Csc-387 Parallel Programming with MPI
Instructor : Fikret Ercal

Programming Project 3
PARALLEL BITONIC SORT

Write a program to sort N arbitrary numbers in parallel on P processors (N $\gg$ P) using a *bitonic sort* algorithm as explained in the notes and this handout.

I. (25 pts) Run your program on MST Cluster and obtain timing results for N= 8M, 32M, 64M, and 128M and P=1, 2, 4, 8, 16, and 32 (a total of 24 timing results). Use 4-byte integers (randomly generated) as the numbers to sort. Tabulate your findings.

II. (20 pts) Run your program for N=400K using the input file at *http://web.mst.edu/~ercal/387/input.400K* and print the following for the sorted list:
(i) 10 numbers starting at index (position) 100,000 and
(ii) 10 numbers starting at index (position) 200,000.

III. (15 pts) Try to obtain an approximate formula for the *speedup* and *efficiency* in terms of N and P. Comment briefly on the asymptotic behaviour of your formula for the following cases:

i) N = P   ii) N $\gg$ P

IV. (30 pts) Answer the following questions with respect to your timing measurements:

1. How does the speedup change when you increase N for a fixed P ? Is your answer consistent with the formula you derived in Part III ? Why or why not ?
2. How does the efficiency change when you increase N for a fixed P ? Is your answer consistent with the formula you derived in Part III ? Why or why not ?
3. How does the efficiency change when you increase P for a fixed N ? Is your answer consistent with the formula you derived in Part III ? Why or why not ?

V. (5 pts) Electronic copy of your program. Your program must: (i) contain a program overview/summary, your id/name etc. in the header, (ii) execute correctly, (iii) be well documented (Must include comments before every major statement, function/subroutine calls, and every MPI call explaining what the call is for. Must include specs for every function/subroutine)

VI. (5 pts)
Read the file “How to submit projects?” on the class website and follow all the instructions in there. You must store the requested files in your subdirectories.

BONUS: The best running time for P=16 and N= 128 Million will get bonus points as described in the syllabus. Therefore, do not forget to report your running time for P=16 and N= 128 Million.

IMPORTANT NOTE:
1) Time only the sorting part of your program. Do not time the portion where the random numbers are generated.

2) To be more accurate, take the average of 4-5 runs for the same program and report it as one timing result.

PARALLEL BITONIC SORT FOR $N >> P$

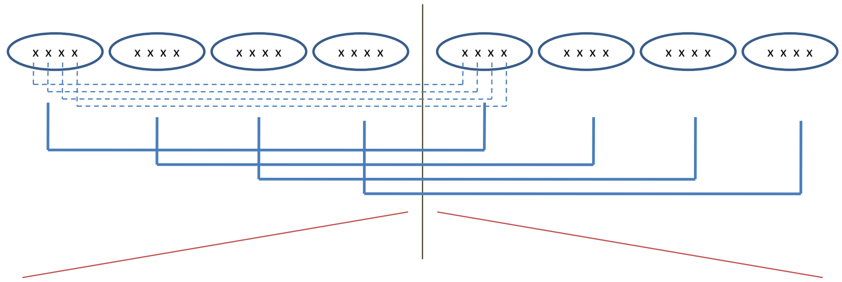

Figure 1

1. Sorting a bitonic sequence of length N sequentially, using the bitonic sort algorithm, takes $T_{seq} = O(NlogN)$ steps.

2. Sorting a bitonic sequence of length N sequentially, using a "merge sort" algorithm, takes $T_{seq} = O(N)$ steps.

3. Let's consider sorting a bitonic sequence in parallel when $N >> P$. In this case, each PE gets N/P elements and emulates N/P virtual processors. In other words, they compare/exchange N/P elements each time as shown in Fig.1. After $\log P$ steps, each PE will end up with a local bitonic sequence of size N/P to sort. Therefore, the parallel time T_p can be written as a composition of two terms; first one to represent the time taken during the compare/exchange interactions between the processors, and the second one to represent the time to sort sequentially a bitonic sequence of size N/P in each processor. If we use the "merge-sort" algorithm for sorting the final bitonic subsequences, then we get the following:

$$\begin{aligned} T_p &= \frac{N}{P}logP + \frac{N}{P} \\ &= \frac{N}{P}logP \end{aligned}$$

4. Let's now consider sorting an arbitrary sequence using bitonic merge when $N >> P$. Each PE gets N/P elements and sorts them locally in O(N/P log(N/P)) time. Processors then run the original bitonic sort algorithm in parallel using $\log P$ major steps; each time doubling the size of the bitonic sequences; N/P 2N/P, 4N/P, 8N/P, etc. From step 3, we know that the time to sort a bitonic sequence is $\frac{N}{P}logP$. Therefore, the total time complexity can be computed as follows by summing the times taken by all steps:

Let

$$\begin{aligned} N_i &= \frac{N}{P}2^i \\ P_i &= 2^i \quad where \quad i = 1, 2 \ldots logP \end{aligned}$$

We have

$$\begin{aligned} T_p = \frac{N}{P}log(N/P) + \sum_i \frac{N_i}{P_i}logP_i &= \frac{N}{P}log(N/P) + \sum_i \frac{\frac{N}{P}2^i}{2^i}log2^i \quad where \quad i = 1, 2 \ldots logP \\ &= \frac{N}{P}log(N/P) + \frac{N}{P}\sum_1^{logP} i \\ &= \frac{N}{P}(log(N/P) + \frac{1}{2}(1 + logP)(logP)) \end{aligned}$$

The time complexity is $O(\frac{N}{P}(log(N/P) + log^2 P))$

Parallel Bitonic Sort Algorithm

The input data need to be partitioned and distributed to the processors first, and then a standard sequential sorting method (e.g., Quicksort) must be used to locally sort the data. Once the processors complete local sorting, they communicate and exchange data in a certain fashion to merge the local subsequences. **Bitonic Sort** uses a predetermined order to merge the locally sorted sequences and to obtain a totally sorted sequence, as depicted in Figure 2. The first $(d^2 - d)/2$ steps[1] are for obtaining a full bitonic sequence and the last d steps are for sorting this Bitonic sequence.

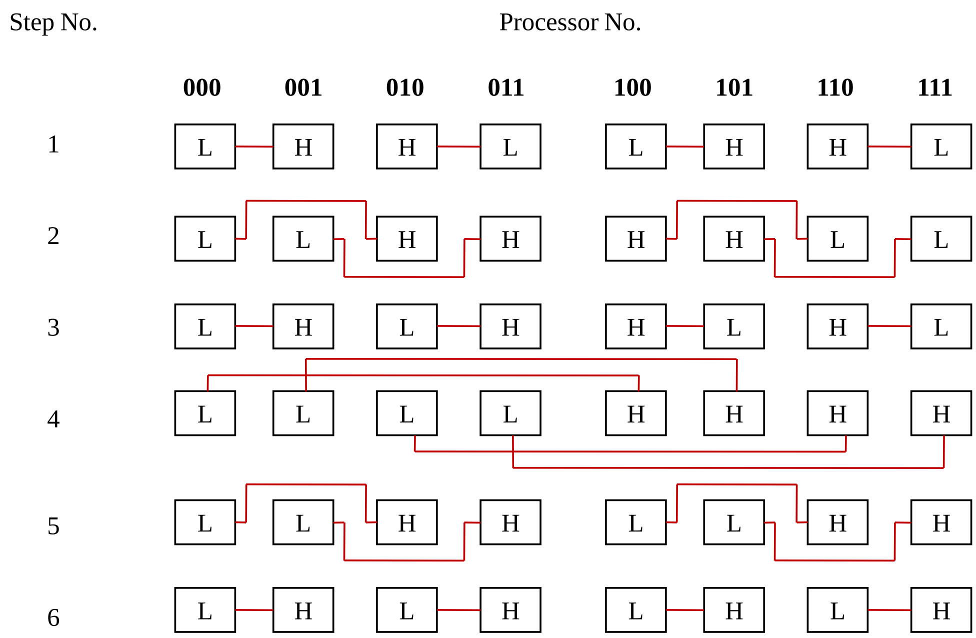

Figure 2: Six phases of Bitonic Sort on a hypercube of dimension 3

There are two main routines, called $CompareHigh$ and $CompareLow$, in the algorithm for Bitonic Sort given below. During merging, some processors call the first routine while their neighbors call the second one, so that the lower and the higher subsequences of a sequence are split apart between the two processors. The lower part is gathered on the processor which calls $CompareLow$ and the upper part on the processor which calls $CompareHigh$. A $CompareLow(j)$ or $CompareHigh(j)$ means that a processor will compare and exchange with a neighbor whose (d-bit binary) processor number differs only at the j^{th} bit.

```
Parallel Bitonic Sort Algorithm for processor P_k (for k := 0 ... P − 1)
d:= log P      /* cube dimension */
sort(local − data_k)     /* sequential sort */
/* Bitonic Sort follows */
for i:=1 to d do
    window-id = Most Significant (d-i) bits of P_k
    for j:=(i-1) down to 0 do
        if((window-id is even AND j^th bit of P_k = 0)
           OR (window-id is odd AND j^th bit of P_k = 1))
          then   call  CompareLow(j)
          else   call  CompareHigh(j)
        endif
    endfor
endfor
```

[1] d is the cube dimension and d = $\log_2 P$ where P is the number of processors.