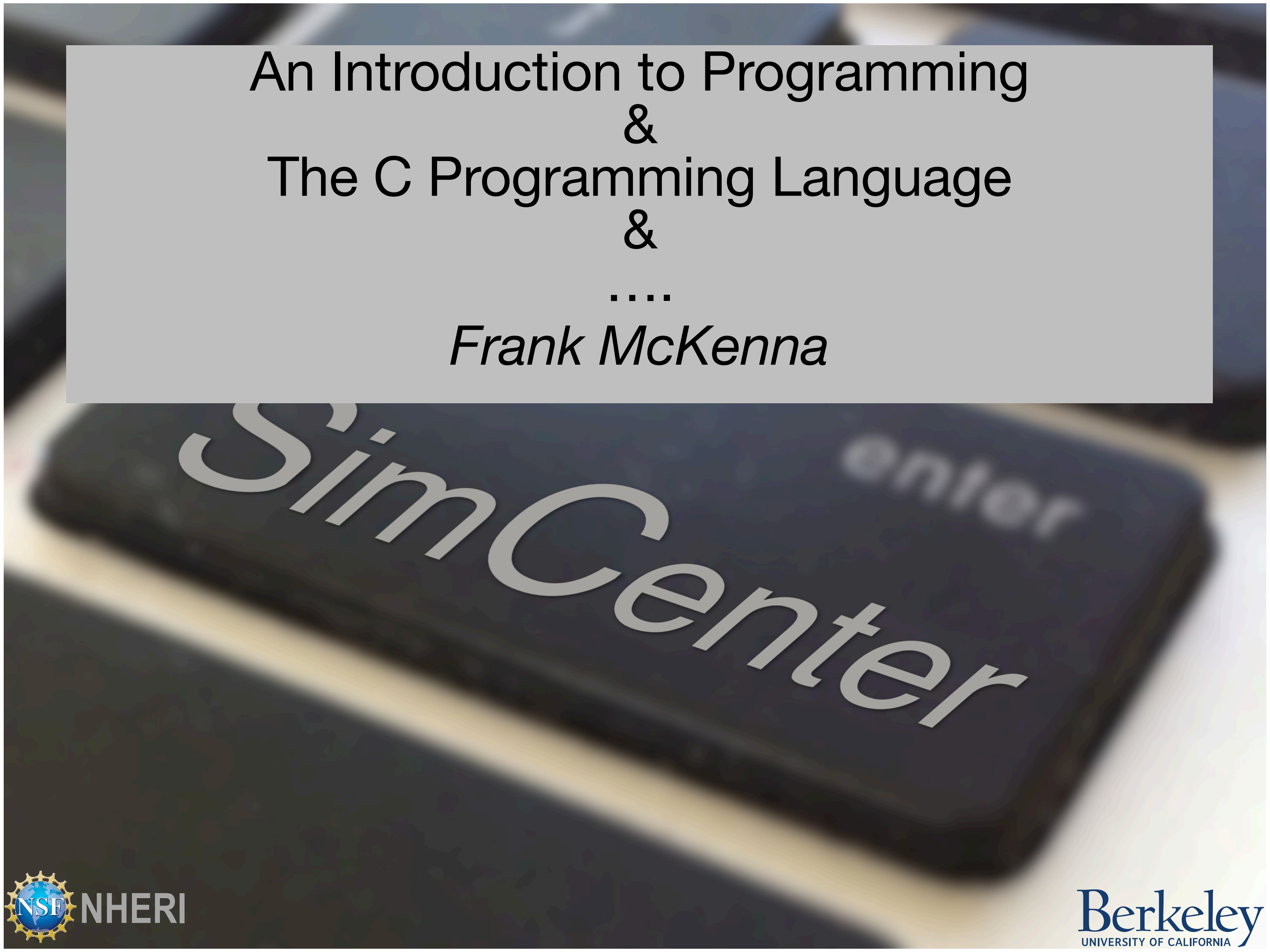
Agenda – Day 1

Time	Title	Presenter
9:30-10:00	Welcome	Frank McKenna & You
10:30-11:00	SimCenter Overview	
11:30-12:30	The VM, Linux & Git	Peter Mackenzie-Helnwein
12:30-1:30	LUNCH	Frank McKenna
1.30-2:00	An Introduction to Programming	
2:30-3:30	The C Programming Language	Frank McKenna
3:30-5:00	Exercises	You
Day 2	Abstraction, More C & C++	
Day 3	Python	
Day 4	Data Gathering & AI	
Day 5	SimCenter Regional Workflows	

SimCenter NHERI
Center for Computational Modeling and Simulation



An Introduction to Programming & The C Programming Language & ....

Frank McKenna

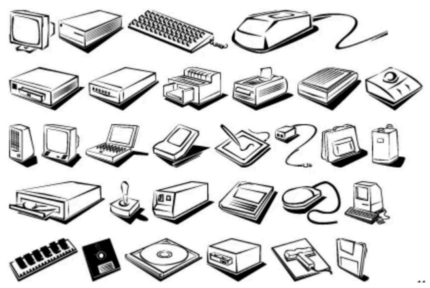
NHERI

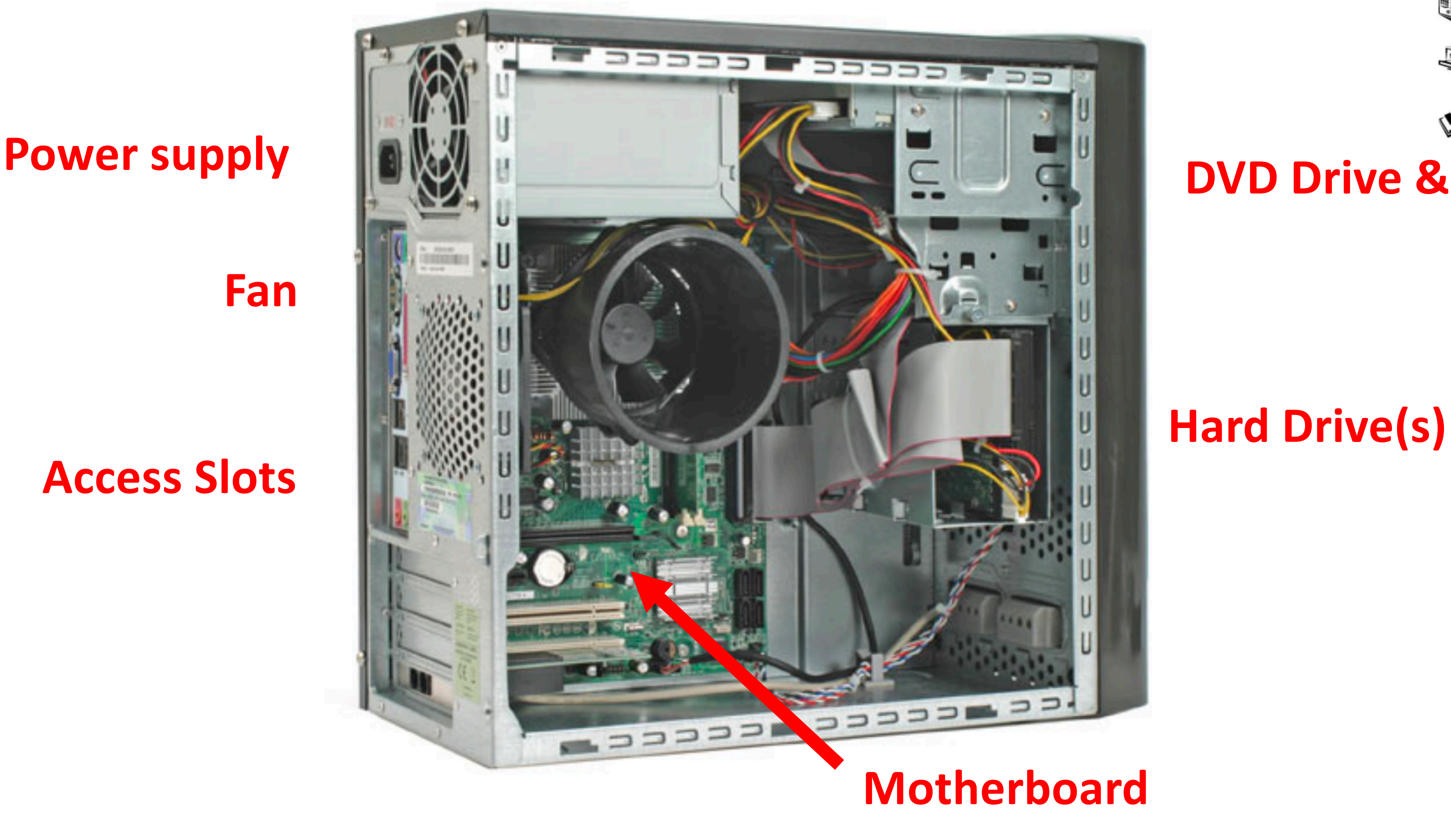
Berkeley UNIVERSITY OF CALIFORNIA

Outline

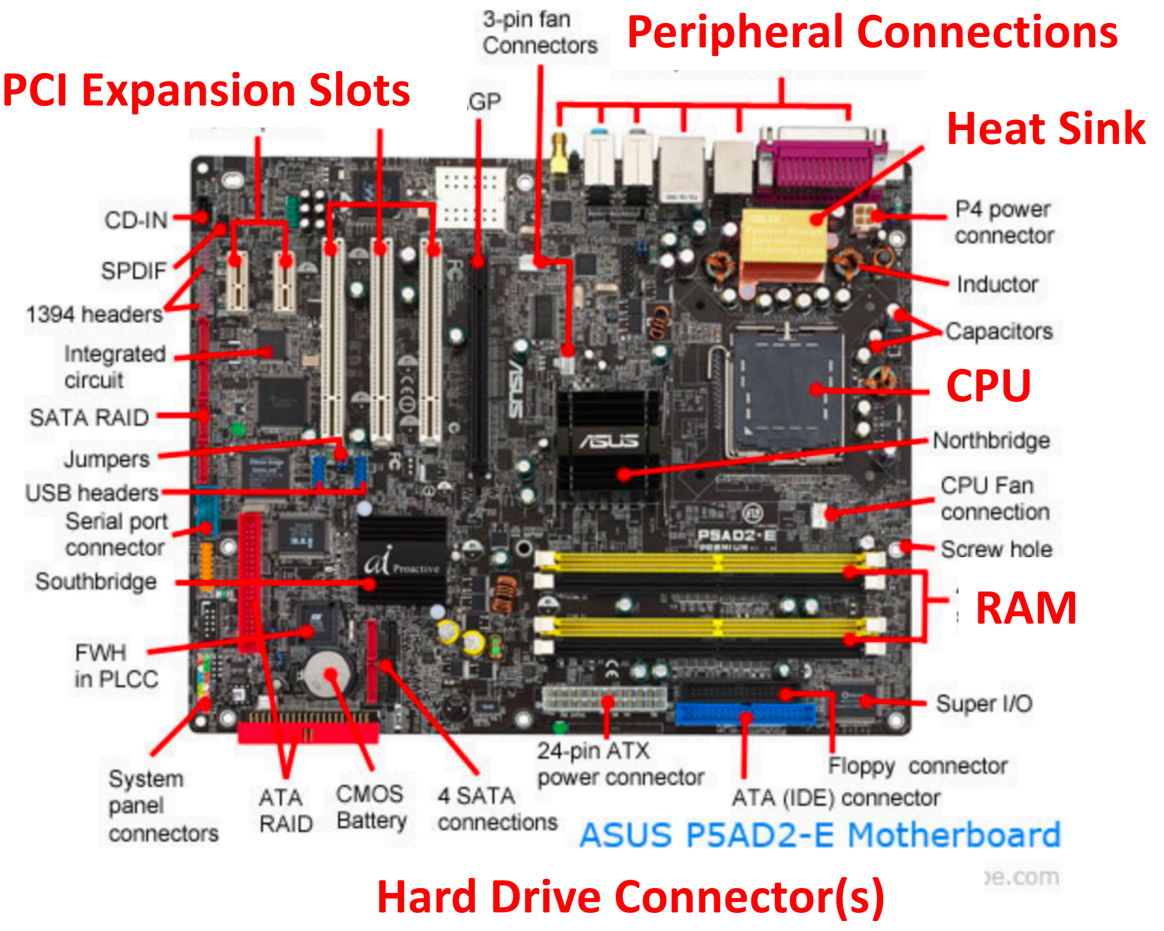
- A Computer Program and the Computer on Which it Runs
- C Programming Language
  - Variables
  - Operations
  - Program Control
  - Functions
  - Pointers & Arrays
  - Other Things
- (maybe Parallel Programming)

What is a Computer?

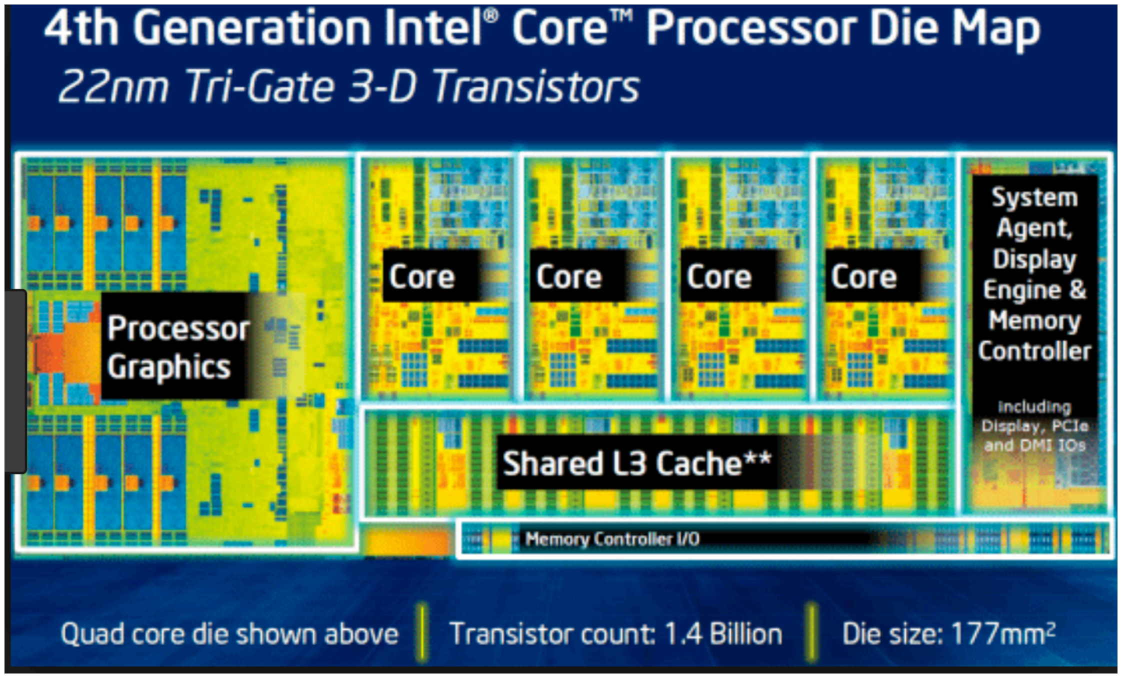




What is on a Motherboard?



What is in a CPU?


4th Generation Intel® Core™ Processor Die Map
22nm Tri-Gate 3-D Transistors
Processor Graphics
Core
Core
Core
Core
System Agent, Display Engine & Memory Controller
including Display, PCIe and DMI IOs
Shared L3 Cache**
Memory Controller I/O
Quad core die shown above
Transistor count: 1.4 Billion
Die size: 177mm²

What is a Computer Program?

- A sequence of separate instructions one after another
- Each instruction tells Core to do 1 small specific task
- When the instructions are completed the Computer has done something we wanted/needed done.

Art of Programming - I

- To take a problem, and continually break it down into a series of smaller tasks until ultimately these tasks become a series of small specific individual instructions.

Art of Programming - II

- To take a problem, and continually break it down into a series of smaller ideally concurrent tasks until ultimately these tasks become a series of small specific individual instructions.
- Mindful of the architecture on which the program will run, identify those tasks which can be run concurrently and map those tasks onto the processing units of the target architecture.

What is Programming?

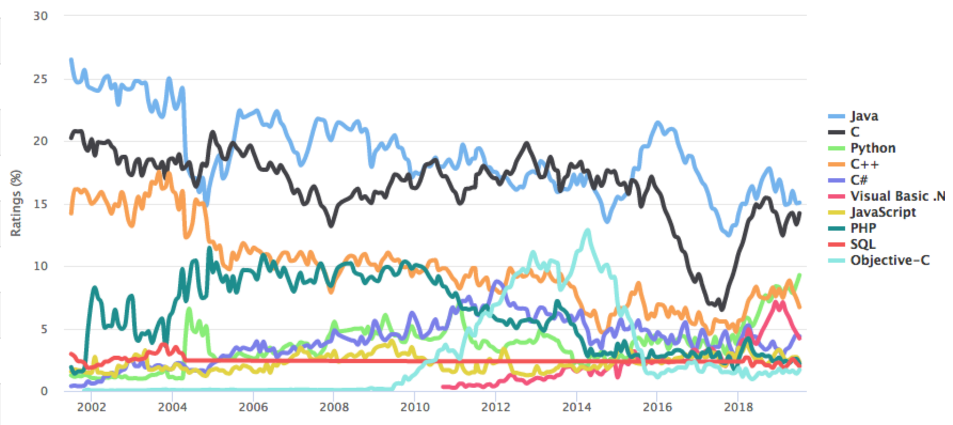
- Writing these instructions as a series of statements.
- A statement uses words, numbers and punctuation to detail the instruction. They are like properly formed sentences in English.
- A poorly formed statement -> compiler error
- Each programming language has a unique "syntax" that defines what constitutes correct statements in that language

What Programming Language?

- Hundreds of languages ….
- Only a dozen or so are popular at any time
- We will be looking at C, C++ and Python

TIOBE Index - Popular Programming Languages

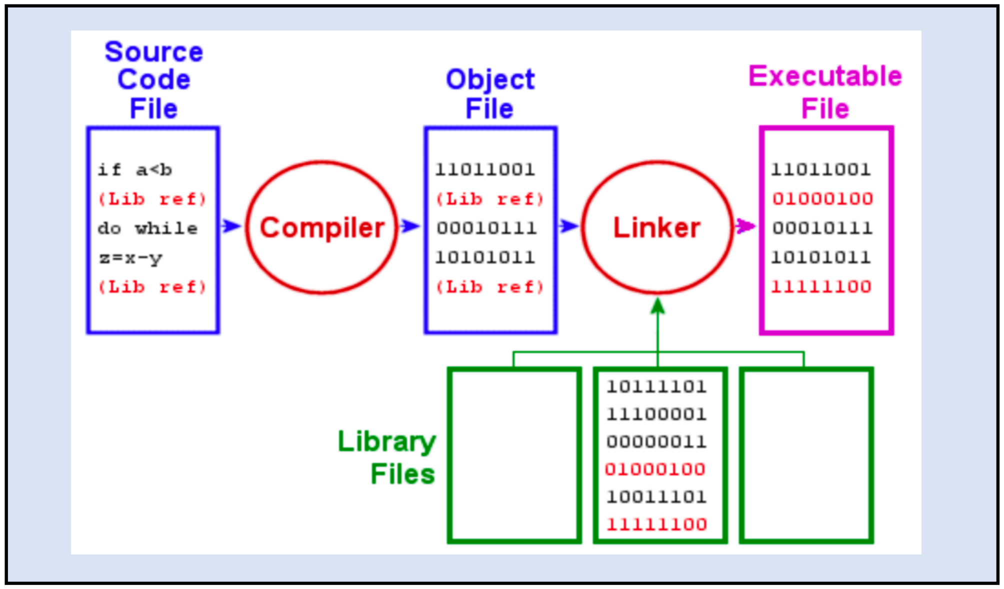
Jul 2019	Jul 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.058%	-1.08%
2	2		C	14.211%	-0.45%
3	4	⌃	Python	9.260%	+2.90%
4	3	⌄	C++	6.705%	-0.91%
5	6	⌃	C#	4.365%	+0.57%
6	5	⌄	Visual Basic .NET	4.208%	-0.04%
7	8	⌃	JavaScript	2.304%	-0.53%
8	7	⌄	PHP	2.167%	-0.67%
9	9		SQL	1.977%	-0.36%
10	10		Objective-C	1.686%	+0.23%
11	12	⌃	Ruby	1.636%	+0.43%
12	13	⌃	Assembly language	1.390%	+0.24%
13	11	⌄	Swift	1.121%	-0.29%
14	15	⌃	MATLAB	1.078%	-0.05%
15	81	⏫	Groovy	1.070%	+0.96%
16	18	⌃	Go	1.016%	+0.05%
17	19	⌃	Visual Basic	1.009%	+0.12%
18	16	⌄	Delphi/Object Pascal	0.950%	-0.16%
19	17	⌄	Perl	0.918%	-0.18%
20	14	⏬	R	0.837%	-0.31%

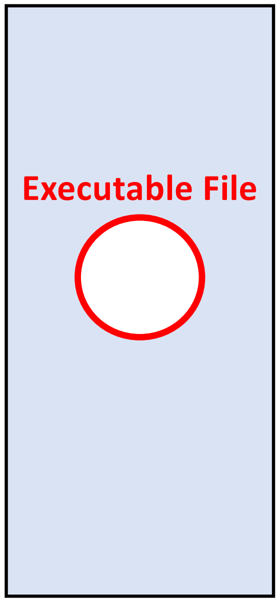


Types of Languages – Compiled/Interpreted

- Compiled:

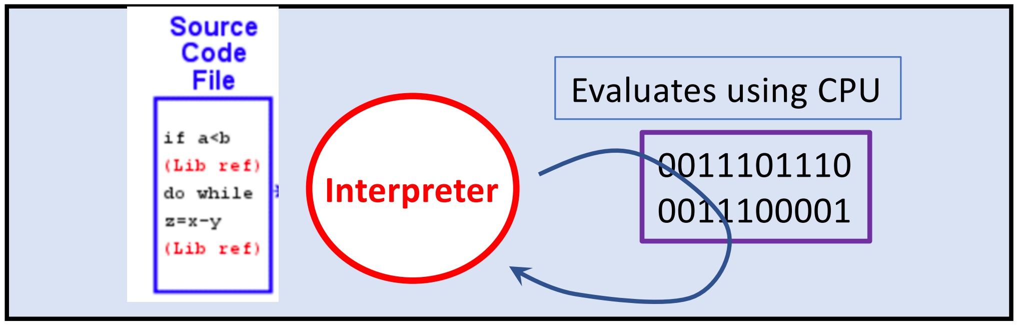
CPU





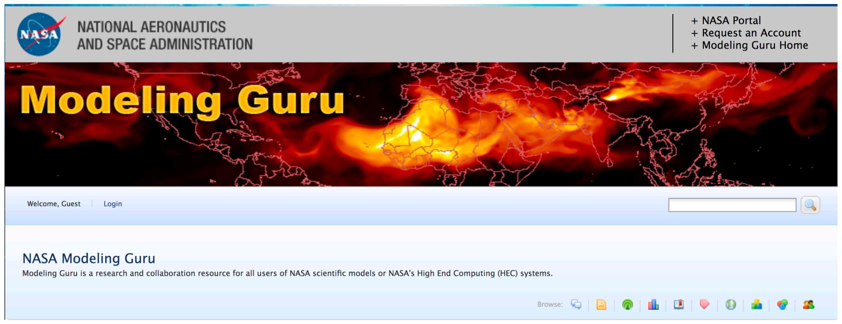
- Interpreted.

CPU



- Hybrid, e.g. Java. Compiler converts to another language, e.g. bytecode. Interpreter runs on machine and interprets this language, e.g. javaVM.

… On Performance


NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

+ NASA Portal
+ Request an Account
+ Modeling Guru Home

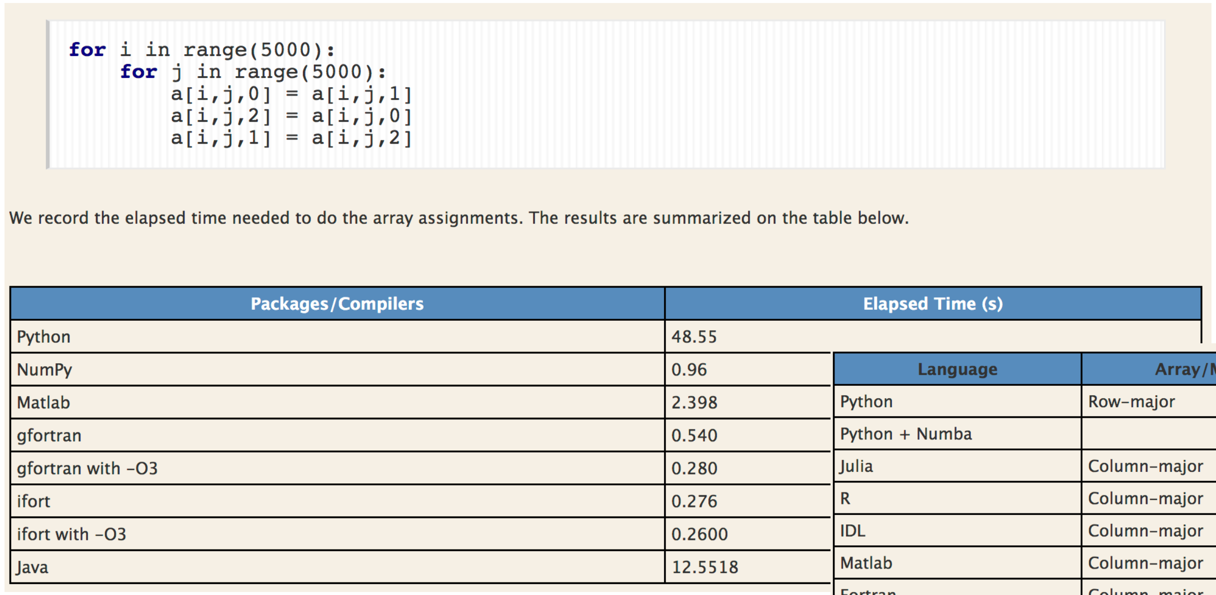
Modeling Guru

Welcome, Guest | Login

NASA Modeling Guru

Modeling Guru is a research and collaboration resource for all users of NASA scientific models or NASA's High End Computing (HEC) systems.

