# **Parallel Computing Assignment 3:**

Hrishikesh Deshpande
hdeshpa@purdue.edu

**NOTE**: for questions 1-3 the machine mc17.cs.purdue.edu was used for evaluation with run time =120 seconds.

**Question 1.**

In question 1, process 0 acts like a broker. Processes with odd rank are producers while those with even rank are considered consumers. After 120 seconds, the final consumed count is collected at process 0.

Command to run: mpirun -n <4/8/16> a.out

| No of processes | Consumed Count |
|---|---|
| 4 | 31028366 |
| 8 | 44287502 |
| 16 | 49653277 |

As we can see, with an increase in the no of processes, we see a corresponding increase in the consumed count. Since a single processor acts like a broker, we get increased overhead at processor 0 which needs to handle multiple requests from different clients.

**Question 2.**

In question 2, each process acts as producer as well as the consumer. A process sends a message to a random processor and then waits for an ACK. At the same time, the process checks whether another process has sent it a message. If yes, it responds with an ACK. Towards the end of 120 seconds, the results are available at Process 0.

Command to run: mpirun -n <4/8/16> a.out

| No of processes | Consumed Count |
|---|---|
| 4 | 31398718 |
| 8 | 40726347 |
| 16 | 42609741 |

As seen from the above table, as we increase the process count, we see a corresponding increase in the number of items that have been consumed.

**Question 3.**

In question 3, we map 2 processes to each core. One process acts as a producer and the other process acts like a consumer. At the end of 120 seconds, process with rank 0 gets the final consumed count.

Command to run:

mpiexec -np 4 -H localhost -rf rankfile_mc17 ./a.out (this used 2 cores and 2 processes/core)

NOTE: we use rankfile_mc17 to run the command on mc17 while rankfile_mc18 needs to be used to run it on MC 18. This is because these systems have different socket to core mappings.

| **No of Cores** | **No of processes** | **Consumed Count** |
|---|---|---|
| 2 | 4 | 4008210 |
| 4 | 8 | 10702110 |
| 8 | 16 | 11340426 |
| 16 | 32 | 16775810 |

We use a static rankfile to bind 2 processes per core and thus a different rankfile might be needed based on the organization of cores and sockets in the SPMD/MPMD platforms.

As seen from the results above, as the no of processes increase we see a sustained increase in consumed count since we now have more producers and consumers running at the same time.

**Q4. All-to-k reduction**

In this problem, every processor has a vector of m entries. The target is to reduce these entries and make them available to a group of k processors.

Now the question does not say whether the values available to the group of k processors should be same or different.

**Solution 1: (all-all reduce(k)) if values are different (based on Piazza discussion I vouch for solution 1. Since question isn't clear I'm giving an alternate solution 2 in case it's all-reduce)**

In all-all reduction, all processors end up having unique values. So, if all-k reduction is modeled as a special case of all-all reduction then the target k processes end up having different values.

With that assumption, let us compute the runtime of all-k reduce.

Let's assume that each process starts with a vector of m entries which it needs to distributed to only k processors. So at the end of this operation, each of the k processors should get these k entries. This is the **reverse of k-to all broadcast.**

In case of k to all broadcast, the message size doubles in each of the first log(k) iterations and then remains mk in the remaining log(p/k) iterations. The total communication time of first k iterations is $t_s$logk + $t_w$m(k-1)

The total communication time of the last log(p/k) iterations is ($t_s$ + $t_w$mk)log(p/k).

Thus the entire operation is performed in $t_s$logp + $t_w$m(k log(p/k) + k -1) time.

Similarly, for all-k reduce we just do the reverse i.e. first collect messages in each of the k processes so that each process in this subset now have messages of size mk.

Now do a reduce wherein the message size halves in each of the log(p/k) iterations.

This gives us an effective run time of $t_s$logp + $t_w$m(k log(p/k) + k -1).

This has an effective runtime of O(mklog(p/k)) which the best possible runtime that we can achieve.

**Solution 2: all-reduce(k) → k-reduce**

In all-reduce all processors end up having the same value. From class discussion, all-reduce can be done in the same way as all-all broadcast expect that the message size does not double in each step. So, each node starts with a buffer of size m and final results are identical buffers of size m formed by combining the original p buffers using an associative operator. So if all-k reduction is modeled using all-reduce then all k processors end up having same values.

Thus messages are added instead of being concatenated and takes $(t_s + t_w m)\log p$.

Now, with all-k reduce we want the final result to appear at k of the processors (out of p).

This is equivalent to an all-one reduce followed by a one-k broadcast.

$(t_s + t_w m)\log p$ → for all to one reduce

$(t_s + t_w m)\log k$ → for one-k broadcast

So the overall runtime is $(t_s + t_w m) \log p)$ → $O(m \log p)$ which is the most optimal solution possible for all-reduce.

**Q5. k-to-all Scatter**

In this problem, initially k processors have a vector, each of p elements. At the end of the operation, each processor should get the k entries originating at each of the k processors.

This needs to be done in 2 steps:

1. First, we need to do an all-to-all scatter for k processes that have each of these p elements. These k processors share p/k amount of data in pairs. Assuming that this communication happens on a hypercube, in each iteration the k processes exchange data with a partner (computed as my_id xor j) where j = 1 to k – 1.
   E-cube routing is used to ensure that there is no congestion.

   Now, the run time complexity for all-all scatter with k processes is:
   $(t_s + t_w p/k)(k-1) = t_s(k-1) + t_w p(1-1/k)$
2. Second, we now have each of these k processors have the data for the remaining (p-k) processes which can communicate to the other (p-k) processes in log(p/k) no of steps using a technique like one-all scatter along each of these subcubes. (remember one-to-all scatter requires $t_s \log p + t_w m (p-1)$)
   Since each of the k processors in each subcube already have the messages, it needs to be scattered among the remaining processes of the subcubes.
   This can be achieved in $t_s \log(p/k) + t_w(p-k)$ [since we only perform log(p/k) iterations)
   [1, ½, …… k/p]

   Last transmission will transfer k entries between each pair of processors.

   **Total time:**
   $T = t_s(k - 1 + \log p/k) + t_w p(1-1/k) + t_w p(1-k/p)$

$= t_s(k - 1 + \log p/k) + t_w\, p(2 - 1/k - k/p)$
$= t_s(k - 1 + \log p/k) + t_w(2p - k - p/k)$

The computation assumes that message size m = 1. (else just need to add m to the above formula and re-compute). This is O(p) which is optimal since fan in and fan out are each are each p.