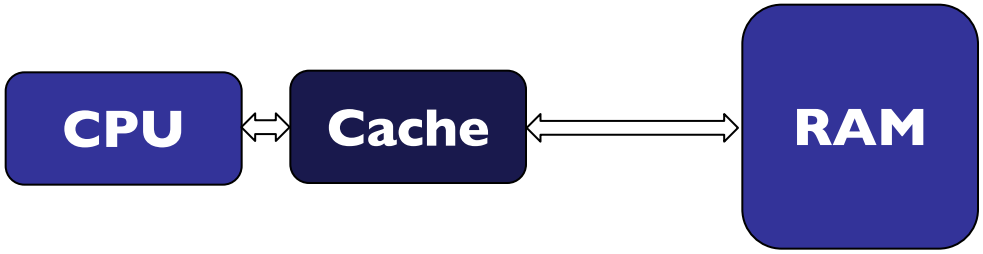


- Cache hits and misses:
  - 2D array in *C*: `density[2][3]`
  - memory alignment:

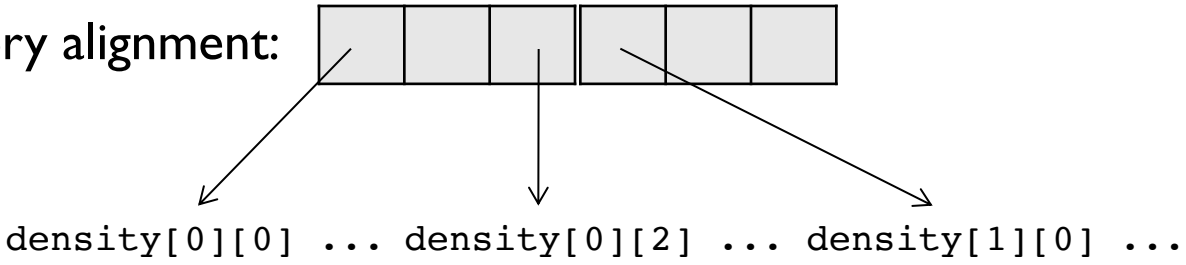
Computer Architectures

- serial machine

=



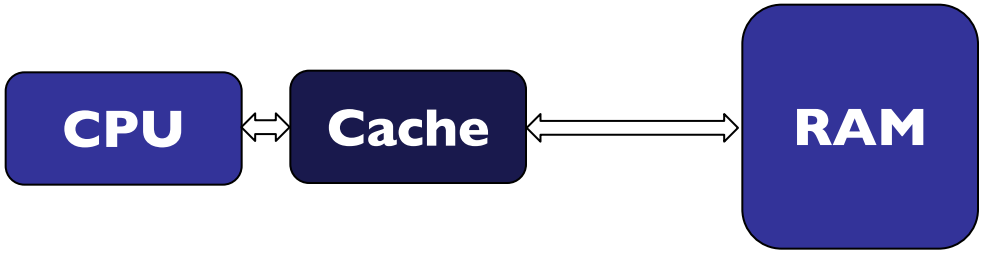
- Cache hits and misses:
  - 2D array in *C*: `density[2][3]`
  - memory alignment:



Computer Architectures

- serial machine

=

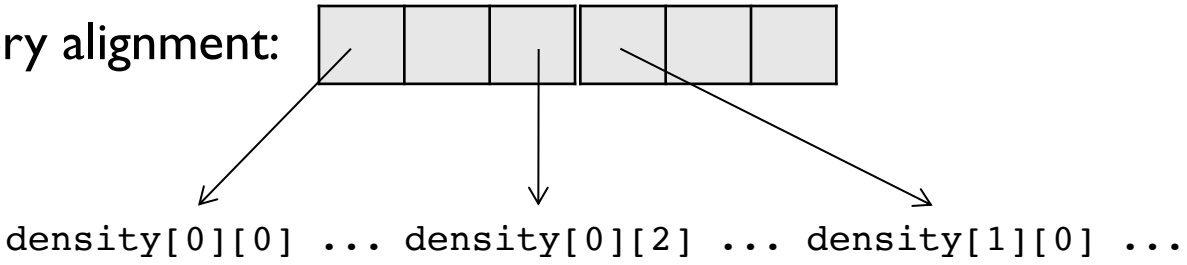


- Cache hits and misses:

```
for(i=0; i<3; i++)
   for(j=0; j<2; j++)
      whatever(density[j][i]);
```

BAD!

  - 2D array in *C*: `density[2][3]`
  - memory alignment:



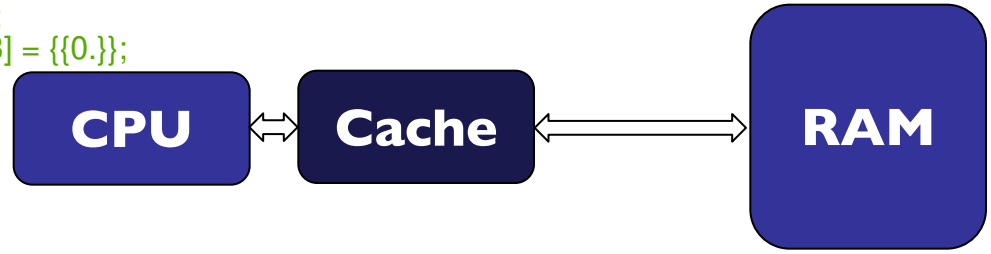
Computer Architectures

- serial machine

Test by completing the code:

```
#include <stdio.h>
#include <stdlib.h> Populate the array with random numbers from 0 to 99 with rand()%100
#include <time.h>

void main(){
  int i=0, j=0;
  double time=0.;
  clock_t begin, end;
  double density[2][3] = {{0.}};
```

=

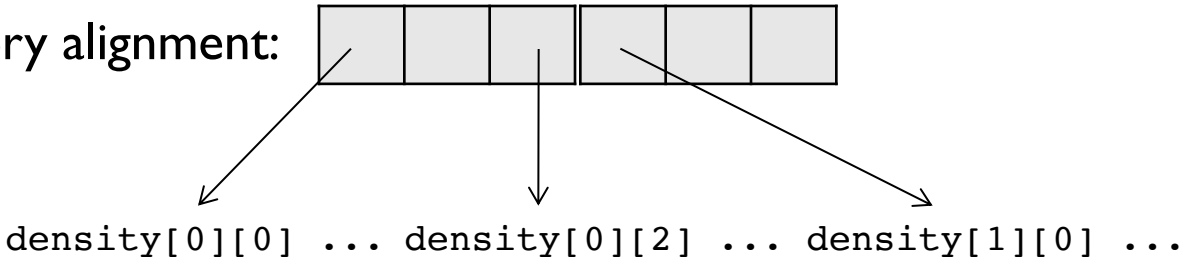


- Cache hits and misses:

```
for(j=0; j<2; j++)
   for(i=0; i<3; i++)
      whatever(density[j][i]);
```

GOOD!

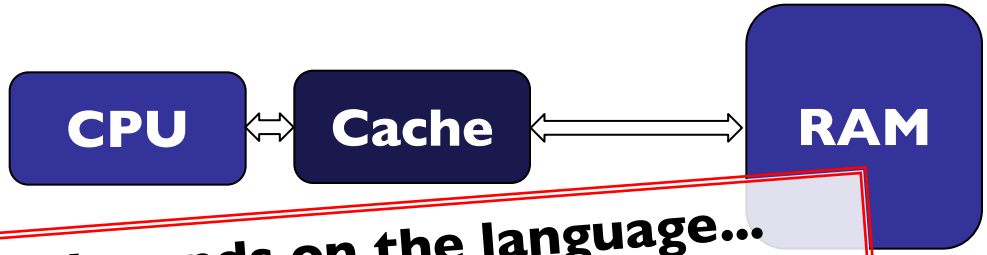
- 2D array in *C*: `density[2][3]`
- memory alignment:



Computer Architectures

- serial machine

=



memory alignment depends on the language...

check before programming!

(e.g. C uses row-major ordering while Fortran & MATLAB use column-major ordering)

- Cache hits and misses:

+ Python

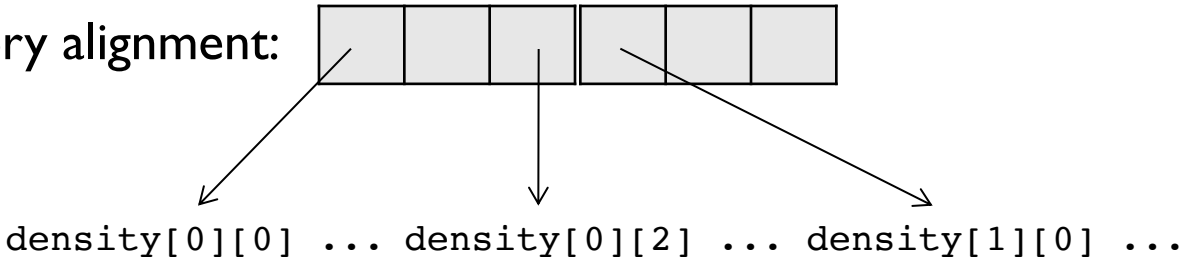
+ Julia

```
for(j=0; j<2; j++)
   for(i=0; i<3; i++)
      whatever(density[j][i]);
```

GOOD!

- 2D array in *C*: `density[2][3]`

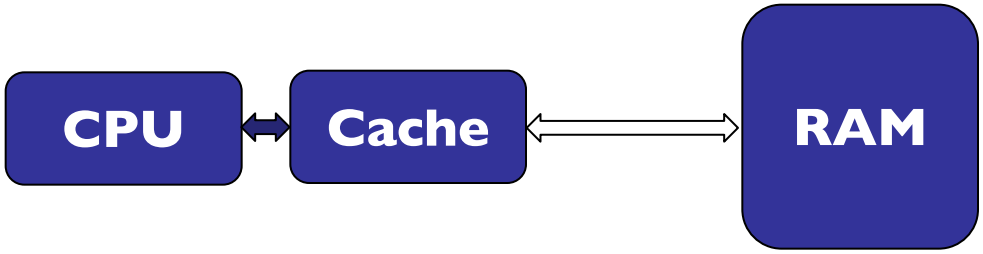
- memory alignment:



Computer Architectures

- serial machine

=



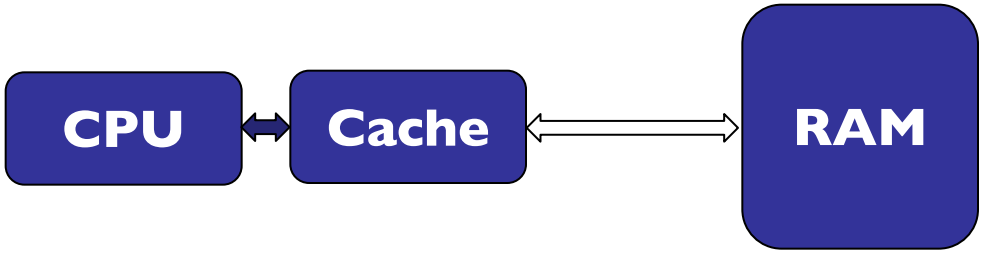
CPU clock frequency vs. bus frequency:

- CPU frequency determines execution speed of commands
- bus frequency determines how quickly to get new commands/data

Computer Architectures

- serial machine

=



CPU clock frequency vs. bus frequency:

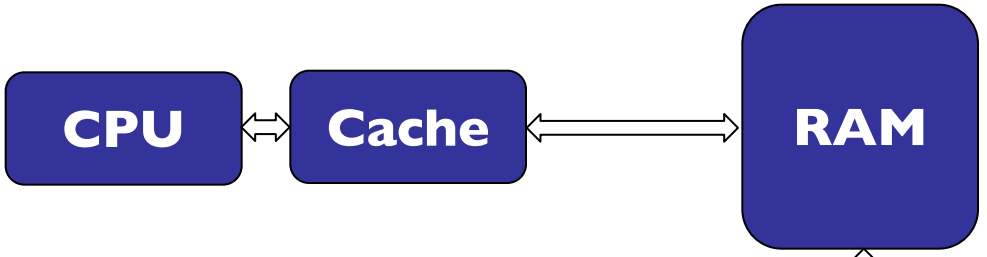
- CPU frequency determines execution speed of commands
- bus frequency determines how quickly to get new commands/data

=> bus frequency (and width) is more relevant for actual speed!

Computer Architectures

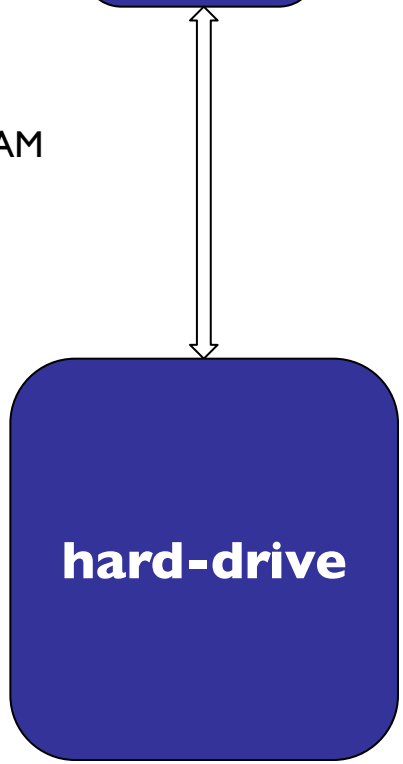
- serial machine

=



out-of-core applications:

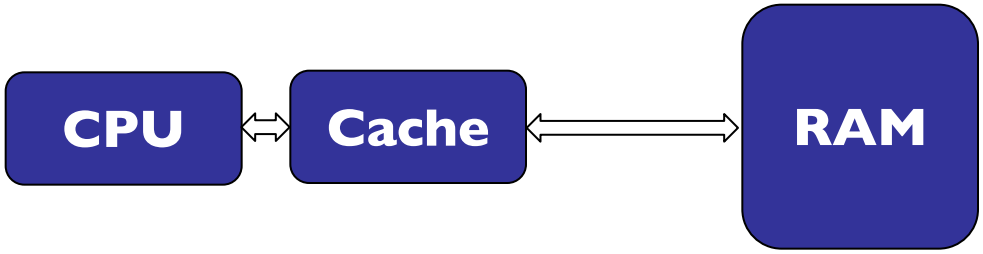
- not all data fits into RAM => use hard-drive like RAM
- practically all databases work like this...



Computer Architectures

- serial machine

=



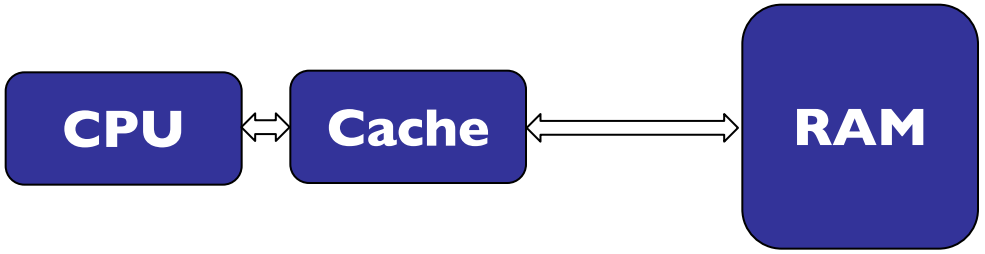
possible speed-ups by the programmer:

- improve your algorithm to require less instructions, e.g. `f=4*PI/Grav`
- improve your algorithm to use more adequate instructions, e.g. `x*x` instead of `pow(x,2)`
- proper usage of cache, e.g. check memory alignment

Computer Architectures

- serial machine

=



any other possibility to speed things up?

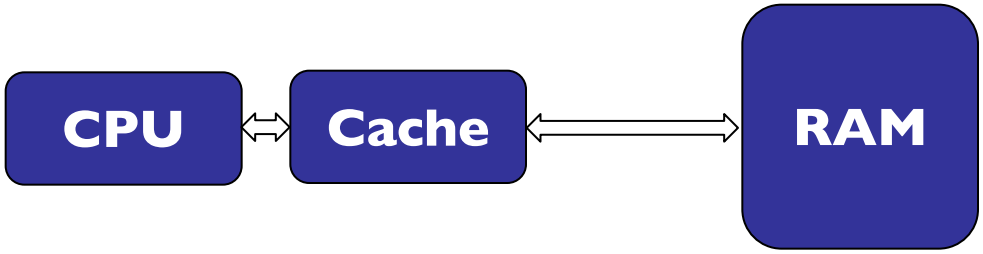
possible speed-ups by the programmer:

- improve your algorithm to require less instructions, e.g. `f=4*PI/Grav`
- improve your algorithm to use more adequate instructions, e.g. `x*x` instead of `pow(x,2)`
- proper usage of cache, e.g. check memory alignment

Computer Architectures

- serial machine

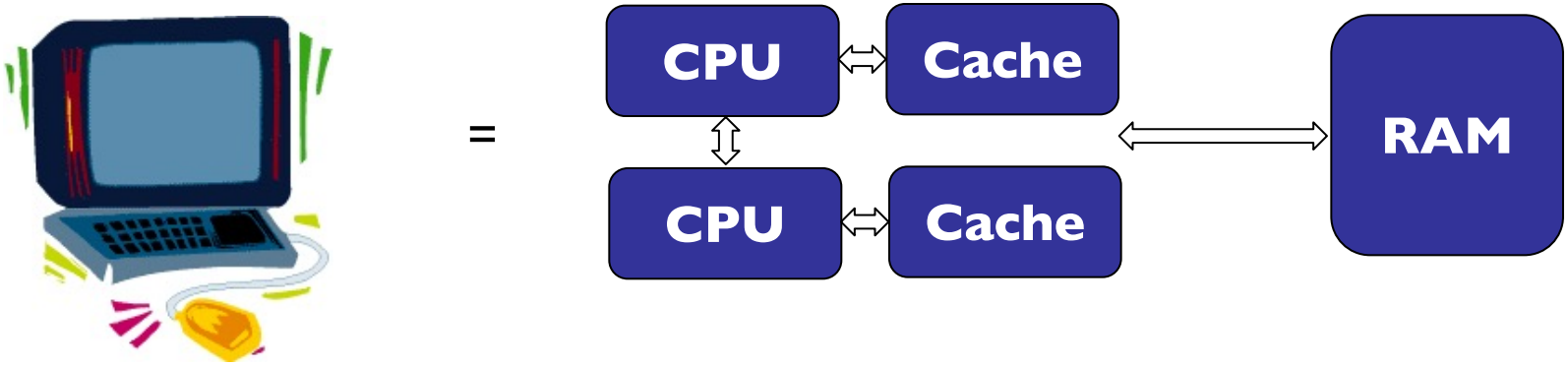
=



Accelerate your code by using GPUs, which are highly parallelised. Not explored here.

