MPI Parallel Merge Sort

Feb 27, 2014

Joe Houng

Abstract

Merge Sort is a very well known sorting algorithm traditionally seen implemented and taught in a serial manner, with one core. However the nature of this algorithm also makes it very implementable in a multi-core system. How does the time and efficiency compare between different number of cores as the size of array rises?

Introduction

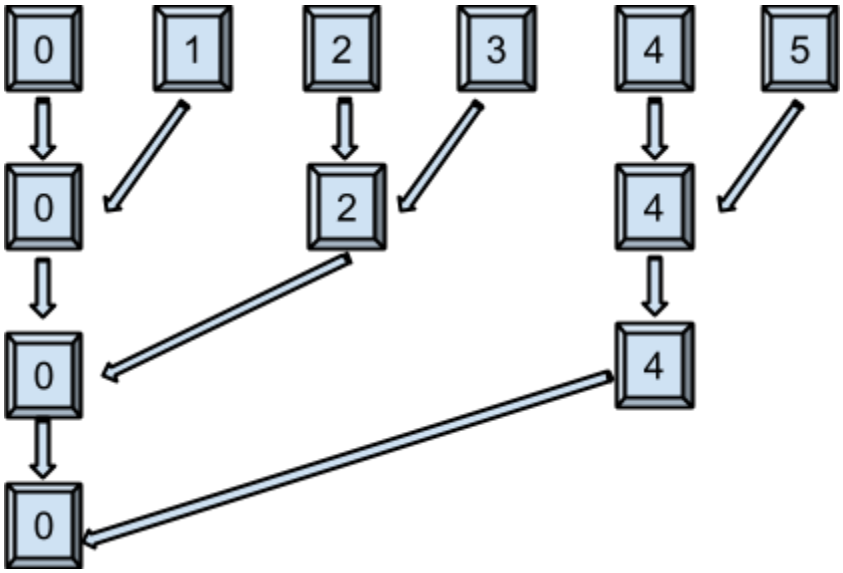
Problem 3.8 from “An Introduction to Parallel Programming”

Parallel merge sort starts with *n* / comm_sz keys assigned to each process. It ends with all the keys stored on process 0 in sorted order. To achieve this, it uses the same tree-structured communication that we used to implement a global sum. However, when a process receives another process’ keys, it merges the new keys into its already sorted list of keys. Write a program that implements parallel mergesort. Process 0 should read in *n* and broadcast it to the other processes. Each process should use a random number generator to create a local list of n/comm_sz int s. Each process should then sort its local list, and process 0 should gather and print the local lists. Then the processes should use tree-structured communication to merge the global list onto process 0, which prints the result.

Approach

To calculate the efficiency, I wrote C programs for MPI Parallel Merge Sort and then timed the amount of time it took to output a sorted list of numbers. I tested for a list of sizes between 16 and 104 incrementing by 8, so 16, 24, 32,...,104, with the sequence of 1, 2, 4, and 8 cores, for more data to compare to the serial mergesort. Efficiency is calculated by the serial run time over the parallel run time. Testing with 1 core represents a serial program run time of merge sort since in my program each core sorts their local numbers by merge sort. The timing was obtained by using MPI_Barrier at the start, and then assigning a start time with MPI_WTime and then a final time at the end before the output of the sorted list. I ran the program 50 times and only took the time for the 50th run to avoid flukes.

Parallel MergeSort tree communication diagram



Hypothesis

My hypothesis is that the more cores there are the quicker a large array size will be sorted. This is because each processor only has arraysize/#ofcores numbers initially, and then that is sorted by merge sort. So the smaller amount of numbers there are for each core, the quicker that local list would be sorted. However I think the efficiencies will be largest for smaller cores, because since, efficiency = Time serial/(cores * time parallel). So time parallel should be getting larger as the number of cores stays constant and the array size increases but the time parallel shouldn't be increasing by that much because most of the computation time would be from the local sorting that each process does.

