Parallel Programming with Coarray Fortran: Exercises

Introduction

The aim of these exercises is to familiarise you with writing parallel programs using the coarray features of Fortran. In particular, they illustrate the importance of synchronisation calls in ensuring correctness.

Logging on

You will be given an account on `kraken`, a Cray XT5 supercomputer at the University of Tennessee. The account name is `std0###` (where ### is 001, 002, ...) as already assigned to you. To log on and enable graphics use: `ssh -X std0###@kraken.nics.tennessee.edu`

Obtaining the exercises

The exercises are supplied in a compressed tar file `CAF-exercises.tar.gz` – this is available in `/nics/b/home/std0069/CAF/`. Unpack using `tar xvf CAF-exercises.tar.gz`. You must work in the Lustre file system `/lustre/scratch/std0###/`, not in your home directory.

Code notes

There are three directories in the exercise package: `templates`, `solutions` and `doc`. The first two contain a separate subdirectory for every exercise. You are provided with sample input files and reference output files, and a source file `pgmio.f90` which contains a module with simple routines for reading and writing PGM pictures. You can find a PDF copy of this document in `doc/`.

You are also given template codes which provide a convenient framework for completing each exercise. These should compile and run as supplied, but will not produce the correct output as they generally contain useful variable and array definitions but only a small amount of actual code. In particular, they will not have any synchronisation in them so may not work correctly on multiple images. The solutions, however, are full working codes which are based on the templates.

We recommend that you start from the templates, at least for the initial exercises. However, they are just suggestions on how to tackle the exercises and take a very simple approach – feel free to write the solutions in your own way if you wish. For example, you might prefer to use allocatable arrays rather than the static definitions proposed in the templates.

Compilation

Coarray features are available using the Cray compiler. This requires you to load a non-default module. If this is not already loaded then you must load it each time you log in:

```
module switch PrgEnv-pgi PrgEnv-cray
```

The compiler is called `ftn`. For example, to enable coarray features and create an executable `ex1a` from the file `ex1a.f90`:

```
  ftn -h caf -o ex1a ex1a.f90
```

For later exercises that read or write PGM files, you will need to compile the file pgmio.f90 first then include it in subsequent compilations, e.g.

```
  ftn -h caf -c pgmio.f90
  ftn -h caf -o ex1b ex1b.f90 pgmio.o
```

Linking can be quite slow on Lustre so do not be surprised if compilation takes ten seconds or more.

Execution

Coarray codes cannot be executed interactively on `kraken` – you must submit jobs to the PBS batch system. You are given a template batch script `ex1a.pbs` in the `templates/ex1/` subdirectory:

```
#!/bin/bash
#PBS -l size=12
#PBS -l walltime=00:01:00
#PBS -j oe
#PBS -A UT-NTNLEDU

# Set number of images which must be the same as specified in your code
numimage=8

# change directory to where the job was submitted from
cd $PBS_O_WORKDIR

echo "Starting job $PBS_JOBID at ‘date‘"

aprun -n $numimage ./ex1a

echo "Finished job $PBS_JOBID at ‘date‘"
```

This will run the executable `ex1a` using 8 images with a maximum runtime of 1 minute. When you copy this script to run other jobs, make sure you change the name of the executable supplied to `aprun`.

The total number of images to run on is specified using `numimage`. The value of `size` is the number of cores reserved for your job, which must be greater than or equal to `numimage`. As each XT5 node has 12 cores, `size` must be a multiple of 12. For long runs you may have to increase the wallclock time.

Submit to the batch system using `qsub ex1a.pbs`. Once the job has finished running, the output and any errors will appear in a file called `ex1a.pbs.oXXXXX`, where XXXXX is a unique job number assigned by PBS. You can follow the progress of your jobs using `qstat -u std0###`.

Viewing PGM pictures

You can view PGM pictures using the `xview` program, e.g. `xview output.pgm`. This can be quite slow so be patient! Alternatively, I have installed a version of the ImageMagick `display` program which you should be able to access by updating your PATH:

```
  export PATH=$PATH:/nics/b/home/std0069/bin
```

and you can then just use `display output.pgm` to see any pictures. The `display` program is more forgiving than `xview`, and will let you visualise the input files (which contain negative values!) as well as the more well-formed output pictures.

