Unit 2 - Parallel hardware and parallel software design

Overview:

Before you can design and write efficient parallel programs, you must also understand the parallel hardware on which your parallel programs will execute. As you learn about parallel hardware, you will also be learning general parallel program design techniques and patterns.

Learning Objectives

Course Learning Objective #1: Explain the operation of uniprocessor computer components including the processor (pipelined and superscalar) and memory hierarchy (cache and virtual memory).
Supporting Unit 2 Learn Objectives:

- (HW 4) Explain the snoopy bus cache coherence scheme.

Course Learning Objective #2: Demonstrate an understanding of uniprocessor computer architecture by designing and writing C programs that make efficient use of the processor and memory hierarchy.
Supporting Unit 2 Learn Objectives:

- (Lab 4) Write correct programs that dynamically allocate and use 2D arrays.
- (Lab 4) Write correct programs that embeds a 2D array into a 1D array.
- (Lab 4) Write correct programs that read and write text files.
- (Lab 5) Utilize the random number generator in C to generate 2D arrays.
- (Lab 5) Write code to perform the matrix multiplication calculation efficiently.
- (Lab 5) Write correct programs that read and write binary files.

Course Learning Objective #3: Explain the operation of parallel hardware including cache-coherence and mutexes on shared-memory machines, and interconnect performance (bisection bandwidth, bandwidth and latency) characteristics on distributed-memory machines.
Supporting Unit 2 Learn Objectives:

- (HW 4) Contrast the architectural differences between shared-memory vs. distributed-memory machines.
- (HW 4) Compare the general characteristics of interconnection networks.

Course Learning Objective #4: Demonstrate an understanding of parallel hardware and general parallel program design techniques and patterns by producing efficient parallel program designs to minimize parallel program overhead.
Supporting Unit 2 Learn Objectives:

- (HW 5) List sources of parallelization overhead
- (HW 5) Distinguish between task vs. data parallelism
- (HW 5) Apply Foster's methodology for parallel program design to produce a design that minimizes parallel overhead.
- (HW 5) Apply Amdalh's law to calculate the maximum speedup given a fraction of serial processing.
- (HW 5) Calculate parallel performance metrics: speedup and efficiency

Unit 2 Calendar

Unit	Week	Course Objectives	Reading, Video, Discussion Board	Graded Assignments	Due Dates
2	4	3	Ch. 2.3 & "Monday" Video	Video Quiz	9/14 by 3 PM
		2	Lab 4 Video	Lab 4	9/19 by 3 PM
		1, 3	Ch. 2.3.4 & "Friday" Video	Video Quiz	9/18 by 3 PM
		3	Week 4 Discussion Questions	Discussion Questions	9/20 by 11 PM
		1, 4		HW 4	9/23 by 3 PM
	5	4	Ch. 2.4 & "Monday" Video	Video Quiz	9/21 by 3 PM
		1, 2	Lab 5 Video	Lab 5	9/26 by 3 PM
		4	Ch. 2.7 & "Friday" Video	Video Quiz	9/25 by 3 PM
		4	Week 5 Discussion Questions	Discussion Questions	9/27 by 11 PM
		4		HW 5	9/30 by 3 PM

Week 4 Discussion Questions Due: Sunday, Sept 20 at 11 PM

Chapter 2.3 - 2.4:

1. We used Flynn's taxonomy to identify three types of parallel systems: SISD, SIMD, and MIMD. How might a MISD (multiple Instruction, single data) work? (Give an example if possible)

2. The TOP500 list (http://www.top500.org/) provides a ranked list of the most powerful general purpose supercomputers systems. It also provides "historical data" at http://www.top500.org/statistics/overtime/ If you change the category to "Architecture" and hit the Submit button, you can see a graph for the last 10 years.

a) What architectures dominate the past 10 years and why?

b) What is meant by the architecture "MPP"?

At the top of the graph is a gray "slider" in the middle of the graph. Slide it to the left all the way to expand the time frame another 10 years or so. If you position the mouse pointer along the time-line at the bottom of the graph you can see data values for each architecture at specific dates.

c) What do you observer about the Single Processor and SIMD architecture types?

d) What is meant by the architecture "Constellations"?

e) Between Nov 01, 1998 and Nov 01, 2000 the SMP % drastically decreased while the Constellations % drastically increased. What do you supposed happened to explain this?