2009

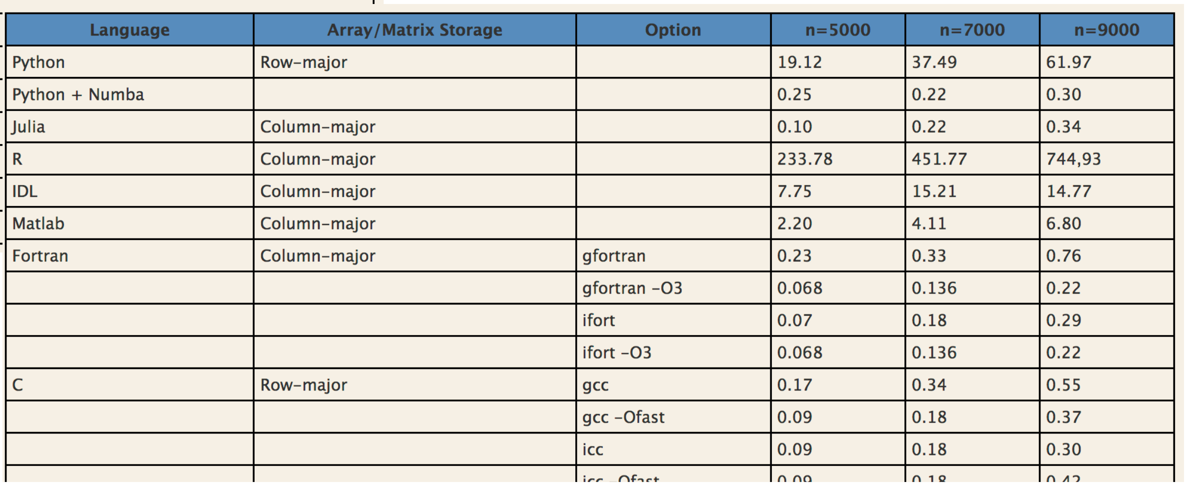


```
for i in range(5000):
    for j in range(5000):
        a[i,j,0] = a[i,j,1]
        a[i,j,2] = a[i,j,0]
        a[i,j,1] = a[i,j,2]
```

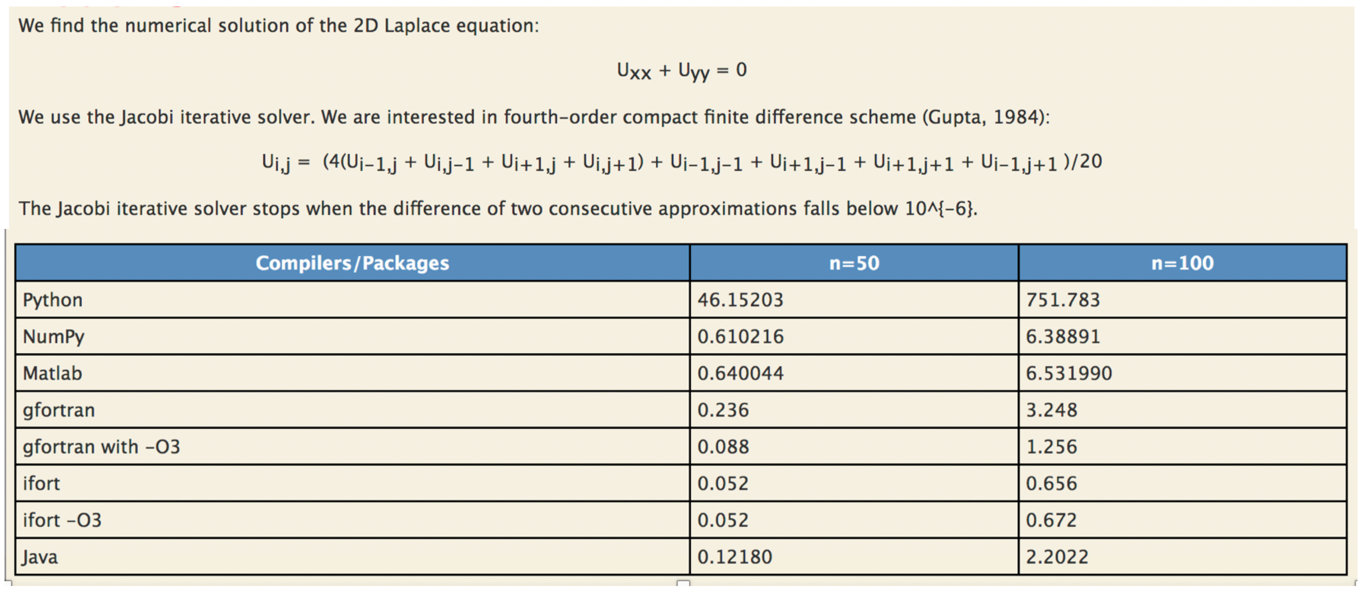
We record the elapsed time needed to do the array assignments. The results are summarized on the table below.

Packages/Compilers	Elapsed Time (s)
Python	48.55
NumPy	0.96
Matlab	2.398
gfortran	0.540
gfortran with -O3	0.280
ifort	0.276
ifort with -O3	0.2600
Java	12.5518

2019



Language	Array/Matrix Storage	Option	n=5000	n=7000	n=9000
Python	Row-major		19.12	37.49	61.97
Python + Numba			0.25	0.22	0.30
Julia	Column-major		0.10	0.22	0.34
R	Column-major		233.78	451.77	744,93
IDL	Column-major		7.75	15.21	14.77
Matlab	Column-major		2.20	4.11	6.80
Fortran	Column-major	gfortran	0.23	0.33	0.76
		gfortran -O3	0.068	0.136	0.22
		ifort	0.07	0.18	0.29
		ifort -O3	0.068	0.136	0.22
C	Row-major	gcc	0.17	0.34	0.55
		gcc -Ofast	0.09	0.18	0.37
		icc	0.09	0.18	0.30
		icc -Ofast	0.09	0.18	0.42



We find the numerical solution of the 2D Laplace equation:

$$U_{xx} + U_{yy} = 0$$

We use the Jacobi iterative solver. We are interested in fourth-order compact finite difference scheme (Gupta, 1984):

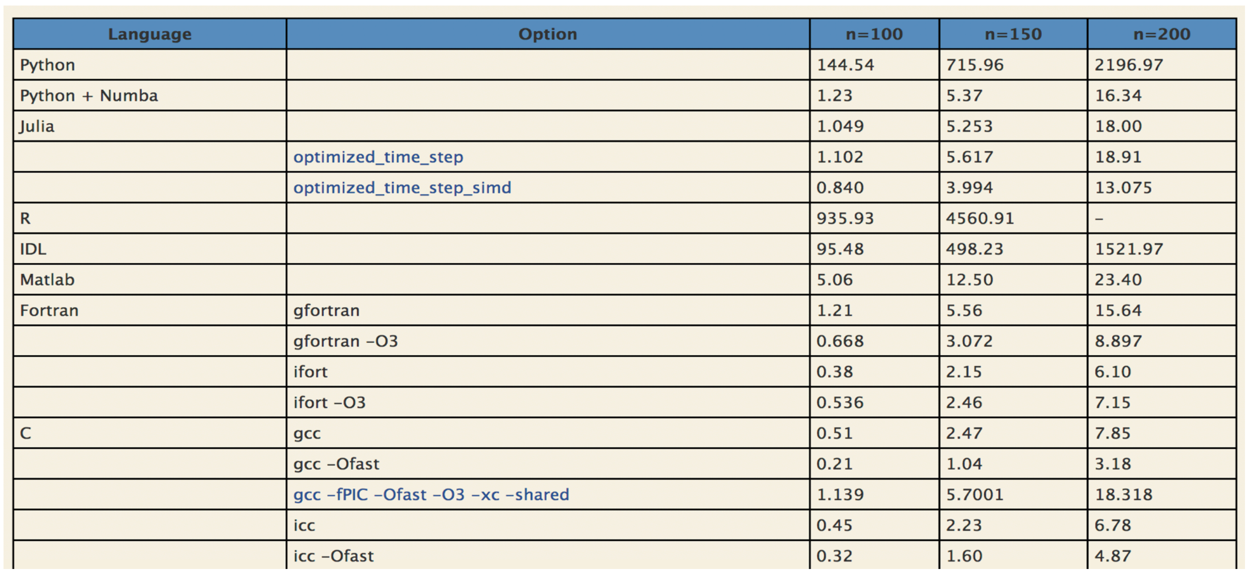
$$U_{i,j} = (4(U_{i-1,j} + U_{i,j-1} + U_{i+1,j} + U_{i,j+1}) + U_{i-1,j-1} + U_{i+1,j-1} + U_{i+1,j+1} + U_{i-1,j+1})/20$$

The Jacobi iterative solver stops when the difference of two consecutive approximations falls below 10^{-6}.

Compilers/Packages	n=50	n=100
Python	46.15203	751.783
NumPy	0.610216	6.38891
Matlab	0.640044	6.531990
gfortran	0.236	3.248
gfortran with -O3	0.088	1.256
ifort	0.052	0.656
ifort -O3	0.052	0.672
Java	0.12180	2.2022

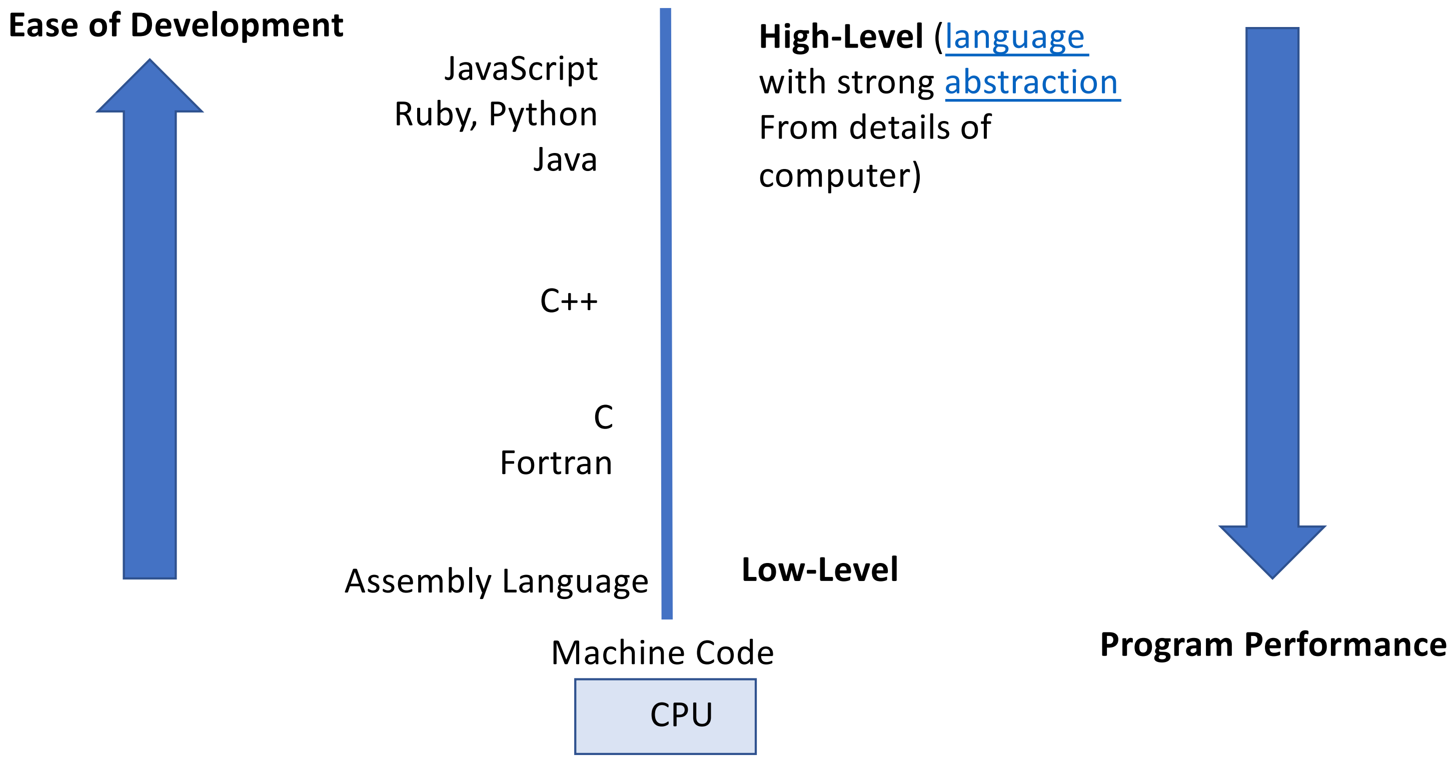
2009

2019



Language	Option	n=100	n=150	n=200
Python		144.54	715.96	2196.97
Python + Numba		1.23	5.37	16.34
Julia		1.049	5.253	18.00
	optimized_time_step	1.102	5.617	18.91
	optimized_time_step_simd	0.840	3.994	13.075
R		935.93	4560.91	-
IDL		95.48	498.23	1521.97
Matlab		5.06	12.50	23.40
Fortran	gfortran	1.21	5.56	15.64
	gfortran -O3	0.668	3.072	8.897
	ifort	0.38	2.15	6.10
	ifort -O3	0.536	2.46	7.15
C	gcc	0.51	2.47	7.85
	gcc -Ofast	0.21	1.04	3.18
	gcc -fPIC -Ofast -O3 -xc -shared	1.139	5.7001	18.318
	icc	0.45	2.23	6.78
	icc -Ofast	0.32	1.60	4.87

Programming Language Hierarchy

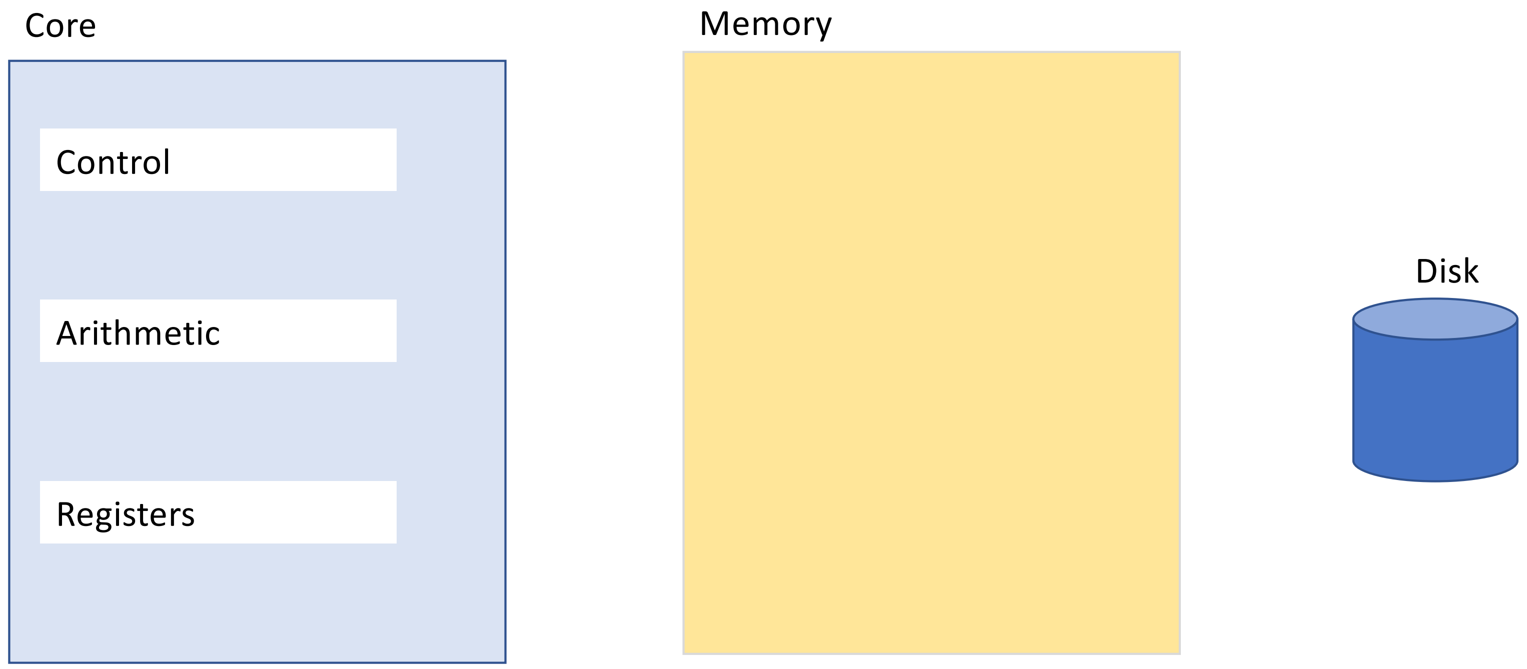


Single Processor Machine – Idealized Model

Fetches, decodes
& dispatches instructions

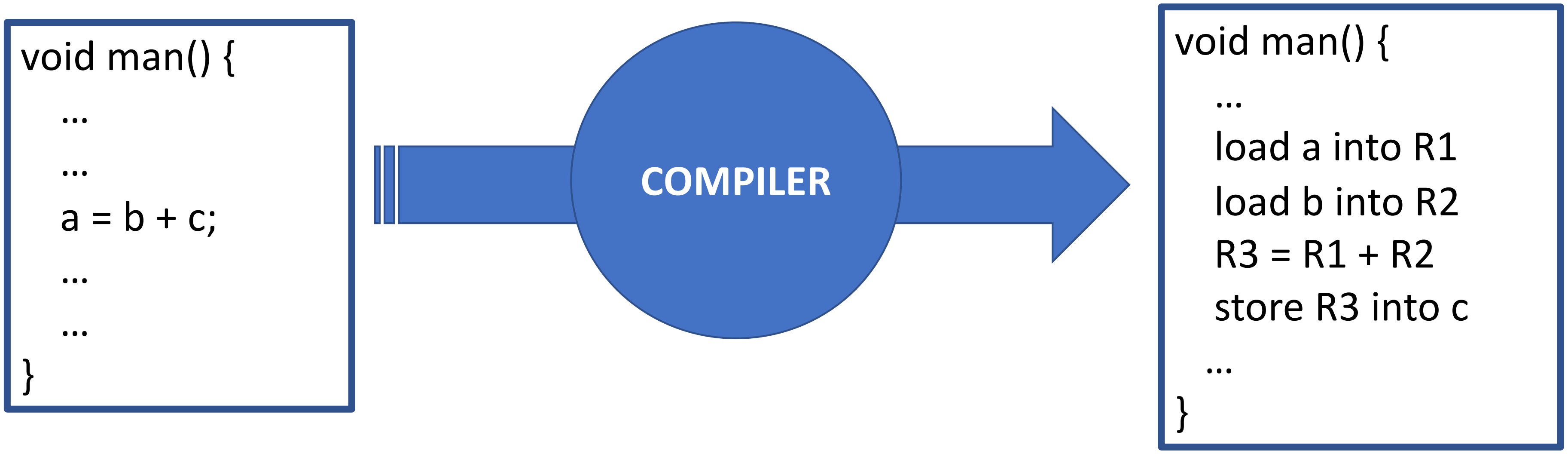
Performs numerical
operations

Data from Memory



What is a Compiler?

- An application whose purpose is to:
  - Check a Program is legal (follows the syntax)
  - Translate the program into another language (assembly, machine instruction)



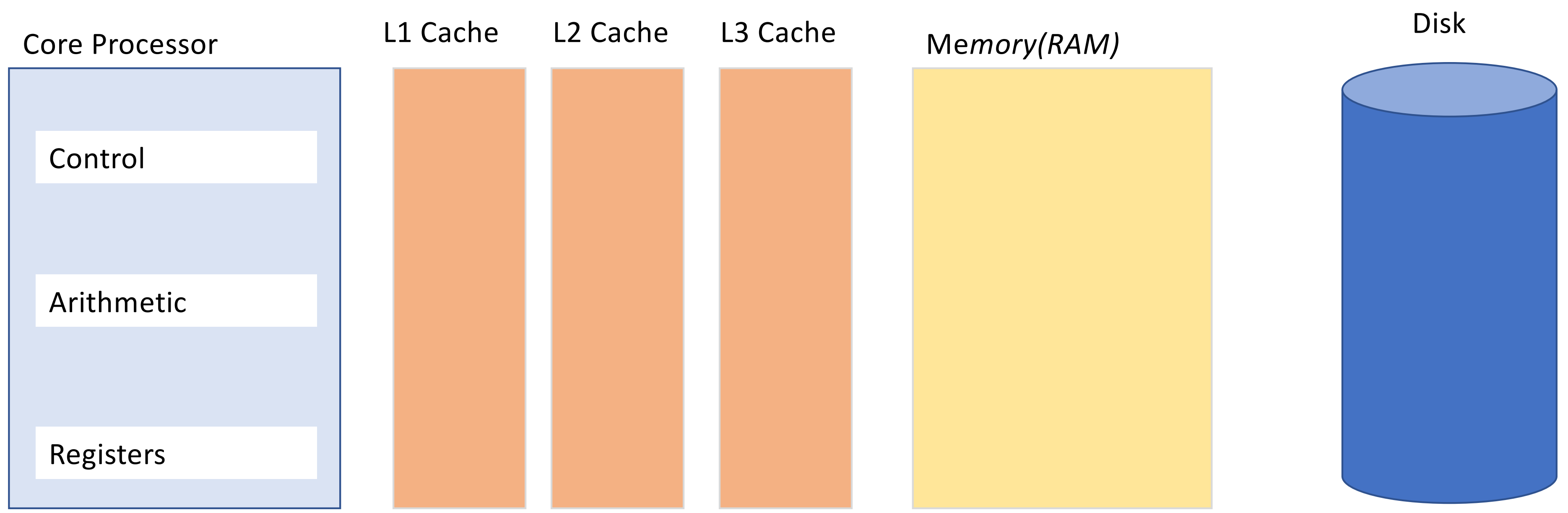
```
void man() {
    …
    …
    a = b + c;
    …
    …
}
```

```
void man() {
    …
    load a into R1
    load b into R2
    R3 = R1 + R2
    store R3 into c
   …
}
```

Cores only Work on Data in Registers

Which is Why Understanding Memory Hierarchy Very Important if you want to write fast code

Memory Hierarchy



						Hard Drive	SSD
Size	1000 Bytes	64 KB	256 KB	2-4 MB	4-16 GB	4-16 TB	*.25-1TB*
Latency	0.3 ns	1 ns	3-10 ns	20-30 ns	50-100 ns	5-10e6 ns	25-50e3 ns
	Compiler	HW	HW	HW	Operating System	Operating System	

What is Cache?

- Small, Fast Memory
- Placed Between Registers and Main Memory
- It keeps a copy of data in memory
- It is hidden from software (neither compiler or OS can say what gets loaded)

```
void main() {
   …
 load b into R2
   …
}
```

- Cache-hit: data in cache (b in cache)
- Cache-miss: data not in cache, have to go get from memory (b in memory)
- Cache-line-length: number of bytes of data loaded into cache with missing data (32 to 128bytes)

Why Do Caches Work?

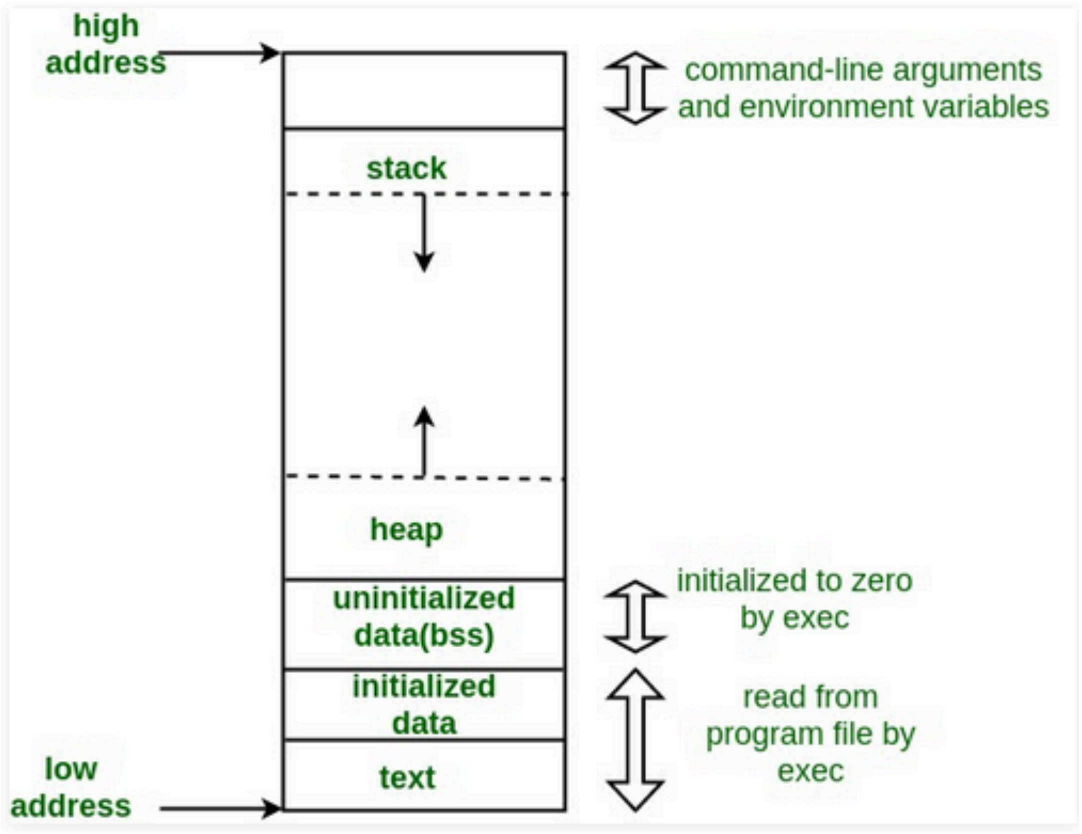
- **Spatial Locality** – probability is high that if program is accessing some memory on 1 instruction, it is going to access a nearby one soon
- **Temporal Locality** – probability is high that if program is accessing some memory location it will access same location again soon.

