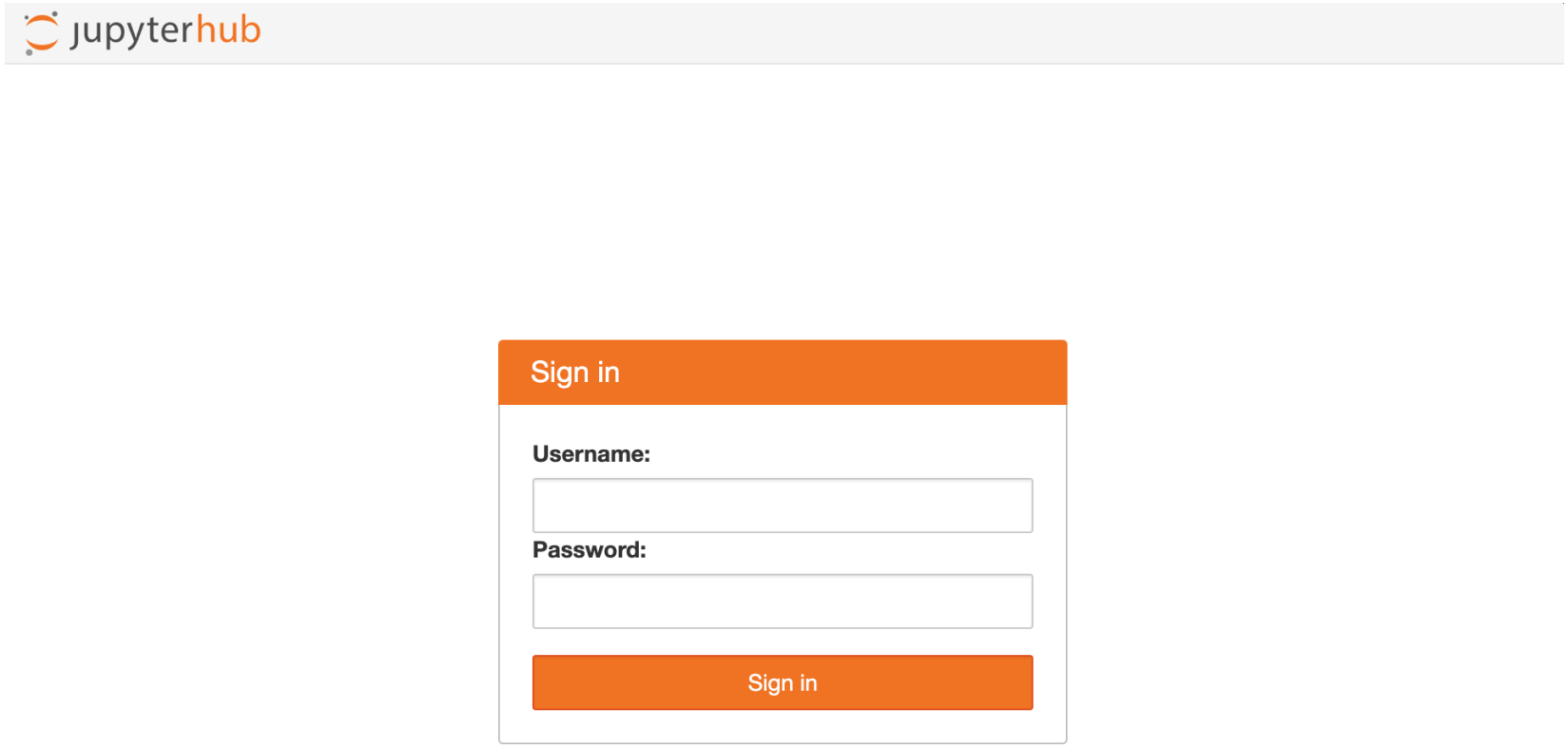
High Performance Computing and Parallel Programming

Teeraparb Chantavat

Institute for Fundamental Study
Naresuan University

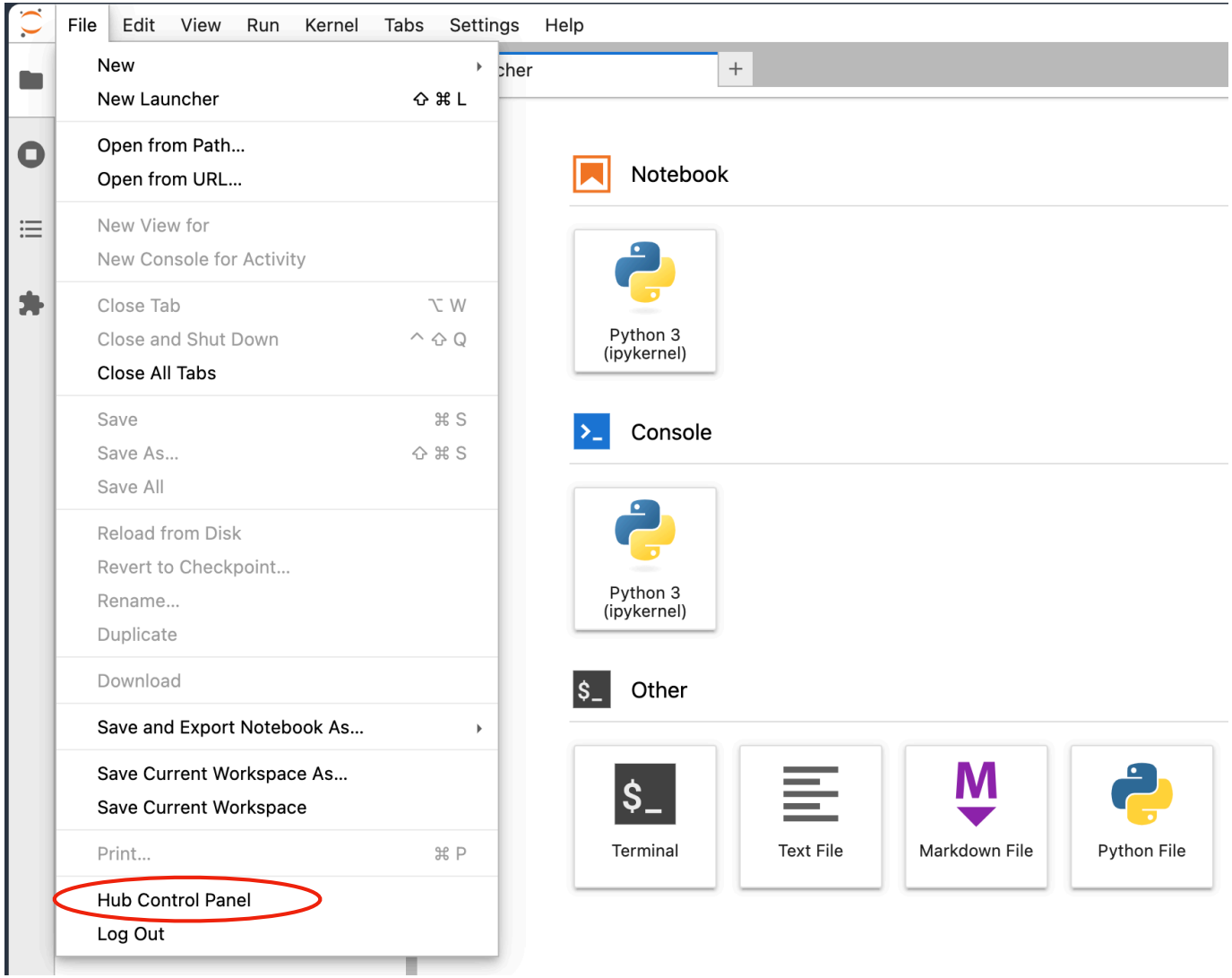
Chalawan Cluster

https://if-nu.narit.or.th/



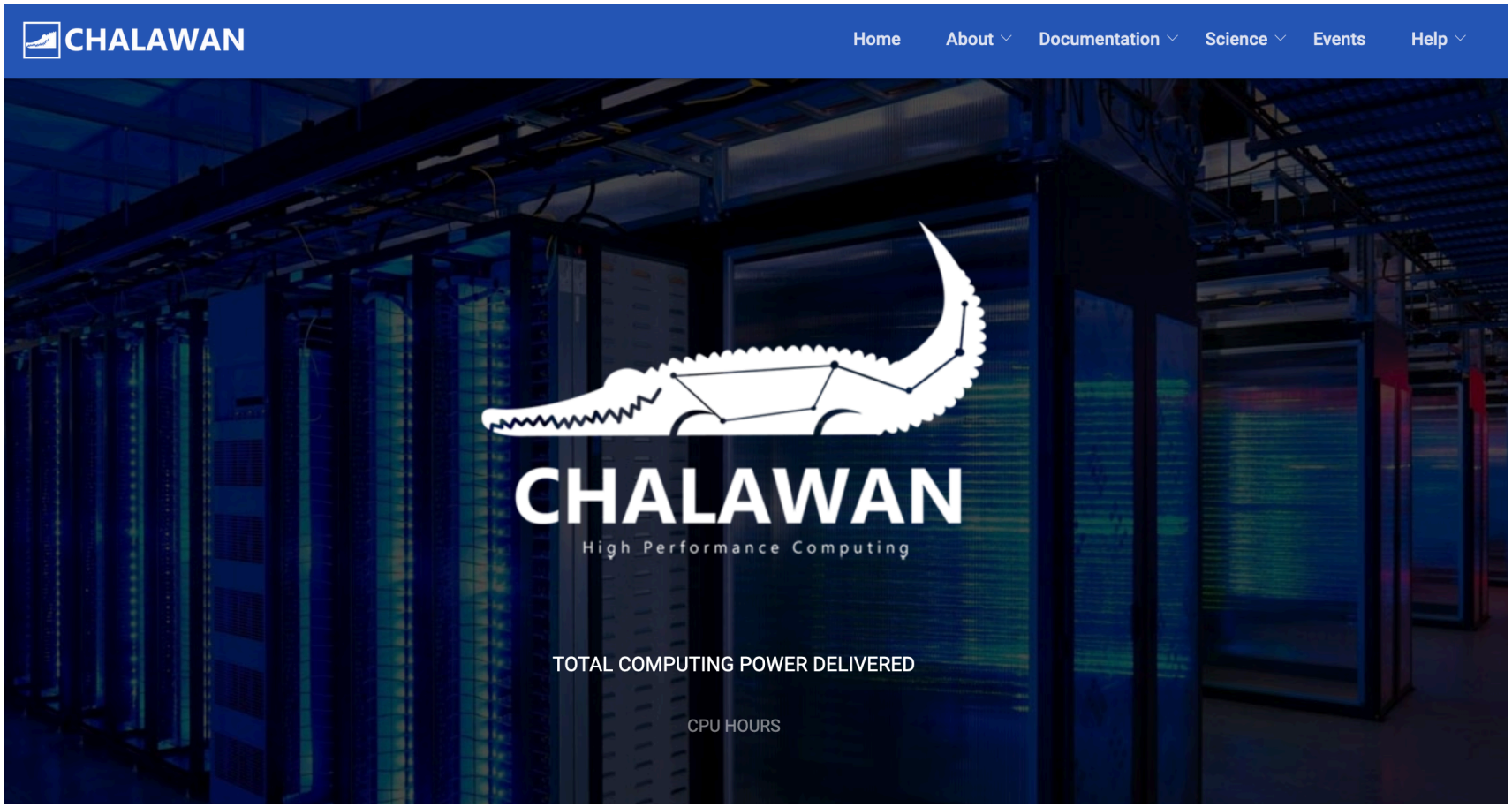
Chalawan Cluster

https://if-nu.narit.or.th/

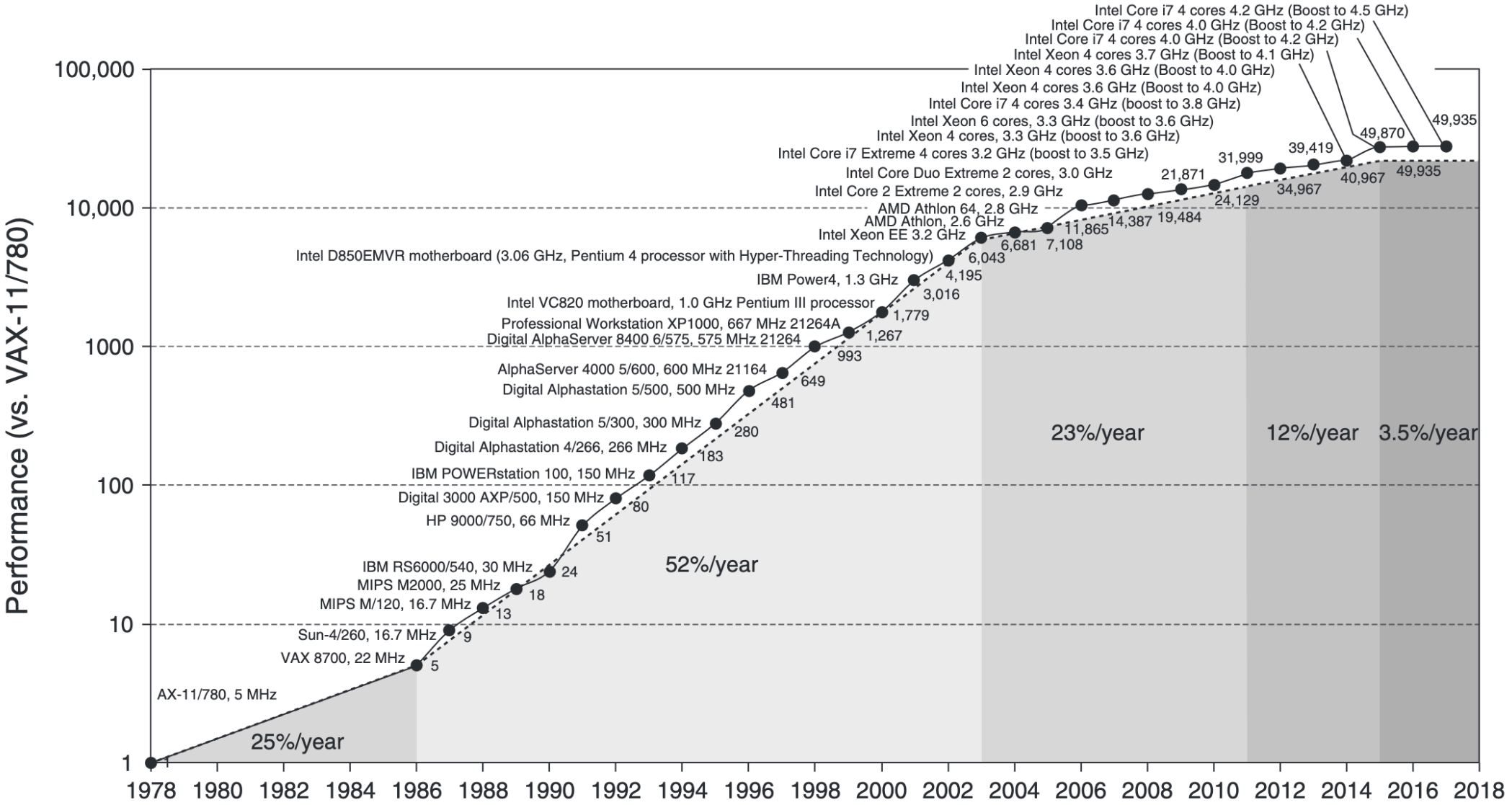


Chalawan Cluster

http://chalawan.narit.or.th/



Why Parallel Computing?



Why Parallel Computing?

- From 1986 to 2003, the performance of microprocessors increased, on average, more than 50% per year. (doubling the performance every 2 years)
- Since 2003, single-processor performance improvement has slowed to the point that in period 2015 - 2017, it increased less than 4% per year.
- By 2005, most microprocessor manufacturers move in the direction of parallelism.

Why Parallel Computing?

- Adding more processors will not improve the performance of the **serial** programs.
- Such programs are unaware of the existence of multiple processors.
- To efficiently exploit parallelism, codes has to be written in such a way that parallelism is realized.

Why Parallel Computing?

Main Reasons:

- Save time and/or Money
- Solve larger / more complex problems
- Provide concurrency
- Take advantage of non-local resources through network
- Make better use of the underlying parallel hardware

Glossaries

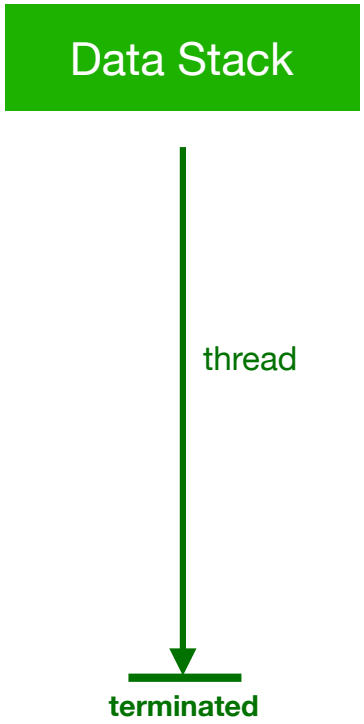
- Process, Thread
- Core, Socket, CPU, Hyper-Thread
- Compiler and Interpreter
- SIMD and MIMD
- Shared-memory, Distributed-memory
- Uniform Memory Access and Non-uniform Memory Access
- Currency and Parallelism

Process VS Thread

- A **process** is a program in execution.
- A process is an independent entity and does not share memory with other processes.
- A **thread** is a part of a process.
- Multiple threads can be in a process.
- Threads share the resources of a process among themselves.

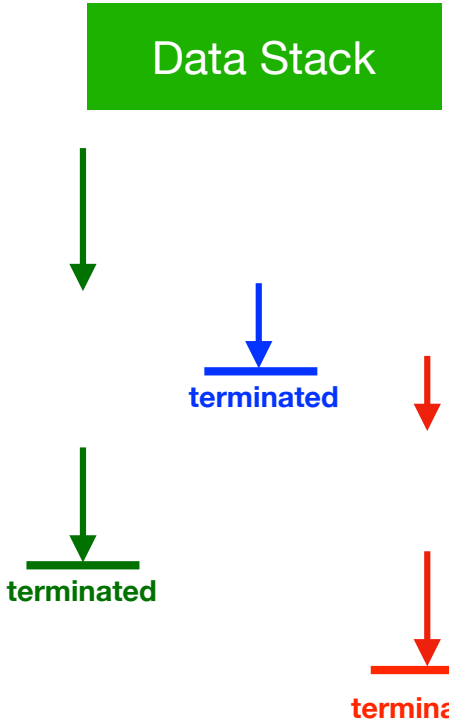
Process VS Thread

Single-thread Process

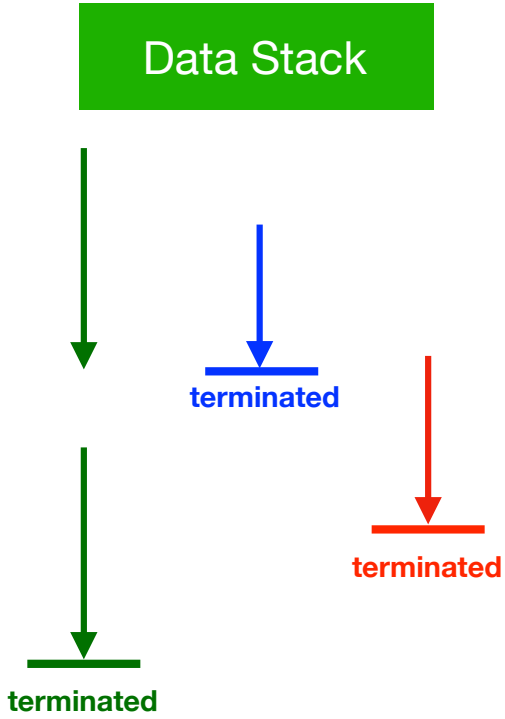


Multi-thread Process

Single-Core CPU



Multi-Core CPU



Distributed Memory System

```
[POLLUX:~] $ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                28
On-line CPU(s) list:   0-27
Thread(s) per core:    1
Core(s) per socket:    14
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Gold 6132 CPU @ 2.60GHz
Stepping:              4
CPU MHz:               2600.000
BogoMIPS:              5200.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              19712K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27
```

Core, Socket, CPU, Hyper-thread

My Desktop Computer

logical

hyper-thread

```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                12
On-line CPU(s) list:   0-11
Thread(s) per core:    2
Core(s) per socket:    6
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 44
Model name:            Intel(R) Xeon(R) CPU           E5645  @ 2.40GHz
Stepping:              2
CPU MHz:               1600.472
BogoMIPS:              3200.01
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              12288K
```

Core, Socket, CPU, Hyper-thread

CPU (physical) = Core(s) per socket x Socket(s)

