# Agenda – Day 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Title</th>
<th>Presenter</th>
</tr>
</thead>
<tbody>
<tr>
<td>9:30-10:00</td>
<td>Welcome</td>
<td>Frank McKenna &amp; You</td>
</tr>
<tr>
<td>10:30-11:00</td>
<td>SimCenter Overview</td>
<td></td>
</tr>
<tr>
<td>11:30-12:30</td>
<td>The VM, Linux &amp; Git</td>
<td>Peter Mackenzie-Helnwein</td>
</tr>
<tr>
<td>12:30-1:30</td>
<td>LUNCH</td>
<td>Frank McKenna</td>
</tr>
<tr>
<td>1.30-2:00</td>
<td>An Introduction to Programming</td>
<td></td>
</tr>
<tr>
<td>2:30-3:30</td>
<td>The C Programming Language</td>
<td>Frank McKenna</td>
</tr>
<tr>
<td>3:30-5:00</td>
<td>Exercises</td>
<td>You</td>
</tr>
<tr>
<td></td>
<td>Day 2</td>
<td>Abstraction, More C &amp; C++</td>
</tr>
<tr>
<td></td>
<td>Day 3</td>
<td>Python</td>
</tr>
<tr>
<td></td>
<td>Day 4</td>
<td>Data Gathering &amp; AI</td>
</tr>
<tr>
<td></td>
<td>Day 5</td>
<td>SimCenter Regional Workflows</td>
</tr>
</tbody>
</table>
An Introduction to Programming & The C Programming Language & ....
Frank McKenna
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?

Power supply
Fan
Access Slots

Motherboard

DVD Drive & Other Peripherals

Hard Drive(s)
What is on a Motherboard?

- Heat Sink
- RAM
- CPU
- Peripheral Connections
- PCI Expansion Slots
- Hard Drive Connector(s)
What is in a CPU?
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
<table>
<thead>
<tr>
<th>Jul 2019</th>
<th>Jul 2018</th>
<th>Change</th>
<th>Programming Language</th>
<th>Ratings</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>Java</td>
<td>15.058%</td>
<td>-1.08%</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>C</td>
<td>14.211%</td>
<td>-0.45%</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>↑</td>
<td>Python</td>
<td>9.260%</td>
<td>+2.90%</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>↓</td>
<td>C++</td>
<td>6.705%</td>
<td>-0.91%</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>↑</td>
<td>C#</td>
<td>4.365%</td>
<td>+0.57%</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>↓</td>
<td>Visual Basic .NET</td>
<td>4.208%</td>
<td>-0.04%</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>↑</td>
<td>JavaScript</td>
<td>2.304%</td>
<td>-0.53%</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>↓</td>
<td>PHP</td>
<td>2.157%</td>
<td>-0.67%</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td></td>
<td>SQL</td>
<td>1.977%</td>
<td>-0.36%</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td>Objective-C</td>
<td>1.888%</td>
<td>+0.23%</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>↑</td>
<td>Ruby</td>
<td>1.636%</td>
<td>+0.43%</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>↑</td>
<td>Assembly language</td>
<td>1.390%</td>
<td>+0.24%</td>
</tr>
<tr>
<td>13</td>
<td>11</td>
<td>↓</td>
<td>Swift</td>
<td>1.121%</td>
<td>-0.29%</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
<td>↑</td>
<td>MATLAB</td>
<td>1.078%</td>
<td>-0.05%</td>
</tr>
<tr>
<td>15</td>
<td>81</td>
<td>↑</td>
<td>Groovy</td>
<td>1.070%</td>
<td>+0.96%</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>↑</td>
<td>Go</td>
<td>1.016%</td>
<td>+0.06%</td>
</tr>
<tr>
<td>17</td>
<td>19</td>
<td>↑</td>
<td>Visual Basic</td>
<td>1.009%</td>
<td>+0.12%</td>
</tr>
<tr>
<td>18</td>
<td>16</td>
<td>↓</td>
<td>Delphi/Object Pascal</td>
<td>0.950%</td>
<td>-0.16%</td>
</tr>
<tr>
<td>19</td>
<td>17</td>
<td>↓</td>
<td>Perl</td>
<td>0.918%</td>
<td>-0.18%</td>
</tr>
<tr>
<td>20</td>
<td>14</td>
<td>↓</td>
<td>R</td>
<td>0.837%</td>
<td>-0.31%</td>
</tr>
</tbody>
</table>
Types of Languages – Compiled/Interpreted