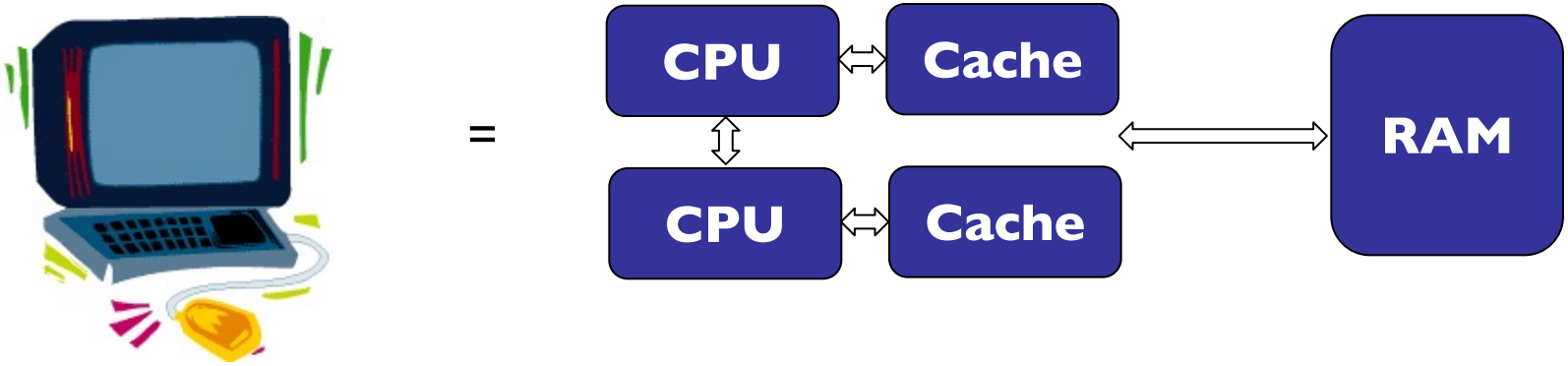
Computer Architectures

- serial machine – multi-cored



Computer Architectures

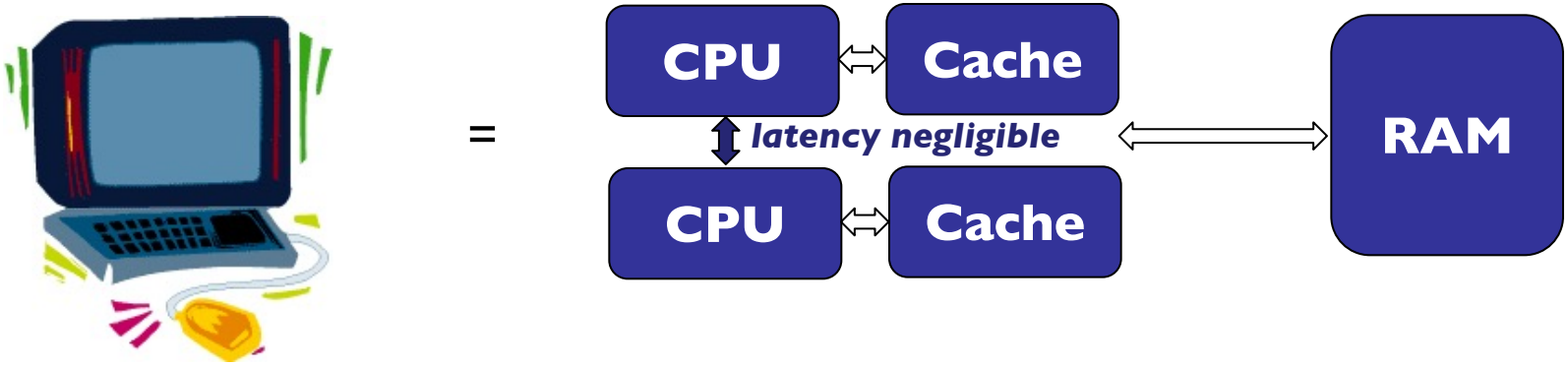
- serial machine – multi-cored



- shared memory architecture
  - easy to adapt existing serial code
  - limited by RAM to be placed into a single machine

Computer Architectures

- serial machine – multi-cored

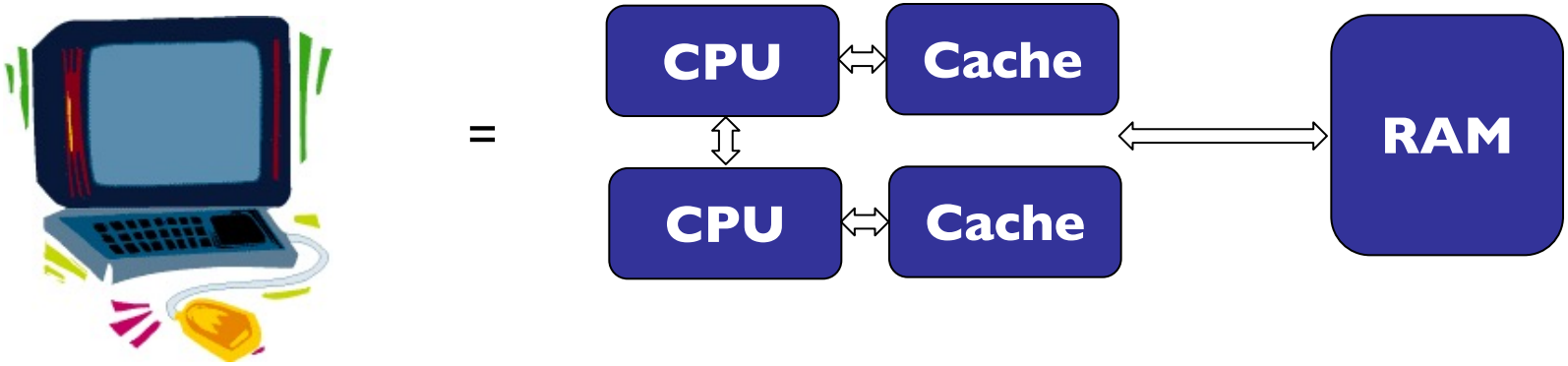


- shared memory architecture
  - easy to adapt existing serial code
  - limited by RAM to be placed into a single machine

How do we make use of the multicore capabilities?
Note that if you compile normally, you'll just use 1 CPU

Computer Architectures

- serial machine – multi-cored



- shared memory architecture
  - easy to adapt existing serial code
  - limited by RAM to be placed into a single machine

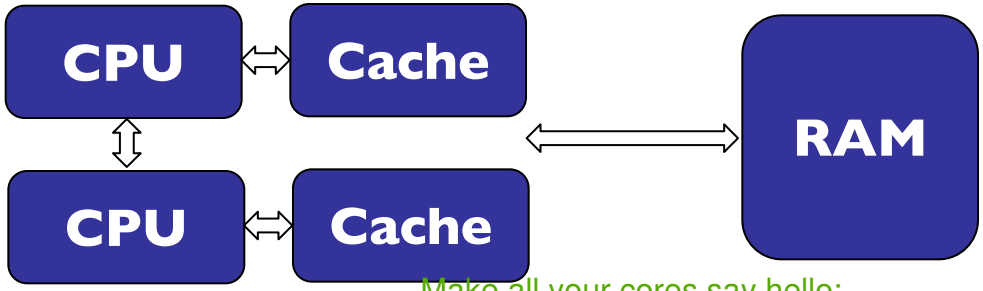
- OpenMP – `www.openmp.org`
  - most commonly used standard to parallelize code on shared memory architectures
  - primarily distribute `for`-loop components onto different CPU's
  - natively by supported by `gcc` since v4.2

Check the version of your compiler:
$ gcc --version

Computer Architectures

- serial machine – multi-cored

=



- shared memory architecture
  - easy to adapt existing serial code
  - limited by RAM to be placed into a single machine

- OpenMP – `www.openmp.org`
  - most commonly used standard to parallelize
  - [illegible] ctures
  - [illegible] supported by gcc since v4.2

