Homework 3 of CS7344

8.11

Write a function to transpose an $n \times n$ matrix A. Assume that before the function call, A is rowwise block decomposed among the p process. After the function returns, A should be columnwise block decomposed among the p process.

Answer: Assume n is a multiple of p, then each process hold $\frac{n}{p}$ rows of A. To transpose the matrix, each process need to exchange its data with all other processes. Algorithm 1 shows the transposing process.

Algorithm 1 Transpose Matrix

1: **Input:** $n \times n$ matrix A
2: **Output:** Transposed Matrix
3: p ← MPI_size()
4: rank ← MPI_rank()
5: block ← n/p
6: localRows = A[block*rank:block*(rank+1),:]
7: localColumns = A[block*rank:block*(rank+1),:]
8: for r from 0 to p-1
9: send localRows[:,r*block:(r+1)*block] to process r and receive the corresponding data from r, then transpose the received data and save it to localColumns[:,r*block:(r+1)*block]

In Algorithm 1, the terms `localRows` and `localColumns` refer to the matrix rows held by the current process and the columns after the matrix has been transposed, respectively. `Lines 8 and 9` illustrate the data exchange mechanism between the current process and all other processes. Specifically, for processes $p1$ and $p2$, $p1$ sends $\frac{n}{p}$ columns, ranging from $\frac{n}{p} \times p2$ to $\frac{n}{p} \times (p2+1)$, to $p2$ within `localRows`. Conversely, $p2$ sends $\frac{n}{p}$ columns, spanning from $\frac{n}{p} \times p1$ to $\frac{n}{p} \times (p1+1)$, to $p1$. Upon receiving this data, $p1$ transposes it and places it into `localColumns` at the corresponding indices as it was sent. This process is facilitated using the `MPI_Sendrecv()` API. To handle noncontiguous data efficiently, we employ `MPI_Type_vector()` combined with `MPI_Type_create_resized()` to redefine the datatype. This approach ensures that the received data is stored correctly in `localColumns` without necessitating an additional transposition operation.

After implementing Algorithm 1 using `MPI` on a desktop equipped with 6 CPU cores and 16 GB of memory, with n set to 8192, we observe the performance results as illustrated in Figure 1. The data shows that when the number of processes is fewer than 8, performance improves as the number of processes increases. Optimal performance is achieved when the number of processes is 8. However, when the number of processes exceeds 8, performance deteriorates due to constraints in hardware resources. This degradation likely results from the overhead associated with managing more processes than there are physical CPU cores, leading to inefficient context switching and resource contention.

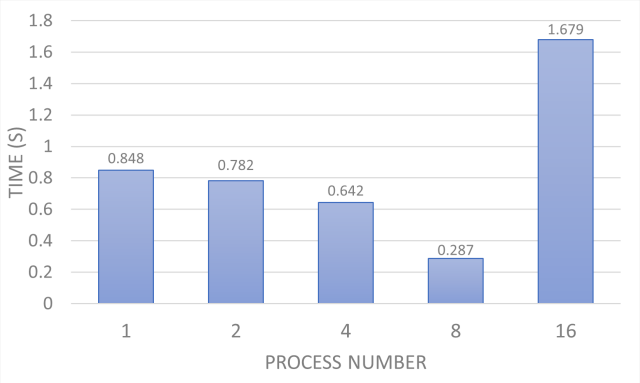

Figure 1: The Cost Time(s) of the Transpose Matrix Algorithm (8.11) under different number of processes.

9.8

Write a manager/worker-style parallel program that finds the smallest positive root of the equation

$$f(x) = -2 + \sin x + \sin x^2 + \sin x^3 + \cdots + \sin x^{100}$$

This root is the unique value r between 0 and 1 such that $f(r) = 0$. The program should divide the interval [0,1] into several subintervals and create as set of tasks, one for each subinterval. Each task computes the value of $f(x)$ at both ends of its subinterval. One task will find the subinterval where the function changes from negative to positive. The algorithm can then iterate, dividing this subinterval into pieces. When the subinterval size becomes less than 10^{-11}, the program should terminate, printing the root of the function.

Answer: Assume there is a manager and n workers, then the manager divides the interval into n subintervals and then assign each subinterval to a worker. The worker does the computation and return the result to the manager. When manger receive the result, then it update the interval where root exists and divides the new interval and assgin tasks to workers. With the process, when the manager find the interval is less than 10^{-11}, it terminate and print the root. Algorithm 2 shows our design.

In Algorithm 2, the number of workers must bigger than 1, so the process number must bigger than 2. And in `Line 12`, we use `MPI_Abort()` api to terminate all worker processes.