- **Compiled:**
  - If compiled:
    - CPU
    - Source Code File
    - Compiler
    - Object File
    - Linker
    - Executable File
    - Evaluates using CPU
  - If interpreted:
    - Interpreter
    - Evaluates using 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

2009

```python
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.

<table>
<thead>
<tr>
<th>Packages/Compilers</th>
<th>Elapsed Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>48.55</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.96</td>
</tr>
<tr>
<td>Matlab</td>
<td>2.398</td>
</tr>
<tr>
<td>gfortran</td>
<td>0.540</td>
</tr>
<tr>
<td>gfortran with -O3</td>
<td>0.280</td>
</tr>
<tr>
<td>ifort</td>
<td>0.276</td>
</tr>
<tr>
<td>ifort with -O3</td>
<td>0.2600</td>
</tr>
<tr>
<td>Java</td>
<td>12.5518</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language</th>
<th>Array/Matrix Storage</th>
<th>Option</th>
<th>n=5000</th>
<th>n=7000</th>
<th>n=9000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>Row-major</td>
<td></td>
<td>19.12</td>
<td>37.49</td>
<td>61.97</td>
</tr>
<tr>
<td>Python + Numba</td>
<td></td>
<td></td>
<td>0.25</td>
<td>0.22</td>
<td>0.30</td>
</tr>
<tr>
<td>Julia</td>
<td>Column-major</td>
<td></td>
<td>0.10</td>
<td>0.22</td>
<td>0.34</td>
</tr>
<tr>
<td>R</td>
<td>Column-major</td>
<td></td>
<td>233.78</td>
<td>451.77</td>
<td>744.93</td>
</tr>
<tr>
<td>IDL</td>
<td>Column-major</td>
<td></td>
<td>7.75</td>
<td>15.21</td>
<td>14.77</td>
</tr>
<tr>
<td>Matlab</td>
<td>Column-major</td>
<td></td>
<td>2.20</td>
<td>4.11</td>
<td>6.80</td>
</tr>
<tr>
<td>Fortran</td>
<td>Column-major</td>
<td>gfortran</td>
<td>0.23</td>
<td>0.33</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gfortran -O3</td>
<td>0.068</td>
<td>0.136</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ifort</td>
<td>0.07</td>
<td>0.18</td>
<td>0.29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ifort -O3</td>
<td>0.068</td>
<td>0.136</td>
<td>0.22</td>
</tr>
<tr>
<td>C</td>
<td>Row-major</td>
<td>gcc</td>
<td>0.17</td>
<td>0.34</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>gcc -Ofast</td>
<td>0.09</td>
<td>0.18</td>
<td>0.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>icc</td>
<td>0.09</td>
<td>0.18</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>icc -Ofast</td>
<td>0.09</td>
<td>0.18</td>
<td>0.42</td>
</tr>
</tbody>
</table>
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):

\[ U_{i,j} = \frac{(U_{i-1,j} + U_{i+1,j} + U_{i,j-1} + U_{i,j+1}) + (U_{i-1,j-1} + U_{i+1,j+1} + U_{i+1,j-1} + U_{i-1,j+1})}{20} \]

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

<table>
<thead>
<tr>
<th>Compilers/Packages</th>
<th>n=50</th>
<th>n=100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>46.15203</td>
<td>751.783</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.610216</td>
<td>6.38891</td>
</tr>
<tr>
<td>Matlab</td>
<td>0.640044</td>
<td>6.531990</td>
</tr>
<tr>
<td>gfortran</td>
<td>0.236</td>
<td>3.248</td>
</tr>
<tr>
<td>gfortran with -O3</td>
<td>0.088</td>
<td>1.256</td>
</tr>
<tr>
<td>ifort</td>
<td>0.052</td>
<td>0.656</td>
</tr>
<tr>
<td>ifort -O3</td>
<td>0.052</td>
<td>0.672</td>
</tr>
<tr>
<td>Java</td>
<td>0.12180</td>
<td>2.2022</td>
</tr>
</tbody>
</table>
Programming Language Hierarchy

Ease of Development

High-Level (language with strong abstraction from details of computer)

- JavaScript
- Ruby, Python
- Java
- C++
- C
- Fortran

Low-Level

- Assembly Language
- Machine Code
- CPU

Program Performance
Single Processor Machine – Idealized Model

Core
- Fetches, decodes & dispatches instructions
- Performs numerical operations
- Data from Memory
  - Control
  - Arithmetic
  - Registers

Memory

Disk
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)

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