Make all your cores say hello:

```
$ gcc -o p -fopenmp test_openmp.c
```

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

void main(){
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("hello(%d)", ID);
    printf("world(%d)\n", ID);
  }
}
```

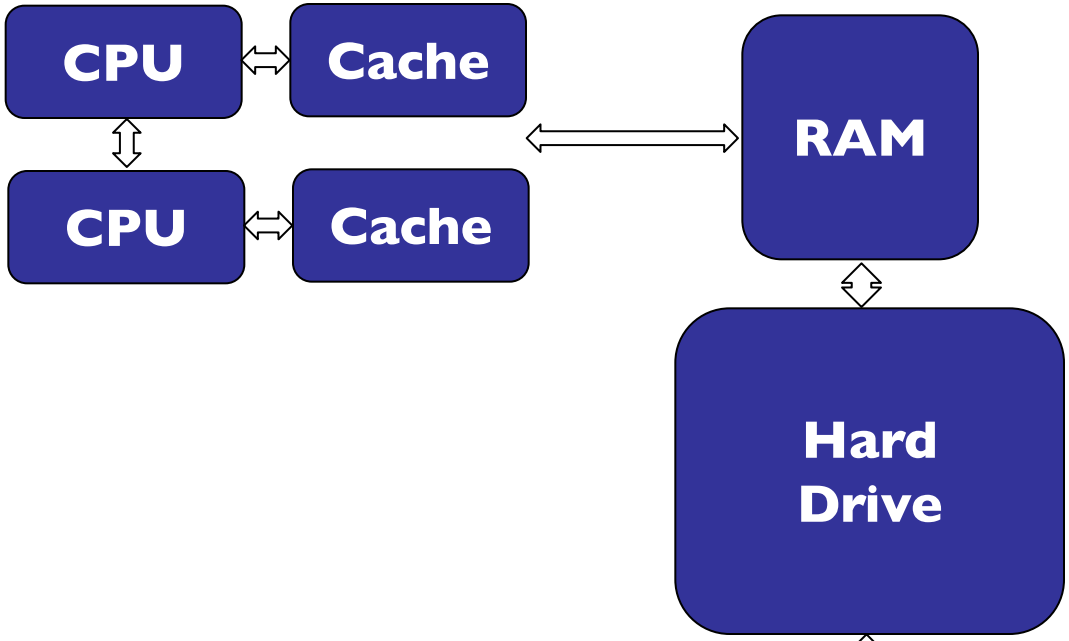
you(!) have to add extra commands to the code

Computer Architectures

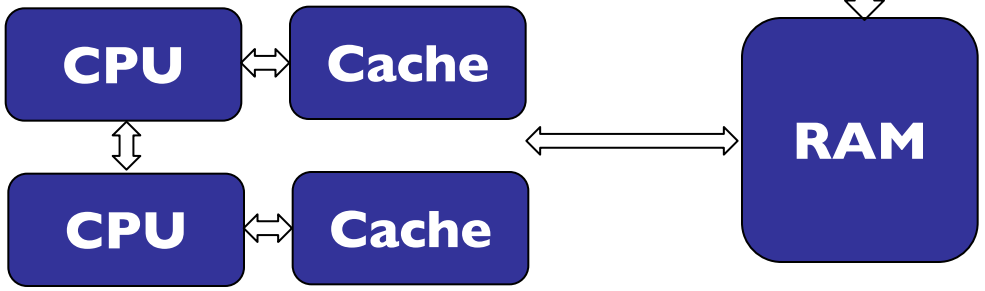
- parallel machine Distributed memory architecture

Check the number of sockets (shared memory) in your laptop with
$ sudo lscpu

=



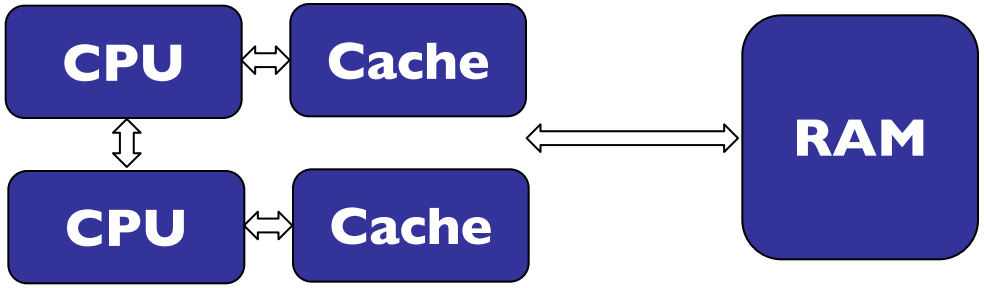
=

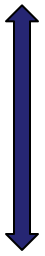


Computer Architectures

- parallel machine

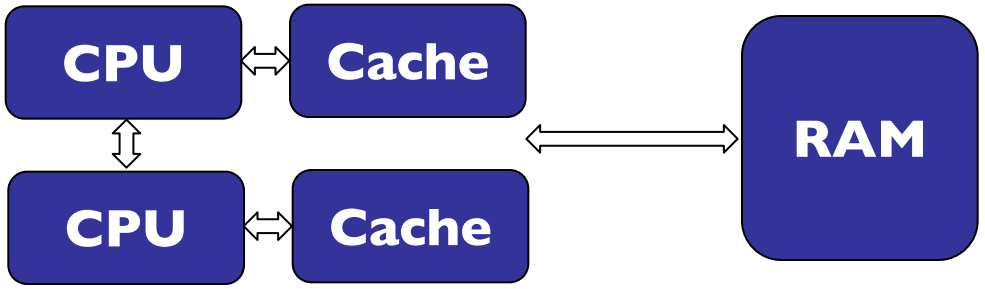
=





- performance highly sensitive to interconnect, e.g.
  - FireWire 50 MB/s
  - Gigabit 125 MB/s
  - Myrinet 250 MB/s
  - Infiniband 1000 MB/s
  - ...

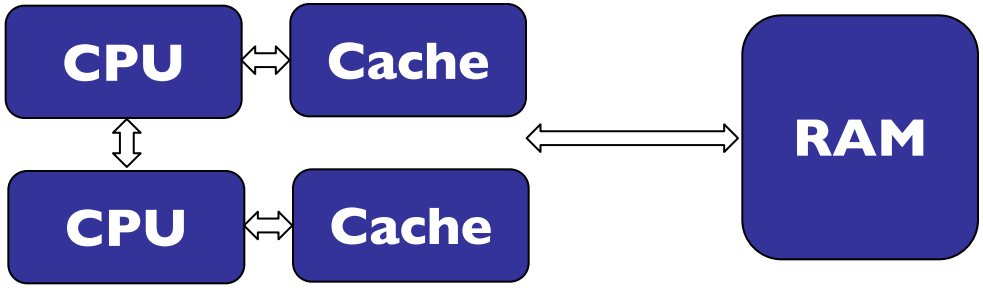
=



Computer Architectures

- parallel machine

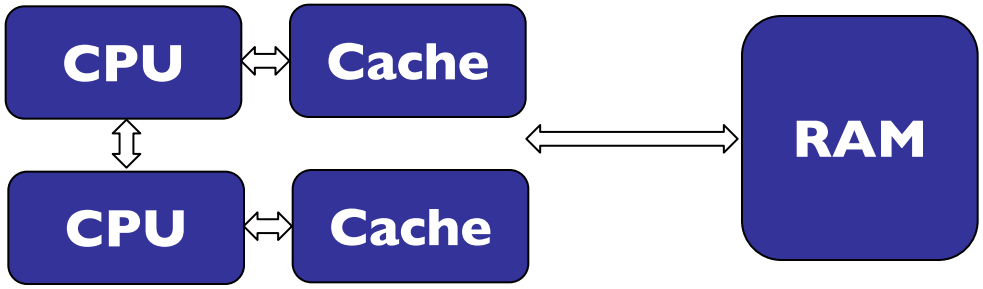
=



- distributed memory architecture:
  - existing code difficult to adapt
  - easy to built (cluster of PC's)
  - speed-up limited by inter-computer communication

⇩

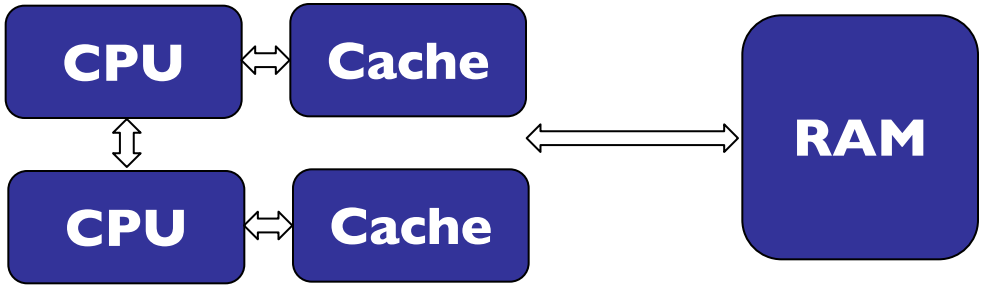
=



Computer Architectures

- parallel machine

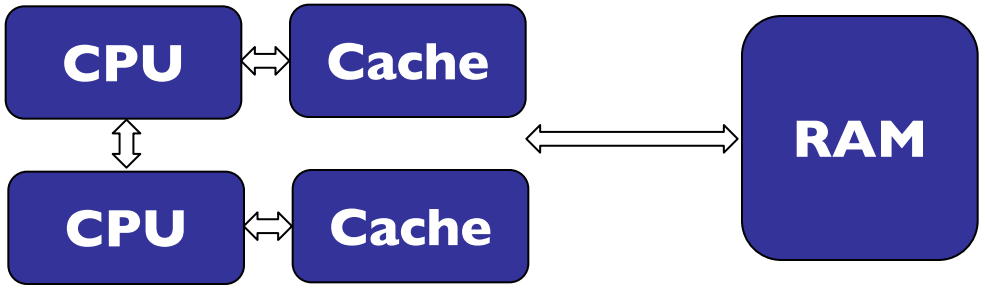
=



- distributed memory architecture:
  - existing code difficult to adapt
  - easy to built (cluster of PC's)
  - speed-up limited by inter-computer communicatior

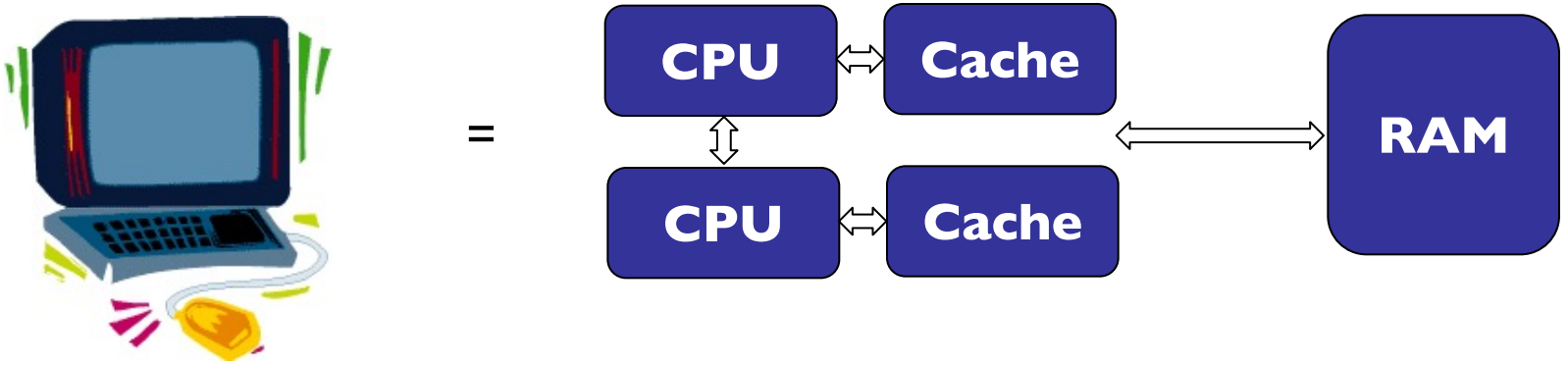
- MPI – Message Passing Interface
  - "standard" library for work dispersal on distributed memory architectures
  - e.g., `www.open-mpi.org`

=



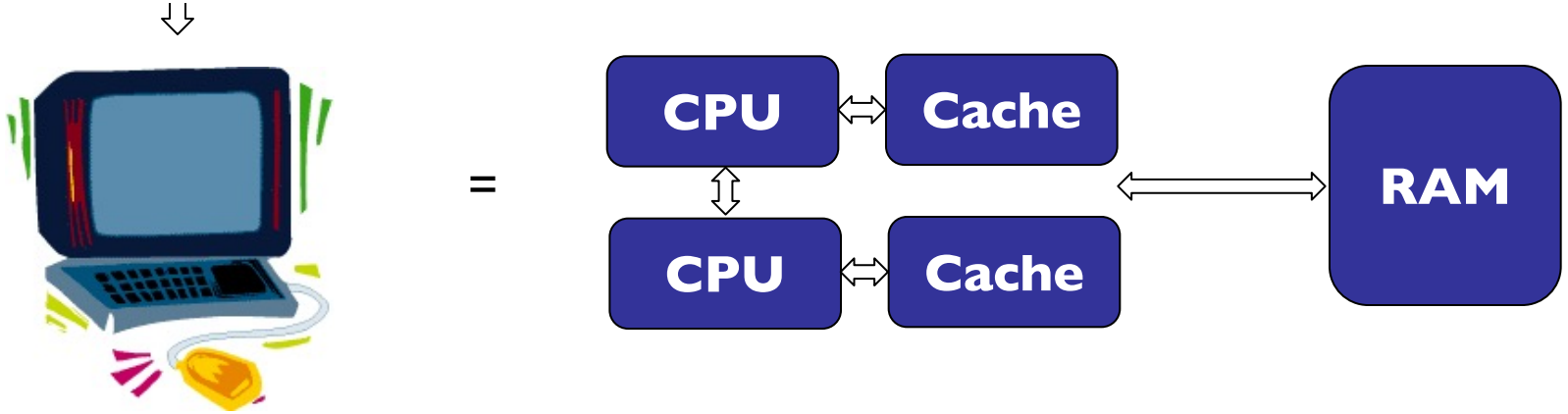
Computer Architectures

- parallel machine



- distributed memory architecture:
  - existing code difficult to
  - eas
  - speed-up limited by inter-computer communicatior
- MPI – Message Passing Interface
  - dispersal
  - on distributed memory architectures
  - e.g., `www.open-mpi.org`

you(!) have to substantially restructure your code

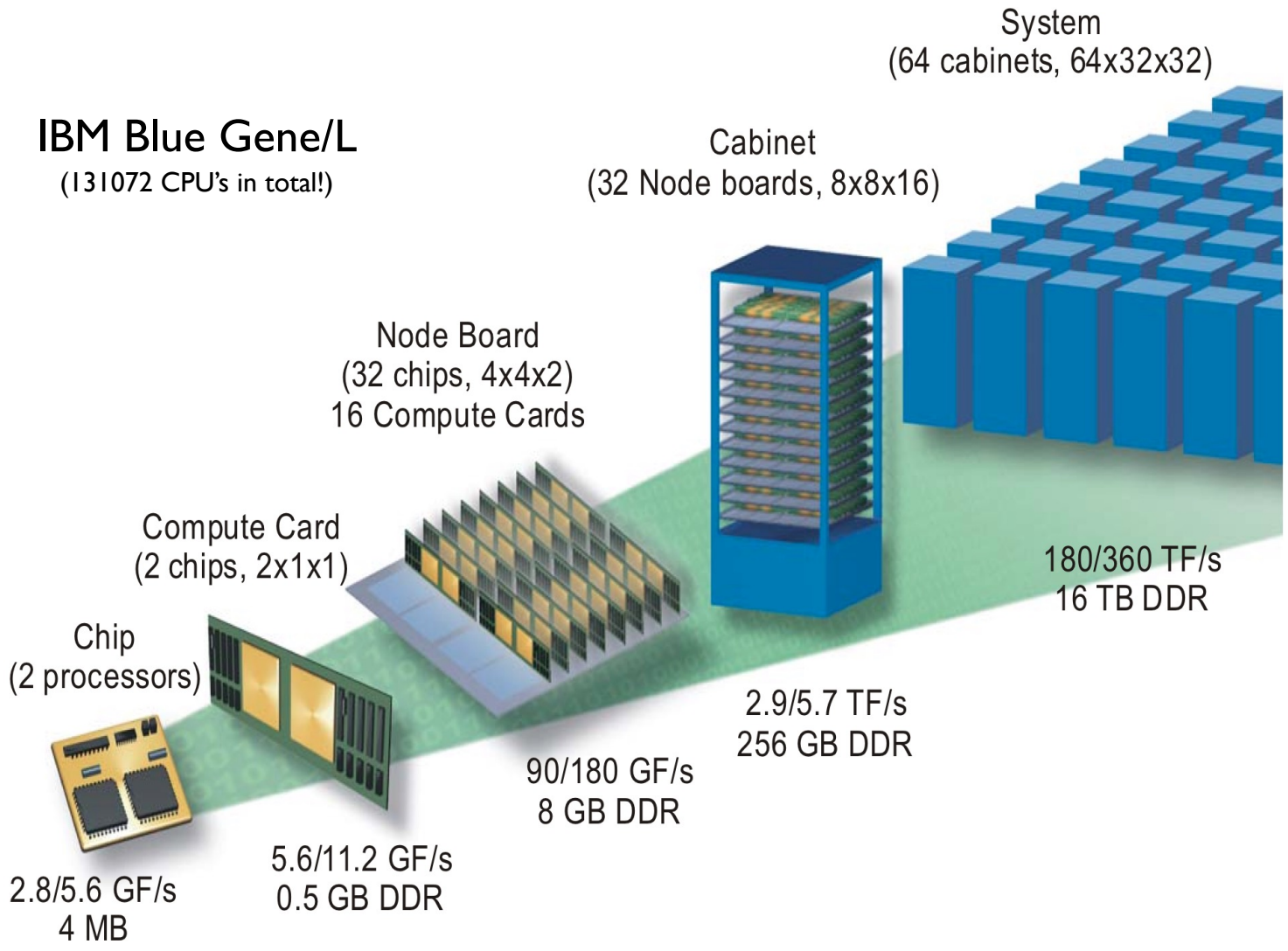


- architectures
- **real machines**
- computing concepts
- parallel programming

- parallel machine in reality = multi-level hybrid machines

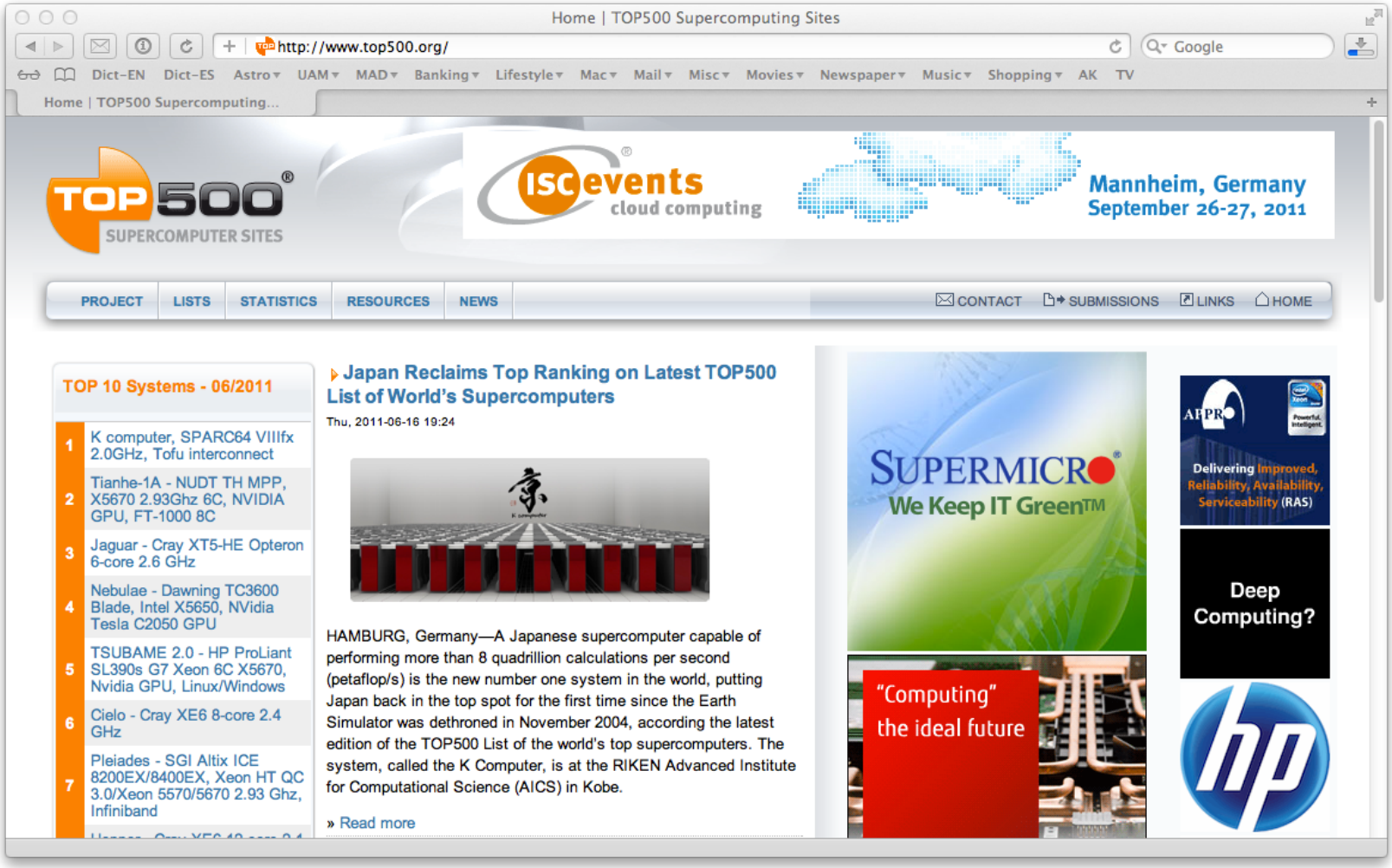
IBM Blue Gene/L
(131072 CPU's in total!)

- parallel machine in reality = multi-level hybrid machines



Computer Architectures

- www.top500.org



Computer Architectures

- www.top500.org

#136 in 06/2011

Name:	Magerit
Vendor:	IBM
#CPU's:	3920
performance:	100 Tflops/sec

#170 in 06/2011

Name:	MareNostrum
Vendor:	IBM
#CPU's:	10240
performance:	60 Tflops/sec

Computer Architectures

- www.top500.org

#136 in 06/2011

Name:	Magerit
Vendor:	IBM
#CPU's:	3920
performance:	100 Tflops/sec

#170 in 06/2011

Name:	MareNostrum
Vendor:	IBM
#CPU's:	10240
performance:	60 Tflops/sec

- www.top500.org

#136 in 06/2011

Name:	Magerit
Vendor:	IBM
#CPU's:	3920
performance:	100 Tflops/sec

interconnect: Infiniband, up to 1500 Gbits/sec

#170 in 06/2011

Name:	MareNostrum
Vendor:	IBM
#CPU's:	10240
performance:	60 Tflops/sec

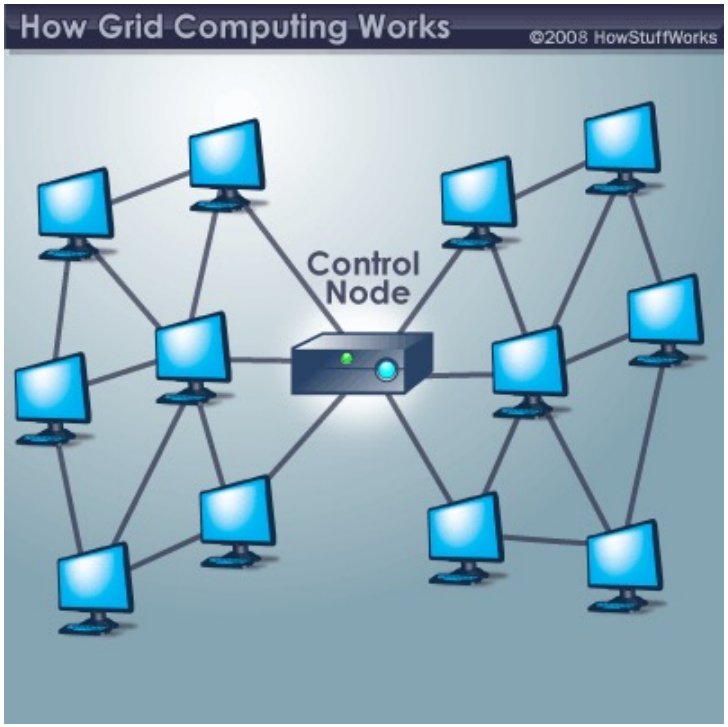
interconnect: Myrinet, ca. 2Gbit/sec

- architectures
- real machines
- **computing concepts**
- parallel programming

- GRID computing / Cloud computing?

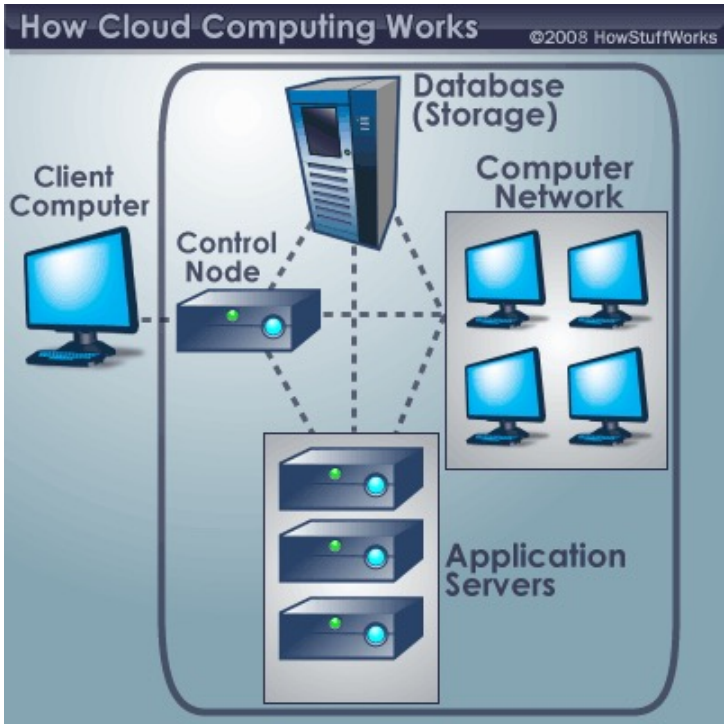
Computer Architectures

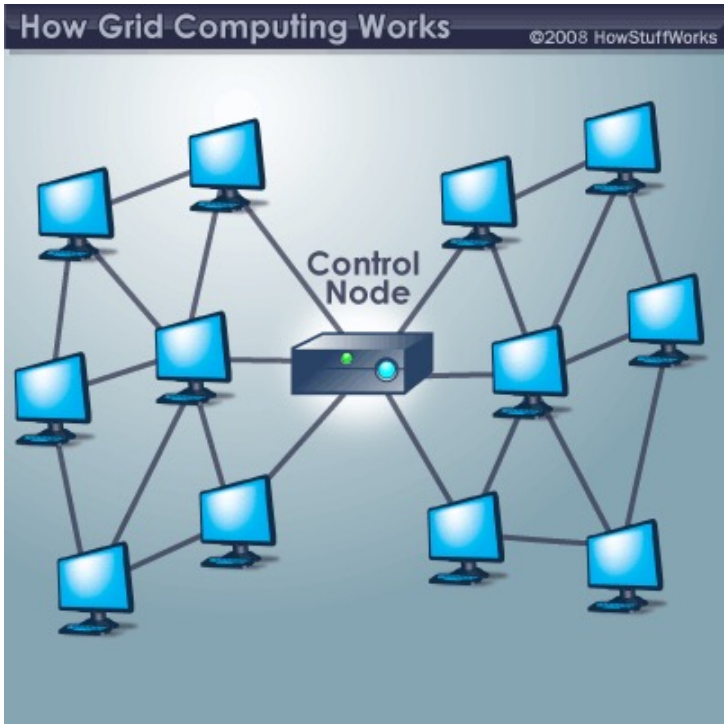
- GRID computing:
  - distributed computing where resources are linked together to solve a single problem



Computer Architectures

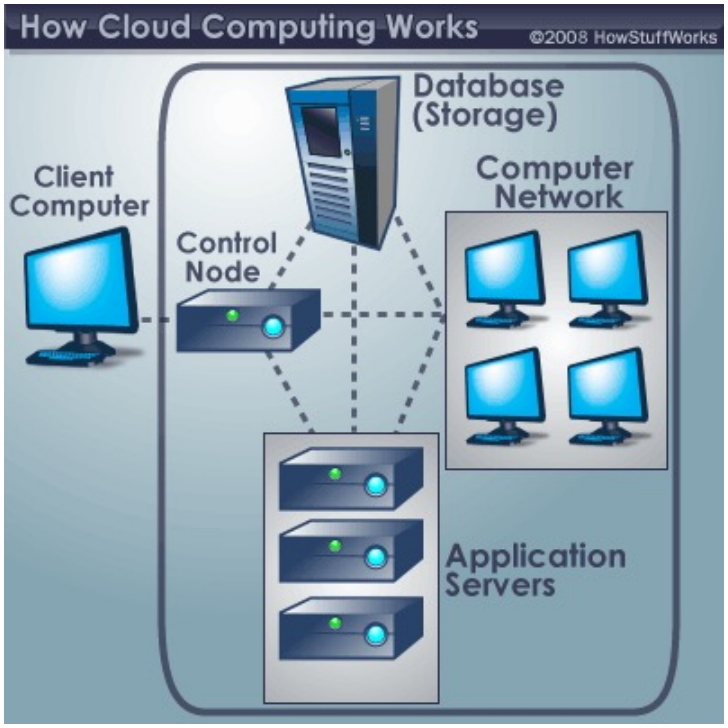
- GRID computing:
  - distributed computing where resources are linked together to solve a single problem
- Cloud computing:
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)

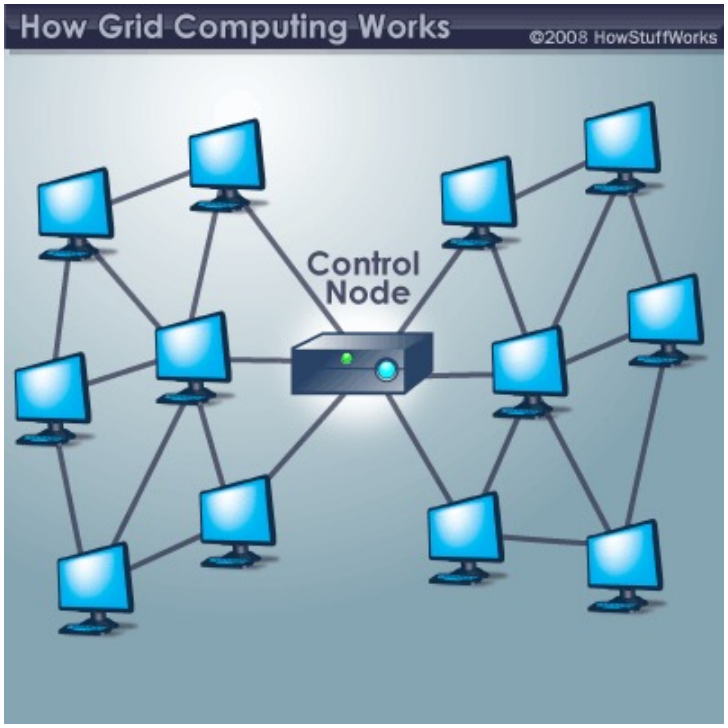




Computer Architectures

- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem
- Cloud computing: ⇨ **access to results via databases**
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)





Computer Architectures

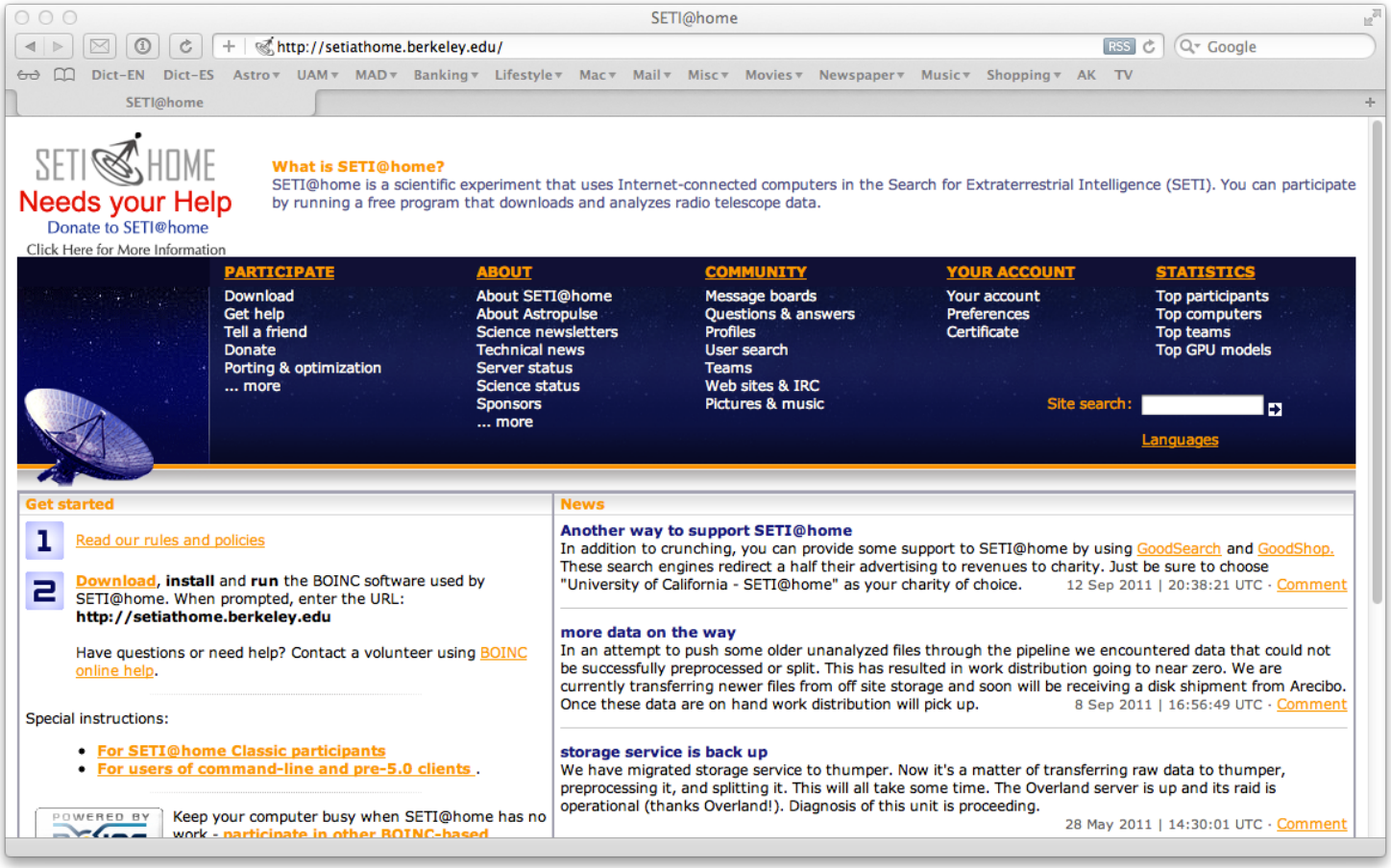
- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem
- Cloud computing: ⇨ **access to results via databases**
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)

first GRID computing?

Computer Architectures

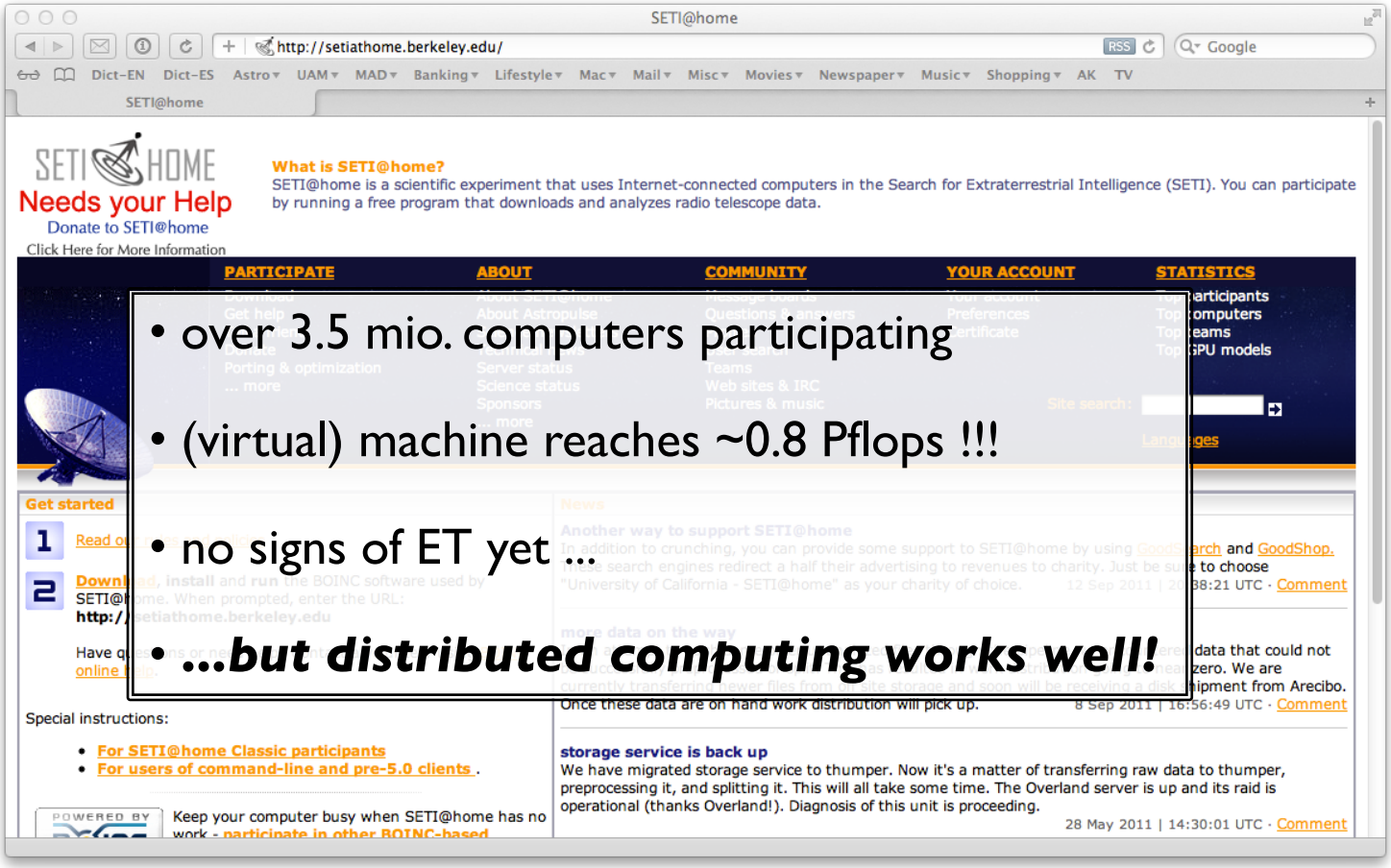
- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem

- Cloud computing: ⇨ **access to results via databases**
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)



Computer Architectures

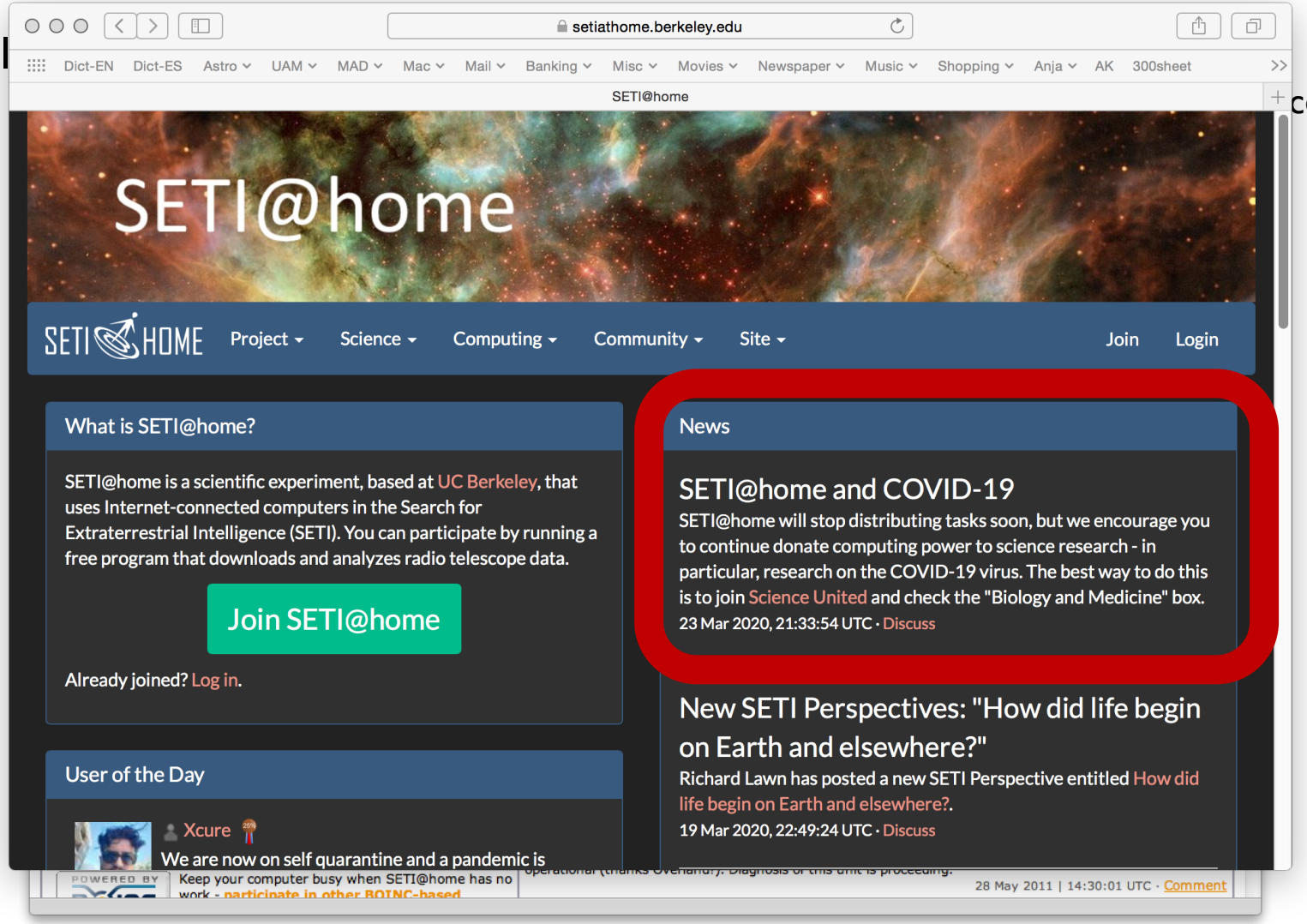
- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem
- Cloud computing: ⇨ **access to results via databases**
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)



- over 3.5 mio. computers participating
- (virtual) machine reaches ~0.8 Pflops !!!
- no signs of ET yet ...
- ***...but distributed computing works well!***

Computer Architectures

- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem
- Cl



ce, ...)

Computer Architectures

- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem
- Cloud computing: ⇨ **access to results via databases**
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)

there actually exists a GRID network with 21×10^6 Pflops* !!!!!!

*as of Oct 2016

Computer Architectures

- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem
- Cloud computing: ⇨ **access to results via databases**
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)

there actually exists a GRID network with 21×10^6 Pflops:

Bitcoin Network

Computer Architectures

- GRID computing: ⇨ **actually running simulations**
  - distributed computing where resources are linked together to solve a single problem

- Cloud computing: ⇨ **access to results via databases**
  - use remote resources for your (personal) needs (music storage, email correspondence, ...)

- architectures
- real machines
- computing concepts
- **parallel programming**

Computer Architectures

- how to program such machines?

- how to program such machines?

your algorithm must be parallel,

then it's only a matter of using parallel libraries to distribute the work...

- how to program such machines?

your algorithm must be parallel,

then it's only a matter of using parallel libraries to distribute the **work**...

data parallelisation:
all CPUs execute the same code, but have different parts of the data

task parallelisation
all CPUs have the same data, but execute different calculations

- how to program such machines?

your algorithm must be parallel,

then it's only a matter of using parallel libraries to distribute the **work**...

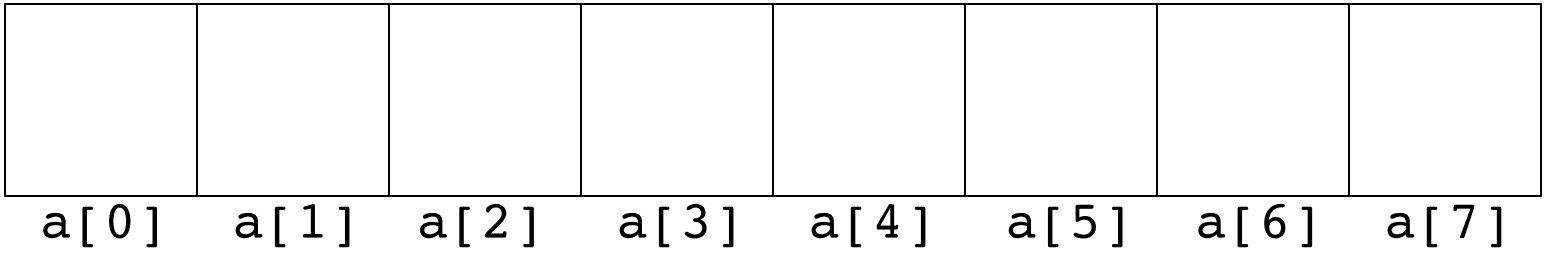
data parallelisation:

all CPUs execute the same code, but have different parts of the data

Computer Architectures

- how to program such machines?

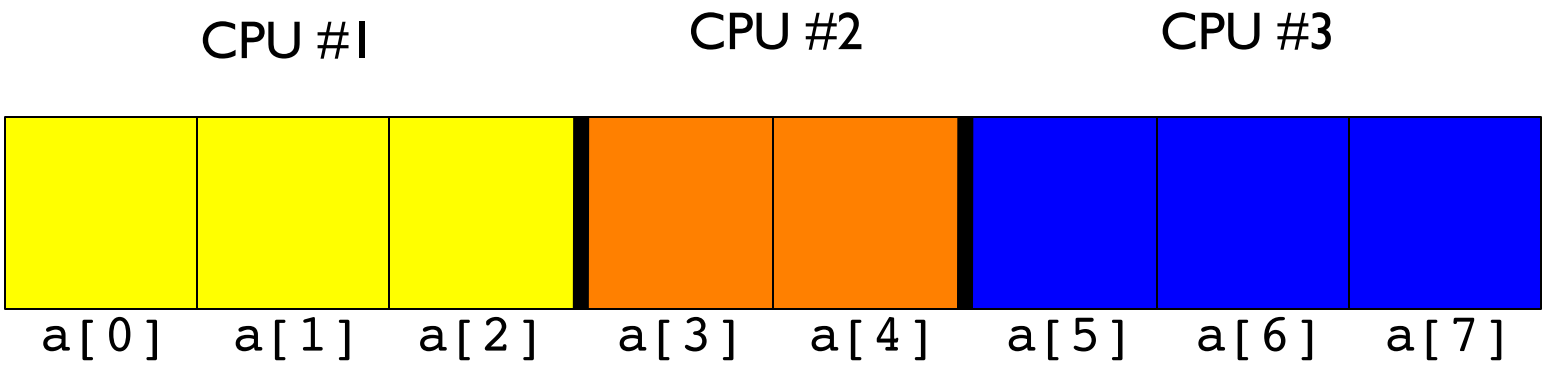
1D array



Computer Architectures

- how to program such machines?

1D array



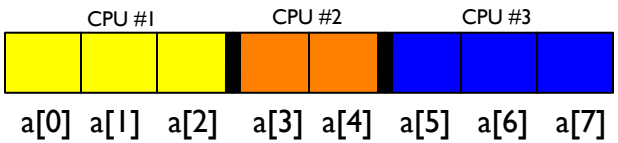
- how to program such machines?

1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   a[i] = function(a[i-1]);

}
```