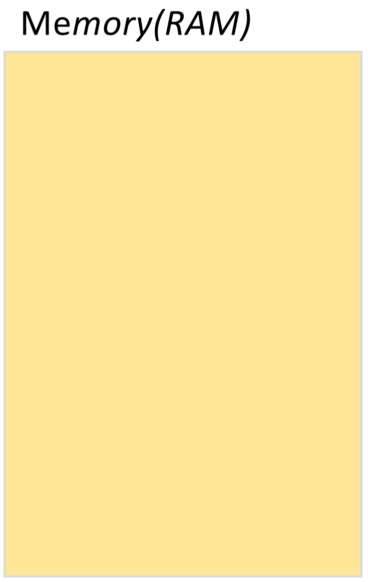
```
int main() {
    …
   double dotProduct = 0
    for (int i=0, i<vectorSize; i++)
        dotProduct += x[i] * y[i];
    …
}
```

So Why Did I Bring Cache Up If No Control Over It?

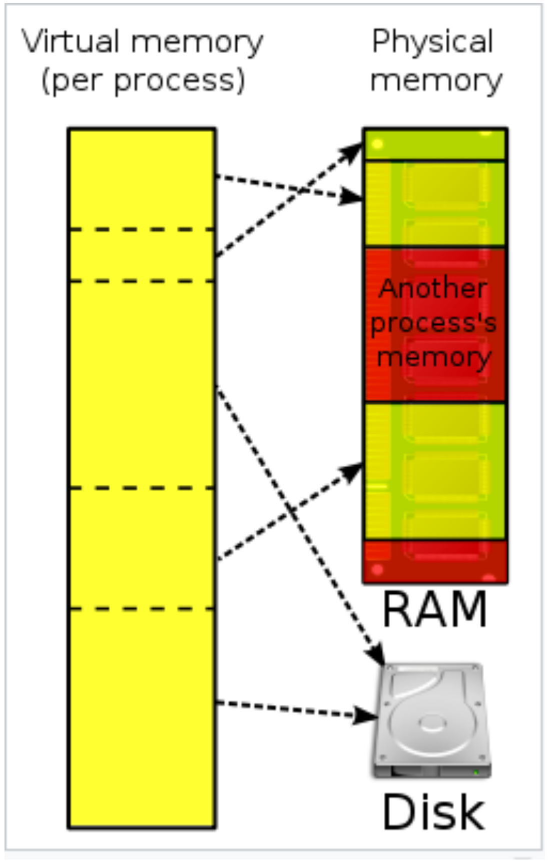
- Knowing caches exist, understanding how they work, allows you as a programmer to take advantage of them when you write the program, i.e. Allocate chunks of memory cache, think about where you store your variables, …

Program Memory – Main Memory Mismatch


high address
stack
heap
uninitialized data(bss)
initialized data
text
low address
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec



Virtual Memory



- Is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" wikipedia.
- Program Memory is broken into a number of pages. Some of these are in memory, some on disk, some may not exist at all (segmentation fault)
- CPU issues virtual addresses (load b into R1) which are translated to physical addresses. If page in memory, HW determines the physical memory address. If not, page fault, OS must get page from Disk.
- Page Table: table of pages in memory.
- Page Table Lookup – relativily expensive.
- Page Fault (page not in memory) very expensive as page must be brought from disk by OS
- Page Size: size of pages
- TLB Translation Look-Aside Buffer HW cache of virtual to physical mappings.
- Allows multiple programs to be running at once in memory.

Major page fault

- **Major => need to retrieve page from disk**
  1. CPU detects the situation (valid bit = 0)
     - It cannot remedy the situation on its own;
     - It doesn't communicate with disks (nor even knows that it should)
  2. CPU generates interrupt and transfers control to the OS
     - Invoking the OS page-fault handler
  3. OS regains control, realizes page is on disk, initiates I/O read ops
     - To read missing page from disk to DRAM
     - Possibly need to write victim page(s) to disk (if no room & dirty)
  4. OS suspends process & context switches to another process
     - It might take a few milliseconds for I/O ops to complete
  5. Upon read completion, OS makes suspended process runnable again
     - It'll soon be chosen for execution
  6. When process is resumed, faulting operation is re-executed
     - Now it will succeed because the page is there

The C Programming Language

- Originally Developed by Dennis Ritchie at Bell Labs in 1969 to implement the Unix operating system.
- It is a **compiled** language
- It is a **structured** (PROCEDURAL) language
- It is a **strongly typed** language
- The most widely used languages of all time
- It’s been #1 or #2 most popular language since mid 80’s
    - It works with nearly all systems
    - As close to assembly as you can get
    - Small runtime (embedded devices)

C Program Structure

A C Program consists of the following parts:

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Everyone’s First C Program

no space between # and include

hello1.c

```
#include <stdio.h>

int main() {
   /* my first program in C */
   printf("Hello World! \n");
   return 0;
}
```

statements end with **;**

Function that indicates they will return an integer, MUST return an integer

- The first line of the program ***#include <stdio.h>*** is a preprocessor command, which tells a C compiler to include the stdio.h file before starting compilation.
- The next line ***int main()*** is the main function. Every program must have a main function as that is where the program execution will begin.
- The next line **/*...*/** will be ignored by the compiler. It is there for the programmer benefit. It is a comment.
- The next line is a statement to invoke the ***printf(...)*** function which causes the message "Hello, World!" to be displayed on the screen. The prototype for the function is in the stdio.h file. It’s implementation in the standard C library.
- The next statement **return 0;** terminates the main() function and returns the value 0.

Exercise: Compile & Run Hello World!

1. With a text editor create the file hello.c

   in a terminal window type: **gedit hello.c**
2. Compile it

   in a terminal window type: **gcc hello.c**
3. Run it

   in a terminal window type: **./a.out**

```c
#include <stdio.h>

int main() {
   // my first program in C
   printf("Hello World! \n");
   return 0;
}
```

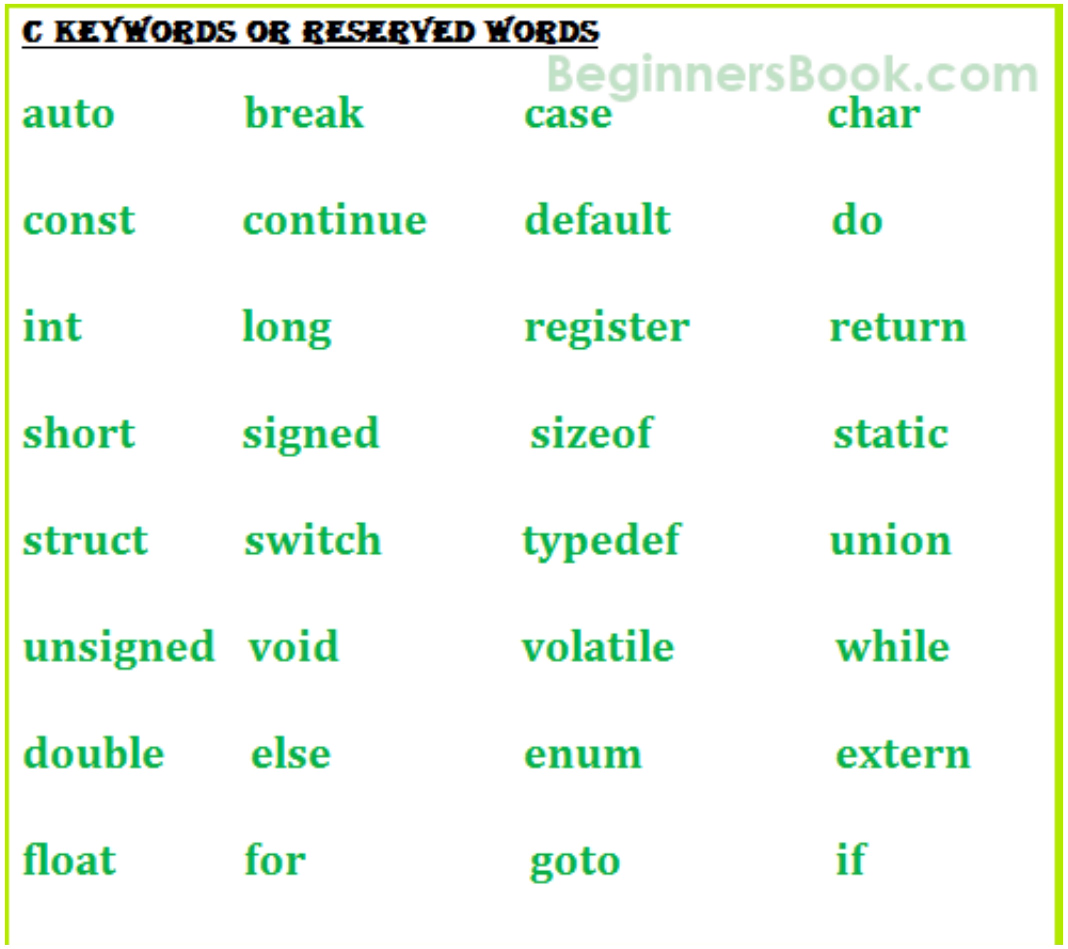
A comment may also be Specified using a //. The compiler ignores all text from comment to EOL

Variables and Types

- Except in simplest of programs we need to keep track of data, e.g. current and max scores in a game, current sum in vector product calculation
- A **Variable** is a **name** a programmer can set aside for storing & accessing accessing a **memory location.**
- C is a **strongly typed language.** The programmer **must specify the data type associated with the variable.**

- Names are **made up of letters and digits;** they are **case sensitive;** names must **start with a character,** for variable names **'_' counts as a chracacter**
- **Certain keywords are reserved, i.e. cannot be used as variable names**

Reserved Keywords in C



C KEYWORDS OR RESERVED WORDS

auto	break	case	char
const	continue	default	do
int	long	register	return
short	signed	sizeof	static
struct	switch	typedef	union
unsigned	void	volatile	while
double	else	enum	extern
float	for	goto	if

Variable Example

var1.c

```
#include <stdio.h>
// define and then set variable
int main(int argc, char **argv) {
  int a;
  a = 1;
  printf("Value of a is %d \n",a);
  return 0;
}
```

Uninitialized Variable

var2.c

```
#include <stdio.h>
// define & set in 1 statement
int main(int argc, char **argv) {
  int a = 1;
  printf("Value of a is %d \n",a);
  return 0;
}
```

Initialized Variable

Allowable Variable Types in C - I

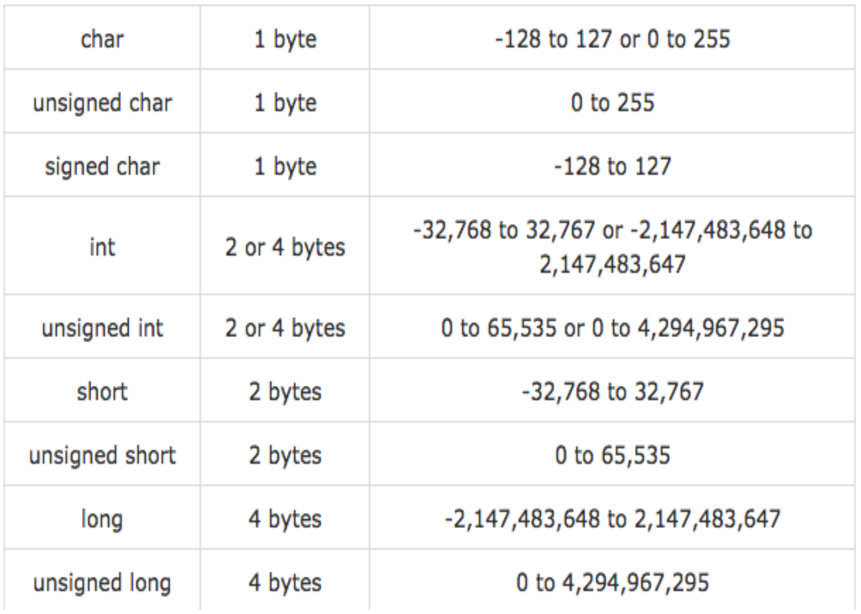
char
int
float
double
void

var3.c

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

int main(int argc, char **argv) {
   int    i1 = 5;
   float  f1 = 1.2;
   double d1 = 1.0e6;
   char   c1 = 'A';
   printf("Integer %d, float %f, double %f, char %c \n", i1, f1, d1, c1);
   return 0;
}
```

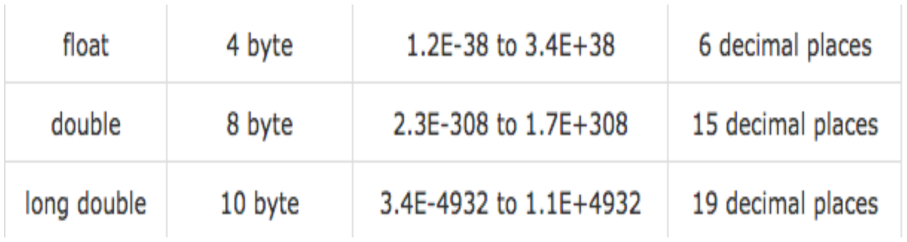
Allowable Variable Types in C – II
qualifiers: unsigned, short, long

1. Integer Types



char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

2. Floating Point Types



float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

3. Enumerated Types

4. **void** Type

5. Derived Types

Structures,
Unions,
Arrays

Arrays - I

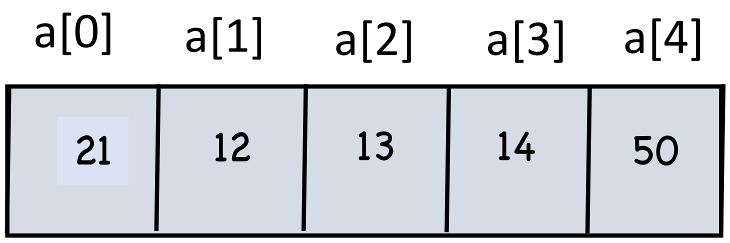
- A fixed size sequential collection of elements laid out in memory of the *same* type. We access using an index inside a square brackets, indexing start at 0
- to declare: `type arrayName [size];`

  `type arrayName [size] = {size comma separated values}`

array1.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int intArray[5] = {19, 12, 13, 14, 50};
  intArray[0] = 21;
  int first = intArray[0];
  int last = intArray[4];
  printf("First %d, last %d \n", first, last);
  return 0 ;
 }
```

WARNING: indexing starts at 0

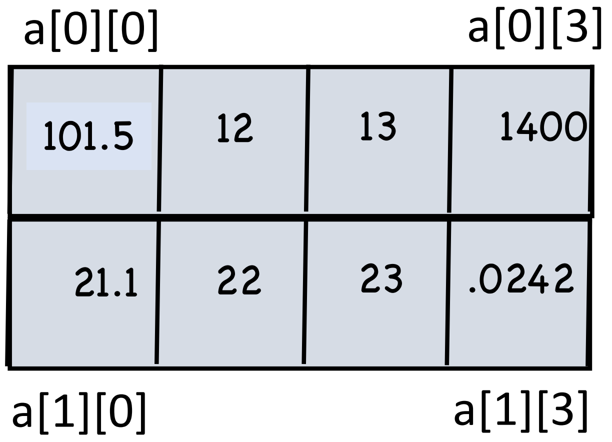


Multidimensional Arrays- I

- A fixed size sequential collection of elements laid out in memory of the *same* type
  We access using an index inside a square brackets, indexing start at 0 var1.c
- to declare: type arrayName [l1][l2][l3]…;
  type arrayName [l1][l2][l3] = {l1*l2*… comma separated values

array2.c

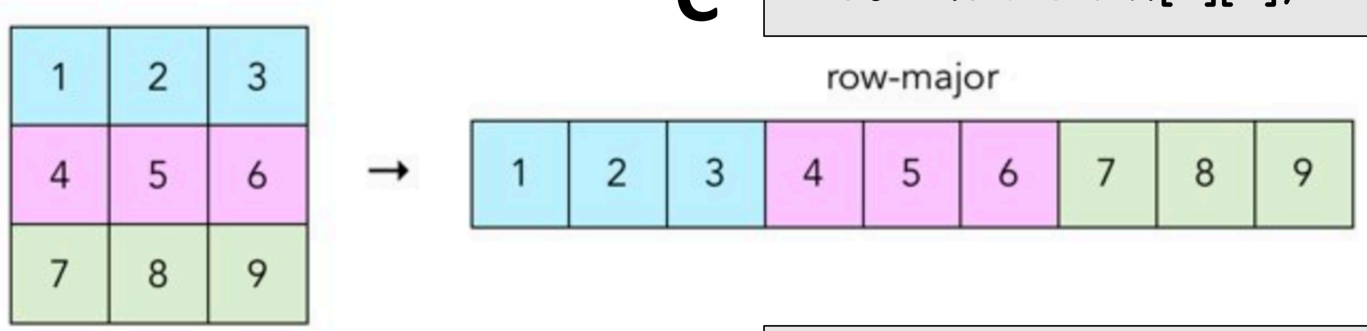
```
#include <stdio.h>

int main(int argc, char **argv) {
   double dArray[2][4]= {{19.1, 12, 13, 14e2},
                         {21.1, 22, 23, 24.2e-3}};
  dArray[0][0] = 101.5;
  int first = dArray[0][0];
  int last = dArray[1][3];
  printf("First %f, last %f \n", first, last);
  return(0);
 }
```



Memory Layout of Arrays in C and Fortran

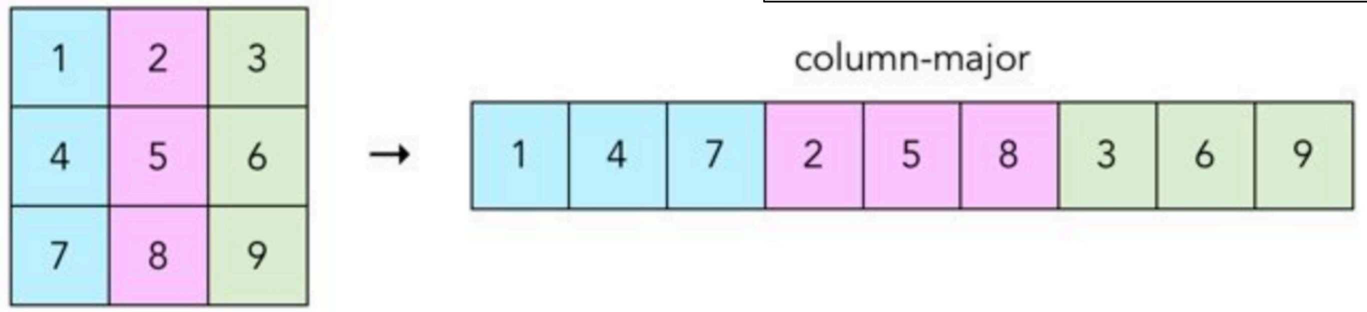
C

```
double matrix[3][3];
```



Fortran

```
REAL matrix(3,3);
```



Operations

- We want to do stuff with the data, to operate on it
- Basic Arithmetic Operations

+, -, *, /, %

op1.c

```
#include <stdio.h>

int main(int argc, const char **argv) {
  int a = 1;
  int b = 2;
  int c = a+b;
  printf("Sum of %d and %d is %d  \n",a,b,c);
  return(0);
}
```

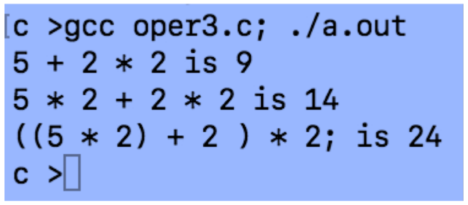
You Can String Operations Together –

op2.c

```c
#include <stdio.h>
int main(int argc, char **argv) {
  int a = 5;
  int b = 2;
  int c = a + b * 2;
  printf("%d + %d * 2 is %d  \n",a,b,c);

  c = a * 2 + b * 2;
  printf("%d * 2 + %d * 2 is %d  \n",a,b,c);

  // use parentheses
  c = ((a * 2) + b ) * 2;
  printf("((%d * 2) + %d ) * 2; is %d  \n",a,b,c);
  return(0);
}
```

What is c? Operator precedence!

USE PARENTHESES



```
[c >gcc oper3.c; ./a.out
5 + 2 * 2 is 9
5 * 2 + 2 * 2 is 14
((5 * 2) + 2 ) * 2; is 24
c >
```

C Operator Precedence Table

This page lists C operators in order of *precedence* (highest to lowest). Their *associativity* indicates in what order operators of equal precedence in an expression are applied.

Operator	Description	Associativity
( )	Parentheses (function call) (see Note 1)	left-to-right
[ ]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (convert value to temporary value of *type*)	
*	Dereference	
&	Address (of operand)	
sizeof	Determine size in bytes on this implementation	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right

Some Operations are so Common there are special operators

```
#include <stdio.h>
int main() {
    …
    a = a + 1;
    …
}
```

+=

-=

***=**

/=

++

--

```
a += 1;
```

```
a ++;
```

Conditional Code – if statement

```
if (condition) {
   // code block
}
```

- So far instruction sequence has been sequential, one instruction after the next .. Beyond simple programs we need to start doing something, if balance is less than 0 don't withdraw money

if1.c

```
#include <stdio.h>
int main(int argc, char **argv) {
 int a = 15;
  if (a < 10) {
    printf("%d is less than 10 \n", a);
  }
  if (a == 10) {
    printf("%d is equal to 10 \n", a);
  }
  if (a > 10) {
    printf("%d is greater than 10 \n", a);
  }
  return(0);
}
```

Conditional Operators

```
<
<=
>
>=
==
!=
```

If-else

```
if (condition) {
   // code block
} else {
  // other code
}
```

if2.c

```
#include <stdio.h>
int main(int argc, char **argv) {
  int a = 15;
  if  (a <= 10) {
    if (a != 10) {
      printf("%d is less than 10 \n", a);
    } else {
     printf("%d is equal to 10 \n", a);
    }
  } else {
    printf("%d is greater than 10 \n", a);
   }
  return(0);
}
```

else-if

```
if (condition) {
   // code block
} else if (condition) {
   // another code block
} else {
  // and another
}
```

if3.c

```
#include <stdio.h>
int main(int argc, char **argv) {
  int a = 15;
  if (a < 10) {
    printf("%d is less than 10 \n", a);
  } else if ( a == 10) {
       printf("%d is equal to 10 \n", a);
  } else {
     printf("%d is greater than 10 \n", a);
   }
  return(0);
}
```

Can have multiple else if in if statement

Logical and/or/not

&&

||

|

```
#include <stdio.h>
int main(int argc, char **argv) {
  int a = 15;
  if ((a < 10) && (a == 10)) {
    if  !(a == 10) {
      printf("%d is less than 10 \n", a);
    } else {
     printf("%d is equal to 10 \n", a);
    }
  } else {
     printf("%d is greater than 10 \n", a);
   }
   return(0);
}
```

Conditional Code – switch statement

- Special multi-way decision maker that tests if an expression matches one of a number of **constant** values

```
switch(expression) {
   case constant-expression :
        statement(s);
        break;  /* optional */
  case constant-expression :
       statement(s);
       break; /* optional */
   .....
 default : /* Optional */
       statement(s);
}
```

```
#include <stdio.h>
int main(int argc, char **argv) {
  char c='Y';
  switch (c) {
   case 'Y':
   case 'y':
     c = 'y';
     break;
   default:
     printf("unknown character %c \n",c);
   }
  return(0);
}
```

```
while (condition) {
   // code block
}
```

Iteration/loops - while

- Common task is to loop over a number of things, e.g. look at all files in a folder, loop over all values in an array,…

while1.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int intArray[5] = {19, 12, 13, 14, 50};
  int sum = 0, count = 0;
  while (count < 5) {
    sum += intArray[count];
    count++;   // If left out =>infinite loop ..
  }            // Something must happen in while to break out of loop
  printf("sum is: %d \n", sum);
}
```

If you do enough while loops you will recognize a pattern

1) Initialization of some variables,
2) condition,
3) increment of some value

Hence the for loop

```
for (init; condition; increment) {
   // code block
}
```

for loop

for1.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int intArray[5] = {19, 12, 13, 14, 50};
  int sum = 0;
  for (int count = 0; count < 5; count++) {
    sum += intArray[count];
  }
  printf("sum is: %d \n", sum);
 }
```

```
for (init; condition; increment) {
   // code block
}
```

for loop – multiple init & increment

for2.c

```
#include <stdio.h>

int main(int argc, char **argv) {
  int intArray[6] = {19, 12, 13, 14, 50, 0};
  int sum = 0;
  for (int i = 0, j=1; i < 5; i+=2, j+=2) {
    sum += intArray[i] + intArray[j];
  }
  printf("sum is: %d \n", sum);
 }
```

Exercise: Code to count number of digits, white spaces (‘ ‘, ‘\n’,’\t’) and other char in a file. Write info out.

```
#include <stdio.h>
int main() {
  char c;
  int nDigit =0, nWhite =0, nOther = 0;
  while ((c = getchar()) != EOF) {
     // your code
  }
  // some more code here
}
```

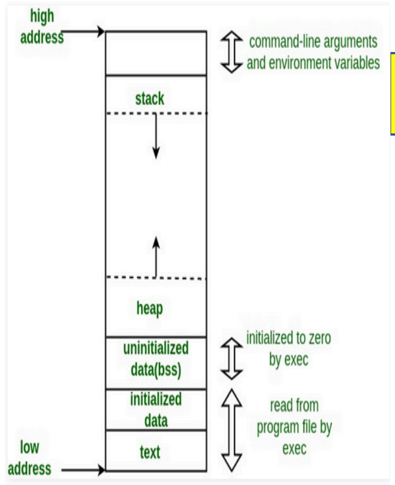
1. **gedit count.c**
2. **gcc count.c**
3. **./a.out < count.c**

NOW COMES
THE HARD PART ……….

