|epcc|

PGAS Languages: Exercise Notes

UPC

Getting Started

Make sure your PATH points to the installation of the Berkley UPC compiler and the job launcher:

```
export PATH=/home/dsg/OPT/x86_64-unknown-linux/opt/bin:$PATH
```

Copy the archive UPC_practical.tar (which you will need for Exercise 2) into your home directory and unpack:

```
cp /tmp/UPC_practical.tar .
tar -xvf UPC_practical
```

In order to execute a UPC job, you need to use the upcrun job launcher script and specify the number of threads you want to use. In the following example, the executable myupc.exe is launched on 3 threads:

```
upcrun -n=3 ./myupc.exe
```

Exercise 1: Hello World

The aim of this exercise is to introduce you to compiling and running UPC code, to use the `MYTHREAD` and `THREADS` intrinsic and to use UPC syntax to perform inter-process communication.

Exercise

1. Write a UPC program where each UPC thread prints its index and the total number of available threads.
2. Write a program which

- declares a scalar shared array: `shared int x[THREADS];`
- initialises the value of `x` on the master thread to: `MYTHREAD`
- broadcasts the value of `x` from the master thread to each of the other threads
- and prints out the value of `x` on each thread.

Try rewriting this program so that after the master initialises its value of `x`, the other threads get `x` from the master, rather than the master copying it to each thread.

3. Modify the program so that each thread initialises its value of `x` to its value of `MYTHREAD`. Then have the master thread print the value of `x` on each remote thread.

Exercise 2: Parallel Sorting - Odd-Even Transposition sort

The aim of this exercise is to write a UPC program to sort a sequence of floating point numbers in ascending order, using the Odd-Even Transposition parallel sorting algorithm.

To sort an ***N*** element sequence in parallel on ***P*** processors (with **N>P**), each processor gets N/P elements of the sequence. The sequence is defined as sorted (in ascending order) when each local sequence is sorted in ascending order and each element on processor P_i is less than all elements on processor P_j, for all i<j.

Exercise

The code for this exercise is found in the `UPC_practical.tar` file. Unpack the archive by doing

```
tar xvf UPC_practical .tar
```

This will create the directory `OddEven_exercise`, which contains the skeleton code for this exercise. The template for the main part of the program is in the file `parallelSort.c`, with pointers on how to complete the exercise.

The exercise is divided into two parts. In the first part, the list to be sorted is distributed from the master thread, sorted sequentially by each thread and sent back to the master. The result is a partially sorted list, where elements that were on the same thread are sorted. In part two, the Odd-Even sorting algorithm is implemented to sort the whole list in ascending order.

Part 1

1. The master thread generates the list to be sorted using the `initialiseArray( masterList, totalListSize)` routine.

   The `totalListSize`, `numThreads` and `localListSize` variables are declared as defined values in the `parallelSort.h` header file. Change the value of `totalListSize` to change the size of the array to sort and change `numThreads` to match the number of processes that you are using.

   (Choose `totalListSize` so that it divides evenly by `THREADS`).
2. Declare a `workingList` shared array with `localListSize` elements per thread. Copy the list from the master thread to the `workingList` shared array. Synchronisation is needed after this step to ensure that each thread has received its portion of the list.
3. Each thread sorts its local array using sequential quick sort. The `quickSortDriver(A, n)` routine is provided. This sorts the array `A` with `n` elements.
4. After each thread has sorted its local array, gather the locally sorted data to the master and print the list. The elements of the list that were on the same thread should now be sorted. Synchronisation is needed before gathering to ensure that all threads have finished sorting their local array before the master gets it.

Part 2

Now the Odd-Even transposition sort algorithm is applied.

Algorithm

The algorithm, as shown in Figure 1, starts with `localListSize` locally sorted elements on each thread (as we have from Part 1). It then works by alternating between odd and even phases:

- during an *odd* phase, each even-numbered thread exchanges its data with its left-numbered neighbour and odd-numbered threads exchange with their right neighbour.
- during an *even* phase, even-numbered threads exchange data with their right neighbour and, odd-numbered threads with their left neighbour.

Each thread then compares the received data to its own, keeping the smallest/biggest `localListSize` elements depending on if `MYTHREAD` is less/greater than the index of the

thread it exchanges data with. After `THREADS` iterations of odd-even exchanges, the sequence is sorted.

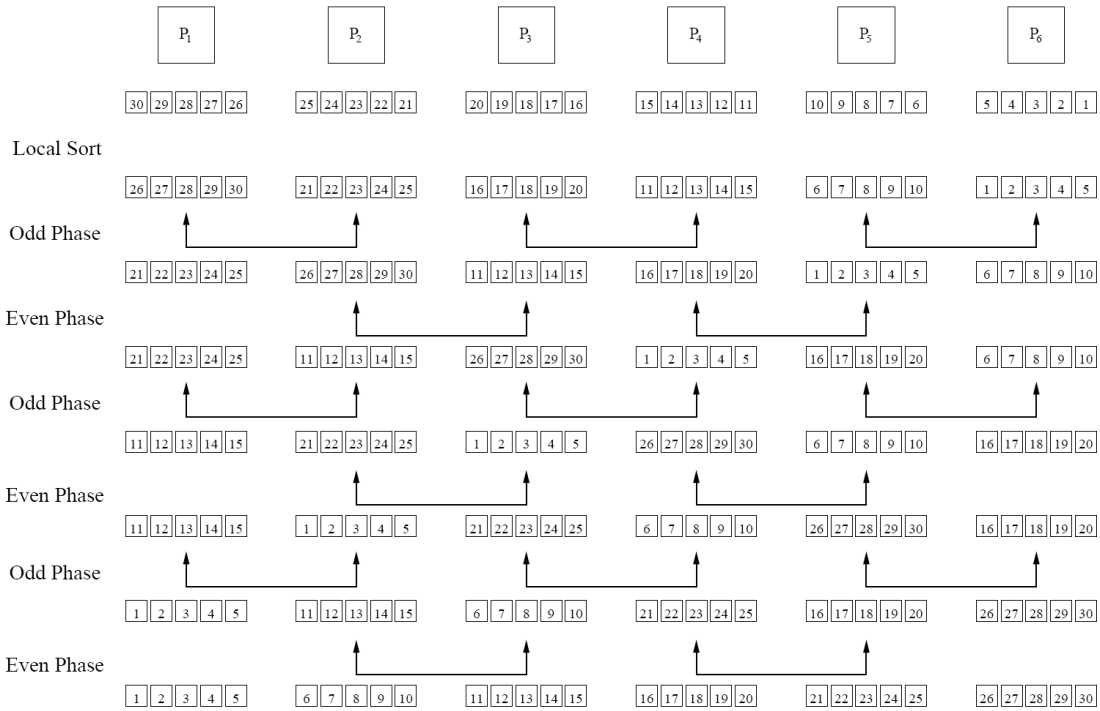

Figure 1: Odd-Even Transposition Sort algorithm

Code

To implement this algorithm, add code (after the sequential sort of each local list) which:

1. works out the indices of the threads to communicate with during the odd and even phases of the algorithm:
   - ```
     if (MYTHREAD%2 == 0){
        oddPhaseNeighbour = MYTHREAD - 1
        evenPhaseNeighbour = MYTHREAD + 1
     }
     ```
   - ```
     if (MYTHREAD%2 != 0){
        oddPhaseNeighbour = MYTHREAD + 1
        evenPhaseNeighbour = MYTHREAD - 1
     }
     ```
2. starts the Odd-Even phase loop. Loop `THREADS` times. Depending on whether the iteration is odd or even, threads get the `workingList` array from their odd-phase neighbour or even-phase neighbour and store it in the `neighbourList` array.
3. calls the provided `compareSplit` routine to compare the received values with its own data. The `compareSplit(n, workingList, neighbourList,`

`smaller)` routine compares the elements of `workingList` with those of `neighbourList`, both of size `n`, keeping the values which are smaller or bigger depending on if `MYTHREAD` is smaller or greater (`smaller` works as a flag) than the value of neighbour.

4. gathers the sorted data to the master and prints the sorted list, as in Step 1.

NOTE:

- Synchronisation statements are needed throughout the code to ensure that data has been sent/received before it is altered/used.
- Control statements will be needed to ensure that only threads with valid values of `oddPhaseNeighbour` and `evenPhaseNeighbour` will communicate and call `compareSplit`.