Make a (serial) program that doubles the numbers.



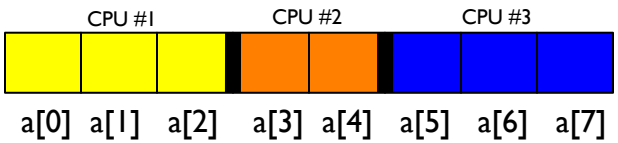
- how to program such machines?

1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   a[i] = function(a[i-1]);

}
```

⇨ *not parallelizable as* `a[i]` *depends on all previous* `a[]`*'s!*



- how to program such machines?

1D array

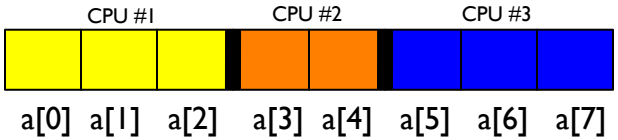
serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   a[i] = function(a[i-1]);

}
```

⇨ *not parallelizable as* `a[i]` *depends on all previous* `a[]`*'s!*

general remark:
recursion is elegant yet not parallelizable...

How can we make the tasks independent?



Computer Architectures

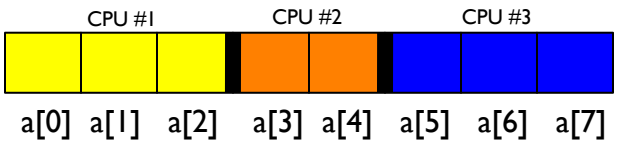
- how to program such machines?

1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   a[i] = function(a[i-1]);

}
```

parallel algorithm



Computer Architectures

- how to program such machines?

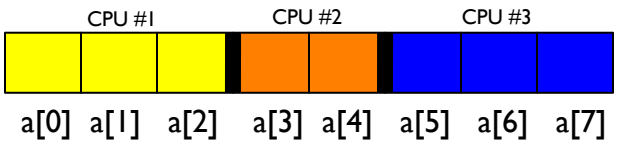
1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   a[i] = function(a[i-1]);

}
```

parallel algorithm

each CPU runs the same code,
but on a different part of the problem...



Computer Architectures

- how to program such machines?

