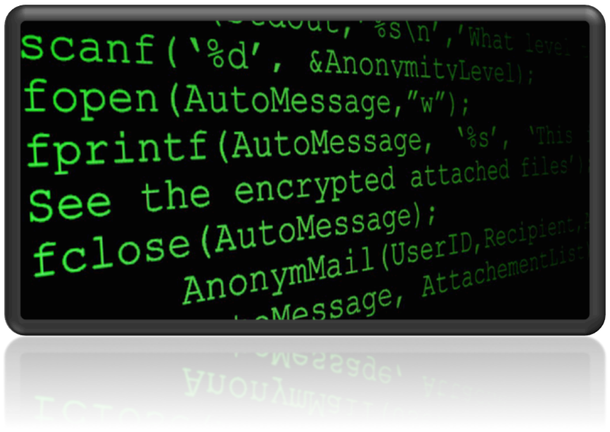
Comp 428
Parallel Computing

Assignment One



```
scanf('%d', &AnonymityLevel);
fopen(AutoMessage,"w");
fprintf(AutoMessage, '%s', 'This
See the encrypted attached files')
fclose(AutoMessage);
    AnonymMail(UserID,Recipient,
    Message, AttachementList
```

Due date: 11:55 PM, Sunday, September 27.

Note: All submissions are made via Moodle (see "deliverables" section).
Late penalties (< 1 hour = 10%, < 24 hours = 25%) will be applied automatically.

After you upload the assignment, please verify that your submission is recorded as submitted.

Note: Graduate students will be required to solve additional problems and/or provide additional software components. These additional components will be clearly indicated in the description below.

Part A: Concept questions

Total Points for Undergads: 30
Total Points for Grad Students: 40

1] (8 points) (a) Assume you have a multi-stage network, supporting 16 processors and 16 memory banks. Draw the network structure, labeling the processors and memory banks in binary format. (b) Then, in the diagram, clearly indicate the route for a request from P9 to M2. You must show your work, in terms of how the routing decisions were determined.

2] (6 points) Draw a 5D hypercube, including the binary node labels, using the recursive construction techniques described in the notes.

3] (4 points) Provide the diameter, bisection width, arc connectivity, and cost of a *p* node Ring network.

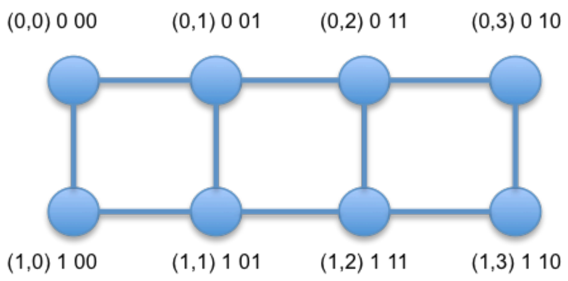
4] (2 points) What is the connectivity of an 8 dimension hypercube? Your answer should be in the form of an integer, not a function. Explain your answer.

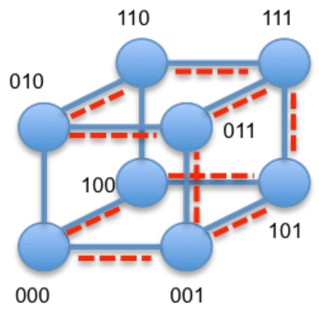
5] (10 points) (a) Given the image of the 4D hypercube in the notes, draw the route from P3 to P14. You must show your work, in terms of how the path was determined. (b) Given a message startup cost of 8 ms, a per word transfer time of 6 ms, and per hop routing overhead of 2 ms, what would be the total communication time for a 1 KB message sent along this path, assuming a word size of 32 bits.

6] **Grad students only**. (10 points) When mapping the Ring network into the hypercube, we made use of binary Reflected Gray Codes. The notes described the tabular technique for extending the Gray codes into additional dimensions. The codes can also be represented in a compact form as follows:

G(i, d) = the *i*th entry in the sequence of Gray codes of *d* bits. For example, G(1,2) = 01.

We can use this approach to embed a 2^r x 2^s wraparound mesh into a 2^{r+s} hypercube, by mapping process (i,j) of the mesh onto processor *G(i,r)* || *G(j, s)* of the hypercube (where || denotes concatenation of the two gray codes). Consider a 2 x 4 mesh mapped to an 8-node hypercube. Here, processor (i, j) of the mesh is mapped to processor G(i, 1)|| G(j,2) of the hypercube. So, for example, processor (0,0) of the mesh is mapped to processor 000 of the hypercube since G(0,1) = 0 and G(0,2) is 00. This leads to the following mapping:





Using this same logic, create a diagram to illustrate the mapping of a 4 x 4 mesh to its corresponding hypercube.

Part B: Small C program

Total Points for Undergads: 30
Total Points for Grad Students: 40

In this first assignment, you will get a chance to practice your C programming…just in case you are a little rusty. The program that you will write is not large, but it should be enough to ensure that you are familiar with the differences between C and Java and that you are ready to proceed with more complex MPI programming.

You will be required to utilize basic C functions from the standard C library. You will also have to use a few *system calls* provided by the operating system. These are also just C functions. However, system call syntax varies between operating systems. As such, you MUST test your program on a Unix/Linux/Mac system that follows the POSIX standard (all of you have access to the Faculty's Unix/Linux systems). Windows use different OS functions and the code may not run properly if ported to a Unix system by the grader.

Keep in mind that there are countless online examples of the proper use of both C library functions and the various system calls. Feel free to use them as a starting point.

Description: The goal is to construct a very simple, interactive file management utility. You will provide the user with a basic option screen. From there, they will select a specific option and a new screen may appear.

Main Menu

When the program starts, the existing screen contents will be cleared and the following text menu will appear.

```
+++++++++++++++++++++++

 Sys Admin Utility

   1. File Explorer

   2. Find (undergrads)/ Find and Replace (grad students)

   3. Show the Log (graduate students only)

   4. Exit

+++++++++++++++++++++++

Selection:
```

Once the user selects an option (basic error checking must be applied of course), the screen will change to a new screen (you can just use **system("clear");** to get the OS to clear the screen for you). It just looks a little less cluttered this way.

File Explorer

If the user selects File Explorer, they will see a new screen that will provide basic display functions. So the user will see the following screen:

```
+++++++++++++++++++++++

File Explorer

   1. File List (sorted by name)

   2. File List (sorted by size)

   3. Change Directory

   4. Main Menu

+++++++++++++++++++++++

Selection:
```

Again, the user will make a selection. Once a selection is made, the results will be printed to a fresh screen. If a file listing is displayed, the user will then see:

```
Press any key to continue
```

Once a key is pressed, the File Explorer screen will be displayed again, waiting for new input.

With respect to the 4 options, the basic output is listed below:

File List (sorted by name)

You will list the contents of the current directory (don't list contents of sub directories), in this case sorted in ascending order by file name. In this case, you will be using the opendir(), readdir(), and closedir() system calls (i.e., C functions) to access the directory entries. The readdir() function is use to loop through the names of files in the directory. The stat() system call is then used to get info about each file.

The output will be displayed as follows

```
File_Type    Size (Kb)     File_Name
```

The file type will be listed in text form as "Regular", "Directory", "Symlink", etc. Size should be listed in kilobytes. The name will be listed last (since it can be of arbitrary length).

Note: Do NOT write your own sorting routines (unless you are a glutton for punishment). Instead you will use the qsort( ) function from the standard C library. If you haven't used it before, you can check the man pages or just look at an example on the web. Basically, you just have to define a small comparison function and qsort will do the rest.

Sorted by Size

Same as above but the list is sorted by file size, with the largest file listed first.

Change Directory

Directories form a tree structure (a graph if you include symbolic links). The file listings above only show the contents of the current directory. So this option provides a prompt to allow the user to specify a sub-directory, or to move up to the parent directory (“..”). In case you haven’t guessed, there is a chdir() system call.

```
Directory name:
```

If the user inputs a valid directory, you move to that directory and display the File Explorer menu again.

Main Menu

This brings you back to the main selection screen.

Find (undergrads)

The idea here is to find a user-specified text string within a user-specified plain text file (the search doesn’t have to work with binary files like *.doc files). So there will be a pair of prompts:

```
File Name:
```

If the user enters a valid file from the current directory, then a second prompt will appear:

```
Enter text to find:
```

You will enter a string. The output will be the line number(s) and line(s) of text that contains the entered string. It might look like this (if you entered “dog”):

```
34      This is the house where the dog lived

143     A dog is man’s best friend
```

This is an exact match search, so the case must match exactly (this is easier).

Find and Replace (grad students)

This is similar to the previous Find version, but you will provide three prompts, as follows:

```
File Name:
```

You will then prompt for the text string to match:

```
Enter text to find:
```

You will then be prompted with:

```
Replace text with:
```

At this point, you replace all versions of the old string with the new string. Again, you will only have to worry about plain text files. When the user selects this option and enters valid data, you will display the old version of the file, followed by the new version (you do not have write the new version back to disk). In the new version, the new text string will be surround by "**" on both sides so that it is easy to find. For example you might have something like this, assuming the original string is "dog" and the new string is "cat":

OLD FILE:

```
The black dog was mean.
I don't like dogs.
```

NEW FILE:

```
The black **cat** was mean.
I don't like **cat**s.
```

Show the log (grad students only)

Each time a user selects an option in one of the screens you will append a descriptive string to an array (e.g., "create new file abc.txt"). If you want to see a log of all the things you have done, you can just select this option from the main menu and the contents of the array will be displayed to the screen. That's easy you say. Well, there is just one other thing. You don't know how many menu selections will be made during a session. So you can't create a fixed size array to hold the elements unless you make an enormous array…and this would be poor programming practice. So you will use a dynamic string array. C doesn't provide such a thing in the standard libraries but conceptually this is much the same as the vector structures seen in C++ (you can't use that one) or Java. Basically, you will initialize your array to a certain size (initialize size = one in this case). You will then add strings to the array. When the current size is exceeded, you must transparently resize the array (you will double it each time it needs to be increased). In this case, your array of strings will grow as large as it needs to be without ever being more than a factor of two larger than its minimal size. When the user selects the Show the Log option, you print the contents of the current log.

The log file code must be in a separate source file called "log.c". The marker will check this source file to make sure that you really do dynamically resize the array.

Deliverables:

Part A: The concept questions should be prepared in a Word Processor and then saved in a pdf file named ***A1_concepts.pdf***

Part B: All source code is to be written in C. Multiple source files will be used as follows:

The bulk of the code will go into a source file called "utility.c". The code related to the find or find and replace will go into a file called "find.c". The code for the log file (grads only) will go in a file called "log.c". Finally, you will have a simple makefile called "makefile" that will compile the code into an executable called *utitlity.exe* (Unix doesn't normally use the .exe extension but we will use it here for simplicity).

You should include a README text file to indicate what parts of the programming assignment work or don't work. This may make it easier for the grader to give you points if there are problems/limitations with your solution.

Final Submission: All files (concept questions and source code) should then be combined into a single zip file as indicated below and submitted to Moodle.

LastName_FirstName_A1.zip **(e.g., Smith_John_A1.zip)**

No other files should be included (no class files or extra files generated by your development environment). The source code will be extracted and compiled by the marker using the **make** command. The marker will then run the code to ensure correctness.