CPU (logical) = CPU (physical) x Thread(s) per core

Thread(s) per core > 1 ⟶ Hyper-thread

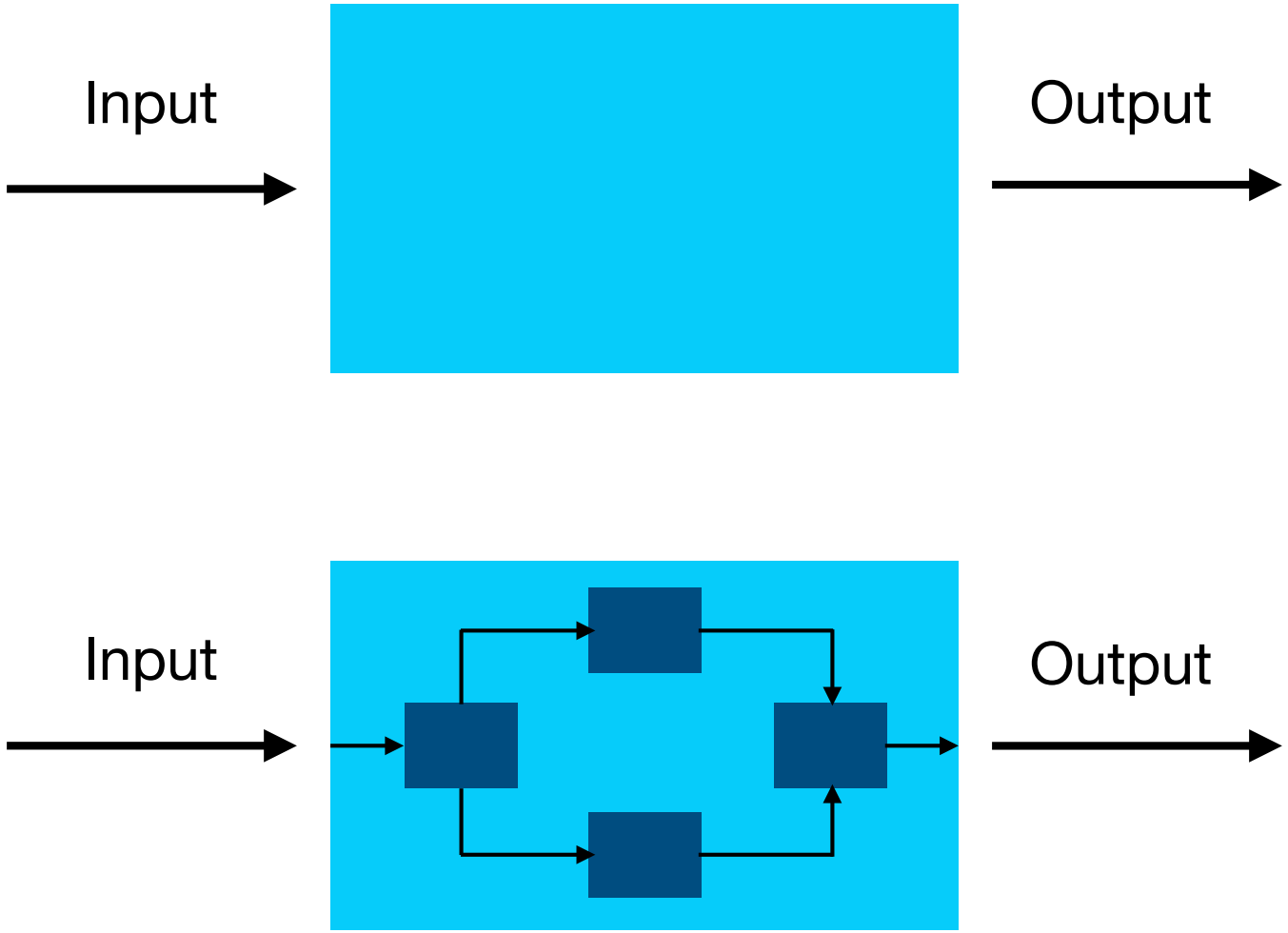
Compiler VS Interpreter

high/slow/easy/low efficiency

- Interpreted / Script Language - Interpreter
  Python, Mathematica, Matlab, R, Maple, Ruby, Perl, …
- Compiled Language - Compiler
  C, C++, Fortran, Ada, Pascal, …
- Assembly Language - Assembler
- Machine Language

low/fast/difficult/high efficiency

Level of Encapsulation



C

```c
nsize = 1000;
matrix = random_matrix(nsize, nsize);

total = 0.0;
for(i = 0; i < nsize; i++)
    for(j = 0; j < nsize; j++)
        total += matrix[i][j];
```

Python

```python
import numpy as np

nsize = 1000
matrix = np.random.rand(nsize, nsize)

for i in range(nsize):
    for j in range(nsize):
        total = total + matrix[i,j]
```

C

```
nsize = 1000;
matrix = random_matrix(nsize, nsize);

total = 0.0;
for(i = 0; i < nsize; i++)
    for(j = 0; j < nsize; j++)
        total += matrix[i][j];
```

Python

```
import numpy as np

nsize = 1000
matrix = np.random.rand(nsize, nsize)

for i in range(nsize):
    for j in range(nsize):
        total = total + matrix[i,j]
```

```
import numpy as np

nsize = 1000
matrix = np.random.rand(nsize, nsize)
total = marray.sum()
```

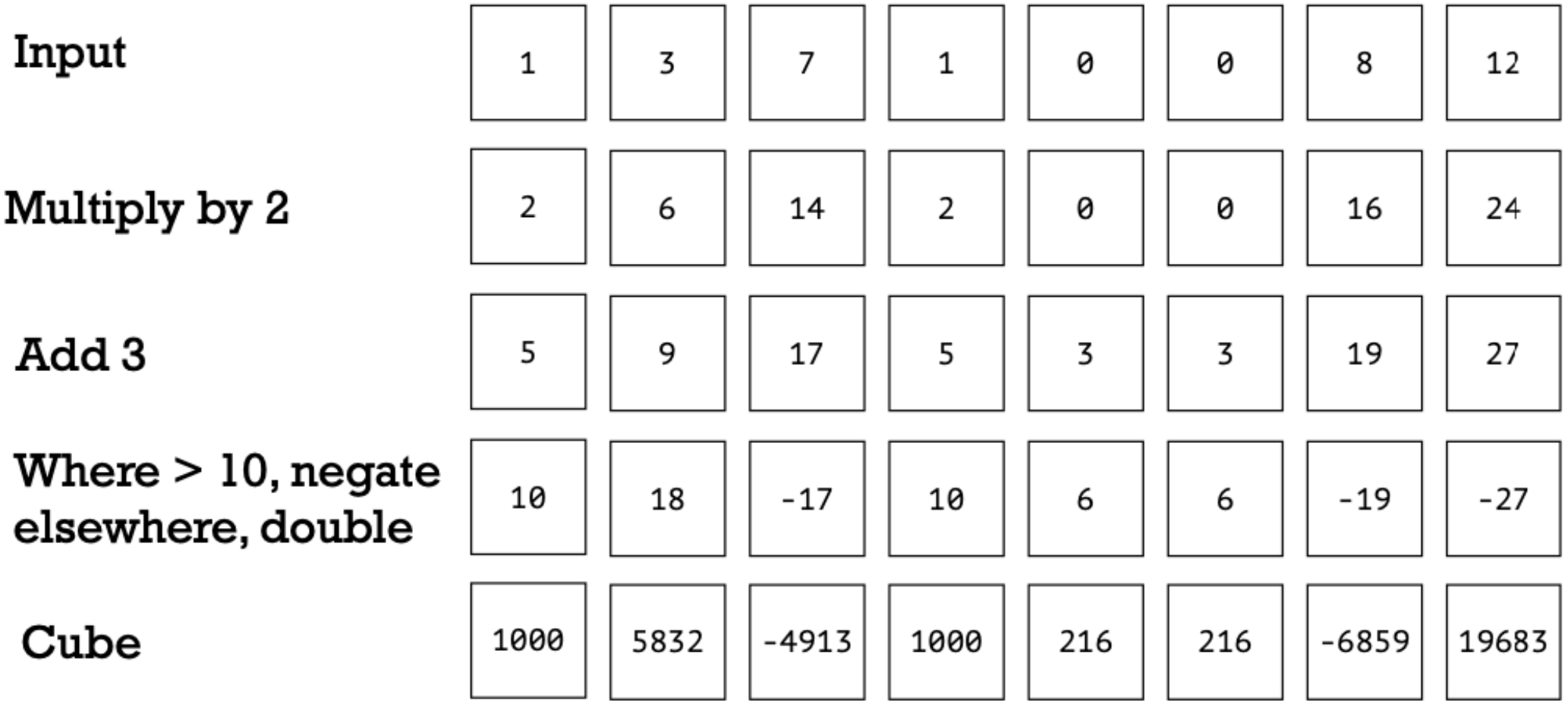
SIMD and MIMD

- Single instruction, multiple data (SIMD)
  - data parallelism
  - Graphic Processing Unit (GPU)
- Multiple instruction, multiple data (MIMD)
  - task parallelism
  - shared memory
  - distributed memory

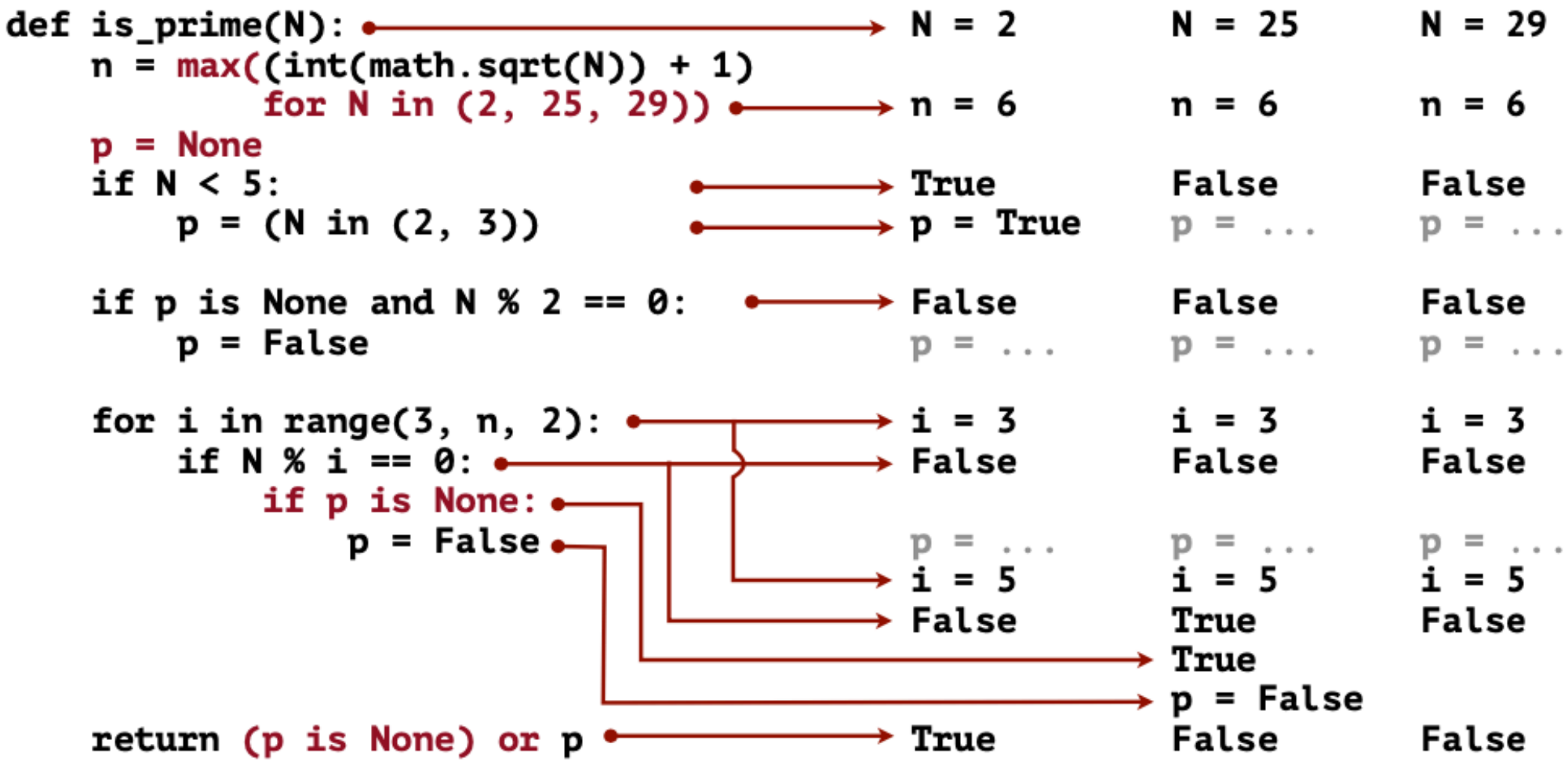
SIMD: Single Instruction, Multiple Data

- Single program
- Single instruction pointer
- Executes on multiple processors, with different data,
  - All branches execute on all data.
  - No early exit for loop/functions on some branches only
  - Ideal for graphics (GPU; same operation on entire image), physical simulations etc.

SIMD: Single Instruction, Multiple Data



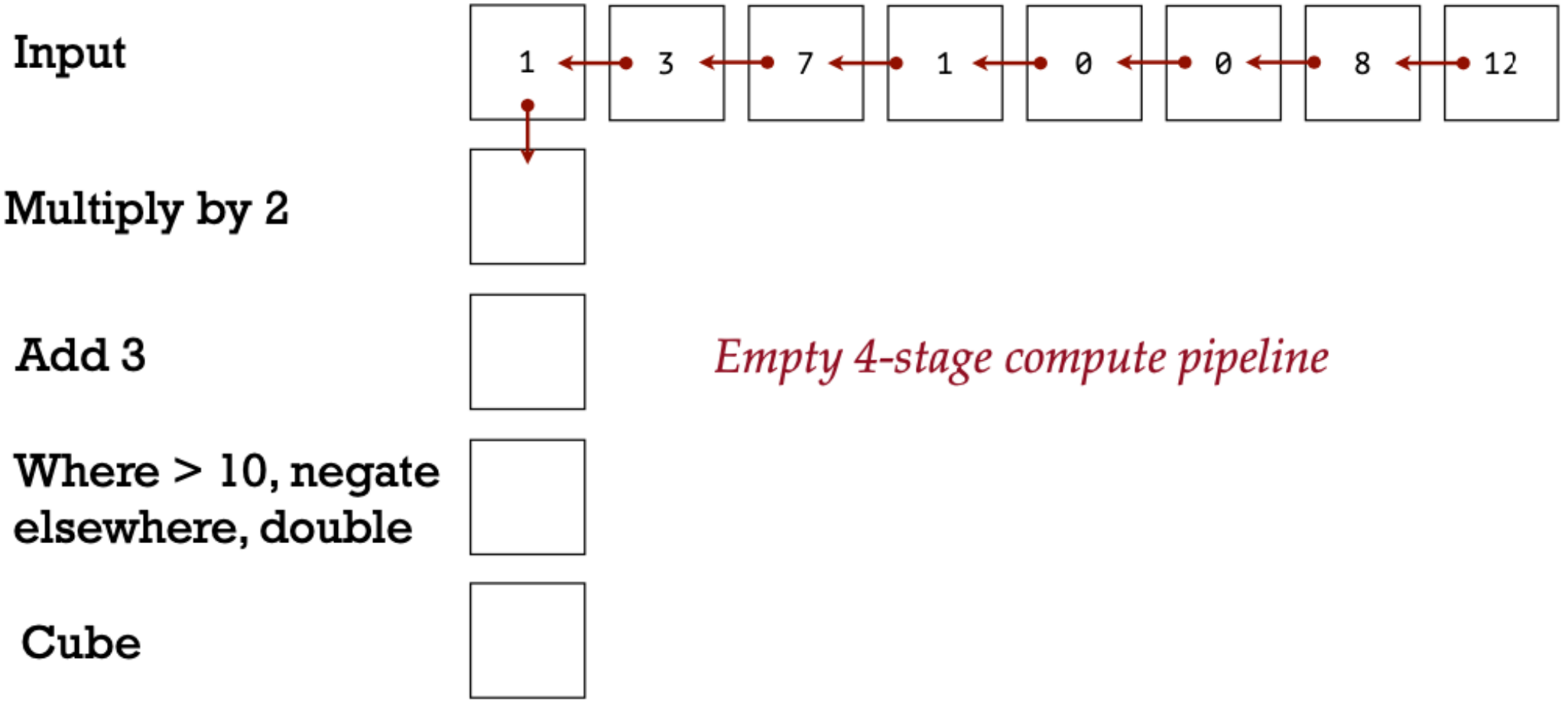
SIMD: Single Instruction, Multiple Data



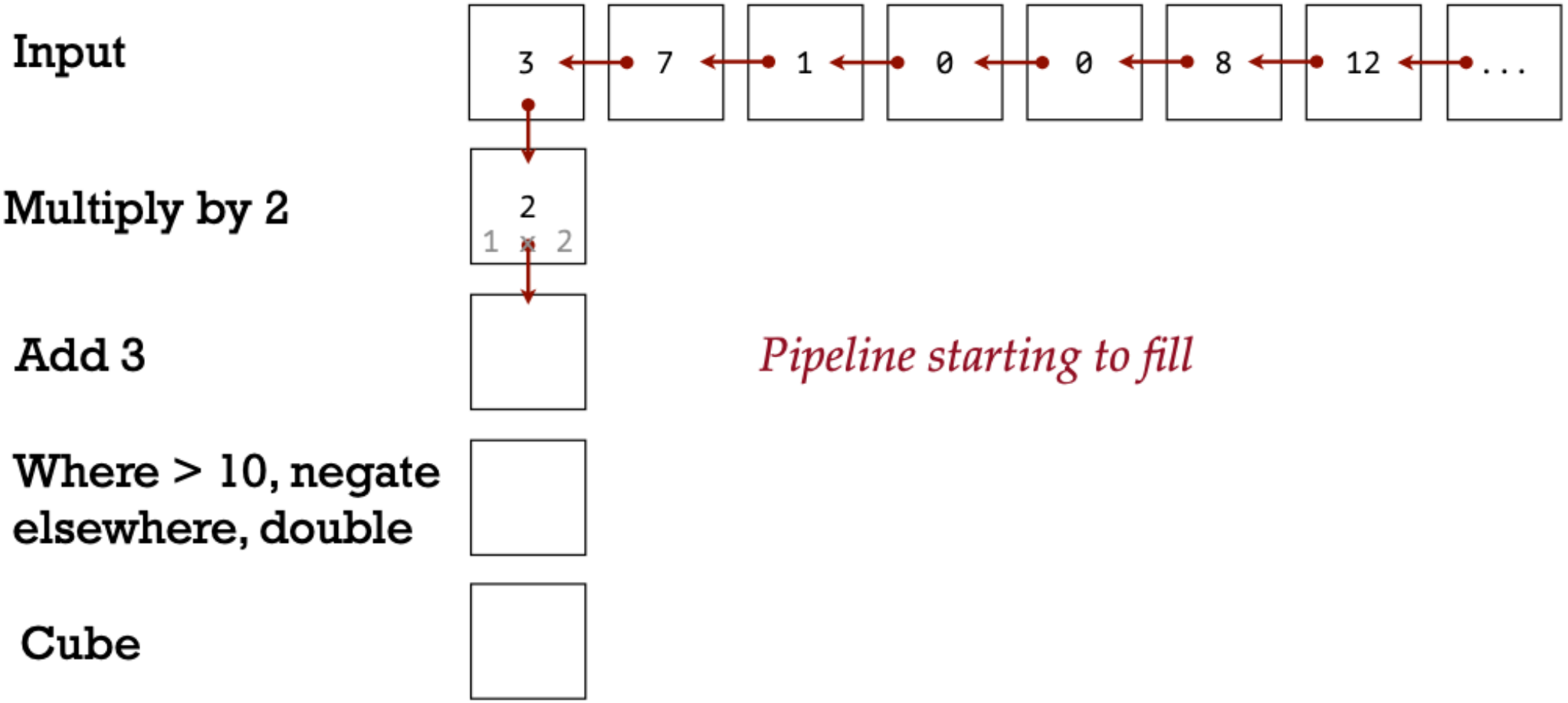
Vector Processing

- Single program
- Multiple execution units performing different functions (fixed or programmable)
- Data passes through the pipeline, moves through the execution units in turn
- Once the pipeline fills, an N-stage pipeline executes_ N instructions each clock cycles.

