- **Profiling:**

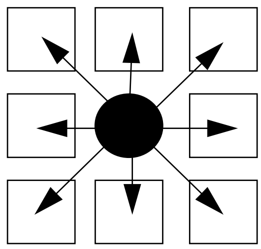
  Profile the code by including the **`-profile`** option on everythe command line. Remember to comment out all the input/output in the original code. Examine the output produced.

11.2 Exercise 2: The Game of Life

The aim of this exercise is to show how Fortran 90 can be used to program the Game of Life, a simple grid based problem with complex behaviour. It will show how Fortran 90 can be used to produce code in a very neat form and exposes the potential for coding in a data parallel programming style.

11.2.1 The Game of Life

The game of life is a simple cellular automata where the world is a 2D grid of cells which have two states: alive or dead. At each iteration the new state of a cell is determined by the state of its neighbours at the previous iteration. This includes both the nearest neighbours and diagonal neighbours, i.e.



The rules for the evolution of the system are;

- if a cell has exactly two alive neighbours it maintains state.
- if it has exactly three alive neighbours it is alive.
- otherwise, it is dead.

Your code will need to

1. Initialise board

Start loop

2. Print board
3. Calculate number of neighbours
4. if (neighbours = 3) then live
if (neighbours < 2) or (neighbours > 3) then die

End loop

The number of neighbours can be calculated using shifts, e.g.,

```
target = CSHIFT(source, shift, dimension)
```

sets **target** to be the same as **source** but with its elements shifted a distance **shift** along dimension of the array **dimension**. For example,

```
target = CSHIFT(source, -1, 1)
```

would set

```
target(i) = source(i - 1)
```

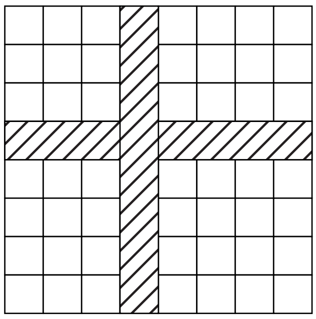
CSHIFT automatically performs periodic boundary conditions. Otherwise references would be made to elements outside the bounds of the array.

11.2.2 The Exercise

A template for this exercise is provided, which includes pointers to the completion of this exercise and also all the print statements necessary for viewing the results.

- **Initialisation**:

  Use array syntax to initialise a board (an **N×N INTEGER** array with value 1 for a live cell and 0 for a dead one) with the following pattern, where shaded cells are alive, where **N = 8** in this case. Set the row and column nearest the centre (i.e. **N/2**)



- **Print board:**

  The board can be printed either as a plain text file using the standard Fortran print format statements, or using the following pieces of code can be included to produce a series of bitmaps which can be animated using **xv**.

```
!      Include in the declarations; strings for filenaming
       CHARACTER (LEN=10) :: picfile
!...
! Include in the time loop
! hack to write time into a string
       WRITE(picfile,fmt='(''life'',I2.2,''.pgm'')') count

       OPEN(UNIT=10,FILE=picfile)
```

```
! write the board to a pgm file for viewing
      WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 1
! must be a capital P for the MAGIC NUMBER
      WRITE(10,*) board
       CLOSE(10)
```

This should, when run, produce **life_*.pgm** files which can be viewed using **xv**. Alternativly, these files can be viewed as animation using,

```
xv -expand 10 -wait 0.5 -wloop -raw *.pgm
```

- **Update**:

  Declare an array the same size as the board (**N**×**N**)to contain the number of neighbours for each point. Include all nearest neighbours, including diagonal neighbours. This update can be done using the following Fortran 90 features.

  - Use **CSHIFT** to calculate the number of neighbours. In order to access the diagonal neighbours you may need to used nested **CSHIFT**s. For example,

    ```
    target = CSHIFT(CSHIFT(source, 1, 2), -1, 1)
    ```

    This would shift array **source** initially in the 2nd dimension and then in the 1st.
  - Use **WHERE** to decide whether to create or kill new organisms at each grid point.