3. If you change the category to "Interconnect Family" and hit the Submit button, you can see a graph for the last 10 years.

a) What Interconnection families have dominate the past 10 years and why?

4. A common Snooping cache-coherence scheme used on multi-core PCs is the MESI protocol. Some helpful links are at:

- Wikipedia - http://en.wikipedia.org/wiki/MESI_protocol
- Animation of MESI - https://www.cs.tcd.ie/Jeremy.Jones/vivio/caches/MESIHelp.htm

a) What is the main difference between the M and E states?

b) How does the M state reduce accesses to the shared main memory?

c) How does the E state reduce accesses to the shared main memory?

Learning Objectives:
- Write correct programs that dynamically allocate and use 2D arrays.
- Write correct programs that embeds a 2D array into a 1D array.
- Write correct programs that read and write text files.

To start the lab:
- watch the Lab 4 Video on the eLearning system
- download `lab4.zip` from the eLearning system and unzip/extract it locally on your computer
- copy the `lab4` directory to `student.cs.uni.edu` using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to `student.cs.uni.edu` using Putty/ssh

Part A: Using an editor on `student.cs.uni.edu` open the file `cmdLineMultTable.c` which contains a simple C program that allows the user to enter two integers on the command-line, and prints the corresponding multiplication table.

Answer the following questions about the `cmdLineMultTable.c` program:

a) What is the maximum size of value1 and value2 that the program is designed for?

b) Compile (`gcc -o table cmdLineMultTable.c`) and run the program with values 15 and 10 (`./table 15 10`). Did it produce the correct results?

c) Re-run the program with values 150 and 100 (`./table 150 100`). Explain the results.

Notice that the multiplication table is stored in a statically declared 2D array defined in the `main` as:

```
int multiplicationTable[SIZE][SIZE];
```

The `main` passes `multiplicationTable` to the `calculateProducts` function to be "filled" with the call:

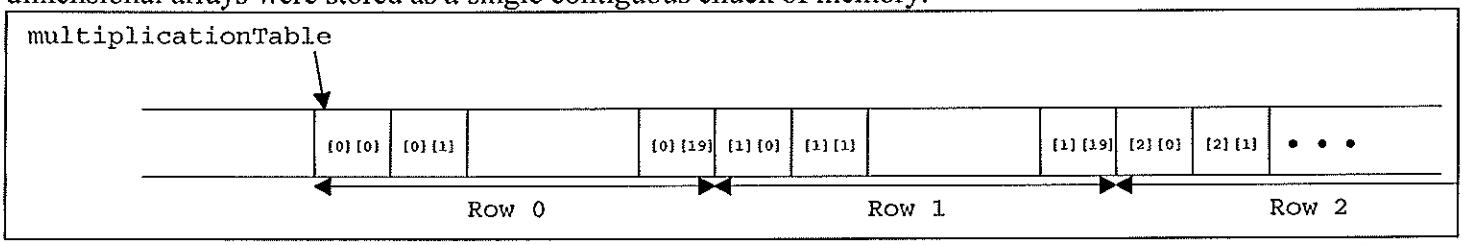
```
calculateProducts(value1, value2, multiplicationTable);
```

d) Why is a `&` not needed before the `multiplicationTable` parameter eventhough it "returns" a changed value?

The `calculateProducts` function definition is:

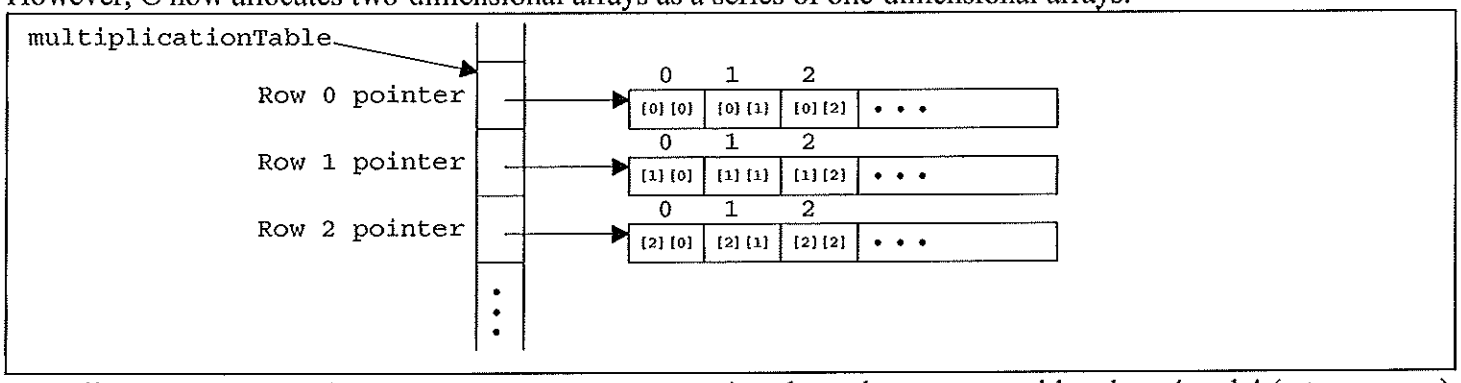
```
void calculateProducts(int rows, int columns, int multiplicationTable[][SIZE])
```

Note: C requires sizes for all, but the first dimension of arrays to be specified. This is a holdover of when multi-dimensional arrays were stored as a single contiguous chuck of memory:



Thus, address of `multiplicationTable[r][c]` = starting address of array + (r * SIZE + c) * (size of an element).

However, C now allocates two-dimensional arrays as a series of one-dimensional arrays:



Compile `printAddrs.c` (`gcc -o addrs printAddrs.c`) and run the program with values 4 and 4 (`./addrs 4 4`) which is the same as the `cmdLineMultTable.c` program, except it also prints the addresses of each array element.

e) How many bytes is used to store each integer product?

f) How many bytes are used to store each row (recall that SIZE is 20)?

g) How big of a "gap" it there between the end of one row and the start of another?

Part B: Using an editor open `printAddrsDyn.c` which contains a similar C program to print a multiplication table. Notice that the multiplication table storage is NOT allocated when the `main` starts, instead only the pointer:

```
int ** multiplicationTable;
```

The `main` assigns `multiplicationTable` a pointer value to a dynamically allocated 2D array by:

```
multiplicationTable = allocate2DArray(value1, value2);
```

Answer the following questions about the `allocate2DArray` function:

a) What is the maximum size of value1 and value2 that the program is designed for?

b) How much "wasted" storage space is allocated for storing integer products using dynamically allocated 2D array?

c) Explain each part of the assignment state in the `allocate2DArray` function:

```
local2DArray = (int **) malloc(sizeof(int *)*rows);
```

d) What is the purpose of the loop in the `allocate2DArray` function?

```
for (r=0; r < rows; r++) {
  local2DArray[r] = (int *) malloc(sizeof(int)*columns);
} // end for
```

Compile `printAddrsDyn.c` (`gcc -o addrsDyn printAddrsDyn.c`) and run the program with values 4 and 4 (`./addrsDyn 4 4`) which is the same as the `printAddrs.c` program, except it prints the pointer address contained in `multiplicationTable` , each element (i.e., a pointer addresses to the start of each 1-D array for a row) in the 1-D array pointed at by `multiplicationTable`, and the *address* of each element.

e) How many byte are between each element in the 1-D array pointed at by `multiplicationTable`?

f) To practice passing 2-D arrays as parameters, modify the printAddrsDyn.c program so that `allocate2DArray` is passed the `multiplicationTable` as a parameter which gets modified inside of `allocate2DArray`, instead of the `allocate2DArray` returning a `int ** value`. Re-compile and re-run the program to make sure that it still works.