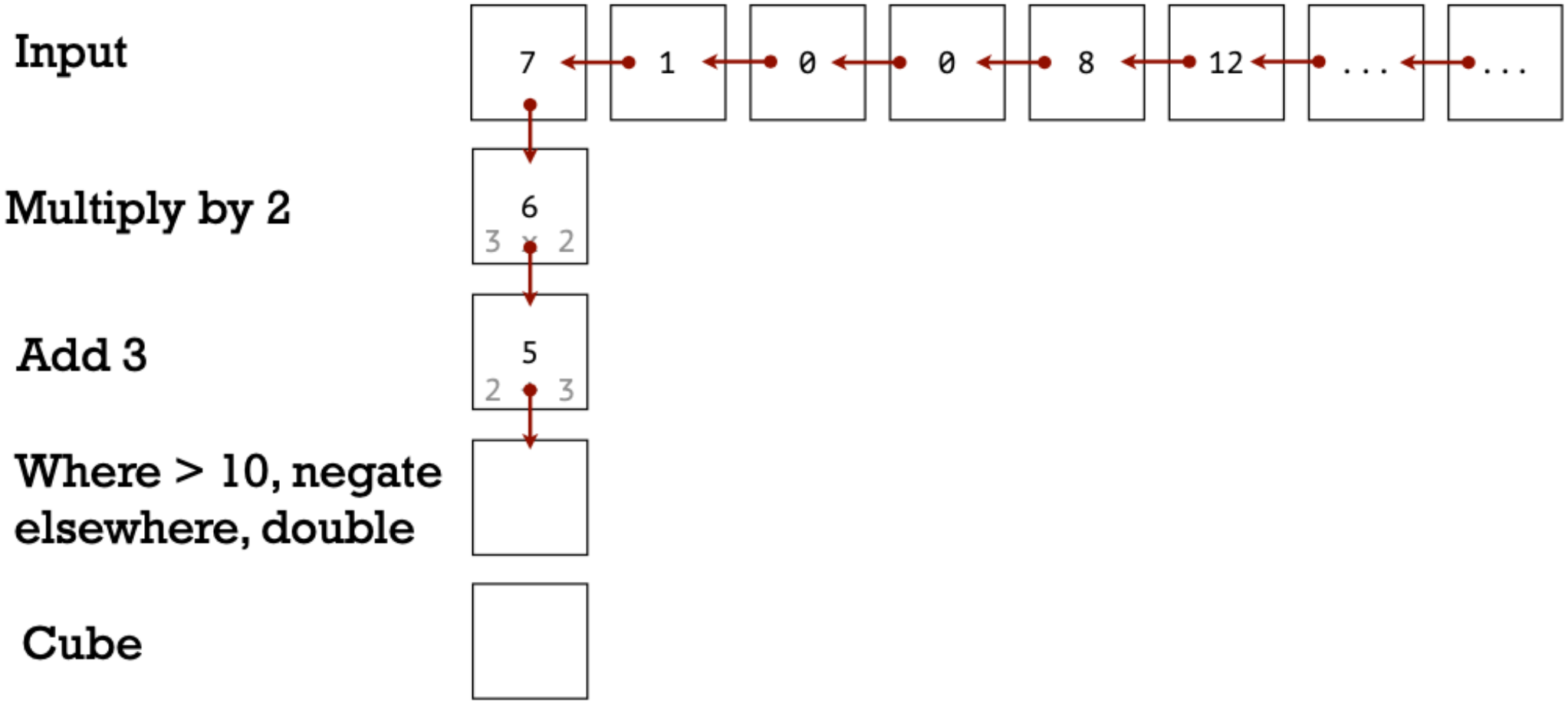
Vector Processing



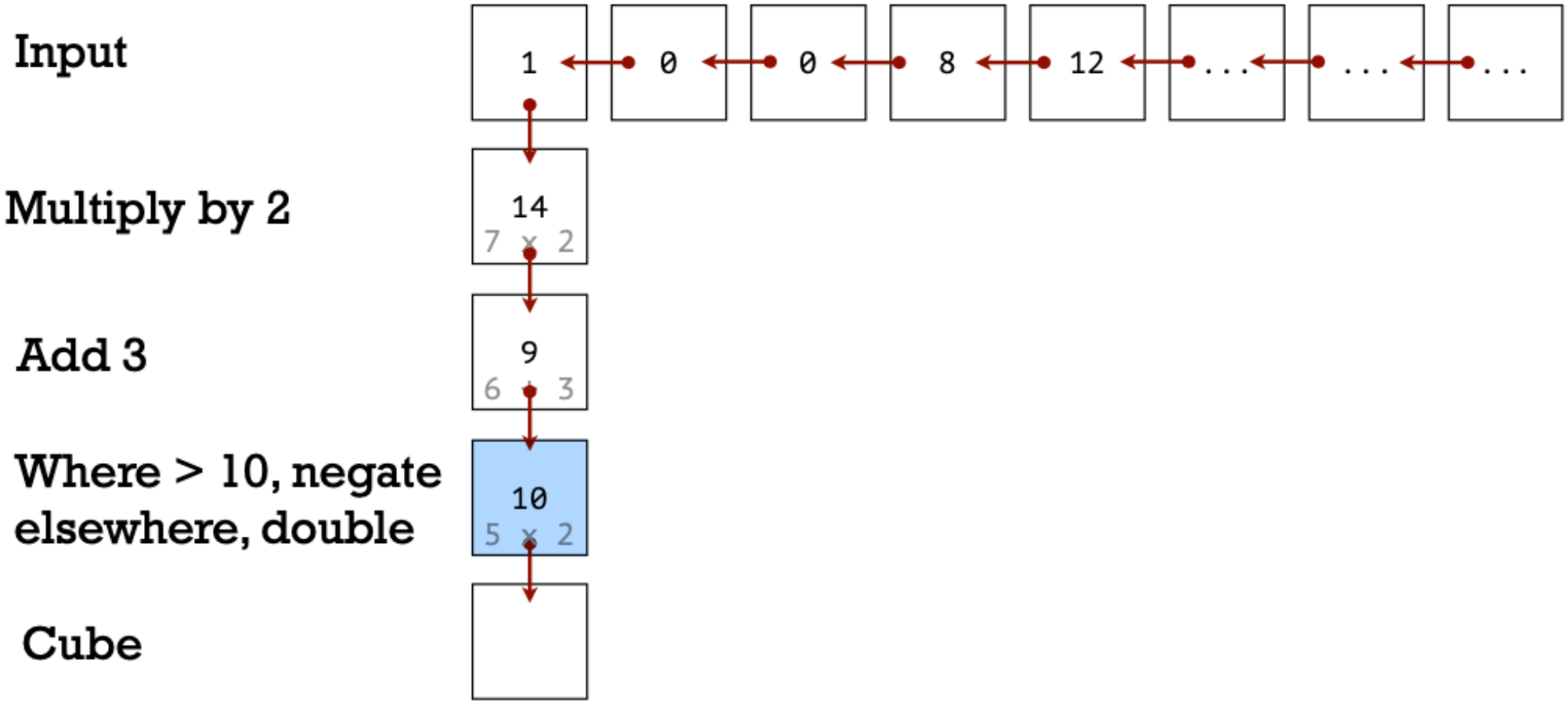
Vector Processing



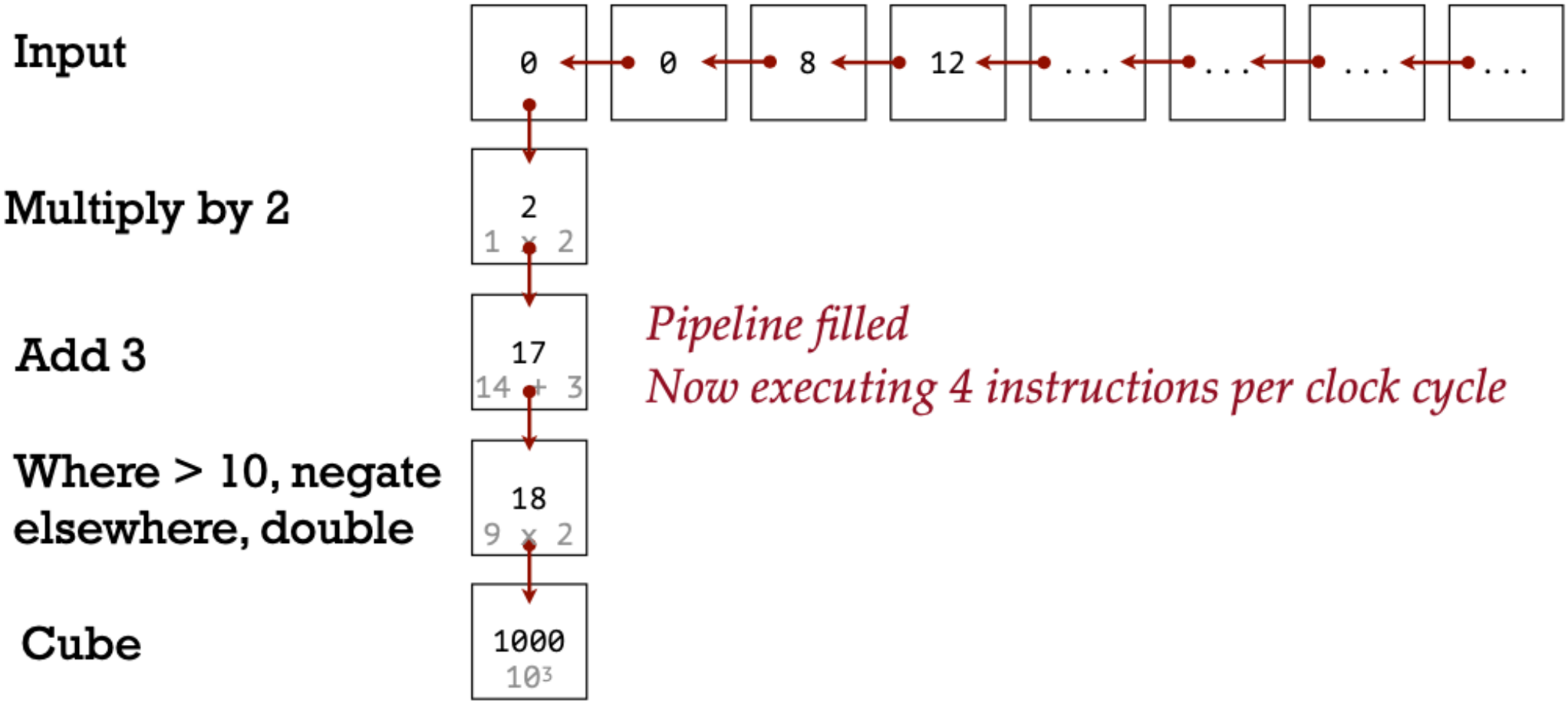
Vector Processing



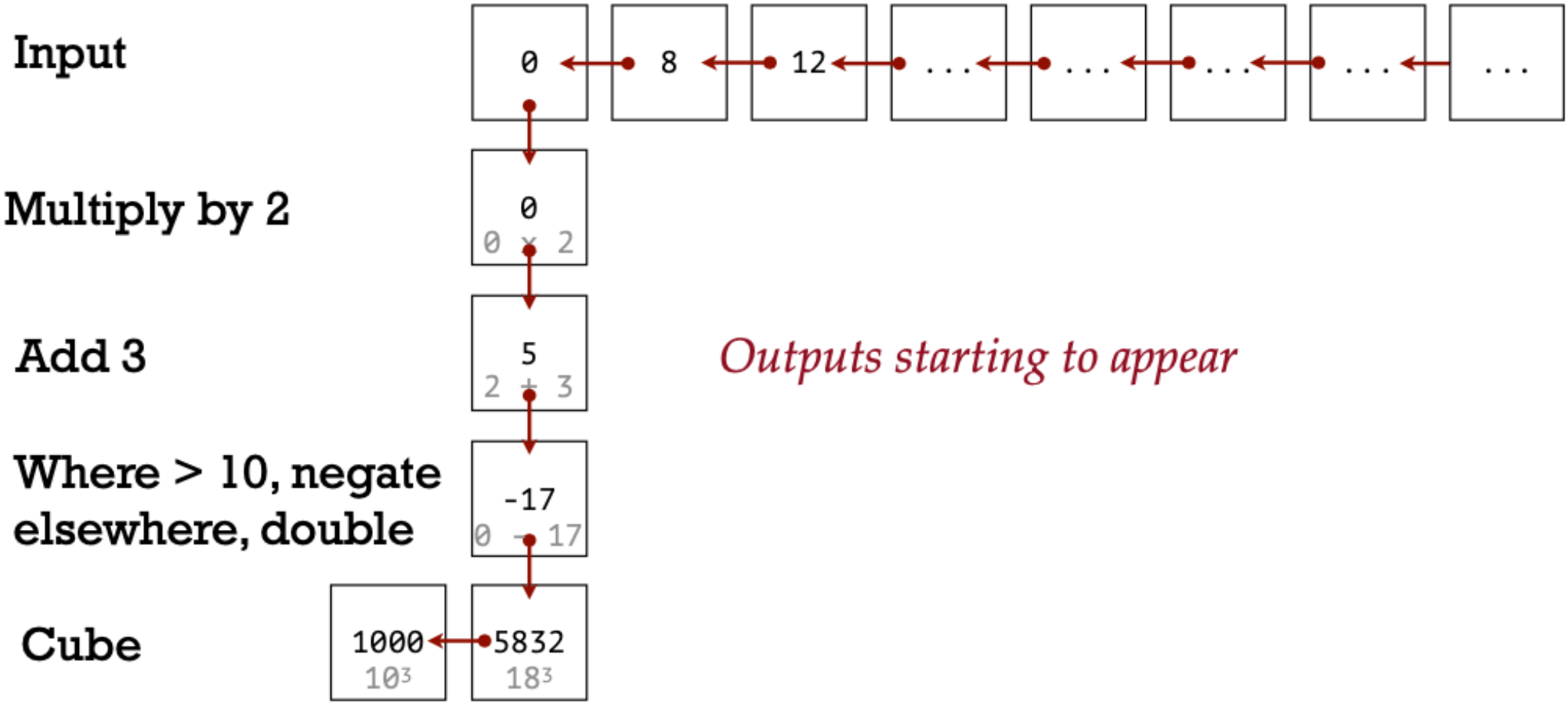
Vector Processing



Vector Processing



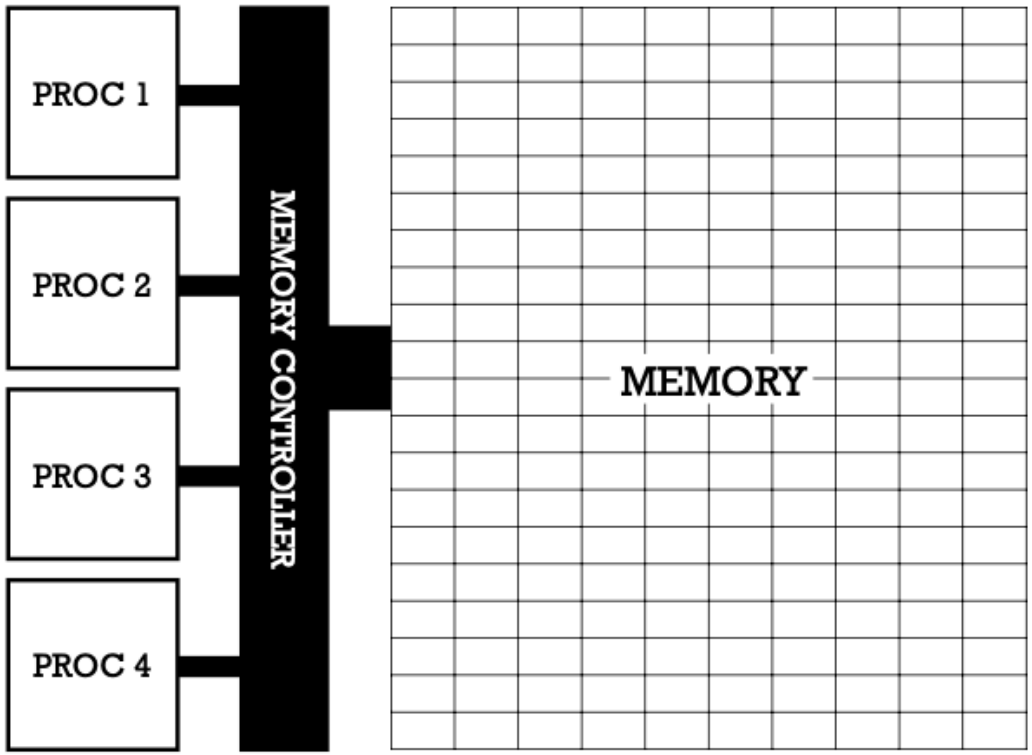
Vector Processing



MIMD: Multiple Instruction, Multiple Data

- A (potentially) different program on each processor
  - in practice, very often the same program; so-called Single Program, Multiple Data (SPMD)
- Each processor has own instruction pointer: they run independently apart from any communications, shared resources etc.

Shared-memory



PROC = PROCESSOR *or* CORE *or* CPU

All processors access a single pool of memory

If access speeds are equal for all processors, known as "symmetric multiprocessing"

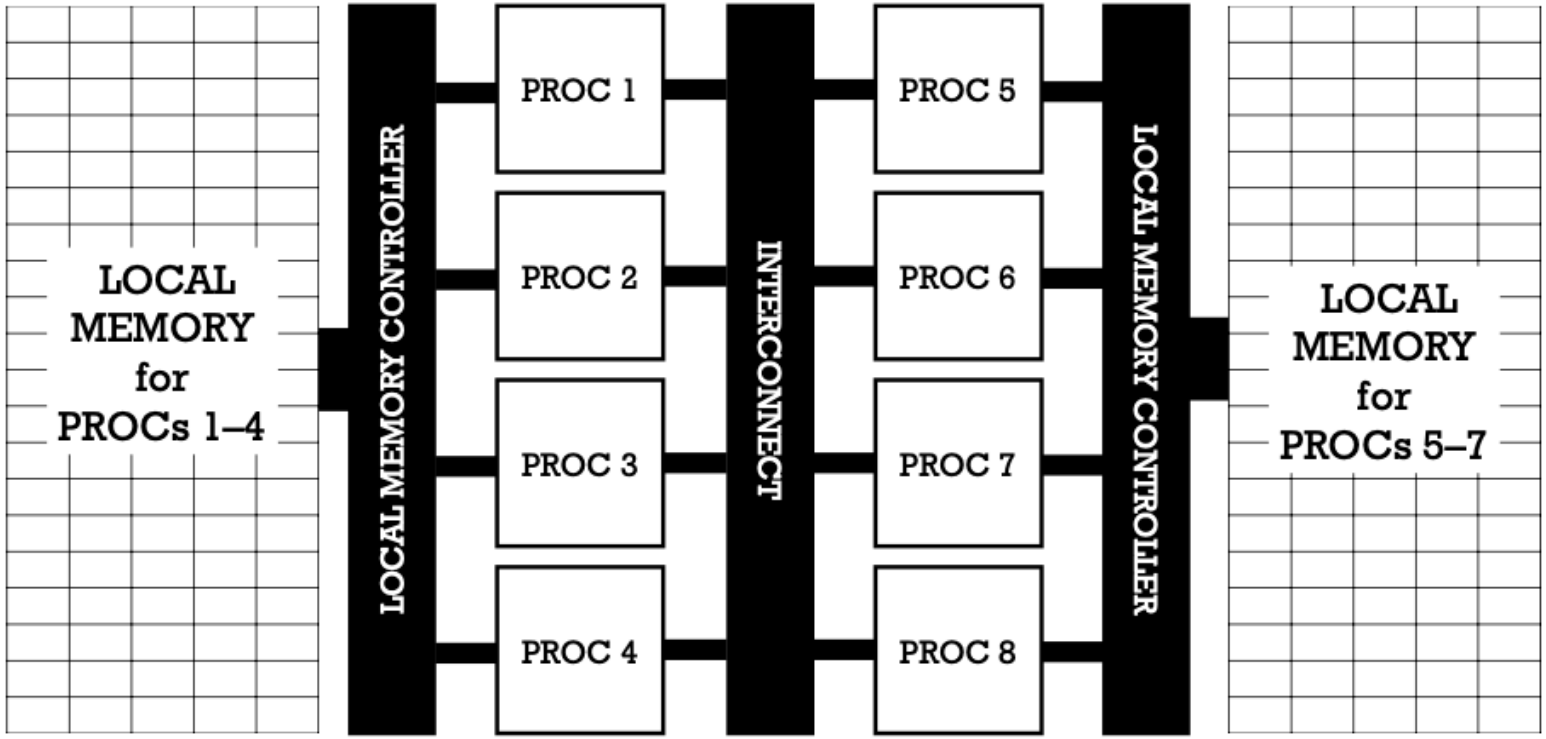
PROS:

- Conceptually Simple
- Direct access to anything

CON:

- Contention
- Need to manage writes, lock etc.

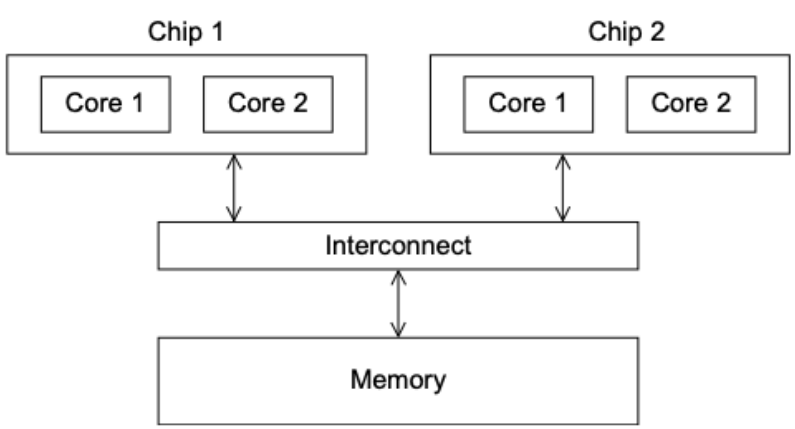
Asymmetric Shared-memory

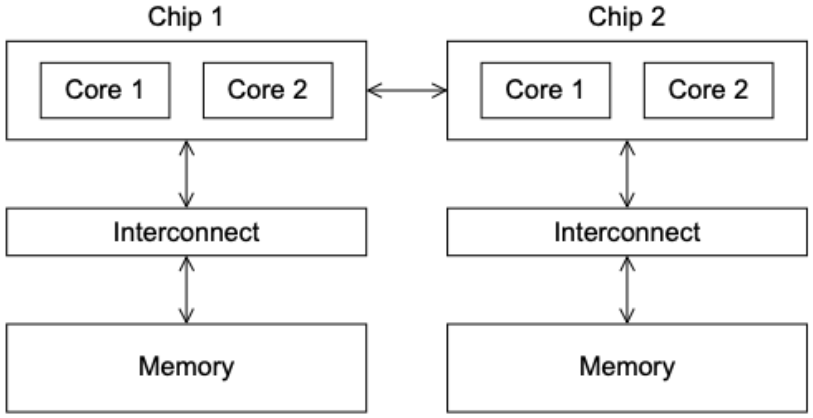


- Free access to local memory
- Slower access to non-local memory through interconnect

Non-Uniform Memory Architecture (NUMA)

Uniform Memory Access
VS
Non-uniform Memory Access

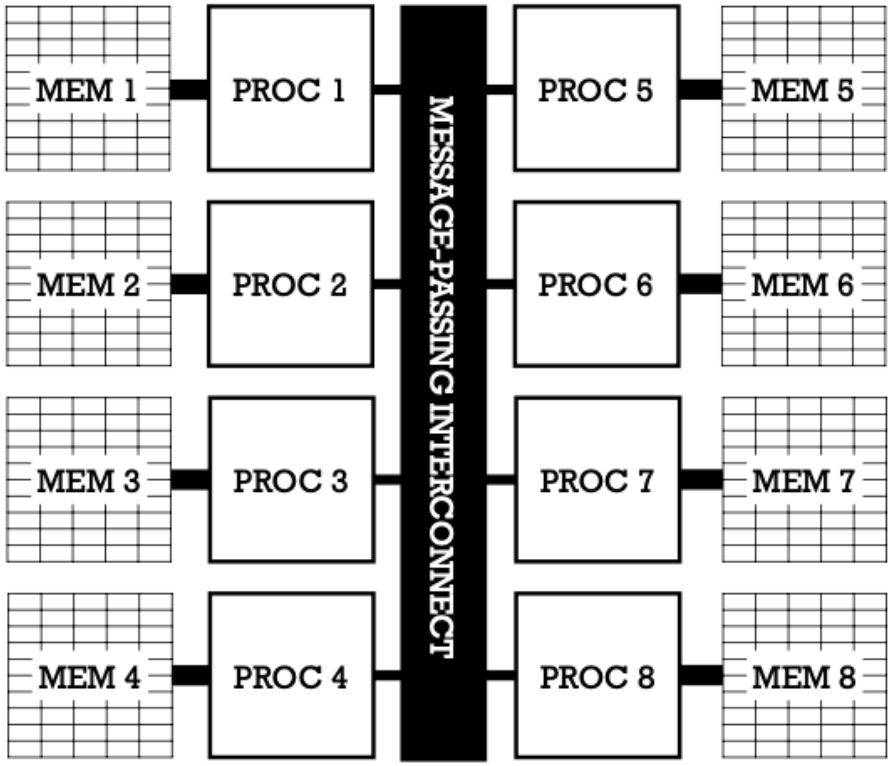




- Uniform Memory Access (UMA)
  share the same physical memory and have equal access time to any memory location.