- **Compilation**:

  Compile your code (**life.hpf**, say, board size **N=8** for the test, for 10 iterations of evolution) with the script as in the previous exercise (**hpf -compile life**) to produce an executable **life**.

- **Single node**:

  Run the code on a single processor for 10 iterations to check produce a set of (**hpf -run life**).

  Use **xv** to view the resulting arrays to check your code is working.

11.2.3 Extra Exercise 1

CSHIFT was used to count the number of neighbouring live cells. You can optimise the code to do the count with only four **CSHIFT** operations instead of eight. This requires the use of a temporary array the same size as **board**. For each array element,

- create the temporary by adding the current value of **board** to the value of the left and right partners (two **CSHIFTS**)
- find the upper and lower partners for this temporary array (two more **CSHIFTS**) and add these to the temporary array
- each element now contains the number of neighbours, plus the original value of the cell. Subtract the original state to obtain the number of neighbours with four **CSHIFT**s.

11.2.4 Extra Exercise 2

Try to count and display the number of alive/dead/new/killed cells at each iteration. You will probably need to use some additional Fortran 90 features.

11.2.5 Extra Exercise 3

Define an extra array which is used for display. In this array keep track of which generation (modulo 256) that the cell was born. This array can be used for a colour display and so the age of the cell can be seen. To print out this colour array, change the header statement from

```
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 1
```

to

```
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 255
```

11.3 Exercise 3: Life Distributed

In this exercise we return to Exercise 2 and rewrite the codes including explicit data distribution directives.

- **Distribute Data**:

  Introduce data mapping directives into the code to distribute the data. So, for example, for 2D arrays **`source`** and **`target`** you could use

  ```
  !HPF$ DISTRIBUTE (BLOCK,BLOCK) :: source, target
  ```

  to distribute the elements of these arrays in a block form.

- **Compilation**:

  Compile the codes as before, using the **`hpf`** script. Increase the size of your array to **`N=100`**.

- **Single Node**:

  Run the executable on a single node. Check it gives the same results as before.

- **Multiple Nodes**:

  First create a **`MYHOSTS`** file in the working directory with four workstation names including your own (the training room machines are called **`young`**, **`sutherland`**, **`stewart`**, **`crawford`**, **`ramsay`**, **`blair`** and **`coventry`**). You need to have a **`MYHOSTS`** file in each directory where you run executables on more than one processor).

  Run the code on four processors, using

```
hpf -run -nodes 4 life
```

and check you still get the same answer!

- **Profiling**: Repeat the last stages with the **-profile** option set in each command. Remember to comment out all the input/output statements when profiling.

Use the resulting information to investigate how the performance of these examples changes with different data distributions. In particular profile the codes and see how the amount of communication changes. What is the most efficient distribution for the Game of Life and why?

For example, change the distribution of the 2D arrays in the Game of Life from **(BLOCK,BLOCK)** to, say, **(CYCLIC,CYCLIC)** or using a degenerate distribution.

11.4 Exercise 4: The Mandelbrot Set

The Mandelbrot Set is the set of numbers resulting from repeated iterations of the following complex function:

$$Z = Z^2 + C$$

which, separating real and imaginary form, looks like,

$$z_i = 2.0 * z_r\, z_i + c_i$$

$$z_r = {z_r}^2 - {z_i}^2 + c_r$$

for complex numbers, Z and C. This function is defined for complex values of $C=(c_r,c_i)$ in the range $([0.0,1.0], [0.0,1.0])$, with the initial conditions, $z=c$. In the case study, we will mainly be concerned with the Mandelbrot Set defined in the first quadrant of the complex plane, i.e. we will consider c in the range ([0.0, 1.0], [0.0,1.0]).

What is normally plotted is the number of iterations taken for z to reach some threshold value. We will take this threshold as

$$|z|^2 > 4.0$$

and set an upper iteration limit of 255. This number is converted into a greyscale or colour, and plotted at the point whose coordinates are (c_r , c_i).