```plaintext
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

<table>
<thead>
<tr>
<th>Size</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000 Bytes</td>
<td>0.3 ns</td>
</tr>
<tr>
<td>64 KB</td>
<td>1 ns</td>
</tr>
<tr>
<td>256 KB</td>
<td>3-10 ns</td>
</tr>
<tr>
<td>2-4 MB</td>
<td>20-30 ns</td>
</tr>
<tr>
<td>4-16 GB</td>
<td>50-100 ns</td>
</tr>
<tr>
<td>4-16 TB</td>
<td>5-10e6 ns</td>
</tr>
<tr>
<td>.25-1TB</td>
<td>25-50e3 ns</td>
</tr>
</tbody>
</table>

- Core Processor
  - Control
  - Arithmetic
  - Registers

- L1 Cache
  - 64 KB
  - HW

- L2 Cache
  - 256 KB
  - HW

- L3 Cache
  - 2-4 MB
  - HW

- Memory(RAM)
  - 4-16 GB
  - HW

- Disk
  - Hard Drive
    - 4-16 TB
    - Operating System
  - SSD
    - .25-1TB
    - 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)

- 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 128 bytes)

```c
void main() {
    ...  
    load b into R2
    ...
}
```
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.

```plaintext
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
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 – relatively 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

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 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 character

• Certain keywords are reserved, i.e. cannot be used as variable names
## Reserved Keywords in C

<table>
<thead>
<tr>
<th>C Keywords or Reserved Words</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto</td>
</tr>
<tr>
<td>const</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>short</td>
</tr>
<tr>
<td>struct</td>
</tr>
<tr>
<td>unsigned</td>
</tr>
<tr>
<td>double</td>
</tr>
<tr>
<td>float</td>
</tr>
</tbody>
</table>
Variable Example

#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;
}

#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;
}

Uninitialized Variable

Initialized Variable
Allowable Variable Types in C - I

```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

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
<td>-128 to 127 or 0 to 255</td>
</tr>
<tr>
<td>unsigned char</td>
<td>1 byte</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>1 byte</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>int</td>
<td>2 or 4 bytes</td>
<td>-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>2 or 4 bytes</td>
<td>0 to 65,535 or 0 to 4,294,967,295</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>2 bytes</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>long</td>
<td>4 bytes</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>4 bytes</td>
<td>0 to 4,294,967,295</td>
</tr>
</tbody>
</table>

2. Floating Point Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Range</th>
<th>Decimal Places</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>4 byte</td>
<td>1.2E-38 to 3.4E+38</td>
<td>6</td>
</tr>
<tr>
<td>double</td>
<td>8 byte</td>
<td>2.3E-308 to 1.7E+308</td>
<td>15</td>
</tr>
<tr>
<td>long double</td>
<td>10 byte</td>
<td>3.4E-4932 to 1.1E+4932</td>
<td>19</td>
</tr>
</tbody>
</table>