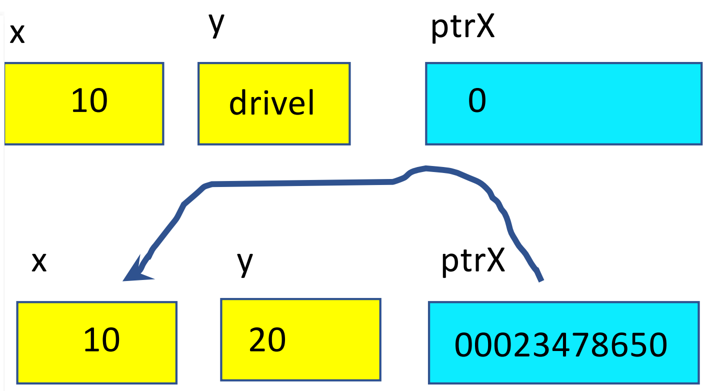
Pointers & Addresses (before I start using them in examples)

- You will use pointers an awful lot if you write any meaningful C code.
- Remember when you declare variables you are telling compiler to set aside some memory to hold a specific type and you refer to that memory when you use the name, e.g. int x. When you specify a pointer, you are seeting aside a mem address.
- The unary **&** gives the "**address**" of an object in memory.
- The unary ***** in a declaration indicates that the object is a pointer to an object of a specific type
- The unary ***** elsewhere treats the operand as an address, and depending on which side of operand either sets the contents at that address or fetches the contents.

```
#include <stdio.h>
int main() {
    int x =10, y;
    int *ptrX =0;

    ptrX = &x;
    y = *ptrX + x;
}
```



high address
command-line arguments and environment variables
stack
heap
uninitialized data(bss)
initialized to zero by exec
initialized data
read from program file by exec
text
low address



```
void man() {
    ...
    load ptrX into R1
    load R1 into R2
    load x into R3
    R4 = R2 + R3
    store R4 into y
```

Functions

- **Art of Programming I**: *"To take a problem, and recusivily break it down into a series of smaller tasks until ultimately these tasks become a series of small specific individual instructions."*
- For large code projects the we do not put all the code inside a single main block
- We break it up into logical/meaningful blocks of code. In object-oriented programming we call these blocks **classes**, in procedural programming we call these blocks **procedures or functions.**
- Functions make large programs manageable: easier to understand, allow for code re-use, allow it to be developed by teams of programmers,..

C Function

```
returnType funcName (funcArgs) {
   codeBlock
}
```

- **returnType** <optional>: what data type the function will return, if no return is specified returnType is **int.** If want function to return nothing the return to specify is **void.**
- **funcName**: the name of the function, you use this name when "invoking" the function in your code.
- **funcArgs**: comma seperated list of args to the function.
- **codeBlock**: contains the statements to be executed when procedure runs. These are only ever run if procedure is called.

function1.c

int *: data is a pointer to an int

```
#include <stdio.h>
// function to evaluate vector sum
int sumArray(int *data, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += data[i];
  }
  return sum;
}

int main(int argc, char **argv) {
  int intArray[6] = {19, 12, 13, 14, 50, 0};
  int sum = sumArray(intArray, 6);
  printf("sum is: %d \n", sum);
  return(0);
 }
```

function2.c

```
#include <stdio.h>
// function to evaluate vector sum
int sumArray(int *data, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += *data++;
  }
  return sum;
}

int main(int argc, char **argv) {
  int intArray1[6] = {19, 12, 13, 14, 50, 0};
  int intArray2[3] = {21, 22, 23};
  int sum1 = sumArray(intArray1, 6);
  int sum2 = sumArray(intArray2, 3);
  printf("sums: %d and %d\n", sum1, sum2);
  return(0);
 }
```

Function Prototype

function3.c

```
#include <stdio.h>
int sumArray(int *arrayData, int size);
int main(int argc, char **argv) {
  int intArray1[6] = {19, 12, 13, 14, 50, 0};
  int intArray2[3] = {21, 22, 23};
  int sum1 = sumArray(intArray1, 6);
  int sum2 = sumArray(intArray2, 3);
  printf("sums: %d and %d\n", sum1, sum2);
  return(0);
 }
// function to evaluate vector sum
int sumArray(int *data, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += *data++;
  }
  return sum;
}
```

Good practice to give the args names

Good Practice:

1. For large programs it is a good idea to put functions into different files (many different people can be working on different parts of the code)
2. If not too large, put them in logical units, i.e. all functions dealing with vector operations in 1 file, matrix operations in another.
3. Put prototypes for all functions in another file.
4. If function large, put in separate file.
5. Get into a system of documenting inputs and outputs.

main.c

```
#include <stdio.h>
#include “myVector.h”
int main(int argc, char **argv) {
  int intArray[6] = {19, 12, 13, 14, 50, 0};
  int sum;
  sum = sumArray(intArray, 6);
  printf(”sum is: %d \n”, sum);
 }
}
```

myVector.h

```
int sumArray(int *arrayData, int size);
int productArray(int *arayData, int size);
int normArray(int *arrayData, int size);
int dotProduct(int *array1, int *array2, int size);
```

myVector.c

```
// function to evaluate vector sum
//  inputs:
//   data: pointer to integer array
//   size: size of the array
// outputs:
//
// return:
//    integer sum of all values
int sumArray(int *data, int size) {
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += data[i];
  }
  return sum;
}
```

Exercise: Write a function to sum two values

```
#include <stdio.h>
int sumInt(int a, int b);
int main() {
   int integer1, integer2, sum;
   printf("Enter first integer: ");
   scanf("%d", &integer1); // read input to integer 1
   printf("Enter second integer: ");
   scanf("%d", &integer2); // Read input into integer2
   sum = sumInt(integer1, integer2);
   printf("sum %d + %d = %d\n", integer1, integer2, sum);
   return(0);
  }
// your code here
int sumInt(int a, int b) {
    // your code here
}
```

&integer1: memory address of integer1

&integer2: memory address of integer2

1. **gedit sumc**
2. **gcc sum.c**
3. **./a.out**

Pass By Value, Pass by Reference

- C (unlike some languages) all args are passed by value

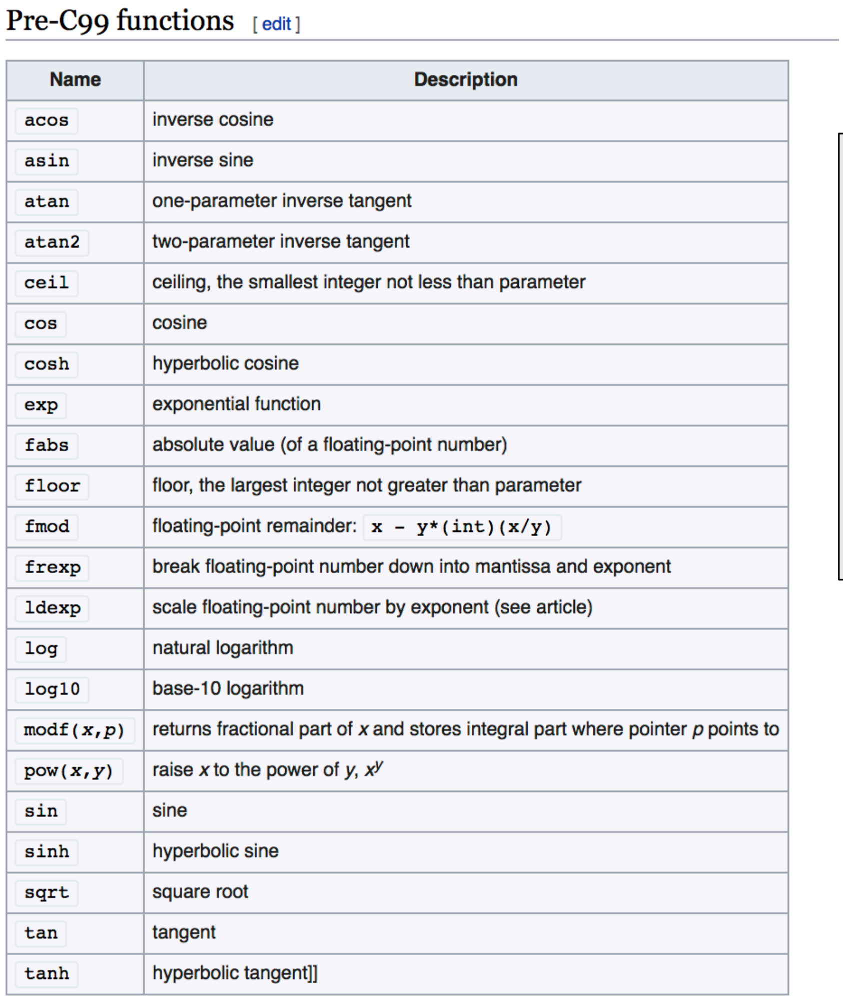
to change the function argument in the callers "memory" we can pass pointer to it, i.e it's address in memory.

This is Useful if you want multiple variables changed, or want to return an error code with the function.

function4.c

```
#include <stdio.h>
sumInt(int1, int2, int *sum);
int main() {
    int int1, int2, sum=0;
    printf("Enter first integer: ");
    scanf("%d", &int1);
    printf("Enter second integer: ");
    scanf("%d", &int2);
    sumInt(int1, int2, sum);
    print("%d + %d = %d \n", int1, int2, sum)
  }
void sumInt(int a, int b, int *sum) {
   *sum = a+b;
}
```

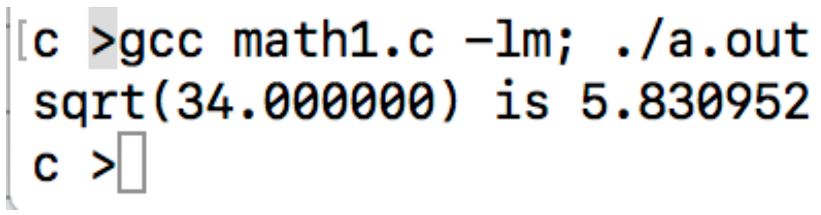
Math Functions in <math.h>, link with -lm



Pre-C99 functions [edit]

Name	Description
`acos`	inverse cosine
`asin`	inverse sine
`atan`	one-parameter inverse tangent
`atan2`	two-parameter inverse tangent
`ceil`	ceiling, the smallest integer not less than parameter
`cos`	cosine
`cosh`	hyperbolic cosine
`exp`	exponential function
`fabs`	absolute value (of a floating-point number)
`floor`	floor, the largest integer not greater than parameter
`fmod`	floating-point remainder: `x - y*(int)(x/y)`
`frexp`	break floating-point number down into mantissa and exponent
`ldexp`	scale floating-point number by exponent (see article)
`log`	natural logarithm
`log10`	base-10 logarithm
`modf(x,p)`	returns fractional part of *x* and stores integral part where pointer *p* points to
`pow(x,y)`	raise *x* to the power of *y*, x^y
`sin`	sine
`sinh`	hyperbolic sine
`sqrt`	square root
`tan`	tangent
`tanh`	hyperbolic tangent]]

```
#include <stdio.h>
int main() {
double a = 34.0;
    double b = sqrt(a);
    print("%f + %f = %f \n", a, b)
    return 0;
  }
```

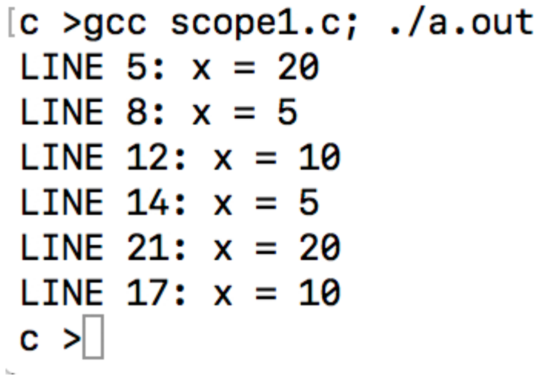


```
c >gcc math1.c -lm; ./a.out
sqrt(34.000000) is 5.830952
c >
```

Scope of Variables

scope1.c

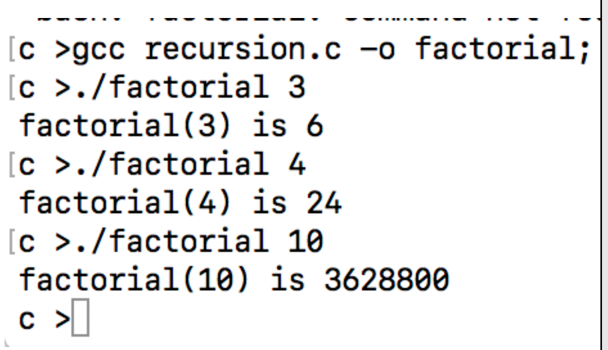
```c
#include <stdio.h>
int sum(int, int);
int x = 20; // global variable
int main(int argc, char **argv) {
   printf("LINE 5: x = %d\n",x);

   int x = 5;
   printf("LINE 8: x = %d\n",x);

   if (2 > 1) {
      int x = 10;
      printf("LINE 12: x = %d\n",x);
   }
   printf("LINE 14: x = %d\n",x);

   x = sum(x,x);
   printf("LINE 17: x = %d\n",x);
}

int sum(int a, int b) {
   printf("LINE 21: x = %d\n",x);
   return a+b;
}
```



```
c >gcc scope1.c; ./a.out
LINE 5: x = 20
LINE 8: x = 5
LINE 12: x = 10
LINE 14: x = 5
LINE 21: x = 20
LINE 17: x = 10
c >
```

Recursion

- Recursion is a powerful programming technique commonly used in divide-and-conquer situations.



```
c >gcc recursion.c -o factorial;
c >./factorial 3
factorial(3) is 6
c >./factorial 4
factorial(4) is 24
c >./factorial 10
factorial(10) is 3628800
c >
```

recursion1.c

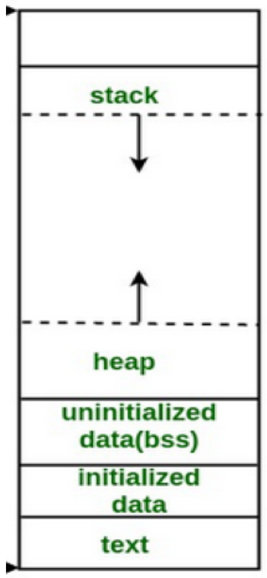
```
#include <stdio.h>
#include <stdlib.h>
int factorial(int n);
int main(int argc, char **argv) {
  if (argc < 2) {
    printf("Program needs an integer 
argument\n");
    return(-1);
  }
  int n = atoi(argv[1]);
  int fact = factorial(n);
  printf("factorial(%d) is %d\n",n, fact);
  return 0;
}
int factorial(int n) {
  if (n == 1)
    return 1;
  else
    return n*factorial(n-1);
}
```

Arrays - II

- An array is fixed size sequential collection of elements laid out in memory of the *same* type. We access using an index inside a square brackets, indexing start at 0
- to declare: `type arrayName [size];`

  `type arrayName [size] = {size comma separated values}`
- Works for arrays where we know the size at compile time. There are many times when we do not know the size of the array.
- Need to use **pointers** and functions **free()** and **malloc()**

```
type *thePointer = (type *)malloc(numElements*sizeof(type));
  …
free(thePointer)
```



- Memory for the array using free() comes from the heap
- **Always remember to free() the memory** .. Otherwise can run out of memory.

pointer, malloc() and free()

memory1.c

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

int main(int argc, char **argv) {
  int n;
  double *array1=0, *array2=0, *array3=0;

 // get n
  printf("enter n: ");
  scanf("%d", &n);
  if (n <=0) {printf ("You idiot\n"); return(0);}

  // allocate memory & set the data
  array1 = (double *)malloc(n*sizeof(double));
  for (int i=0; i<n; i++) {
    array1[i] = 0.5*i;
  }
  array2 = array1;
  array3 = &array1[0];

  for (int i=0; i<n; i++, array3++) {
    double value1 = array1[i];
    double value2 = *array2++;
    double value3 = *array3;
    printf("%.4f %.4f %.4f\n", value1, value2, value3);
  }
  // free the array
  free(array1);
  return(0);
}
```

```
c >gcc memory1.c; ./a.out
enter n: 5
0.0000 0.0000 0.0000
0.5000 0.5000 0.5000
1.0000 1.0000 1.0000
1.5000 1.5000 1.5000
2.0000 2.0000 2.0000
c >./a.out
enter n: 3
0.0000 0.0000 0.0000
0.5000 0.5000 0.5000
1.0000 1.0000 1.0000
c >
```

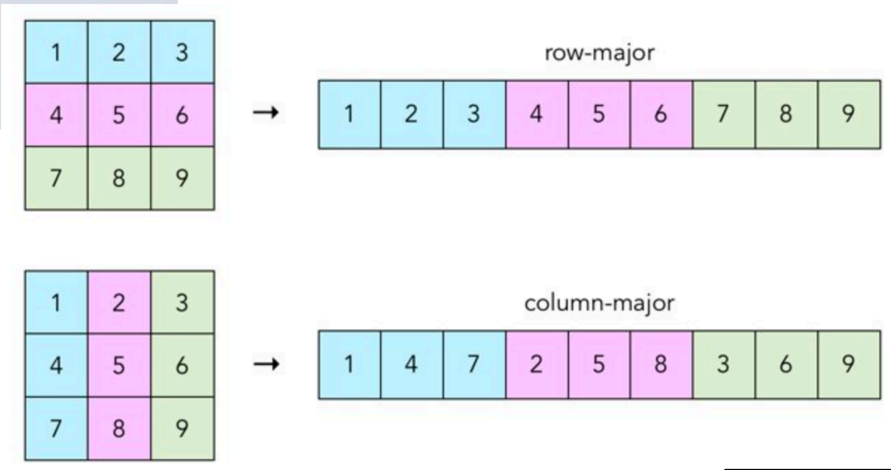
Pointers to pointers & multi-dimensional arrays

memory2.c

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

 int main(int argc, char **argv) {
  int n;
  double **matrix1 =0;

  printf("enter n: ");
  scanf("%d", &n);

  // allocate memory & set the data
  matrix1 = (double **)malloc(n*sizeof(double *));
  for (int i=0; i<n; i++) {
      matrix1[i] = (double *)malloc(n*sizeof(double));
      for (int j=0; j<n; j++)
          matrix1[i][j] = i;
  }
  for (int i=0; i<n; i++) {
      for (int j=0; j<n; j++)
          printf("(%d,%d) %.4f\n", i,j, matrix1[i][j]);
  }
  // free the data
  for (int i=0; i<n; i++)
       free(matrix1[i]);
  free(matrix1);
}
```

for Compatibility with many matrix libraries this is poor code:

```
  double **matrix2 =0;
matrix2 = (double  **)malloc(numRows*sizeof(double  *));
  for  (int  i=0;  i<numRows;  i++)  {
      matrix2[i]  =  (double  *)malloc(numCols*sizeof(double));
        for  (int  j=0;  j<numCols;  j++)
            matrix2[i][j]  =  i;
  }
```



Because many prebuilt libraries work assuming continuous layout and Fortran column-major order:

```
  double  *matrix2 =0;
matrix2 = (double  *)malloc(numRows*numCols*sizeof(double  *));
 for  (int  i=0;  i<numRows;  i++)  {
      for  (int  j=0;  j<numCols;  j++)
          matrix2[I + j*numRows]  =
 }
```

```
 double *matrix2 =0;
matrix2 = (double  *)malloc(nu
 for  (int  j=0;  j<numCols;  j++
     for  (int  i=0;  i<numRows
          matrix2[I + j*numRo
 }
```

memory3.c

```
 double **matrix2 =0;
matrix2 = (double  **)malloc(numRows*numCols*sizeof(double  *));
 double *dataPtr = matrix2;
 for  (int  j=0;  j<numCols;  j++)
     for  (int  i=0;  i<numRows;  i++)  {
          *dataPtr++  =  i;
 }
```

Special Problems: char * and Strings

- No string datatype, string in C is represented by type char *
- There are special functions for strings in <string.h>
  - strlen()
  - strcpy()
  - ….
- To use them requires a special character at end of string, namely '\0'
- This can cause no end of grief, e.g. if you use malloc, you need **size+1** and need to append '\0'

```
#include <string.h>
….
char greeting[] = "Hello";
int length = strlen(greeting);
printf("%s a string of length %d\n",greeting, length);

char *greetingCopy = (char *)malloc((length+1)*sizeof(char));
strcpy(greetingCopy, greeting);
```

WARNING

- Arrays and Pointers are the source of most bugs in C Code
  - You will have to use them if you program in C
  - Always initialize a pointer to 0
  - Be careful you do not go beyond the end of an array
    - Be thankful for segmentation faults
    - If you have a race condition (get different answers every time you run, probably a pointer issue)

What We Neglected

- File I/O
- Struct
- …. And some other stuff (not necessarily minor)
  - References
  - Operating on bits

Practice Exercises (1 hour): as many as you can

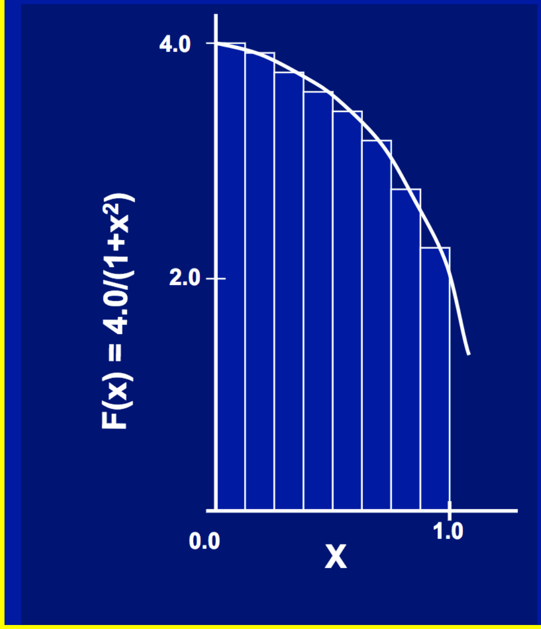
1. Write a program that when running prompts the user for two floating point numbers and returns their product.
   - i.e. ./a.out would prompt for 2 numbers a and b will output a * b = something
2. Write a program that takes a number of integer values from argc, stores them in an array, computes the sum of the array and outputs some nice message. Try using recursion to compute the sum. (hint start with recursion1.c and google function atoi(), copy from memory1.c)
   - i.e. ./a.out 3 4 5 6 will output 3 + *4 + 5 + 6 = 18*

3. *Taking the previous program. Modify it to output the number of unique numbers in the output.*
   - *i.e. ./a.out 3 1 2 1 will output 1*3 + 2*1 + 1* 2 = 7*

4. *Write a program that takes a number of input values and sorts them in ascending order.*
   - *I.e. ./a.out 2 7 4 5 9 will output* 2 4 5 7 9

Exercise: Compute PI

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

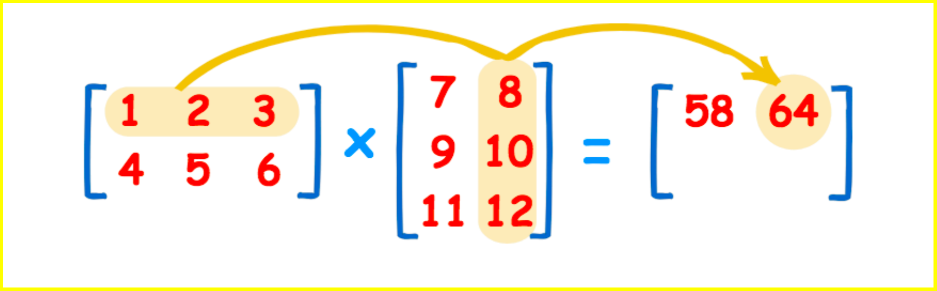
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

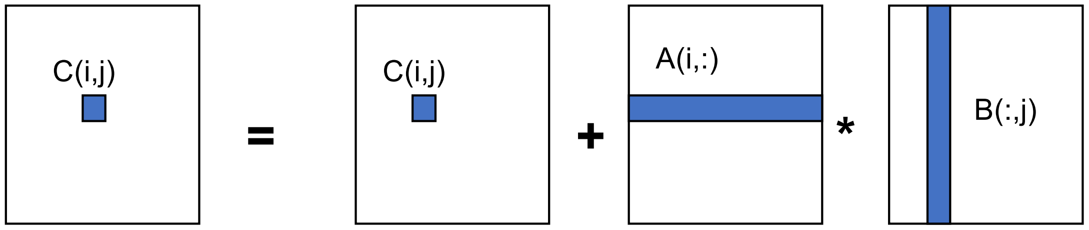
```
#include <stdio>
static int long numSteps = 100000;
int main() {
  double pi = 0; double time=0;
  // your code
  for (int i=0; i<numSteps; i++) {
     // your code
  }
   // your code
  printf("PI = %f, duration: %f ms\n",pi, time);
  return 0;
}
```

Exercise: Matrix-Matrix Multiply


$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 58 & 64 \\ & \end{bmatrix}$$

Naïve Matrix Multiply

```
{implements C = C + A*B}
for i = 1 to n
 {read row i of A into fast memory}
  for j = 1 to n
    {read C(i,j) into fast memory}
    {read column j of B into fast memory}
    for k = 1 to n
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
    {write C(i,j) back to slow memory}
```



Source: UC Berkeley, Jim Demmell, CS267

Blocked (Tiled) Matrix Multiply

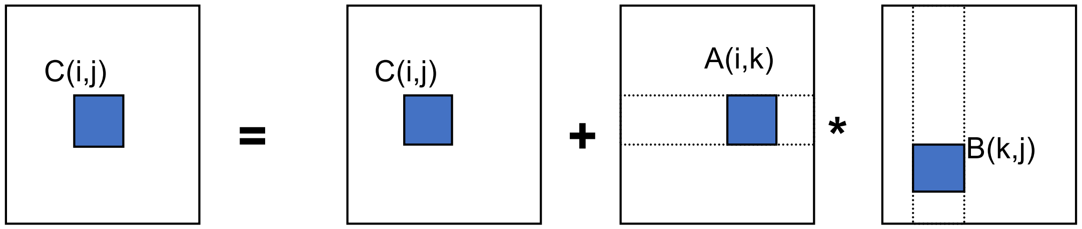
Consider A,B,C to be N-by-N matrices of b-by-b subblocks where b=n / N is block size

```
for i = 1 to N
  for j = 1 to N
     {read block C(i,j) into fast memory}
     for k = 1 to N
          {read block A(i,k) into fast memory}
          {read block B(k,j) into fast memory}
           C(i,j) = C(i,j) + A(i,k) * B(k,j) {do a matrix multiply on blocks}
     {write block C(i,j) back to slow memory}
```

cache does this automatically

3 nested loops inside

block size = loop bounds



Tiling for registers (managed by you/compiler) or caches (hardware)

Source: UC Berkeley, Jim Demmell, CS267

Recursive Matrix Multiplication (RMM) (1/2)

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = A \cdot B = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$= \begin{pmatrix} A_{11} \cdot B_{11} + A_{12} \cdot B_{21} & A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ A_{21} \cdot B_{11} + A_{22} \cdot B_{21} & A_{21} \cdot B_{12} + A_{22} \cdot B_{22} \end{pmatrix}$$

$$\begin{array}{|c|c|} \hline C_{11} & C_{12} \\ \hline C_{21} & C_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_{11} & A_{12} \\ \hline A_{21} & A_{22} \\ \hline \end{array} \bullet \begin{array}{|c|c|} \hline B_{11} & B_{12} \\ \hline B_{21} & B_{22} \\ \hline \end{array} = \begin{array}{|c|c|} \hline A_{11} * B_{11} + A_{12} * B_{21} & A_{11} * B_{12} + A_{12} * B_{22} \\ \hline A_{21} * B_{11} + A_{22} * B_{21} & A_{21} * B_{12} + A_{22} * B_{22} \\ \hline \end{array}$$

- True when each bock is a 1x1 or n/2 x n/2
- For simplicity: square matrices with $n = 2^m$
  - Extends to general rectangular case

Source: UC Berkeley, Jim Demmell, CS267

Recursive Matrix Multiplication (2/2)

```
func C = RMM (A, B, n)
   if n=1, C = A * B, else
     {  $C_{11}$ = RMM ($A_{11}$ , $B_{11}$ , n/2) + RMM ($A_{12}$ , $B_{21}$ , n/2)
        $C_{12}$ = RMM ($A_{11}$ , $B_{12}$ , n/2) + RMM ($A_{12}$ , $B_{22}$ , n/2)
        $C_{21}$ = RMM ($A_{21}$ , $B_{11}$ , n/2) + RMM ($A_{22}$ , $B_{21}$ , n/2)
        $C_{22}$ = RMM ($A_{21}$ , $B_{12}$ , n/2) + RMM ($A_{22}$ , $B_{22}$ , n/2)  }
   return
```

A(n) = # arithmetic operations in RMM(. , . , n)

$= 8 \cdot A(n/2) + 4(n/2)^2$ if $n > 1$, else 1

$= 2n^3$ … same operations as usual, in different order

W(n) = # words moved between fast, slow memory by RMM(. , . , n)

$= 8 \cdot W(n/2) + 4 \cdot 3(n/2)^2$ if $3n^2 > M_{fast}$, else $3n^2$

$= O(n^3 / (M_{fast})^{1/2} + n^2)$ … same as blocked matmul

Don't need to know M_{fast} for this to work!

