

- architectures
- real machines
- computing concepts
- parallel programming

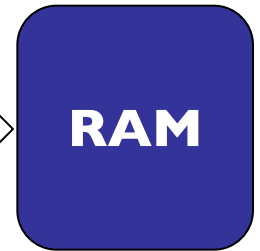
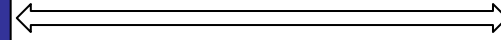
- **architectures**
- real machines
- computing concepts
- parallel programming

## Computer Architectures

- serial machine



=

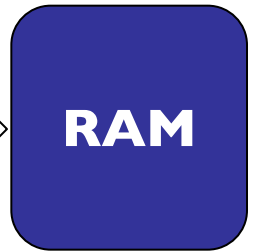
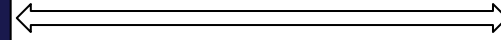


## Computer Architectures

- serial machine



=



### CPU:

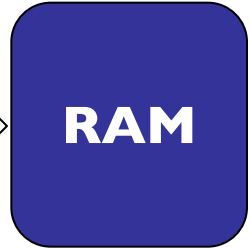
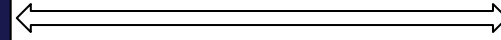
- very primitive commands,  
obtained from compilers or interpreters of higher-level languages

## Computer Architectures

- serial machine



=



### CPU:

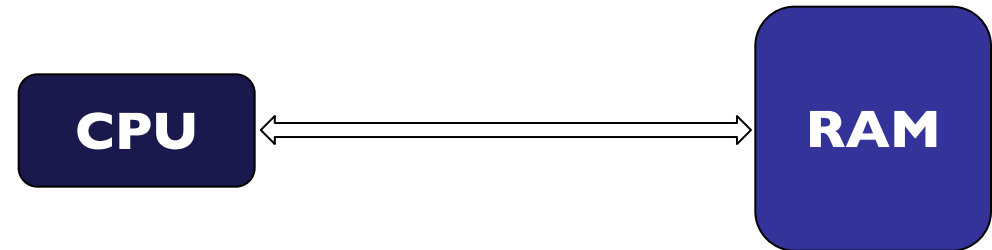
- very primitive commands, obtained from compilers or interpreters of higher-level languages
- cycle chain:
  - **fetch** – get instruction and/or data from memory
  - **decode** – store instruction and/or data in register
  - **execute** – perform instruction

## 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**

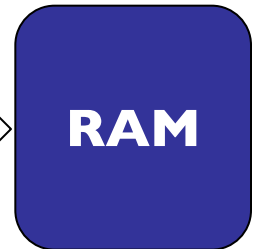
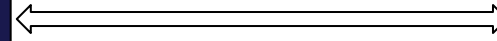
**arithmetical/logical instructions:** +, -, \*, /, bitshift, if

## Computer Architectures

## ▪ serial machine



=

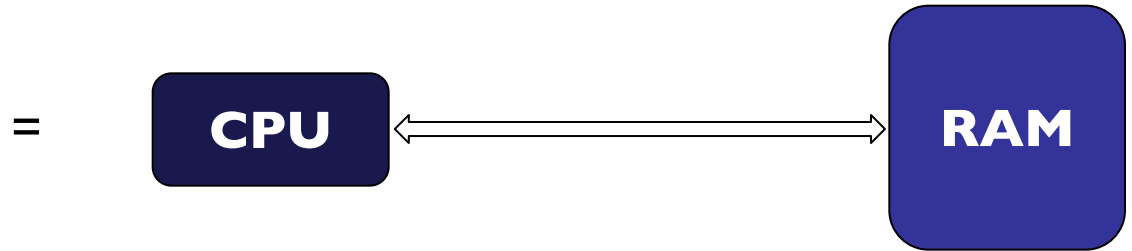
**CPU:**

- 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



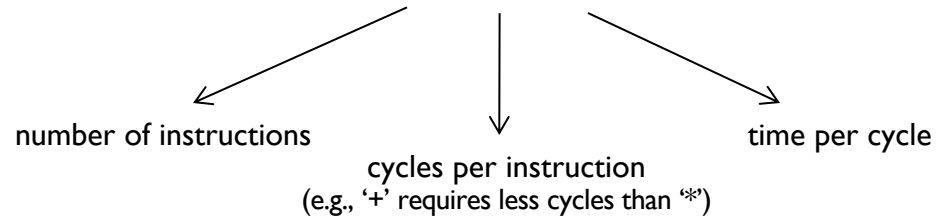
## Computer Architectures

- serial machine



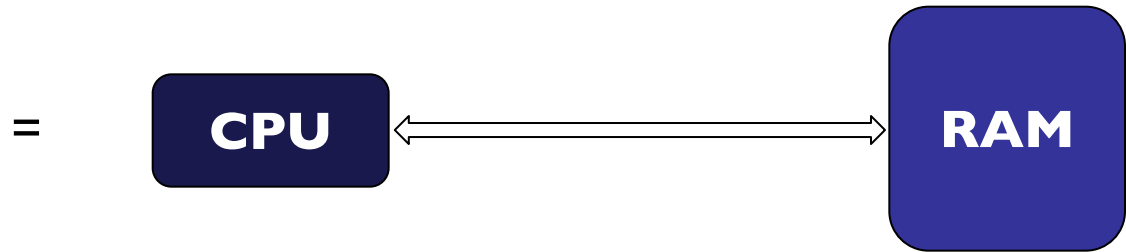
**CPU:**

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



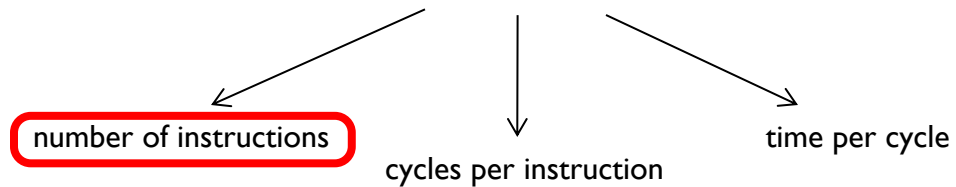
## Computer Architectures

- serial machine



### CPU:

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



### speed-ups:

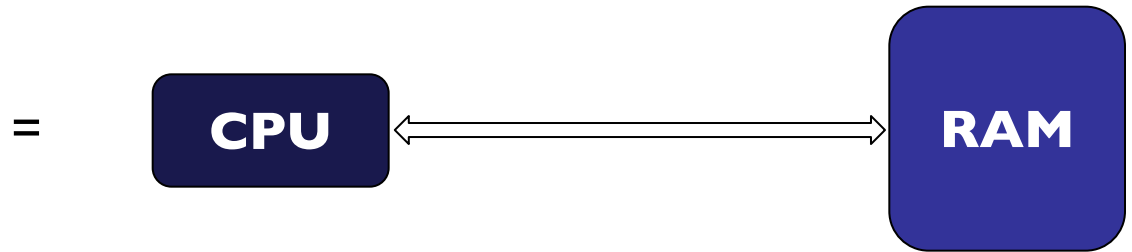
**improve your algorithm to require less instructions**

### 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...

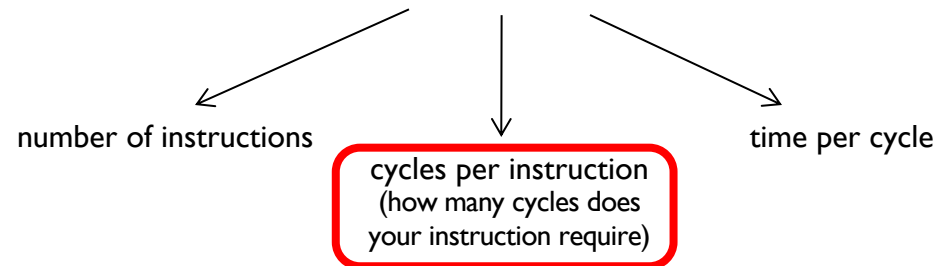
## Computer Architectures

- serial machine



### CPU:

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



### speed-ups:

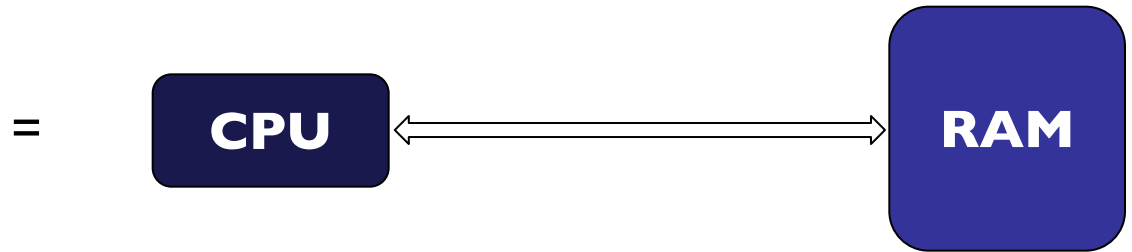
**improve your algorithm to use more adequate instructions**

### example:

avoid at all costs  $\text{pow}()$ ,  $\text{log}()$ , etc.,  
e.g.  $\text{pow}(x, 2)$  should be replaced with  $x*x$

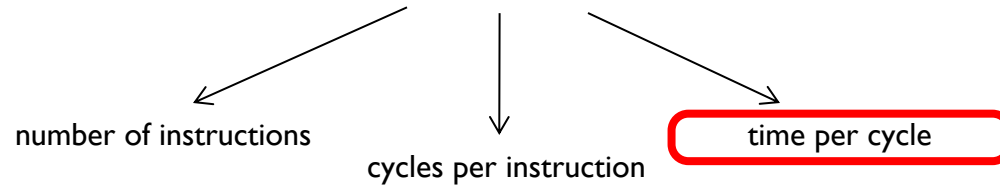
## Computer Architectures

- serial machine



**CPU:**

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

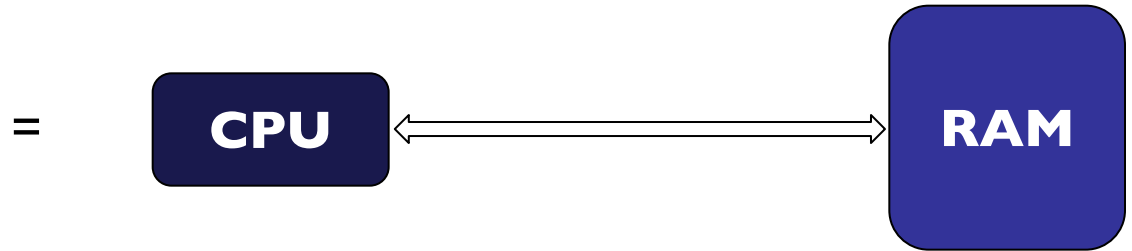


speed-ups:

**buy a machine with higher clock-frequency**

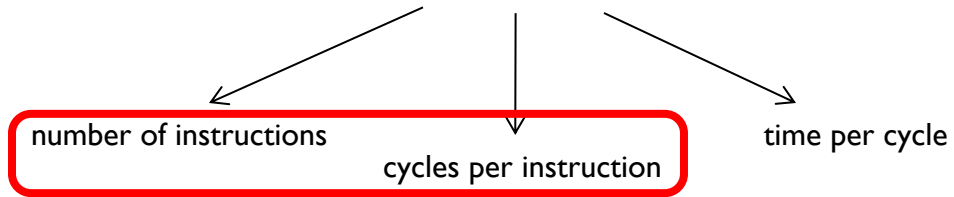
## Computer Architectures

- serial machine



**CPU:**

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

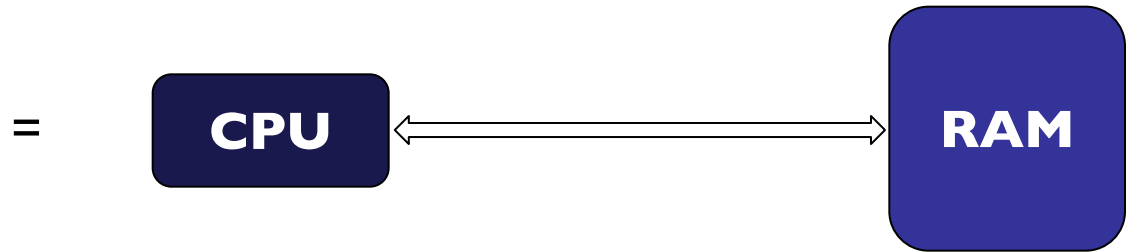


speed-ups:

**improve your algorithm!**

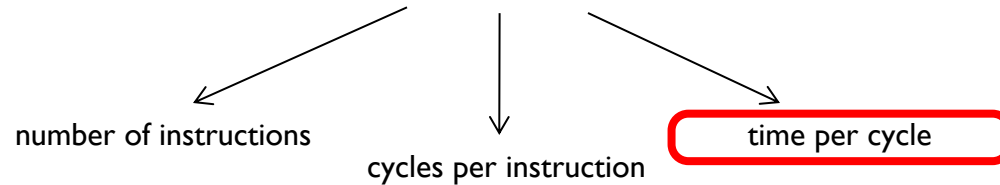
## Computer Architectures

- serial machine



**CPU:**

▪ execution time:  $t = n_i \times \text{CPI} \times t_c$



speed-ups:

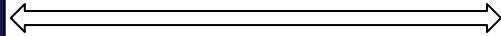
...or wait for technology to advance ;-)

## Computer Architectures

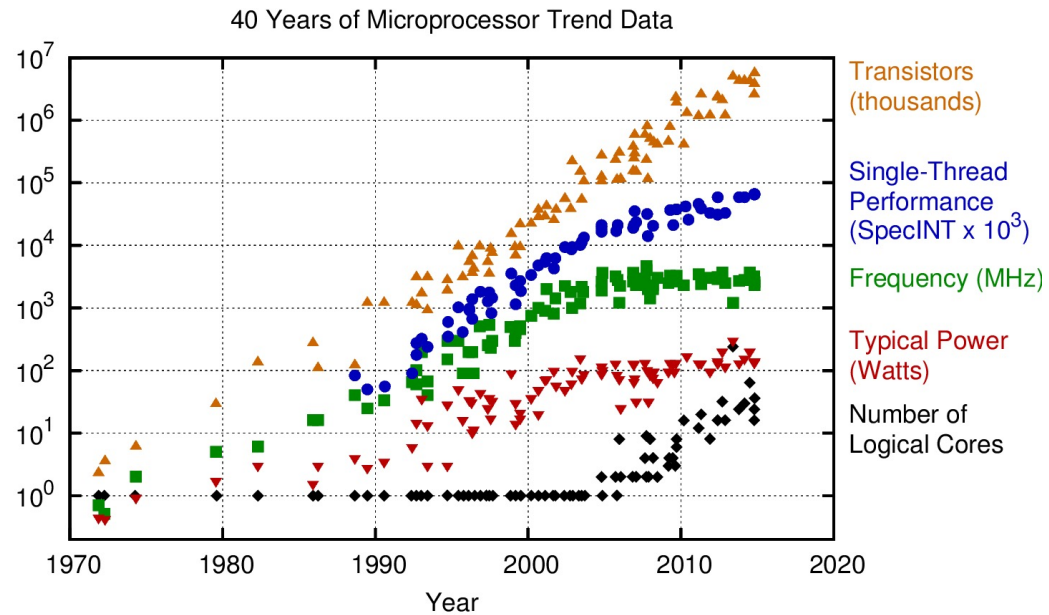
- serial machine



=



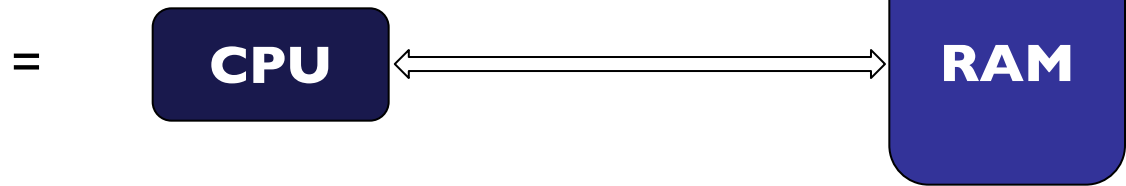
### CPU - evolution during the past years:



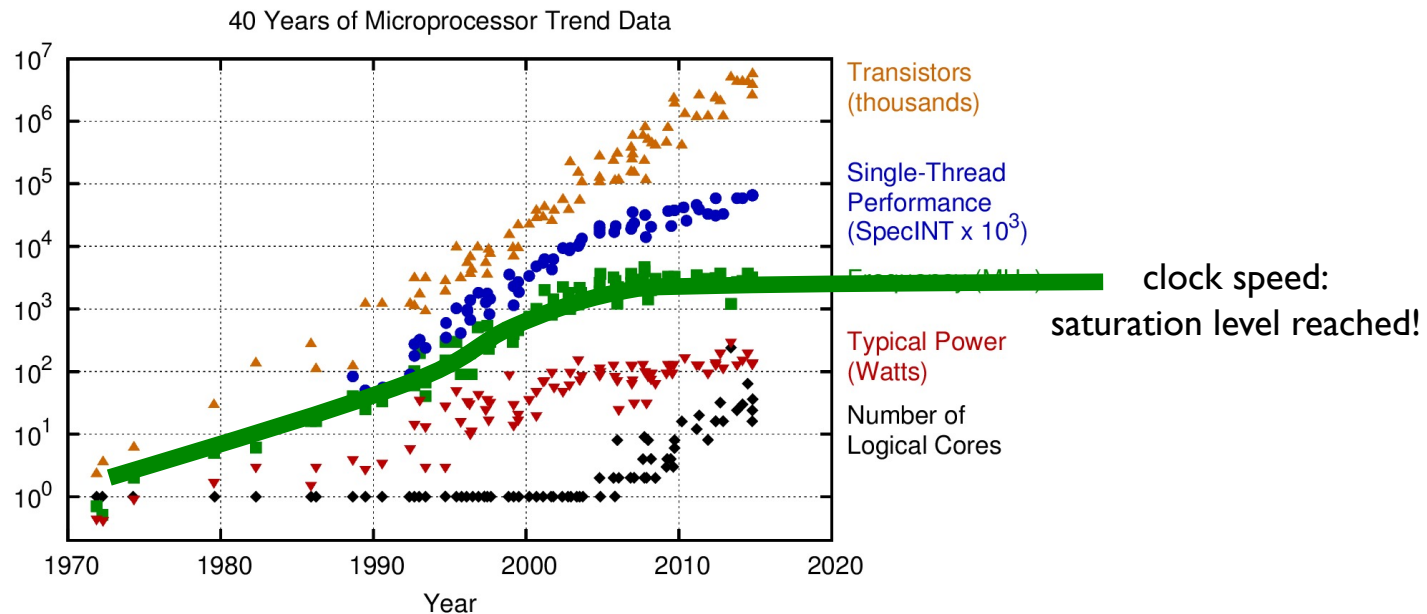
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

## Computer Architectures

- serial machine



### CPU - evolution during the past years:

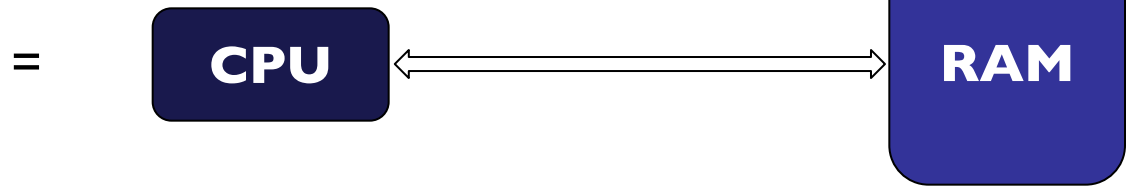


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

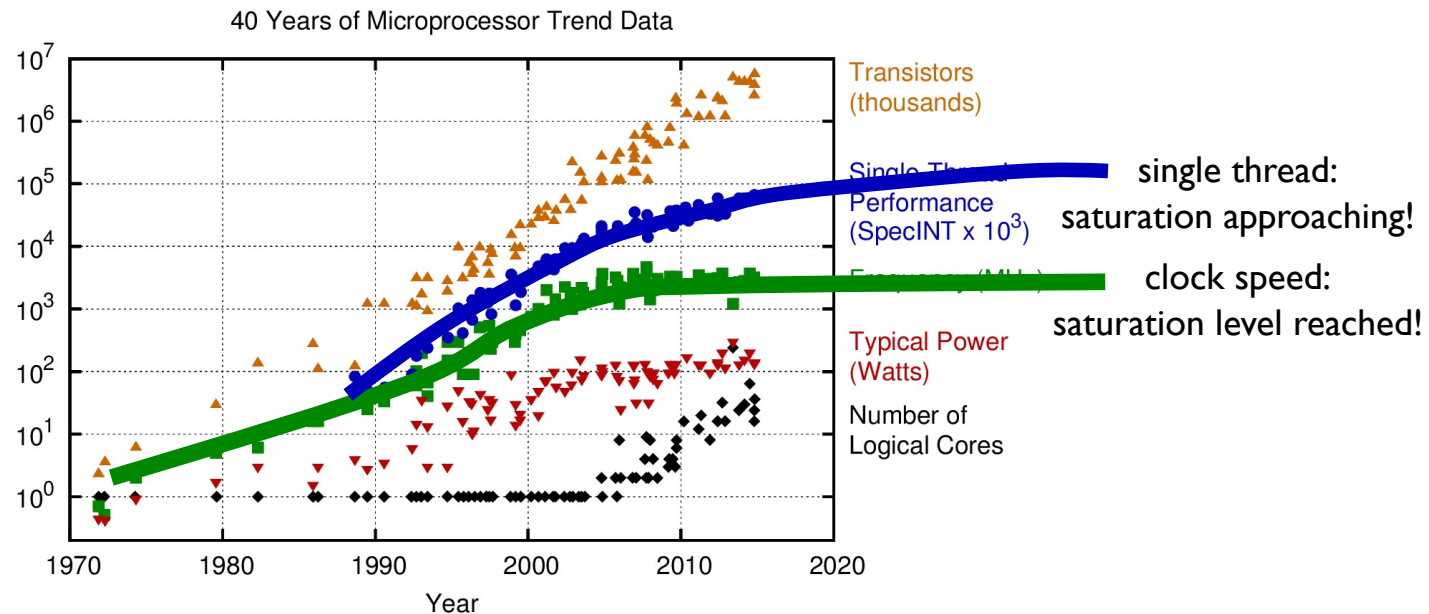


## Computer Architectures

- serial machine



### CPU - evolution during the past years:



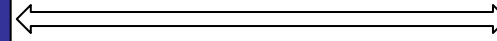
Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten  
New plot and data collected for 2010-2015 by K. Rupp