Exercise 1: Hello World

The aim of this initial exercise is to ensure you can compile and run simple coarray programs on your target platform. You will also do some basic parallel operations involving communication (and synchronisation) between different images.

(a) Write a simple program where each image prints out its image index (using `this_image()`) and a single master image, e.g. image number 1, prints out the total number of images (using `num_images()`). Compile and run the program on different numbers of images.

(b) The template code `ex1b.f90` defines an array to store a picture which is written to file for subsequent viewing (we will build on this in later exercises). For simplicity, the number of images is set as a parameter in the code – we must therefore check at runtime that the actual number of images is equal to this value. In a real program you would probably use allocatables and dynamically create arrays of the appropriate size. However, using a compile-time parameter makes it easier to declare arrays of the right dimensions at the expense of having to recompile for different numbers of images.

Extend the program to:

- Declare a coarray `smallpic(nx, nylocal)[*]` to allow `bigpic(nx, ny)` to be distributed across images in the y-dimension.
- Initialise `smallpic` on each image to be equal to `this_image()`.
- On the master, copy each piece of `smallpic` from the other images to the appropriate location in `bigpic`.
- Write the resulting picture on the master.
- View the result.

You will need to put in synchronisation to ensure that the local images are initialised before they are copied back by the master. What happens if you omit this synchronisation?

(c) As an additional exercise, have each image write its data to the correct part of `bigpic` on the master. You will need to make `bigpic` a coarray, and perhaps introduce additional synchronisation.

Exercise 2: Edge Detection

The aim of this exercise is to illustrate how parallel halo-swapping algorithms can be implemented using coarrays. We will do this by implementing a simple graphics-processing method for detecting the edges of features contained in a picture.

For simplicity, we define the edges of a picture by comparing the values of each pixel to its four nearest neighbours:

$$edge(i,j) = pic(i-1,j) + pic(i+1,j) + pic(i,j-1) + pic(i,j+1) - 4\,pic(i,j)$$

If a pixel has the same value as its four surrounding neighbours (i.e. no edge) then the value of $edge(i,j)$ will be zero. If the pixel is very different from its four neighbours (i.e. a possible edge) then $edge(i,j)$ will be large in magnitude. If you are familiar with the discretisation of partial differential equations, you will recognise that $edge$ is the second derivative of pic.

We will always consider i and j to lie in the range $1, 2, \ldots nx$ and $1, 2, \ldots ny$ respectively. Pixels that lie outside this range (e.g. $pic(i,0)$ or $pic(nx+1,j)$) are set to zero.

(a) The template `ex2a.f90` simply reads in a picture and writes it out again. Note that the arrays have been extended in each dimension to accommodate the boundary conditions – in the serial program, we can implement the boundary conditions by setting these halos to zero. You should take care that computation only take place on the interior of the pictures, e.g. loops should start at 1 and not 0.

Extend this to:

- Use the approach of the first exercise to scatter the picture to `oldpic` on multiple images, copy to `newpic`, and then gather `newpic` back to `bigpic` on the master. In the template, the `bigpic` array is deliberately overwritten immediately after input to help you in verifying this scatter/gather operation.
- Implement the edge detection method above to compute the edges in `newpic` based on the input stored in `oldpic`, and look at the output picture. You should compare to the reference picture `reference.pgm`. Without halo-swapping, the result will not be entirely correct. However, the output should look the same everywhere except at the image boundaries where you will see artifacts.
- In parallel we must populate the halos with data from neighbouring images prior to edge detection, i.e. copy the upper halo from the upward neighbouring image etc.

  ```
  oldpic(:, nylocal+1) = oldpic(:, 1)[myimage+1]
  oldpic(:, 0) = oldpic(:, nylocal)[myimage-1]
  ```

  You should ensure that you introduce appropriate synchronisation, and that the first and last images do not attempt to access non-existent neighbouring images.
- Run the code on multiple images and check that it works correctly. Note that incorrect synchronisation can lead to quite subtle errors so you should `diff` your result with the reference output in addition to performing a visual check.

