

# Computer Architecture Laboratory

## SIMD Programming Using Parallaxis - I

This laboratory module has three goals:

1. To understand better SIMD and MIMD algorithms
2. To introduce the Parallaxis language and simulator
3. To learn Parallaxis by using it to write SIMD programs

There are two sessions for this lab

### Part 1

Games ... play each game using the following algorithms ... follow the instructor's directions.

- 1 SIMD
- 2 SIMD with processors idle
- 3 MIMD Shared Memory Prime Number Generator
- 4 MIMD Distributed Memory Prime Number Generator

### Part 2

The instructor will lecture on the language Parallaxis. Each of you will have a copy of the Parallaxis manual to use for this and the succeeding two labs.

### Part 3

Parallaxis is available on our ftp site (ftp anonymous to alcuin and look in /pub/arch/Parallaxis). Download the four files. There are two executable files, the compiler (pa.exe) and the interpreter (pz.exe) and two manual pages (pa.man and pz.man). This is the version for DOS. To run a program in Parallaxis, one does the following steps.

1. Use any ASCII editor to type in the program (for example, notepad) . Give it an extension of .p  
For example test.p
2. In a DOS window of Windows95 compile the program using the Parallaxis compiler pa  
For example pa test.p  
Will produces "object code" test.z
3. In a DOS window of Windows95, run the compiled code on the simulator pz  
For example pz test.z

Use Parallaxis to complete the four tasks given on the following two pages.

### Task #1

Following is SIMD algorithm for a "Sieve" program - a parallel version of a prime number generator - written in Parallax.

1. Study the program to understand how it functions. If you have questions, ask your instructor.
2. Type in, compile and execute the program.

```
SYSTEM sieve;

CONFIGURATION list[200];
CONNECTION (* none *);

SCALAR prime : integer;
VECTOR candidate : boolean;

BEGIN
  PARALLEL
    candidate := id_no >=2;
  WHILE candidate DO
    prime := REDUCE.First(id_no);
    WriteInt(prime, 10); WriteLn;
    IF id_no MOD prime = 0      (* remove multiples *)
      THEN candidate := FALSE
    END
  END
ENDPARALLEL
END sieve.
```

### Task #2

On page 85 of the parallax manual, in Program 8, is source code for a cellular automaton. A cellular automaton is a program that creates life and death on a grid. Think of the screen as a grid ... an 80X24 grid. Each point on the grid is alive (marked with a \*) or dead (an empty space). Copy this program into your account and execute it. Be sure you understand how it works.

You will need to add the procedure "out" which is found below. Put the code for this procedure where the "..." is in the example code.

```
PROCEDURE out;
  SCALAR values : array[1..n] of boolean;
         j : integer;
BEGIN
  PARALLEL
    STORE(val, values);
  ENDPARALLEL;

  FOR j := 1 to n DO
    IF values[j] THEN Write('#') ELSE Write(" ") END;
  END; (* FOR *)
END out;
```

### Task #3

The "Game of Life" is another cellular automaton. The game is very simple. The world is a grid. Some cells in the grid are alive and some aren't. As time passes, each living cell may continue to live, or it may die; each dead cell may stay dead or it may "spring to life" ... all depending on the number of neighbors the cell has. The neighbors are the 8 cells around the individual cell. The rules for life and death are:

1. Time is discrete. Generations of cells change all at once.
2. If a cell is dead and has exactly three living neighbors around it then, in the next generation it springs to life, otherwise it stays dead.
3. If a cell is alive and has less than two neighbors, in the next generation it dies from loneliness.
4. If a cell is alive and has more than three neighbors, in the next generation it dies from overcrowding.

Write a Parallaxis program for the Game of Life on a grid of 51 X 21. Use the simpler cellular automaton as a model.

You will need to:

Declare a processor structure to model your grid (a 2D structure)

Declare variables on each processor (i.e. vector) for current status, status in the next generation and any other "local" variables

Initialize the grid by putting values in a scalar 2D array and "loading" the processors

For each generation

Each processor must:

calculate its nearest neighbors by propagating data from those neighbors to itself (one-by-one) and counting (you'll need a loop for this)

if a cell is currently alive, determine if it will stay alive or die

if a cell is currently dead, determine if it will stay dead or spring to life

store the processor values in a scalar matrix

Print the results.

You can handle the boundaries by putting an empty "border" of empty cells around the dynamic matrix and not allowing the border cells to participate in birth and death processes (i.e. not all cells will be active).

Begin the process with a cross which has a 3X3 grid of live cells in the center of the matrix.

The program should display a generation, wait for [Enter] and display the next generation. The program should run 12 generations.

Mail the instructor the source code. Name the program life.p

#### Task #4

Modify the program by removing the "artificial border" of empty cells to create an unbounded space in the sense of a 2-D torus. I.e., the boundaries "wrap around" to themselves to make a "seamless" world.

In addition, allow the user to input the initial situation. I.e. ask the user:

- how many live cells there are
- the cell locations x,y of each live cell (use 1,1 as the lower left corner (for the user, not necessarily internal designation). Be sure to give the user an example of the input format.

Mail the instructor the source code. Name the program torus.p

## Algorithms for the Games of Part 1

- 1 SIMD  
Parallel  
    Flip coin  
    If heads then raise hand  
    Reduce.add  
Endparallel
  
- 2 SIMD with processors idle      Does nothing special  
Parallel  
    Flip coin  
    If heads  
        then flip coin  
            if heads then raise hand  
    Reduce.add  
Endparallel
  
- 3 MIMD Shared Memory      Prime Number Generator  
Parallel  
    As long as there are available numbers on the board  
        Obtain the first available number and mark it as unavailable  
        Cross out any remaining numbers that your number divides evenly  
Endparallel
  
- 4 MIMD Distributed Memory      Prime Number Generator  
Parallel  
    Receive message from proc on left  
    Copy message to local variable I  
    While message <> -1  
        Receive message from proc on left  
        If I does not divide the message evenly  
            Then Send message to proc on right  
Endparallel