Source: UC Berkeley, Jim Demmell, CS267

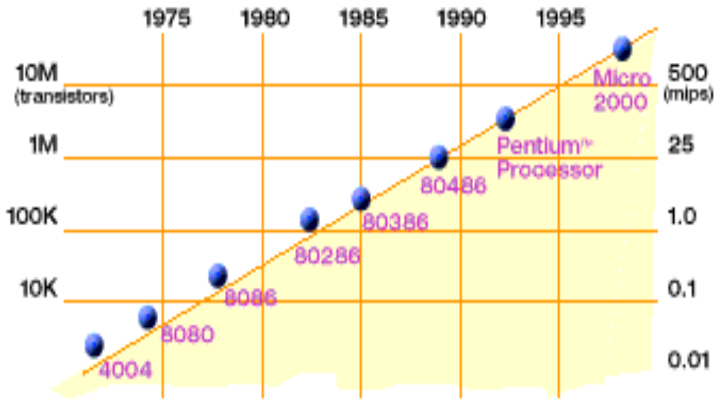
Extra Material

- Parallel Machines & Parallel Machine Models
- Parallel Programming
    - Message Passing With MPI
    - Shared Memory Programming with OpenMP

Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell

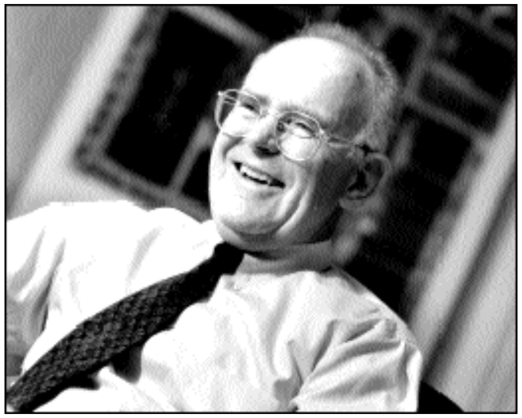
Why is Parallel Programming Important



2X transistors/Chip Every 1.5 years

Called "[Moore' s Law](#)"

Microprocessors have become smaller, denser, and more powerful.

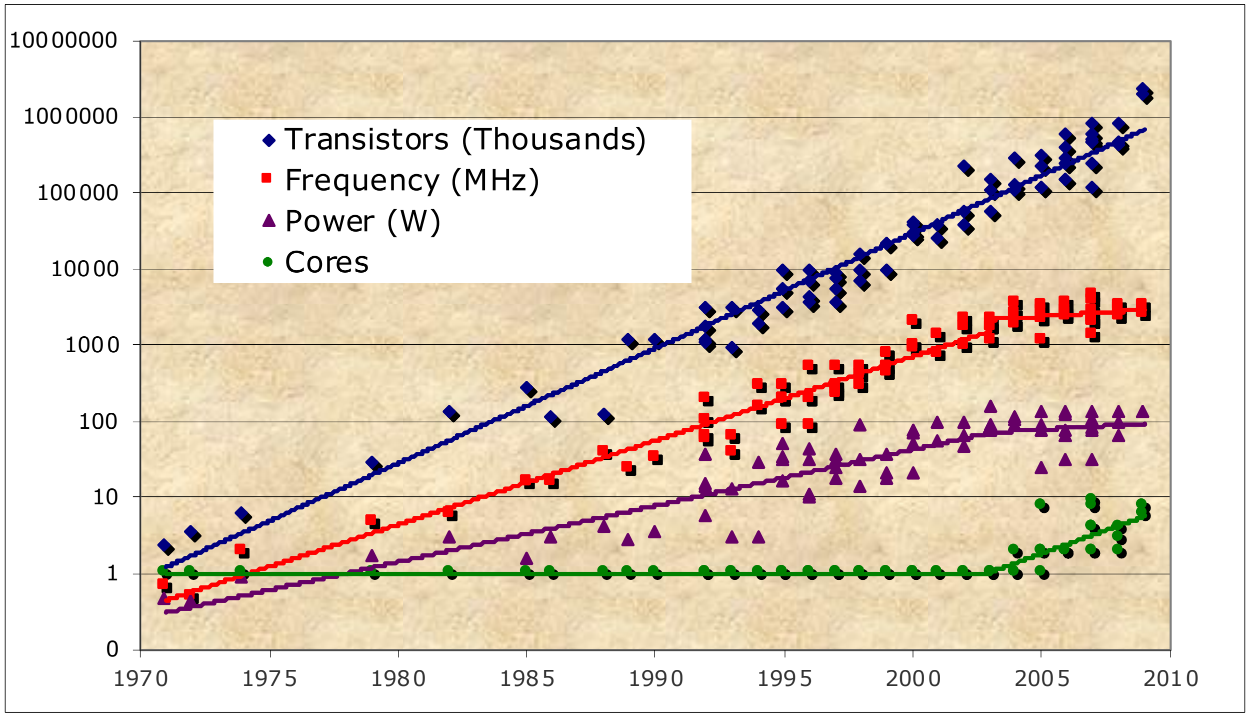


Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Slide source: Jack Dongarra

source: CS267, Jim Demmell

Revolution in Processors



- Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double instead
- Power is under control, no longer growing

source: CS267, Jim Demmell

Multiple Cores!

What does it mean for Programmers

“The Free Lunch is Over” Herb Sutter

- Up until 2003 programmers had been relying on Hardware to make their programs go faster. No longer. They had to start programming again!
- Performance now comes from Software
- To be fast and utilize the resources, Software must run in parallel, that is it must run on multiple cores at same time.

How many cores?

1,736 Intel Xeon Skylake nodes, each with 48 cores + 192GB of RAM
4,200 Intel Knights Landing nodes, each with 68 cores + 96GB of DDR RAM

TACC DESIGNSAFE-CI

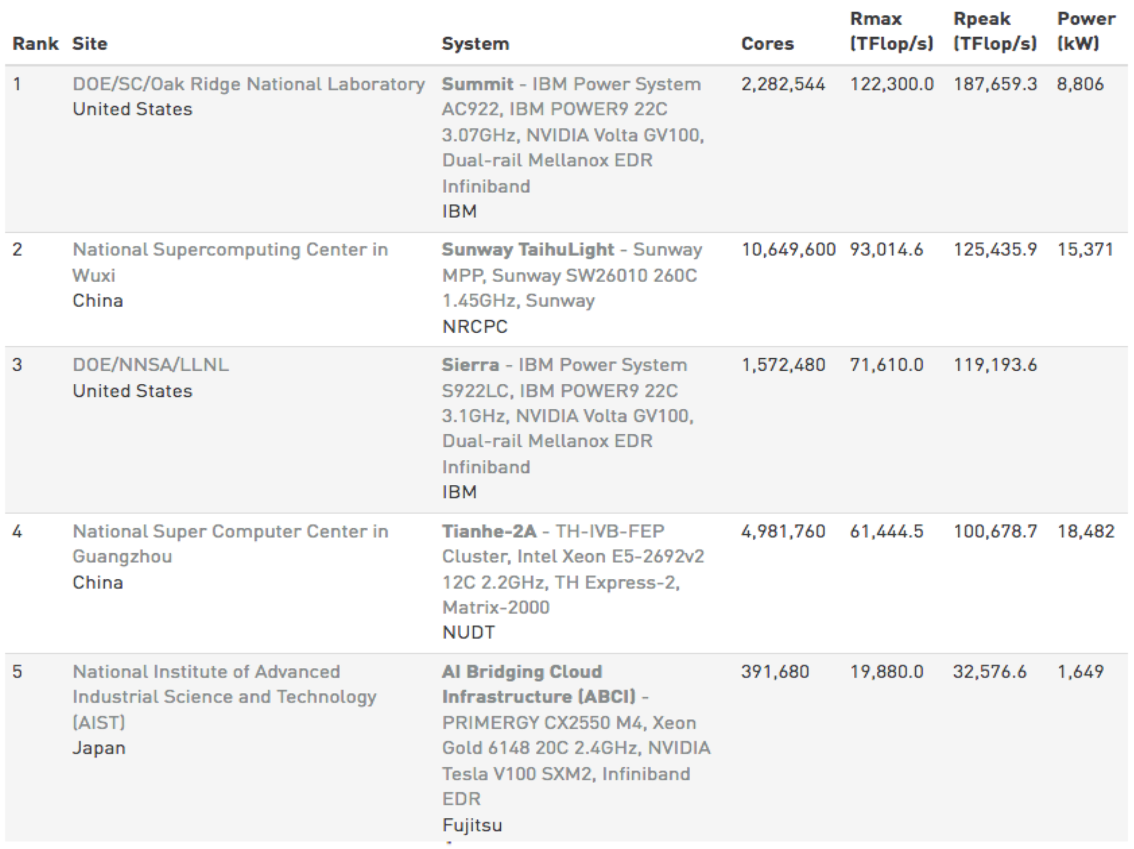
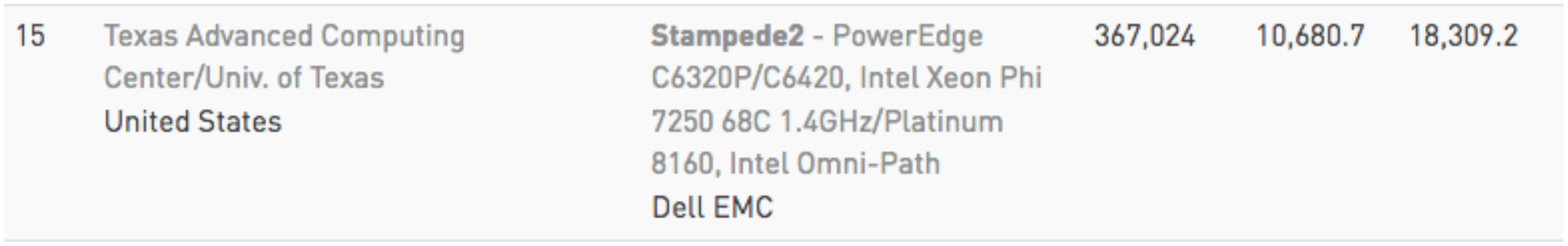
1 Intel i7 node, 6 cores + 16GB RAM

Apple A11, 6 cores + 64GB RAM

How Big & Fast!

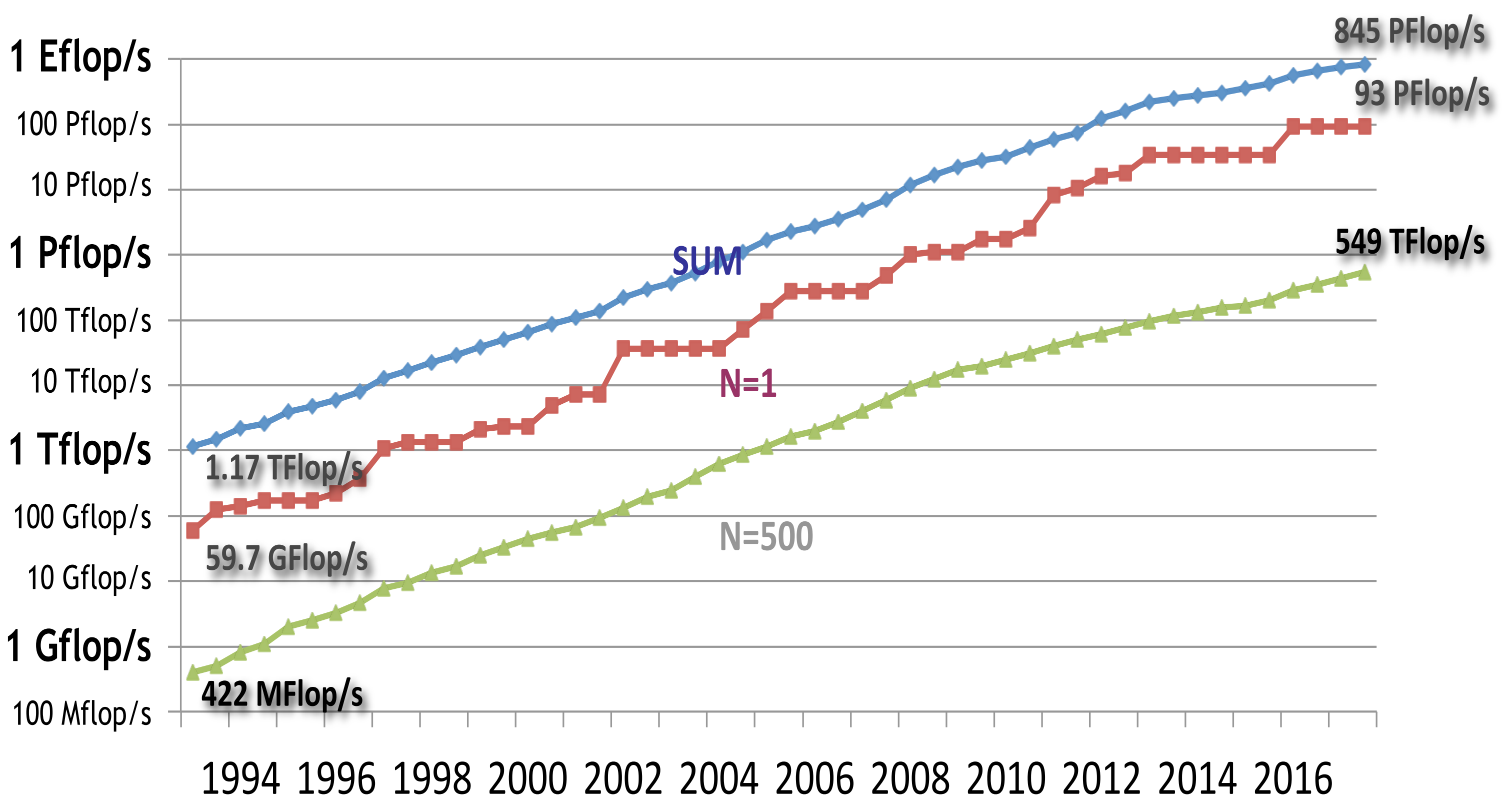
TOP500 SUPERCOMPUTER SITES

Rmax of Linpack
Solve Ax = b

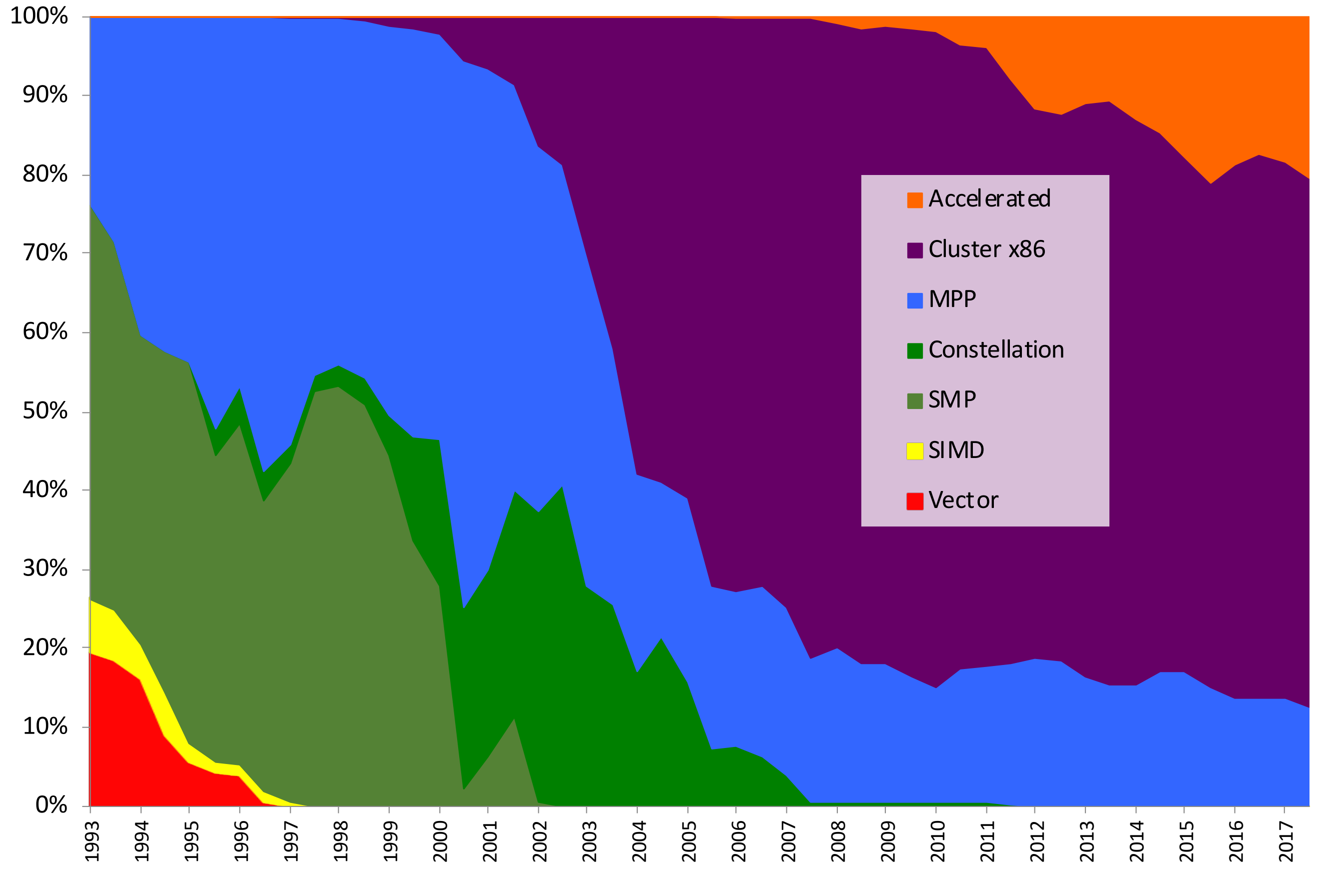



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,282,544	122,300.0	187,659.3	8,806
2	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
3	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	1,572,480	71,610.0	119,193.6	
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	National Institute of Advanced Industrial Science and Technology (AIST) Japan	**AI Bridging Cloud Infrastructure (ABCI)** - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR Fujitsu	391,680	19,880.0	32,576.6	1,649
15	Texas Advanced Computing Center/Univ. of Texas United States	**Stampede2** - PowerEdge C6320P/C6420, Intel Xeon Phi 7250 68C 1.4GHz/Platinum 8160, Intel Omni-Path Dell EMC	367,024	10,680.7	18,309.2	

Performance Development (2018)



From Vector Supercomputers to Massively Parallel Accelerator Systems



Moore’s Law reinterpreted

- Number of cores per chip can double every two years
- Clock speed will not increase (possibly decrease)
- Need to deal with systems with millions of concurrent threads
- Need to deal with inter-chip parallelism as well as intra-chip parallelism

source: CS267, Jim Demmell

Can All Programs Be Made to Run Faster?

- Suppose only part of an application can run in parallel
- Amdahl's law
  - let s be the fraction of work done sequentially, so (1-s) is fraction parallelizable
  - P = number of processors

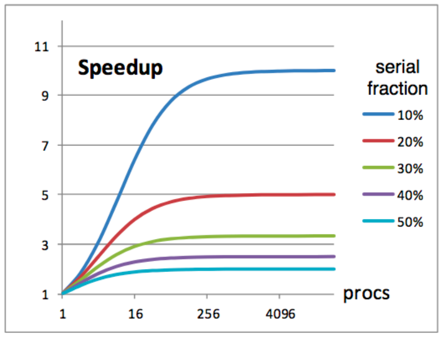
Speedup(P) = Time(1)/Time(P)

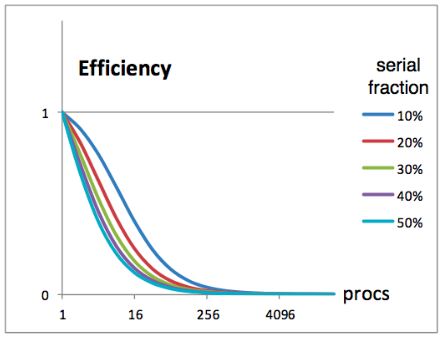
<= 1/(s + (1-s)/P)

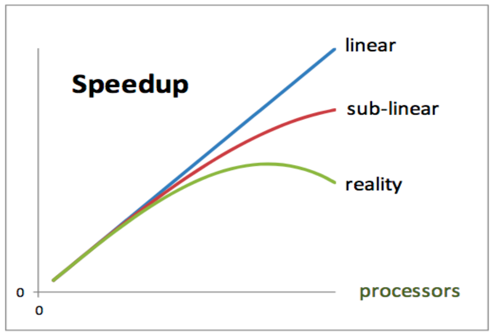
<= 1/s

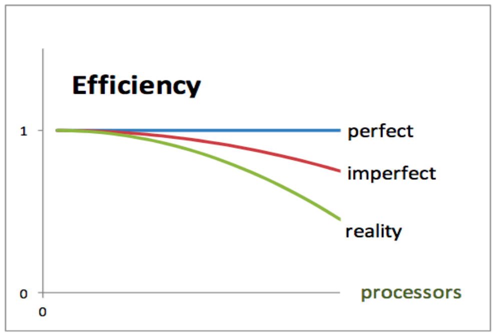
QUIZ: if 10% of program is sequential, What is the maximum speedup I Can obtain?

- **Even if the parallel part speeds up perfectly performance is limited by the sequential part**
- Top500 list: currently fastest machine has P~2.2M; Stampede2 has 367,000









Source: Doug James, TACC

This Does not Take into Account Overhead of Parallelism

- Parallelism overheads include:
  - cost of starting a thread or process
  - cost of communicating shared data
  - cost of synchronizing
  - extra (redundant) computation
- Each of these can be in the range of milliseconds (=millions of flops) on some systems
- Tradeoff: Algorithm needs sufficiently large units of work to run fast in parallel (i.e. large granularity), but not so large that there is not enough parallel work

source: CS267, Jim Demmell

Load Imbalance

- Load imbalance is the time that some processors in the system are idle due to
  - insufficient parallelism (during that phase)
  - unequal size tasks
- Examples of the latter
  - adapting to "interesting parts of a domain"
  - tree-structured computations
  - fundamentally unstructured problems
- Algorithm needs to balance load
  - Sometimes can determine work load, divide up evenly, before starting
    - "Static Load Balancing"
  - Sometimes work load changes dynamically, need to rebalance dynamically
    - "Dynamic Load Balancing," eg work-stealing

Improving Real Performance

Peak Performance grows exponentially, a la Moore's Law

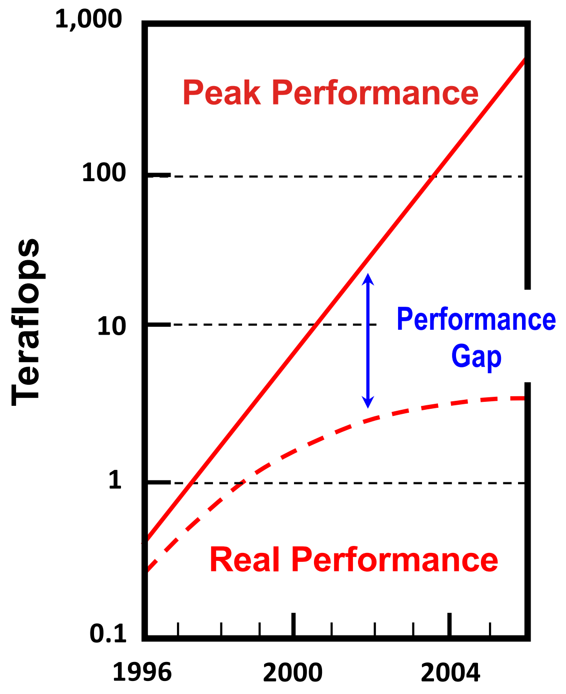
- In 1990's, peak performance increased 100x; in 2000's, it will increase 1000x

But efficiency (the performance relative to the hardware peak) has declined

- was 40-50% on the vector supercomputers of 1990s
- now as little as 5-10% on parallel supercomputers of today

Close the gap through ...