## 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)

## 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

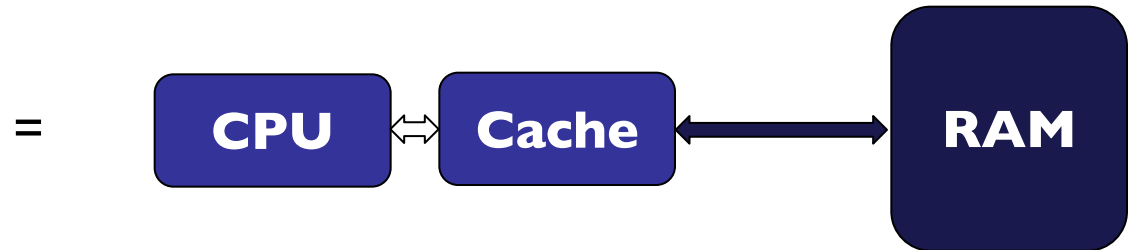
## ▪ serial machine

**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

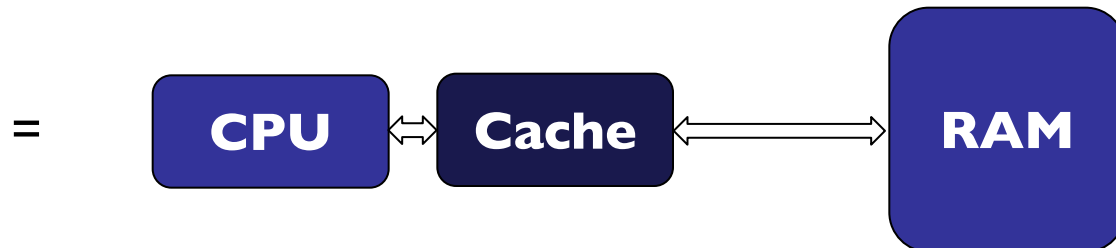
- serial machine

**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**

## 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”:

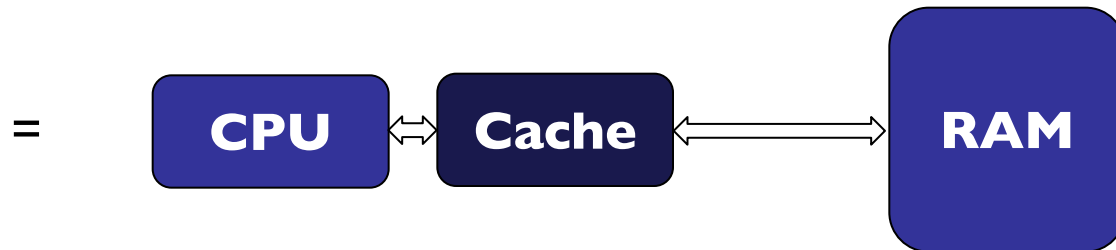
$$t = f \times t_{\text{Cache}} + (1 - f) \times t_{\text{RAM}}$$

$f$  → Cache hit rate     
  $t_{\text{Cache}}$  → Cache access time     
  $t_{\text{RAM}}$  → 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":

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

$\swarrow$                        $\swarrow$                        $\swarrow$   
**Cache hit rate**      **Cache access time**      **RAM access time**

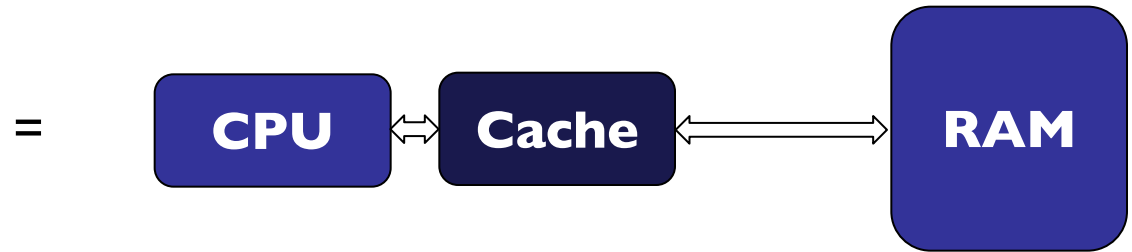
example:

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

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

## 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”:

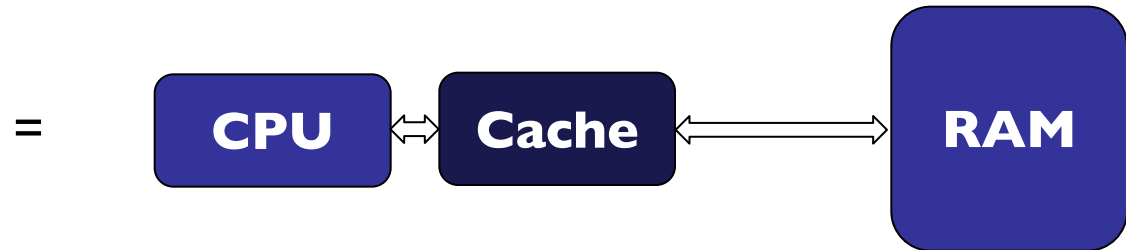
$$\bullet t = f \times t_{\text{Cache}} + (1-f) \times t_{\text{RAM}}$$

$\swarrow$                        $\searrow$                        $\searrow$   
**Cache hit rate**      **Cache access time**      **RAM access time**

example:  $f = 0.1, t_{\text{Cache}} = 10ns, t_{\text{RAM}} = 100ns$   
 $f = 0.9, t_{\text{Cache}} = 10ns, t_{\text{RAM}} = 100ns$

## 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”:

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

$\swarrow$                        $\searrow$                        $\searrow$   
**Cache hit rate**      **Cache access time**      **RAM access time**

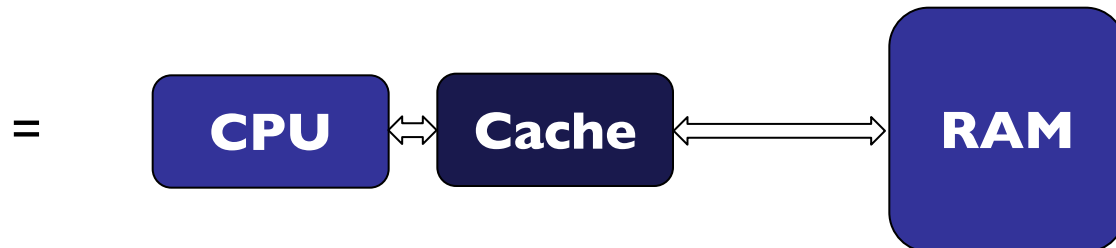
example:

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

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

## 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”:

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

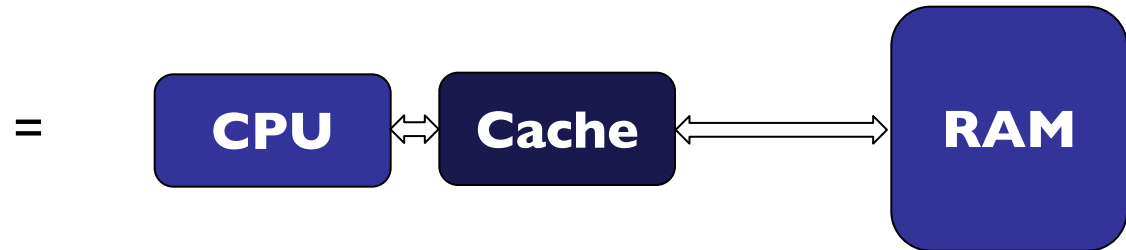
$\swarrow$  Cache hit rate       $\swarrow$  Cache access time       $\swarrow$  RAM access time

example:

$f = 0.1,$	$t_{\text{Cache}} = 10\text{ns},$	$t_{\text{RAM}} = 100\text{ns}$	$\Rightarrow 91\text{ns}$	↪ factor of 4.5!
$f = 0.9,$	$t_{\text{Cache}} = 10\text{ns},$	$t_{\text{RAM}} = 100\text{ns}$	$\Rightarrow 19\text{ns}$	

## 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”:

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

$f$  → Cache hit rate     
  $t_{\text{Cache}}$  → Cache access time     
  $t_{\text{RAM}}$  → RAM access time

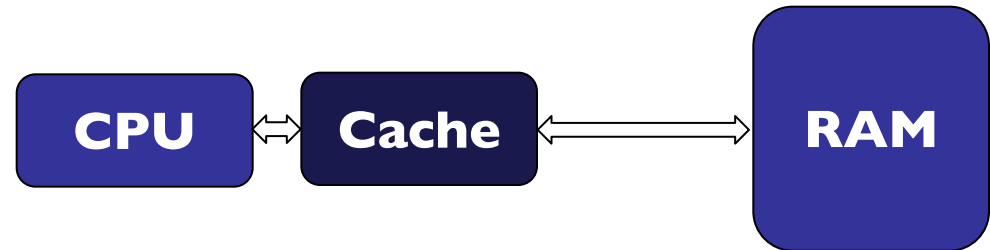
**design your code in order to access contiguous memory blocks!**

## Computer Architectures

- serial machine



=

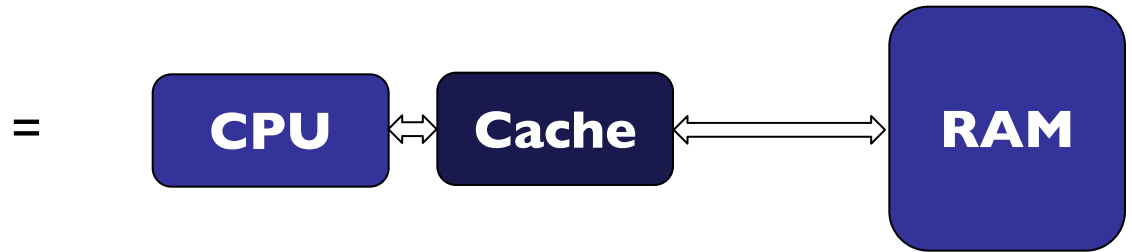


- Cache hits and misses:

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

## Computer Architectures

- serial machine



- Cache hits and misses:

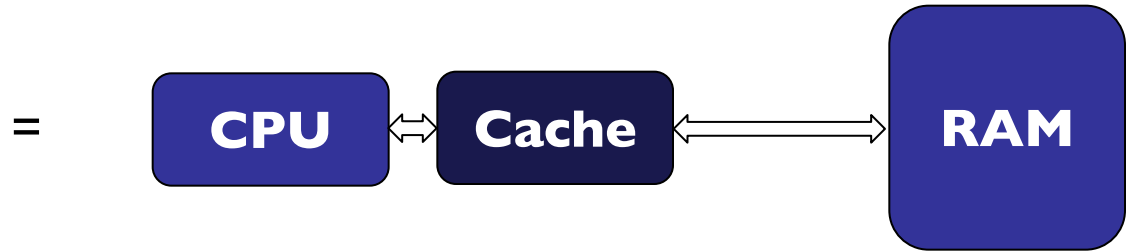
- 2D array in C: `density[2][3]`

- memory alignment:



## Computer Architectures

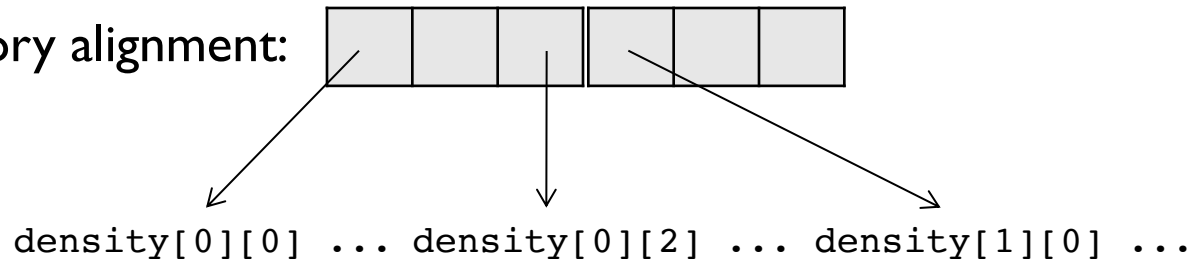
- serial machine



- Cache hits and misses:

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

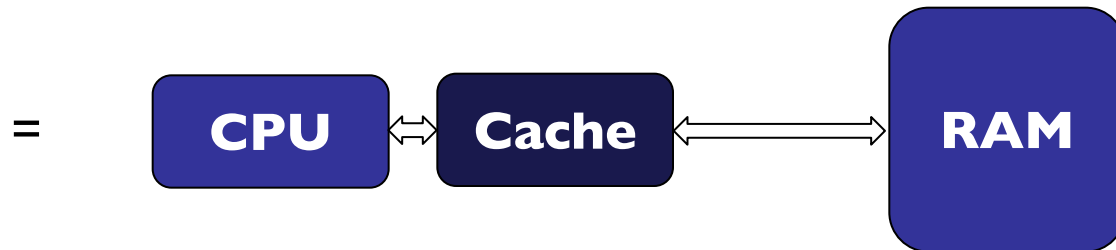
- memory alignment:





## Computer Architectures

- serial machine



- Cache hits and misses:

```

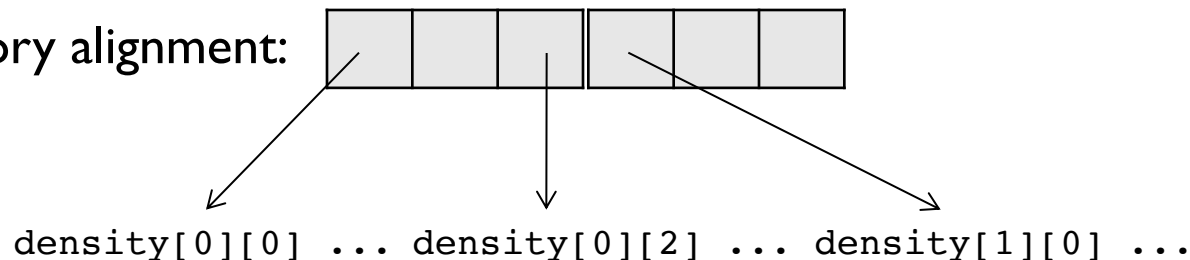
for(i=0; i<3; i++)
  for(j=0; j<2; j++)
    whatever(density[j][i]);
  
```

- 2D array in C:

density[2][3]

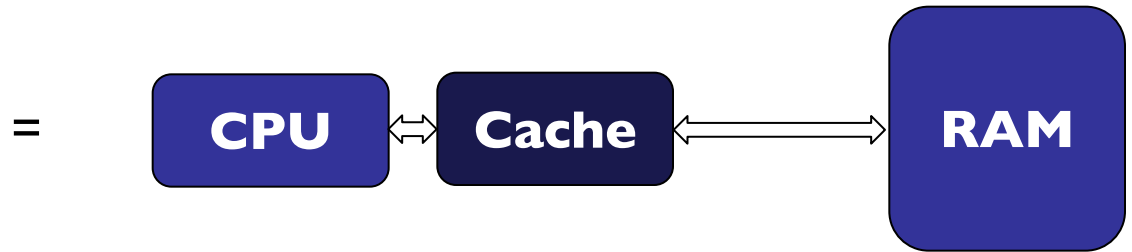
**BAD!**

- memory alignment:



## Computer Architectures

- serial machine



- Cache hits and misses:

- 2D array in C:

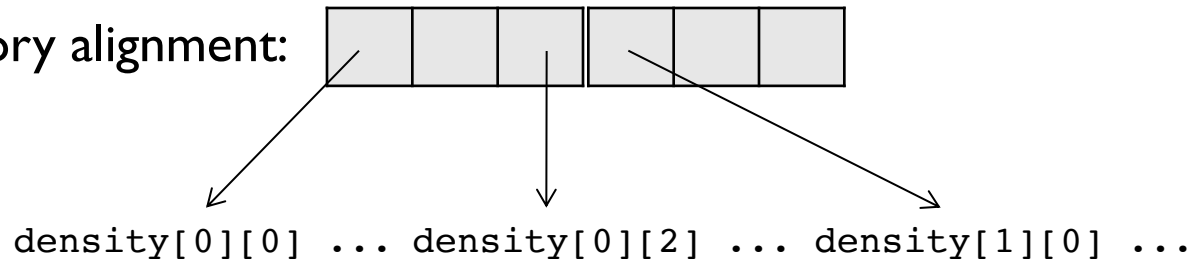
density[2][3]

```

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

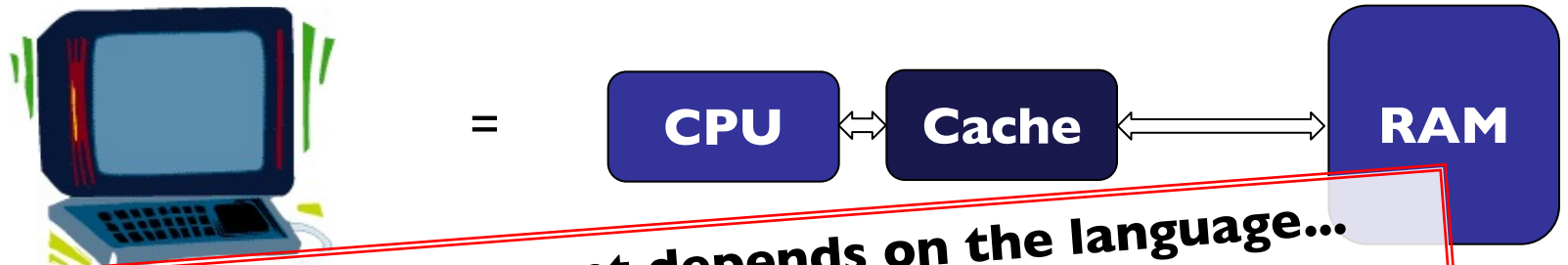
**GOOD!**

- memory alignment:



Computer Architectures

- serial machine

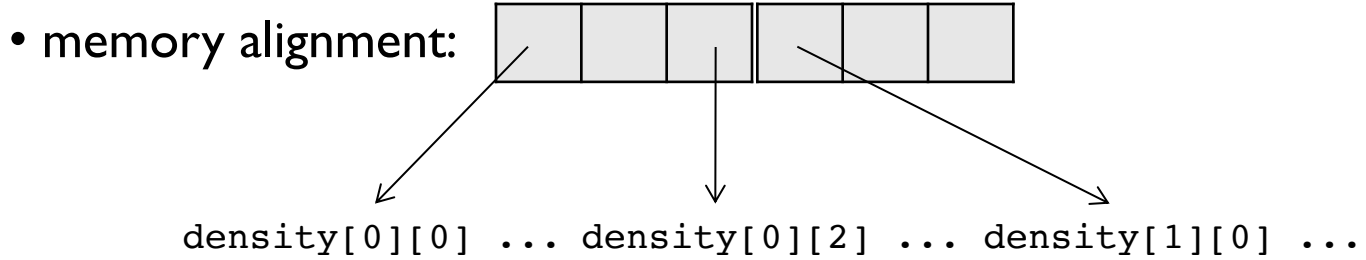


**memory alignment depends on the language...  
check before programming!**  
(e.g. C uses row-major ordering while Fortran & MATLAB use column-major ordering)

- Cache hits and misses
  - for(j=0; j<2; j++)
    - for(i=0; i<3; i++)
      - whatever(density[j][i]);

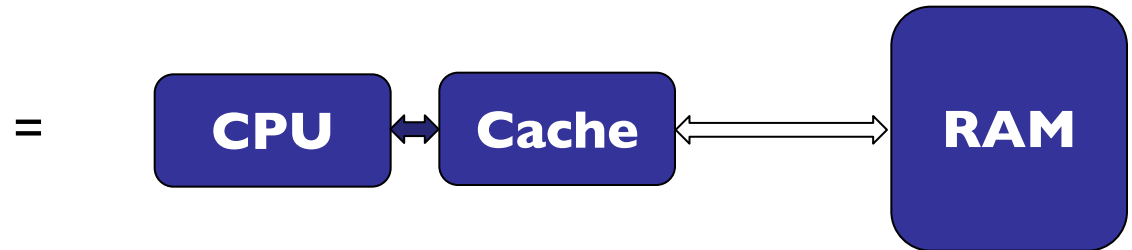
- 2D array in C: `density[2][3]`

**GOOD!**



## Computer Architectures

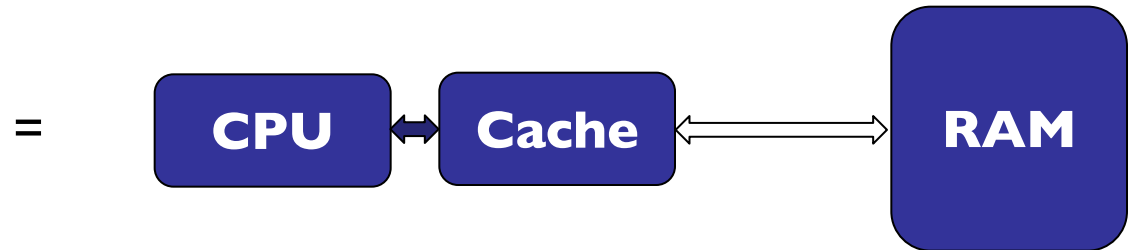
## ▪ serial machine

**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

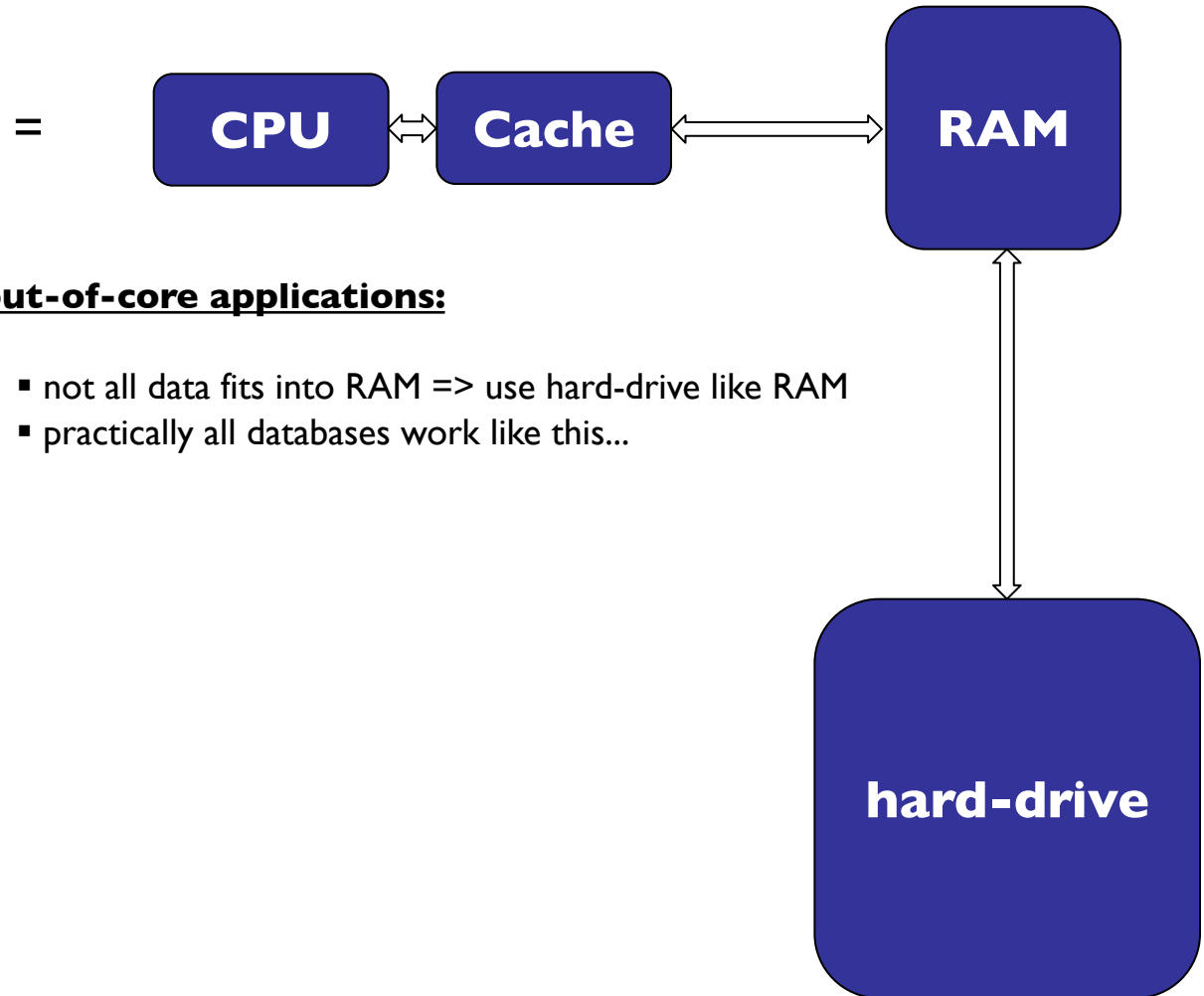
## ▪ serial machine

**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!

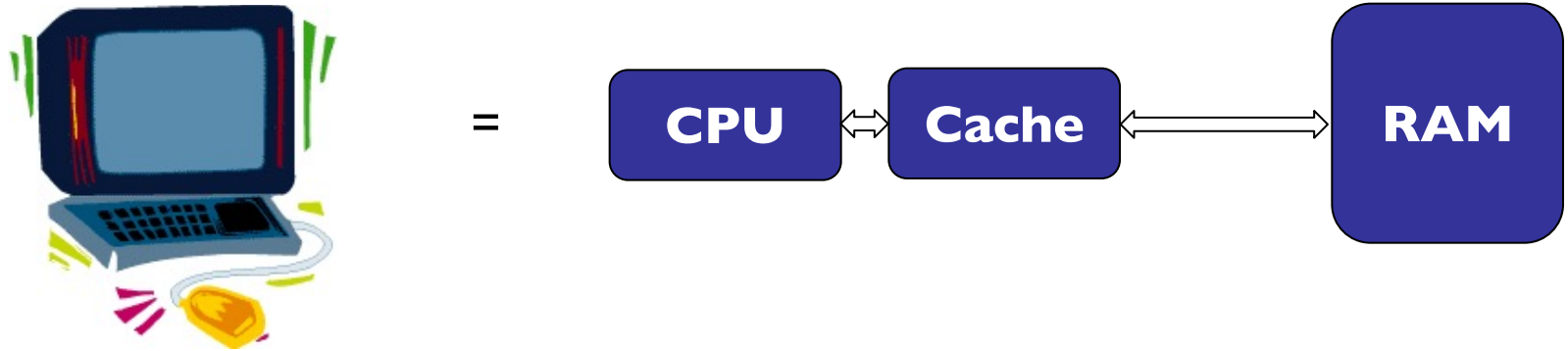
## Computer Architectures

- serial machine



## Computer Architectures

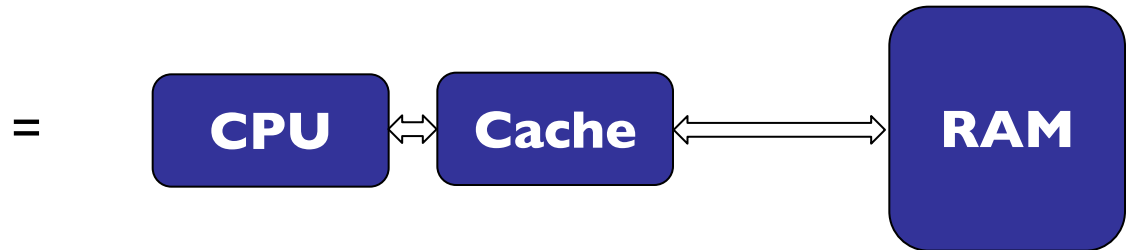
## ▪ serial machine

possible speed-ups by the programmer:

- improve your algorithm to require less instructions, e.g.  $f=4 \cdot \pi / G r a v$
- improve your algorithm to use more adequate instructions, e.g.  $x \cdot x$  instead of  $\text{pow}(x, 2)$
- 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.  $f=4 \cdot \pi / G_{\text{rav}}$
- improve your algorithm to use more adequate instructions, e.g.  $x \cdot x$  instead of  $\text{pow}(x, 2)$
- proper usage of cache, e.g. check memory alignment

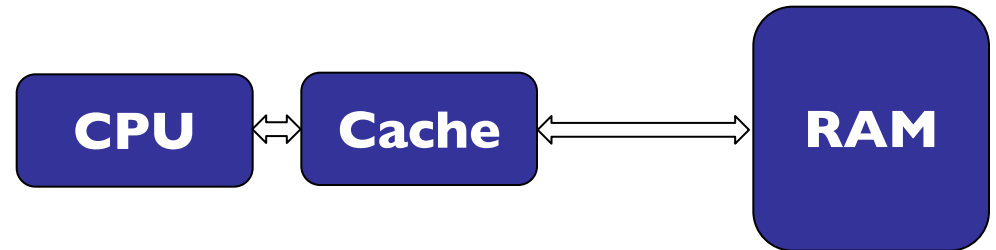


## Computer Architectures

- serial machine



=

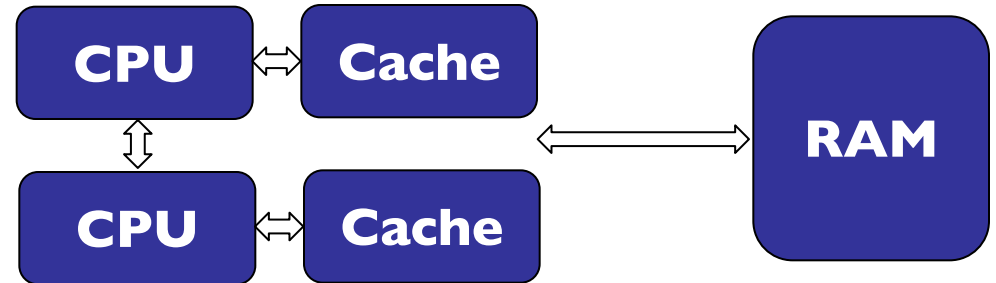


## Computer Architectures

- serial machine – multi-cored

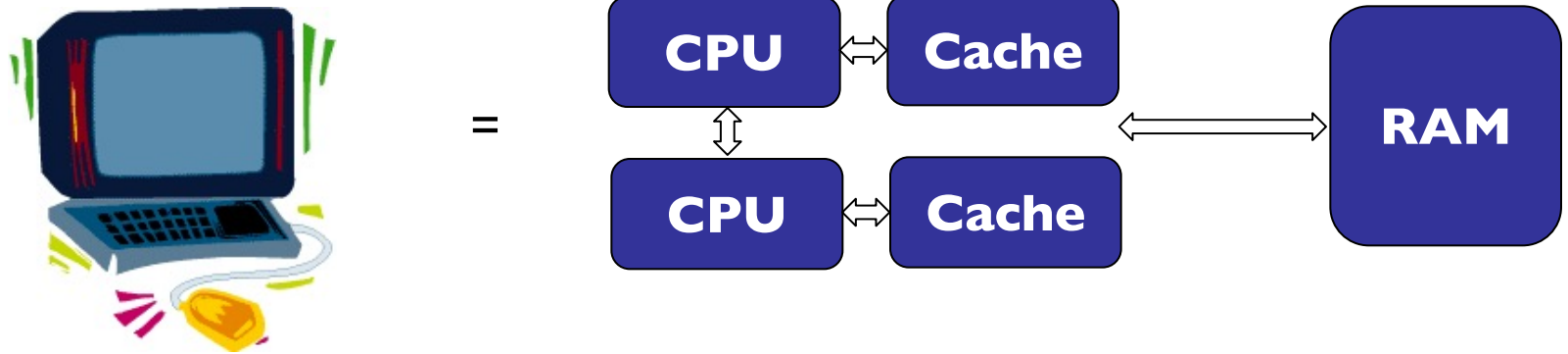


=



## Computer Architectures

- serial machine – multi-cored



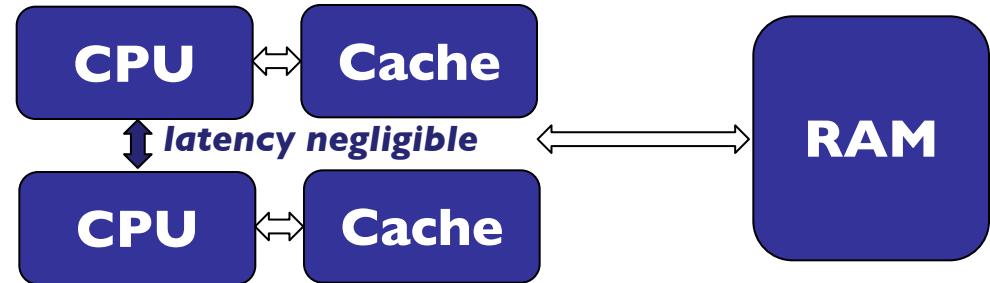
- shared memory architecture
  - easy to adapt existing serial code
  - limited by RAM to be placed into a single machine

## Computer Architectures

- serial machine – multi-cored



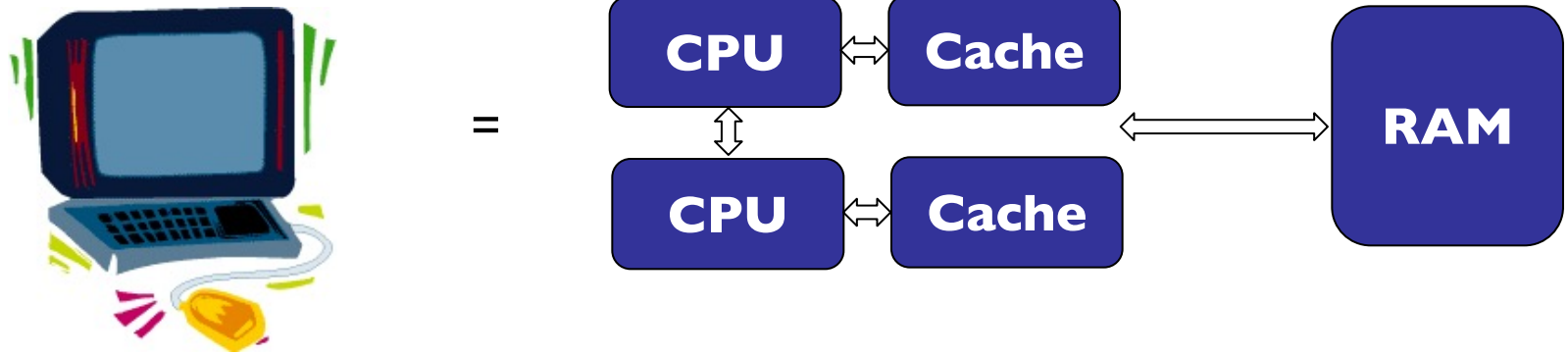
=



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

## 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](http://www.openmp.org)

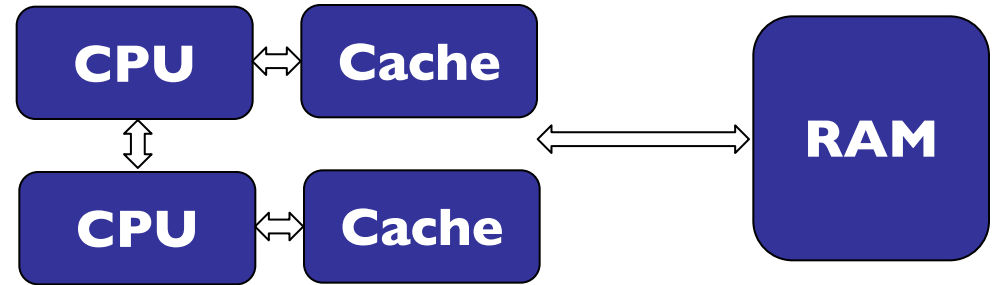
- most commonly used standard to parallelize code on shared memory architectures
- primarily distribute `for`-loop components onto different CPU's
- natively by supported by gcc since v4.2

## Computer Architectures

- serial machine – multi-cored



=



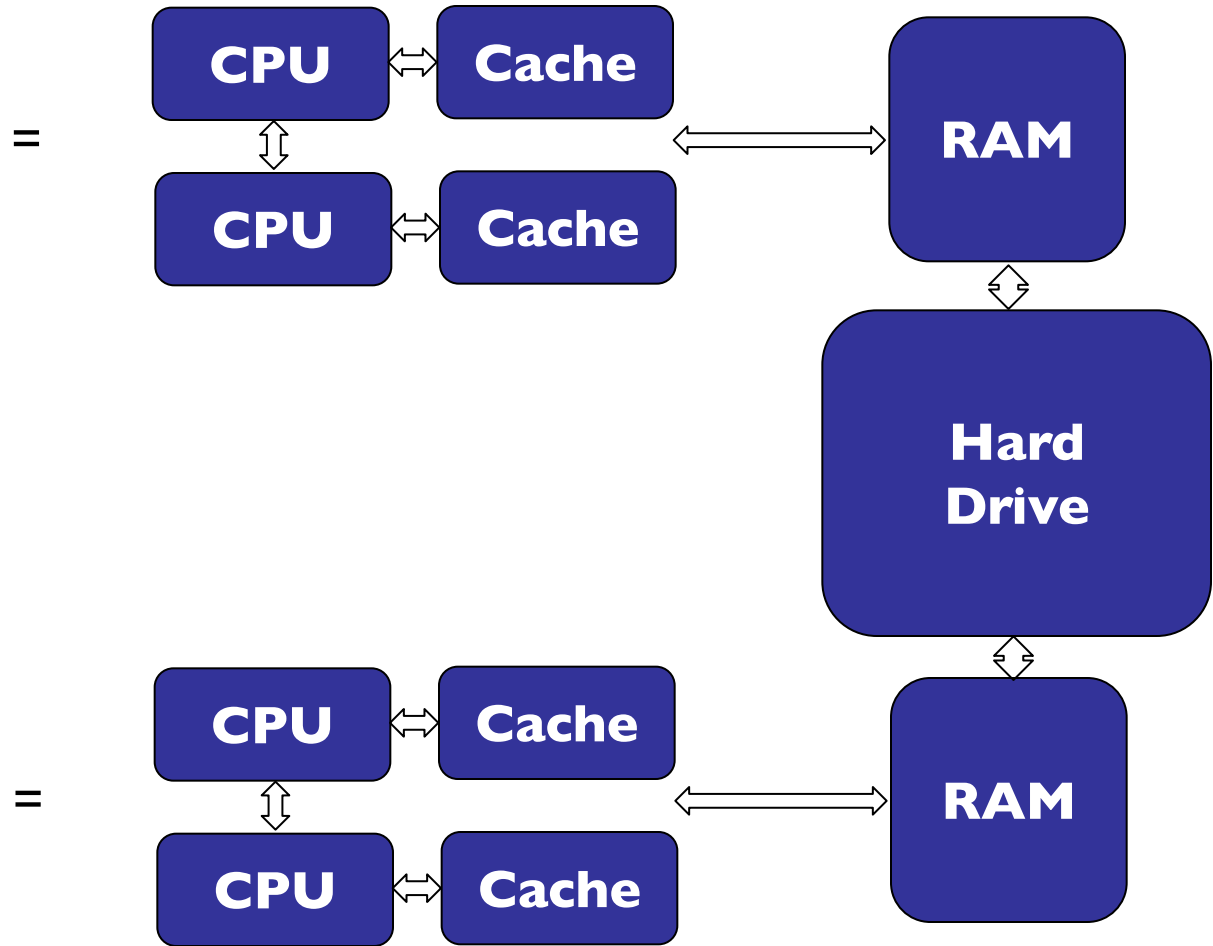
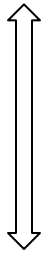
- shared memory architecture
  - easy to adapt existing serial code
  - limited by RAM to be placed into a single machine

- OpenMP – [www.openmp.org](http://www.openmp.org)

- most commonly used standard to parallelize code
- **you(!) have to add extra commands to the code**
- supported by gcc since v4.2

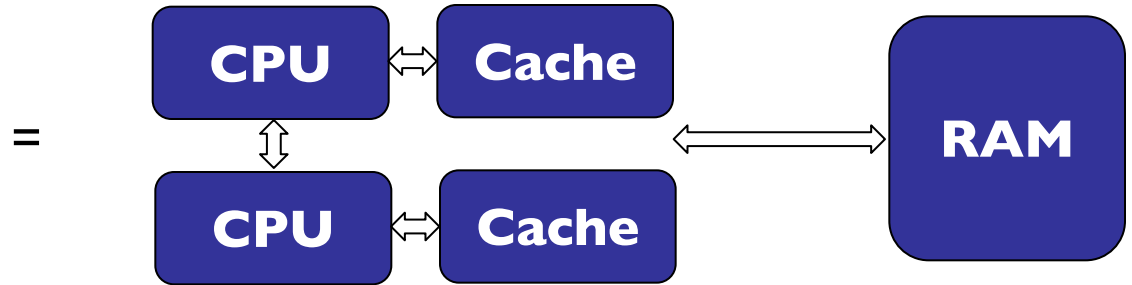
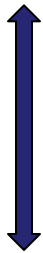
## Computer Architectures

- parallel machine



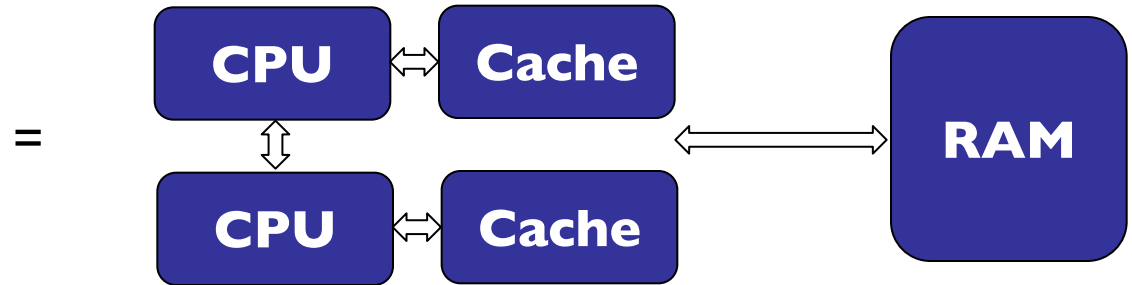
## Computer Architectures

- 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
- ...



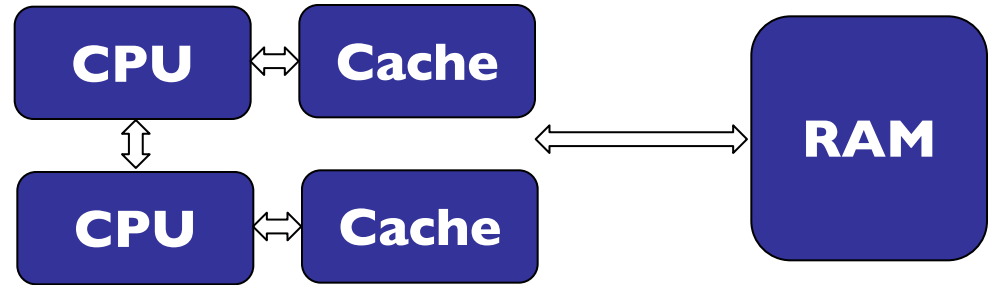


## Computer Architectures

- parallel machine



=

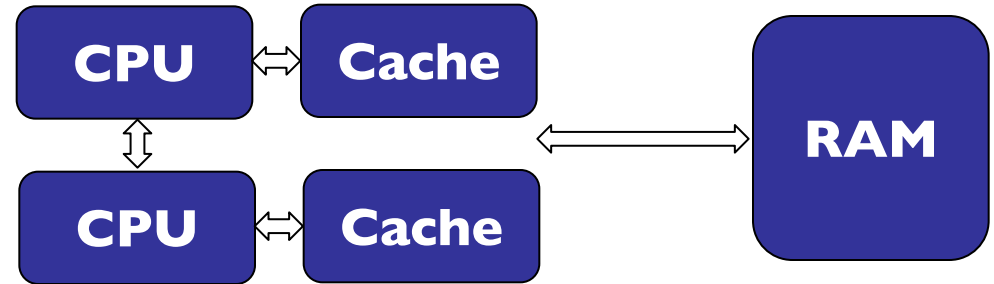


- distributed memory architecture:

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



=

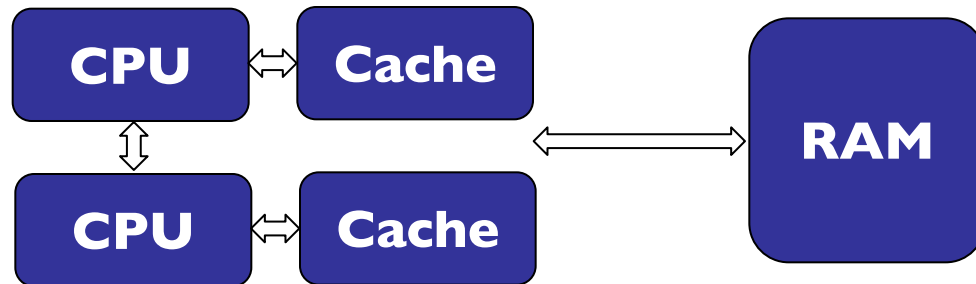


## Computer Architectures

- parallel machine



=

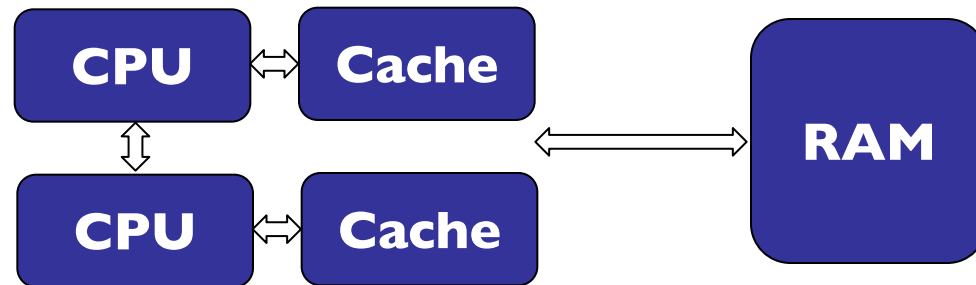


- distributed memory architecture:

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



=



- MPI – Message Passing Interface

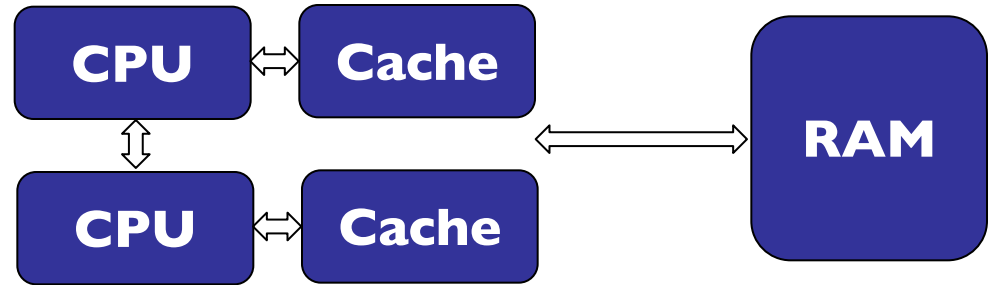
- “standard” library for work dispersal on distributed memory architectures
- e.g., [www.open-mpi.org](http://www.open-mpi.org)

Computer Architectures

- parallel machine



=



- distributed memory architecture:

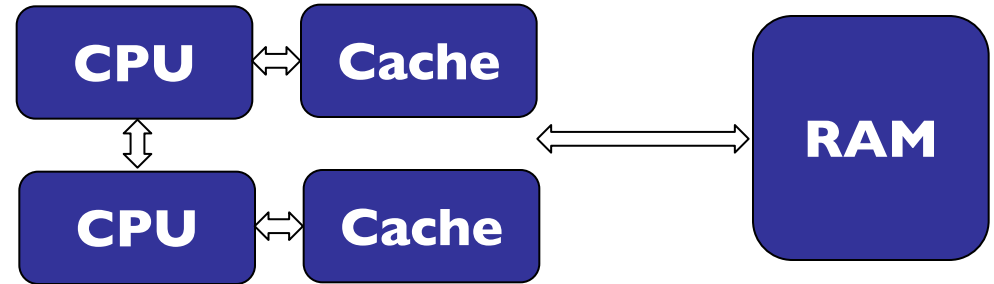
- MPI – Message Passing Interface

- existing code difficult to port to distributed memory architectures
- easy to write code on distributed memory architectures
- speed-up limited by inter-computer communication
- e.g., [www.open-mpi.org](http://www.open-mpi.org)

**you(!) have to substantially restructure your code**



=



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

## Computer Architectures

- parallel machine in reality = multi-level hybrid machines

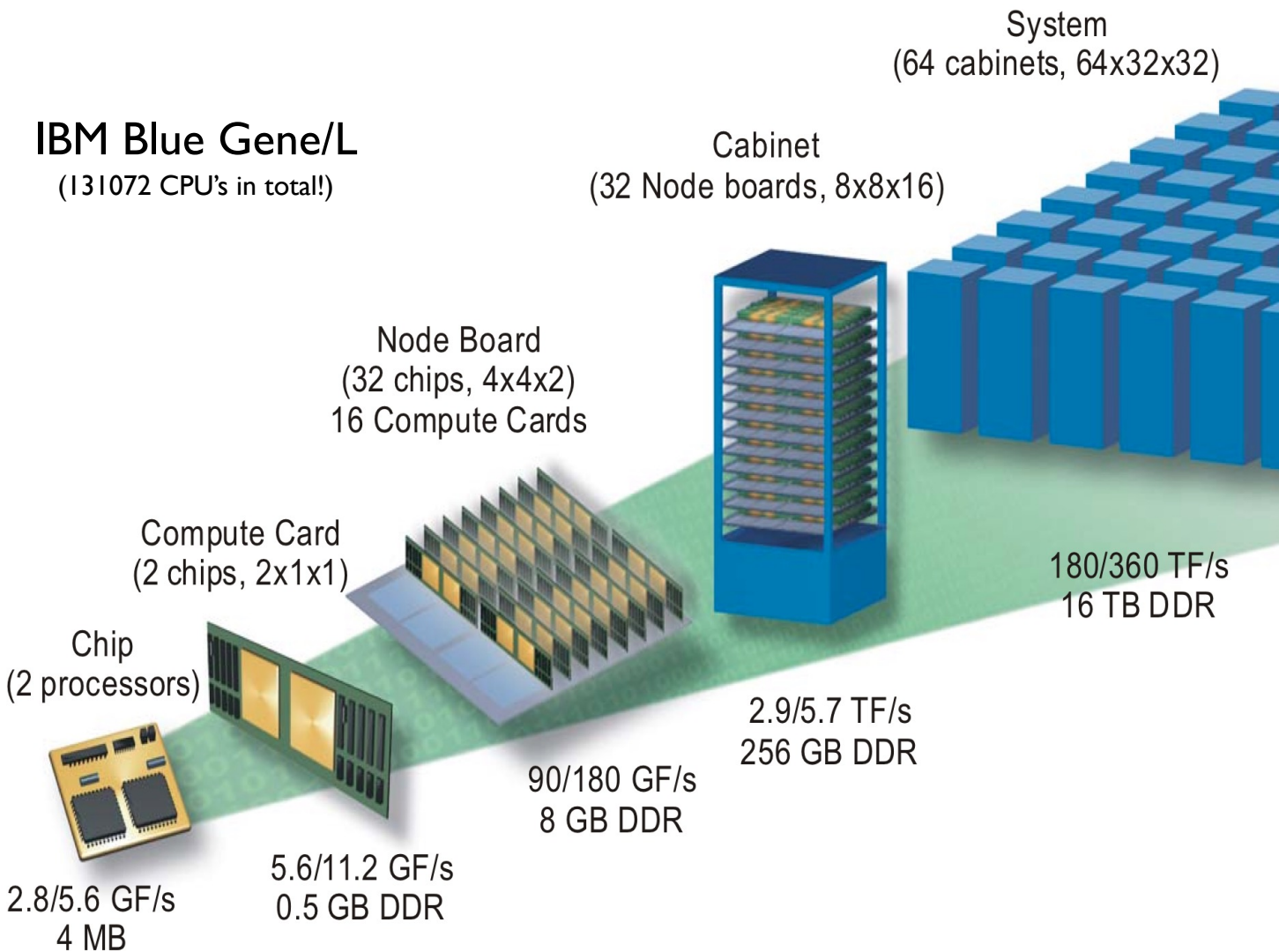
### IBM Blue Gene/L

(131072 CPU's in total!)



## Computer Architectures

- parallel machine in reality = multi-level hybrid machines



- www.top500.org

The screenshot shows the homepage of the TOP500 Supercomputing Sites. The browser address bar displays <http://www.top500.org/>. The page features a navigation menu with links for PROJECT, LISTS, STATISTICS, RESOURCES, NEWS, CONTACT, SUBMISSIONS, LINKS, and HOME. A prominent banner for ISCEvents cloud computing is displayed, advertising an event in Mannheim, Germany, from September 26-27, 2011.

**TOP 10 Systems - 06/2011**

1	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect
2	Tianhe-1A - NUDT TH MPP, X5670 2.93Ghz 6C, NVIDIA GPU, FT-1000 8C
3	Jaguar - Cray XT5-HE Opteron 6-core 2.6 GHz
4	Nebulae - Dawning TC3600 Blade, Intel X5650, NVidia Tesla C2050 GPU
5	TSUBAME 2.0 - HP ProLiant SL390s G7 Xeon 6C X5670, Nvidia GPU, Linux/Windows
6	Cielo - Cray XE6 8-core 2.4 GHz
7	Pleiades - SGI Altix ICE 8200EX/8400EX, Xeon HT QC 3.0/Xeon 5570/5670 2.93 Ghz, Infiniband
8	Uttarakhand - Cray XE6 10-core 2.4 GHz
9	Uttarakhand - Cray XE6 10-core 2.4 GHz
10	Uttarakhand - Cray XE6 10-core 2.4 GHz

**Japan Reclaims Top Ranking on Latest TOP500 List of World's Supercomputers**  
Thu, 2011-06-16 19:24

HAMBURG, Germany—A Japanese supercomputer capable of performing more than 8 quadrillion calculations per second (petaflop/s) is the new number one system in the world, putting Japan back in the top spot for the first time since the Earth Simulator was dethroned in November 2004, according to the latest edition of the TOP500 List of the world's top supercomputers. The system, called the K Computer, is at the RIKEN Advanced Institute for Computational Science (AICS) in Kobe.

[» Read more](#)

Advertisements for SUPERMICRO (We Keep IT Green™), AFPR (Delivering Improved, Reliability, Availability, Serviceability (RAS)), and HP (Deep Computing?) are visible on the right side of the page.

## Computer Architectures

- [www.top500.org](http://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](http://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](http://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

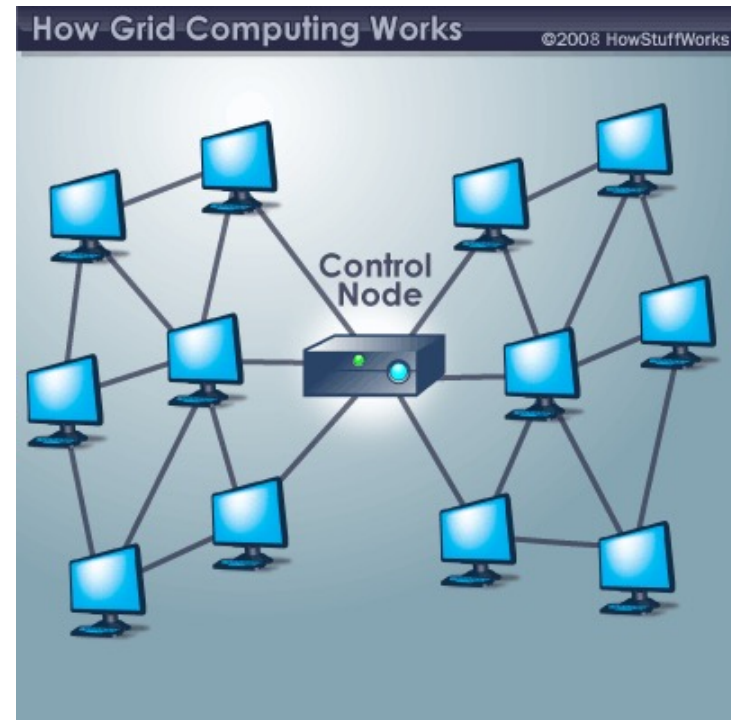
- architectures
- real machines
- **computing concepts**
- parallel programming