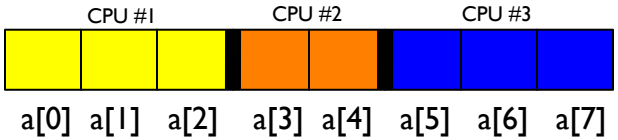
1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   a[i] = function(a[i-1]);

}
```

parallel algorithm

example:
shared memory architecture
(i.e. all CPU's can access the same memory)



Computer Architectures

- how to program such machines?

1D array

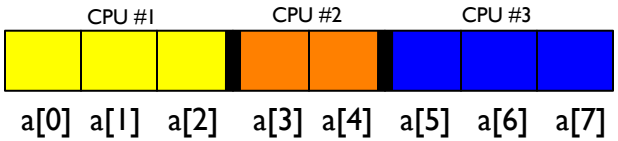
serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {



   a[i] = function(a[i-1]);
}
```

parallel algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```

Code this up doubling values (still in series).



- how to program such machines?

1D array

serial algorithm

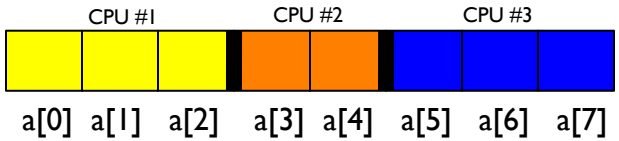
```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {




   a[i] = function(a[i-1]);
}
```

parallel algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```

the `i`-loop can now be parallelized as all `a[i]` are calculated independently

Filling the array are independent tasks, even if this program is slower.



Computer Architectures

- how to program such machines?

1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {


   a[i] = function(a[i-1]);
}
```

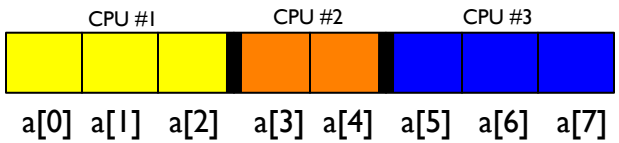
parallel algorithm

Make yor last program parallel:

```
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   # pragma omp for
   a[0] = STARTVALUE;
   for(i=1; i<N; i++) {

      b = a[0];
      for(j=0; j<i; j++) {
       b = function(b);
      }
      a[i] = b;
      printf("Thread%d: a[%d]=%.1f \n",ID,i,a[i]);
   }

}
```

we eliminated the recursion/dependence by expanding it explicitly.

(by introducing yet another recursion, but c'est la vie...)



Compilation:
gcc -o p -fopenmp par_b.c

What time do you get using clock()?

- how to program such machines?

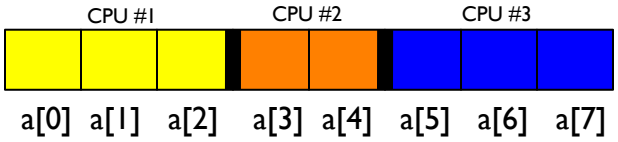
1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {



   a[i] = function(a[i-1]);
}
```

parallel algorithm

```
double start = omp_get_wtime();
  a[0] = STARTVALUE;
#pragma omp parallel for private(i,j,b) shared(a)
  for(i=1; i<N; i++) {

     b = a[0];
     for(j=0; j<i; j++) {
      b = function(b);
     }
     a[i] = b;
  }

     double time = 1000.*(omp_get_wtime() - start); //ms
     printf("\n Time=%E ms \n",time);
```



- how to program such machines?

1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {



   a[i] = function(a[i-1]);
}
```

parallel algorithm

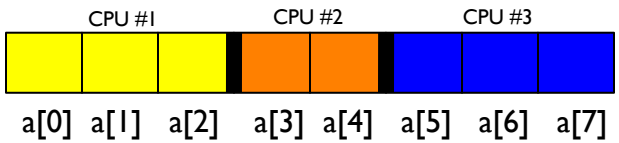
```
  a[0] = STARTVALUE;
#pragma omp parallel for private(i,j,b) shared(a)
  for(i=1; i<N; i++) {

     b = a[0];
     for(j=0; j<i; j++) {
      b = function(b);
     }
     a[i] = b;
  }
```

each CPU gets its own private copy of these variables

Observe what the program has assumed by default and what happens if you modify here varibles from private to shared.

What do you expect for N?

What happens if you set b as shared?



- how to program such machines?

1D array

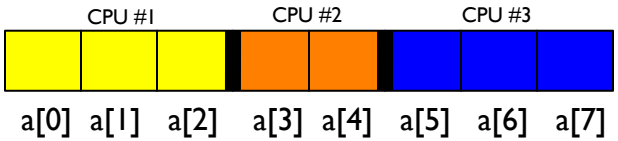
serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {



   a[i] = function(a[i-1]);
}
```

parallel algorithm

```
  a[0] = STARTVALUE;
#pragma omp parallel for private(i,j,b) shared(a)
  for(i=1; i<N; i++) {

     b = a[0];
     for(j=0; j<i; j++) {
      b = function(b);
     }
     a[i] = b;
  }
```

these variables remain where they are in RAM and can be accessed by each CPU



- how to program such machines?

1D array

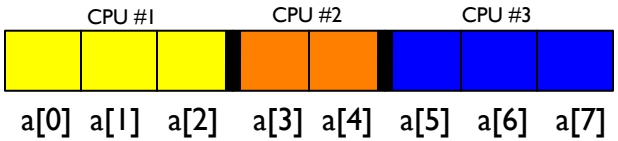
serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {



   a[i] = function(a[i-1]);
}
```

parallel algorithm

```
  a[0] = STARTVALUE;
#pragma omp parallel for private(i,j,b) shared(a)
  for(i=1; i<N; i++) {

     b = a[0];
     for(j=0; j<i; j++) {
      b = function(b);
     }
     a[i] = b;
  }
```

N: shared or private?



Computer Architectures

- how to program such machines?

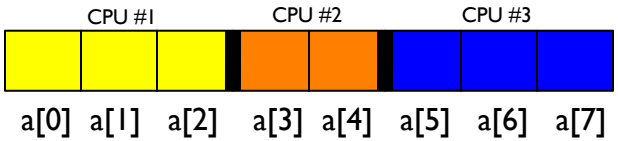
1D array

serial algorithm

```
a[0] = STARTVALUE;

for(i=1; i<N; i++) {



   a[i] = function(a[i-1]);
}
```

parallel algorithm

```
   a[0] = STARTVALUE;
#pragma omp parallel for private(i,j,b) shared(a,N)
   for(i=1; i<N; i++) {

      b = a[0];
      for(j=0; j<i; j++) {
       b = function(b);
      }
      a[i] = b;
   }
```

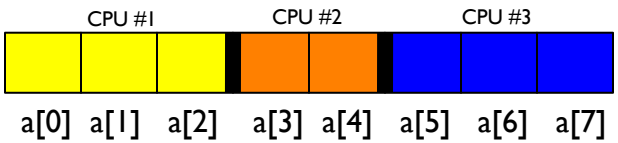


Computer Architectures

- how to program such machines?

1D array

```
STARTVALUE = 1.35;
N          = 8;
a          = (float *) calloc(N, sizeof(float));
a[0]       = STARTVALUE;
```



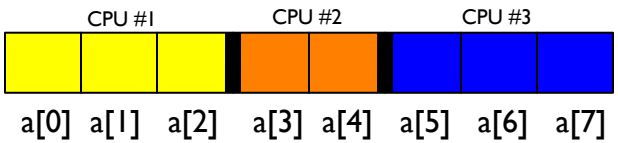
Computer Architectures

- how to program such machines?

1D array

```
STARTVALUE = 1.35;
N          = 8;
a          = (float *) calloc(N, sizeof(float));
a[0]       = STARTVALUE;
```

N=8, a[0]=1.35, a[1:7]=0



Computer Architectures

- how to program such machines?

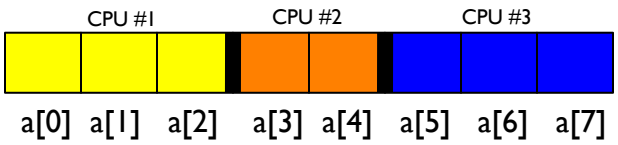
1D array

```
STARTVALUE = 1.35;
N          = 8;
a          = (float *) calloc(N, sizeof(float));
a[0]       = STARTVALUE;
```

N=8, a[0]=1.35, a[1:7]=0

```
#pragma omp parallel for private(i,j,b) shared(a,N)
```

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;

   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```



Computer Architectures

- how to program such machines?

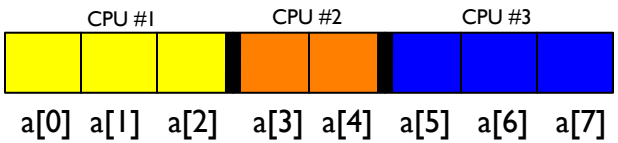
1D array

```
STARTVALUE = 1.35;
N          = 8;
a          = (float *) calloc(N, sizeof(float));
a[0]       = STARTVALUE;
```

N=8, a[0]=1.35, a[1:7]=0

```
#pragma omp parallel for private(i,j,b) shared(a,N)
```

i=1,2	i=3,4	i=5,6,7
j,b only known to CPU #1	j,b only known to CPU #2	j,b only known to CPU #3
a[0:7] fully accessible	a[0:7] fully accessible	a[0:0] fully accessible
N fully accessible	N fully accessible	N fully accessible



Computer Architectures

- how to program such machines?

1D array

```
STARTVALUE = 1.35;
N          = 8;
a          = (float *) calloc(N, sizeof(float));
a[0]       = STARTVALUE;
```

N=8, a[0]=1.35, a[1:7]=0

#pragma omp parallel for private(i,j,b) shared(a,N)

i=1,2
j,b only known to CPU #1
a[0:7] fully accessible
N fully accessible

```
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```

no communication

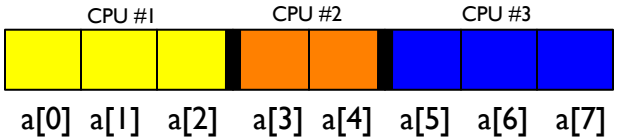
i=3,4
j,b only known to CPU #2
a[0:7] fully accessible
N fully accessible

```
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```

no communication

i=5,6,7
j,b only known to CPU #3
a[0:0] fully accessible
N fully accessible

```
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```



Computer Architectures

- how to program such machines?

1D array

```
STARTVALUE = 1.35;
N          = 8;
a          = (float *) calloc(N, sizeof(float));
a[0]       = STARTVALUE;
```

N=8, a[0]=1.35, a[1:7]=0

```
#pragma omp parallel for private(i,j,b) shared(a,N)
```

i=1,2
j,b only known to CPU #1
a[0:7] fully accessible
N fully accessible

```
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```

no communication

i=3,4
j,b only known to CPU #2
a[0:7] fully accessible
N fully accessible

```
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```

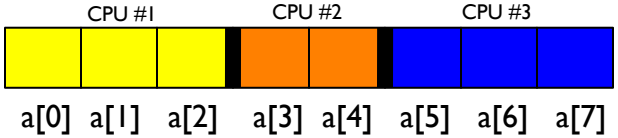
no communication

i=5,6,7
j,b only known to CPU #3
a[0:0] fully accessible
N fully accessible

```
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }
   a[i] = b;
}
```

N=8
a[0:7] filled with desired values
i, j, b undefined values

...



Computer Architectures

- how to program such machines (OpenMP standard):

```
#pragma omp parallel for private() shared()
```

- how to program such machines (OpenMP standard):

```
#pragma omp parallel for private() shared()
```

start parallel environment
(can be started everywhere in code...)

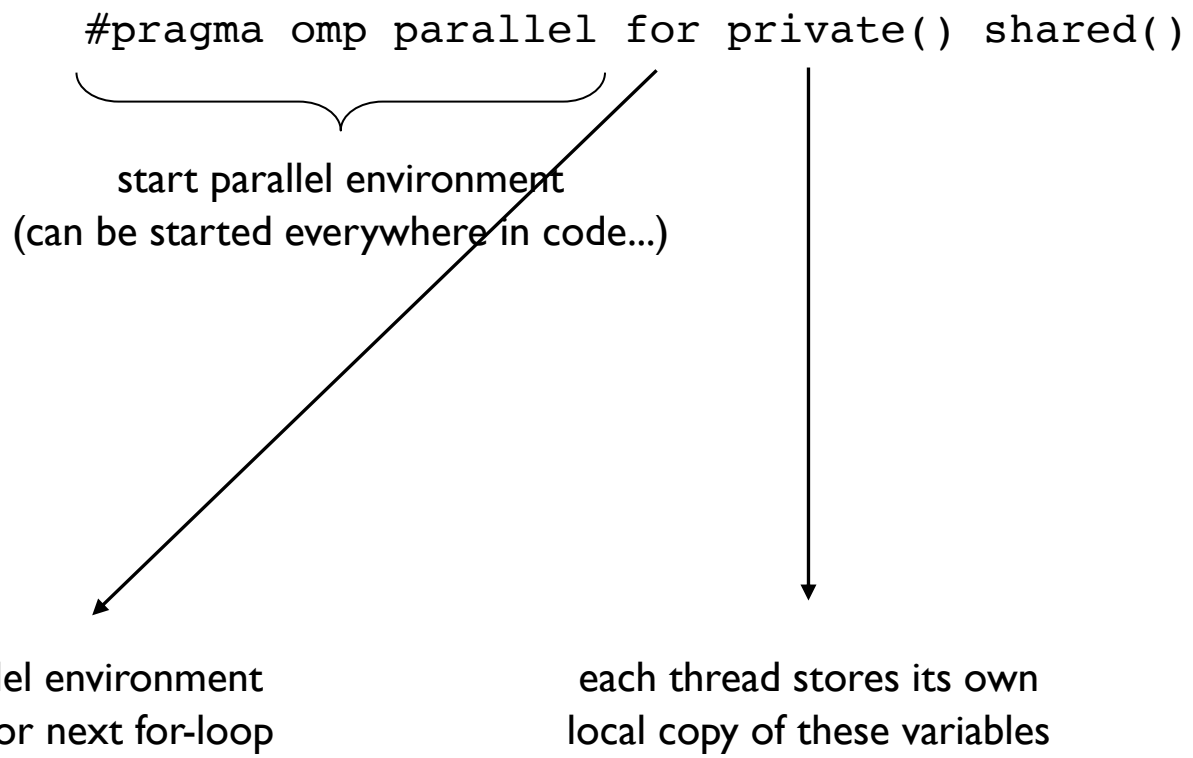
- how to program such machines (OpenMP standard):

```
#pragma omp parallel for private() shared()
```

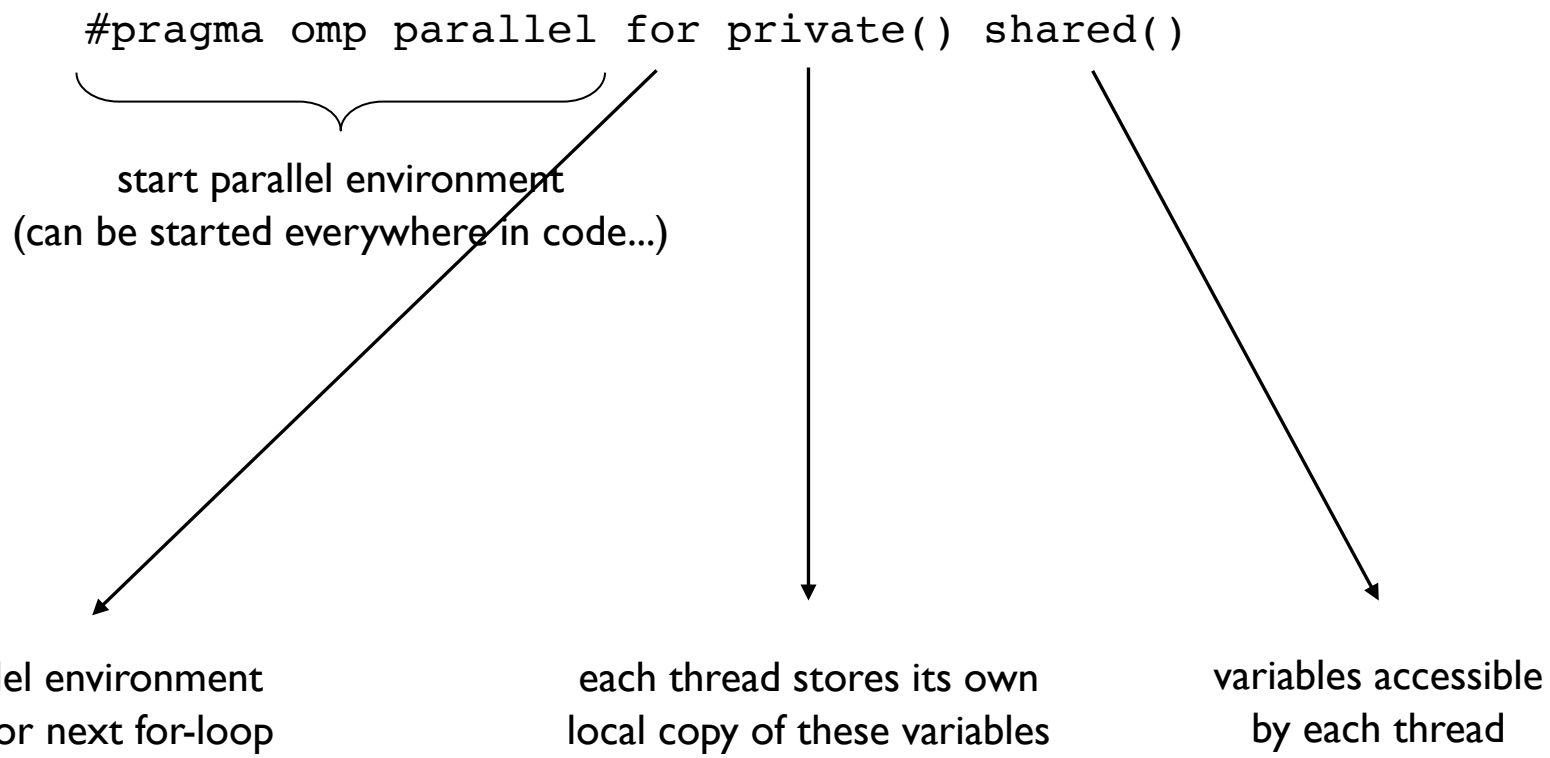
start parallel environment
(can be started everywhere in code...)

