MP1
CS 598: Parallel Migratable Objects

Fall 2014

Due date: Thursday, September 25, 10 PM CDT

Description: Parallel prefix can be used to implement a load balancing strategy. In this assignment, you will implement a load balancer using parallel prefix. You can implement the parallel prefix program given in the class slides or you can implement your own version.

To implement the parallel prefix program, create a chare array of size s. Each chare array element, i, generates n_i random integers where the value of n_i is randomly generated between min and max. The set of numbers generated can be stored in an array or stl vector which we will refer to as $elems$. Your program takes min, max, and s as the command line arguments. Use $rand()$ function to generate random numbers. This can create load imbalance depending on the value of n_i, which is the number of $elems$ the i^{th} chare array element owns. You will perform manual load balancing using parallel prefix so as to distribute the $elems$ such that each chare array element owns approximately $\frac{n}{s}$ $elems$ where $n = n_1 + n_2 \cdots n_s$. **Note that the goal is not sorting or finding the parallel prefix of the integers, it is just load balancing the integers that each chare owns while keeping the order of the integers the same globally.**

The prefix sum and the average number of integer elements per chare array element are used to determine which integer elements each chare array element needs to send. To obtain the average number of integers per chare array element, use reductions. See Charm++ reductions manual section for details on reductions.

Figure 1 gives an example of how load balance can be achieved.

Detailed Algorithm:

1. Create a chare array of size s and generate the random integers on each chare array element. Chare array element i contains n_i random integers where n_i is a random number between min and max. min, max and s

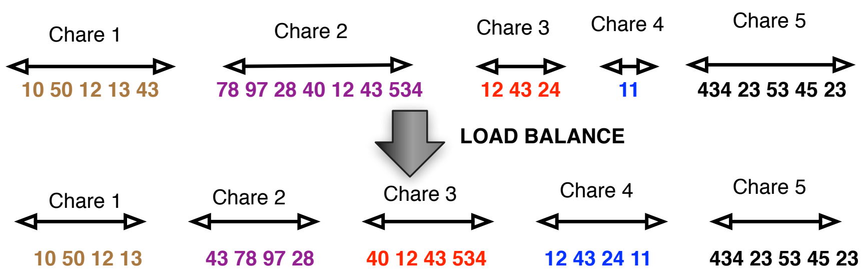


Figure 1: Load balancing integers among chares.

are input arguments to your program.
Here is the pseudocode for this.

```
$n_i = rand()\%(max - min) + min;$
$for(int\ k = 0;\ k < n_i;\ k++)\ elems[k] = rand();$
```

2. Execute the parallel prefix code on the number of elements in a chare array element, n_i, to obtain the prefix sum. Note that prefix sum is not done on *elems* but on n_i. Let p_i be the prefix sum of the chare array element at index i where $p_i = \sum_{k=0}^{i-1} n_k$.

3. Find the average number of elements, *avg*, either using a reduction or using the prefix sum.

4. Once you have the prefix sum, p_i, and *avg*, the data exchange required to achieve load balance can be done as follows:

   - consider a logical ranking of all the elements 0 to $(n-1)$.

   - in a balanced distribution, elements with ranks $avg*i$ to $avg*(i+1)-1$ will reside on chare i

   - Every chare array element identifies the rank for *elems*, which is the integer elements it owns. For chare array element i, the ranks of the integers in *elems* would range from p_i to $p_i + (n_i - 1)$. The integer element with rank j should be sent to chare array element with index $\lfloor (j/avg) \rfloor$. Send these elements to the correct destination using messages. Also send the starting rank so that the destination can place these elements in the right location in the array. Do not send the elements in separate messages. Instead bundle them together so that all the elements from a chare array element that need to be sent to a destination are sent in a single message.

5. Upon receiving a message, every chare array element copies these new elements into a new list in their correct position (using the rank provided by the sender). Note, we are simply load balancing the number of integers

contained by each chare array element and the original overall order of the integers should be preserved.

Correctness Test: The checksum of the integers in the beginning should be the same as the checksum of the integers after load balancing. Use the *bitvec_xor* operation over all the integers to compute a checksum (see the Charm++ reductions section in the manual).

Exit condition: You can either use Quiescence Detection (with the CkStartQD(..) function, for more information about QD please see the related manual section) to terminate or your own method to detect when the load balancing is complete.

Running your code: The program should have the following signature:
./charmrun ./mp1 +p4 chare-array-size min max

You should test your code on Taub cluster up to 64 processors. You can start with a small number of processors and once you make sure your algorithm works, you can test with more number of processors. You can use 1024 as *chare-array-size* and *min/max* in the range of [5,000 - 30,000].

A Case Study:

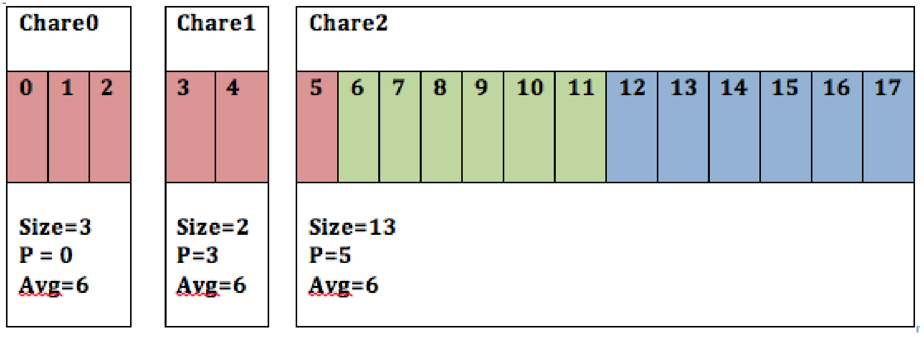


Figure 2: Before load balancing.

After load balancing, all the reds will be in Chare 0, all greens in Chare 1, all blues in Chare2.
Chare 0 will not send anything.
Chare 1 will send everything [3-4] to Chare 0.
Chare 2 will send [5] to Chare 0 and [6-11] to Chare 2.

P = Exclusive Prefix Sum

Fact #1: If Chare n has k integers, the global rank of the integers are $[P, P + k - 1]$.

Fact #2: After the load balance, Chare n needs to have values in the range $[thisIndex() * avg, (thisIndex() + 1) * avg - 1]$

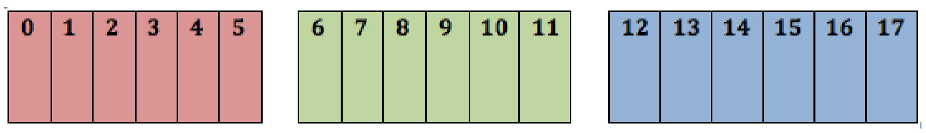


Figure 3: After load balancing.

Load Balance Algorithm:

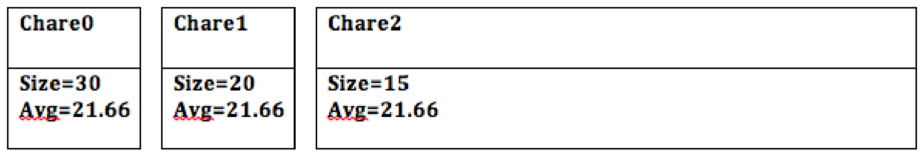
- Decide which part of your array belongs to which chare by looking at the global rank of your local array.
  - E.g: For Chare 3, local integers have global rank of [5-17]
  - [5] should be in Chare 0 and [6-11] should be in Chare 1 because of FACT #2.
- Extract values from local array and send it to the chare together with the index information.
  - int[] values, int *start_index*, int *end_index*
- Receiving side: The final array can be pre-allocated according to the average. Whenever a chare receives data, put into the pre-allocated place looking at the index information

Note: Here is a suggested way for the case where the total number of integers is not evenly divisible by s (size of the chare array).

- Store the average as double, not as integer.
- When calculating the index range $[thisIndex() * avg, (thisIndex() + 1) * avg - 1]$, the values needs to be rounded down.

Another example showing global index values:

Global index values: $[thisIndex() * avg, (thisIndex() + 1) * avg - 1]$
Chare 0: [0-20] $- >$ 21 values after balance
Chare 1: [21-42] $- >$ 22 values after balance
Chare 2: [43-64] $- >$ 22 values after balance



Submission: Create an mp1 directory in your SVN repository folder and add your code into that folder and check in the your code.

- For each file F you create, that you want to check in, do:
  svn add F
  and frequently (after you have modified F, and have the next better version) do:
  svn ci F
- There will be a penalty for late submissions.