## Computer Architectures

- GRID computing / Cloud computing?

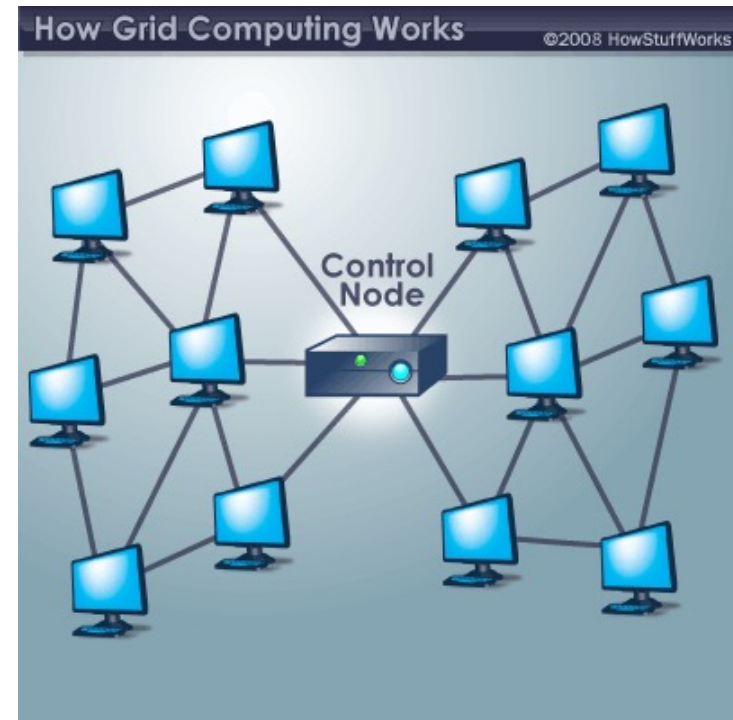
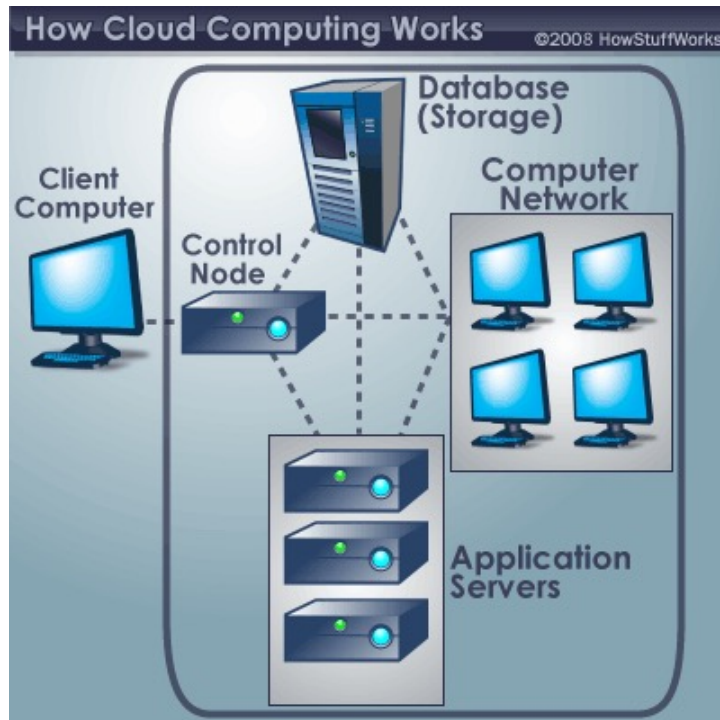
## Computer Architectures

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



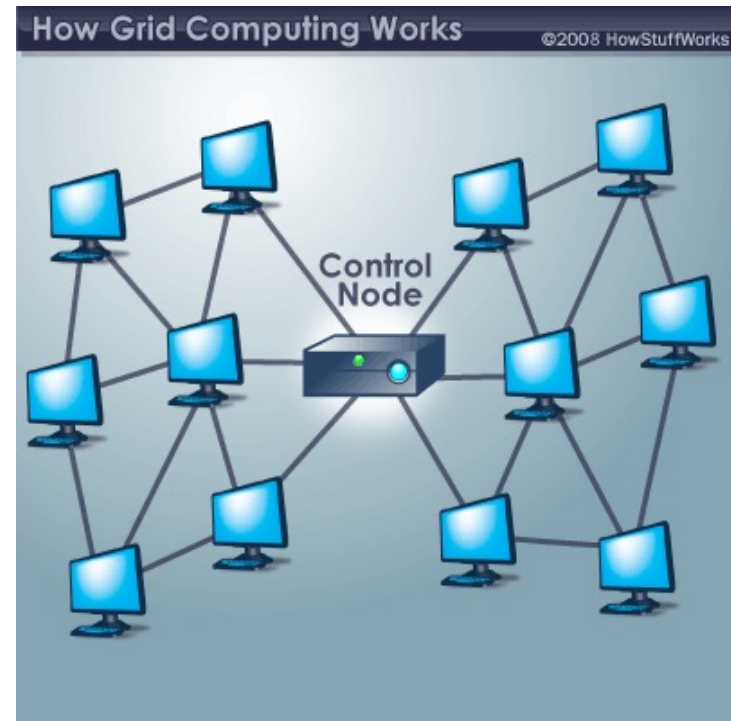
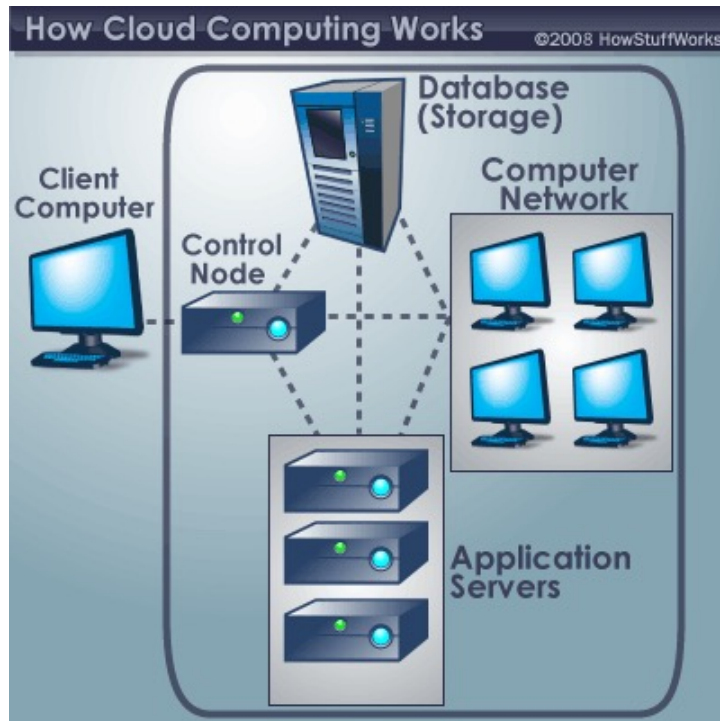
## Computer Architectures

- **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, ...)

SETI@home

What is SETI@home?  
SETI@home is a scientific experiment that uses Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI). You can participate by running a free program that downloads and analyzes radio telescope data.

**PARTICIPATE**

- Download
- Get help
- Tell a friend
- Donate
- Porting & optimization
- ... more

**ABOUT**

- About SETI@home
- About Astropulse
- Science newsletters
- Technical news
- Server status
- Science status
- Sponsors
- ... more

**COMMUNITY**

- Message boards
- Questions & answers
- Profiles
- User search
- Teams
- Web sites & IRC
- Pictures & music

**YOUR ACCOUNT**

- Your account
- Preferences
- Certificate

**STATISTICS**

- Top participants
- Top computers
- Top teams
- Top GPU models

Site search:

Get started

- 1 [Read our rules and policies](#)
- 2 **Download, install and run** the BOINC software used by SETI@home. When prompted, enter the URL: <http://setiathome.berkeley.edu>

Have questions or need help? Contact a volunteer using [BOINC online help](#).

Special instructions:

- [For SETI@home Classic participants](#)
- [For users of command-line and pre-5.0 clients](#)

News

**Another way to support SETI@home**  
In addition to crunching, you can provide some support to SETI@home by using [GoodSearch](#) and [GoodShop](#). These search engines redirect a half their advertising to revenues to charity. Just be sure to choose "University of California - SETI@home" as your charity of choice. 12 Sep 2011 | 20:38:21 UTC · [Comment](#)

**more data on the way**  
In an attempt to push some older unanalyzed files through the pipeline we encountered data that could not be successfully preprocessed or split. This has resulted in work distribution going to near zero. We are currently transferring newer files from off site storage and soon will be receiving a disk shipment from Arecibo. Once these data are on hand work distribution will pick up. 8 Sep 2011 | 16:56:49 UTC · [Comment](#)

**storage service is back up**  
We have migrated storage service to thumper. Now it's a matter of transferring raw data to thumper, preprocessing it, and splitting it. This will all take some time. The Overland server is up and its raid is operational (thanks Overland!). Diagnosis of this unit is proceeding. 28 May 2011 | 14:30:01 UTC · [Comment](#)

POWERED BY Keep your computer busy when SETI@home has no work - [participate in other BOINC-based](#)

## 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, ...)

The screenshot shows the SETI@home website interface. A text box is overlaid on the page, containing the following text:

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

The background website content includes a navigation menu with categories like PARTICIPATE, ABOUT, COMMUNITY, YOUR ACCOUNT, and STATISTICS. There are also sections for 'Get started' with numbered steps, 'Special instructions' for classic and command-line participants, and a 'storage service is back up' announcement.

## Computer Architectures

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

▪ C

The image shows a screenshot of the SETI@home website. The browser address bar shows 'setiathome.berkeley.edu'. The website has a dark blue header with the SETI@home logo and navigation menus for Project, Science, Computing, Community, and Site. There are 'Join' and 'Login' buttons. The main content area is divided into several sections. A 'What is SETI@home?' section includes a description and a green 'Join SETI@home' button. A 'News' section, highlighted with a red rounded rectangle, contains an article titled 'SETI@home and COVID-19' with the text: 'SETI@home will stop distributing tasks soon, but we encourage you to continue donate computing power to science research - in particular, research on the COVID-19 virus. The best way to do this is to join Science United and check the "Biology and Medicine" box. 23 Mar 2020, 21:33:54 UTC · Discuss'. Below this is another news article titled 'New SETI Perspectives: "How did life begin on Earth and elsewhere?"' with the text: 'Richard Lawn has posted a new SETI Perspective entitled How did life begin on Earth and elsewhere?. 19 Mar 2020, 22:49:24 UTC · Discuss'. At the bottom, there is a 'User of the Day' section featuring a profile for 'Xcure' and a banner that reads 'We are now on self quarantine and a pandemic is operational (thanks Overhand); Diagnosis of this unit is proceeding.' A footer at the very bottom says '28 May 2011 | 14:30:01 UTC · Comment'.

## 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  $21 \times 10^6$  Pflops\* !!!!!

## 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  $21 \times 10^6$  Pflops:

Bitcoin Network

## Computer Architectures

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

The screenshot shows the website [www.cosmosim.org](http://www.cosmosim.org) in a browser window. The page features a dark blue background with a galaxy simulation pattern. The main heading is "CosmoSim". A navigation bar includes links for "CosmoSim", "Blog", "Documentation", "Database", "Files", "Query", "Contact", and "Login". Below the heading, a text block states: "The CosmoSim database provides results from cosmological simulations performed within different projects: MultiDark and Bolshoi, CLUES, and Galaxies." Three project cards are displayed: "MultiDark Bolshoi" (describing the Spanish MultiDark Consolider project), "Galaxies" (describing the MDPL2 simulation), and "CLUES" (describing the Constrained Local Universe Simulations project). A "Register to CosmoSim" button is visible on the right. The footer includes the URL <https://www.cosmosim.org/#> and logos for AIP and GAVO.

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

## Computer Architectures

- how to program such machines?



## Computer Architectures

- 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

## Computer Architectures

- how to program such machines?

**your algorithm must be parallel,**

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

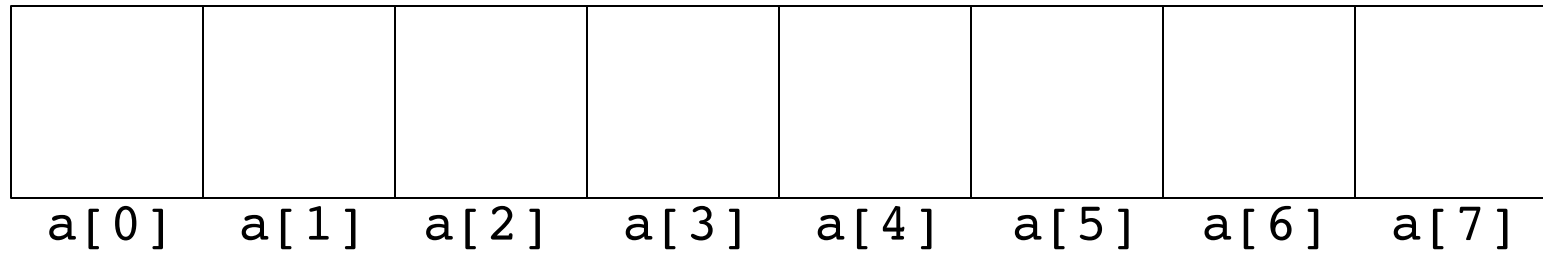
**data parallelisation:**

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

## Computer Architectures

- how to program such machines?

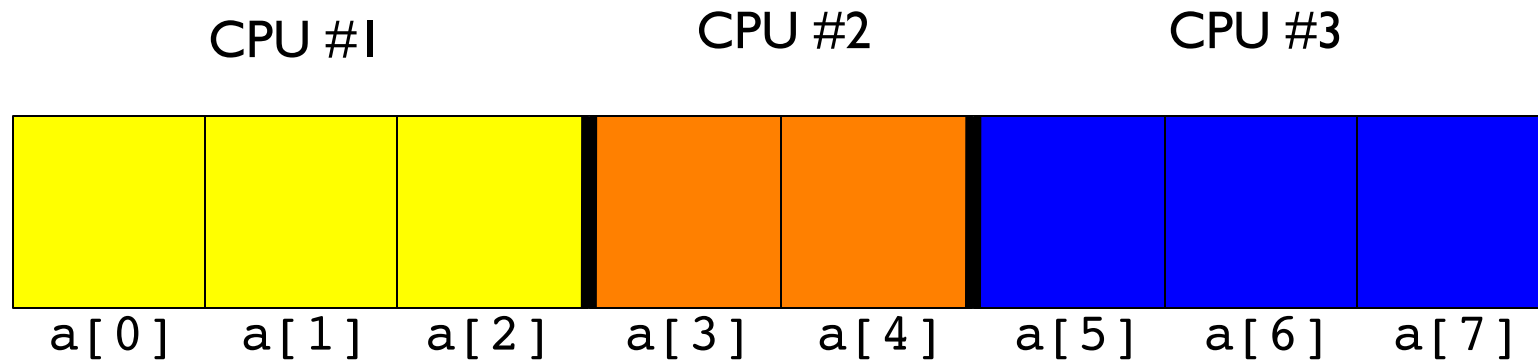
ID array



## Computer Architectures

- how to program such machines?

1D array



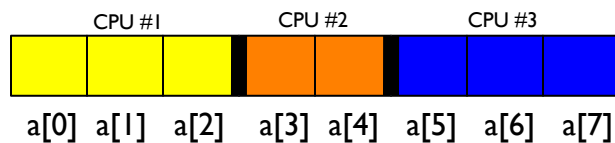
## Computer Architectures

- how to program such machines?

1D array

**serial algorithm**

```
a[0] = STARTVALUE;  
for(i=1; i<N; i++) {  
    a[i] = function(a[i-1]);  
}
```



## Computer Architectures

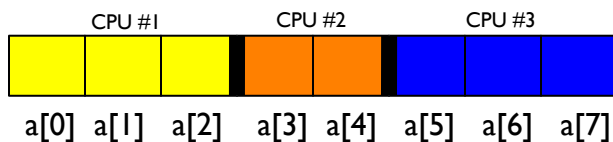
- how to program such machines?

1D array

**serial algorithm**

```
a[0] = STARTVALUE;  
for(i=1; i<N; i++) {  
    a[i] = function(a[i-1]);  
}
```

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



## Computer Architectures

- how to program such machines?

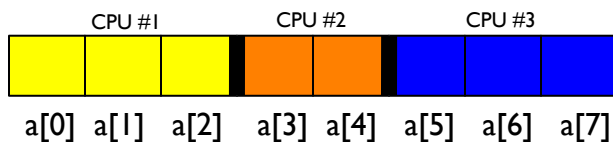
1D array

**serial algorithm**

```
a[0] = STARTVALUE;  
for(i=1; i<N; i++) {  
    a[i] = function(a[i-1]);  
}
```

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

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





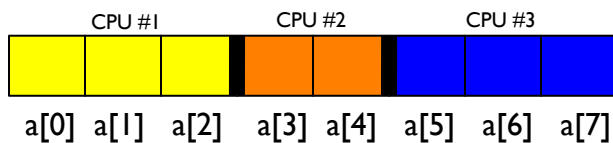
## Computer Architectures

- 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?

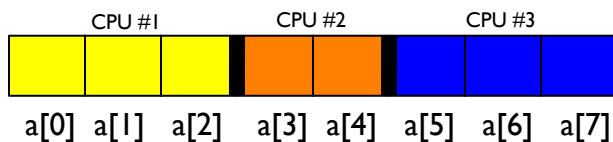
1D array

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?

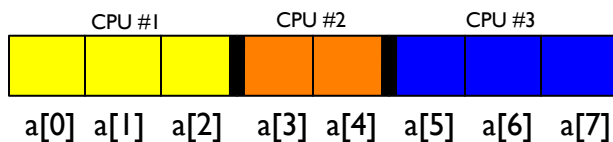
1D array

**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)



- how to program such machines?

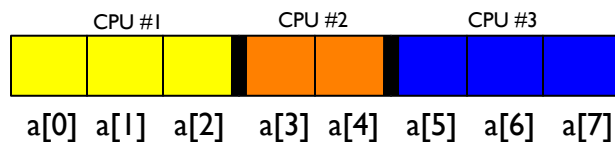
1D array

### 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;
}
```



- how to program such machines?

ID array

## serial algorithm

```

a[0] = STARTVALUE;

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

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

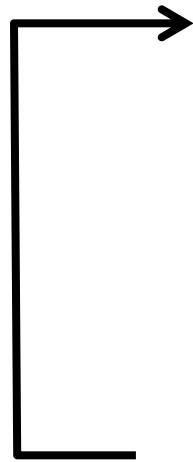
## parallel algorithm

```

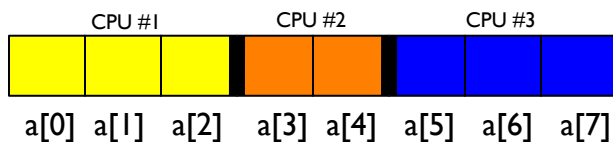
a[0] = STARTVALUE;

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*



- how to program such machines?

ID array

## serial algorithm

```

a[0] = STARTVALUE;

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

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

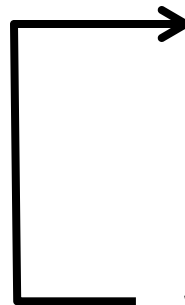
## parallel algorithm

```

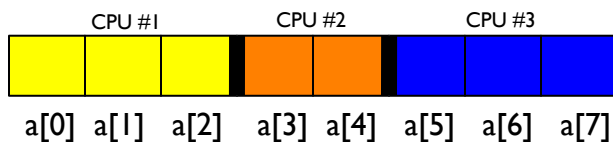
a[0] = STARTVALUE;

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

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



*we eliminated the recursion/dependence  
by expanding it explicitly.  
(by introducing yet another recursion, but c'est la vie...)*



## Computer Architectures

- how to program such machines?

ID array

**serial algorithm**

```

a[0] = STARTVALUE;

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

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

```

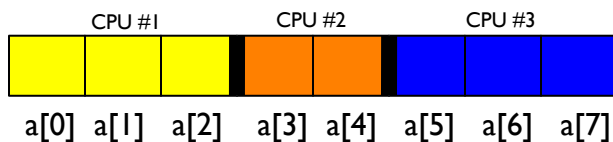
**parallel algorithm**

```

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;
}

```



- how to program such machines?

ID array

### serial algorithm

```

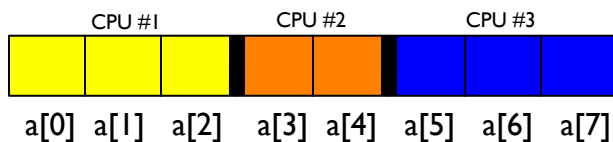
a[0] = STARTVALUE;
for(i=1; i<N; i++) {
    a[i] = function(a[i-1]);
}
    
```

### 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**





## Computer Architectures

- how to program such machines?

ID array

### serial algorithm

```

a[0] = STARTVALUE;
for(i=1; i<N; i++) {

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

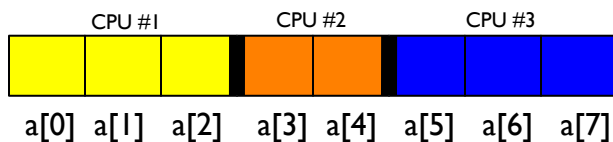
### parallel algorithm

```

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?

ID array

## serial algorithm

```

a[0] = STARTVALUE;

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

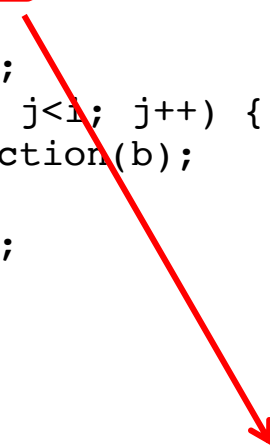
    a[i] = function(a[i-1]);
}
    
```

## parallel algorithm

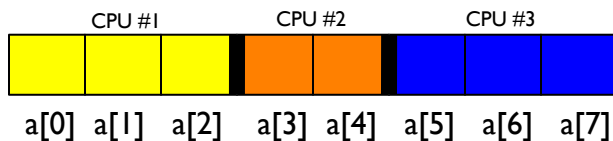
```

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?

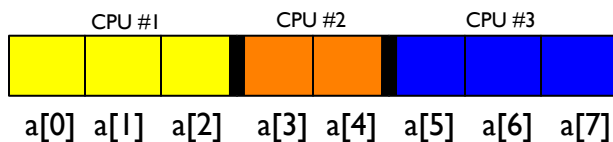
1D array

**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;
}
```

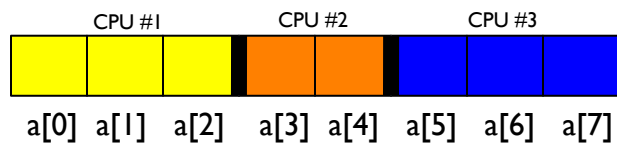


## Computer Architectures

- how to program such machines?

1D array

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



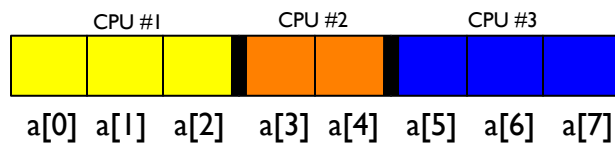
## Computer Architectures

- 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
```



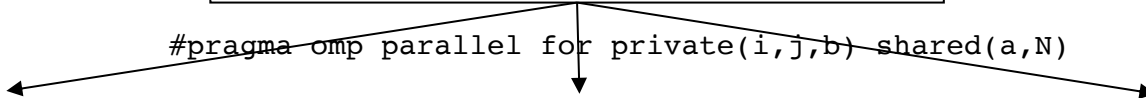
## Computer Architectures

- how to program such machines?

ID 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



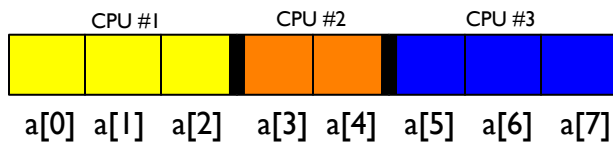
~~#pragma omp parallel for private(i,j,b) shared(a,N)~~

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

    b = sv;

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

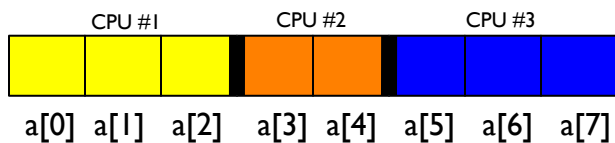
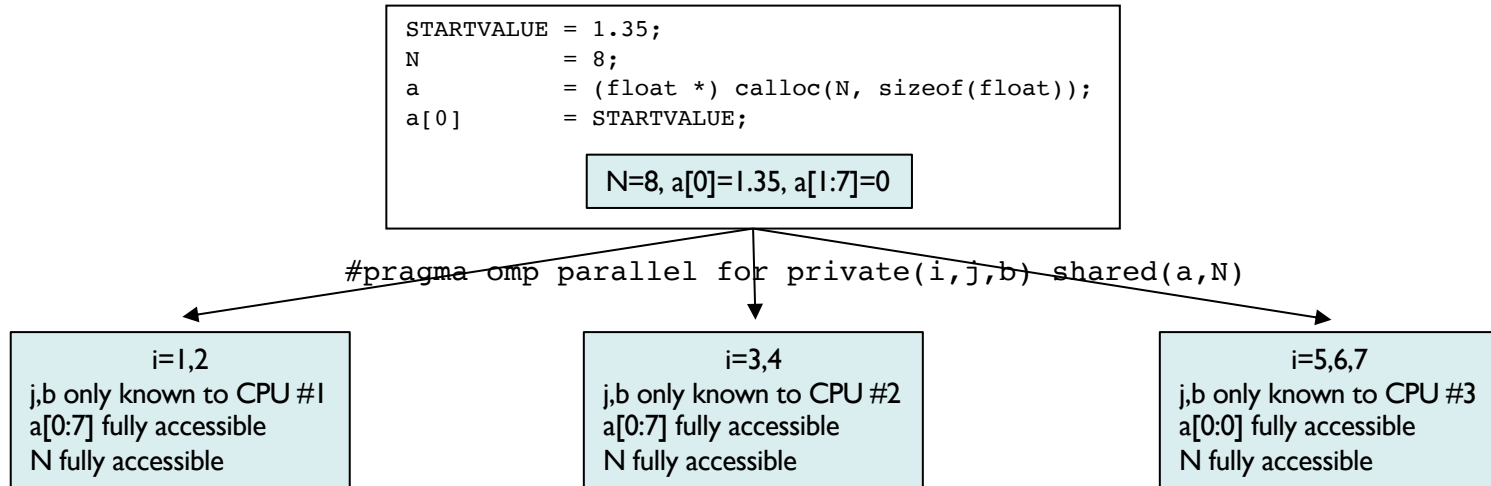
    return(b);
}
```



## Computer Architectures

- how to program such machines?

ID array



▪ how to program such machines?

