Parallel Programming with OpenSHMEM: Exercises

Introduction

The aim of these exercises is to familiarise you with writing parallel programs using OpenSHMEM commands. In particular, emphasis is placed on the use of synchronisation calls in ensuring correctness.

Logging on

You should already have access to an account on ARCHER, the UK National Supercomputer, which is a large Cray XC30 system. To log on and enable display of graphics use: `ssh -X login.archer.ac.uk`

Obtaining the exercises

The exercises are supplied as a single compressed tar file called `synchronisation.tar.gz` – this can be retrieved from the ARCHER course web pages using `wget`.

Unpack the tar file using `tar xvf synchronisation.tar.gz` on ARCHER. You should use the Lustre `/work/` file system rather than your home directory.

Code notes

There are two directories in the exercise package: `templates` and `solutions`. These each contain separate subdirectories for Fortran and C versions of the exercise code. You are provided with sample input files and reference output files, and a source file `pgmio.f90/pgmio.c` which contains a module with simple routines for reading and writing PGM pictures.

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 PEs. 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 template, taking care to ensure symmetric memory is used as required.

Compilation

The code can be built using the Cray compiler suite, the default on ARCHER. To ensure the appropriate build paths are set, the `cray-shmem` module must be loaded prior to compilation.

```
module load cray-shmem
```

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

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

Required commands are as follows:

Fortran

To create an executable `ex1a` from the file `ex1a.f90`:

```
ftn -o ex1a ex1a.f90
```

To build and include `pgmio.f90`:

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

C

To create an executable `ex1a` from the file `ex1a.c`:

```
cc -o ex1a ex1a.c
```

To build and include `pgmio.c`:

```
cc -c pgmio.c
cc -o ex1b ex1b.c pgmio.o
```

Execution

You may run the code by submitting jobs to the PBS batch system. You are given a template batch script called `ex1a_archer.pbs` in the `ex1` subdirectory within `templates`:

```
#!/bin/bash --login
#PBS -A y14
#PBS -l select=1
#PBS -l walltime=00:01:00
#PBS -j oe

NPROC=8
NTASK=8

# change directory to where the job was submitted from

cd $PBS_O_WORKDIR

echo "Starting job $PBS_JOBID at `date`"

aprun -n $NPROC -N $NTASK ./ex1a

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

This will run the executable `ex1a` using 8 processing elements (PEs) with a maximum runtime of 1 minute.

The total number of PEs is set using the `NPROC` variable, which is directly linked to the numerical argument of aprun's `-n` option. The number of PEs per node can be set using the `NPROC` variable, similarly linked to the `-N` argument.

For long runs you may have to increase the wallclock time.

When you copy this script to run other jobs, make sure you change the name of the executable supplied to `aprun`.

Submit to the batch system using `qsub ex1a_archer.pbs`. Once the job has finished running, the output and any errors will appear in a file called `ex1a_archer.pbs.oXXXXX`, where XXXXX is a unique job number assigned by PBS. You can follow the progress of your jobs using `qstat -u $USER`.

During the course we have set up a high-priority queue. This can be selected as follows:

```
qsub -q course1 ex1a_archer.pbs
```

This reservation is available only for the duration of this course.

Viewing PGM pictures

You can view the output PGM pictures using the `display` program from the Image Magick suite:

```
display output.pgm
```

Exercise 1: Hello World

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

Header files `shmem.fh` (Fortran) or `shmem.h` (C) must be included to make available the required SHMEM declarations and definitions.

(a) Use the provided `ex1a` template to aid you in writing a simple program where each PE, initialises shmem (using `shmem_init()`) prints out its index (using `shmem_my_pe()`) and a single master PE, e.g. PE number 0, prints out the total number of PEs (using `shmem_n_pes()`) before all PEs finalise (using `shmem_finalize()`). Compile and run the program on different numbers of PEs. Note that, despite convention, SHMEM PEs start from 0 rather than 1 even in Fortran.

(b) The template code `ex1b` 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 PEs is set in the code – we must therefore check at runtime that the actual number of PEs 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 PEs.

Extend the program to:

- Declare a second array `smallpic` to allow `bigpic(nx, ny)` to be shared across PEs.

  Fortran: use `smallpic(nx, nylocal)` and distribute over the y-dimension.

  C: use `smallpic(nxlocal, ny)` and distribute over the x-dimension.

  See the Appendix for more information on picture decomposition.

- Initialise `smallpic` on each PE to be equal to `shmem_my_pe()`.

- On the master, copy each piece of `smallpic` from the other PEs 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 arrays are initialised before they are copied back by the master. What happens if you omit this synchronisation?

(c) As an additional exercise, have each PE write its data to the correct part of `bigpic` on the master. You may need to introduce additional synchronisation points.

Exercise 2: Edge Detection

The aim of this exercise is to illustrate how parallel halo-swapping algorithms can be implemented using SHMEM functions. 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` 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 PEs, 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 PEs prior to edge detection, e.g. copy the upper halo from the upward neighbouring image etc.

  Fortran

  ```
  call shmem_double_get(oldpic(1,nylocal+1),oldpic(1,1),nxlocal,(mype+1))
  call shmem_double_get(oldpic(1,0),oldpic(1,nylocal),nxlocal,(mype-1))
  ```

  C

  ```
  shmem_double_get(&oldpic[nxlocal+1][1],&oldpic[1][1],nylocal,(mype+1));
  shmem_double_get(&oldpic[0][1],&oldpic[nxlocal][1],nylocal,(mype-1));
  ```

  You should ensure that you introduce appropriate synchronisation, and that the first and last PEs do not attempt to access non-existent neighbouring PEs.

- Run the code on multiple PEs and check that it works correctly. Note that incorrect synchronisation can lead to quite subtle errors so you should compare your result with the reference output in addition to performing a visual check.

  ```
  diff output.pgm reference.pgm
  ```

  Other than different whitespace in the initial header block, all the pixel values should agree.

(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 `edge360x240.pgm`, and you should be able to reconstruct the original picture as follows:

- define a new array 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 use `compare.sh` to inspect 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 PE, then the partial results summed over PEs. Write a subroutine or function that globally sums private double precision numbers by gathering them on a single PE, adding them together and then distributing the result across PEs.

(d) Common collective operations such as a global sum have optimised implementations provided by the SHMEM library. In this specific environment,

```
call shmem_real8_sum_to_all(target, source, nreduce, PE_start,
    logPE_stride, PE_size, pWrk, pSync)
```

can be used to perform the sum of double precision numbers in Fortran. In C, the:

```
void shmem_double_sum_to_all(double *target, double *source,
    size_t nreduce, int PE_start, int logPE_stride, int PE_size,
    double *pWrk, long *pSync)
```

function can be used.

Use a collective function to replace the global sum subroutine/function. Recall the definitions of pWrk and pSync (from the Cray SHMEM man pages):

```
pWrk      A symmetric work array. The pWrk argument must have the same
          data type as target.

          In C/C++, this contains max(nreduce/2+1,
          _SHMEM_REDUCE_MIN_WRKDATA_SIZE) elements.

          In Fortran, this contains max(nreduce/2+1,
          SHMEM_REDUCE_MIN_WRKDATA_SIZE) elements.

pSync     A symmetric work array.

          In C/C++, pSync is of type long and size
          _SHMEM_REDUCE_SYNC_SIZE.

          In Fortran, pSync is of type integer and size
          SHMEM_REDUCE_SYNC_SIZE. It must be a default integer value.

          Every element of this array must be initialized with the
          value _SHMEM_SYNC_VALUE (in C/C++) or SHMEM_SYNC_VALUE (in
          Fortran) before any of the PEs in the active set enter the
          reduction routine.
```

(e) Rather than using global synchronisation, try using the routines `shmem_fence()` and `shmem_quiet()` to use pairwise synchronisation between neighbours when swapping the halos.

(f) 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.

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.