- Mathematical methods and algorithms that achieve high performance on a single processor and scale to thousands of processors
- More efficient programming models and tools for massively parallel supercomputers

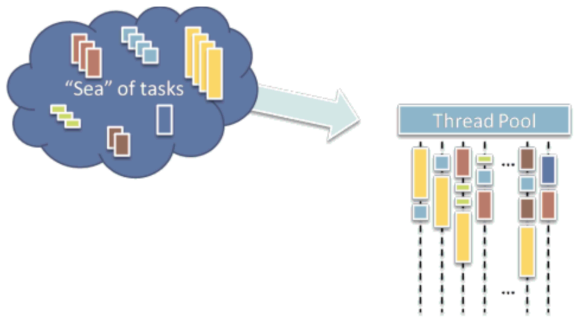


Art of Programming - II

- To take a problem, and continually break it down into a series of smaller ideally concurrent tasks until ultimately these tasks become a series of small specific individual instructions.
- Mindful of the architecture on which the program will run, identify those tasks which can be run concurrently and map those tasks onto the processing units of the target architecture.

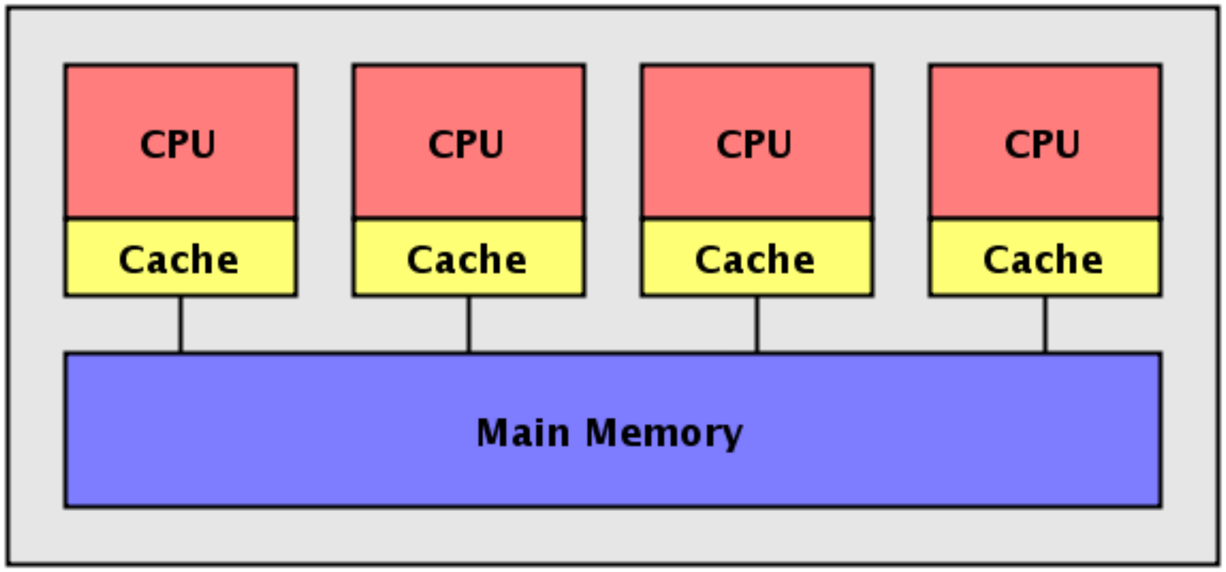
Considerations for Parallel Programming:

- Finding enough parallelism (Amdahl's Law)
- Granularity – how big should each parallel task be
- Locality – moving data costs more than arithmetic
- Load balance – don't want 1K processors to wait for one slow one
- Coordination and synchronization – sharing data safely
- Performance modeling/debugging/tuning
- Where to put the task,



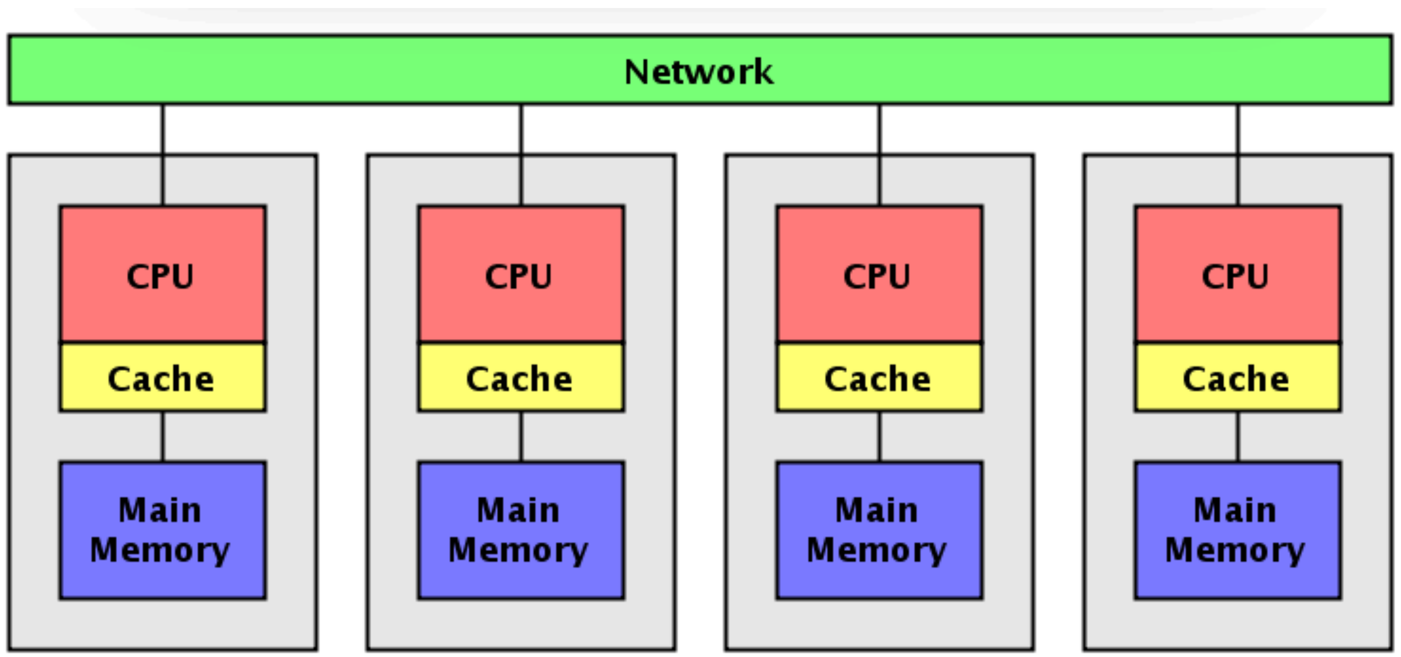
All of these things makes parallel programming even harder than sequential programming.

Simplified Parallel Machine Models



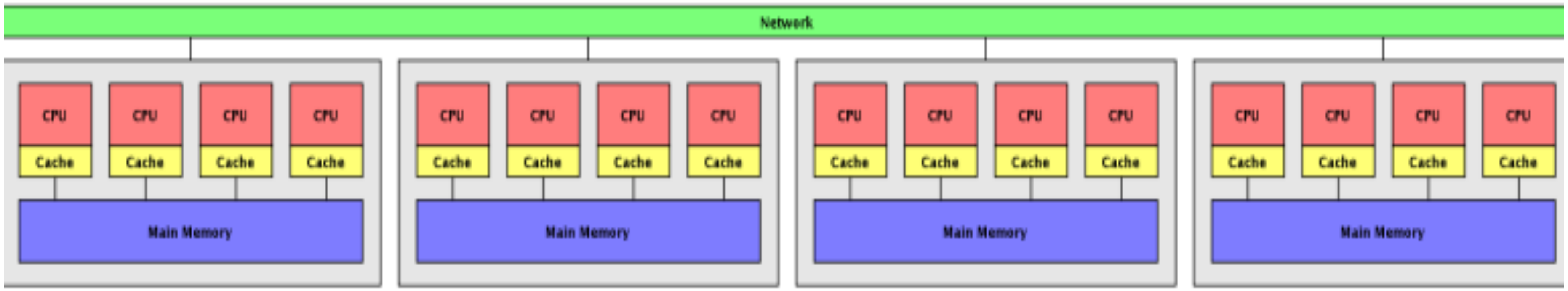
Shared Memory Model

Simplified Parallel Machine Models



Distributed Memory Model

Simplified Parallel Machine Models



Hybrid Model

Writing Programs to Run on Parallel Machines

- C Programming Libraries Exist that provides the programmer an API for writing programs that will run in parallel.
- They provide a **Programming Model** that is portable across architectures, i.e. can provide a message passing model that runs on a shared memory machine.
- We will look at 2 of these Programming Models and Libraries that support the model:
  - Message Passing Programming using MPI (message passing interface)
  - Thread Programming using OpenMP
- As will all libraries they can incur an overhead.

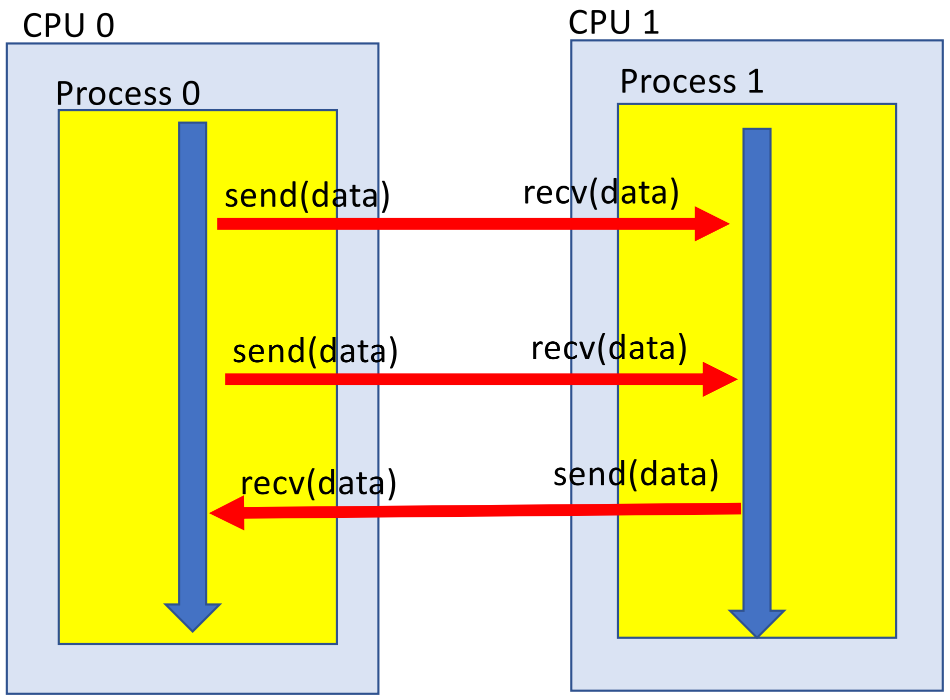
- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP

Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell

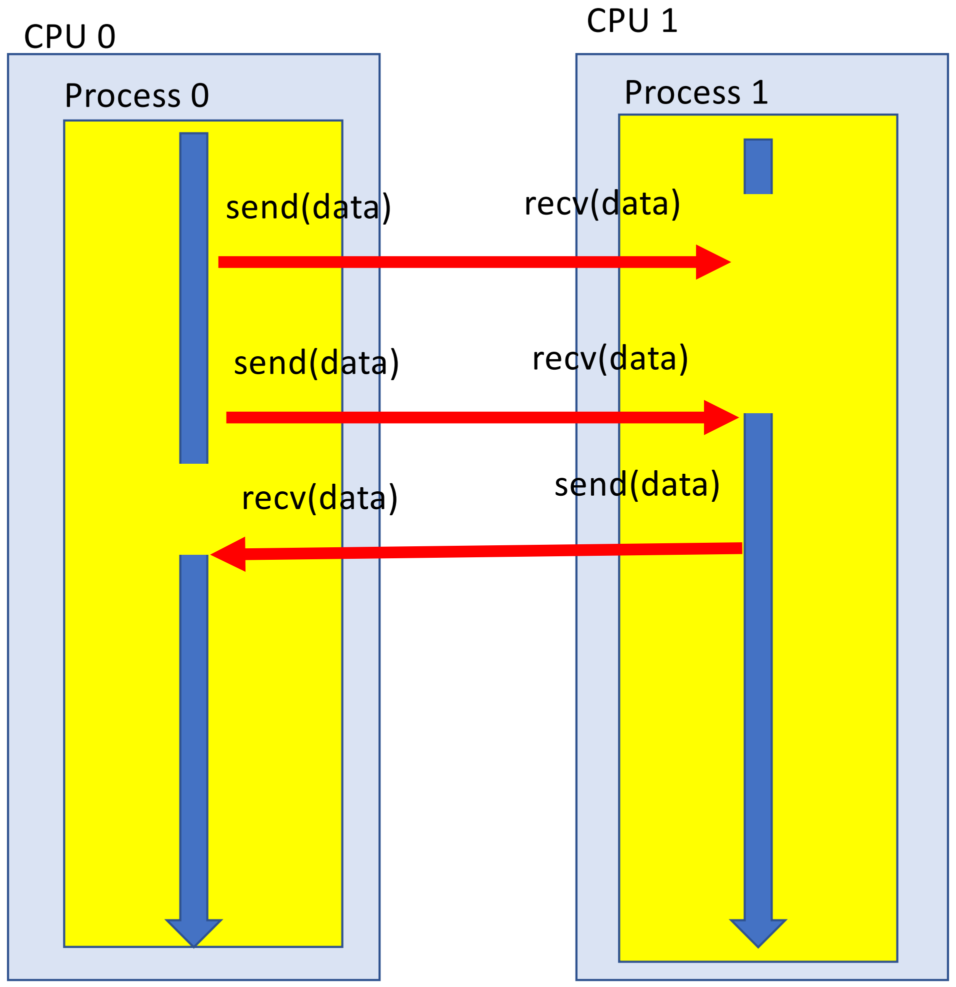
Message Passing Model

- Processes run independently in their own memory space and processes communicate with each other when data needs to be shared

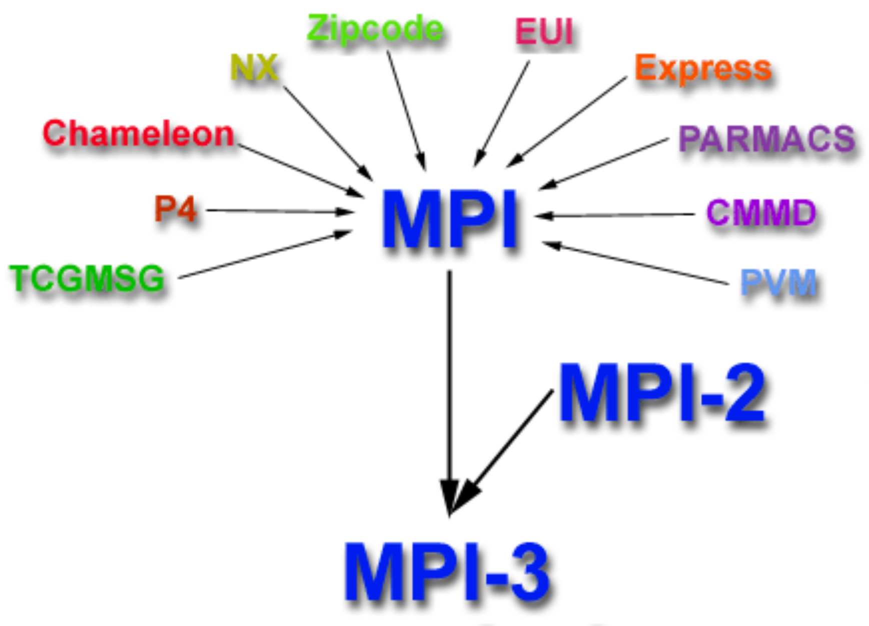


- Basically you write sequential applications with additional function calls to send and recv data.

Only get Speedup if processes can be kept busy



Programming Libraries



- Coalasced around a single standard MPI
- Allows for portable code

MPI

Provides a number of functions:

1. Enquiries
   - How many processes?
   - Which one am I?
   - Any messages Waiting?
2. Communication
   - Pair-wise point to point send and receive
   - Collective/Group: Broadcast, Scatter/Gather
   - Compute and Move: sum, product, max …
3. Synchronization
   - Barrier

REMEMBER:

What I am about to show is just C code with some functions you have not yet seen.

Hello World

Code/Parallel/mpi/hello1.c

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

int main( int argc, char **argv)
{
    int procID, numP;

    MPI_Init( &argc, &argv );
    MPI_Comm_size( MPI_COMM_WORLD, &numP );
    MPI_Comm_rank( MPI_COMM_WORLD, &procID );

    printf( "Hello World, I am %d of %d\n", procID, numP );
    MPI_Finalize();
    return 0;
}
```

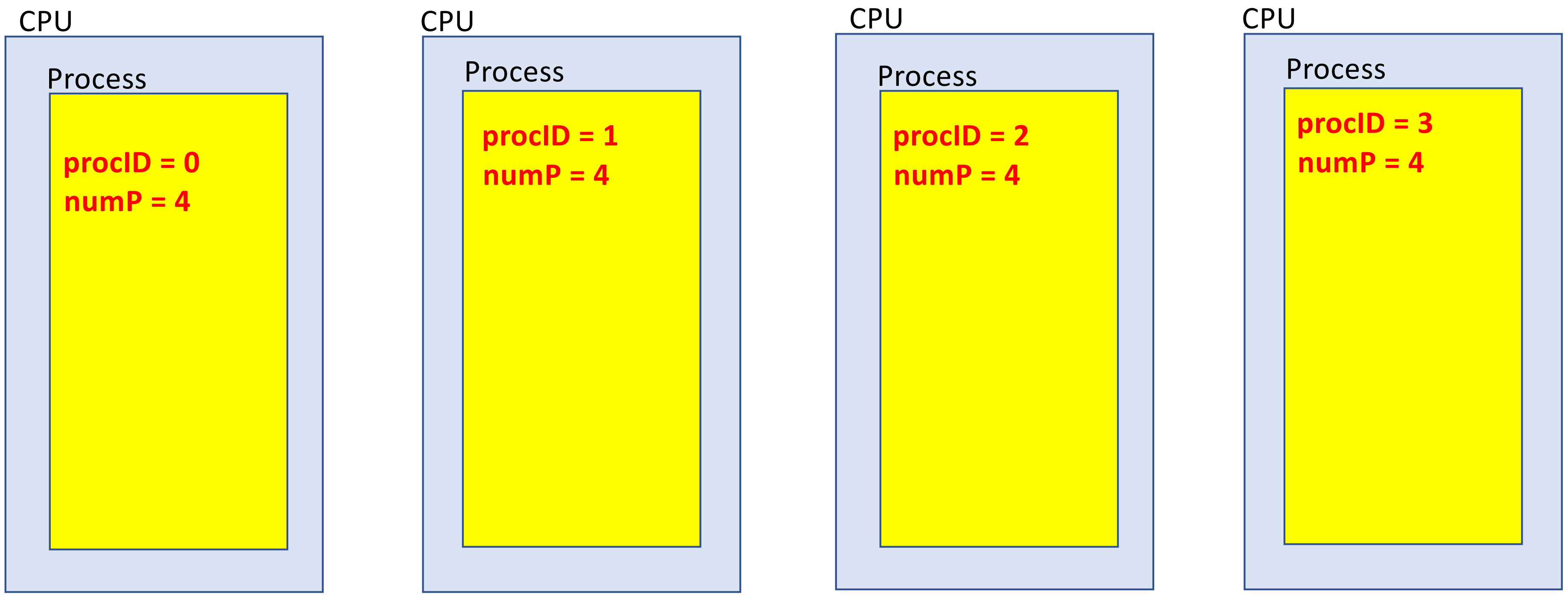
MPI functions (and MPI_COMM_WORLD) are defined in mpi.h

MPI_Init() and MPI_finalize() must be first and last functions called

MPI_COMM_WORLD is a default group contaning all processes

MPI_Comm_size returns # of processes in the group

MPI_Comm_rank returns processes unique ID the group, 0 through (numP-1)


CPU
Process
procID = 0
numP = 4
CPU
Process
procID = 1
numP = 4
CPU
Process
procID = 2
numP = 4
CPU
Process
procID = 3
numP = 4

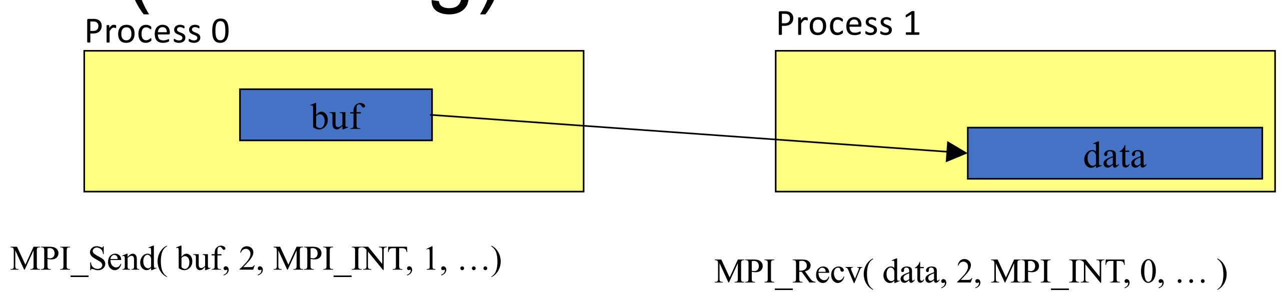
Send/Recv blocking

Code/Parallel/mpi/send1.c

```
#include <mpi.h>
#include <stdio.h>
int main( int argc, char **argv) {
  int procID;
  MPI_Status status;
  MPI_Init(&argv, &argc);
  MPI_Comm_rank( MPI_COMM_WORLD, &procID );

  if (procID == 0) { // process 0 sends
    int buf[2] = {12345, 67890};
    MPI_Send( &buf, 2, MPI_INT, 1, 0, MPI_COMM_WORLD);
  }
  else if (procID == 1) { // process 1 receives

    int data[2];
    MPI_Recv( &data, 2, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );
    printf( "Received %d %d\n", data[0], data[1] );
  }

  MPI_Finalize();
  return 0;
}
```

NOTE the PAIR of Send/Recv

MPI Basic (Blocking) Send

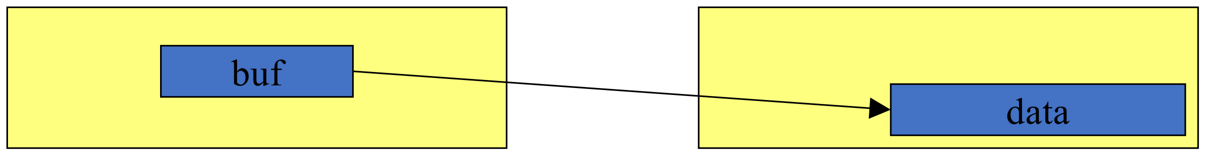

Process 0
Process 1
buf
data
MPI_Send(buf, 2, MPI_INT, 1, …)
MPI_Recv(data, 2, MPI_INT, 0, …)

`MPI_SEND(start, count, datatype, dest, tag, comm)`

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**. The message has an identifier **tag.**
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI_CHAR	char
MPI_WCHAR	wchar_t - wide character
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_LONG_LONG_INT **MPI_LONG_LONG**	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX **MPI_C_FLOAT_COMPLEX**	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_C_LONG_DOUBLE_COMPLEX	long double _Complex
MPI_C_BOOL	_Bool
MPI_INT8_T **MPI_INT16_T** **MPI_INT32_T** **MPI_INT64_T**	int8_t int16_t int32_t int64_t
MPI_UINT8_T **MPI_UINT16_T** **MPI_UINT32_T** **MPI_UINT64_T**	uint8_t uint16_t uint32_t uint64_t
MPI_BYTE	8 binary digits
MPI_PACKED	data packed or unpacked with MPI_Pack()/ MPI_Unpack

MPI Basic (Blocking) Receive



MPI_Send(buf, 2, MPI_INT, 1, …)

MPI_Recv(data, 2, MPI_INT, 0, …)

`MPI_RECV(start, count, datatype, source, tag, comm, status)`

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system, and the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

Retrieving Further Information

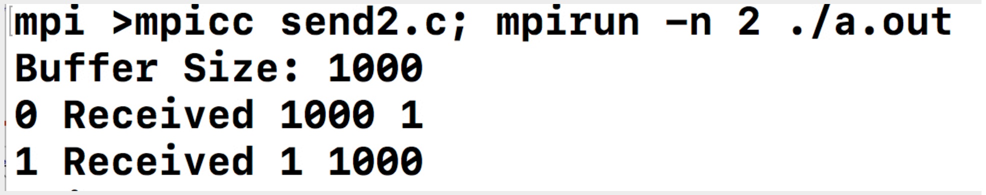
- **`Status`** is a data structure allocated in the user' s program.
- In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, ..., &status )
recvd_tag  = status.MPI_TAG;
recvd_from = status.MPI_SOURCE;
MPI_Get_count( &status, datatype, &recvd_count );
```

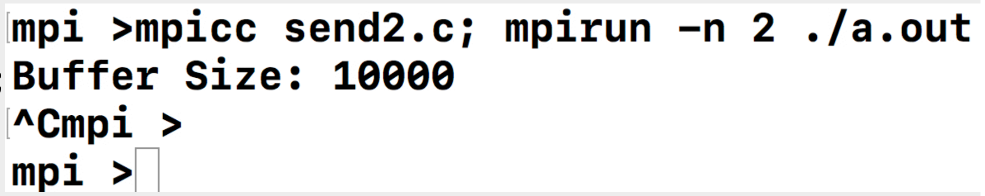
Not Quite So Simple as ensuring PAIRS of send/recv

Code/Parallel/mpi/send2.c

```c
#include <mpi.h>
#include <stdio.h>
#define DATA_SIZE 1000
int main(int argc, char **argv) {
   int procID, numP;
   MPI_Status status;
   int buf[DATA_SIZE];

   MPI_Init( &argc, &argv );
   MPI_Comm_size( MPI_COMM_WORLD, &numP );
   MPI_Comm_rank( MPI_COMM_WORLD, &procID );

   If (procID == 0) {
      for (int i=0; i<DATA_SIZE; i++) buf[i]=1+i;
      MPI_Send(&buf, DATA_SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD);
      MPI_Recv(&buf, DATA_SIZE, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
      printf("%d Received %d %d\n", procID, buf[0], buf[DATA_SIZE-1]);
   } else if (procID == 1) {
      for (int i=0; i<DATA_SIZE; i++) buf[i]=DATA_SIZE-i;
      MPI_Send(&buf, DATA_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD);
      MPI_Recv(&buf, DATA_SIZE, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
     printf("%d Received %d %d\n", procID, buf[0], buf[DATA_SIZE-1]);
   }
   MPI_Finalize();
   return 0;
}
```



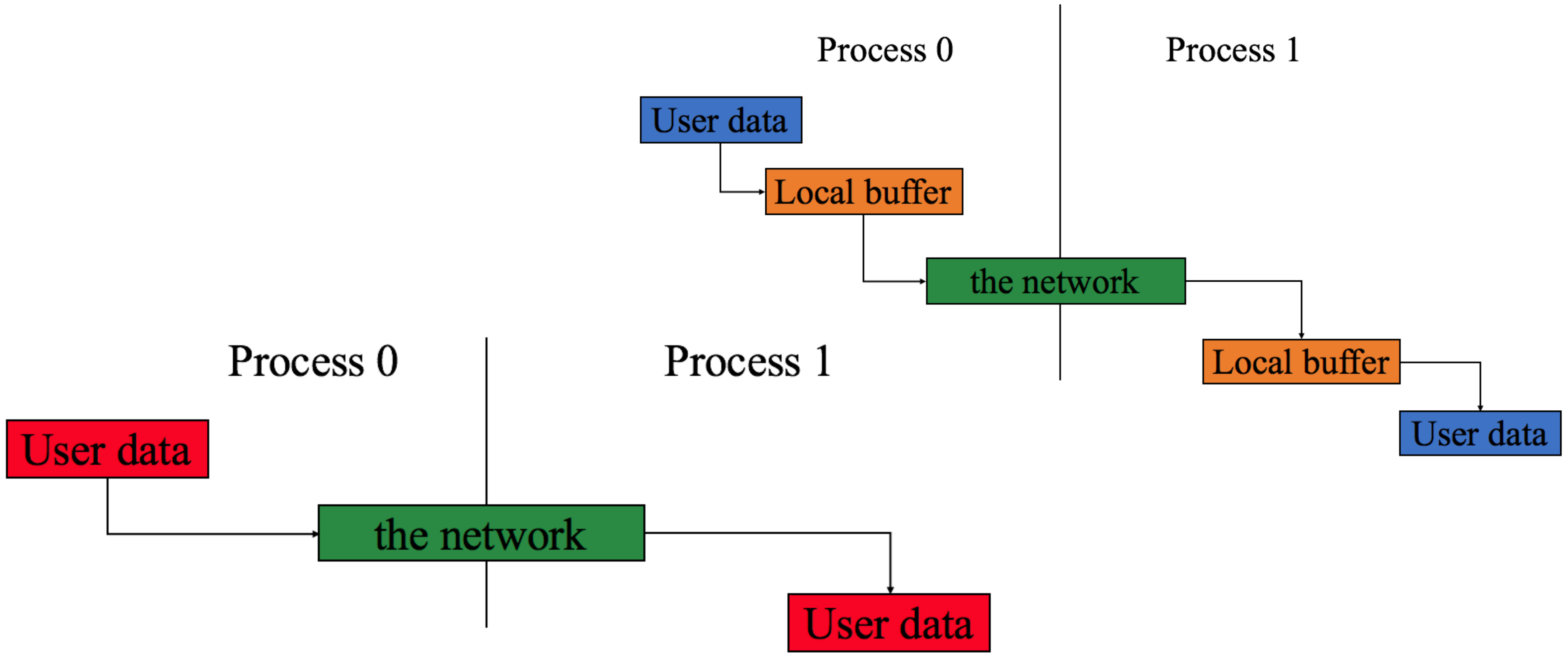
```
mpi >mpicc send2.c; mpirun -n 2 ./a.out
Buffer Size: 1000
0 Received 1000 1
1 Received 1 1000
```



```
mpi >mpicc send2.c; mpirun -n 2 ./a.out
Buffer Size: 10000
^Cmpi >
mpi >
```

DEADLOCK .. PROGRAM HANGS .. WHY?