Part C: Sometimes in this course we will want to explicitly embed a 2D array inside a 1D array as in the first picture (p. 1). Typically, we do this if we want to send a 2D array as a single message, or copy a 2D array easily. Open the multTable2Din1D.c program which prints a multiplication table using a single 1D array containing an embedded 2D array.

a) How did the `allocate2DArray` function change?

b) In the `calculateProducts` function explain the pointer arithmetic on the left-hand side of the assignment:

```
*(multiplicationTable+r*columns+c) = (r+1) * (c+1);
```

c) An alternate way of accessing the 1D array is to "walk a pointer" down it. Comment out the above assignment statement and uncomment the alternate using the `nextElementPtr`. Re-compile and re-run the code to see that it works. This alternative uses the pointer and then "post-increments" it (the ++ operator after the pointer name). How does the ++ operator know how much to increment the pointer by?

d) Modify the `printRow` function so that it also walks a pointer down the array. Re-compile and re-run the code to see that it works.

Part D: Open the file `fileMultTable.c` which allows the user to enter two integers **and a text-file name** on the command-line, and prints the corresponding multiplication table to the specified text file.
In C as in most languages, to use a file you need to:
1) Open the file for reading or writing - connects the program to the file
2) Read or Write to the file
3) Close the file - if the file was opened for writing, the operating system often buffers the writes in main memory because the disk is so slow. Closing the file flushes the writes to disk.

a) An open file is C use a "file pointer" variable of type: FILE *. What is the variable name of the file pointer used in this program?

b) What statement(s) cause the file to open?

c) The `printTableHeading` and `printRow` funtions write to the text-file using `fprintf` statements. What's different about the parameters to the `fprintf` statement vs. the `printf` statement?

d) What statement(s) cause the file to be closed?

e) To read text-input from the keyboard `scanf` is used, to read input from a text-file what statement (and parameters) do you think would be used?

Submit lab4.zip containing question answers and completed programs on the eLearning system

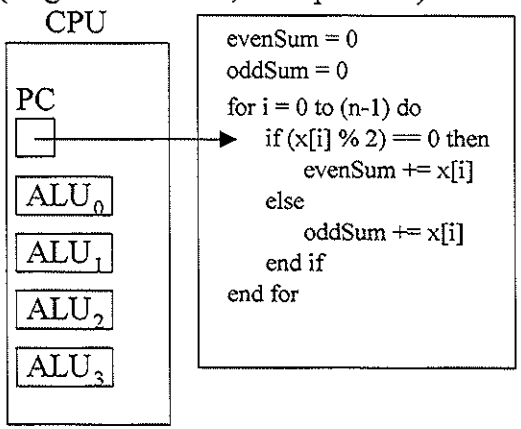
Flynn's Taxonomy

SISD
(single-instruction, single-data)
sequential uniprocessor computer

MISD
???

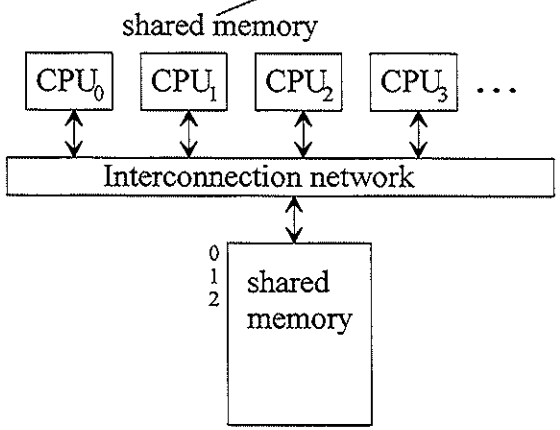
MIMD (multiple-instruction, multiple-data)
CPU's execute different programs (or different points in the same program -- SPMD)

SIMD
(single-instruction, multiple-data)



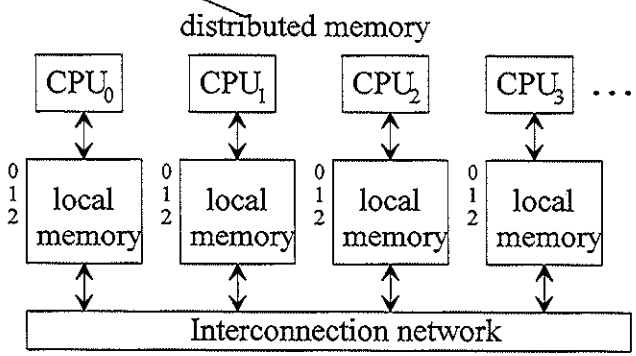
Single program, but multiple ALUs executing on different data value or not (e.g., ALU_0 on x[0], ALU_1 on x[1], etc...) executing if the condition is not satisfied.

(GPU programming is SIMD-like)



Programming model: (e.g. pthreads)
* start single process that "forks" threads
* each thread carries out a task
* threads communicate and synchronize through shared variable in memory

Shared-Memory Interconnection Networks:
* bus - parallel collection of wires shared by all connected devices
* switched interconnect - uses switches to control routing of data (e.g., crossbar)



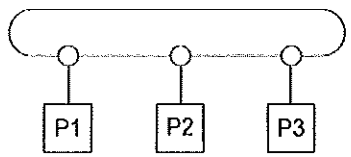
Programming model: (e.g., MPI)
* start multiple processes on each CPU
* each process carries out a task
* processes communicate and synchronize by send messages through the interconnection network

Distributed-Memory Interconnections:
* direct interconnect - switches connected directly to processor/memory pairs (e.g., ring, toroidal mesh, hypercube, etc.
* indirect interconnect - switches on connected directly to processors (e.g., omega)

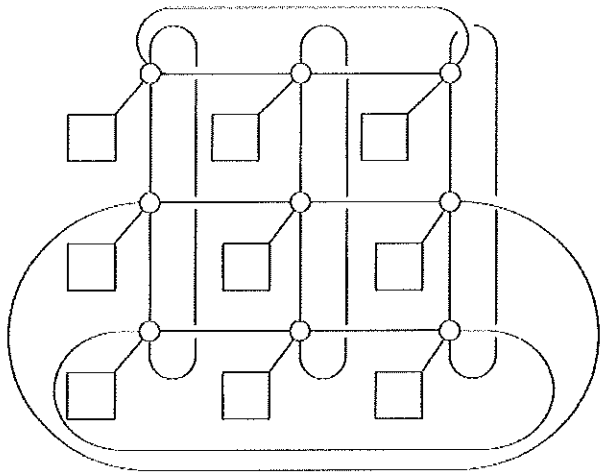
Interconnection Network - Effects performance on both distributed and shared memory systems

Distributed Memory Interconnects:

Direct Interconntect:



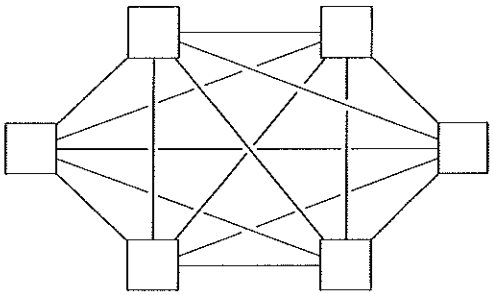
(a)



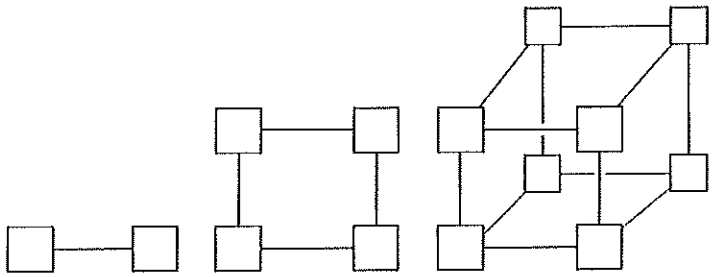
(b)

Ring

2D Toroidal Mesh



Fully Connected



(a) (b) (c)

1D 2D 3D

Hypercubes

Terminology:

Bandwidth - rate at which a link can transmit data (e.g, megabits / second)

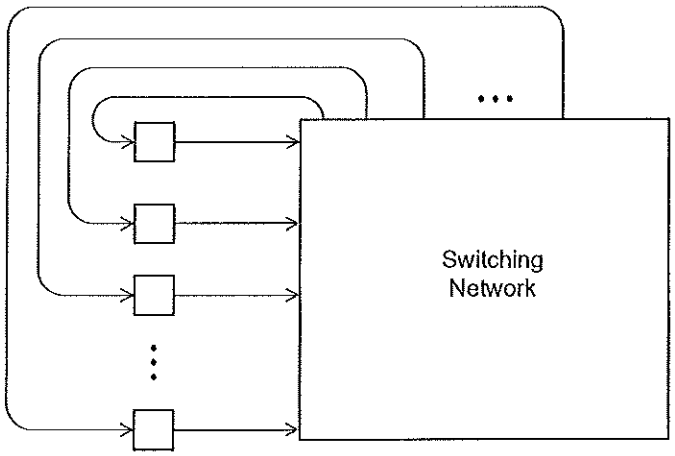
Bisection bandwidth - a measure of the network quality which sums the bandwidth connecting halves of the processors.

The transmission time of single data transmission of *n* bytes between two processors (the *source* and *destination*) is effected by:

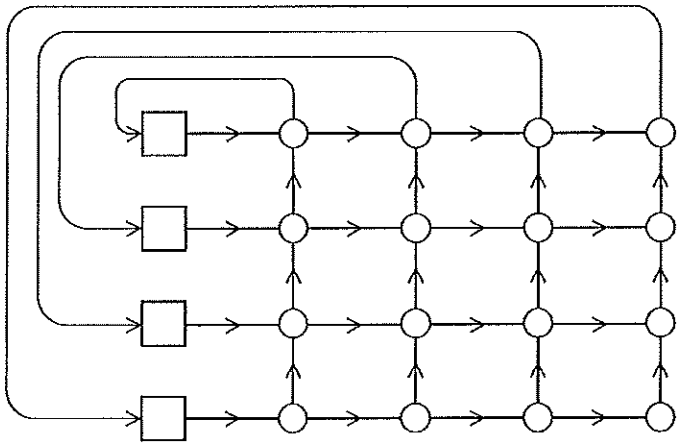
- *latency* (*l*) - the time that elapses between the source's beginning to transmit the data and the destination 's receiving the first byte of data.
- *bandwidth* (*b*) - the rate at at which the destination receives data after it has started to receive the first byte

message transmission time $= l + n / b$

Generic Indirect Interconnection Network: Processors (boxes on left) have an in-coming and an out-going link from and to the switching network

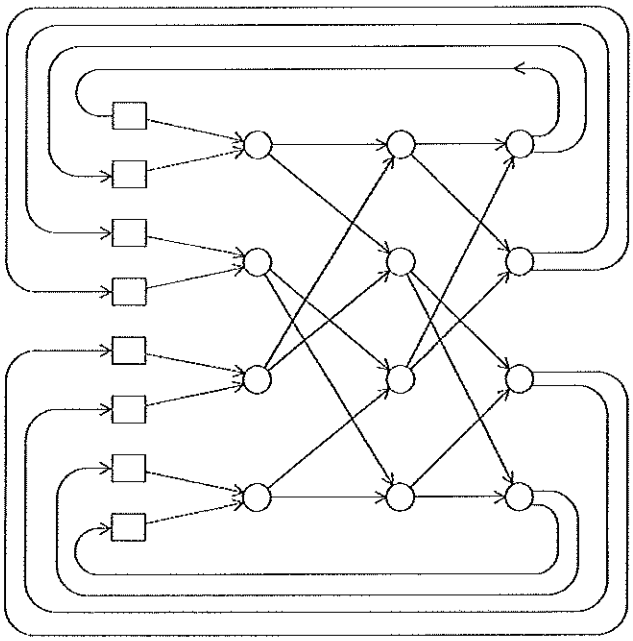


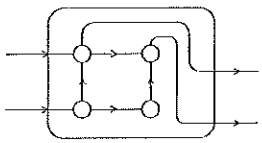
Crossbar interconnection:



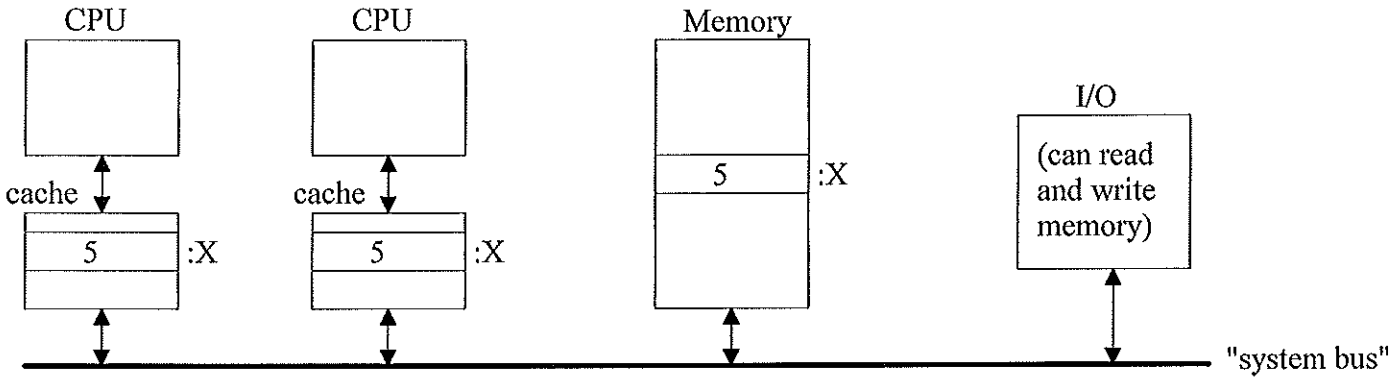
Omega network:

An Omega switch





Cache Coherence Solution - bus watching with write through / Snoopy caches - caches eavesdrop on the bus for other caches write requests. If the cache contains a block written by another cache, it take some action such as invalidating it's cache copy.



The MESI protocol is a common write-back cache-coherency protocol. Each cache line is marked as: Modified, Exclusive, Shared or Invalid.

	Modified	Exclusive	Shared	Invalid
This cache line valid?	Yes	Yes	Yes	No
The memory copy is ...	out of date	valid	valid	-
Copies exist in other caches?	No	No	Maybe	Maybe
A write to this line ...	does not go to the bus	does not go to the bus, but change to M	goes to the bus and update cache	goes directly to bus

3. How can distributed shared memory machines do cache coherency?

Week 5 Discussion Questions Due: Sunday, Sept 27 at 11 PM

Chapter 2.5 - 2.10:

1. What are the main motivation(s) of writing a parallel program?

We can categorize sources of overhead in parallel programs that limit speedup as follows:

- interprocess communication/interaction - processors need to communicate data (i.e., intermediate results)
- idle processors - processors can be idle for a variety of reasons:
  - load imbalance - a processor is assigned less work than others so it sits idle waiting of others to finish
  - synchronization - processor need to to coordinate their operations (e.g., barrier synchronization) so processors done sooner must wait for others to complete
  - non-parallelizable computation/task - some serial component that cannot be done in parallel. Amdahl's law applies here.
- parallelization overhead - additional costs in the parallel solution that are not in the sequential computation

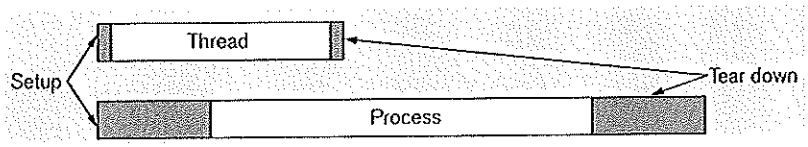
2. For each of the following scenarios identify the type of overhead(s) that occur.

a) theads doing extra computation to determine which part of the parallel computation they need to perform

b) parallel computation is unevenly distributed to processors so some finish before others

c) a spin lock in which a waiting thread repeated checks for the availability of a lock on a shared variable

d) thread/process setup and teardown time when a thread/process is created and later destroyed



e) Sequential computation performed redundantly across all processors

Learning Objectives:
- Utilize the random number generator in C to generate 2D arrays.
- Write code to perform the matrix multiplication calculation efficiently.
- Write correct programs that read and write binary files.

To start the lab:
- watch the Lab 5 Video on the eLearning system
- download `lab5.zip` from the eLearning system and unzip/extract it locally on your computer
- copy the `lab5` directory to `student.cs.uni.edu` using a secure ftp client (winSCP, FileZilla, scp, etc.)
- log-on to `student.cs.uni.edu` using Putty/ssh

Part A: Using an editor on `student.cs.uni.edu` open the file `writeRandom2DArray.c` which contains a simple C program that allows the user to enter two integers (# rows and # columns), a binary file name, and two reals (min. random # and max. random # range) on the command-line. The program writes to the specified file name binary data consisting of:
- integer number of rows
- integer number of columns
- the randomly generated 2D array of doubles in row-major order

Answer the following questions about the `writeRandom2DArray.c` program:

a) How does the creation (opening) of the binary file differ than the creation of the text file in lab4?

Consider the The `generateRandom2DArray` function defined as:

```
void generateRandom2DArray(int rows, int columns,
                           double min, double max, double ** random2DArray) {
  int r, c;
  double range, div;

  // seed the random number generator
  srand( time(NULL) );
  for (r = 0; r < rows; r++) {
    for (c = 0; c < columns; c++) {
      range = max - min;
      div = RAND_MAX / range;
      random2DArray[r][c] = min + (rand() / div);
    } // end for (c...
  } // end for (r...
} // end generateRandom2DArray
```

b) By seeding the random number generator using the current time in seconds (i.e., `srand( time(NULL) );`) then we should mostly get a different randomly generated 2D array everytime we run the program. What problem might occur if we called the `generateRandom2DArray` function twice within the same program to generate two difference 2D arrays, say A and B?

c) How could we fix the above problem?

d) Explain each of the three assignment statements what generate a random array element.

Part B: Since the output file generated is binary (i.e., not a text file), we cannot just open it in an editor to verify its correctness. To verify its correctness, un-comment the main program and write code to complete the functions:

- `read2DArray` - passed a binary file pointer opened for reading and returns the # rows, the # columns, and the 2D array of doubles. You'll need to use the `fread` function to read the contents of the binary file -- see the tutorial on files at: http://www.cprogramming.com/tutorial/cfileio.html
- `equal2DArray` - passed the # rows, # columns, two 2D arrays, and a tolerance. It returns TRUE if corresponding array elements are equal within the specified tolerance; otherwise it returns FALSE.

Re-compile and run the program with values (`./write2D 5 7 myFile.dat 5.0 9.0`) to make sure that it works.

Part C: Matrix Multiplication is a frequently used numeric calculation that takes two matrices (i.e. 2D arrays) A (m rows x q columns) and matrix B (q rows x n columns) and produces matrix C (m x n), where c_{ij} (i.e., C[i][j]) is the *dot product* of the ith row of A with the jth column of B. In other words,

$$\sum_{k=0}^{q-1} c_{ij} = a_{ik} * b_{kj}$$

For example:

$$A \times B = C$$

$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 2 & 1 \\ 2 & 2 & 1 \\ 0 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 2 & 2 \\ 1 & 0 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 9 & 5 \\ 4 & 1 \\ 8 & 5 \\ 7 & 2 \end{pmatrix}$$

The sequential algorithm for matrix multiplication is:

```
for i = 0 to m-1 do
    for j = 0 to n-1 do
        c_ij = 0
        for k = 0 to q-1 do
            c_ij = c_ij + a_ik * b_kj
        end for k
    end for j
end for i
```

Using an editor on `student.cs.uni.edu` open the file `mmultSeqOptions.c` which contains a program that allows the user to enter an integer (# rows and # columns of square matrices A, B, and C) on the command-line, and times the calculation of the matrix multiplication of randomly generated A and B matrices. It actually calculates the product matrix C as described above (i.e., `matrixMultiplication` function) and product matrix C_alt using a slight variation by `matrixMultiplicationAlt` function.

Answer the following questions about the `mmultSeqOptions.c` program:

a) How is the time to perform the `matrixMultiplication` function determined?

b) Compile (`gcc -o mmult mmultSeqOptions.c -lm`) and run the program (e.g., `./mmult 1000`) several times to complete the below table:

Matrix Sizes	Time of `matrixMultiplication` function (in seconds)	Time of `matrixMultiplicationAlt` function (in seconds)
500 x 500		
750 x 750		
1000 x 1000		
1500 x 1500		
2000 x 2000		
2500 x 2500		
3000 x 3000		

c) Explain why the `matrixMultiplicationAlt` function is faster than the `matrixMultiplication` function.

Submit lab5.zip containing question answers and completed programs on the eLearning system