After implementing Algorithm 2 using `MPI` on a desktop with 6 CPU cores and 16 GB of memory, we analyzed the performance results shown in Figure 2. The results indicate that the speedup

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Algorithm 2 Find Root

```
Input: f(x)
Output: root
p ← MPI_size()
rank ← MPI_rank()
num_workers = p-1
if(rank==0) //manager process
    set interval = [0,1], eps = 10^-11
    while(interval[1]-interval[0]>eps)
        split interval into num_workers subinterval
        assign each subinterval to corresponding worker
        receive the subinterval that has root from the worker and update the interval with subinterval
    terminate all workers and print root as the average of interval[0] and interval[1]
if(rank>0) //worker process
    while(1)
        receive the interval from manager process
        compute f(interval[0]) and f(interval[1]).
        if f(interval[0]) * f(interval[1]) < 0, send the interval back to manager
```

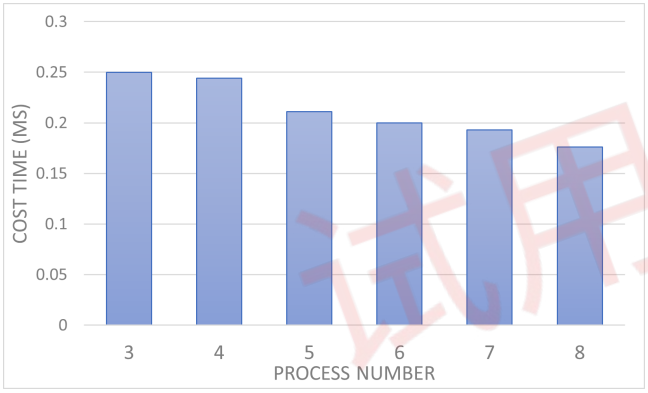

Figure 2: The Cost Time(ms) of the Find Root Algorithm (9.8) under different number of processes.

increases slowly as the number of processes increases. This modest improvement in speedup could be attributed to the relatively small computation time required to find the root, such that the timing measurements introduce significant variability. In scenarios where the computational load is light, the overhead associated with process management and communication in MPI can overshadow the benefits of parallelization, leading to minimal gains in speedup as more processes are added. And the root we find is 0.674527953966.

11.4

Consider the optimization of overlapping communication steps with computation steps in the two parallel matrix multiplication algorithms discussed in this chapter. Suppose $p = 16, \beta = 1.5 \times 10^6$/sec, $\lambda = 250\mu$sec, and $\chi = 10$ nanosec.

a For what values of n can we expect the communication time per iteration of the rowwise algorithm to be less than the computation.
Answer: With the rowwise algorithm, we have following nonequality:

$$\frac{\chi n^3}{p^2} > \lambda + \frac{n^2}{p\beta}$$

Then we have

$$\frac{10 \times 10^{-9} \times n^3}{16^2} > 250 \times 10^{-6} + \frac{n^2}{16 \times 1.5 \times 10^6}$$
$$n > 1072$$

Because n is a multiple of p, the minimum value of n should be 1088.

b For what values of n can we expect the communication time per iteration of Cannon's algorithm to be less than the computation time.
Answer: With the Cannon's algorithm, we have following nonequality:

$$\frac{\chi n^3}{p^{3/2}} > 2(\lambda + \frac{n^2}{p\beta})$$

Then we have

$$\frac{10 \times 10^{-9} \times n^3}{16^{3/2}} > 2(250 \times 10^{-6} + \frac{n^2}{16 \times 1.5 \times 10^6})$$
$$n > 544$$

Because n is a multiple of $\sqrt{p}$, the minimum value of n should be 548.

14.9

This chapter has found focused on parallel implementation of quicksort, $O(n \log n)$ sorting algorithm. Another well-known, recursive, $O(n \log n)$ sorting algorithm is mergesort. Here is pseudocode for mergesort. It relies on function Merge, which merges two sorted lists.

```
Mergesort(list):
    if length(list)=1 then
        return list
    else
        part1 = Mergesort(first half of list)
        part2 = Mergesort(remainder of list)
        return Merge(part1, part2)
    endif
```

a Design a parallel version of mergesort for a multicomputer. Make these three assumptions. At the begining of the algorithm's execution all unsorted values are in the memory of one processor. At the end of the algorithm's execution the sorted list is in the memory of one processor. Then number of processor p is power of 2.
Answer: We can partition the array into p segments, with each segment sorted independently using a standard mergesort algorithm by separate processes. Subsequently, we employ a tree-based reduction method to merge these sorted segments. Specifically, during each iteration, pairs of processes merge their respective segments through send and receive operations. This merging process requires a total of