(b) Although edge detection can be parallelised as above, it only requires a single pass of the picture and is therefore too simple and fast an operation to show any useful parallel speedup. However, it is actually possible to do the inverse operation, i.e. to reconstruct the initial picture from its edges. This requires multiple iterations, and so might run significantly faster on multiple images when parallelised.

You are supplied with an input edges file `edge240x320.pgm`, and you should be able to reconstruct the original picture as follows:

- define a new coarray called `edge`
- read the initial edges data file into `bigpic`
- zero the arrays `oldpic`, `newpic` and `edge`
- scatter `bigpic` to `edge` and set `oldpic = edge`
- repeat for many iterations:
  - loop over $i = 1, 2 \ldots nx; j = 1, 2, \ldots ny$

    $$newpic(i,j) = \frac{1}{4}\left\{\, oldpic(i-1,j) + oldpic(i+1,j) + oldpic(i,j-1) + oldpic(i,j+1) - edge(i,j) \,\right\}$$
  - end loop
  - set `oldpic = newpic`
- end loop over iterations
- write out the final picture as before

For information, this is the Jacobi algorithm for solving the 2D Poisson equation $\nabla^2 pic = edge$.

For verification, compare your output after 100 iterations to `reference100.pgm`. Again, incorrect synchronisation can lead to errors that may not be immediately visible by eye so you should also `diff` the two pictures. You should start to see the original picture emerging after around 1,000 iterations; after 100,000 it should be practically indistinguishable from the original.

(c) Rather than running for a fixed number of iterations, it is better to stop when the output picture has converged to some tolerance, i.e. when it changes very little between each iteration. We can quantify this by computing Δ defined by:

$$\Delta^2 = \frac{1}{nx \times ny} \sum_{i=1;j=1}^{nx\ ;\ ny} \left(newpic(i,j) - oldpic(i,j)\right)^2$$

and stopping when Δ is less than some tolerance, e.g. 0.1.

This can be computed locally on each image, then the partial results summed over images. Write a subroutine or function that globally sums private double precision numbers by placing them in a coarray and adding them together across images.

(d) Rather than using global synchronisation, try using the routine `sync images()` to use pairwise synchronisation between neighbours when swapping the halos.

(e) Use a larger picture (as supplied) and time your program to measure the speedups obtained. Note that you may need to reduce the number of iterations to get sensible runtimes. It is likely that the computation of Δ will prevent the code from scaling: you can either run for a fixed number of iterations, or only compute Δ periodically, e.g. every 100 iterations. To avoid recompilation for different numbers of processes, you could introduce allocatable co-arrays.

Exercise 3: Multiple Codimensions

If you have completed the previous exercises then you are doing very well! This exercise expands on them and introduces coarrays with more than one codimension. It will give you scope to continue working after the tutorial should you have access to a coarray compiler.

The exercise is based on picture reconstruction and we therefore assume here that you have already completed part (b) (and ideally also part (c)) of exercise 2. However, you can still usefully investigate arrays with two codimensions starting with just the solution to exercise 2 part (a).

(a) The exercise is to extend the previous picture reconstruction program to use a two-dimensional decomposition of the picture across images. This requires more book-keeping to compute and store the coordinates of each image in the two codimensions, and also requires halo swapping to take place in both the x and y dimensions.

The template `ex3a.f90` contains the framework for this exercise and sets up various useful lookup tables. As supplied it will only run correctly on a single image. To arrive at a complete solution:

- Add code to scatter the picture out after input and gather back before output.
- Run the code on multiple images and visualise the output– you should see partial image reconstruction but with obvious boundary effects from the two-dimensional decomposition.
- Add code and synchronisation for halo-swapping in both dimensions.
- Visualise the output and compare to the reference pictures.

(b) Terminate the iterative loop based on the value of Δ as defined in exercise 2 (c). You should be able to use *exactly* the same subroutine or function that you wrote previously. Within the subroutine you can still define the coarrays to have a single codimension even if they have multiple codimensions in the main program. This means the same global sum code can be used on arrays with any number of codimensions.