parallel environment
only for next for-loop

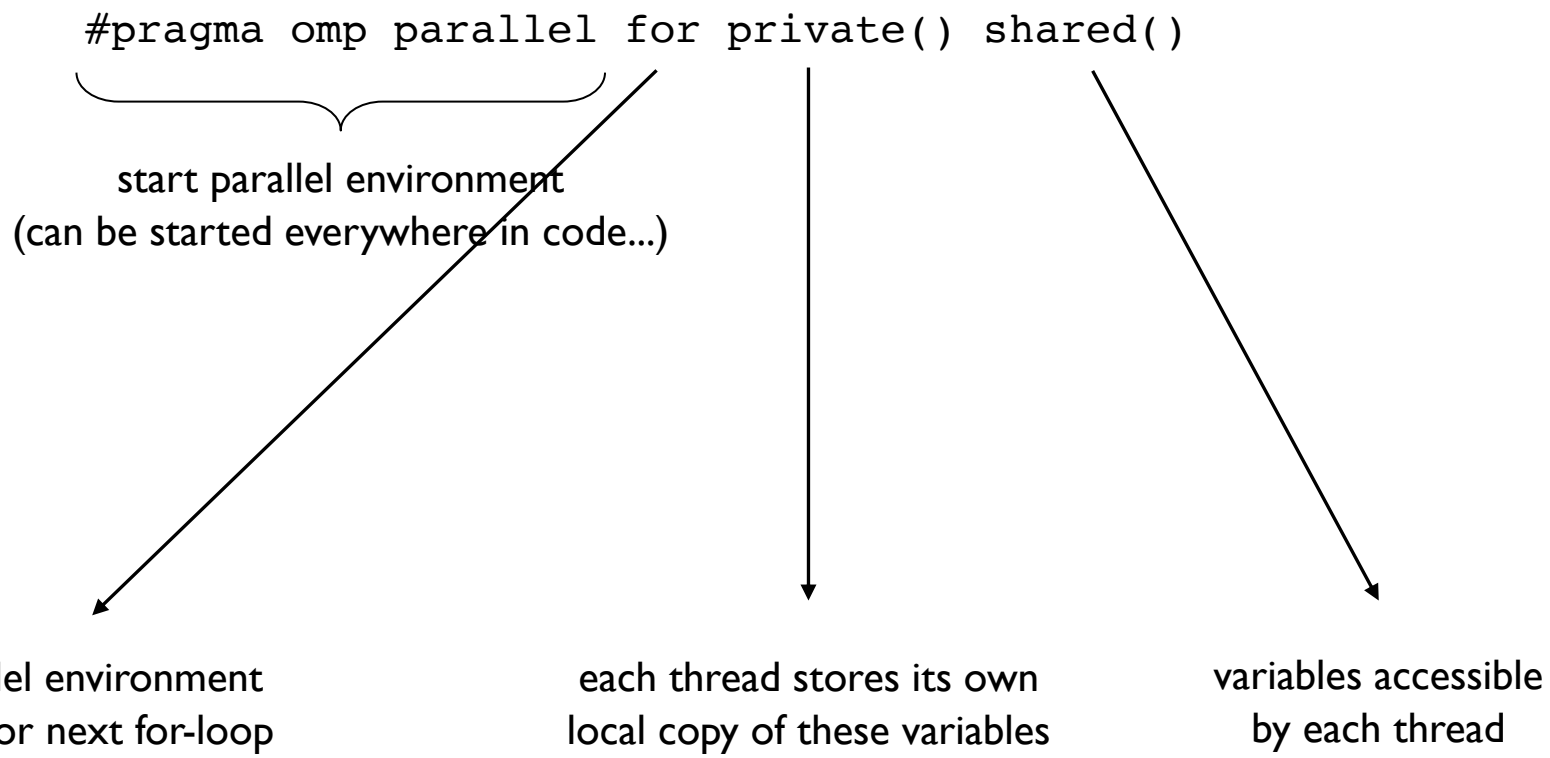
- how to program such machines (OpenMP standard):



- how to program such machines (OpenMP standard):



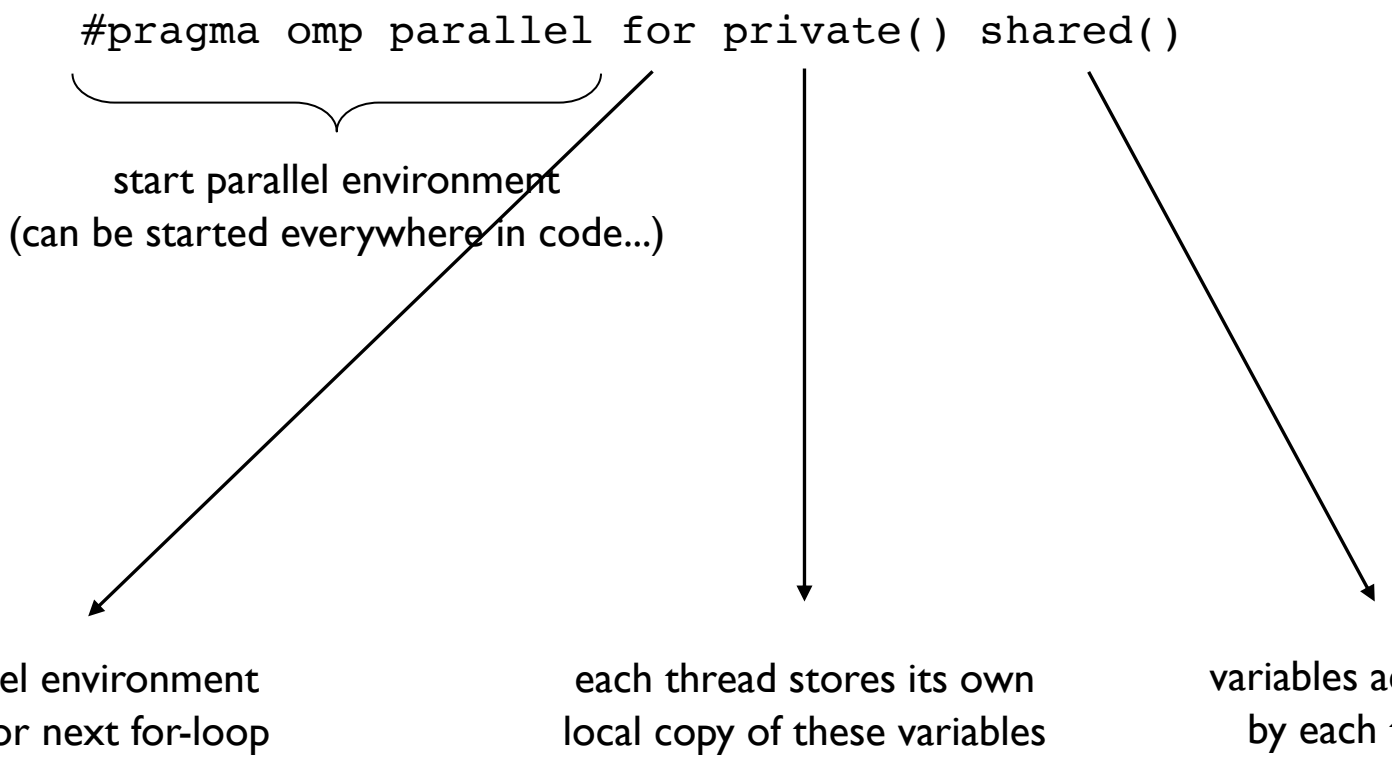
- how to program such machines (OpenMP standard):



Note:

- if you only read the value of a variable, it can be 'shared'
- if you write into a variable, think carefully about its status!

- how to program such machines (OpenMP standard):



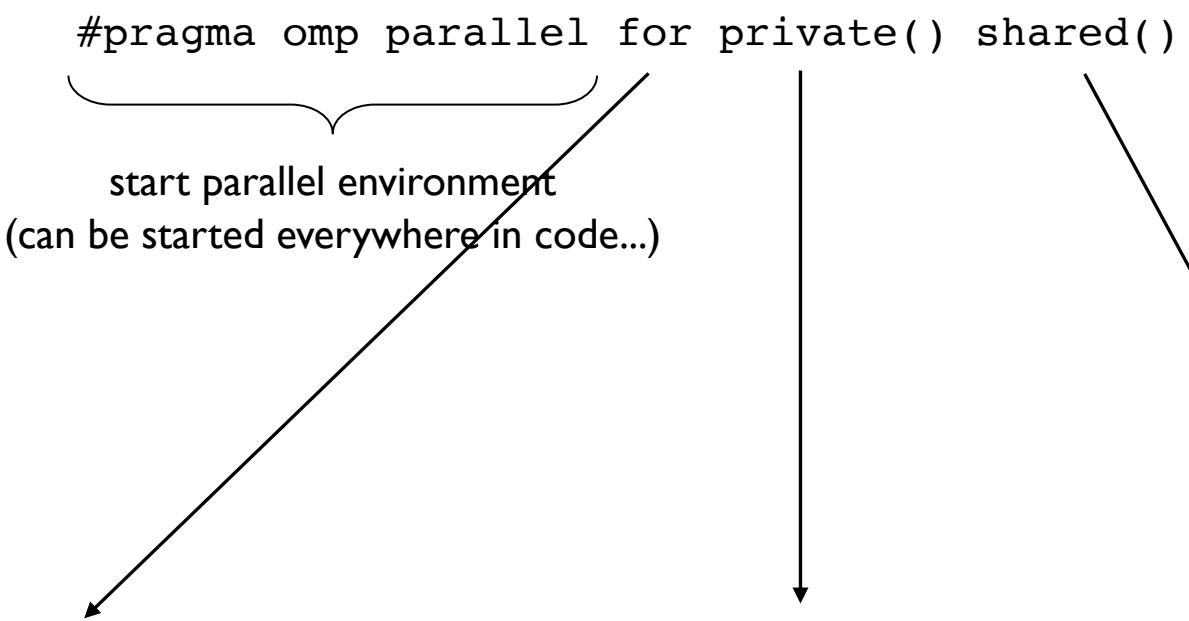

```
#pragma omp parallel for private(i,j,b) shared(a,N)
 for(i=1; i<N; i++) {

    b = a[0];
    for(j=0; j<i; j++) {
     b = function(b);
    }
    a[i] = b;
}
```

Note:

- the loop-counter has to be private
- if you only **read** the value of a variable, it can be 'shared'
- if you **write** into a variable, think carefully about its status

Computer Architectures

- how to program such machines (OpenMP standard):



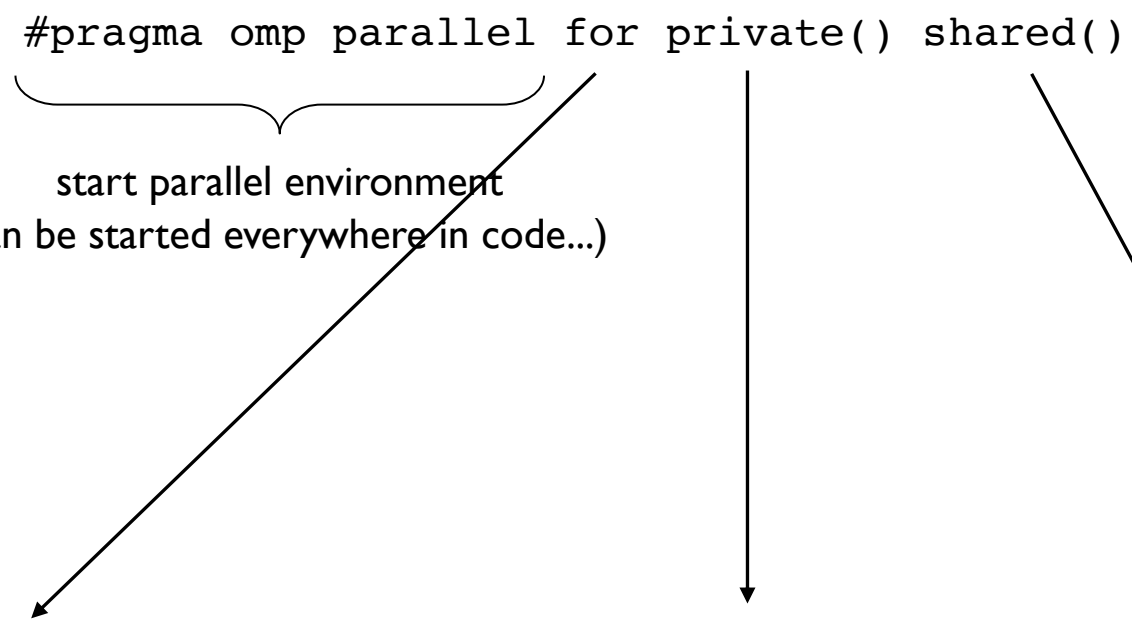

```
#pragma omp parallel for private(i,j,b) shared(a,N)
for(i=1; i<N; i++) {

    b = a[0];
    for(j=0; j<i; j++) {
     b = function(b);
    }
    a[i] = b;
}
```

j ?

Note:

- the loop-counter has to be private
- if you only **read** the value of a variable, it can be 'shared'
- if you **write** into a variable, think carefully about its status

- how to program such machines (OpenMP standard):




```
#pragma omp parallel for private(i,j,b) shared(a,N)
 for(i=1; i<N; i++) {

    b = a[0];
    for(j=0; j<i; j++) {
     b = f(b,a[j]);
    }
    a[i] = b;
}
```

Note:

- the loop counter has to be private
- if you only **read** the value of a variable, it can be 'shared'
- if you **write** into a variable, think carefully about its status

there is a far more elegant way to write this code!

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i,j,b) shared(a,N)
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }

   a[i] = b;
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i,j,b) shared(a,N)
for(i=1; i<N; i++) {
  b = a[0];
  for(j=0; j<i; j++) {
   b = function(b);
  }
  a[i] = b;
}
```

put this into a function!

Try it out!

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i,j,b) shared(a,N)
for(i=1; i<N; i++) {

   b = a[0];
   for(j=0; j<i; j++) {
    b = function(b);
   }

   a[i] = b;
}
```

put this into a function:

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;

   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i,j,b) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;

   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i,j,b) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;

   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;

   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;
?  #pragma omp parallel for...
   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

2 reasons for not parallelizing this for-loop...

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;
?  #pragma omp parallel for...
   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

2 reasons for not parallelizing this for-loop:

- it is a recursion
- we already parallelized outside of `calc_b()`

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;
?  #pragma omp parallel for...
   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;

#pragma omp parallel for private(i) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;

   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

general advise:

- make your code modular, i.e. use functions
- modular code is easier to parallelize

- how to program such machines (OpenMP standard):

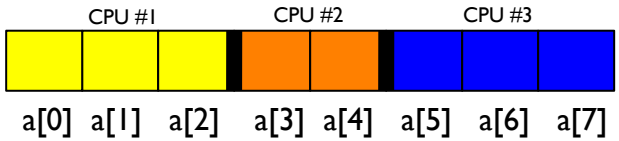
```
a[0] = STARTVALUE;

#pragma omp parallel for private(i) shared(a,N)
for(i=1; i<N; i++) {

   a[i] = calc_b(i, a[0]);
}
```

```
double calc_b(int i, double sv)
{
   double b;
   int j;

   b = sv;

   for(j=0; j<i; j++)  {
      b = function(b);
   }

   return(b);
}
```

general advise:

- make your code modular, i.e. use functions
- modular code is easier to parallelize

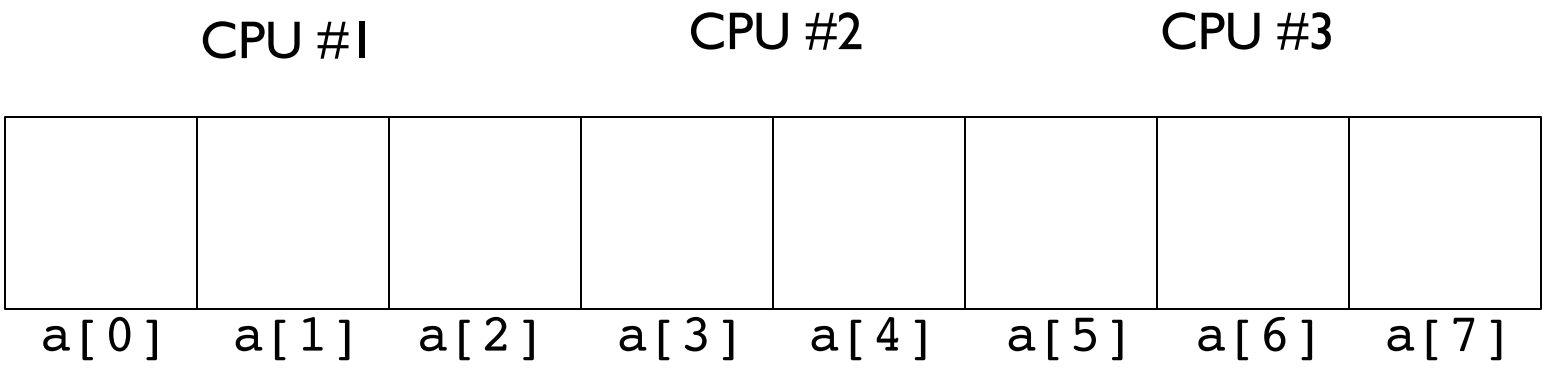
...but which CPU gets what *i* values?



Computer Architectures

- how to program such machines?

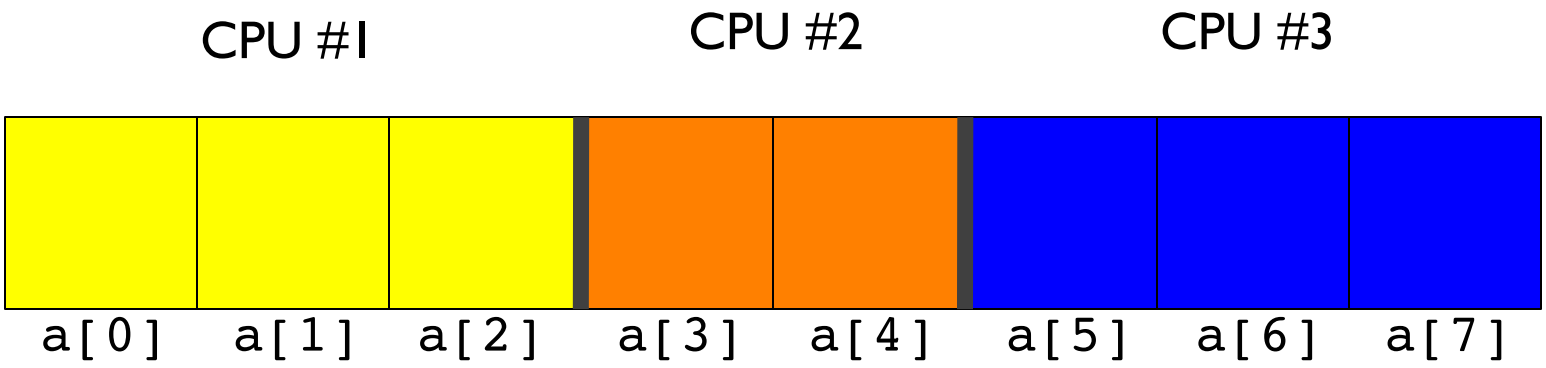
1D array



but how to divide the domain?

- how to program such machines?

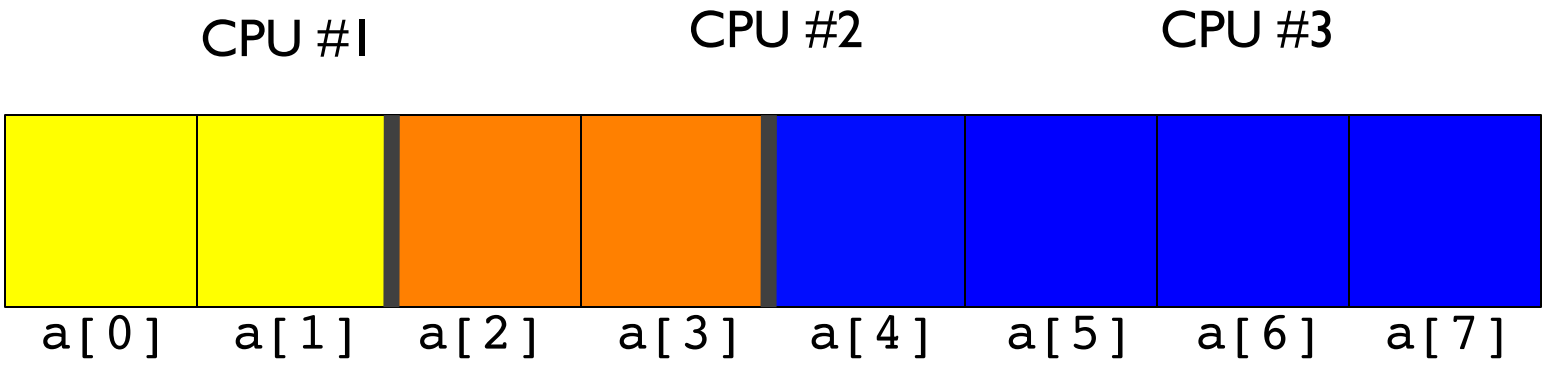
1D array



but how to divide the domain?