Why Deadlock? .. Where Does the Data Go



If large message & insufficient data, the send() must wait for buffer to clear through a recv()

source: CS267, Jim Demmell

Current Problem:

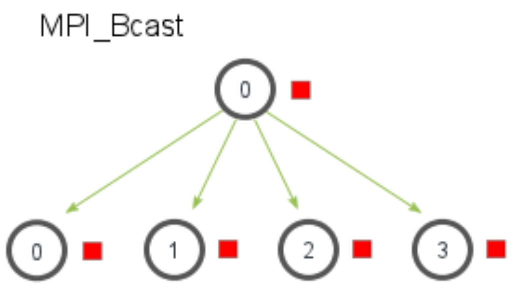
Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

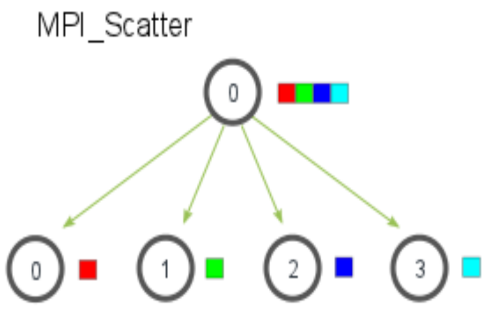
Could revise order
this requires some code rewrite:

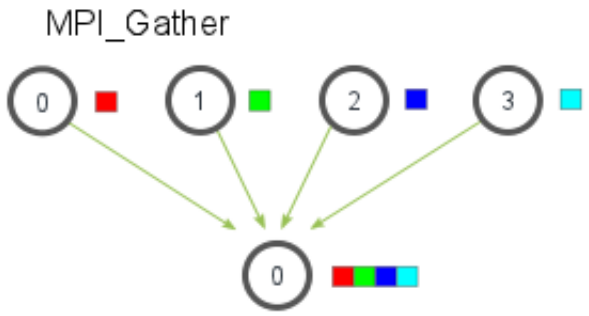
Process 0	Process 1
`Send(1)`	`Recv(0)`
`Recv(1)`	`Send(0)`

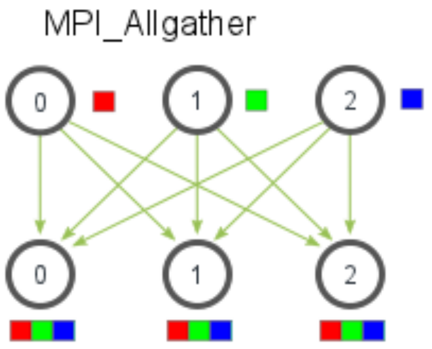
Alternatives use non-blocking sends.

Some Collective Functions





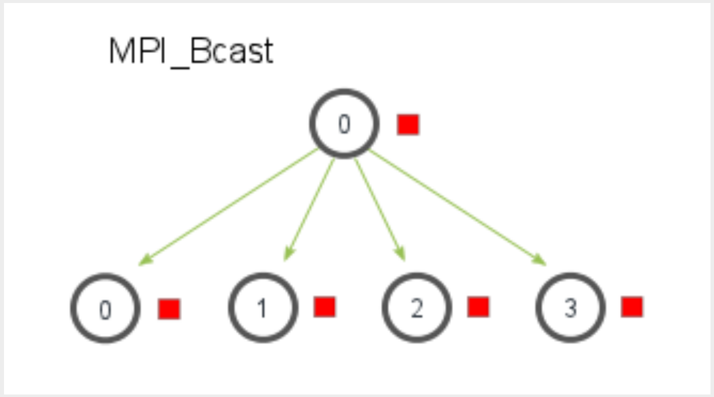




Broadcast

Code/Parallel/mpi/bcast1.c

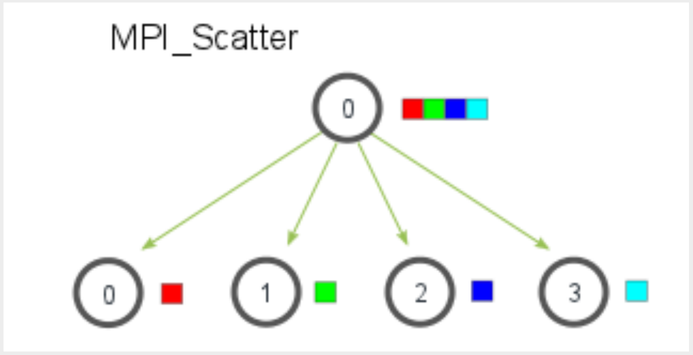
```
#include <mpi.h>
#include <stdio.h>

int main( int argc, char **argv) {
   int procID; buf[2];

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &procID );

    if (procID == 0) {
       buf[0] = 12345;
       buf[1] = 67890;
    }

    MPI_Bcast(&buf, 2, MPI_INT, 0, MPI_COMM_WORLD);

    printf("Process %d data %d %d\n", procID, buf[0], buf[1]);

    MPI_Finalize();
    return 0;
}
```



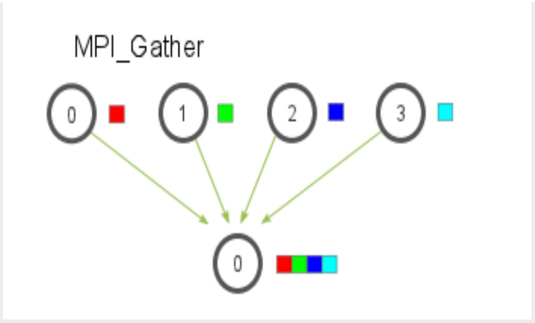
Scatter

Code/Parallel/mpi/scatter1.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define LUMP 5
int main(int argc, char **argv) {
   int numP, procID;
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numP);
   MPI_Comm_rank(MPI_COMM_WORLD, &procID);

  int *globalData=NULL;
  int localData[LUMP];

  if (procID == 0) { // malloc and fill in with data
    globalData = malloc(LUMP * numP * sizeof(int) );
    for (int i=0; i<LUMP*numP; i++)
      globalData[i] = i;
  }
  MPI_Scatter(globalData, LUMP, MPI_INT, &localData, LUMP, MPI_INT, 0, MPI_COMM_WORLD);
  printf("Processor %d has first: %d last %d\n", procID, localData[0], localData[LUMP-1]);

  if (procID == 0)  free(globalData);

  MPI_Finalize();
}
```



```
#include “mpi.h”
#include <stdio.h>
#define LUMP 5
int main(int argc, const charr **argv) {
     int procID, numP, ierr;
     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &numP);
     MPI_Comm_rank(MPI_COMM_WORLD, &procID);

      int *globalData=NULL;
      int localData[LUMP];
      if (procID == 0) { // malloc global data array only on P0
         globalData = malloc(LUMP * numP * sizeof(int) );
      }
      for (int i=0; i<LUMP; i++)
            localData[i] = procID*10+i;

      MPI_Gather(localData, LUMP, MPI_INT, globalData, LUMP, MPI_INT, 0, MPI_COMM_WORLD);

      if (procID == 0) {
          for (int i=0; i<numP*LUMP; i++)
                 printf("%d ", globalData[i]);
          printf("\n");
      }
      if (procID == 0) free(globalData);
      MPI_Finalize();
```



MPI can be simple

- Claim: most MPI applications can be written with only 6 functions (although which 6 may differ)

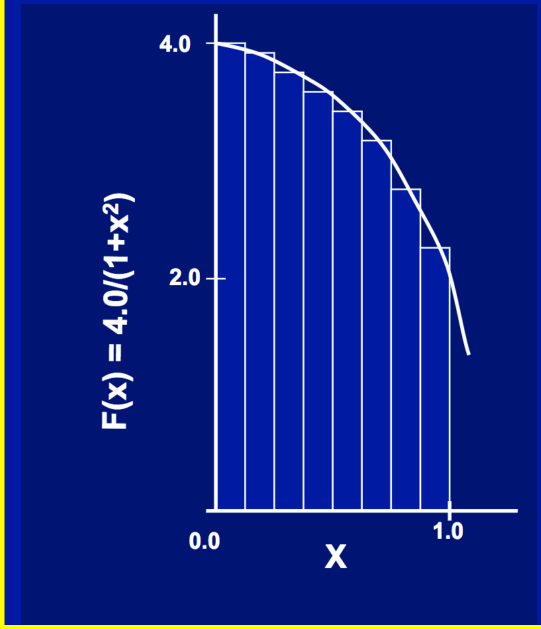
- Using point-to-point:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_SEND`
  - `MPI_RECEIVE`

- Using collectives:
  - `MPI_INIT`
  - `MPI_FINALIZE`
  - `MPI_COMM_SIZE`
  - `MPI_COMM_RANK`
  - `MPI_BCAST/MPI_SCATTER`
  - `MPI_GATHER/MPI_ALLGATHER`

- You may use more for convenience or performance

Exercise: Parallelize Compute PI using MPI

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\,dx = \pi$$

We can approximate the integral as a sum of rectangles:

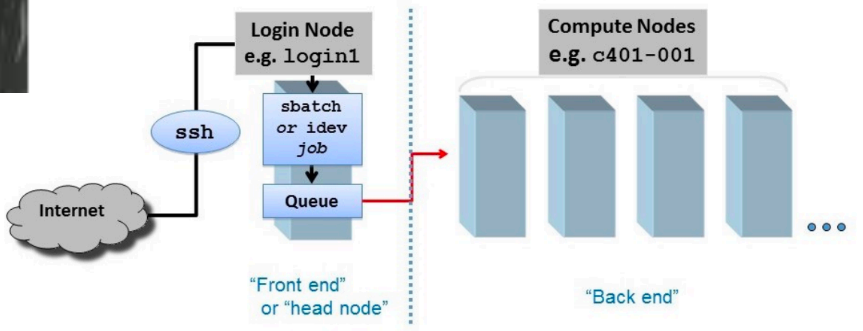
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

```
#include <stdio>
static int long numSteps = 100000;
int main() {
  double pi = 0; double time=0;
  // your code
  for (int i=0; i<numSteps; i++) {
     // your code
  }
   // your code
  printf("PI = %f, duration: %f ms\n",pi, time);
  return 0;
}
```

Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

Stampede2


Login Node
e.g. login1
Compute Nodes
e.g. c401-001
ssh
sbatch
or idev
job
Queue
Internet
"Front end"
or "head node"
"Back end"

sbatch script

```
#!/bin/bash
#----------------------------------------------------------------------
# Generic SLURM script – MPI Hello World
#
# This script requests 1 node and 8 cores/node (out of total 64 avail)
# for a total of 1*8 = 8 MPI tasks.
#----------------------------------------------------------------------
#SBATCH -J myjob          # Job name
#SBATCH -o myjob.%j.out   # stdout; %j expands to jobid
#SBATCH -e myjob.%j.err   # stderr; skip to combine stdout and stderr
#SBATCH -p development    # queue
#SBATCH -N 1              # Number of nodes, not cores (64 cores/node)
#SBATCH -n 8              # Total number of MPI tasks (if omitted, n=N)
#SBATCH -t 00:00:10       # max time

module petsc    # load any needed modules, these just examples
module load list

ibrun ./a.out
```

1. Put your pi.c into your github repository and push it to your fork

```
cp pi.c ~/SimCenterBootcamp/Code/Parallel/mpi/pi.c
cd  ~/SimCenterBootcamp/Code/Parallel/mpi
git add pi.c
git commit –m “pi.c initial import”
git push
```

2. Login to stampede2

```
ssh yourlogin@stampede2.tacc.utexas.edu
```

3. Now on stampede2 login node, clone the repository on stampede2

```
git clone http://???????
```

4. cd to the ~/SimCenterBootCamp/Code/Parallel/mpi directory

```
cd  ~/SimCenterBootcamp/Code/Parallel/mpi
```

5. Compile hello1.c or whatever you want

```
mpicc hello1.c
```

6. Submit the job to SLURM queue

```
sbatch submit.sh
```

7. Look at the output

```
cat myjob.*.out
```

8. Make changes to your code, compile & run!

Your on your own

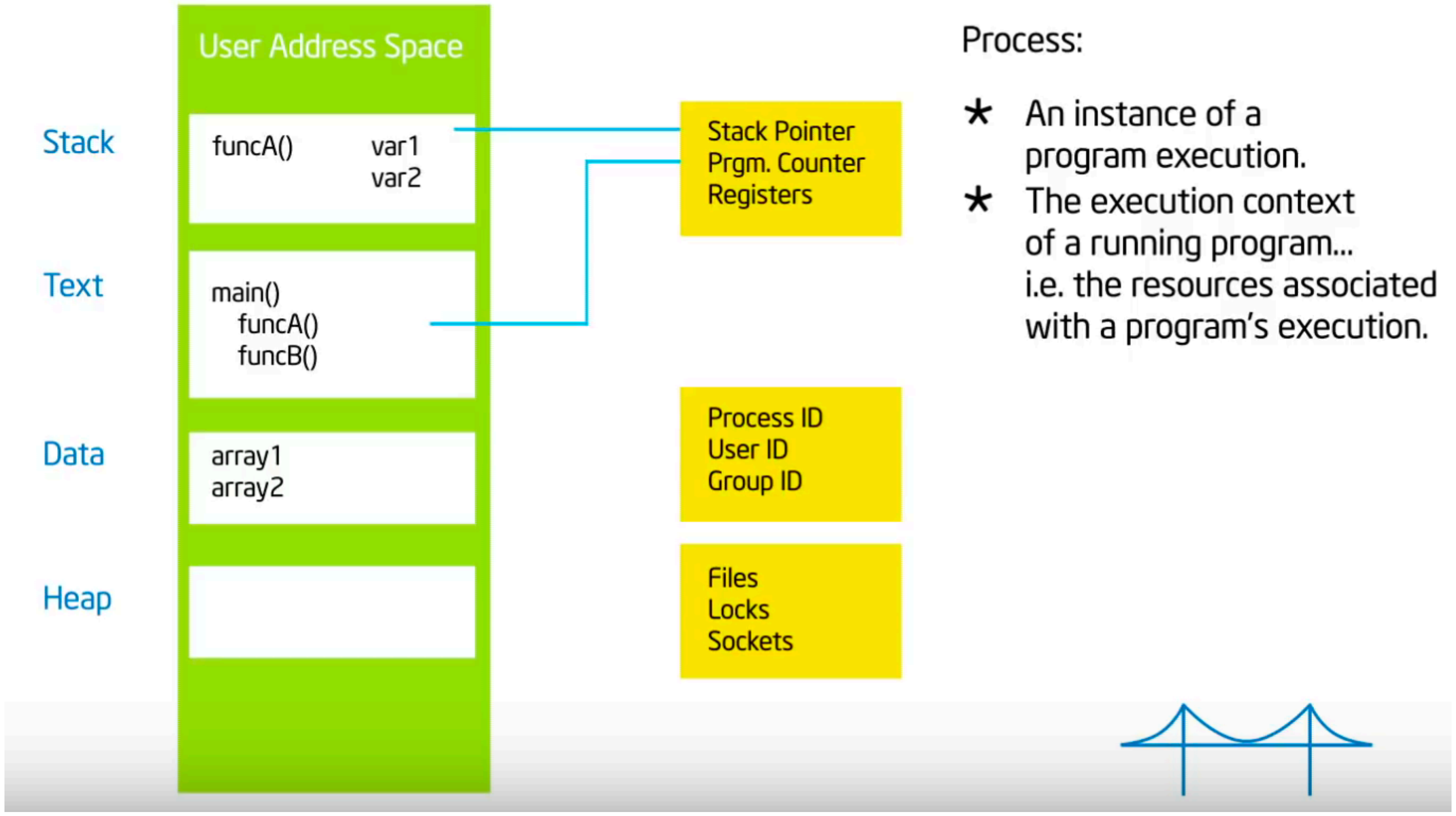
Outline

- Parallel Machines & Parallel Machine Models
- Parallel Programming
  - Message Passing With MPI
  - Shared Memory Programming with OpenMP

Ignoring co-processors and GPUs

many slides source: CS267, Jim Demmell

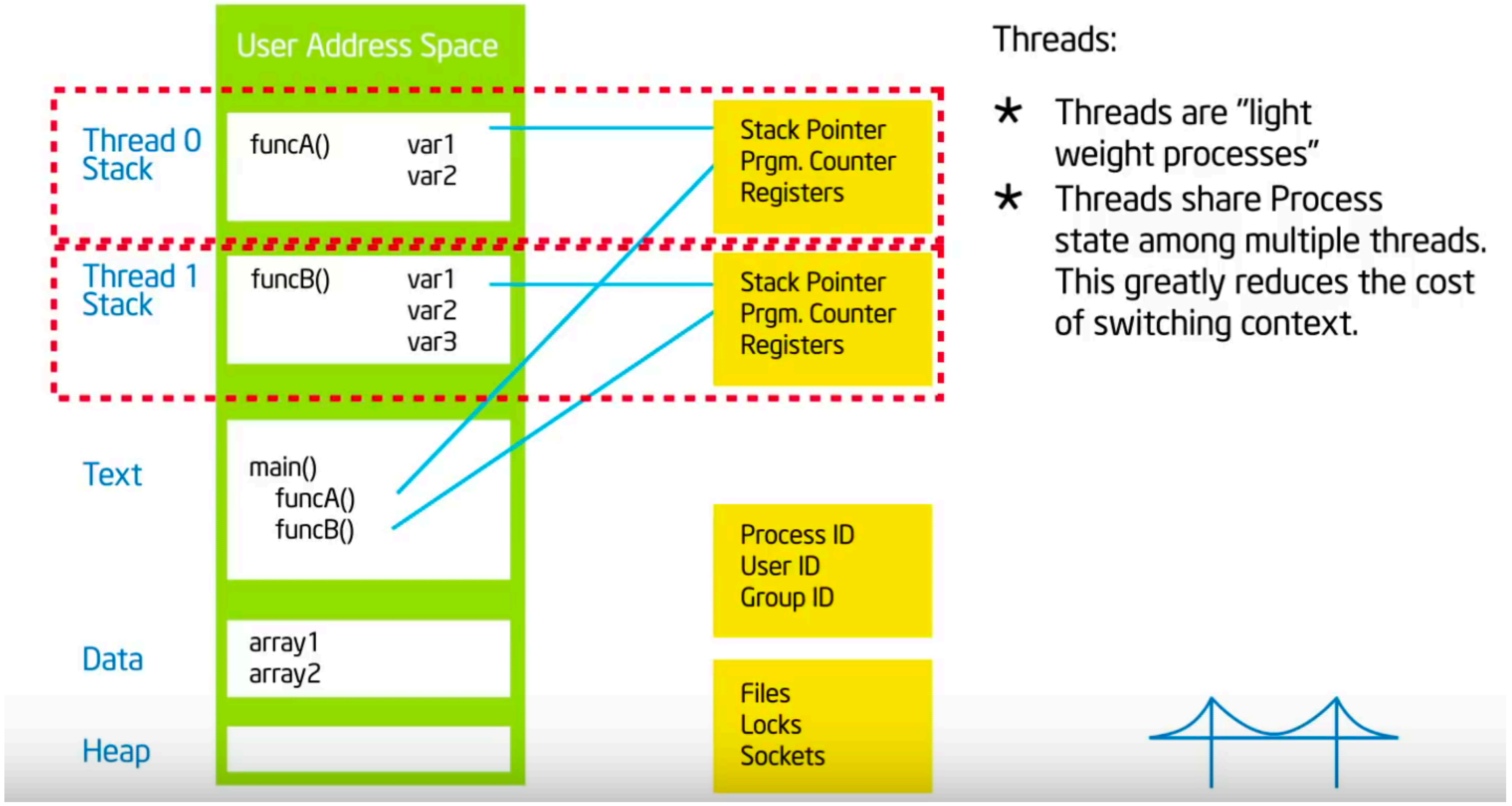
Process



Process:

- An instance of a program execution.
- The execution context of a running program... i.e. the resources associated with a program's execution.

Threads

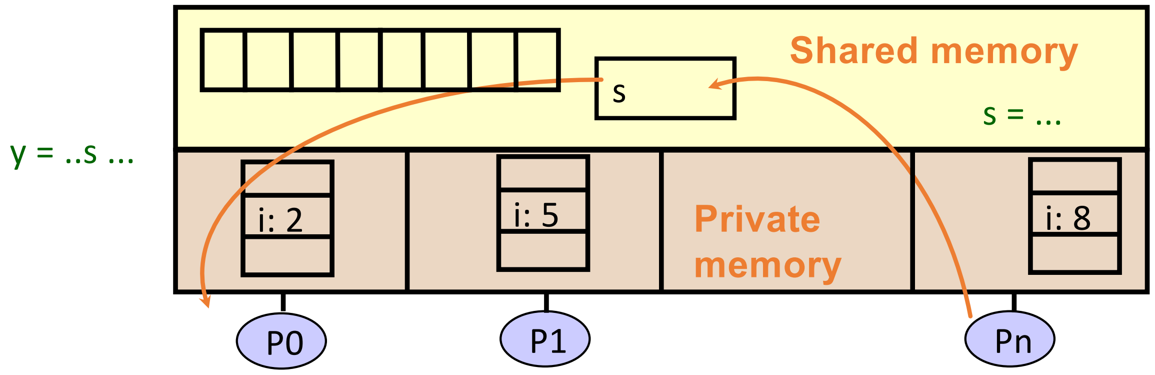


Threads:

* Threads are "light weight processes"
* Threads share Process state among multiple threads. This greatly reduces the cost of switching context.

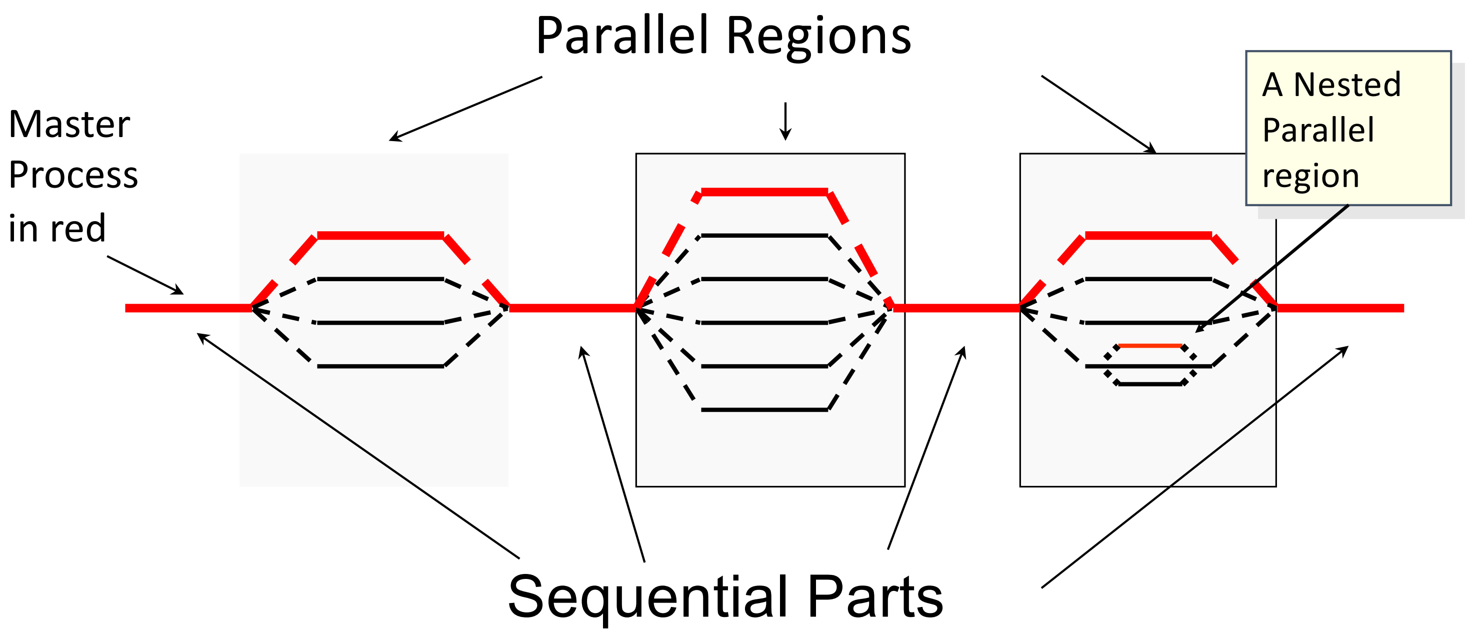
Threads

- Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap.
  - Threads communicate implicitly by writing and reading shared variables.
  - Threads coordinate by synchronizing on shared variables



Programming Model for Threads

- Master Process spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met, i.e., the sequential program evolves into a parallel program.



Runtime Library Options for Shared Memory

POSIX Threads (pthreads)
OpenMP

- OpenMP provides multi-threaded capabilities to C, C++ and Fortran Programs
- In a threaded environment, threads communicate by sharing data
- Unintended sharing of data causes **race conditions**
- **Race Condition**: program output is different every time you run the program, a consequence of the threads being scheduled differently
- OpenMP provides constructs to control what blocks of code are run in parallel and also constructs for providing access to shared data using synchronization
- Synchronization has overhead consequences, you have to minimize them to get good speedup.

- Mostly Set of Compiler directives (#pragma) applying to structured block