Appendix: Array layouts for pictures

The supplied IO routines display the arrays as pictures using an x-y coordinate system, i.e. the first index `i` is taken as the x-coordinate, the second index `j` as the y-coordinate. This convention is also used in the solutions when defining what neighbouring images are "up" and "down".

Here are some pictures of the coordinate systems and possible decompositions as used in the templates.

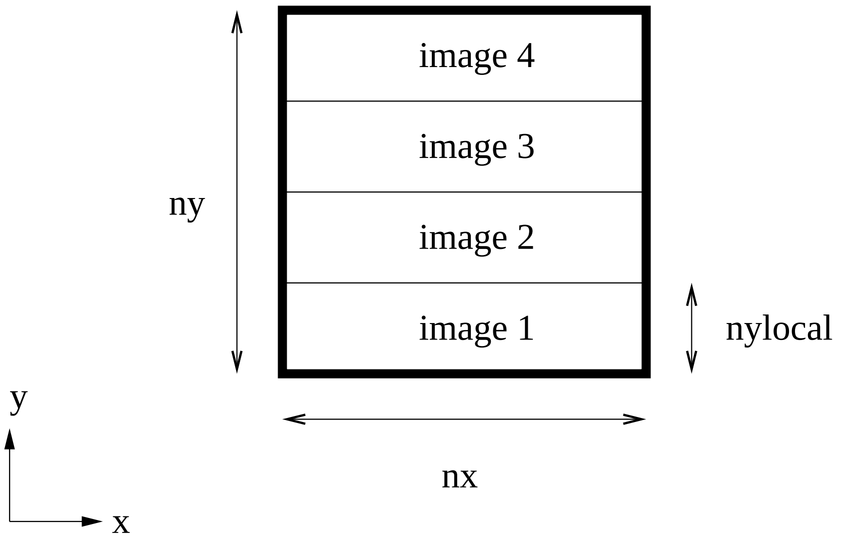
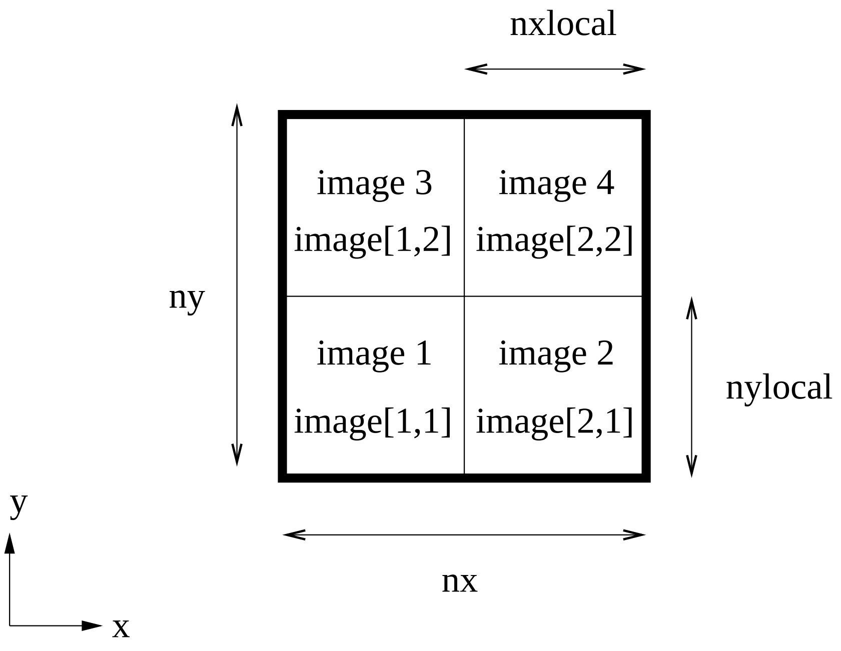


image 4
image 3
image 2
image 1
ny
nylocal
nx
y
x


nxlocal
image 3
image[1,2]
image 4
image[2,2]
image 1
image[1,1]
image 2
image[2,1]
ny
nylocal
nx
y
x