- how to program such machines?

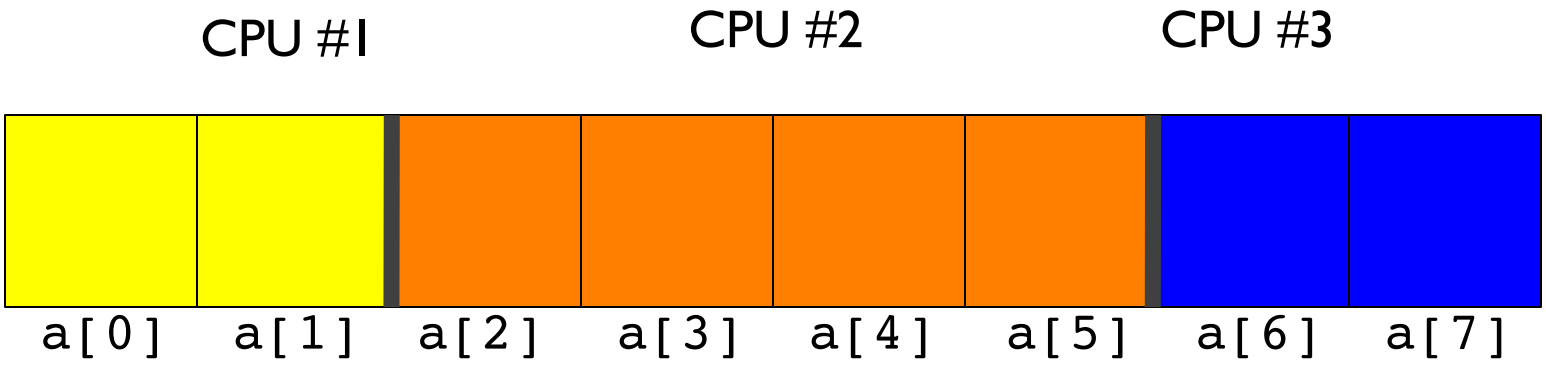
1D array



but how to divide the domain?

- how to program such machines?

1D array

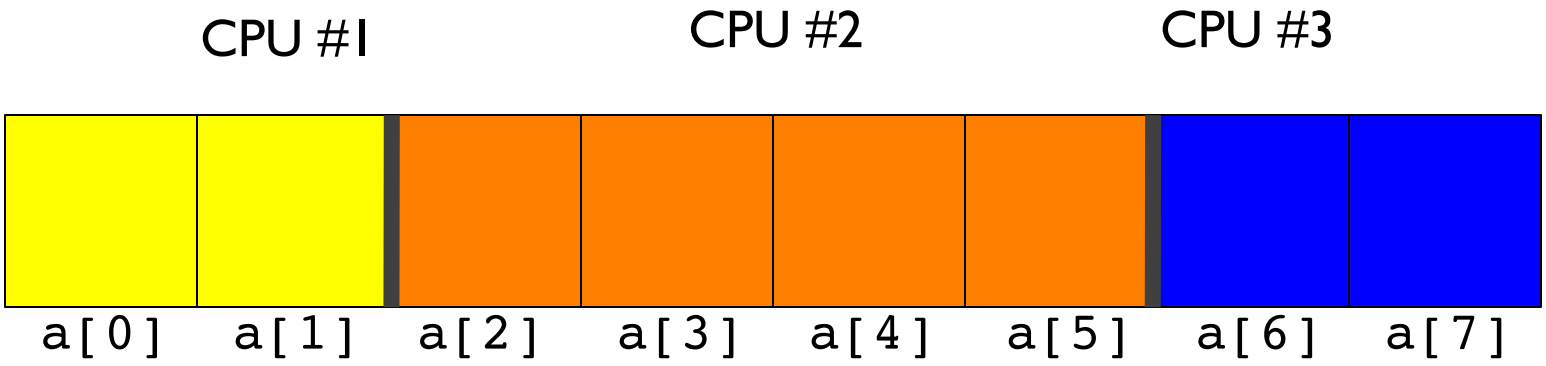


but how to divide the domain?

Computer Architectures

- how to program such machines?

1D array



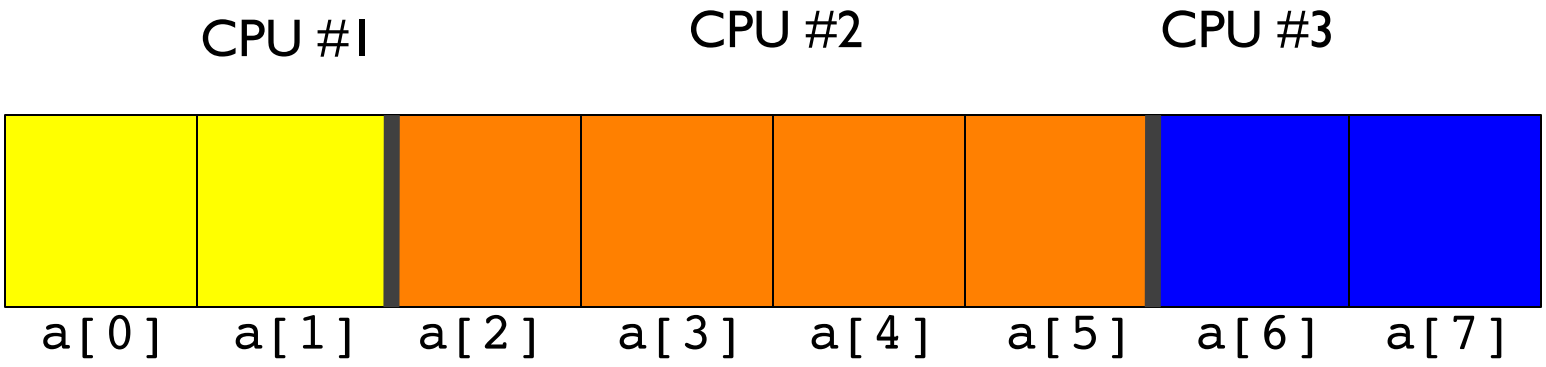
but how to divide the domain:
distribute the ***work*** evenly!

Set the number of threads to 3 in your last program, but using:
omp_set_num_threads(3);

How are the iterations distributed by default in your laptop?

- how to program such machines?

1D array



but how to divide the domain:
distribute the ***work*** evenly!

```
b = a[0];
for(j=0; j<i; j++) {
  b = function(b);
}
```

=> CPU's dealing with higher i 's have more work to do!

- how to program such machines?

1D array

OpenMP work distribution:

The schedule clause affects how loop iterations are mapped onto threads

`schedule(dynamic)`: loop index = 0-Nthreads-1 ⤻ Nthreads ⤻ Nthreads+1 ⤻ etc.

Each thread grabs a "chunk" from the iteration until it's finished. Most work at runtime.

`schedule (static)`: evenly divide loop index amongst Nthreads

Pre-determined and predictable by the programmer.

pragma omp for schedule(static,chunk)

usage:

```
#pragma omp parallel for private(...) shared(...) schedule(...)
```

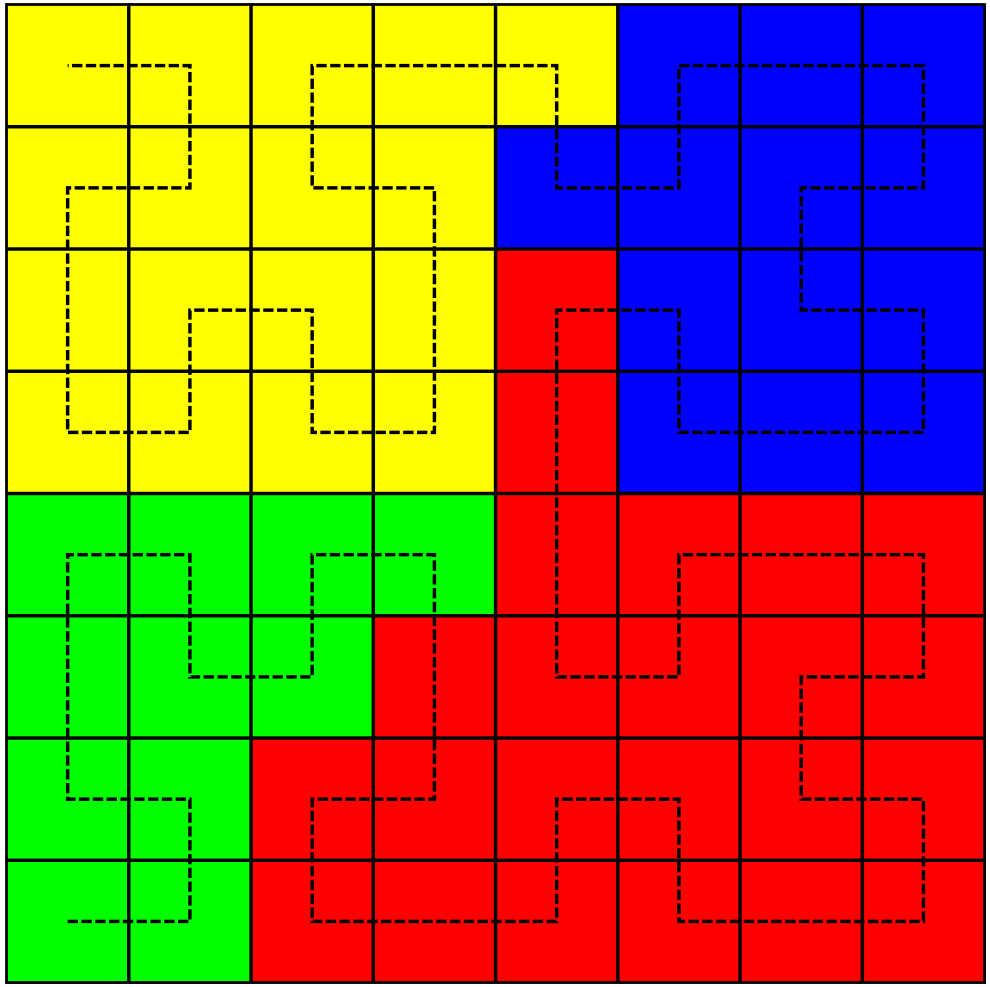
```
b = a[0];
for(j=0; j<i; j++) {
  b = function(b);
}
```

=> CPU's dealing with higher i 's have more work to do!

Explore how your program is perfomed if you set schedule to either dynamic or static with different chunks.

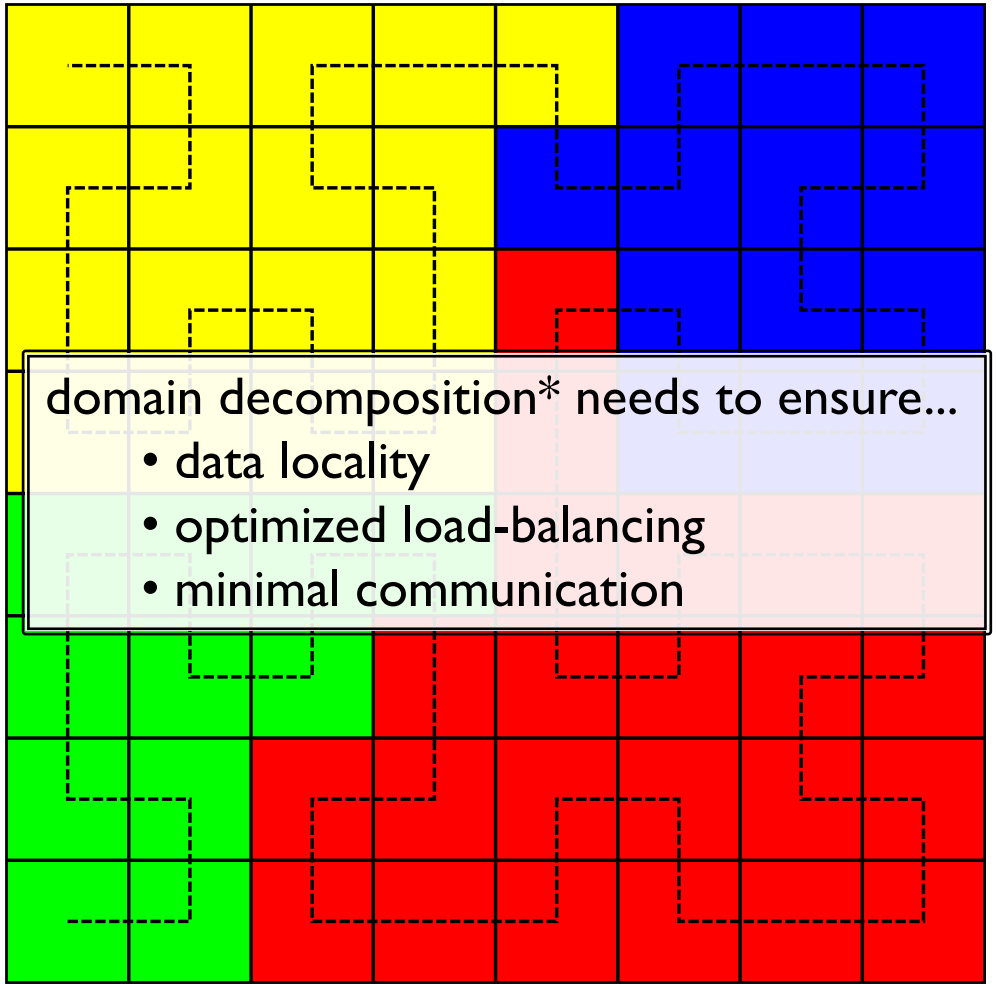
- how to program such machines?

2D array



- how to program such machines?

2D array

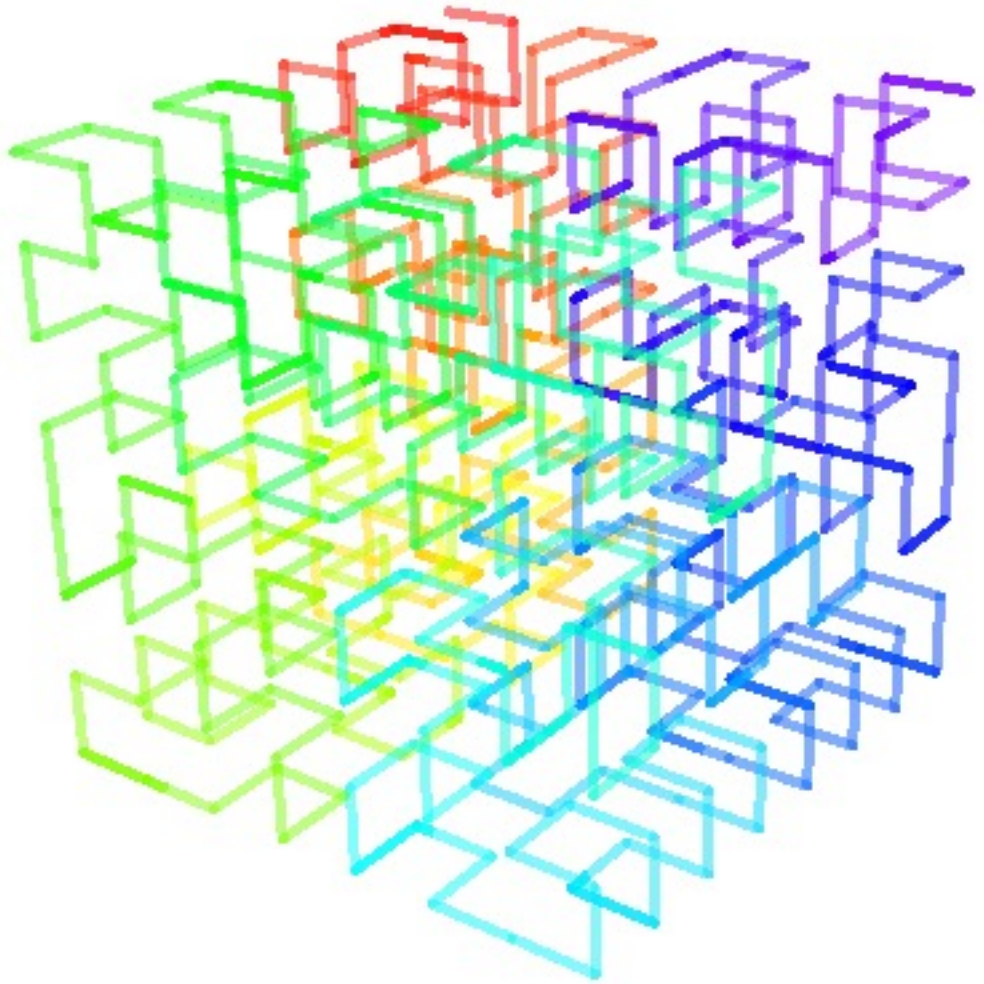


*this is not to be confused with domain discretisation!

Computer Architectures

- how to program such machines?

3D array



Computer Architectures

- how to check the speed-up of your program?

Computer Architectures

- how to check the speed-up of your program?

strong scaling **weak scaling**

- how to check the speed-up of your program?

strong scaling

you keep the problem size fixed,
but increase the number of CPU's

weak scaling

you keep the number of CPU's fixed,
but increase the problem size

- how to check the speed-up of your program?

strong scaling

you keep the problem size fixed,
but increase the number of CPU's

you aim at running a given problem as fast as possible...

weak scaling

you keep the number of CPU's fixed,
but increase the problem size

you aim at running the largest possible problem in a given amount of time...

- how to check the speed-up of your program?

strong scaling

you keep the problem size fixed,
but increase the number of CPU's

you aim at running a given problem as fast as possible...

weak scaling

you keep the number of CPU's fixed,
but increase the problem size

you aim at running the largest possible problem in a given amount of time...

actually more important nowadays

Computer Architectures

- how to actually write a program?

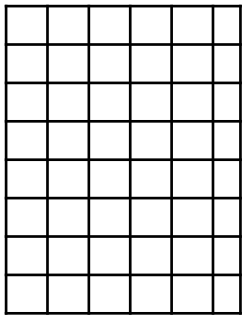
- how to actually write a program?
  - define the problem
  - decide on organisation
    - choose essential elements (variables, structures, etc.)
    - shape relevant tasks
    - design your algorithm to be parallelizable
    - draw a flowchart
  - code in your preferred language
  - test code using simple/known test cases

- how to actually write a program?
  - define the problem
  - decide on organisation
    - choose essential elements (variables, structures, etc.)
    - **shape relevant tasks**
    - **design your algorithm to be parallelizable**
    - draw a flowchart
  - code in your preferred language
  - test code using simple/known test cases

try to break problem down into pieces/modules that can be coded separately from each other...

Computer Architectures

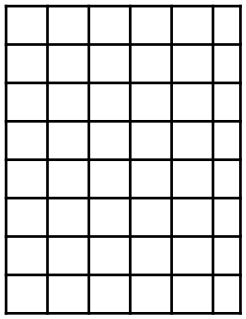
- shaping relevant tasks?