ID 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;
}
```

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;
}
```

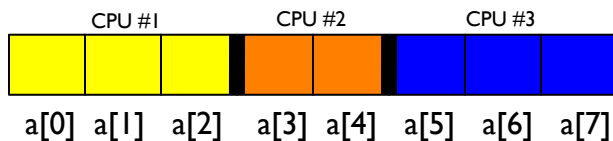
i=5,6,7

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;
}
```

no communication

no communication





## Computer Architectures

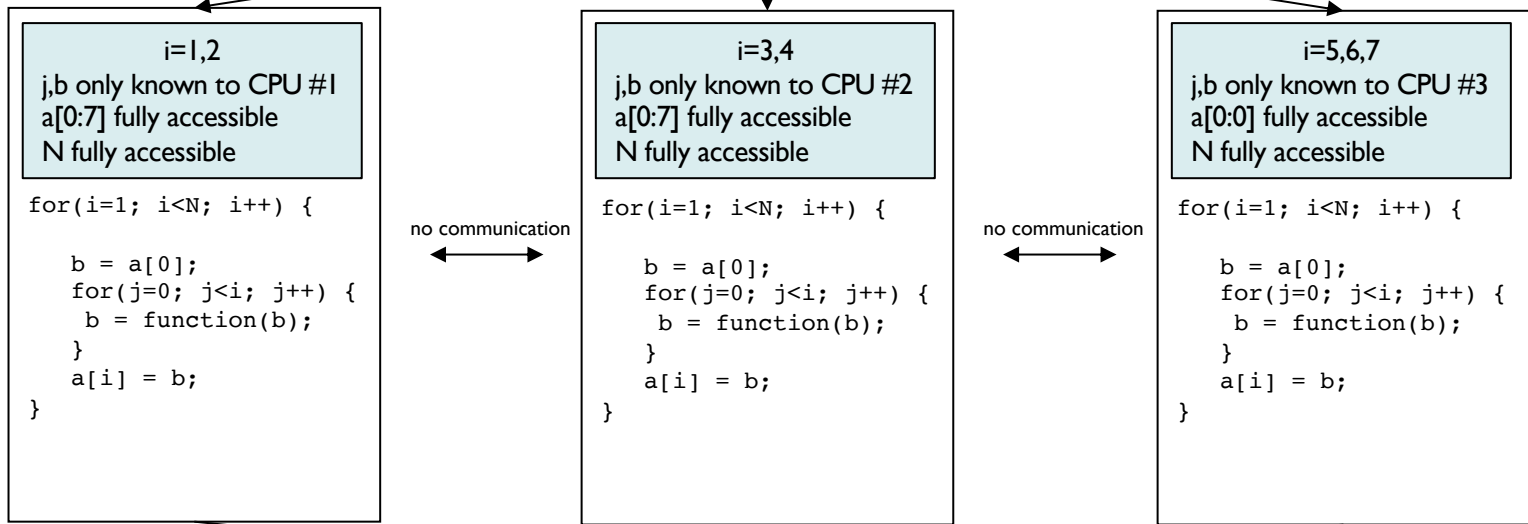
- how to program such machines?

ID array

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

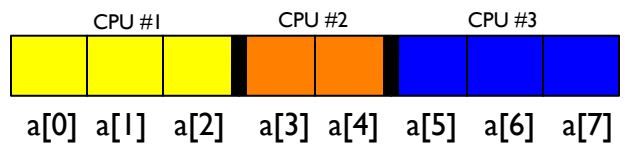
N=8, a[0]=1.35, a[1:7]=0

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



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()
```

## Computer Architectures

- how to program such machines (OpenMP standard):

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



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

## Computer Architectures

- how to program such machines (OpenMP standard):

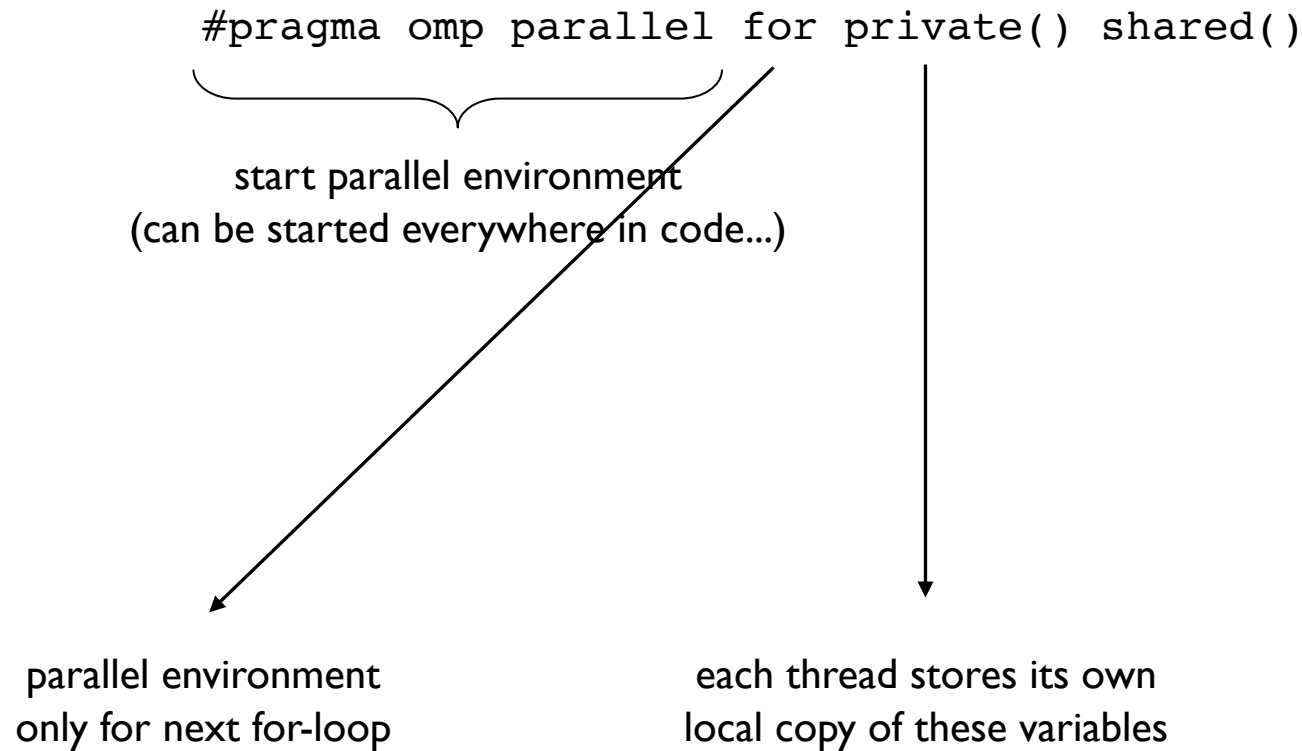
```
#pragma omp parallel for private() shared()
```

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

parallel environment  
only for next for-loop

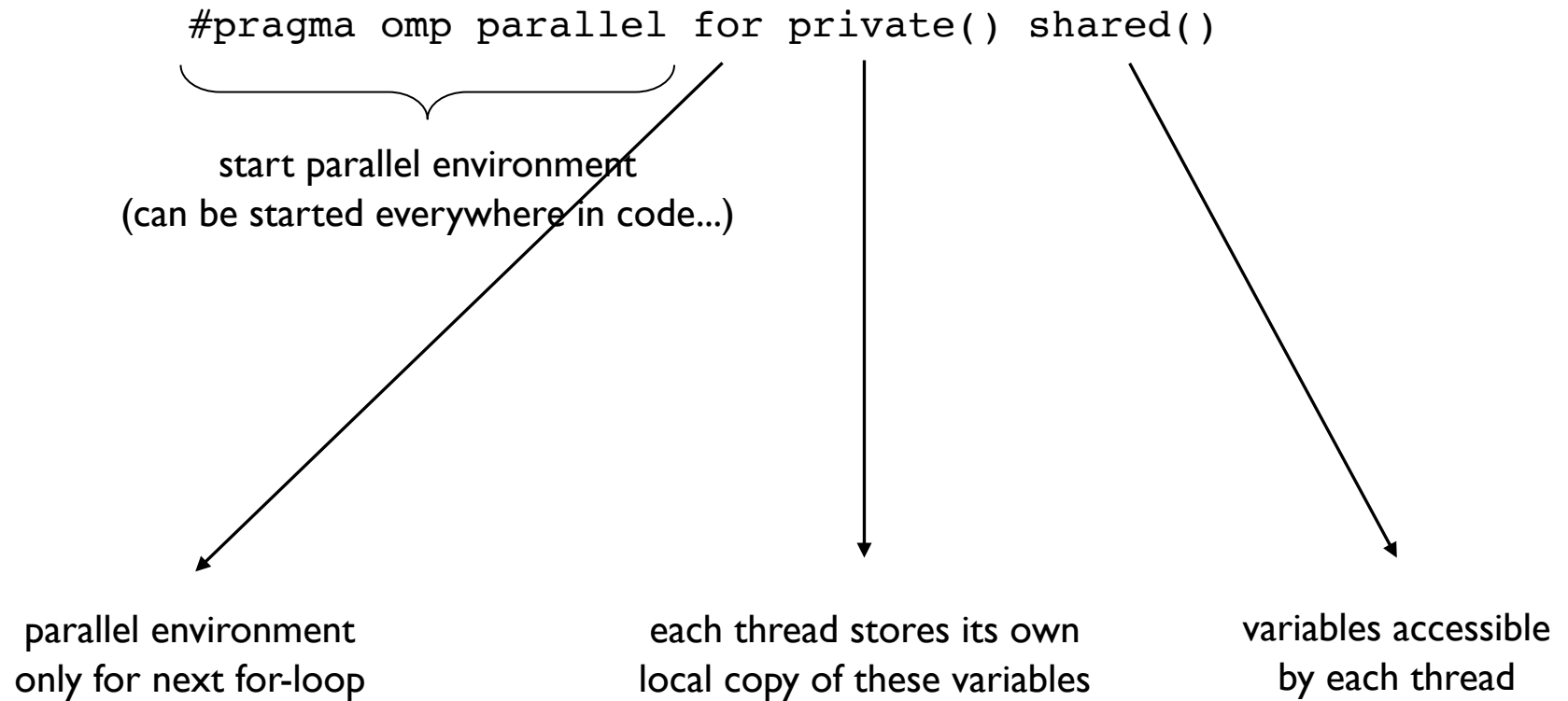
## Computer Architectures

- how to program such machines (OpenMP standard):



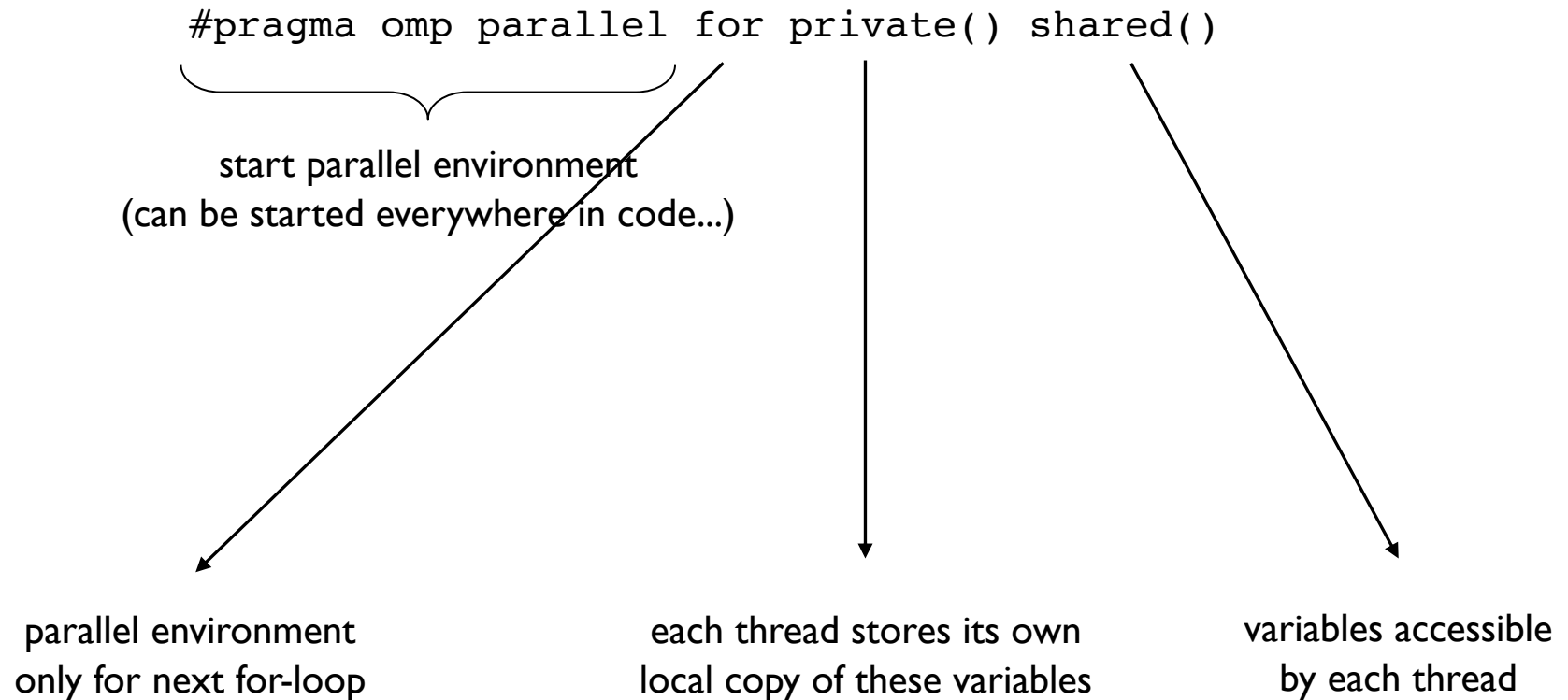
## Computer Architectures

- how to program such machines (OpenMP standard):



## Computer Architectures

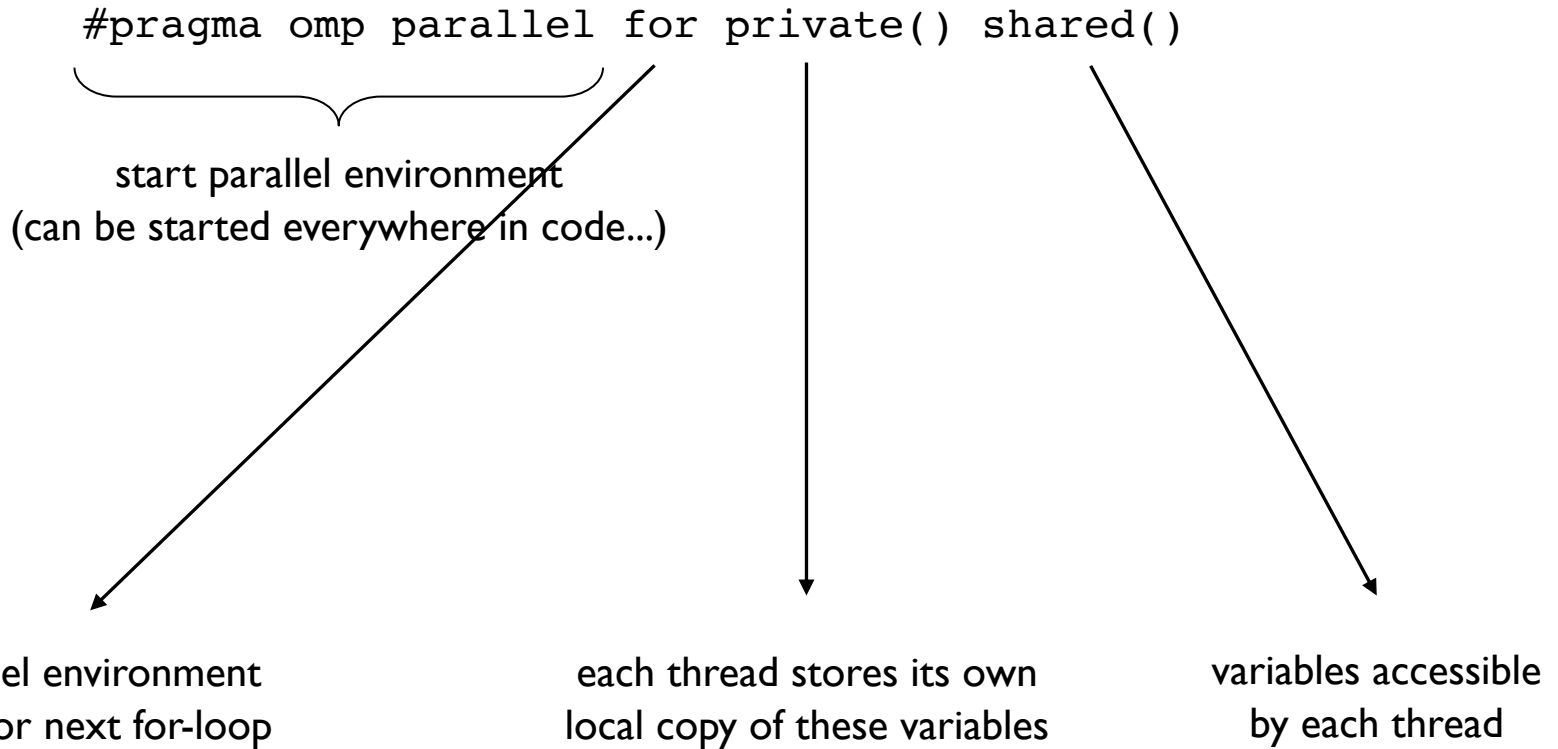
- how to program such machines (OpenMP standard):

**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!

## 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;
```

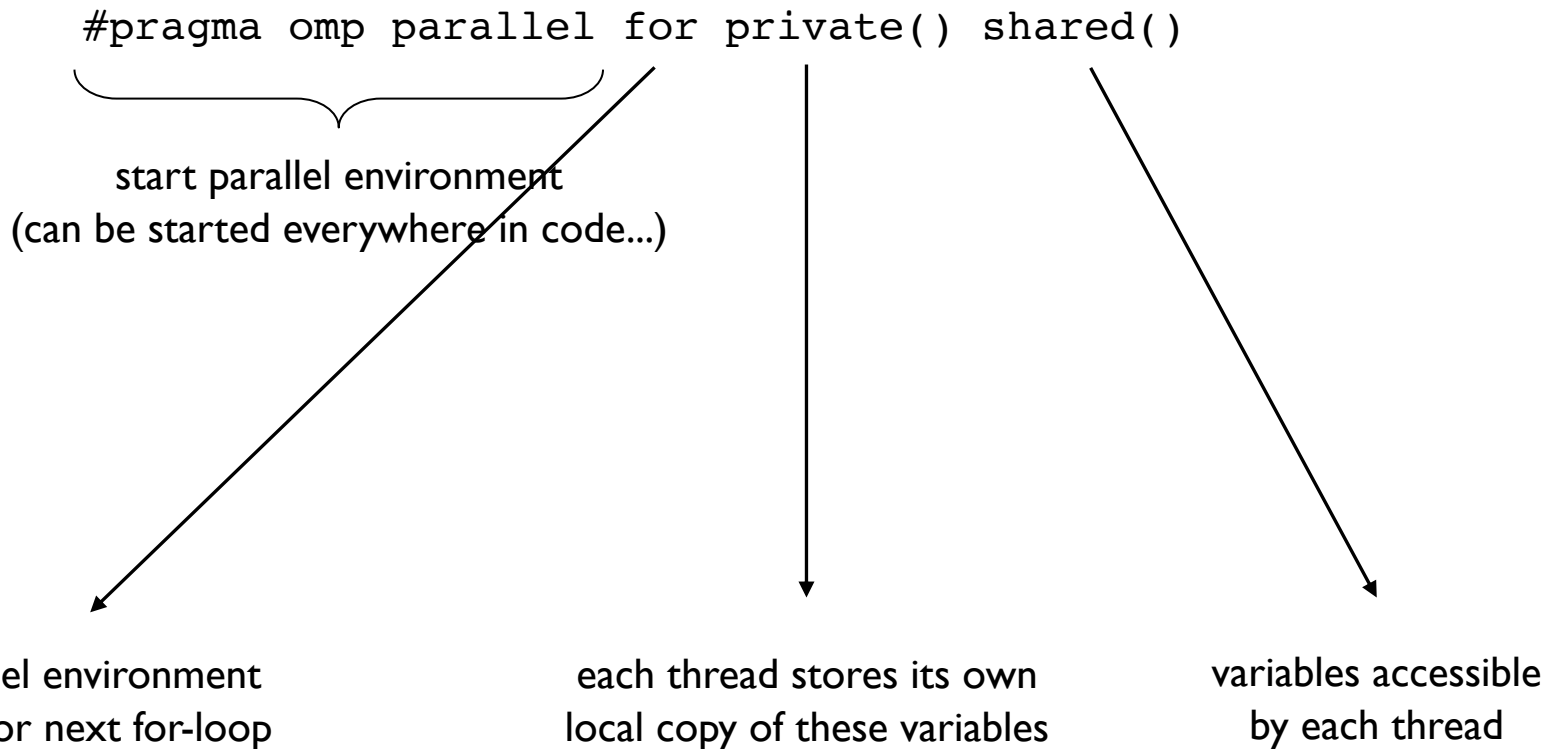
### Note:

- 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;
```

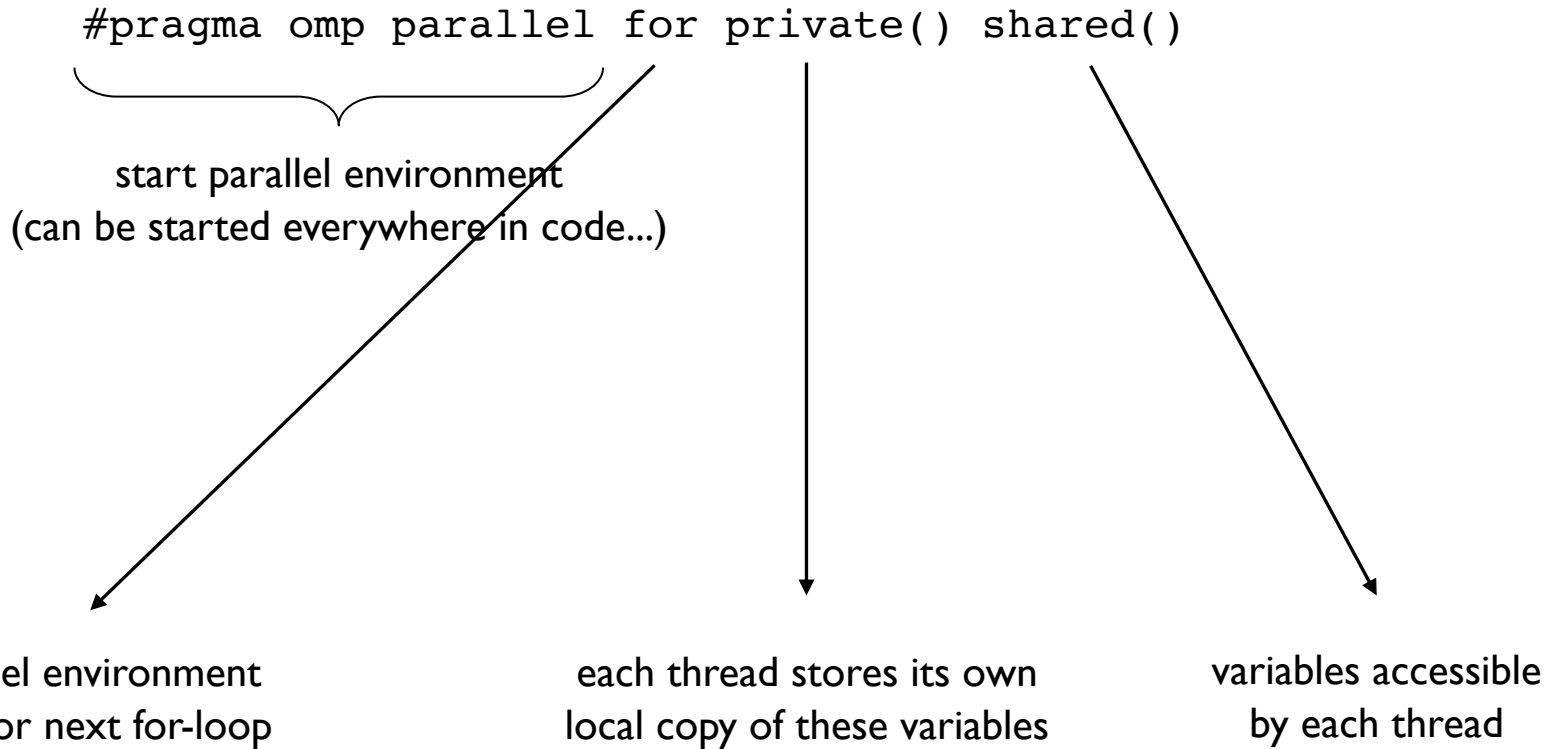
**j?**

### 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

## 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 = b + a[j];
    }
    a[i] = b;
}
```

**Note:**

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

- 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):

```
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 (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);  
    }
```

*put this into a function!*

```
    a[i] = b;  
}
```

## Computer Architectures

- 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);
}
```

*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);
}
```

```
  a[i] = b;
}
```

## Computer Architectures

- 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);
}
```

## Computer Architectures

- how to program such machines (OpenMP standard):

```
a[0] = STARTVALUE;
```

```
#pragma omp parallel for private(i,i,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);  
}
```

## Computer Architectures

- 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);
}
```



## Computer Architectures

- 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);
}
```

## Computer Architectures

- 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);
}
```

## Computer Architectures

- 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);
}
```

## Computer Architectures

- 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]);
}
```

**general advise:**

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

```
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 (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;

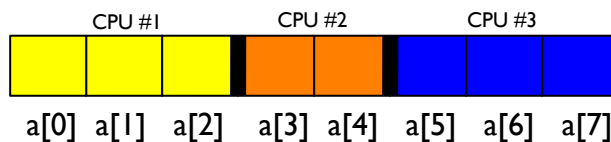
    for(j=0; j<i; j++) {
        b = function(b);
    }

    return(b);
}
```

### general advise:

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

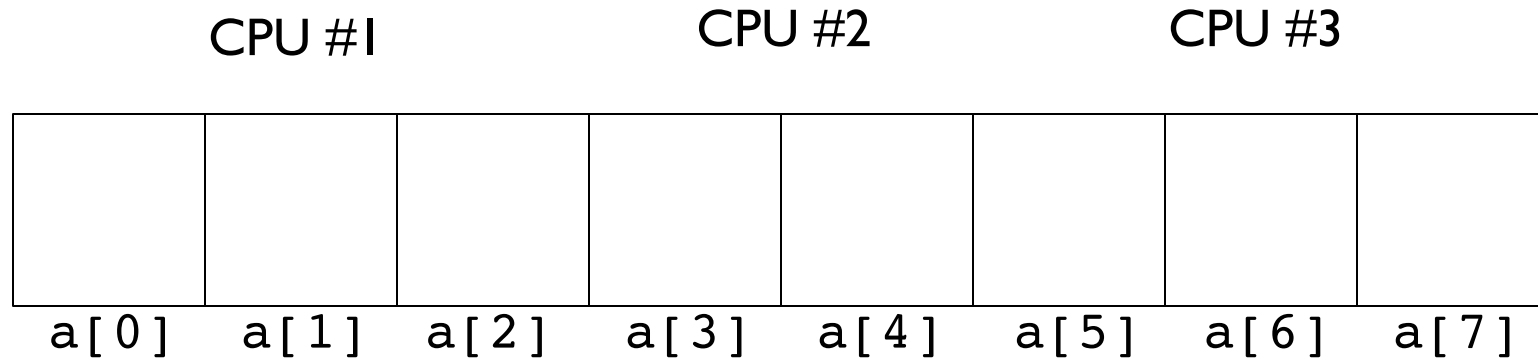
...but which CPU gets what  $i$  values?



## Computer Architectures

- how to program such machines?

ID array

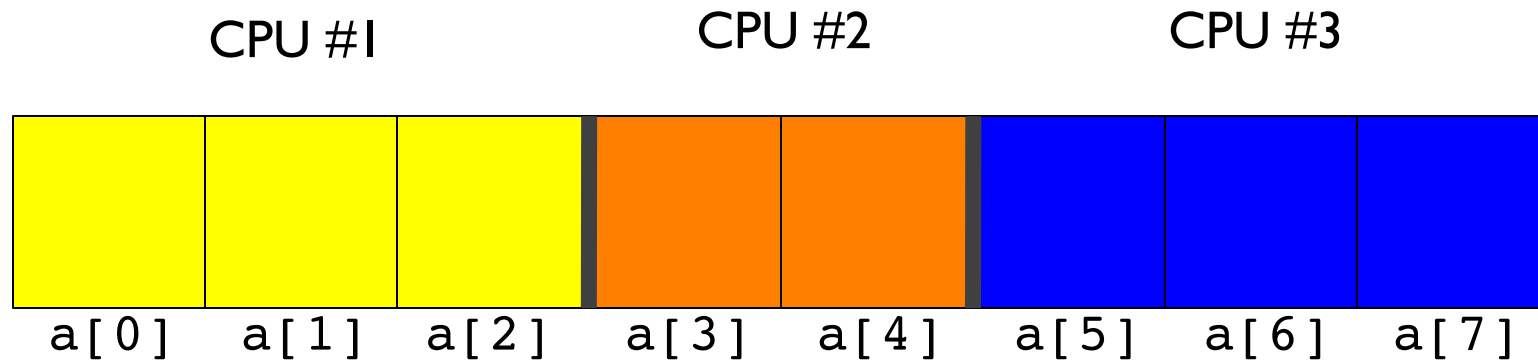


but how to divide the domain?

## Computer Architectures

- how to program such machines?

1D array

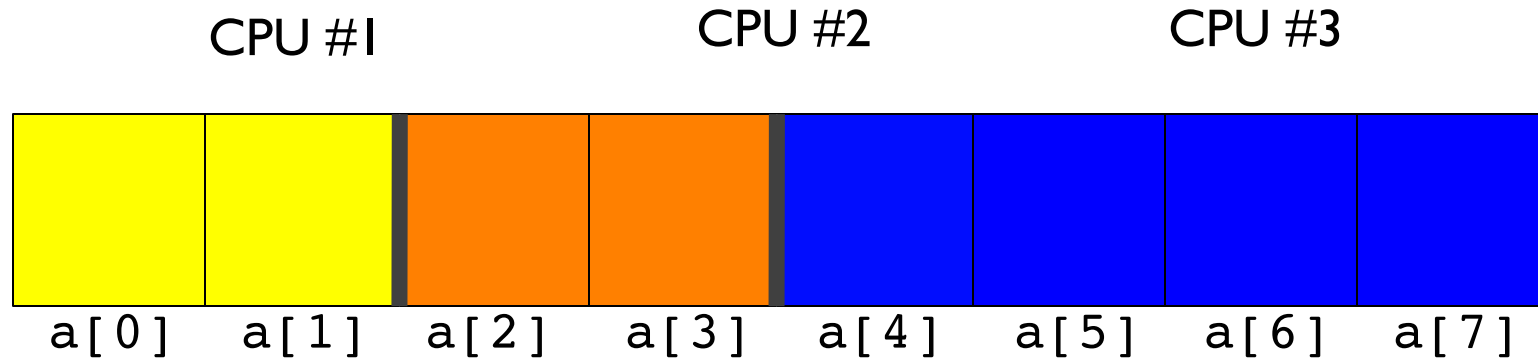


but how to divide the domain?

## Computer Architectures

- how to program such machines?

1D array



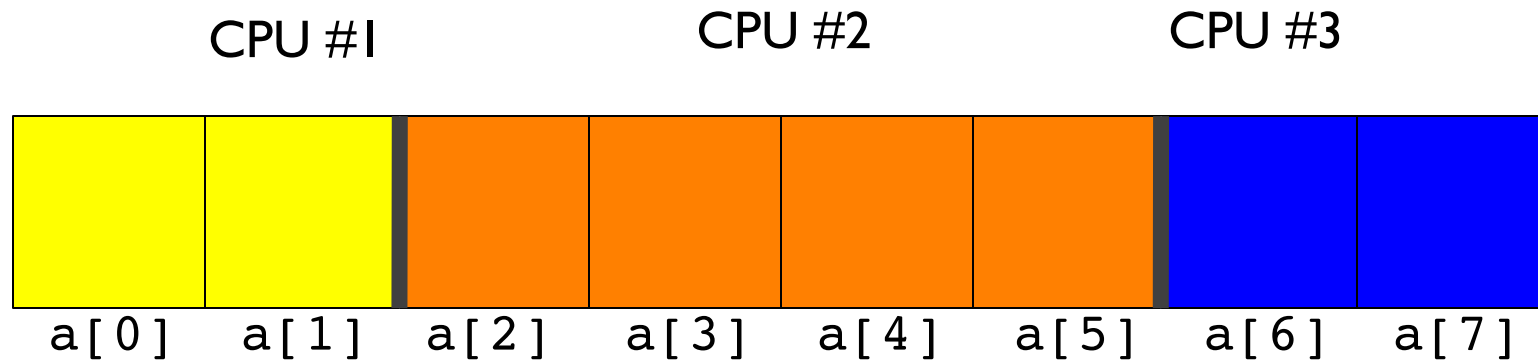
but how to divide the domain?



## Computer Architectures

- how to program such machines?

1D array

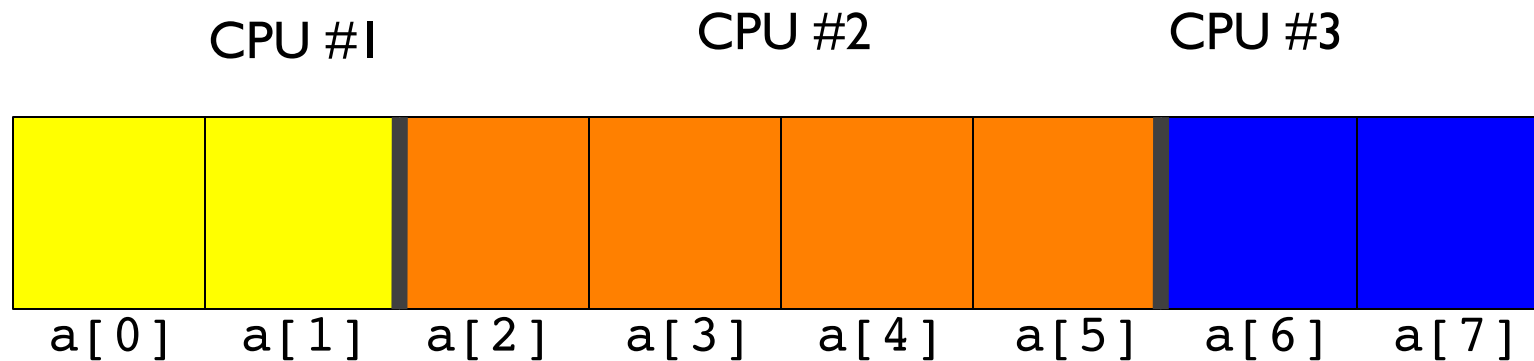


but how to divide the domain?

## Computer Architectures

- how to program such machines?

1D array

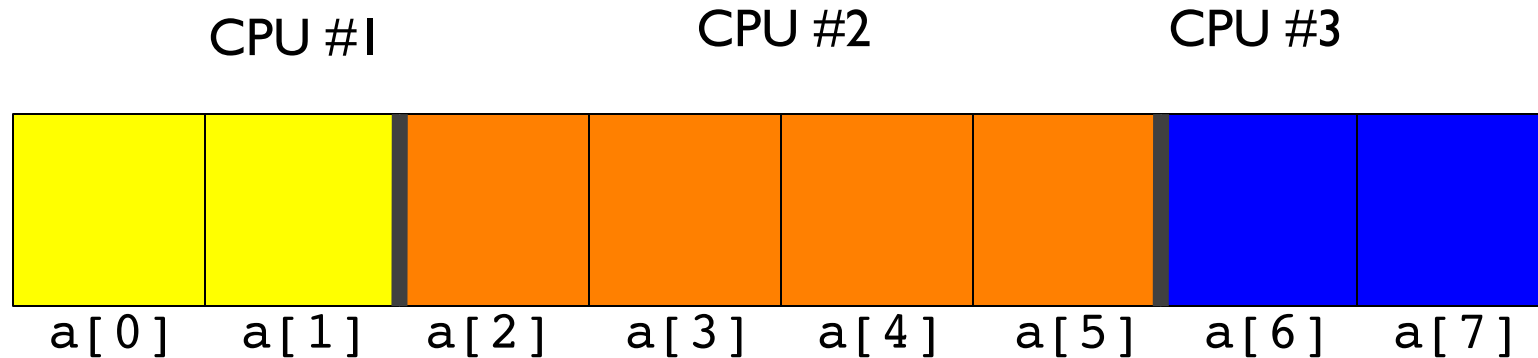


but how to divide the domain:  
distribute the **work** evenly!

## Computer Architectures

- 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?

ID array

**OpenMP work distribution:**

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

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

**usage:**

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

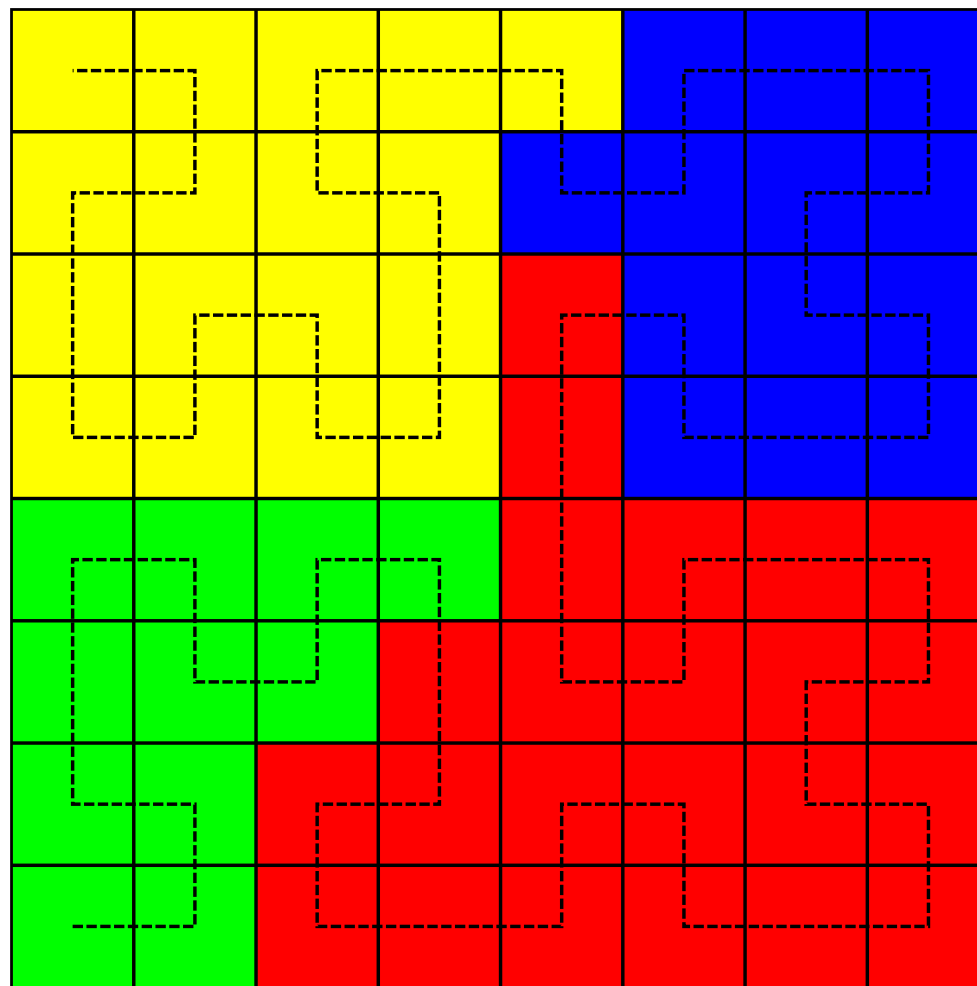
```
b = a[0];  
for(j=0; j<i; j++) {  
    b = function(b);  
}
```

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

## Computer Architectures

- how to program such machines?

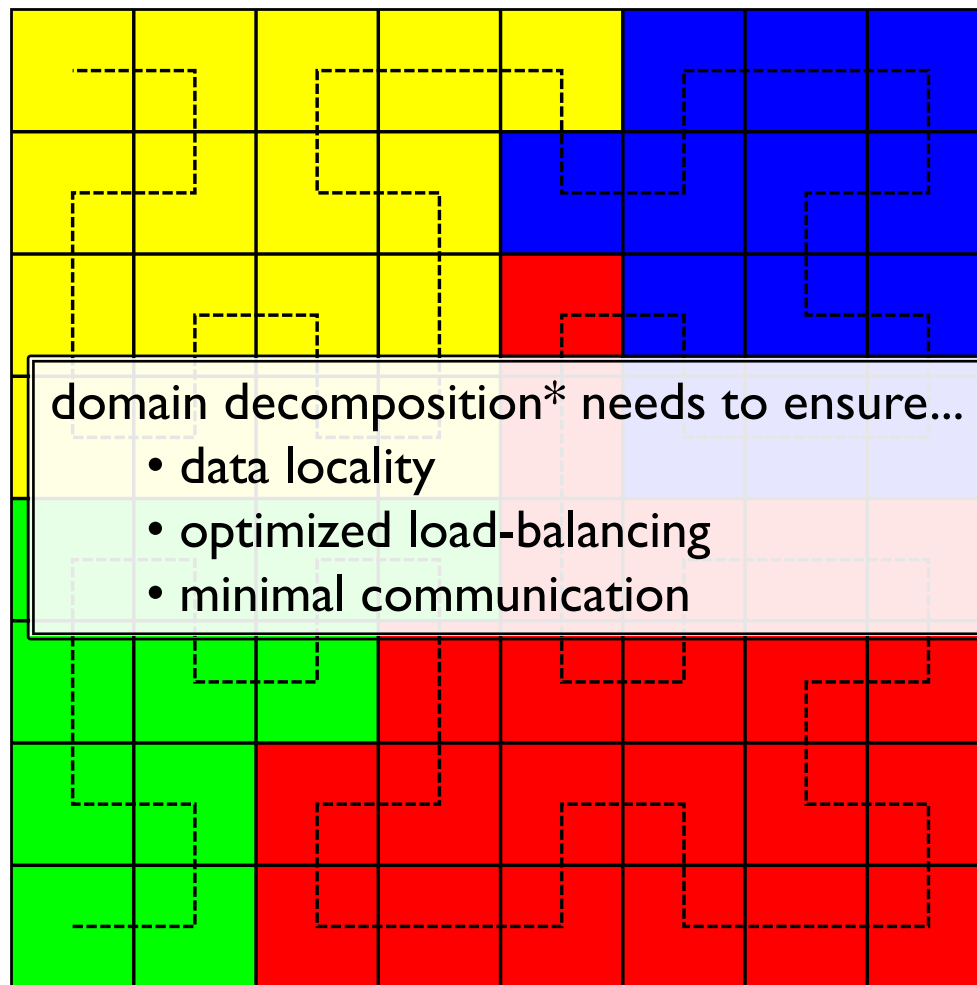
2D array



## Computer Architectures

- how to program such machines?

2D array

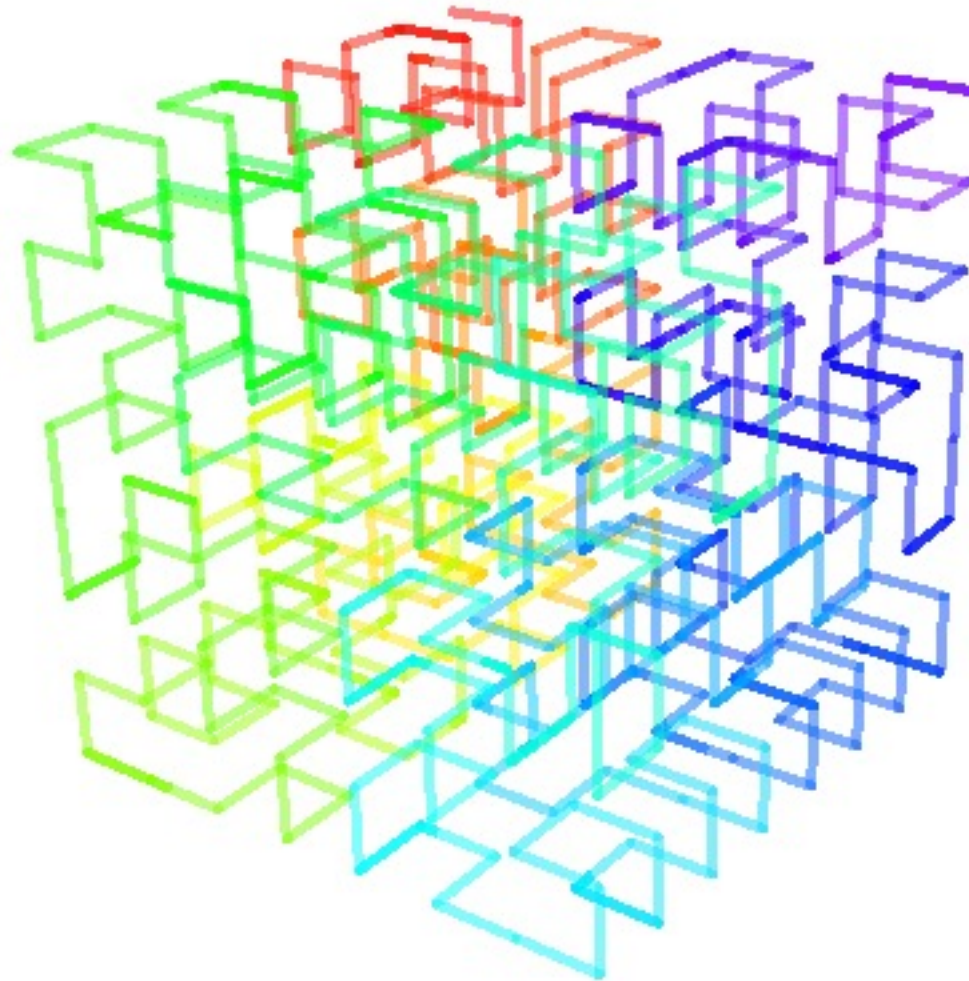


\*this is not to be confused with domain discretisation!

## 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**

## Computer Architectures

- 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

## Computer Architectures

- 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?

## Computer Architectures

- 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

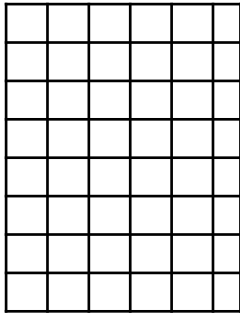
## Computer Architectures

- 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?

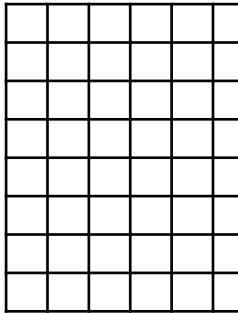




## Computer Architectures

- shaping relevant tasks – data

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

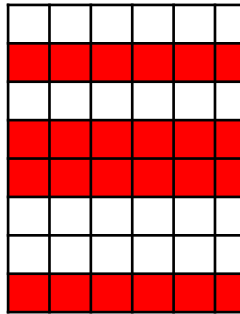




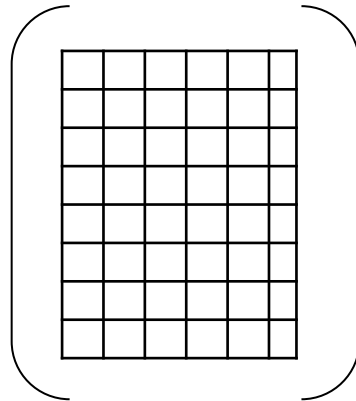
## Computer Architectures

- shaping relevant tasks – data

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



Analyse(Data)

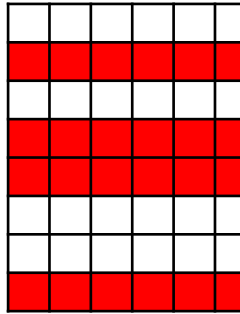