11.4.1 FORTRAN 77 Code

Below is a fragment of a serial code which computes the Mandelbrot set. This should outline the algorithm to be used in the full HPF version.

```
c Declare N*N arrays for real and imaginary parts of
c arrays C and Z
c i.e. declare real arrays CR, CI, ZR, and ZIs
c     ...
c Initialise arrays CR, CI
c Initialise arrays ZR=CR, ZI=CI
c     ...
c Initialise ZIS and ZRS to hold the squares of ZR and ZI
c     ...
      DO i = 0, 255
        DO j = 1, N
          DO k = 1, N
            IF (ZRS(j,k) + ZIS(j,k) .LE. 4.0 ) THEN
              ZRS(j,k) = ZR(j,k) * ZR(j,k)
              ZIS(j,k) = ZI(j,k) * ZI(j,k)
              ZI(j,k) = 2.0 * ZR(j,k) * ZI(j,k) + CI(j,k)
              ZR(j,k) = ZRS(j,k) - ZIS(j,k) + CR(j,k)
              COLOUR(j,k) = i
            END IF
          END DO
        END DO
      END DO
c ...
```

11.4.2 HPF Version: Serial

Write an HPF program, based on the above code fragment, which will compute the Mandelbrot set, using the HPF/Fortran 90 array features to carry out the various stages (a template is available).

First off write a serial HPF version. A template is included with various pointers on how to proceed (and also the print statements required for a graphical output). The following steps are needed:

- **Initialisation**

  Initialise the *rows* of **CR** (real component of *C*) to be in the range [0.0,1.0] (i.e. every element in the first row has value 0.0, every element in the last row has value 1.0, and the intermediate rows have values varying between 0.0 and 1.0). Initialise the *columns* of **CI** (imaginary component of *C*) similarly. Use **FORALL** for this initialisation.

  Set the initial conditions **ZR** and **ZI** to be **CR** and **CI** respectively and define variables **ZRS** and **ZIS** to be the square of **ZR** and **ZI** respectively.

- **Iteration**

  Rewrite the above FORTRAN 77 code using Fortran 90 array syntax, using the **WHERE** construct.

  This program takes each point at a time and iterates the complex function (up to a maximum number of iterations, **RESOLUTION = 255**). The iterations stop when the absolute value of Z reaches or exceeds 2. The colour values are a measure of how many iterations it took to “escape” to 2 at each point.

- **Output**

  To write out the **COLOUR** array as a bitmap, use similar instructions as were used in the Game of Life. Include the following lines of code at the end of your program (this should already be in the template),

```
OPEN(UNIT=10,FILE='mandel.pgm')
WRITE(10,fmt='(''P2'',/,I3,2X,I3,/,I3)') N, N, 255
WRITE(10,*) colour
CLOSE(10)
```

Compile and run the code on a single workstation, with the size of the arrays set with **N = 128**. View the resulting bitmap with

```
xv -gamma 7 mandel.pgm
```

and check that the code works correctly (i.e. you recognise the bitmap to be the well known Mandelbrot set).

11.4.3 HPF Version: Parallel

Include the necessary HPF data mapping directives to distribute and align the arrays. Try both **BLOCK** and **CYCLIC** distributions. With this done, all the work is automatically distributed and the code “should” run in parallel.

Compile and run the code for 4 nodes. Check that the answer is still the same.

Use the profiling option to time the execution of the code and the amount of message passing involved.

Use this to compare how the performance is affected by use of different data distributions. Remember to comment out the input/output statements in the code while profiling.

11.5 Exercise 5: Birthday

The purpose of this exercise is to choose appropriate **DISTRIBUTE** and **ALIGN** directives for a particular problem.

Our example is that of finding the day of a year any given individual’s birthday falls within a particular (non-leap) year. A simple way to do this is to produce a look-up table with the number of days in each month (or more appropriately, a running total of these values), which is consulted for every birthday. The day on which the birthday falls is the sum of the total number of days so far plus the number of days in that particular month.

The following code defines data structures and data initialisation along with a print statement useful for viewing the results (a template is provided with code already written).