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}

```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.

- to declare: `type arrayName[l1][l2][l3]...;`
  `type arrayName[l1][l2][l3] = {l1*l2* ... comma separated values}`

```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);
}
```

```
array2.c
```

---

```
<table>
<thead>
<tr>
<th></th>
<th>a[0][0]</th>
<th>a[0][3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>101.5</td>
<td>1400</td>
</tr>
<tr>
<td>2</td>
<td>21.1</td>
<td>0.0242</td>
</tr>
</tbody>
</table>
```
Memory Layout of Arrays in C and Fortran

C

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

row-major

```
1 2 3
4 5 6
7 8 9
```

Fortran

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

column-major

```
1 4 7
2 5 8
3 6 9
```
Operations

• We want to do stuff with the data, to operate on it
• Basic Arithmetic Operations
  
  +, -, *, /, %

```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 –

```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

```bash
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.

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

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

\[
+= \quad a += 1;
-\quad -=
*\quad *=
/\quad /=
++\quad a ++;
--\quad --
\]
Conditional Code – if statement

- 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

```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

```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

```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

```c
#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

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

```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++;
        // 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 loop

```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 loop – multiple init & increment

```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.

```c
#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 setting 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.

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

- **Art of Programming I**: “To take a problem, and recursively 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 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** <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.
#include <stdio.h>

int *: data is a pointer to an int

// 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);
}
Function Prototype

```c
#include <stdio.h>

int sumArray(int *arrayData, int size); // function to evaluate vector 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);
}
```

Good practice to give the args names

```c
int sumArray(int *data, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += *data++;
    }
    return sum;
}
```
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 
”, sum);
}

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;
}

myVector.h
int sumArray(int *arrayData, int size);
int productArray(int *arrayData, int size);
int normArray(int *arrayData, int size);
int dotProduct(int *array1, int *array2, int size);
Exercise: Write a function to sum two values

```c
#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
}
```

1. `gedit sum.c`
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.

```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);
    printf("%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]

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

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

```bash
[c >gcc math1.c -lm; ./a.out
sqrt(34.000000) is 5.830952
[c >]
```
Scope of Variables

```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
```
Recursion

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

```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);
}
```

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

#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);
}

memory2.c
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] = i;
}
```

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;
}
```

```
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] = i;
}
```

```
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’

```c
#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

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 \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).

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

```c
#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

\{\text{implements } C = C + A \cdot B\}

\begin{align*}
for \ i = 1 \ to \ n \\
\{\text{read row } i \ of \ A \ into \ fast \ memory\} \\
for \ j = 1 \ to \ n \\
\{\text{read } C(i,j) \ into \ fast \ memory\} \\
\{\text{read column } j \ of \ B \ into \ fast \ memory\} \\
for \ k = 1 \ to \ n \\
C(i,j) &= C(i,j) + A(i,k) \cdot B(k,j) \\
\{\text{write } C(i,j) \ back \ to \ slow \ memory\}
\end{align*}

\begin{align*}
C(i,j) & \quad = \quad C(i,j) \\
& \quad + \quad A(i,:) \quad \ast \quad B(:,j)
\end{align*}

Source: UC Berkeley, Jim Demmel, 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}
      $C(i,j) = C(i,j) + A(i,k) \times B(k,j)$ {do a matrix multiply on blocks}
    {write block $C(i,j)$ back to slow memory}

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

Source: UC Berkeley, Jim Demmell, CS267
Recursive Matrix Multiplication (RMM) (1/2)

\[
C = \begin{pmatrix}
    c_{11} & c_{12} \\
    c_{21} & c_{22}
\end{pmatrix} = A \cdot B = \begin{pmatrix}
    A_{11} & A_{12} \\
    A_{21} & A_{22}
\end{pmatrix} \cdot \begin{pmatrix}
    B_{11} & B_{12} \\
    B_{21} & B_{22}
\end{pmatrix}
\]

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

- True when each block is a 1x1 or \( n/2 \times n/2 \)
- 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 · 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 · W(n/2) + 4 · 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 Demmel, 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

1 Intel i7 node, 6 cores + 16GB RAM

Apple A11, 6 cores + 64GB RAM
# How Big & Fast!