- Non-Uniform Memory Access (NUMA)
  memory is divided into several regions, each of which is local to a specific processor or group of processors.

Distributed-memory



PROC = PROCESSOR *or* CORE *or* CPU

Each processor has own money, which is the only memory it can access

Each processor also has access to a messaging system, allowing it to send and receive messages (data) to and from any other processor

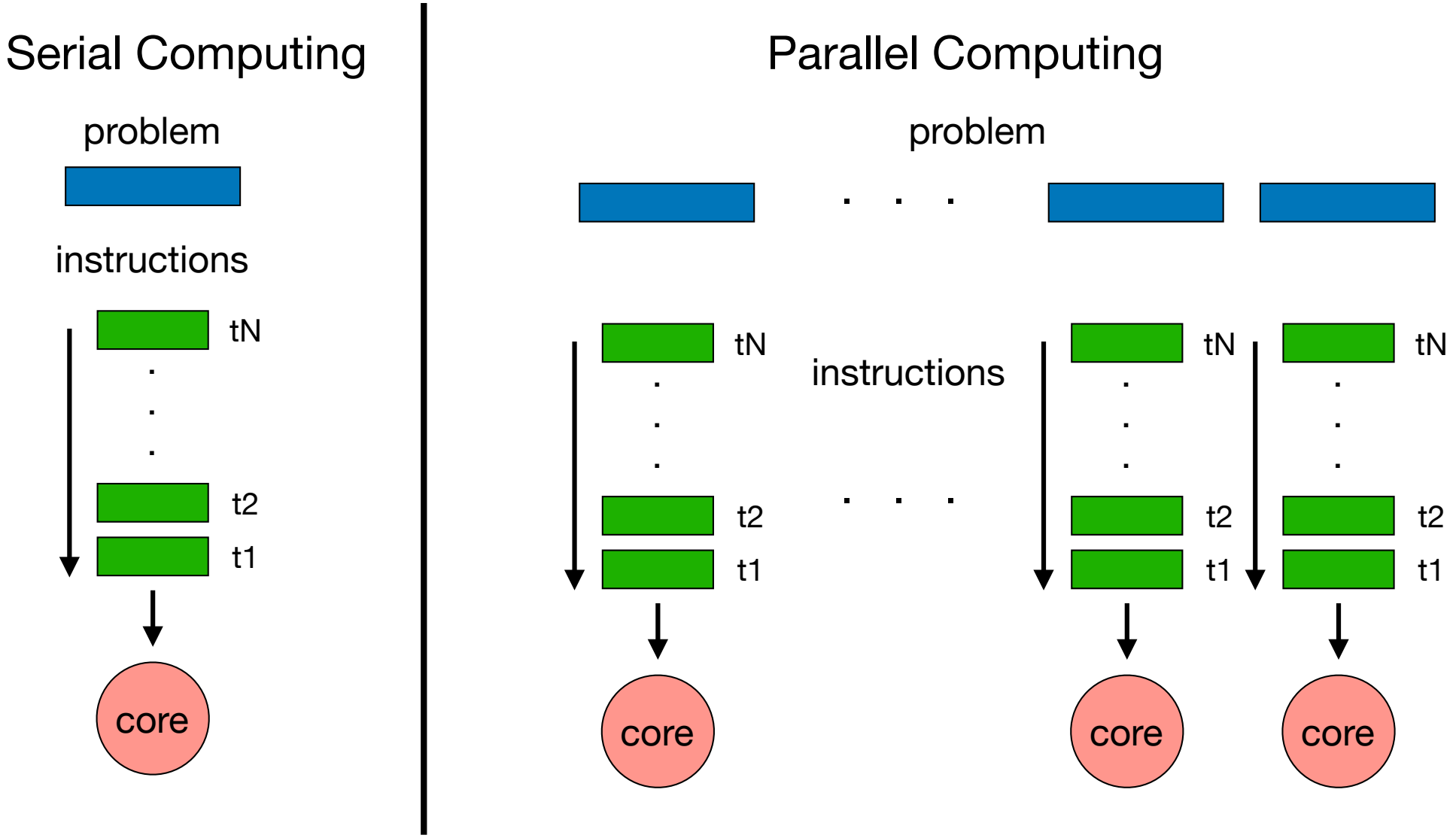
PROS:

- No contention
- Can be used on distributed and shared-memory system

CON:

- Conceptually harder then shared memory
- May cause latency
- Potential for deadlock

Parallel Computing



Parallel Computing: Resources

The computing resources can include:

- A single computer with multiple processors;
- A single computer with (multiple) processor(s) and some specialized computer resources (GPU)
- An arbitrary number of computers connected by a network;
- A combination of both.

Parallel Computing: Computational problem

The computing problem usually demonstrates characteristics such as the ability to be:

- Broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Solved in less time with multiple compute resources than with a single compute resource.

Performance

Ideal $$T_{\text{parallel}} = \frac{T_{\text{serial}}}{p}, \quad p \equiv \text{number of cores.}$$

Practical $$T_{\text{parallel}} = \frac{T_{\text{serial}}}{p} + T_{\text{overhead}}.$$

Performance

Table 2.4 Speedups and Efficiencies of a Parallel Program

p	1	2	4	8	16
S	1.0	1.9	3.6	6.5	10.8
E = S/p	1.0	0.95	0.90	0.81	0.68

Table 2.5 Speedups and Efficiencies of a Parallel Program on Different Problem Sizes

	p	1	2	4	8	16
Half	S	1.0	1.9	3.1	4.8	6.2
	E	1.0	0.95	0.78	0.60	0.39
Original	S	1.0	1.9	3.6	6.5	10.8
	E	1.0	0.95	0.90	0.81	0.68
Double	S	1.0	1.9	3.9	7.5	14.2
	E	1.0	0.95	0.98	0.94	0.89

Speedup

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} > 1.$$

Efficiency

$$E = \frac{S}{p} \leq 1.$$

Multicore Programming

Multicore or **multiprocessor** systems putting pressure on programmers, challenge include:

- Dividing activities
- Load balance
- Data splitting / Data dependency
- Testing and debugging

Parallelism implies a system can perform more than one task simultaneously

Concurrency supports more than one task making progress.

Multicore Programming

Types of parallelism

- Data parallelism - distributes subsets of the same data across multiple cores, same operation on each
- Task parallelism - distributing threads across cores, each thread performing unique operation

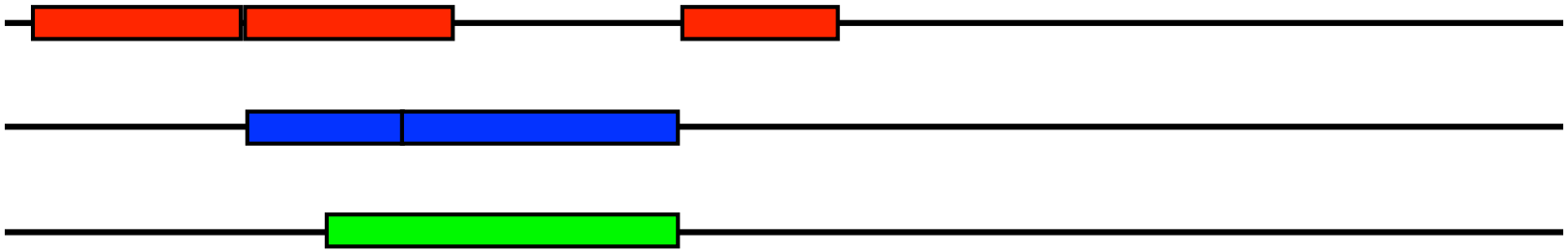
Concurrency vs Parallelism

Types of parallelism

- **Concurrency**: A condition of a system when at least two threads are making progress or multiple tasks are **logically active** at one time.
- **Parallelism**: A condition of a system in which multiple tasks are **actually active** at one time.

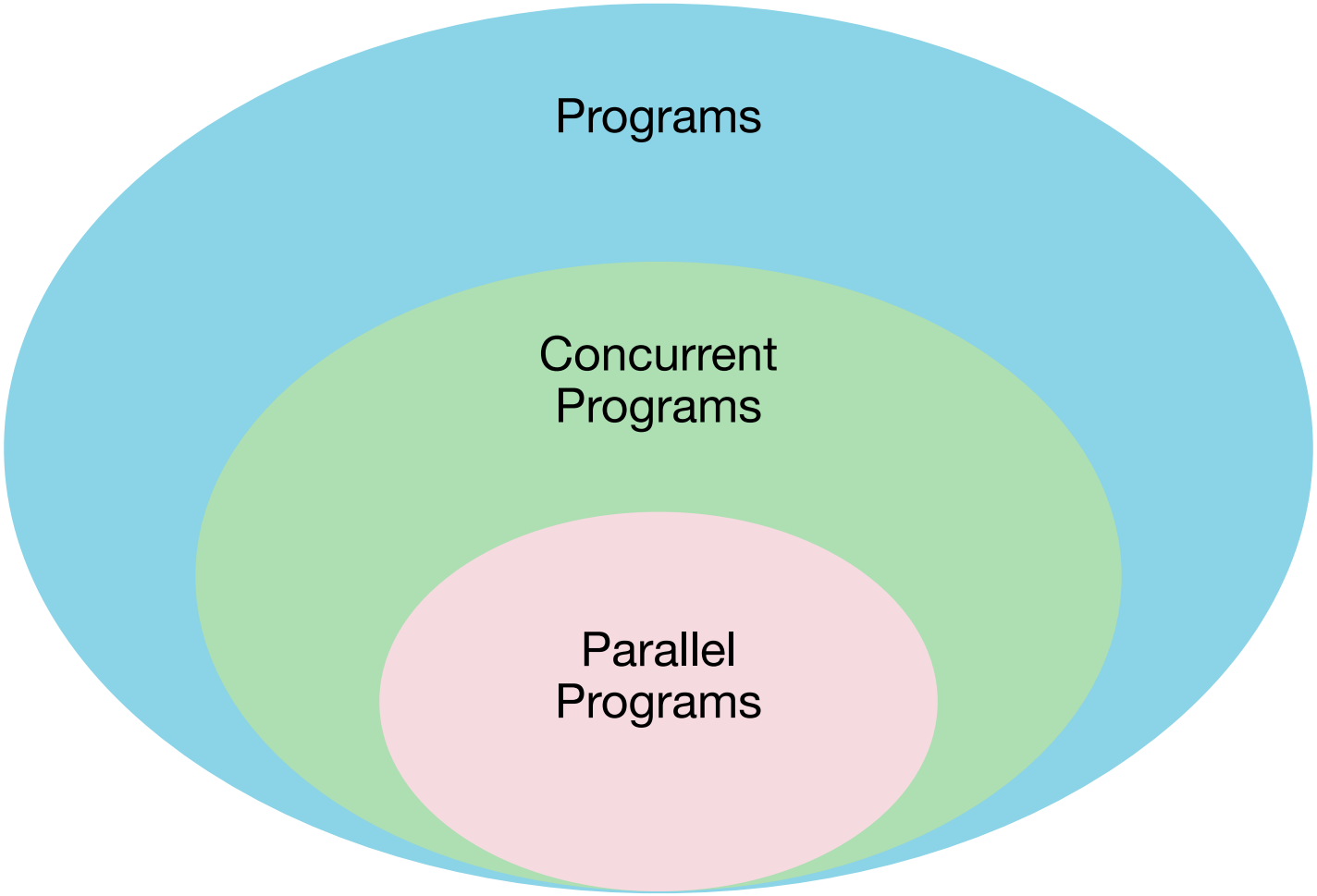
Concurrency vs Parallelism

Concurrent, non-parallel Execution



Concurrent, parallel Execution

Concurrency vs Parallelism



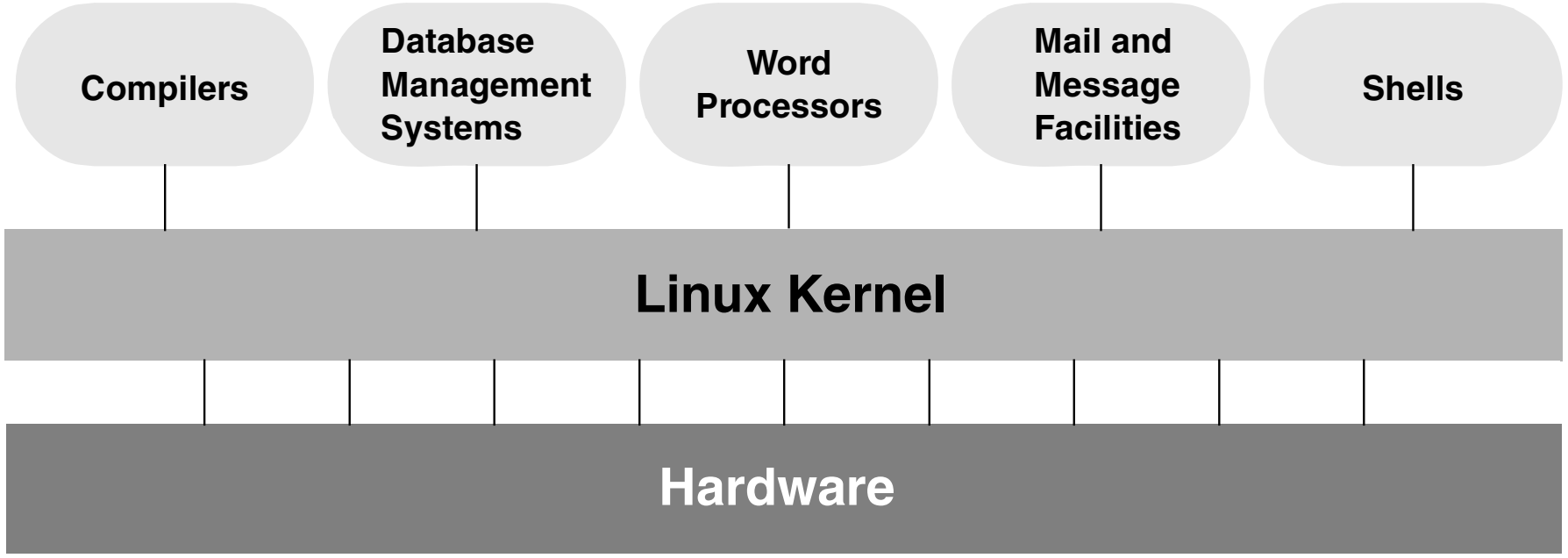
Introduction to Linux Operating System

UNIX Operating System

Why UNIX Operating System?

- It is **free!**
- A suitable environment for research. Many developing tools for programming.
- Non-graphical environment consumes less resources.
- Easy to work with your fellow colleagues and collaborators.
- Many things, if not Windows, are UNIX-like. Linux, Mac OS X, Android, iOS, Chrome OS, Playstation4 etc.

UNIX Operating System



- Linux is a system program as well as a **kernel**.

UNIX History

- **UNIX** is an operating software that run lower-level tasks. Originally, developed for researchers to share resources and information in **Bell Labs 1975**.

- The Bell Labs sold UNIX at nominal cost to **educational institutes** at nominal cost. The students were familiar with UNIX then introduce to **industries**.

UNIX History

- The popularity of UNIX causing many branches such as **System V**, **Solaris**, **BSD** etc.

- The **Linux** kernel was developed by Finnish undergraduate student **Linus Torvalds**, who used the Internet to make the source code available to others for free in **1991**.

UNIX History

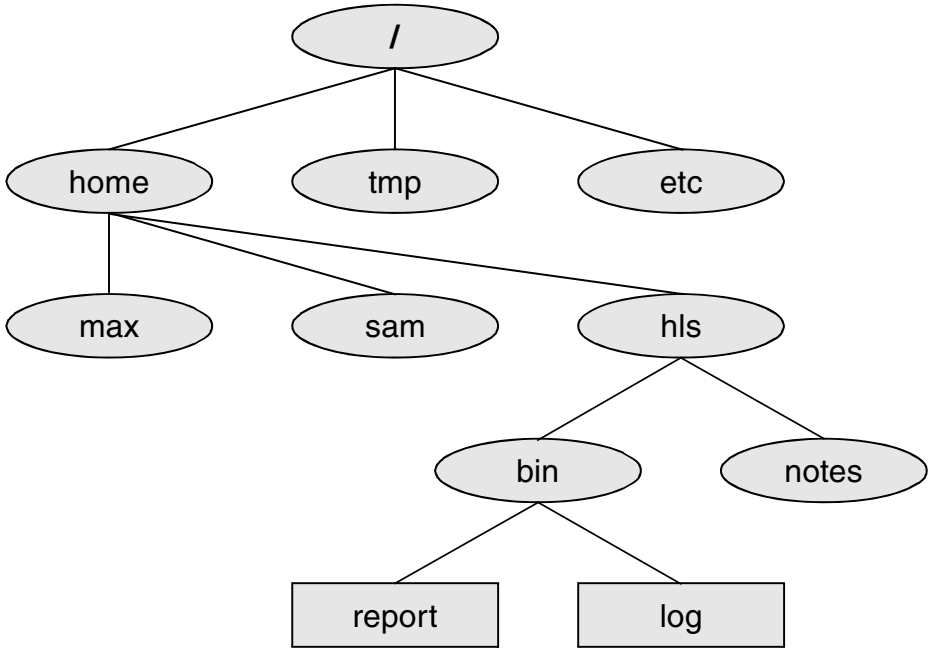
- **Richard Stallman** announced the **GNU Project** for creating an operating system, both kernel and system programs.
- The idea of GNU Project is that the code are free anyone can modified and redistributed with free of charge.
- Most of the Linux we used today are **GNU-Linux**.

Overview of Linux

- Linux operating system is a **kernel** (allocating the computer resources)
- The Linux operating can support many users — Optimizing the use of computing resources.
- Linux allows the users to run his/her job in background.

Overview of Linux

- Linux operating system has a secure hierarchical filesystem.



Overview of Linux

- Linux provides **shell programming languages** that allow the users to interact with system environment.

The four most popular shells are

Bourne Again Shell (bash)

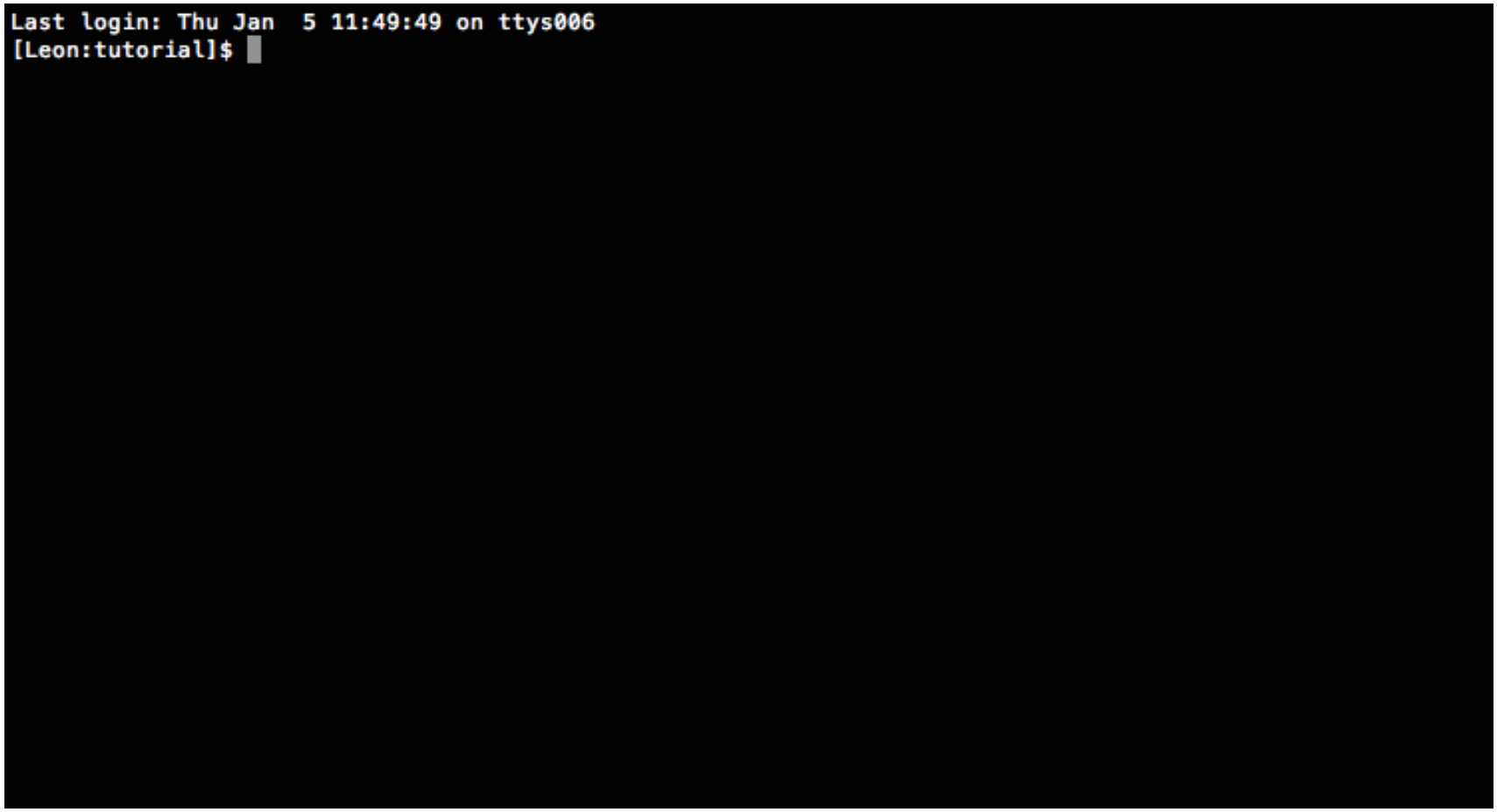
Debian Almquist Shall (dash)

TC Shell (tcsh)

Z Shell (zsh)

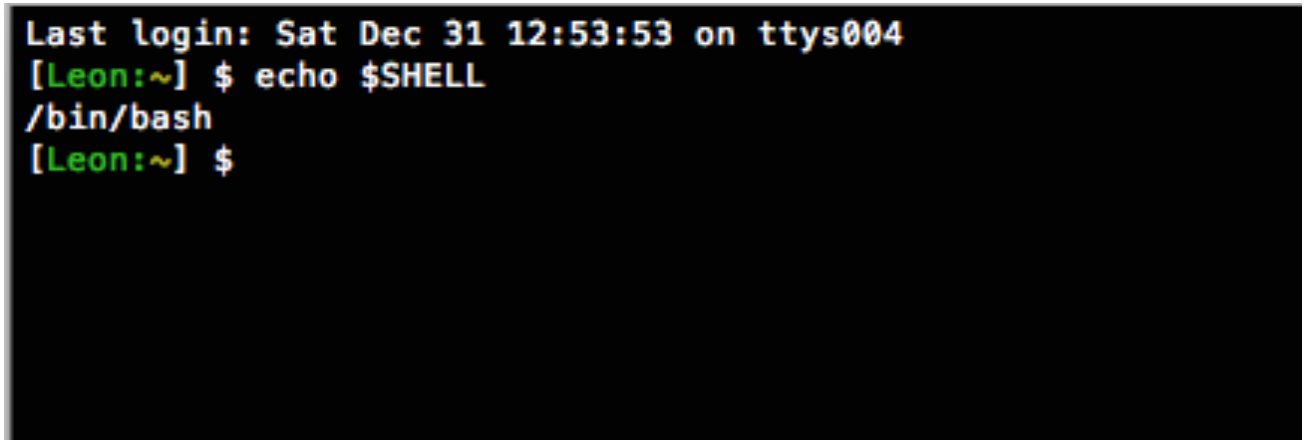
Getting Started with Linux

- **Terminal** is how we interact with the kernel.