```
#pragma omp parallel
```

- Some runtime library calls

```
omp_num_threads(4);
```

- Being compiler directives, they are built into most compilers .
- Just have to activate it when compiling

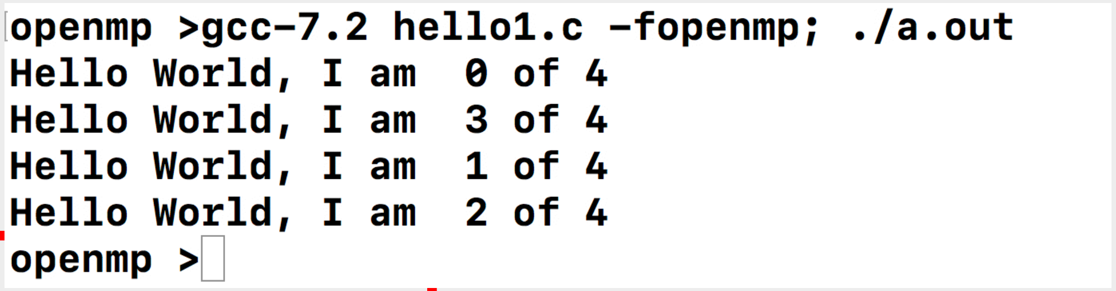
```
gcc hello.c -fopenmp
```

```
icc hello.c /Qopenmp
```

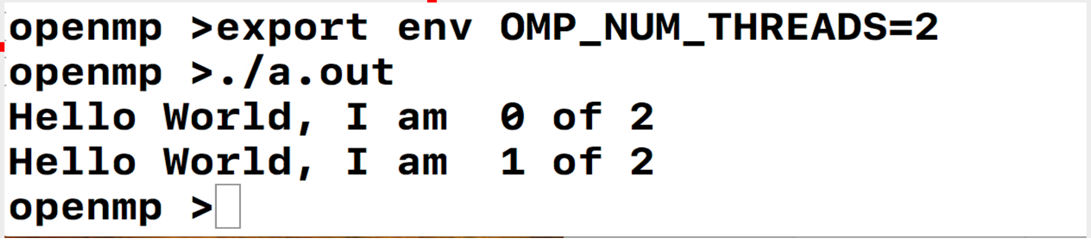
Hello World

Code/Parallel/openmp/hello1.c

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

int main( int argc, char *argv[] )
{
  #pragma omp parallel
  {
     int id = omp_get_thread_num();
     int numP = omp_get_num_threads();
     printf("Hello World, I am  %d of %d\n",id,numP);
  }
  return 0;
}
```



```
openmp >gcc-7.2 hello1.c -fopenmp; ./a.out
Hello World, I am  0 of 4
Hello World, I am  3 of 4
Hello World, I am  1 of 4
Hello World, I am  2 of 4
openmp >
```



```
openmp >export env OMP_NUM_THREADS=2
openmp >./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
openmp >
```

Hello World – changing num threads

Code/Parallel/openmp/hello2.c

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

int main( int argc, char *argv[] )
{
  omp_set_num_threads(2);
  #pragma omp parallel
   {
        int id = omp_get_thread_num();
        int numP = omp_get_num_threads();
        printf("Hello World, I am  %d of
   }
  return 0;
}
```

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

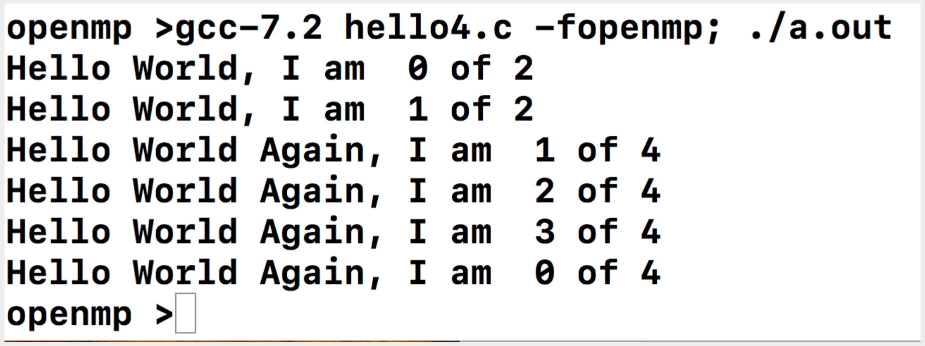
```c
#include <openmp.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
#pragma omp parallel num_threads(2)
   {
        int id = omp_get_thread_num();
        int numP = omp_get_num_threads();
        printf("Hello World, I am  %d of %d\n",id,numP);
   }
  return 0;
}
```

Code/Parallel/openmp/hello3.c

Different # threads in different blocks

Code/Parallel/openmp/hello4.c

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

int main(int argc, const char **argv) {

#pragma omp parallel num_threads(2)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World, I am  %d of %d\n",id,numP);
  }
#pragma omp parallel num_threads(4)
  {
    int id = omp_get_thread_num();
    int numP = omp_get_num_threads();
    printf("Hello World Again, I am  %d of %d\n",id,numP);
  }
  return(0);
}
```

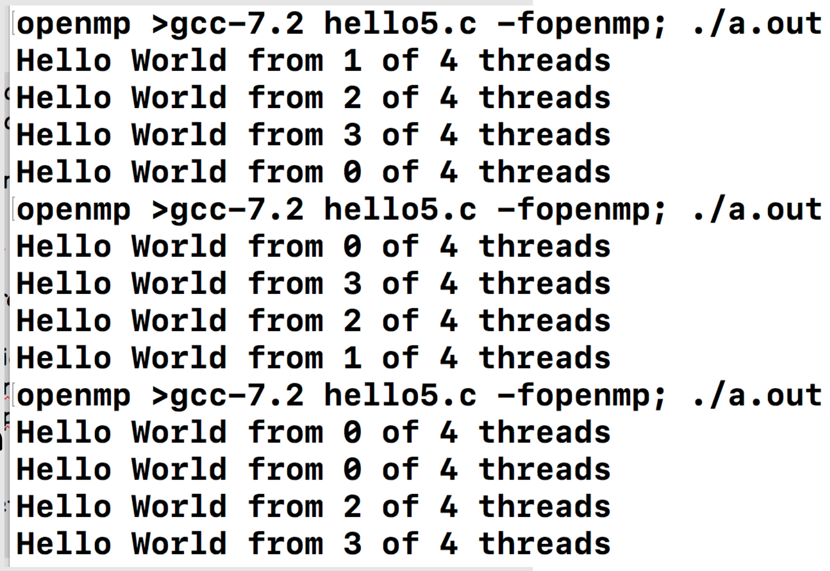


```
openmp >gcc-7.2 hello4.c -fopenmp; ./a.out
Hello World, I am  0 of 2
Hello World, I am  1 of 2
Hello World Again, I am  1 of 4
Hello World Again, I am  2 of 4
Hello World Again, I am  3 of 4
Hello World Again, I am  0 of 4
openmp >
```

Hello World – shared variables & RACE CONDITIONS

Code/Parallel/openmp/hello5.c

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

int main(int argc, const char **argv) {

 int id, numP;

#pragma omp parallel num_threads(4)
  {
    id = omp_get_thread_num();
    numP = omp_get_num_threads();
    printf("Hello World from %d of %d th
  }

  return(0);
}
```



```
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 1 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
Hello World from 0 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 3 of 4 threads
Hello World from 2 of 4 threads
Hello World from 1 of 4 threads
openmp >gcc-7.2 hello5.c -fopenmp; ./a.out
Hello World from 0 of 4 threads
Hello World from 0 of 4 threads
Hello World from 2 of 4 threads
Hello World from 3 of 4 threads
```

What We Want Threads To DO

- Work Independently With Controlled Access at times to Shared Data
  - Parallel Tasks Constructs
    - omp parallel
    - omp for
  - Shared Data

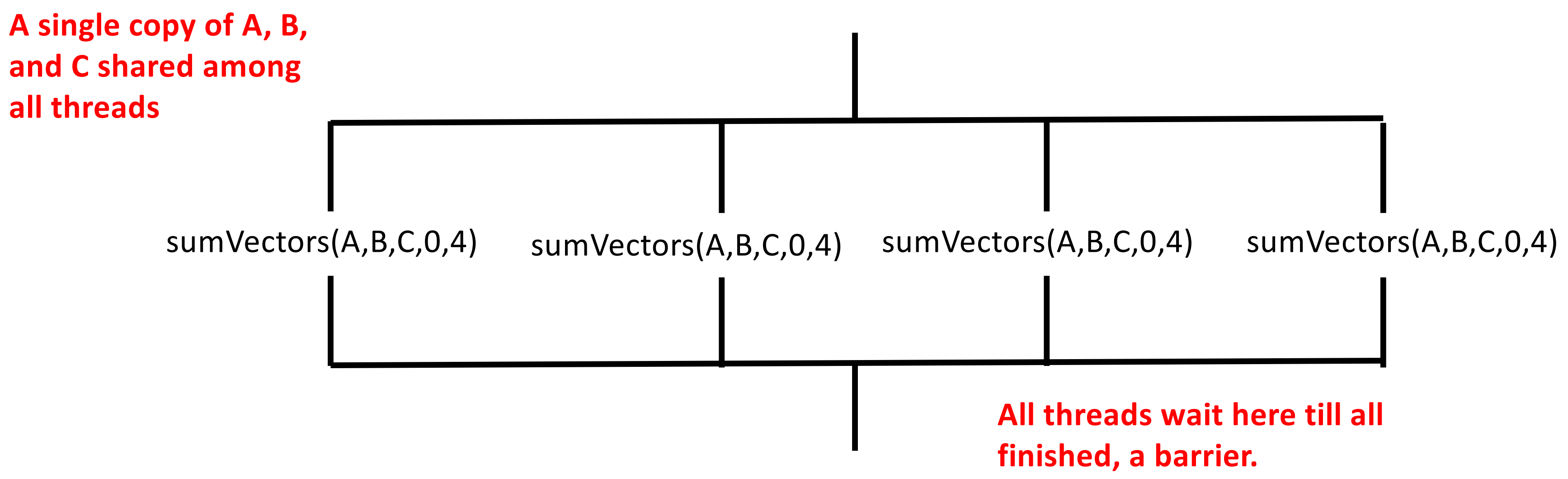
Simple Vector Sum

Code/Parallel/openmp/sum1.c

```
#include <omp.h>
#include <stdio.h>
#define DATA_SIZE 10000

void sumVectors(int N, double *A, double *B, double *C, int tid, int numT);
int main(int argc, const char **argv) {
  double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
  int num;
  for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i+1; }
  double tdata = omp_get_wtime();
#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    num = numT;
    sumVectors(DATA_SIZE, a, b, c, tid, numT);
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f using %d threads\n
  return 0;
}
```

```
void sumVectors(int N, double *A, double *B, double *C, int tid, int numT)
  // determine start & end for each thread
  int start =   tid * N / numT;
  int end = (tid+1) * N / numT;
  if (tid == numT-1)
    end = N;


  // do the vector sum for threads bounds
  for(int i=start; i<end; i++) {
    C[i] = A[i]+B[i];
  }
}
```

Implicit Barrier in Code


A single copy of A, B, and C shared among all threads
sumVectors(A,B,C,0,4)
sumVectors(A,B,C,0,4)
sumVectors(A,B,C,0,4)
sumVectors(A,B,C,0,4)
All threads wait here till all finished, a barrier.

```
openmp >export env OMP_NUM_THREADS=1; ./a.out
first 2.000000 last 200000.000000 in time 0.000902 using 1 threads
openmp >export env OMP_NUM_THREADS=2; ./a.out
first 2.000000 last 200000.000000 in time 0.000678 using 2 threads
openmp >export env OMP_NUM_THREADS=4; ./a.out
first 2.000000 last 200000.000000 in time 0.000652 using 4 threads
openmp >export env OMP_NUM_THREADS=8; ./a.out
first 2.000000 last 200000.000000 in time 0.000693 using 8 threads
```

The for is such an obvious candidate for threads:

Code/Parallel/openmp/sum2.c

```
#include <omp.h>
#include <stdio.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
   double a[DATA_SIZE], b[DATA_SIZE], c[DATA_SIZE];
   for (int i=0; i<DATA_SIZE; i++) { a[i] = i+1; b[i] = i+1; }
   double tdata = omp_get_wtime();
#pragma omp parallel
   {
    #pragma omp for
    for (int i=0; i<DATA_SIZE; i++)
        c[i] = a[i]+b[i];
  }
  tdata = omp_get_wtime() - tdata;
  printf("first %f last %f in time %f \n",c[0], c[DATA_SIZE-1], tdata);
  return 0;
}
```

Code/Parallel/openmp/sum3.c

```
#pragma omp parallel for
     for (int i=0; i<DATA_SIZE; i++)
      c[i] = a[i]+b[i];
```

How About Dot Product?

Code/Parallel/openmp/dot1.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i] = 0;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    if (tid == 0)  nThreads = numT;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum[tid] += a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
     dot += sum[i];
  dot = sqrt(dot);
  printf("dot %f\n", dot);
  return 0;
```

Create a shared array to store data

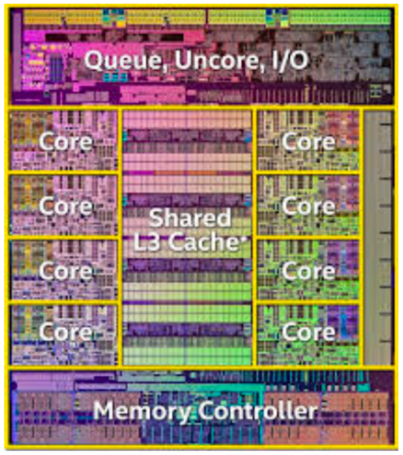
Iterate over big array using thread id and number of threads

Combine sequentially

Poor Performance?

- Want high performance for shared memory: Use Caches!
  - Each processor has its own cache (or multiple caches)
  - Place data from memory into cache
  - Writeback cache: don't send all writes over bus to memory

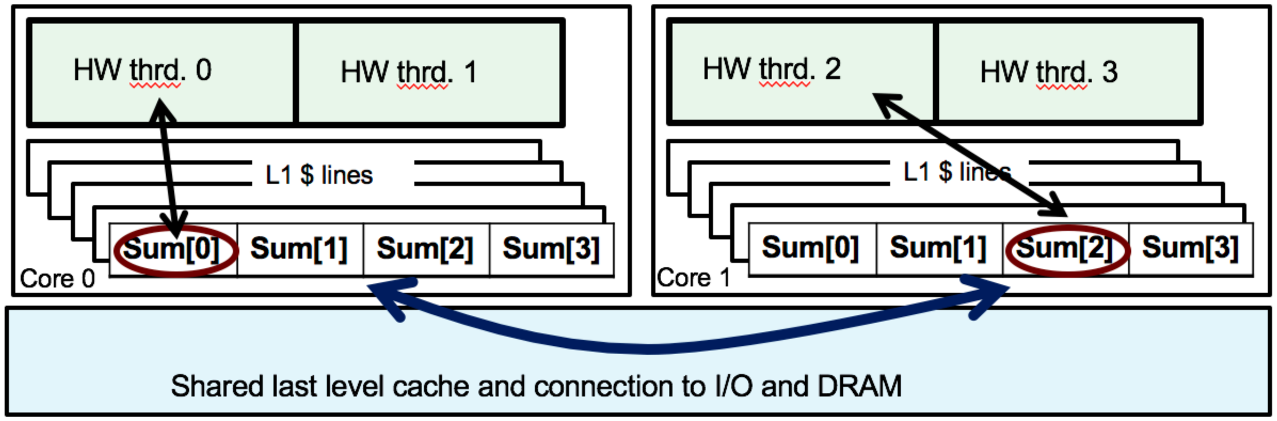
L3 Cache Actually Shared



- Problem is in multi-threaded model with all threads wanting to WRITE same spatially temporal data we have contention at the cache line in the L3 cache

False Sharing

- If independent data elements happen to sit on the same cache line, each update will cause the cache lines to "slosh back and forth" between threads … This is called **false sharing** or sequential **consistency**.



- Sequential Consistency problem is pervasive and performance critical in shared memory

Solution?

source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere

AVOID FALSE SHARING

Code/Parallel/openmp/dot2.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000
#define PAD 64

int main(int argc, const char **argv) {
  int nThreads = 0;
  double dot = 0, a[DATA_SIZE], sum[64][PAD];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;
  for (int i=0; i<64; i++) sum[i][0]= 0;

#pragma omp parallel
  {
     int tid = omp_get_thread_num();
     int numT = omp_get_num_threads();
     if (tid == 0)  nThreads = numT;
     for (int i=tid; i<DATA_SIZE; i+= numT)
       sum[tid][0]+= a[i]*a[i];
  }
  for (int i=0; i<nThreads; i++)
      dot += sum[i][0];
  dot = sqrt(dot);
  printf("dot %f \n", dot);
```

Pad the shared array to store data to avoid false sharing

SYNCHRONIZATION

Code/Parallel/openmp/dot3.c

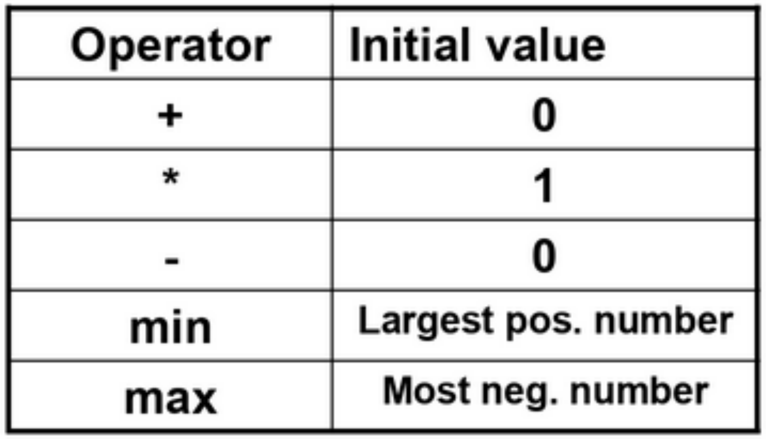
```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];
#pragma omp_critical
    dot += sum;
  }
  dot = sqrt(dot);
    printf("dot %f  \n", dot);
  return 0;
}
```

REDUCTION

Code/Parallel/openmp/dot4.c

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define DATA_SIZE 10000

int main(int argc, const char **argv) {
  double dot = 0;
  double a[DATA_SIZE];
  for (int i=0; i<DATA_SIZE; i++) a[i] = i+1;

#pragma omp parallel reduction(+:dot)
  {
    int tid = omp_get_thread_num();
    int numT = omp_get_num_threads();
    double sum = 0.;
#pragma omp for
    for (int i=tid; i<DATA_SIZE; i+= numT)
      sum += a[i]*a[i];

    dot += sum;
  }
  dot = sqrt(dot);
   printf("dot %f  \n", dot);
  return 0;
```

Additional Reduction Operators



Operator	Initial value
+	0
*	1
-	0
min	Largest pos. number
max	Most neg. number

Are Pitfalls to Parallel Loops might only occur for experienced programmer!

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test

Note: loop index "i" is private by default

```
X = 0.5*dx;
for (int i=0; i<numSteps; i++) {
    pi += 4./(1.+x*x);
    x += dx;
  }
```

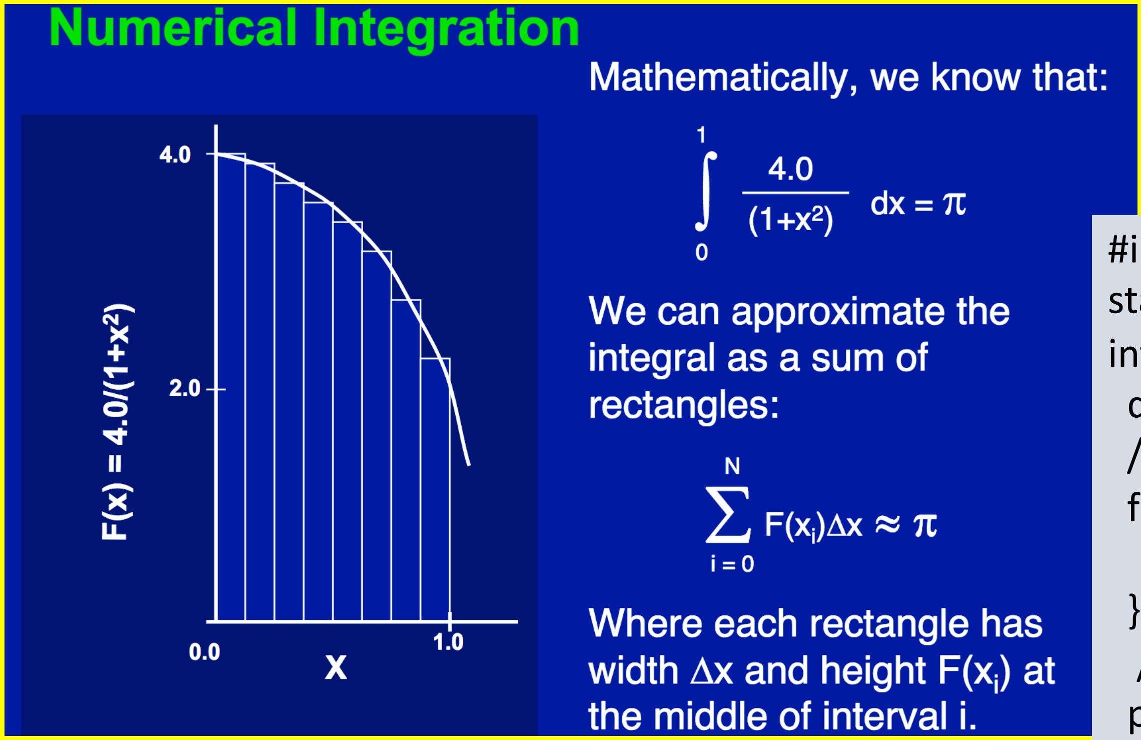
```
#pragma omp parallel for reduction(+:pi)
  for (int i=0; i<numSteps; i++) {
      x = (i+0.5)*dx;
      pi += 4./(1.+x*x);
  }
```

Remove loop carried dependence

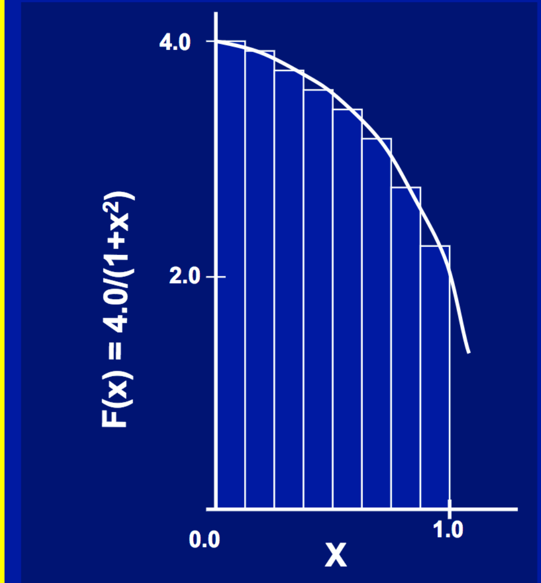
OpenMP®

- OpenMP provides multi-threaded capabilities to C, C++ and Fortran Programs
- In a threaded environment, threads communicate by sharing data
- Unintended sharing of data causes **race conditions**
- **Race Condition**: program output is different every time you run the program, a consequence of the threads being scheduled differently
- OpenMP provides constructs to control what blocks of code are run in parallel and also constructs for providing access to shared data using synchronization
- Synchronization has overhead consequences, you have to minimize them to get good speedup.

Exercise: Parallelize Compute PI using OpenMP



Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

```
#include <stdio>
static int long numSteps = 100000;
int main() {
  double pi = 0; double time=0;
  // your code
  for (int i=0; i<numSteps; i++) {
     // your code
  }
   // your code
  printf("PI = %f, duration: %f ms\n",pi, time);
  return 0;
}
```

Source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere