CISC 372: INTRODUCTION TO PARALLEL PROGRAMMING
Fall 2003
Group Project 1
Due Dates:
Deliverable 1: start of class, Thursday, October 17, 2003
Deliverable 2: start of class, Tuesday, October 28, 2003

1 Objectives

The main objectives of this project are:

- to gain the experience of the project group environment common in the computer industry,
- to creatively solve a somewhat open-ended problem after investigating multiple solutions with varying tradeoffs,
- to learn skills, apply knowledge, and seek new knowledge through a problem-based approach more resembling the real-world situation,
- and to write code for which the specification is being refined to fit the customer's changing requirements.

2 Problem Description

You are working on a big project for the National Security Agency (NSA). The NSA and the Department of Defense want to design a computer system named ATTACK (Automated Threat Trouncer and Covert Kruncher) that will be able to scan images acquired from satellite cameras, determine locations of military stockades, and orchestrate an ATTACK.

This system is to be a very large and rather complex system, so the work has been broken up into manageable pieces. Each unit of work will be assigned to a project group for completion. Your group has been charged with the image processing phase to enable computer visualization and target recognition. Your job is to implement a module which when given an image, returns an altered version of that image that delineates all the edges within the original image. Such a system is called an edge detection processor.

To get a feel for what your project should do, try running the edge detection processor located at: pollock/372_f03/public/edge.dir/edge on the cluster. In that same directory, there is a readme file that explains how to get this edge detector up and running (also where some initial test files can be found). You do not have access to the source code, just the executable and some test files. The test files are in the same directory.

It is important that your code be fast and correct; lives depend on your work. The initial specifications for your module are as follows:

3 Deliverable 1

A first prototype implementation. This prototype may be a correct sequential version of your system, or a first correct version of a parallel system.

1. Your software will run on an 9 processor cluster of machines in a mobile command unit. The machines will run MPICH (coincidentally the same configuration available at University of Delaware!). Your code will use the MPI library with the C or C++ programming language.
2. Images are stored in ascii PGM format (not raw or binary PGM format). Given a filename at runtime, your software must read the PGM file, process the image, and then save the resulting version of the image to a PGM file with the file name also specified at runtime, in a command line like:
   `mpirun -np X edge -i infile.pgm -o outfile.pgm`

3. Your program should accept any arbitrary size image, for which the matrix to store the image can be of any size (not necessarily divisible evenly by the number of processes).

4. Your software must perform all computations as quickly as possible.

5. Your software must perform all computations correctly.

6. You must clearly and thoroughly document your code. You must also provide a well written user manual that covers all aspects of use, explains algorithms used, describes any problems or shortcomings and why they exist, describes possible approaches to eliminating any problems or bugs, and clearly delineates all specifications of your software. You should include a section on how you tested your software.

7. **TURN IN:** Tar up a directory of: (1) your prototype program, (2) sample input files your program works properly on, (3) README file with external documentation (see item on user manual above), how to run the program, and your group members' names. Send the tarred directory to the TA by the deadline. ALSO, hand in a hardcopy of your program and scripted runs (see below). For help with tar, "man tar".

In the design of your initial program, you should consider the following questions.

- What algorithms are available for edge detection? Look at some books or the web for possible algorithms. **Please cite any of your sources if you code their algorithms. Your actual code should be your own group's creation, not from any other sources.**
- What general approach(es) could be used to parallelize the edge detection process? In each approach, where are the communication overheads?
- How does load balancing come into play in this task?
- What characteristics are desirable in a good suite of benchmarks for testing your code thoroughly for both correctness and performance?

4 Deliverable 2

A final parallel implementation showing your best effort toward efficient, effective parallelization. This version should be able to run correctly with 1, 4, 8, and 16 processes. You should show test runs, using a script file, for 1, 4, 8, and 16 processes. The final draft of the documentation should also be included with this deliverable.

The details of the code:

1. Your program needs to satisfy the same criteria as deliverable 1, except now be written general enough to be able to run with 1, 4, 8 and 16 processes.

2. Your code should be efficient in both time and space. For space, efficiency means that process 0 is the only one that should allocate space for the whole data set, and all other processes should only allocate space adequate enough to hold the data they need to hold and any data involved in communications. They should NOT all allocate a fixed size for the whole data set if they are not working on that size data. You need to be careful with two-dimensional arrays in C, especially when dynamically allocating them. See the notes at the end of this handout. You are allowed to store the image internally any way you believe is the most space and time efficient. Discuss your choice in the external documentation.

   For time, efficiency means using parallelism and performing communication efficiently. So, you should think about how best to distribute the data to minimize the amount of communication needed between processes during computation as well as minimizing the amount of duplicate data distribution work. In addition, when communication is needed, you should try to do it in an efficient manner, that is, avoid wait's if possible due to sequentialized communication schemes, and package up as much as possible in a message before sending. You are welcome to use any MPI command you find to achieve these goals, but it is perfectly fine to use only those we have discussed in class so far.

3. You must clearly and thoroughly document your code. You must also provide a well written user manual that covers all aspects of use, explains algorithms used, describes any problems or shortcomings and why they exist, describes possible approaches to eliminating any problems or bugs, and clearly delineates all specifications of your software. You should include a section on how you tested your software.

4. **TURN IN:** Tar up a directory that contains: (1) your final parallel program, (2) sample input files your program works properly on, (3) README file with external documentation (see user manual specifics above), how to run the program, and your group members' names. Send the tarred directory to the TA by the deadline. For help with tar, "man tar". ALSO, turn in a hard copy of your program, script, and graph with discussion.

5 Criteria for Evaluation

Your project will be evaluated according to the following criteria:

1. (45 pts) Deliverable 1.
   - (3) citations: adequate documentation of sources of information.
   - (5) file input/output
   - (4) arbitrary size image
   - (20) perform correct edge detection according to selected algorithm
   - (4) internal documentation of the code
   - (5) external documentation
   - (4) script demonstrating compile and executions

2. (55 pts) Deliverable 2.
   - (4) arbitrary size image
   - (15) perform correct edge detection according to selected algorithm for any number of processors
   - (5) a space-efficient parallel version
   - (9) a time-efficient parallel version in data distribution, load balance, and communication efficiency
   - (2) internal documentation of the code
   - (5) external documentation : choice of algorithm, issues, approaches to addressing time and space efficiency
   - (5) script demonstrating compile and executions
   - (8) single graph of timings from 3 runs (each) for 1,4,8,and 16 processes
   - (2) discussion of timings

3. Individual grades will be assigned as a percentage of the grade of the group project, based on feedback on how well the group worked together and fairly divided the work on the project. Each student will complete a peer review that has the goal of determining how well the group worked together, and how the workload was handled. (see course web site for review form example)

6 Notes on Dynamic Array Allocation in C

For a single dimension array:

I first declare the array as a pointer, because I don't want to have to give it a fixed size like int vector[100]; So, instead, I declare it as:

```
int *vector;
```

This create space for a single pointer called vector, in each process's address space. Now, if I want to allocate space, say for size number of int's, I write:

```
vector = (int *) calloc(size, sizeof(int));
```

or

```
vector = (int *) malloc(size * sizeof(int));
```

You can do a man on malloc or calloc and find out more.

Now, to access vector, I can either use pointer notation and pointer arithmetic, or I can use array notation.

So, I can do:

```
for(i=0; i<size;i++)
   printf("vector[%d]:%d\n", i, vector[i]);
```

to print out the values in the array. They should be all zero at this point, as malloc initializes the locations allocated to 0.

For two-dimensional arrays, it is much trickier: Here are some portions of my code:

In C, two-dimensional arrays are really an array of arrays, such that a[i][j] accesses the jth element of the ith array in the array of arrays. So, to declare a two-dimensional array that does not have space allocated yet, I need to declare a pointer to a pointer as:

```
int **grid;
```

Then, to allocate space for the whole grid of size, I can do it in two steps:

I first allocate the array of pointers which will point to the arrays. This can be done by:

```
grid = (int **) malloc(size * sizeof(int *));
```

This create space for size number of addresses; an address can be stored in the space of a pointer to an int.

Now, I have space as if I had size number of vector declarations. So, now I need to allocate the space for the vectors (single dimension arrays themselves). So, I do the same as I did for allocating a single vector, except I need size number of them, so I put it in a loop:

```
for (i=0;i<size;i++)
  grid[i] = (int *) malloc(size * sizeof(int));
```

Note that I can now access grid by the grid[i][j] notation, because it is fully allocated now.