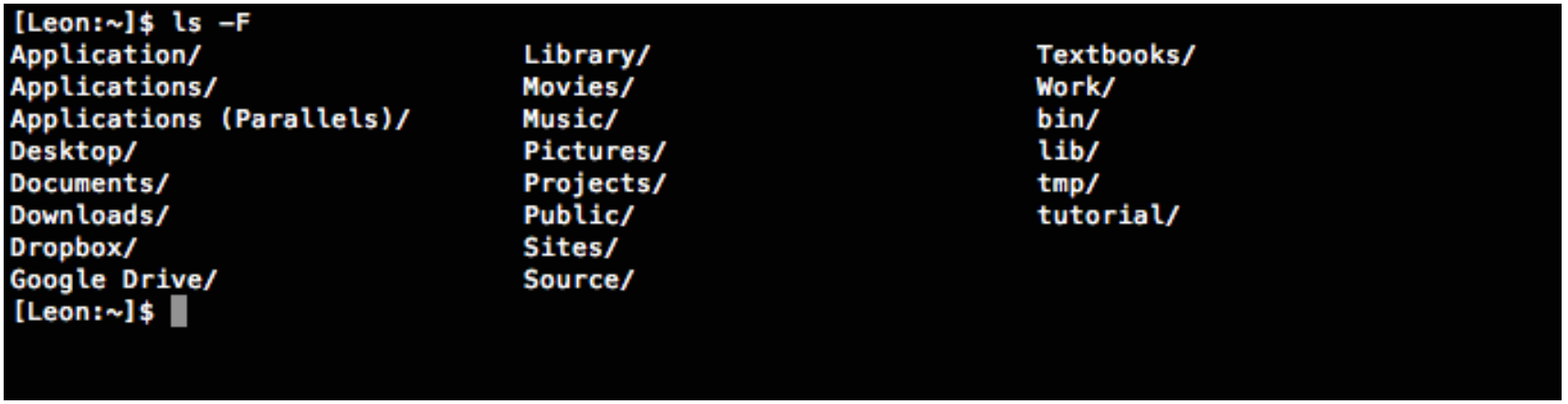
Getting Started with Linux

- When you are using a terminal you interact with the kernel via shell.
- It is good to know which shell you are using.



Getting Started with Linux

- Command in Linux can be executed after pressing ENTER. This is when the commands are needed to shell and being executed.



Getting Started with Linux

- The outcome of the command can be modified by optional tags.

```
[Leon:~]$ ls -l
total 0
drwxr-xr-x  16 Teeraparb  staff   544B Nov  6 00:26 Application/
drwxr-xr-x   6 Teeraparb  staff   204B Jan 10  2016 Applications/
drwxr-xr-x@  6 Teeraparb  staff   204B Mar  7  2016 Applications (Parallels)/
drwx------+ 25 Teeraparb  staff   850B Jan  9 06:23 Desktop/
drwx------+ 25 Teeraparb  staff   850B Nov 15 10:00 Documents/
drwx------+ 31 Teeraparb  staff   1.0K Jan  1 18:27 Downloads/
drwx------@ 19 Teeraparb  staff   646B Dec 26 09:59 Dropbox/
drwx------@ 13 Teeraparb  staff   442B Dec 26 09:59 Google Drive/
drwx------@ 71 Teeraparb  staff   2.4K Jun  1  2016 Library/
drwx------+ 11 Teeraparb  staff   374B Jun  1  2016 Movies/
drwx------+  5 Teeraparb  staff   170B Mar 16  2013 Music/
drwx------+ 14 Teeraparb  staff   476B May 30  2016 Pictures/
drwx------  27 Teeraparb  staff   918B Oct 24 08:14 Projects/
drwxr-xr-x+  6 Teeraparb  staff   204B Feb  2  2016 Public/
drwxr-xr-x+  7 Teeraparb  staff   238B Apr 19  2014 Sites/
drwxr-xr-x  22 Teeraparb  staff   748B Nov 30  2015 Source/
drwxr-xr-x  10 Teeraparb  staff   340B Feb 23  2016 Textbooks/
drwxr-xr-x  11 Teeraparb  staff   374B Feb 15  2016 Work/
drwxr-xr-x   8 Teeraparb  staff   272B Jun 30  2015 bin/
drwxr-xr-x   6 Teeraparb  staff   204B Mar 13  2016 lib/
drwx------  19 Teeraparb  staff   646B May 24  2016 tmp/
drwxr-xr-x   6 Teeraparb  staff   204B Jan  8 16:17 tutorial/
[Leon:~]$
```

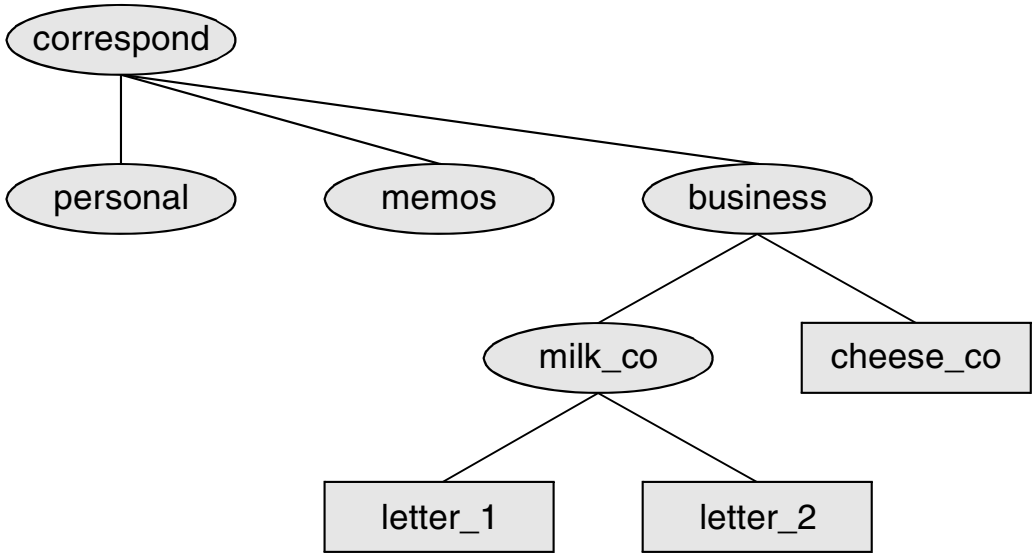
Getting Started with Linux

- **`ls`** is a Linux command that show all the file in the folder.
- **`ls -l`** is the same as **`ls`** but more details.
- **`ls -t`** is the same as **`ls`** but arranged by time.

File and Directory

- Linux arrange files and directories as a family tree.

- However, everything is Linux is a file (directories are another type of file).



File and Directory

`pwd`: Return Working Directory Name

- The `pwd` utility writes the absolute pathname of the current working directory to the standard output.

```
[Leon:~]$ pwd
/Users/Teeraparb
[Leon:~]$
```

File and Directory

`mkdir`: Create a Directory

- The `mkdir` utility creates a directory.

  ```
  mkdir tutorial
  ```

  will create a directory called `tutorial`.

- The syntax for `mkdir` is

  ```
  mkdir directory_name
  ```

File and Directory

`cd`: Change to Another Working Directory

- The `cd` utility makes another directory the working directory.
- The syntax of `cd` utility is

  `cd` *`destination_directory`*

File and Directory

Linux Special Directories

Directory	Description
.	Current working directory
..	Parent directory of the current working directory
~	Home directory

File and Directory

`cat`: Display a Text File

- The `cat` utility reads files sequentially, writing them to the standard output.

- The utility displays the contents of a text file.

```
[Leon:tutorial]$ cat practice
This is the first line.
This is the second line.
Type ctr-c to end the input.
[Leon:tutorial]$
```

File and Directory

`touch`: Create a File or Change Modification Time

- The `touch` utility changes the access and/or modification time of a file to the current time or a time you specify.

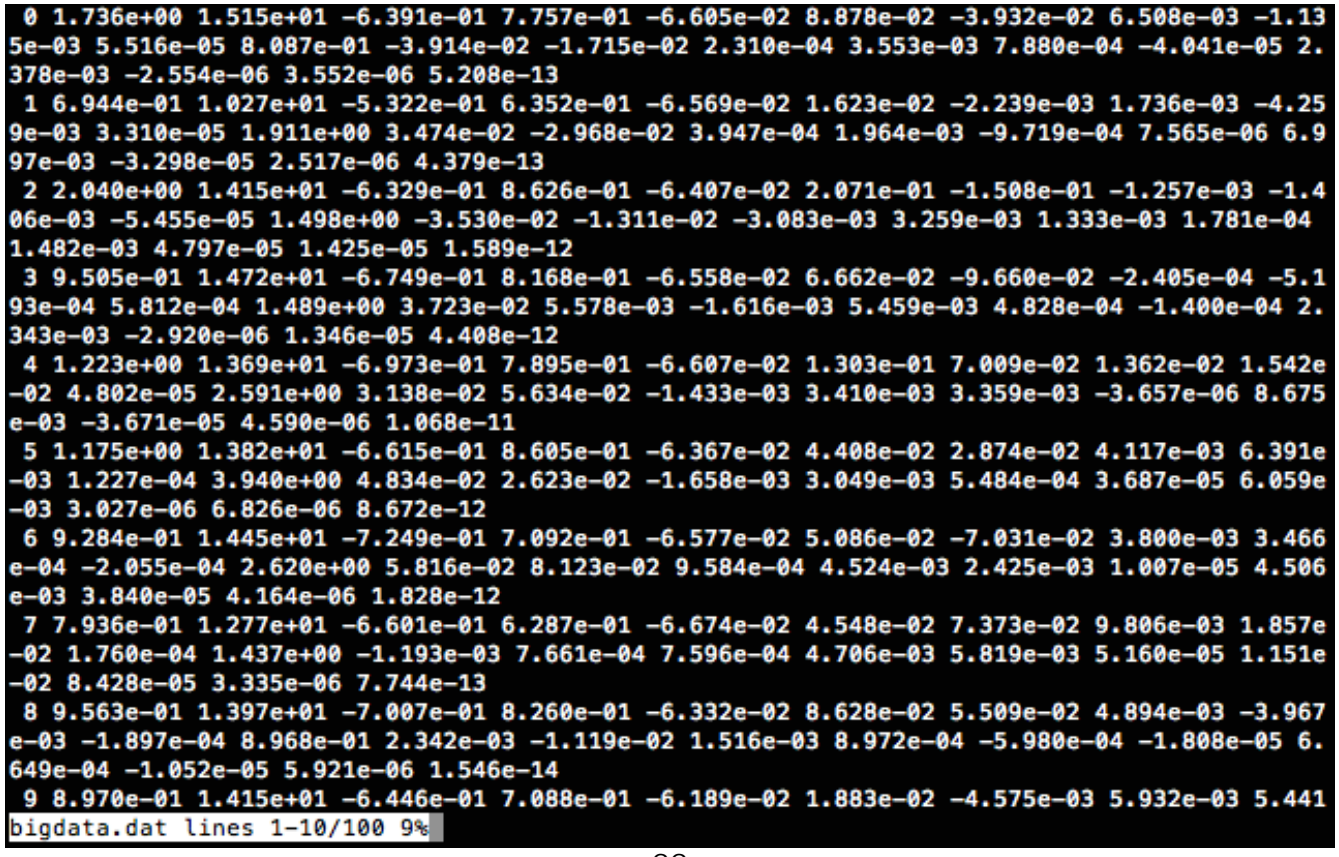
- It can also be used to create a blank file.

```
[Leon:tutorial]$ touch blankfile
[Leon:tutorial]$ ls -ltr
total 32
-rw-r--r--  1 Teeraparb  staff    78B Jan  5 12:54 practice3
-rw-r--r--  1 Teeraparb  staff    68B Jan  8 15:53 file_list.txt
-rw-r--r--  1 Teeraparb  staff    68B Jan  8 16:17 sorted_file_list.txt
-rw-r--r--  1 Teeraparb  staff    78B Jan  9 07:09 practice
-rw-r--r--  1 Teeraparb  staff     0B Jan  9 09:21 blankfile
[Leon:tutorial]$
```

File and Directory

`less`: Display a Text File One Screen at a Time

- The `less` utility display the content of the file — useful for a file that is longer than one screen.



File and Directory

`cp`: Copies Files

- The `cp` utility copies one or more files.
- **Beware!!!** `cp` can delete your file permanently by replacing an existing file.

```
[Leon:tutorial]$ cp -v practice practice3
practice -> practice3
[Leon:tutorial]$
```

File and Directory

`mv`: Changes the Name of Files

- The `mv` (move) utility can rename a file without making a copy of it.

- **Beware!!!** `mv` can also delete your file permanently by replacing an existing file.

```
[Leon:tutorial]$ mv -v practice practice2
practice -> practice2
[Leon:tutorial]$
```

File and Directory

`rm`: Deletes Files

- The `rm` (remove) utility deletes a file.
- **Beware!!!** `rm` can also delete your file permanently.

```
[Leon:tutorial]$ rm -v practice2
remove practice2? y
practice2
[Leon:tutorial]$
```

Linux Documents

`man`: Display the System Manual

- The `man` utility display `man` pages from the system documentation in a textual environment

Linux Documents

`apropos`: Searches for a Keyword

- The `apropos` searches for the keyword in the short description line (the top line) of all `man` pages and displays those that contain a match.

```
XSetCloseDownMode(3), XKillClient(3) - control clients
baudrate(3x), erasechar(3x), erasewchar(3x), has_ic(3x), has_il(3x), killchar(3x), killwcha
r(3x), longname(3x), term_attrs(3x), termattrs(3x), termname(3x) - curses environment query
 routines
xcb_kill_client(3)       - kills a client
baudrate(3x), erasechar(3x), erasewchar(3x), has_ic(3x), has_il(3x), killchar(3x), killwcha
r(3x), longname(3x), term_attrs(3x), termattrs(3x), termname(3x) - curses environment query
 routines
builtin(1), !(1), %(1), .(1), :(1), @(1), {(1), }(1), alias(1), alloc(1), bg(1), bind(1), b
indkey(1), break(1), breaksw(1), builtins(1), case(1), cd(1), chdir(1), command(1), complet
e(1), continue(1), default(1), dirs(1), do(1), done(1), echo(1), echotc(1), elif(1), else(1
), end(1), endif(1), endsw(1), esac(1), eval(1), exec(1), exit(1), export(1), false(1), fc(
1), fg(1), filetest(1), fi(1), for(1), foreach(1), getopts(1), glob(1), goto(1), hash(1), h
ashstat(1), history(1), hup(1), if(1), jobid(1), jobs(1), kill(1), limit(1), local(1), log(
1), login(1), logout(1), ls-F(1), nice(1), nohup(1), notify(1), onintr(1), popd(1), printen
v(1), pushd(1), pwd(1), read(1), readonly(1), rehash(1), repeat(1), return(1), sched(1), se
t(1), setenv(1), settc(1), setty(1), setvar(1), shift(1), source(1), stop(1), suspend(1), s
witch(1), telltc(1), test(1), then(1), time(1), times(1), trap(1), true(1), type(1), ulimit
(1), umask(1), unalias(1), uncomplete(1), unhash(1), unlimit(1), unset(1), unsetenv(1), unt
il(1), wait(1), where(1), which(1), while(1) - shell built-in commands
kill(1)                  - terminate or signal a process
kill(2)                  - send signal to a process
kill.d(1m)               - snoop process signals as they occur. Uses DTrace
killall(1)               - kill processes by name
killpg(2)                - send signal to a process group
pgrep(1), pkill(1)       - find or signal processes by name
pthread_kill(2)          - send a signal to a specified thread
XSetCloseDownMode(3), XKillClient(3) - control clients
:
```

I/O Redirection

- In some case, we may want to **redirect the output or input** to **other devices.**

- Devices could be screen monitor (default), file, printer, remote machine etc.

I/O Redirection

Standard Output

- We can redirect the output to other devices using > and >> command.
- If the output is a file > will create a new file and **destroy** the existing file.
- If the output is a file >> will append the existing file **without destroying** the file.

I/O Redirection

Standard Output

- The > command allows the input from other channel. For example,

```
grep -i "A" myfile > selectedline
```

will select lines in `myfile` with "A" to a new file called `selectedline`.

I/O Redirection

Standard Input

- The < command allows the input from another channel. For example,

```
grep -i "A" < myfile >
selectedline
```

will select lines in `myfile` with "A" to a new file called `selectedline`.

I/O Redirection

Pipelines

- Processes in Linux can be chained together as the output of a process and be an input of another process.

  ```
  cat myfile | grep -i "A"
  ```

  This program will send the content of `myfile` to `grep` command.

Summary of Linux Commands

Files and Directories

Command	Description
`ls`	List directory contents.
`pwd`	Display the pathname for the current directory.
`mkdir`	Create a new directory.
`cd`	Change directory.
`cat`	Display file's contents to the standard output device.
`touch`	Create an empty file with the specified name.

Summary of Linux Commands

Files and Directories

Command	Description
`less`	View the contents of a file one page at a time.
`cp`	Copy files and directories.
`mv`	Rename or move file(s) or directories.
`rm`	Remove (delete) file(s) and/or directories.
`man`	Display the help information for a command.
`apropos`	Search the database for strings.

Summary of Linux Commands

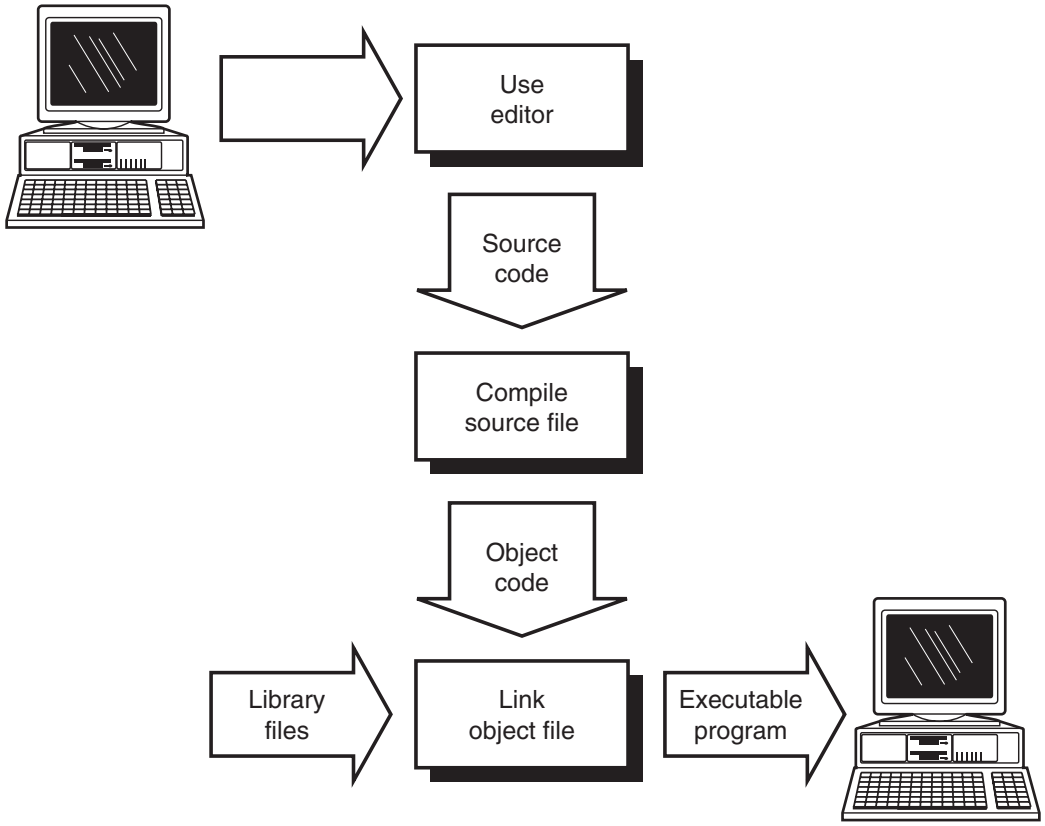
I/O Redirection

Command	Description
>	Redirect the output to other file, creating a new file.
>>	Redirect the output to other file, appended if exists.
<	Allow input from other file.
|	Chain the processes together.

Introduction to C Programming

Programming Development Cycle

- **Edit** the source file (.c)
- **Compile** the source file to get the object file (.o or .obj)
- **Link** the object files to get an executable file (.exe)
- **Run** / **Test** / **Debug**



Programming Development Cycle

Text Editor

- Any text editor that can save an ASCII file is OK.
- Most commonly used text editors are EMACS and VI or VIM.
- For Windows, Notepad is OK, but no syntax coloring.

Programming Development Cycle

Compiling

- You need a **compiler** to compiler your source files.
- There are many compilers for C. For example, GNU Project **gcc**, Intel **icc**.
- Object files compiled from different compilers are **not the same!**

Programming Development Cycle

Linking

- After getting all the object files, the **linker** will combine all the object files to get the **executable**.
- The linker will also search for the necessary **libraries**.

Getting Started with C

Example Program

```
/* This program prints a one-line message */

#include <stdio.h>

int main()
    {
    printf("Hello World\n");

    return 0;
    }
```

- Comment are in between /* */
- The **main program** is called main() — only one.
- Return **integer** to the Operating System.

Getting Started with C

Example Program

```
/* This program prints a one-line message */

#include <stdio.h>

int main()
    {
    printf("Hello World\n");

    return 0;
    }
```

- Return an **integer** to the Operating System by return command.

- #include add library to the file.

Getting Started with C

Example Program

```
/* This program prints a one-line message */

#include <stdio.h>

int main()
    {
    printf("Hello World\n");

    return 0;
    }
```

- stdio.h is the standard input and output header.
- printf() is a library function from stdio.h.
- Semi-colon (;) is used to end a **statement**.

Getting Started with C

Example Program

```
int       miles, yards;       /* global variables */
main()
{
float     kilometres;     /* local variables */
```

- In C programming language, all the variables have to be defined — **static typing**.
- Global variables are defined in main() or module().

Types of Variable in C

Main Types

Declaration	Description
`int`	Integers.
`float`	Real number.
`double`	Real number double precision.
`char`	Character.

Types of Variable in C

Integer Types

Declaration	Description
`short int`	-128 ⇨ 127 (1 byte)
`unsigned short int`	0 ⇨ 255 (1 byte)
`unsigned int`	0 ⇨ 65,535 (2 byte)
`long int`	-2,147,483,648 ⇨ 2,147,483,648
`long unsigned int`	0 ⇨ 4,294,967,295

Types of Variable in C

Real Number Types

Declaration	Description
`float`	single precision floating point (4 byte)
`double`	double precision floating point (8 byte)
`long double`	longer double precision floating point (10 byte)

Types of Variable in C

Character Types

Declaration	Description
`char`	0 ⇨ 255 (1 byte)
`unsigned char`	0 ⇨ 255 (1 byte)
`signed char`	-128 ⇨ 127 (1 byte)

Assignment Operators

`=`	assign
`+=`	assign with add
`-=`	assign with subtract
`*=`	assign with multiply
`/=`	assign with divide
`%=`	assign with remainder
`>>=`	assign with right shift
`<<=`	assign with left shift
`&=`	assign with bitwise *AND*
`^=`	assign with bitwise *XOR*
`|=`	assign with bitwise *OR*

Assignment Operators

Example Program

```
int main()
   {
   double   x = 1.23, y;
   int      i;

   y = i = x;
   printf("%f\n", y);
   return 0;
   }
```

- **Assignment** (=) does **not** mean "**equal to**".
- Assignment can be in the variable declaration.

Assignment Operators

Example Program

```
a = a + 17;
a += 17;
```

```
a = b = c = d = e = 0;
```

- The **modifiable variable** must be on the **left** hand side.
- Multiple assignments is possible. However, all the variable are **independent**.

Assignment Operators

`*`	multiplication
`/`	division
`%`	remainder after division (modulo arithmetic)
`+`	addition
`-`	subtraction and unary minus

- Arithmetic Operators have **precedence** similar to normal arithmetics.
- **Parentheses** are used to change the precedence of the operations.

Increment / Decrement Operators

`++`	increment
`--`	decrement

```
b = 3;
a = b++ + 6;          /* a = 9,  b = 4   */

b = 3;
a = ++b + 6;          /* a = 10, b = 4   */
```

- Both increment and decrement operators can be used as a **prefix operator** or **postfix operator.**

Cast Operators

`(float) sum;`	converts type to `float`
`(int) fred;`	converts type to `int`

- Cast operators allow the conversion of a value of one type to another.

Bitwise Operators

Operator	Operation
~	one's complement
&	bitwise AND
^	bitwise XOR
|	bitwise OR
<<	left shift (binary multiply by 2)
>>	right shift (binary divide by 2)

Bitwise Operators

Operator	Operand	Result
~	0010111	1101000
<<	0010111	0101110
>>	0010111	0001011

AND	XOR	OR
0 & 0 = 0	0 ^ 0 = 0	0 | 0 = 0
1 & 0 = 0	1 ^ 0 = 1	1 | 0 = 1
0 & 1 = 0	0 ^ 1 = 1	0 | 1 = 1
1 & 1 = 1	1 ^ 1 = 0	1 | 1 = 1

Input / Output

Formatted Output

- We used the command printf() function to get a formatted output.

```
printf("%f km per hour\n", kilometres);
```

- Use **format specifier** and **escape characters**.

Format Specifiers

- The format specifiers are in the form

- `%` is mandatory for every format specifiers.

- `+` can be

`-`	print at left of output field
`+`	print number with a sign (even if positive)
space	print a space if the first character to be printed is not a sign character
`0`	pad with leading zeros

Format Specifiers

- The format specifiers are in the form

```
%+D.dX
```

- `D` is the width of the string.
- `d` can be number of decimals places.
- `X` is the type of characters to be printed.

Format Specifiers

- The format specifiers are in the form

```
%+D.dX
```

Character	Form of output	Expected argument type
`c`	character	`int`
`d` or `i`	decimal integer	`int`
`x`	hexadecimal integer	`int`
`o`	octal integer	`int`
`u`	unsigned integer	`int`
`e`	scientific notation floating point	`double`
`f`	“normal” notation floating point	`double`
`g`	e or f format, whichever is shorter	`double`
`s`	string	pointer to `char`
`p`	address format (depends on system)	pointer

Format Specifiers

Statement	Output	Comment

`char a;`
`a='a';`
`printf("what %c day",a);`
	`what a day`	variable printed at position of the conversion specifier

`printf("Pi=%f:",PI);`	`Pi=3.142857:`	print `Pi`
`printf("%-10f",PI);`	`Pi=3.142857 :`	print `Pi` in field width of 10, left aligned.
`printf("%10f",PI);`	`Pi= 3.142857:`	print `Pi` in field width of 10, right aligned.
`printf("Pi=%e:",Pi);`	`Pi=3.142857e+00`	print `Pi` in scientific notation
`printf("%06.2f",PI);`	`Pi=003.14:`	print `Pi` in field width of `6` with `2` decimal places and padded with leading zeros

Escape Character

Escape sequence	Meaning
`\a`	alert (bell)
`\b`	backspace
`\f`	formfeed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\v`	vertical tab
`\\`	backslash
`\?`	question mark
`\'`	quote
`\"`	double quote
`\`*`ooo`*	character specified as an octal number
`\x`*`hh`*	character specified in hexadecimal

Input / Output

Formatted Input

- We used the command `scanf()` function to interpret the input.

`scanf("%d", &x);`	read a decimal integer from the keyboard and store the value in the memory address of the variable `x`

- The input to `scanf` is the address of the input using the address operator (`&`).

Input / Output

Formatted Input

Character	Form of output	Expected argument type
c	character	pointer to `char`
d	decimal integer	pointer to `int`
x	hexadecimal integer	pointer to `int`
o	octal integer	pointer to `int`
u	unsigned integer	pointer to `int`
i	integer	pointer to `int`
e	floating point number	pointer to `float`
f	floating point number	pointer to `float`
g	floating point number	pointer to `float`
s	string	pointer to `char`
p	address format, depends on system	pointer to `void`

Flow Controls

- **Flow controls** allows you to change the direction of the program according the certain **conditions** or **value of parameters**.

- The **relational operators** and **logical operators** are used to change the flow.

Rational Operators

Operator	Meaning	Precedence
>	greater than	2
>=	greater than OR equal	2
<	less than	2
<=	less than OR equal	2
==	equal	1
!=	not equal	1

- Relational operators also have **precedence**.
- The higher the number is the higher the precedence.

Logical Operators

&&	AND
||	OR
!	NOT

- Logical operators also have **precedence**.
- The logical operator NOT has the highest precedence.

If-Condition

Simple If-statement

```
if (expression)    /* if expression is true     */
   statement1;     /*     do statement1         */
else               /* otherwise                 */
   statement2;     /*     do statement2         */
```

Nested If-statement

```
if (a < b)
   if (c < d)
      statement1;
   else
      statement2;
```

If-Condition

Nested If-statement

```
if (a < b)
    {
    if (c < d)
        statement1;
    }
else
    statement2;
```

Switch - Selection

```
switch ( expression)
   {
   case value : statement; statement; ...
   case value : statement; statement; ...
   .
   .
   default :  statement; statement; ...
   }
```

Switch - Selection

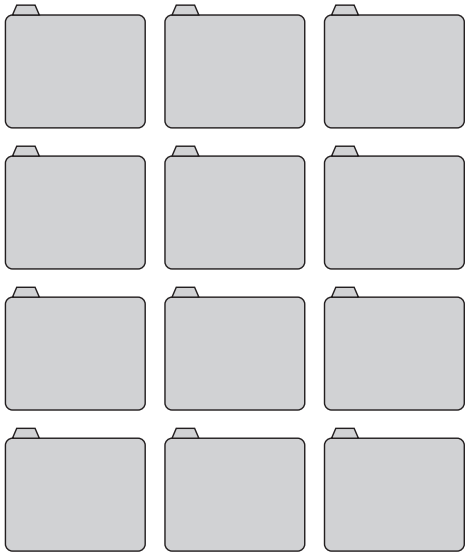
```
int main()
   {
   int    i;

   printf("Enter an integer: ");
   scanf("%d",&i);

   switch(i)
      {
      case 4:  printf("four ");
      case 3:  printf("three ");
      case 2:  printf("two ");
      case 1:  printf("one ");
      default: printf("something else!");
      }

   return 0;
   }
```

Iterations

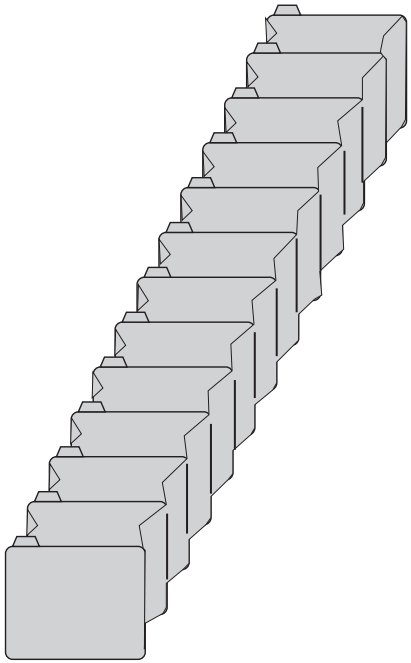
```
while (expression)          /* while expression is true do*/
   statement;               /*    statement
   */

do                          /* do                          */
   statement;               /*    statement                */
while (expression);         /* while expression is true
   */

for (expr1; expr2; expr3)
   statement;

expr1;                      /* equivalent (almost) to above */
while (expr2)               /* for loop                     */
   {
   statement;
   expr3;
   }
```

Array Variables in C

- Arrays in C are **collections of data** and arrange in order which can be accessed by specifying the order.



Individual variables



An array

Declaring Array Variables

```
int       x[3];
```

- The **dimension** of the arrays is defined using **square brackets**.
- The type of the elements of the arrays will be the same (**homogeneous-type declaration**).

Declaring Array Variables

```
double    matrix[10][10];
```

- The 2-dimensional array can be declared by adding more number of square brackets.
- Array in C starting from 0.

```
x[0] = 74;
printf("%d\n", x[2]);
```

- Assignments are done to the element, not the entire array.

Initializing Arrays

```
int       ndigit[10]   = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };
char   greeting1[]  = "hello";
char   greeting2[]  = { 'h', 'e', 'l', 'l', 'o', '\0' };
```

- The arrays can be initialized by specified each element of the arrays by braces ({ }).
- **Implicit** dimension declaration is possible.
- Strings are an array of char.
- Strings can be initialized in two different ways.

Declaring Array Variables

```
int    x[3][2]   = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
```

- For multidimensional array, **nested braces** are needed.
- The first level of the brace is the first dimension, …

Functions

- **Functions** are building blocks of the C programming language — **modular programming**.
- The idea of functions is divide **and conquer**.
- Minimizing code **repetitions**.
- `main()` is also a function which call other functions.

Functions

```
double minimum(double x, double y)
   {
   if (x < y)
      return x;
   else
      return y;
   }
```

- Functions require you to specify **return type** and **return value**.
- The function must return something. If there is noting to return use void.

Functions

```
double minimum(double x, double y)
   {
   if (x < y)
      return x;
   else
      return y;
   }
```

- Functions require **input arguments**.
- ANSI C standard require you to **explicitly specified** the type of the input arguments.
- The order of argument is important.

Functions

```
minimum(a, b*2);
```

- The argument of the **function** can be an expression.
- The value of the function can be passed to the function either **by value** or **by reference**.
- Passing by reference allows you to change the value of the variable.

Functions

```
int max(int a, int b, int *c);

int main()
   {
   int x = 4, y = 5, z;
   max(x, y, &z);          /* generate a pointer to z */
   return 0;
   }

int max(int a, int b, int *c)
   {
   if (a > b)
      *c = a;              /* *c modifies the variable    */
   else                    /* whose address was passed    */
      *c = b;              /* to the function             */
   }
```

Function Declaration & Definition

```
/* declaration of minimum() */
double minimum(double x, double y);
```

- The ANSI standard you **must declare** and **define** your functions before you use them.
- The same function can be declare many times but can only be defined once.
- It is customary that you declare all the functions before the main program all in the header file.

Function Declaration & Definition

```
int main()
    {
    printf("%f\n", minimum(1.23, 4.56));
    return 0;
    }

/* definition of minimum()  */
double minimum(double x, double y)

    double       x, y;
    {
    if (x < y)
        return x;
    else
        return y;
    }
```

- The function definition could be anywhere as long as they are **after** the function declaration.

Function Definition

```
int main()
   {
   printf("%f\n", minimum(1.23, 4.56));
   return 0;
   }

/* definition of minimum()  */
double minimum(double x, double y)

   double       x, y;
   {
   if (x < y)
       return x;
   else
       return y;
   }
```

- It is customary to place the functions after the main program.

Function Prototypes

```
double minimum(double, double);
                                    /* prototype  of minimum() */

int main()
    {
    printf("%f\n", minimum(1.23, 4.56));
    return 0;
    }

double minimum(double x, double y)
                                    /* definition of minimum() */
    {
    if (x < y)
        return x;
     else
        return y;
    }
```

- The function **prototype** is an **interface** between different module.

Function Prototypes

```
double fred();          /* declaration */
double jim(void);       /* prototype */
```

- In the example, `fred` could have as many number of arguments - unspecified number of arguments.
- `jim` can have no argument.

Standard Headers

`assert.h`	assertions
`ctype.h`	character class tests
`float.h`	system limits for floating point types
`limits.h`	system limits for integral types
`math.h`	mathematical functions
`setjmp.h`	non-local jumps
`signal.h`	signals and error handling
`stdarg.h`	variable length parameter lists
`stdlib.h`	utility functions; number conversions, memory allocation, `exit` and `system`, Quick Sort
`string.h`	string functions
`time.h`	date and time functions

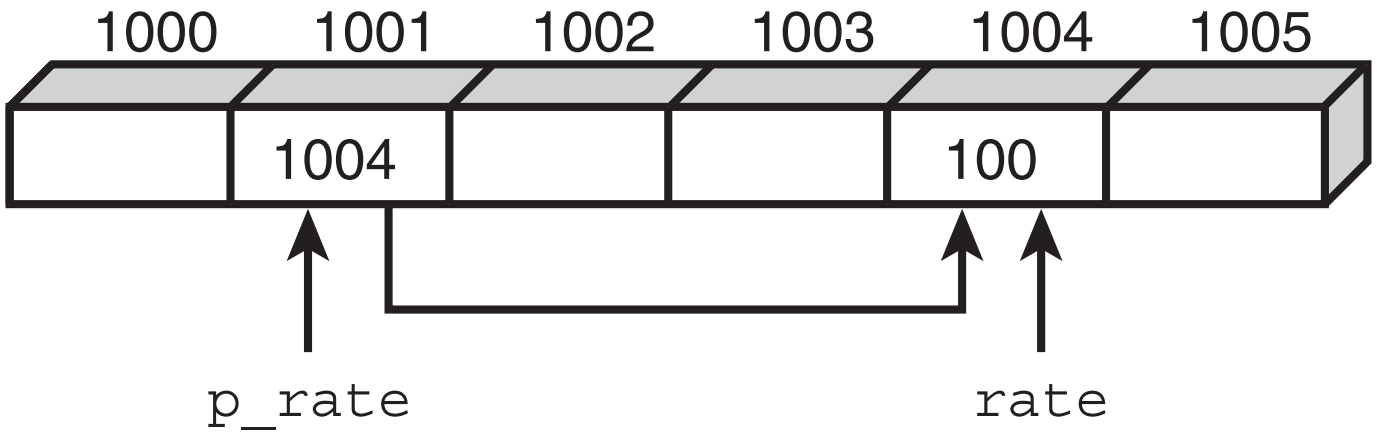
Block and Scope

```
{
int           a = 5;
printf("\n%d", a);
    {
    int       a = 2;
    printf("\n%d", a);
    }
}
```

- C is a **block structured language** — delimited by braces ({ })
- Every block has its own **local variables**.
- No semi-colon is required after the block.

Pointers

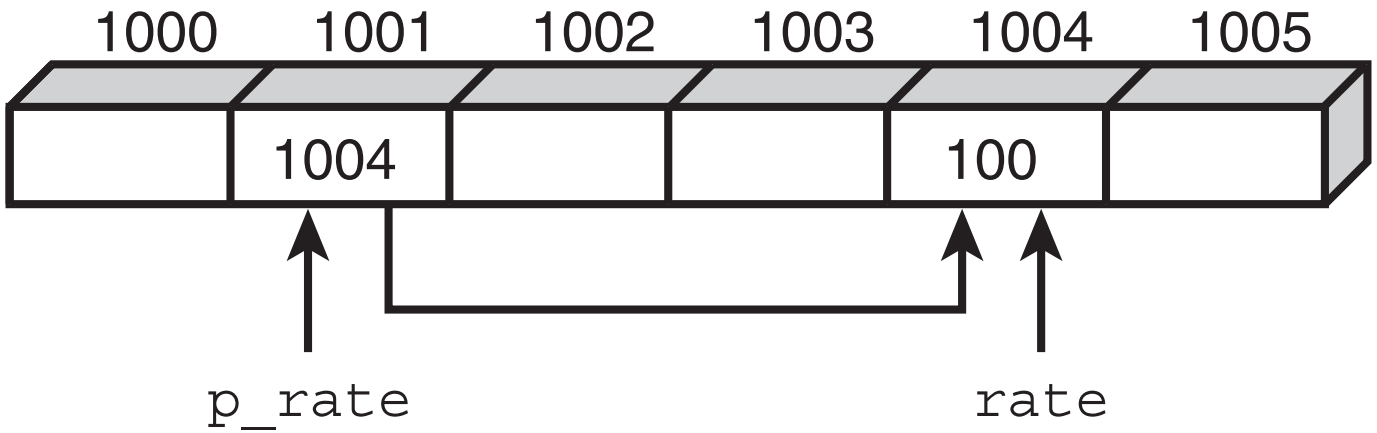
- **Pointers** are the variable that store the **address** of some variables.
- Pointers are **not integers**.



Pointers

- `&` gives the address of the memory. It generates a **pointer** to the object.

- `*` gives the value in the address (called **dereferencing** or **pointer indirection**)



Pointers

Example Program

```
main()
    {
    int i = 17;          /* ordinary int variable */
    int *pi = &i;        /* a pointer to an integer
                            initialis ed to point to i
                            pi now holds address of i */

    *pi = 25;            /* assign to what is pointed at
                            by the pointer,
                            i now contains the value 25*/

    printf("%d",*pi)     /* prints the value of what is in
                            the address pi contains e.g 25*/
    printf("%d2,pi)      /* prints the contents of pi
                            e.g 6582,    (the address of i) */
    return 0;
    }
```

Pointers & Arrays

- An array behaves as thought it is the **pointer to the first element** of the array.