<table>
<thead>
<tr>
<th>Rank</th>
<th>Site</th>
<th>System</th>
<th>Cores</th>
<th>Rmax (TFlop/s)</th>
<th>Rpeak (TFlop/s)</th>
<th>Power (kW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DOE/SC/Oak Ridge National Laboratory</td>
<td>Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM</td>
<td>2,282,544</td>
<td>122,300.0</td>
<td>187,659.3</td>
<td>8,806</td>
</tr>
<tr>
<td>2</td>
<td>National Supercomputing Center in Wuxi</td>
<td>Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCP</td>
<td>10,649,600</td>
<td>93,014.6</td>
<td>125,435.9</td>
<td>15,371</td>
</tr>
<tr>
<td>3</td>
<td>DOE/NNSA/LLNL</td>
<td>Sierra - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM</td>
<td>1,572,480</td>
<td>71,610.0</td>
<td>119,193.6</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>National Super Computer Center in Guangzhou</td>
<td>Tianhe-2A - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.20GHz, TH Express-2, Matrix-2030 NUDT</td>
<td>4,981,760</td>
<td>61,444.5</td>
<td>100,678.7</td>
<td>18,482</td>
</tr>
<tr>
<td>5</td>
<td>National Institute of Advanced Industrial Science and Technology (AIST)</td>
<td>Al Bridging Cloud Infrastructure (ABCII) - PRIMERGY CX2550 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR Fujitsu</td>
<td>391,680</td>
<td>19,880.3</td>
<td>32,576.6</td>
<td>1,649</td>
</tr>
<tr>
<td>15</td>
<td>Texas Advanced Computing Center/Univ. of Texas</td>
<td>Stampede2 - PowerEdge C6320P/C6420, Intel Xeon Phi 7250 68C 1.4GHz/Platinum 8160, Intel Omni-Path Dell EMC</td>
<td>367,024</td>
<td>10,680.7</td>
<td>18,309.2</td>
<td></td>
</tr>
</tbody>
</table>

Rmax of Linpack
Solve Ax = b
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

\[
\text{Speedup}(P) = \frac{\text{Time}(1)}{\text{Time}(P)} \\
\leq \frac{1}{s + (1-s)/P} \\
\leq \frac{1}{s}
\]

**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 Demmel
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

```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_finalizer() must be first and last functions called

MPI_COMM_WORLD is a default group containing 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
#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;
}
**MPI Basic (Blocking) Send**

- The message buffer is described by \((\text{start}, \text{count}, \text{datatype})\).
- The target process is specified by \(\text{dest}\), which is the rank of the target process in the communicator specified by \(\text{comm}\). The message has an identifier \(\text{tag}\).
- 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_SEND**(start, count, datatype, dest, tag, comm)
- The message buffer is described by \((\text{start}, \text{count}, \text{datatype})\).
- The target process is specified by \(\text{dest}\), which is the rank of the target process in the communicator specified by \(\text{comm}\). The message has an identifier \(\text{tag}\).
- 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.
<table>
<thead>
<tr>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>MPI_WCHAR</td>
<td>wchar_t - wide character</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>signed short int</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>signed int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>signed long int</td>
</tr>
<tr>
<td>MPI_LONG_LONG_INT</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>MPI_FLOAT</td>
<td>float</td>
</tr>
<tr>
<td>MPI_DOUBLE</td>
<td>double</td>
</tr>
<tr>
<td>MPI_LONG_DOUBLE</td>
<td>long double</td>
</tr>
<tr>
<td>MPI_C_COMPLEX</td>
<td>float_{Complex}</td>
</tr>
<tr>
<td>MPI_C_FLOAT_COMPLEX</td>
<td>float_{Complex}</td>
</tr>
<tr>
<td>MPI_C_DOUBLE_COMPLEX</td>
<td>double_{Complex}</td>
</tr>
<tr>
<td>MPI_C_LONG_DOUBLE_COMPLEX</td>
<td>long double_{Complex}</td>
</tr>
<tr>
<td>MPI_C_BOOL</td>
<td>bool</td>
</tr>
<tr>
<td>MPI_INT8_T</td>
<td>int8_t</td>
</tr>
<tr>
<td>MPI_INT16_T</td>
<td>int16_t</td>
</tr>
<tr>
<td>MPI_INT32_T</td>
<td>int32_t</td>
</tr>
<tr>
<td>MPI_INT64_T</td>
<td>int64_t</td>
</tr>
<tr>
<td>MPI_UINT8_T</td>
<td>uint8_t</td>
</tr>
<tr>
<td>MPI_UINT16_T</td>
<td>uint16_t</td>
</tr>
<tr>
<td>MPI_UINT32_T</td>
<td>uint32_t</td>
</tr>
<tr>
<td>MPI_UINT64_T</td>
<td>uint64_t</td>
</tr>
<tr>
<td>MPI_BYTE</td>
<td># binary digits</td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>data packed or unpacked with MPI_Pack() or MPI_Unpack()</td>
</tr>
</tbody>
</table>
MPI Basic (Blocking) Receive

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)

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

MPI_RECV(start, count, datatype, source, tag, comm, status)
Retrieving Further Information

• **Status** is a data structure allocated in the user’s program.

• In C:

```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 );
```
Deadlock

.. Program just hangs .. The above can deadlock!

```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;
}
```

Not Quite So Simple as ensuring PAIRS of send/recv

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:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Send (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Recv (0)</td>
</tr>
</tbody>
</table>

Could revise order this requires some code rewrite:

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send (1)</td>
<td>Recv (0)</td>
</tr>
<tr>
<td>Recv (1)</td>
<td>Send (0)</td>
</tr>
</tbody>
</table>

Alternatives use non-blocking sends.
Some Collective Functions

- **MPI_Bcast**
  - Source node 0
  - Destination nodes: 1, 2, 3

- **MPI_Scatter**
  - Source node 0
  - Destination nodes: 1, 2, 3

- **MPI_Gather**
  - Source nodes: 1, 2, 3
  - Destination node: 0

- **MPI_Allgather**
  - Source nodes: 1, 2
  - Destination node: 0
#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

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

#define LUMP 5

int main(int argc, char **argv) {
    int numP, proclID;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numP);
    MPI_Comm_rank(MPI_COMM_WORLD, &proclID);

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

    if (proclID == 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", proclID, localData[0], localData[LUMP-1]);

    if (proclID == 0) free(globalData);
    MPI_Finalize();
}
```
```c
#include "mpi.h"
#include <stdio.h>
#define LUMP 5
int main(int argc, const char **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

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 \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).

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

```c
#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;
}
```
Stampede2
#!/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 Demmel
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 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

```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;
}
```

Code/Parallel/openmp/hello1.c

Hello World – changing num threads

```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 \%d\n", id, numP);
    }
    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;
}
```
Different # threads in different blocks

```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);
}
```

Code/Parallel/openmp/hello4.c
Hello World – shared variables & RACE CONDITIONS

```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 threads\n", id, numP);
    }

    return(0);
}
```

Code/Parallel/openmp/hello5.c

```
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
```
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
#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", c[0], c[DATA_SIZE-1], tdata, numT);
    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];
    }
}

Code/Parallel/openmp/sum1.c
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:

```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;
}
```
How About Dot Product?
```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

- 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**.

![](image.png)

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

**Solution?**

*source: UC Berkeley, Tim Mattson (Intel Corp), CS267 & elsewhere*
#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);
    return 0;
}

Pad the shared array to store data to avoid false sharing.
#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;
}
#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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>min</td>
<td>Largest pos. number</td>
</tr>
<tr>
<td>max</td>
<td>Most neg. number</td>
</tr>
</tbody>
</table>
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

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

Note: loop index “i” is private by default

#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 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

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 \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).

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