```
void Analyse(Data) {
  for(Data) {
    if(redData){
      Calculation(Data)
    }
    else {
      Calculation(Data)
    }
  }
}
```

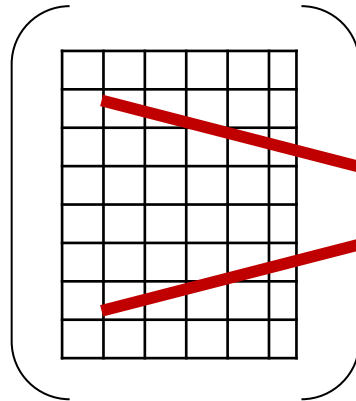
## Computer Architectures

- shaping relevant tasks – data

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



Analyse(Data)



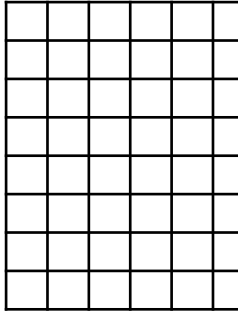
```

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

## Computer Architectures

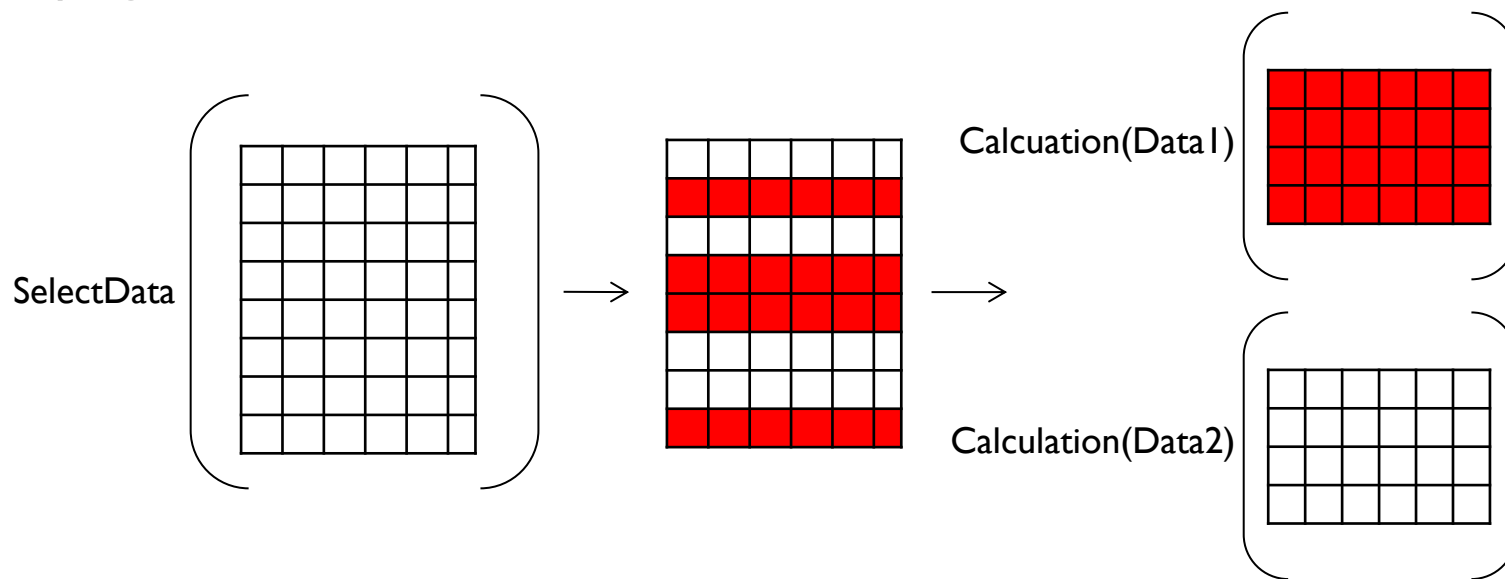
- shaping relevant tasks – data

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



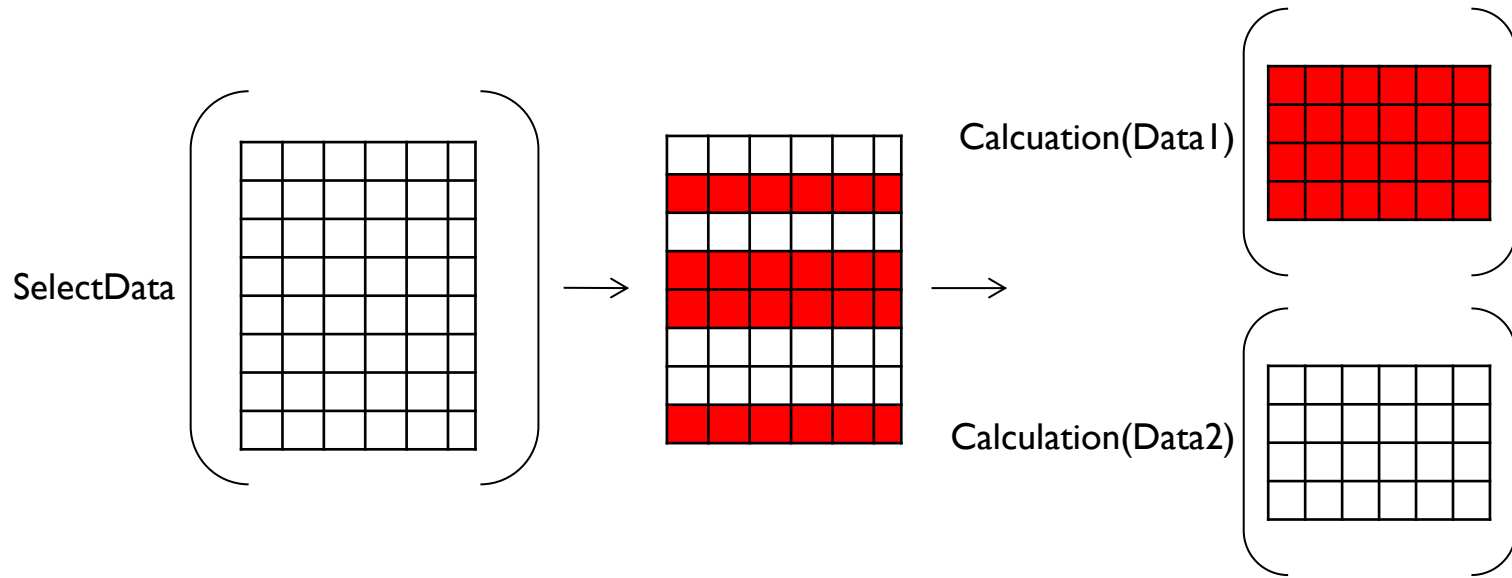
## Computer Architectures

- shaping relevant tasks – data



## Computer Architectures

- shaping relevant tasks – data



**modular and hence more flexible!**

## Computer Architectures

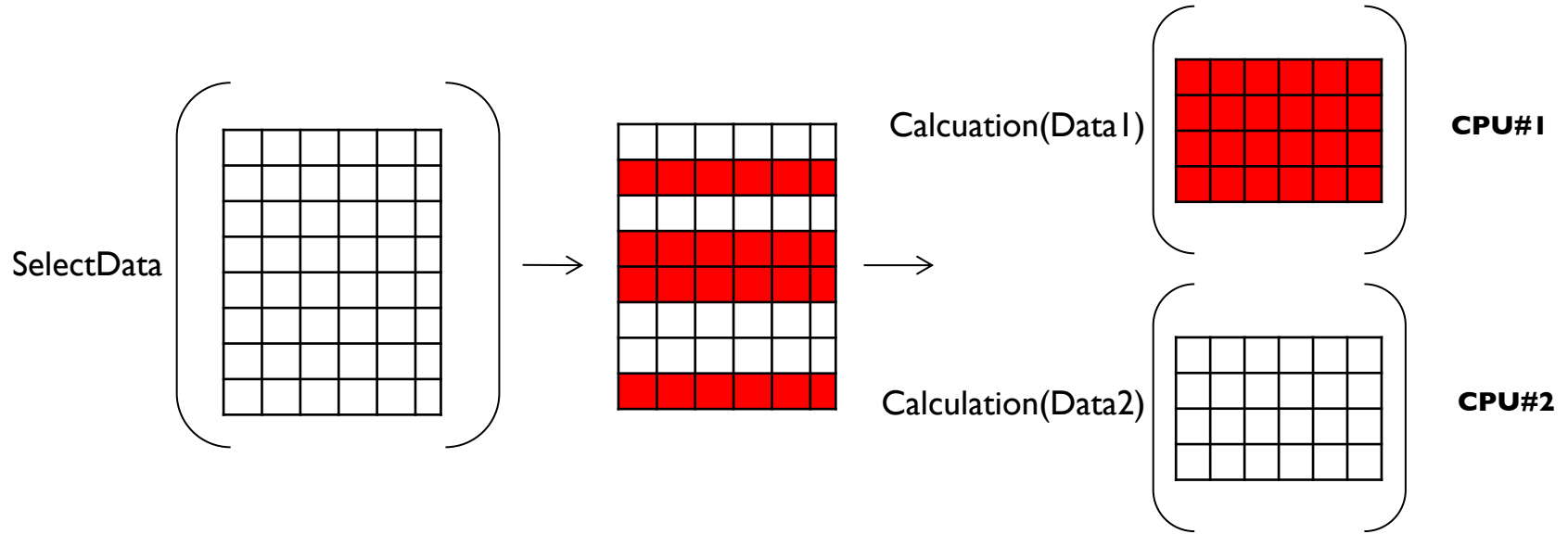
- how to actually write a program?
  - define the problem
  - decide on organisation
    - choose essential elements (variables, structures, etc.)
    - shape relevant tasks
    - **design your algorithm to be parallelizable**
    - draw a flowchart
  - code in your preferred language
  - test code using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

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



## Computer Architectures

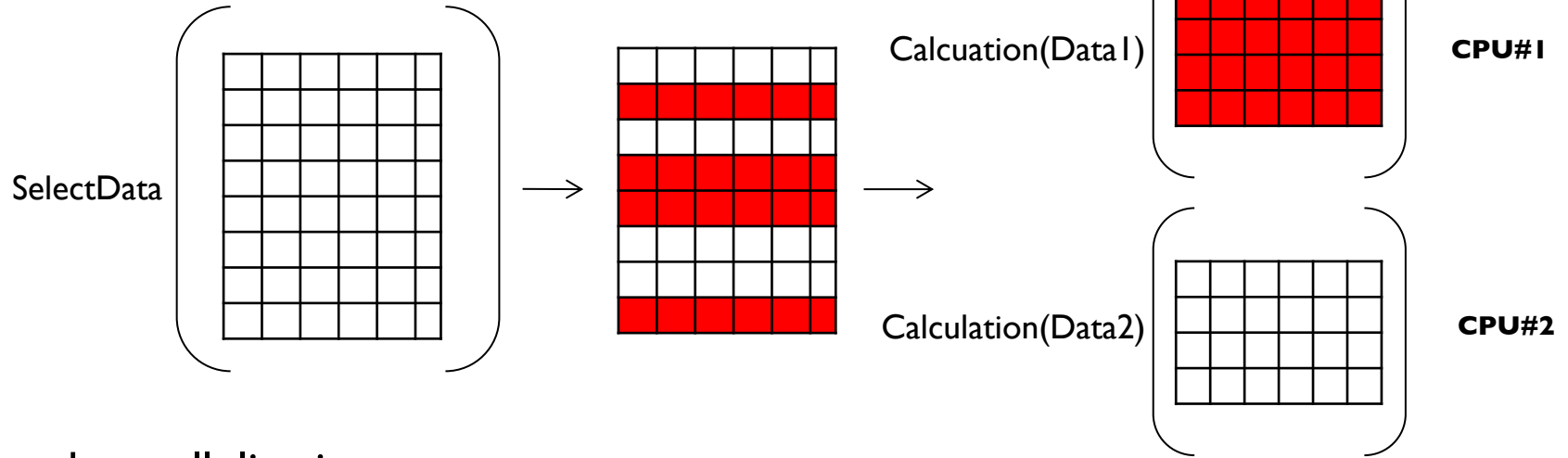
- design your algorithm to be parallelizable!



## Computer Architectures

- design your algorithm to be parallelizable!

### data parallelisation

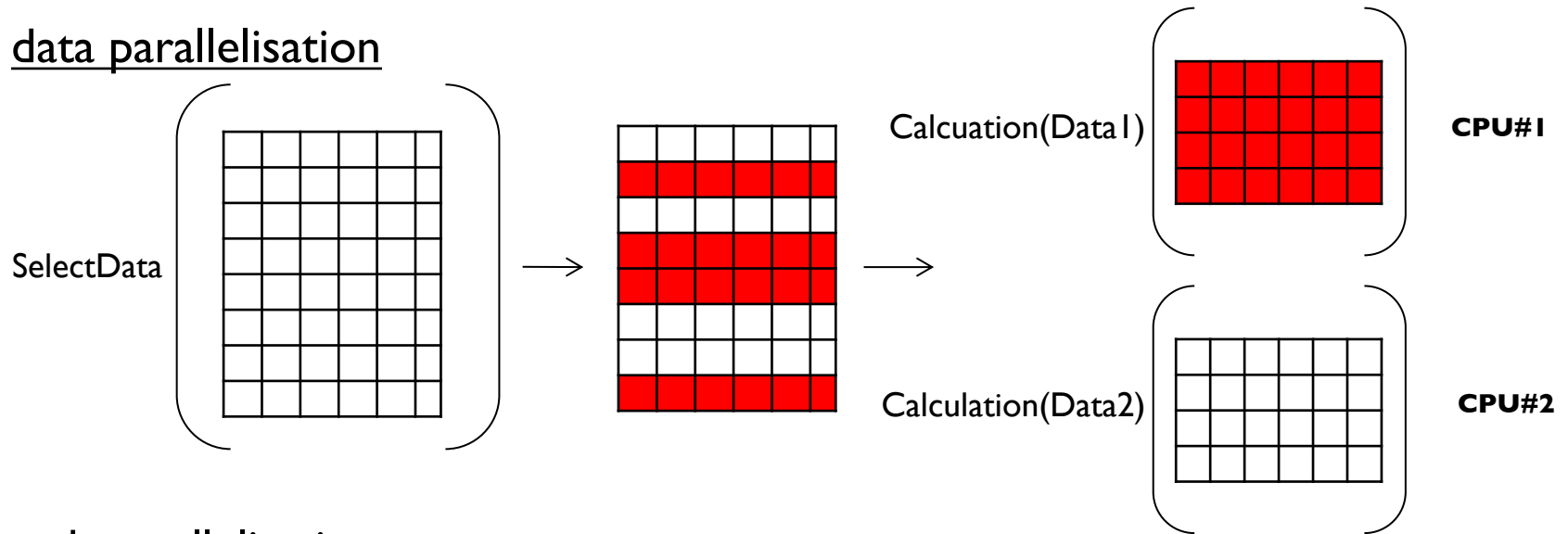


### task parallelisation

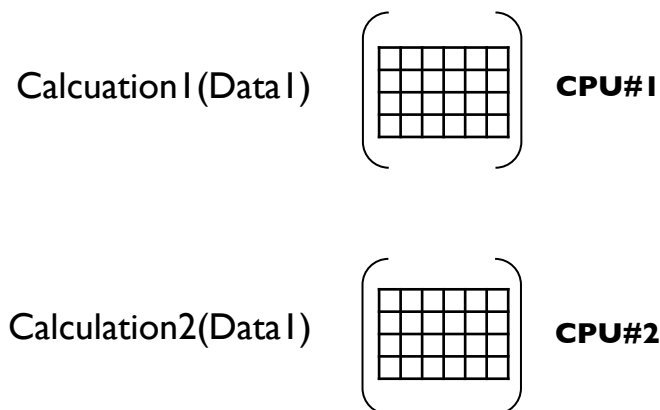
## Computer Architectures

- design your algorithm to be parallelizable!

### data parallelisation



### task parallelisation



## Computer Architectures

- 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!



let's put our knowledge into action:

write a code that calculates

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

**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.);
```