- When we initialize or create an array, we automatically create a pointer.

```
int x[3] = { 23, 41, 17 };
```

Pointers & Arrays

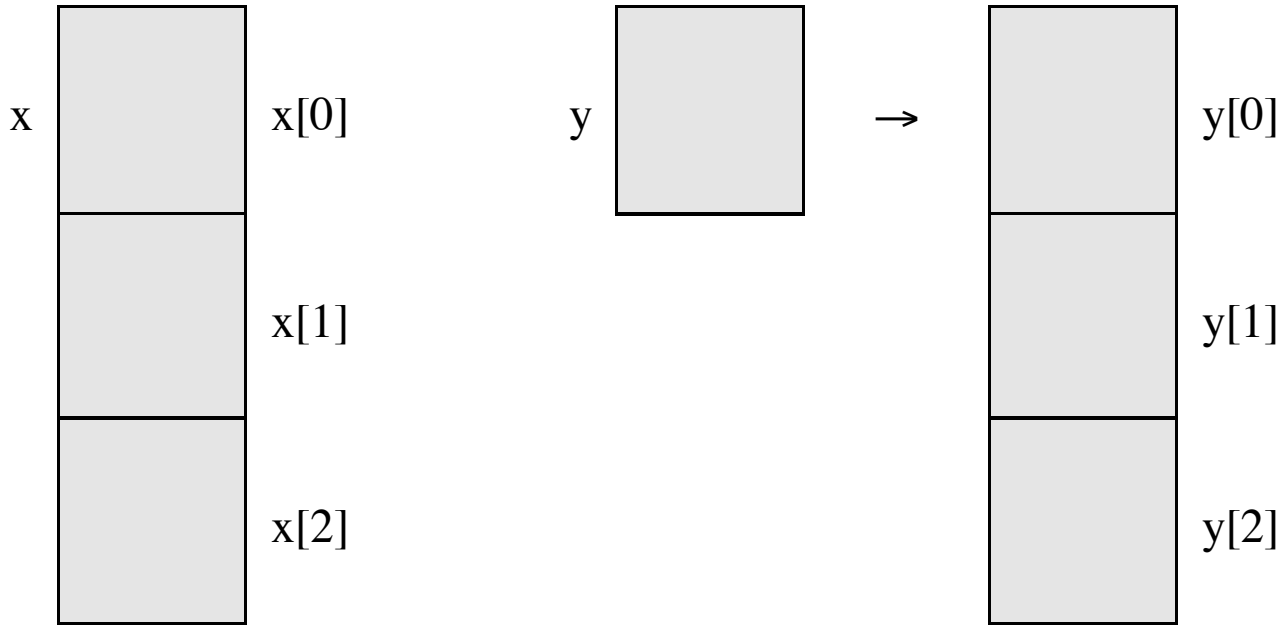
Expression	Value
`x[0]`	23
`x[1]`	41
`x`	an address, where the data is stored
`*x`	23
`x+1`	an address, where the second element is stored
`*(x+1)`	41
`x+2`	an address, where the third element is stored
`*(x+2)`	17
`&(x[0])`	`x`
`*&(x[0])`	23
`&*(x[0])`	*illegal*, the `&` and `*` do not simply "cancel" [This would be a valid expression if the array was an array of pointers!]

Dynamically Sized Array

```
int x[3];
int *y = malloc(3*sizeof(int));
```

- `malloc` (stand for memory allocation) function allows to **dynamically define** the number of elements in run-time.

Dynamically Sized Array



- The arrays define by `malloc` behave differently from **static arrays.**

Pointers & Strings

```
char      s1[50];
char      *s2 = malloc(50);     /* s2 contains the address of
                                s2[0] */
```

- Strings are arrays of characters hence the variable is also a pointer.
- There are two ways to define strings without initializing.
- By convention, the character `'\0'` define the end of the string.
- Hence the length of the string is less than the size of the array by 1.

Pointers & Strings

```
char  *s = "hello world";
```

	meaning	**type**
`&s`	address of the pointer variable `s`	pointer to pointer to `char`, (`char **`)
`s`	value of `s`, where the string `"hello world"` is stored	pointer to `char`, (`char *`)
`*s`	first character of the string (`'h'`)	`char`
`s[0]`	first character of the string (`'h'`)	`char`

Files & Pointers

```
FILE   *fp;
```

`fp` is a pointer to a file.

- There are many ways to use file in C.
- One way is to define a **FILE pointer**.
- Using `fopen` in `stdio.h` header file.

```
fp = fopen(filename, mode);
```

Files & Pointers

Fopen Mode

`"r"`	read
`"w"`	write, overwrite file if it exists
`"a"`	write, but append instead of overwrite
`"r+"`	read & write, do not destroy file if it exists
`"w+"`	read & write, but overwrite file if it exists
`"a+"`	read & write, but append instead of overwrite

Files & Pointers

- There are some restrictions on how to use each `fopen` mode.

	`r`	`w`	`a`	`r+`	`w+`	`a+`
file must exist before open	✓			✓		
old file contents discarded on open		✓			✓	
stream can be read	✓			✓	✓	✓
stream can be written		✓	✓	✓	✓	✓
stream can be written only at end			✓			✓

- `fopen` will return `NULL` (void pointer) if the file failed to open.

Files & Pointers

Example Program

```
#include <stdio.h>

int main()
   {
   char  filename[80];
   FILE  *fp;

   printf("File to be opened? ");
   scanf("%79s", filename);

   fp = fopen(filename,"r");
   if (fp == NULL)
      {
      fprintf(stderr, "Unable to open file %s\n", filename);
      return 1;       /* Exit to operating system */
      }

   code that accesses the contents of the file

   return 0;
   }
```

Files & Pointers

Example Program

```
#include <stdio.h>

int main()
   {
   char  filename[80];
   FILE  *fp;

   printf("File to be opened? ");
   scanf("%79s", filename);

   fp = fopen(filename,"r");
   if (fp == NULL)
      {
      fprintf(stderr, "Unable to open file %s\n", filename);
      return 1;       /* Exit to operating system */
      }

   code that accesses the contents of the file

   return 0;
   }
```

Files & Pointers

- Sequential file access is performed with the following library functions.

`fprintf(fp, formatstring, ...)`	print to a file
`fscanf(fp, formatstring, ...)`	read from a file
`getc(fp)`	get a character from a file
`putc(c, fp)`	put a character in a file
`ungetc(c, fp)`	put a character back onto a file (only one character is guaranteed to be able to be pushed back)
`fopen(filename, mode)`	open a file
`fclose(fp)`	close a file

Program Arguments

```
int main(int argc, char *argv[])
   {
   ....
   }
```

- In many cases, we want to write a program that will accept **input argument from the command line**.
- We shall use `stdio.h` header to use `argc` and `argv` (stands for an argument count and vector).

Program Arguments

```
int main(int argc, char *argv[])
   {
   ....
   }
```

`argc`	number of arguments
`argv`	array of strings (the arguments themselves)

- For example, if the arguments are

  `fred tom dick harry`

- `argv[0]` will be the string `"fred"`
  `argv[1]` will be the string `"tom"`.

Program Arguments

Example Program

```
int main(int argc, char *argv[], char *env[])
   {
   int i;

   for (i = 0; i < argc; i++)
      printf("%s\n", argv[i]);

   for (i = 0; env[i] != NULL; i++)
      printf("%s\n", env[i]);

   return 0;
   }
```

- `env` is another vector that store the **environment variables**.

Introduction to OpenMP with C Programming

OpenMP Introduction

OpenMP is one of the most common parallel programming models in use today.

“MP” is OpenMP stands for “multiprocessing,” a term that is synonymous with shared-memory MIMD computing.

It is relatively easy to use which makes a great language to start with when learning to write parallel software.

OpenMP Introduction

OpenMP is designed for systems in which each thread or process can have access to all available memory (shared-memory)

It is an API for writing multithreaded applications based on a set of compiler directives and library routines.

OpenMP - Getting started

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

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

# pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {

    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);

} /* Hello */
```

header for openmp

compiler directive

OpenMP execution routines

See openmp_c/ex01_omp_hello.c

OpenMP - Getting started

Compile command

```
gcc -fopenmp ex01_omp_hello.c -o omp_hello
```

```
[teeraparb:openmp_c] $ gcc -fopenmp ex01_omp_hello.c -o omp_hello
[teeraparb:openmp_c] $ ./omp_hello 8
Hello from thread 0 of 8
Hello from thread 6 of 8
Hello from thread 2 of 8
Hello from thread 4 of 8
Hello from thread 1 of 8
Hello from thread 3 of 8
Hello from thread 5 of 8
Hello from thread 7 of 8
[teeraparb:openmp_c] $
```

OpenMP - Getting started

Compile command

```
gcc -fopenmp ex01_omp_hello.c -o omp_hello
```

```
[teeraparb:openmp_c] $ gcc -fopenmp ex01_omp_hello.c -o omp_hello
[teeraparb:openmp_c] $ ./omp_hello 8
Hello from thread 0 of 8
Hello from thread 6 of 8
Hello from thread 2 of 8
Hello from thread 4 of 8
Hello from thread 1 of 8
Hello from thread 3 of 8
Hello from thread 5 of 8
Hello from thread 7 of 8
[teeraparb:openmp_c] $
```

Thread are competing for the resource i.e. terminal

Non-deterministic output

OpenMP - Directives

Most of the constructions in OpenMP are compiler directives

```
#pragma omp construct [clause [clause]…]
```

Example

```
   #pragma omp parallel num_threads(4)
```

Function prototypes and types in the files:

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

Most OpenMP constructs apply to a "structured block"

OpenMP - Directives

In OpenMP parlance, the collection of threads executing the **parallel** block—the original thread and the new threads—is called a **team**. OpenMP thread terminology includes the following:

- **master**: the first thread or thread 0
- **parent**: thread that encounter parallel directive and started a team of threads.
- **child**: each thread started by the parent

When the threads return from the call—there is an **implicit barrier**.

See openmp_c/ex02_omp_hello2.c

OpenMP - Execution Environment Routines

Execution environment routines affect and monitor threads, processors, and the parallel environment. The library routines are external functions with “C” linkage.

```
int omp_get_thread_num(void)
```

Return the thread number of the calling thread within the current team.

```
int omp_get_num_threads(void)
```

Return the number of threads in the current team.

OpenMP - Execution Environment Routines

```
void omp_set_num_threads(int)
```

Affects the number of threads used for subsequent parallel regions not specifying a **num_threads** clause.

```
int omp_get_max_threads(void)
```

Affects the number of threads used for subsequent parallel regions not specifying a **num_threads** clause.

OpenMP - Error Checking

Our program does not do the error checking.

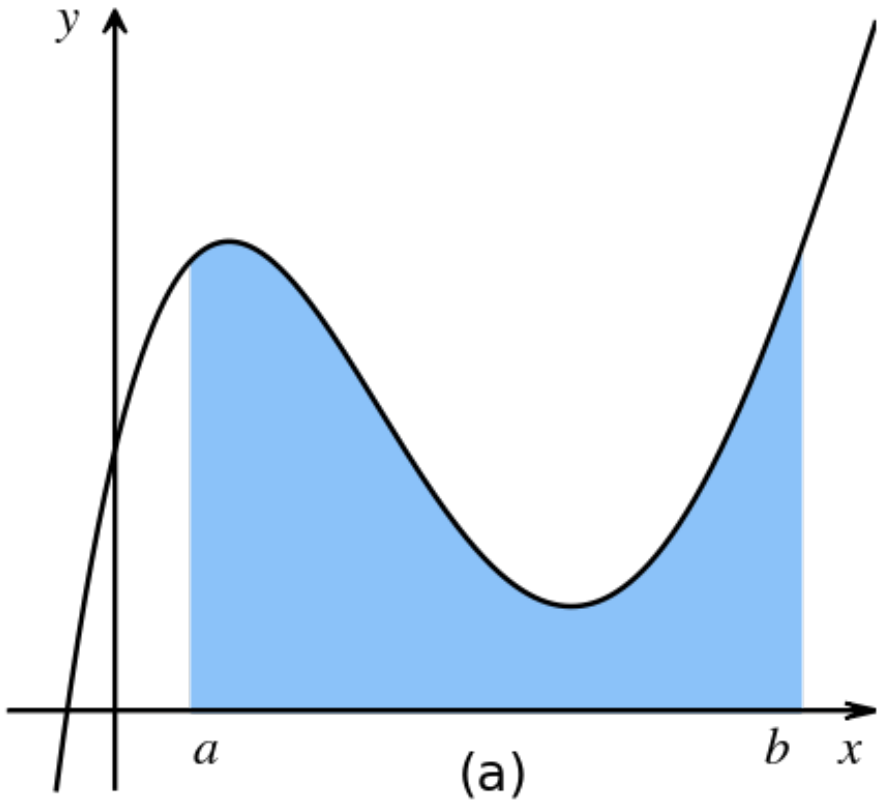
- We should check our command line arguments
- We should check whether our compiler support OpenMP. We can do this by checking whether the macro _OPENMP is defined.

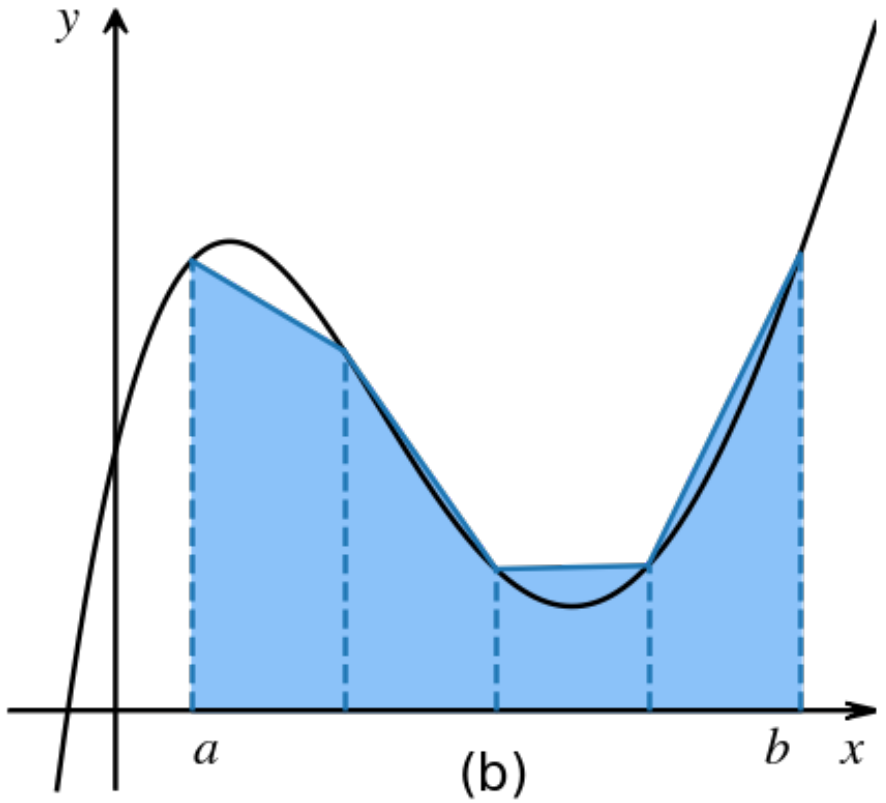
```
#ifdef _OPENMP
#include <omp.h>
#endif
```

See openmp_c/ex03_omp_hello3.c

OpenMP - Example

We will try a more realistic example by using the trapezoidal rule to integrate a function.

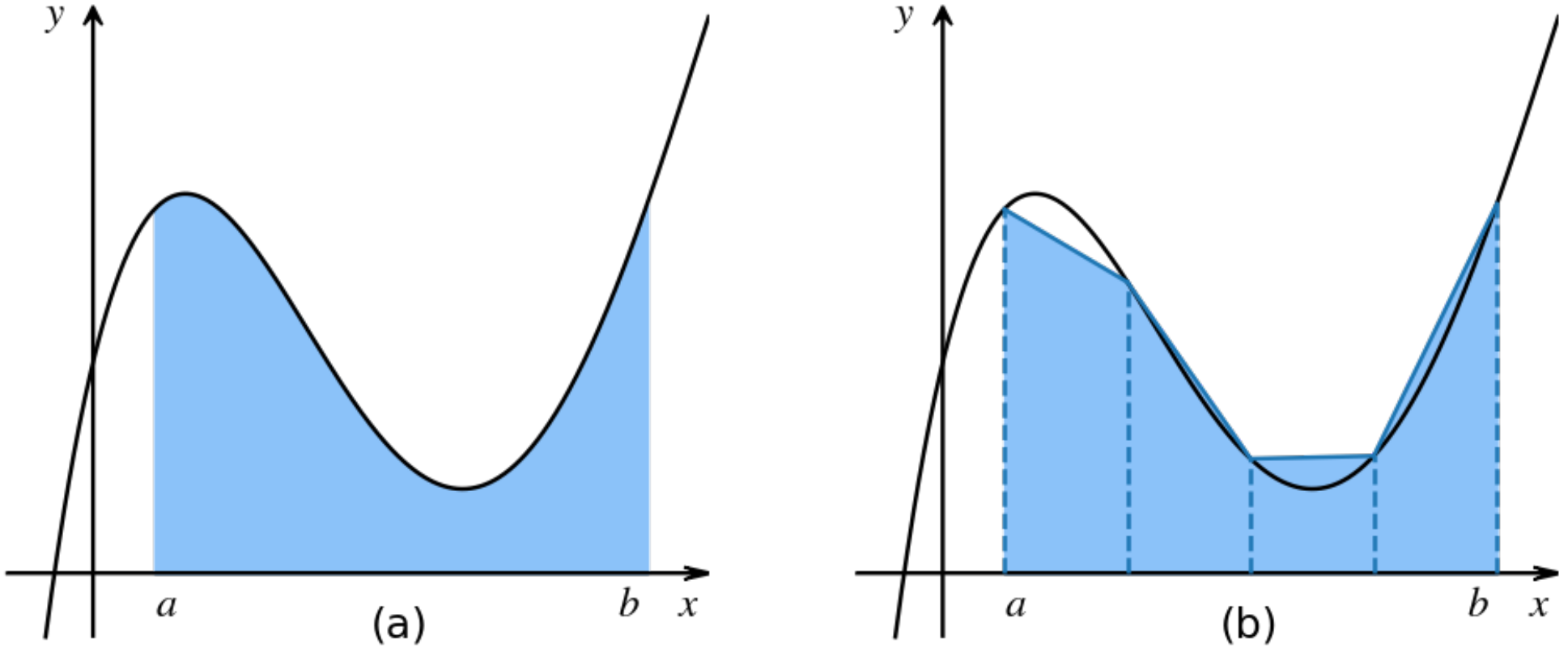




OpenMP - Example

Trapezoidal rule

$$\mathcal{I} = \frac{h}{2}\left(f(x_0) + 2\big(f(x_1) + \ldots f(x_{N-1})\big) + f(x_N) \right)$$



See openmp_c/ex04_omp_trapezoid.c

Scope of Variables

In OpenMP, the **scope** of a variable refers to the set of threads that can access the variable in a parallel block.

A variable that can be accessed by all the threads in the team has **shared** scope, while a variable that can only be accessed by a single thread has **private** scope.

In the previous example,

shared variables: `a, b, n, global_result`

private variables: `my_result, local_a, local_b`

OpenMP - Critical Section

```
#pragma omp critical
```

Restrict execution of the associated structure block to a single thread at a time.

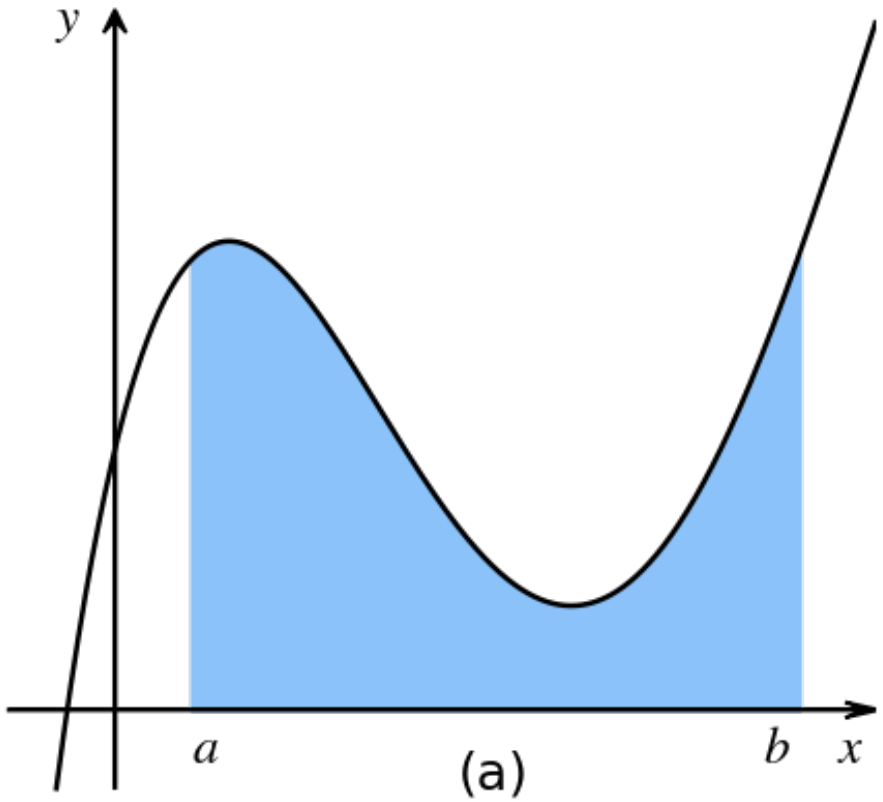
```
#pragma omp critical
    *global_result_p += my_result;
```

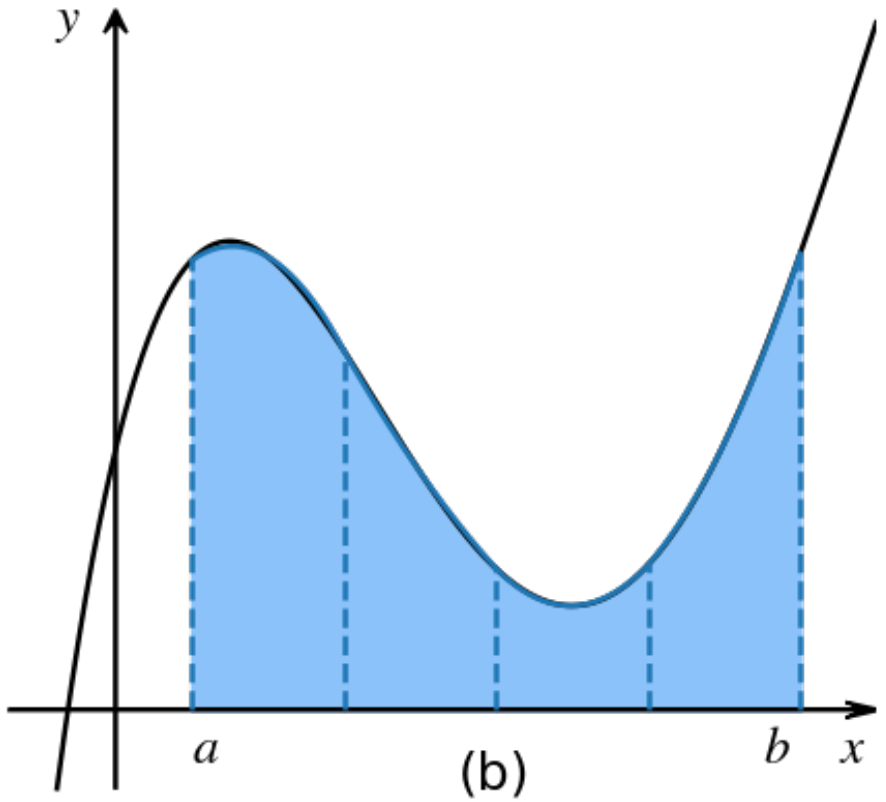
This directive tells the compiler that the system needs to arrange for the threads to have **mutually exclusive** access to the following structured block of code.

OpenMP - Simpson's rule

Simpson's rule

$$\mathcal{I} = \frac{h}{3}\left(f(x_0) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + 2\sum_{i=1}^{n/2-1} f(x_{2i})) + f(x_N)\right)$$





See openmp_c/ex05_simpson.c

OpenMP - Simpson’s rule

Exercise:

Modify the Simpson integration code to run in parallel with OpenMP

OpenMP - Reduction Clause

The **reduction** clause in OpenMP is used to perform a reduction operation on variables that are updated in parallel by multiple threads. A reduction operation combines the results from all threads into a single value.

This is particularly useful when you need to perform any associative and commutative operation on a variable across multiple threads.

The syntax of the reduction clause is

reduction(<operator>:<variable list>)

OpenMP - Reduction Clause

OpenMP supports several operations in the reduction clause

- Arithmetic: `+`, `-`, `*`, `&`, `|`, `^`
- Logical: `&&`, `||`
- Min/Max: `min`, `max`

Example