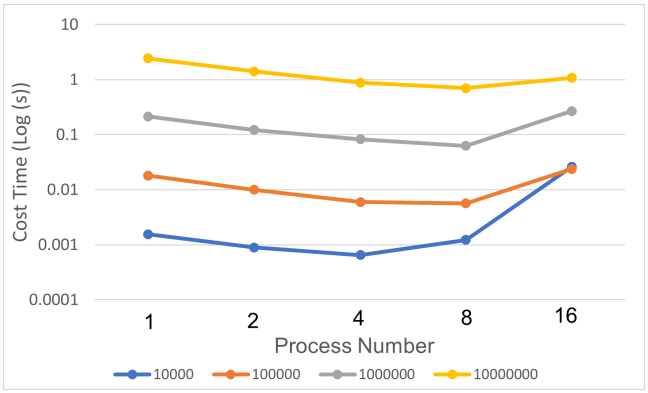

Figure 3: The Cost Time(Log (s)) of the Parallel Mergesort Algorithm (9.8) under different number of processes and different size.

Algorithm 3 Parallel MergeSort

```
1: Input: unsorted array A with size n
2: Output: sorted array
3: p ← MPI_size()
4: rank ← MPI_rank()
5: if(rank==0)
6:     divide A into p parts, each parts has either ⌊n/p⌋ or ⌈n/p⌉ elements.
7:     send p-1 parts to other p-1 process
8:     use normal mergesort to sort the first part
9: else
10:    receive a part of A from process 0
11:    use normal mergesort to sort the part
12: step ← 1
13: while(step<p)
14:    if(rank%(2*step)==0)
15:        receive a part of A from rank+step process and merge the received sorted part with its own sorted part to form a new bigger sorted part of A
16:    else
17:        send its own sorted part to rank-step process
18:        break
19:    step ← 2*step
20: if(rank==0)
21:    print out the sorted array of A
```

$\log p$ iterations to complete. Algorithm 3 outlines the steps involved in this Parallel MergeSort procedure.

b What is the complexity of this algorithm?
Answer: For the computation, we consider two primary components. The first part is where each process sorts its local segment of the array using the standard mergesort algorithm. The complexity of this part is $O\left(\frac{n}{p}\log\frac{n}{p}\right)$. The second component involves the tree-reduction merge process. Given that there are $\log p$ steps, and during step i, $2^i\frac{n}{p}$ elements are merged, the complexity for this process sums to $\sum_{i=1}^{\log p} 2^i\frac{n}{p} = 2(p-1)\frac{n}{p} = O(2n)$. Therefore, the total computational complexity is $O\left(\frac{n}{p}\log\frac{n}{p}+2n\right)$.
For the communication complexity, we include both the initial sending process and the tree-reduction merge process. Initially, process 0 sends n elements. During the tree-reduction merge, which has $\log p$ steps, step i involves sending $2^{i-1}\frac{n}{p}$ elements. The complexity for the communication during the tree-reduction sums to $\sum_{i=1}^{\log p} 2^{i-1}\frac{n}{p} = (p-1)\frac{n}{p} = O(n)$. Thus, the overall complexity of communication is $O(2n)$.
Combining both aspects, the total complexity is $O\left(\frac{n}{p}\log\frac{n}{p}+4n\right)$. Given that n is significantly larger than p, the complexity simplifies to $O\left(\frac{n}{p}\log n+n\right)$.

c What is the isoefficiency of your parallel mergesort algorithm?
Answer: Because the complexity of serial algorithm is $O(n\log n)$, and the communication cost of our designed algorithm is $O(n)$, the isoefficiency is

$$n\log n \geq Cpn$$
$$n \geq 2^{Cp}$$

d Write a program implementing your parallel algorithm. benchmark the program for various combinations of p and n.
Answer: After implementing Algorithm 3 using MPI on a desktop equipped with 6 CPU cores and 16 GB of memory, we assessed the performance as depicted in Figure 3. For this evaluation, we set the y-axis to a logarithmic scale to enhance clarity. The analysis of the results for various combinations of p (number of processes) and n (array size) reveals some notable trends. When p is 4 or fewer, the speedup approximately doubles as the number of processes is doubled for all considered values of n. Conversely, when n is relatively small (10,000 and 100,000) and p exceeds 4, the anticipated speedup diminishes. Thus, to achieve a noticeable speedup, a larger array size n is required. This suggests that the overhead associated with managing a larger number of processes outweighs the benefits of parallel processing for smaller data sizes.