Computer Architectures

- shaping relevant tasks – data

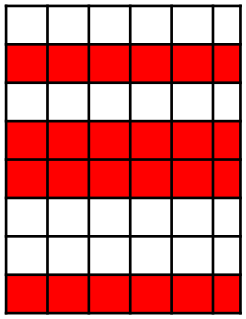
divide data into sub-sets,
and perform different calculation with each sub-set…



Computer Architectures

- shaping relevant tasks – data

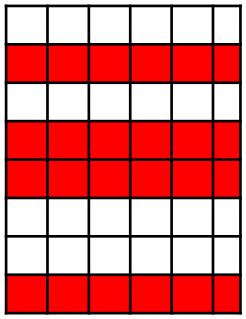
divide data into sub-sets,
and perform different calculation with each sub-set…



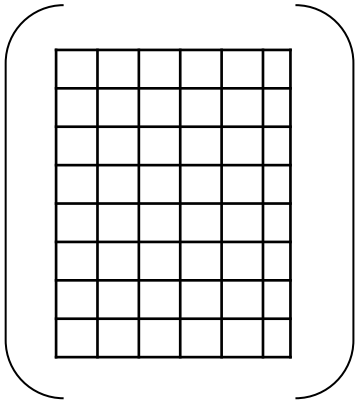
Computer Architectures

- shaping relevant tasks – data

divide data into sub-sets,
and perform different calculation with each sub-set…



Analyse(Data)

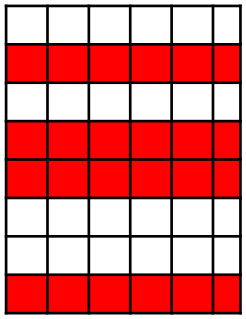


```
void Analyse(Data) {
 for(Data) {
   if(redData){
      Calculation(Data)
      }
   else {
      Calculation(Data)
      }
} }
```

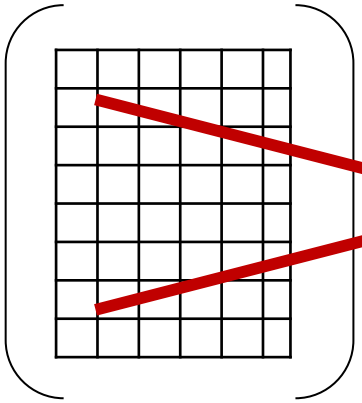
Computer Architectures

- shaping relevant tasks – data

divide data into sub-sets,
and perform different calculation with each sub-set…



Analyse(Data)

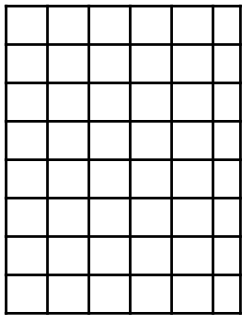


```
void Analyse(Data) {
 for(Data) {
   if(redData){
      Calculation(Data)
      }
   else {
      Calculation(Data)
      }
} }
```

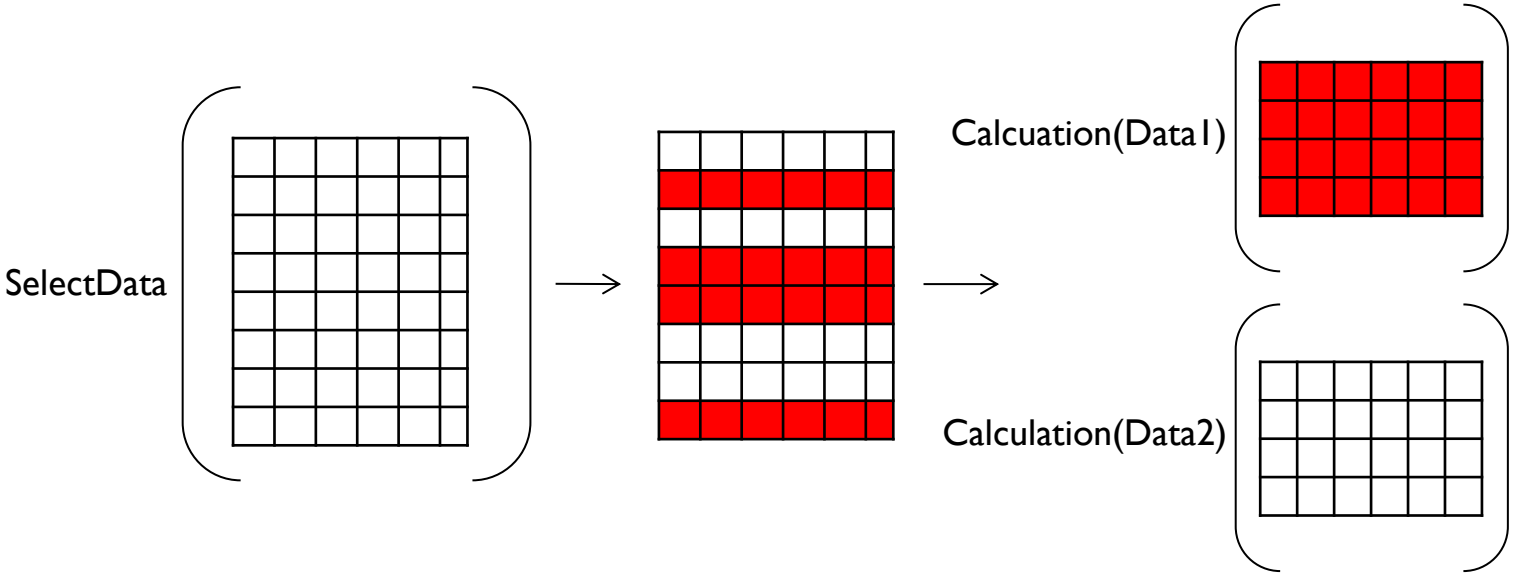
Computer Architectures

- shaping relevant tasks – data

divide data into sub-sets,
and perform different calculation with each sub-set…

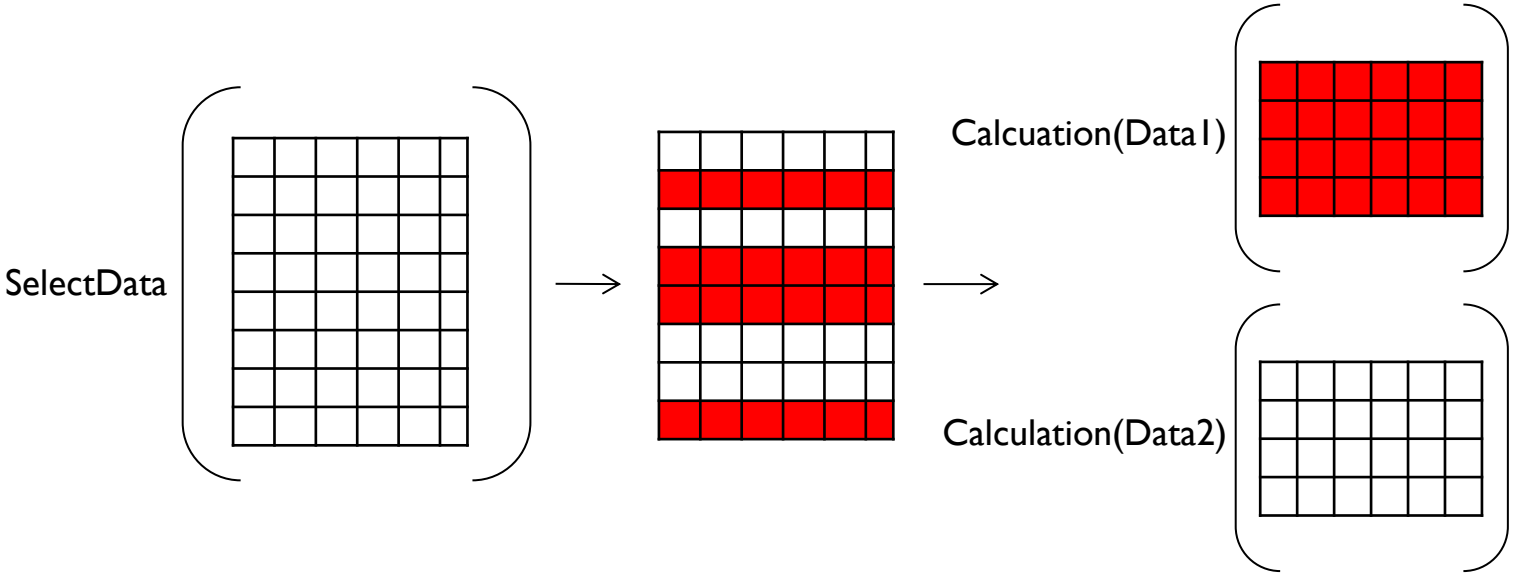


- shaping relevant tasks – data



Computer Architectures

- shaping relevant tasks – data



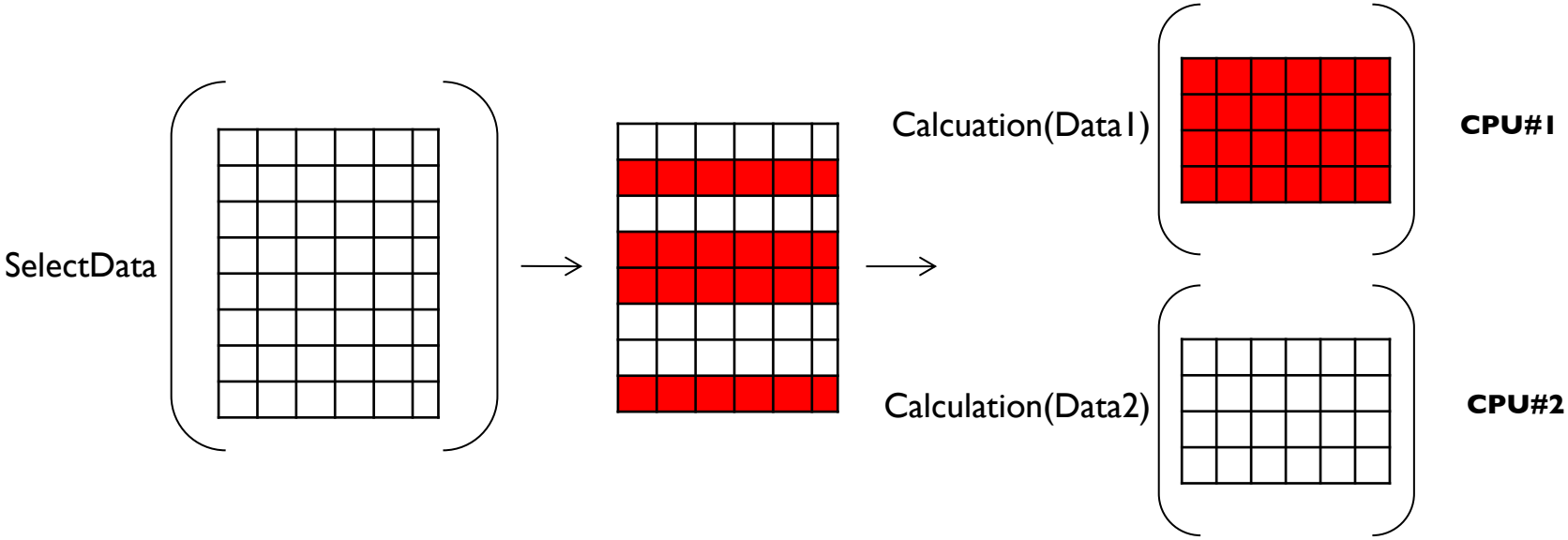
modular and hence more flexible!

- how to actually write a program?
  - define the problem
  - decide on organisation
    - choose essential elements (variables, structures, etc.)
    - shape relevant tasks
    - **design your algorithm to be parallelizable**
    - draw a flowchart
  - code in your preferred language
  - test code using simple/known test cases

each CPU runs the same code,
but on a different part of the problem...

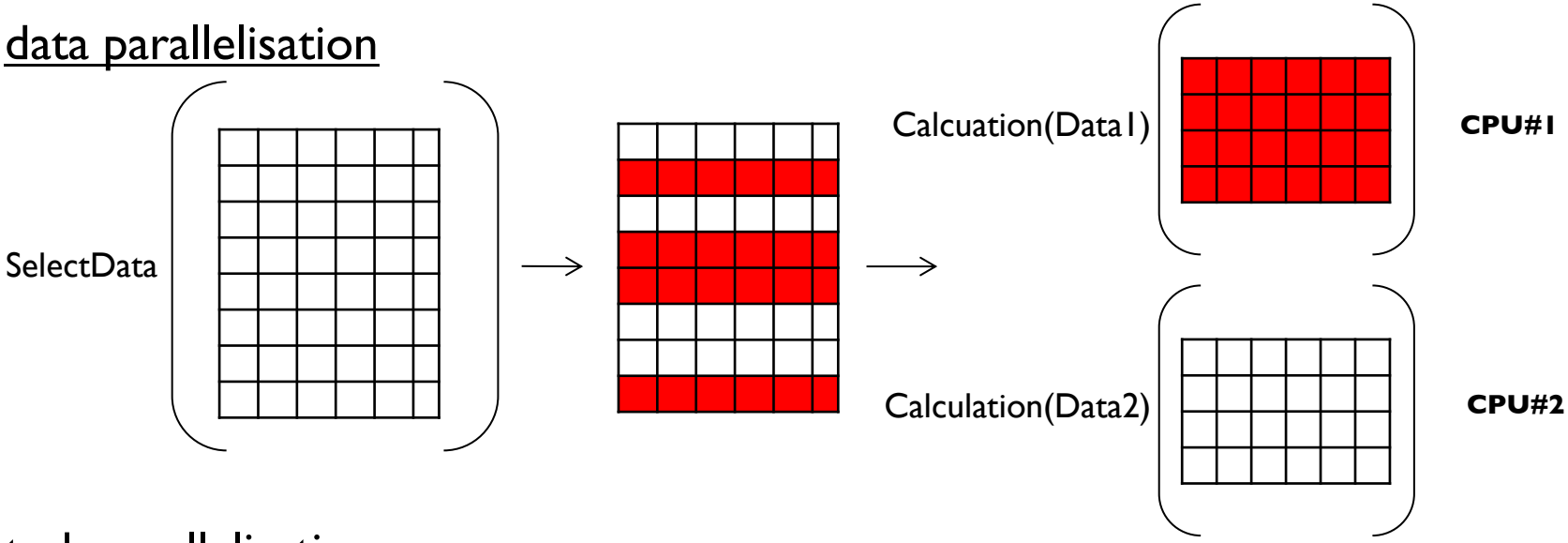
Computer Architectures

- design your algorithm to be parallelizable!



- design your algorithm to be parallelizable!

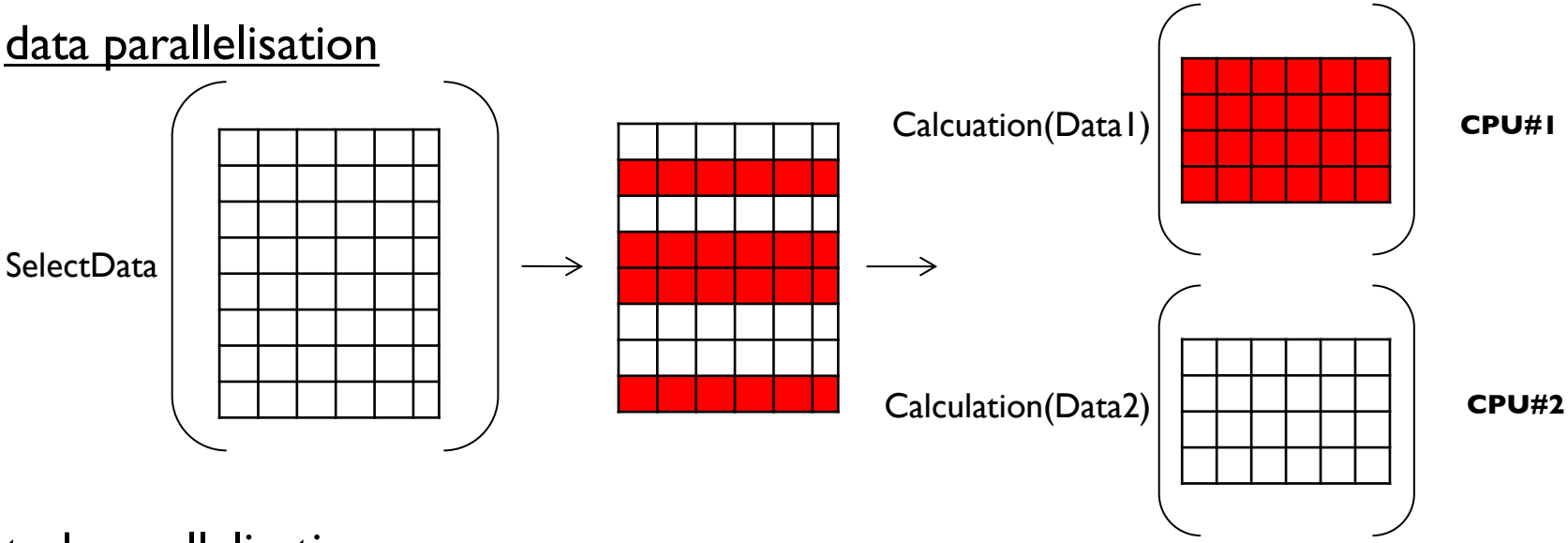
data parallelisation



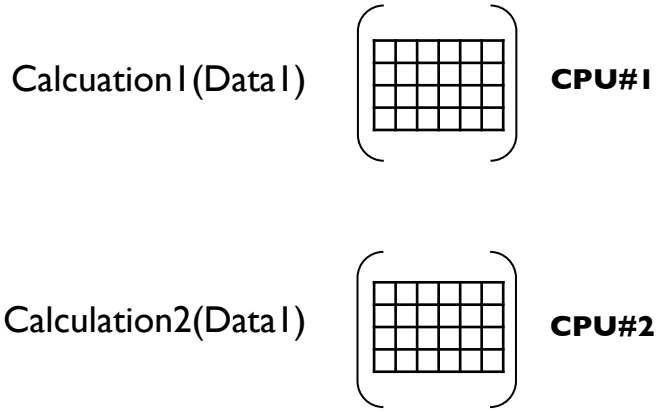
task parallelisation

- design your algorithm to be parallelizable!

data parallelisation



task parallelisation



- some coding recommendations:
  - make proper use of the Cache (see above)
    - avoid complicated indices
    - know how arrays are aligned in memory
  - avoid conditions, I/O, and (sub-)routine calls inside loops
  - avoid unnecessary operations inside loops in general
  - use multiplications rather than divisions or powers
  - keep it simple!

Computer Architectures

let's put our knowledge into action:

write a code that calculates

$$\pi \approx \sqrt{12} \sum_{k=0}^{\infty} \left(-\frac{1}{3}\right)^k \frac{1}{2k+1}$$

You can first write a serial program to ensure your function is working as expected and play with the values of k.

Then think how are you going to parallelise your code.

For the total sum use: reduction(+:sum)

#pragma omp parallel for reduction(+:sum)

(This is by default initialised to 0.)

Compare the performance of your serial and parallel programs, using

k=0,1,2,3,4

```
pi += pow(-1.0/3.0, (double)k) / (2.*(double)k+1.);
```

k=5,6,7,8,9

```
pi += pow(-1.0/3.0, (double)k) / (2.*(double)k+1.);
```

k=10,11,12,13

```
pi += pow(-1.0/3.0, (double)k) / (2.*(double)k+1.);
```