```
#pragma omp parallel num_threads(4) reduction(+: global_result)
```

See openmp_c/ex06_omp_reduction.c

OpenMP - the parallel for directive

The parallel for directive allows multiple iterations of a loop to be executed concurrently by multiple threads.

It is specifically used before the `for` structured block

Key Concepts

- **Shared and Private Variables**: By default, all variables within a parallel for block are shared among threads. However, loop variables are private to each thread unless specified otherwise.
- **Loop Scheduling**: OpenMP allows control over how iterations are distributed among threads

OpenMP - the parallel for directive

```
#pragma omp parallel for [clause [clause]…]
```

Example

```
#pragma omp parallel for collapse(2) num_threads(thread_count)
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
```

See openmp_c/ex07_omp_parallel_for.c

OpenMP - the parallel for directive

- OpenMP will only parallelize **for** loops for which the number of iterations can be determined but it will not parallelize **while** loops or **do—while** loops directly.

- OpenMP will only parallelize **for** loops that is in the **canonical** form

```
for ( index = start  ;  index < end      ;  index++
                        index <= end        ++index
                        index >= end        index-
                        index > end         -index
                                            index += incr
                                            index -= incr
                                            index = index + incr
                                            index = incr + index
                                            index = index - incr )
```

OpenMP - the parallel for directive

```
int Linear_search(int key, int A[], int n) {
    int i;
    /* thread_cound is global */
# pragma omp parallel for num_threads(thread_count)
    for (i = 0; i < n; i++)
        if (A[i] == key) return i;
    return -1; /* key not in list */
}
```

```
parallel.c: In function ‘Linear_search’:
parallel.c:29:26: error: invalid branch to/from OpenMP structured block
  if (A[i] == key) return i;
                          ^
```

This program generates error during compilation
Why?

OpenMP - Scope of a variable

- Understanding the scope of variables in OpenMP is crucial for correct and efficient parallel programming.
- Variable in **parallel for** directive can have different scopes: **shared**, **private**, **firstprivate**, **lastprivate**, and **reduction**.
- We could also set the **default** variable within the for loop.

OpenMP - Scope of a variable

Shared

- Variables declared outside the parallel region are shared by default.
- Accessible by all threads simultaneously.
- Risk: Race condition.

```
int total = 0;
#pragma omp parallel for shared(total)
for(int i = 0; i < N; i++) {
    total += array[i]; // potential race condition
}
```

OpenMP - Scope of a variable

Private

- Each thread has its own instance of the variable.
- Uninitialized by default within the parallel region.
- Private to each thread; change made by one thread are not visible to others.

```
#pragma omp parallel for private(i)
for(int i = 0; i < N; i++) {
    // i is private to each thread
    // No interference between threads
}
```

OpenMP - Scope of a variable

Firstprivate & Lastprivate

- **Firstprivate**: Initializes the private variable with the value from the outside scope.
- **Lastprivate**: Ensures that the private variable's value from the last iteration is copied back to the original variable outside the parallel region.

```
int i = 0;
#pragma omp parallel for lastprivate(i)
for(int j = 0; j < N; j++) {
    i = j * j;  // Some computation
}
printf("Final value of i: %d\n", i);
```

OpenMP - Scope of a variable

Default clause

- **Purpose**: Controls the default data-sharing attributes for variables within a parallel region.

```
#pragma omp parallel for default(kind)
```

- **shared**: All variable are shared among the threads.
- **private:** All variable are private to each thread.
- **none:** Require explicit specification for each variable.

OpenMP - Estimating π

One way to get a numerical approximation to π is to use the formula

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

Implement this formula in your code to estimate π with parallelization.

See openmp_c/ex08_omp_pi_estimate.c

OpenMP - Estimating π

Exercise

Now using Ramanujan formula to estimate the value of π.

$$\frac{1}{\pi} = \frac{2\sqrt{2}}{9801} \sum_{k=0}^{\infty} \frac{(4k)!(1103 + 26390k)}{(k!)^4 396^{4k}}$$

You may need `tgamma` function for factorial.

OpenMP - parallel & for directive

The **parallel directive** and **for directive** could be used separately

- **parallel directive**: creates a team of threads that execute the enclosed block of code.
- **for directive**: distribute loop iterations among the threads in the team created by a parallel region

See openmp_c/ex09_omp_parallel_for2.c

Nested Parallelism in OpenMP

Nested parallelism in OpenMP refers to the creation of parallel regions within other parallel regions.

- **Inner Parallelism**: Enable parallel execution within an already parallelized code section
- **Use Case**: Useful in applications where different levels of loops or tasks can be independently parallelized.

Why Nested Parallelism?

- **Increased Performance:** significantly improve the performance of applications, especially those that can be decomposed into tasks with independent subtasks.
- **Enhanced Scalability:** Nested parallelism can scale well with larger problem sizes and more processors, as it allows for efficient utilization of available resources.
- **Improved Resource Utilization:** By parallelizing both top-level and subtasks, nested parallelism can help to avoid idle resources and maximize the efficiency of parallel systems.

Consideration for Nested Parallelism

- **Overhead:** Implementing nested parallelism can introduce overhead due to task creation, synchronization, and load balancing.
- **Complexity:** Designing and implementing nested parallel algorithms can be more complex than sequential or single-level parallel algorithms.
- **Synchronization:** Careful synchronization is required to ensure correct results and avoid race conditions.

Nested Parallelism in OpenMP

nest_var is an Internal Control Variables (ICV) controls nested parallelism in OpenMP.

```
void omp_set_nested(int)
```

Enable or disable nested parallelism, by setting the **nest_var** ICV.

```
int omp_get_nested(void)
```

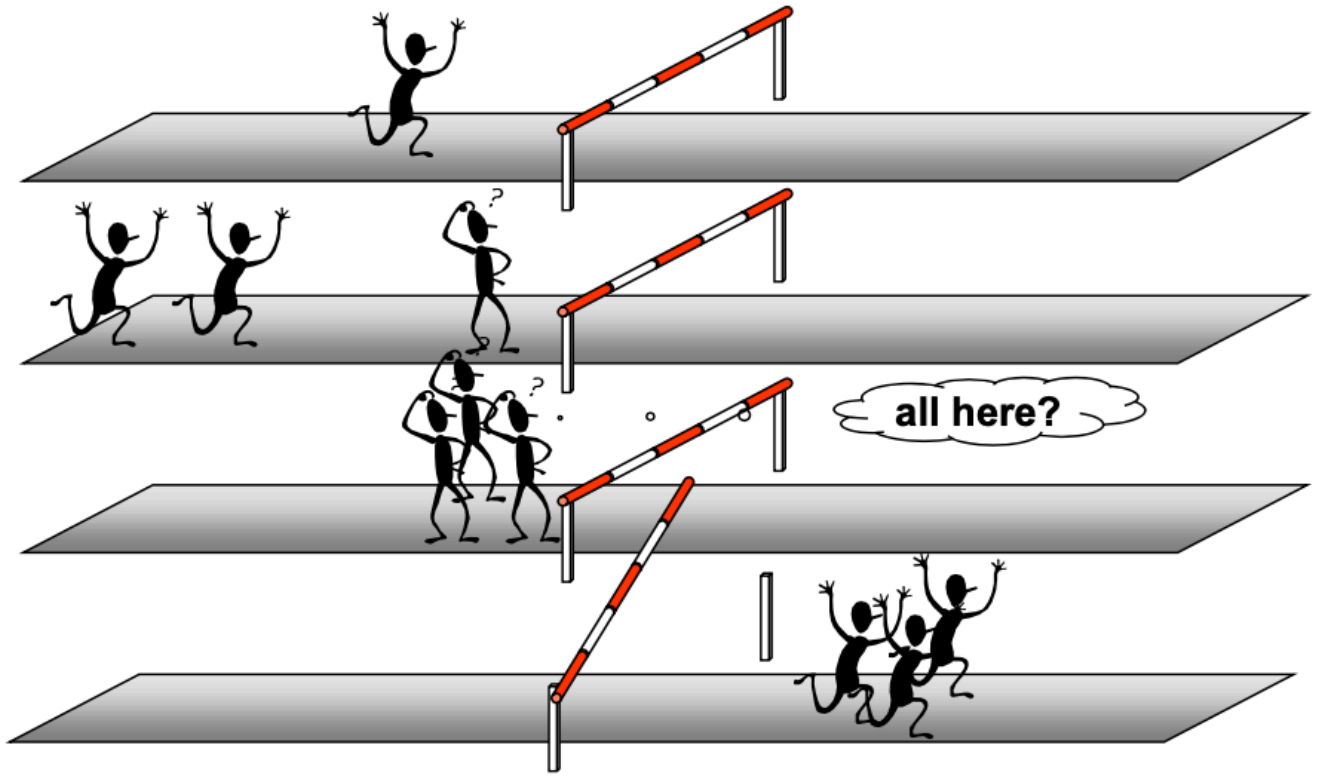
Return the value of the **nest_var** ICV, which indicates if nested parallelism is enabled or disabled.

See openmp_c/ex10_omp_nested_parallel.c

See openmp_c/ex11_omp_nested_parallel2.c

Barrier in OpenMP

In OpenMP, a barrier is a synchronization point where all threads within a parallel region must wait until all other threads have reached that point before proceeding.



Barrier in OpenMP

Example

```
#pragma omp barrier
```

Specifies an explicit barrier at the point at which the construct appears.

See openmp_c/ex12_omp_parallel_for.c

Scheduling Loops

Suppose we have a function `f(i)` where `i` is some integers which we want to parallelize the task of calculating for `i = 1, …, n`.

```
sum = 0.0;
#pragma omp parallel for num_threads(num_threads) reduction(+:sum)
for (int i = 0; i < n; i++)
    sum += f(i)
```

Normally the task will distributed equally among threads.

$$\text{loop per thread} = \frac{n}{\text{num\_threads}}$$

Scheduling Loops

However, if the computational time to compute `f(i)` is proportional to `i`. The block partitioning will assign more work to thread `num_threads` - 1.

```
#pragma omp parallel for
        schedule(kind[, chunk_size])
```

where

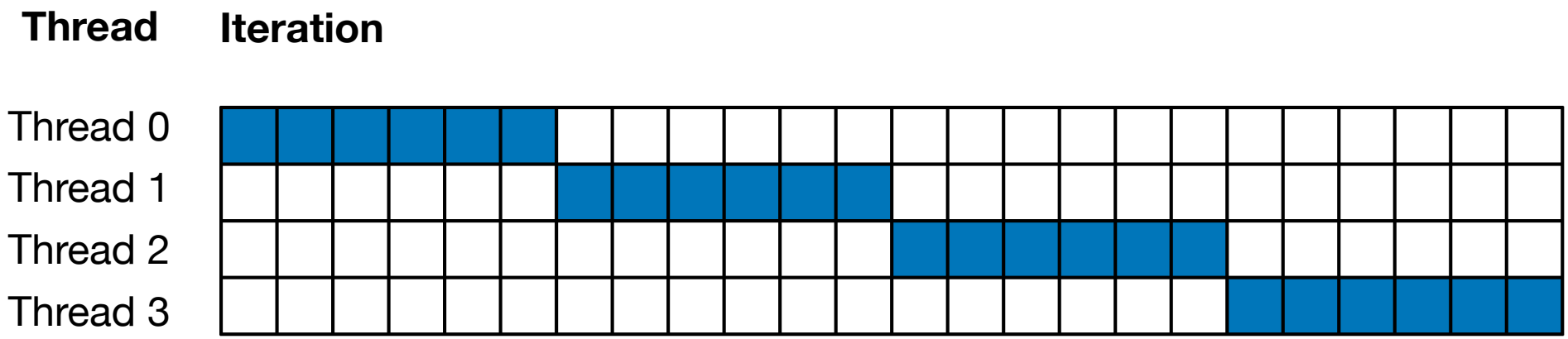
- `kind` specified the method of distributing the loop iterations
- `chunk_size` optional parameter that defines the number of iterations each thread will execute at a time.

Type of Scheduling

static

- Loop iterations are divided into chunks of a fixed size and assigned to threads in a round-robin fashion before the execution of the loop.

```
#pragma omp parallel for numthreads(4) schedule(static)
```

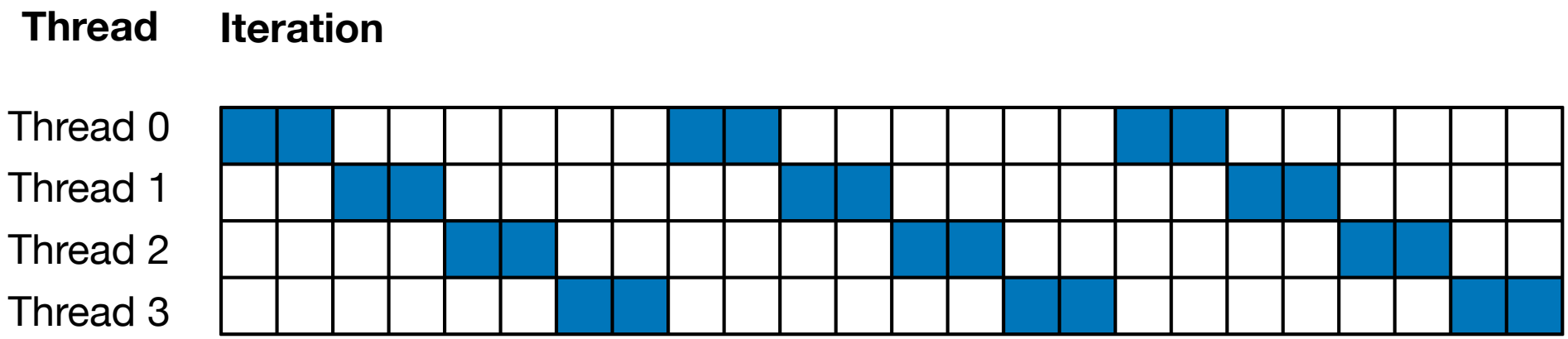


Type of Scheduling

static

- Loop iterations are divided into chunks of a fixed size and assigned to threads in a round-robin fashion before the execution of the loop.

```
#pragma omp parallel for num_threads(4) schedule(static, 2)
```

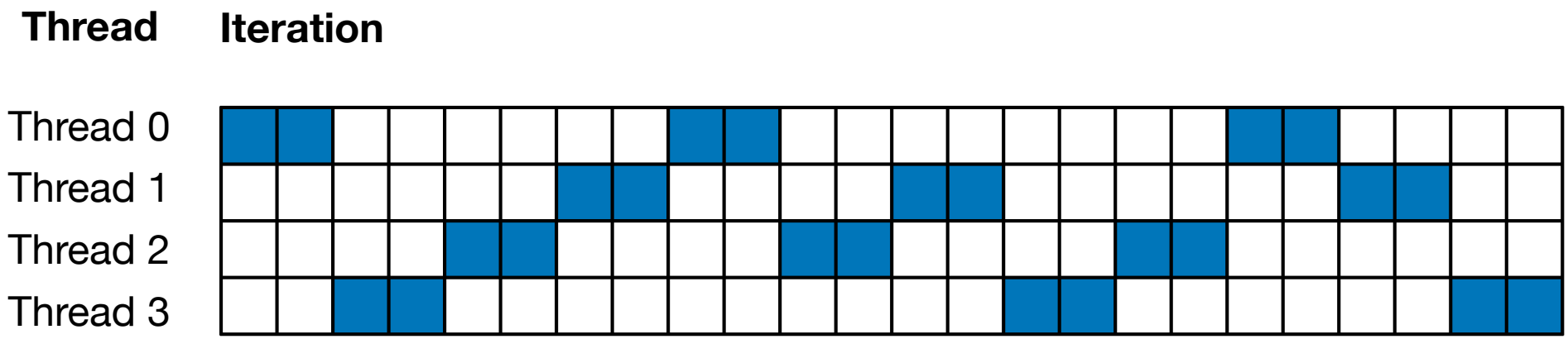


Type of Scheduling

dynamic

- Iterations are assigned to threads in chunks dynamically. Once a thread finishes its current chunk, it requests the next chunk.

```
#pragma omp parallel for num_threads(4) schedule(dynamic, 2)
```

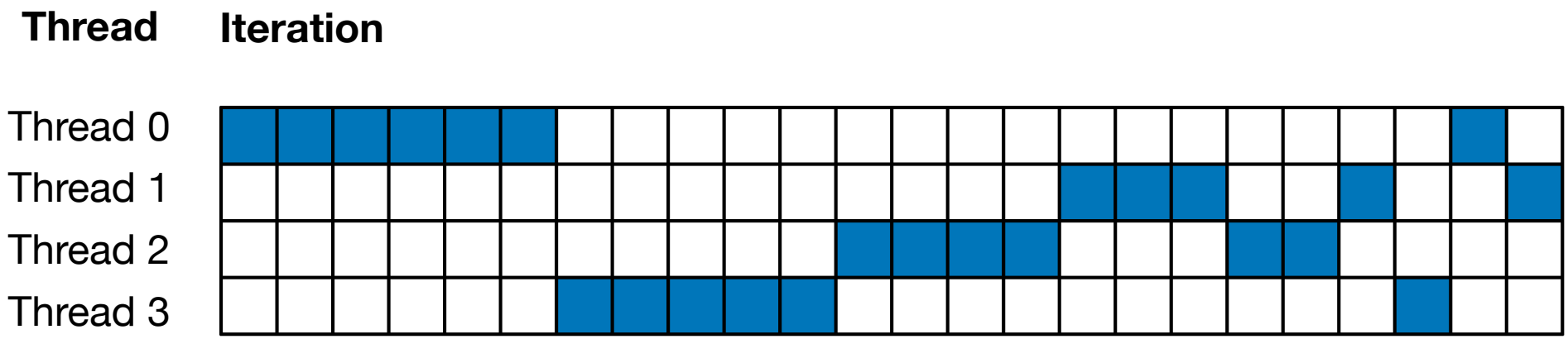


Type of Scheduling

guided

- Similar to dynamic scheduling but with a decreasing chunk size. The size of chunks starts large and reduces as the loop progresses.

```
#pragma omp parallel for num_threads(4) schedule(guided)
```



Type of Scheduling

auto

- The scheduling is left to the compiler/runtime to determine the best method based on the system and the workload.

runtime

- The scheduling method is determined at runtime based on the `OMP_SCHEDULE` environment variable.

Type of Scheduling

Choosing the Right Schedule:

- **static**: For uniform and predictable workloads.
- **dynamic**: For uneven workloads where some iterations take significantly longer.
- **guided**: For reducing overhead while still accommodating uneven workloads.
- **auto/runtime**: When optimization by the system or runtime environment is preferred.

See openmp_c/ex13_omp_schedule.c

See openmp_c/ex14_omp_schedule2.c

Parallel If Directive

What is the 'parallel if' directive?

- The 'parallel if' directive in OpenMP allows conditional parallel execution.
- The parallel region is only created if the condition specified in the if clause evaluates to true; otherwise, the code block is executed serially by the master thread.
- Provides control over parallelization to optimize performance.
- Avoids unnecessary overhead when parallel execution is not beneficial.

Parallel If Directive

Syntax

```
#pragma omp if(condition)
```

The `condition` can be any valid expression. If it evaluates to true, the code inside the block is executed in parallel by multiple threads; if false, it runs serially.

```
#pragma omp parallel if(n > 500)
{
    #pragma omp for
    for (int i = 0; i < n; i++) {
        array[i] = i * i;
    }
}
```

Parallel If Directive

Performance Implication & Consideration

- Use the 'parallel if' directive to avoid overhead in cases where the workload is too small to benefit from parallel execution.
- Be cautious about conditions that frequently switch between true and false, as this can lead to unpredictable performance.
- Keep the condition simple to avoid adding unnecessary complexity to your code.

See openmp_c/ex15_omp_parallel_if.c

Prime Number Generator

The Sieve of Eratosthenes is an ancient algorithm used to find all prime numbers up to a given limit by iteratively marking the multiples of each prime starting from 2.

Prime Number Generator

- Write a serial version of the Sieve of Eratosthenes to generate all prime numbers up to a specified limit (for example 100000).
- Modify the serial version of your program.
- You may need an array `prime[i]` which will give the value 1 if `i` is a prime number and 0 if `i` is not a prime number.
- Compare the performance of serial and parallel code.

Mandelbrot Set

- The Mandelbrot set is a complex fractal, where each point on the complex plane is iteratively checked to determine if it belongs to the set.
- The set is defined in the complex plane as the complex numbers c for which the function $f_c(z) = z^2 + c$ does not diverge to infinity when iterated starting at $z = 0$ i.e. the sequence $f_c(0), f_c(f_c(0)), \ldots$ remain bounded in absolute value $|z| \leq 2$.
- Write a program that test whether c belong to the set under the condition by setting the MAX_ITER parameter above which the would declared bounded.
- Plot the Mandelbrot set using matplotlib using image plot.

Mandelbrot Set