Materials

- CF405 lab machine with 16 physical cores

Procedure

The Code

1. Process 0 receives input n and comm_sz from user
2. Process 0 Broadcasts n to all other processors
3. All processors generate n/comm_sz random numbers
4. Each processors sort their local random numbers through merge sort
5. Each process receives or sends from or to their partner core, like the diagram up above.
6. The receiving process applies the merge sort algorithm to generate a sorted local list
7. Repeat steps 4 - 6 till process 0's local list is n length long
8. The local list on process 0 is the complete sorted list

The Times

1. Write bash script with two loops, one to increment through number of cores you would like to test, the second loop to increment through the array size.
2. put "echo '($t, $i)';mpiexec -n $t ./parallelMergesort $i;" in the loop where $t is the core number, and $i is the array size number
3. run with ./scriptname.sh
4. The output will be in the form of:

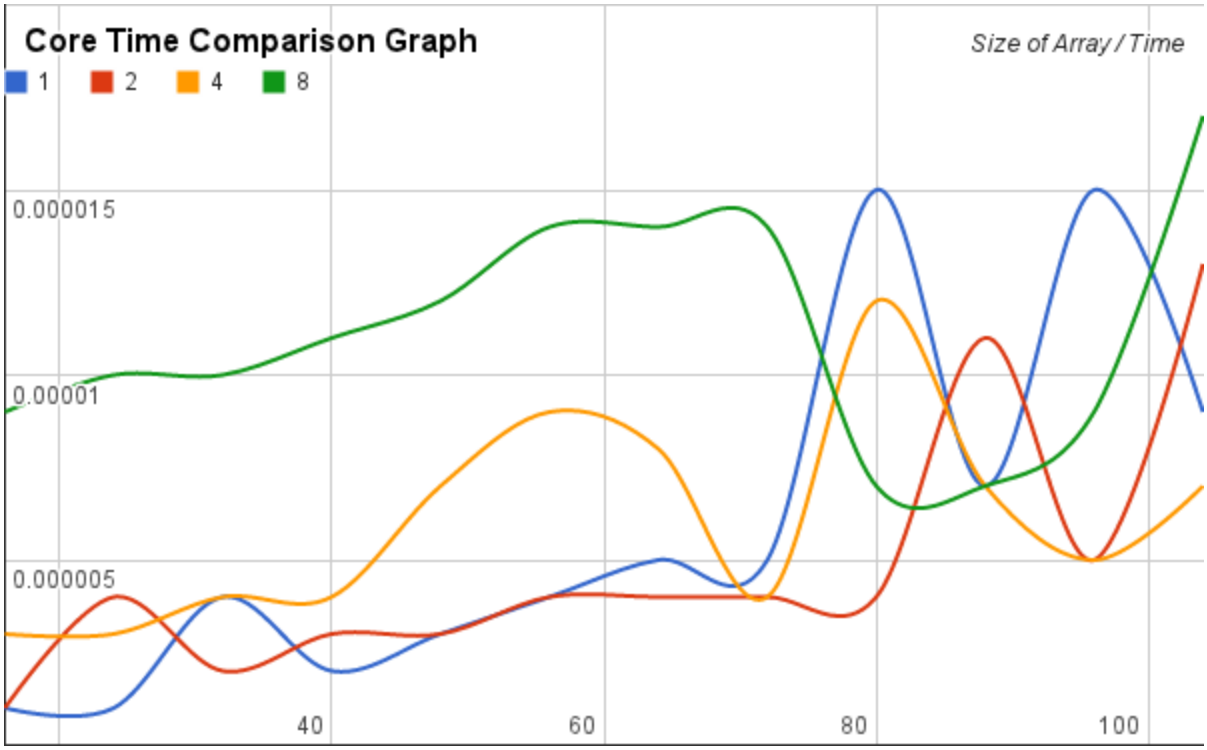
    (number of cores, array size)
    <list of local buffers from each process>
    Rank 0 ends up with: <sorted list of numbers> Time: <run time>

Results

Along the top row, lists the sizes of arrays and the first column lists the number of cores.

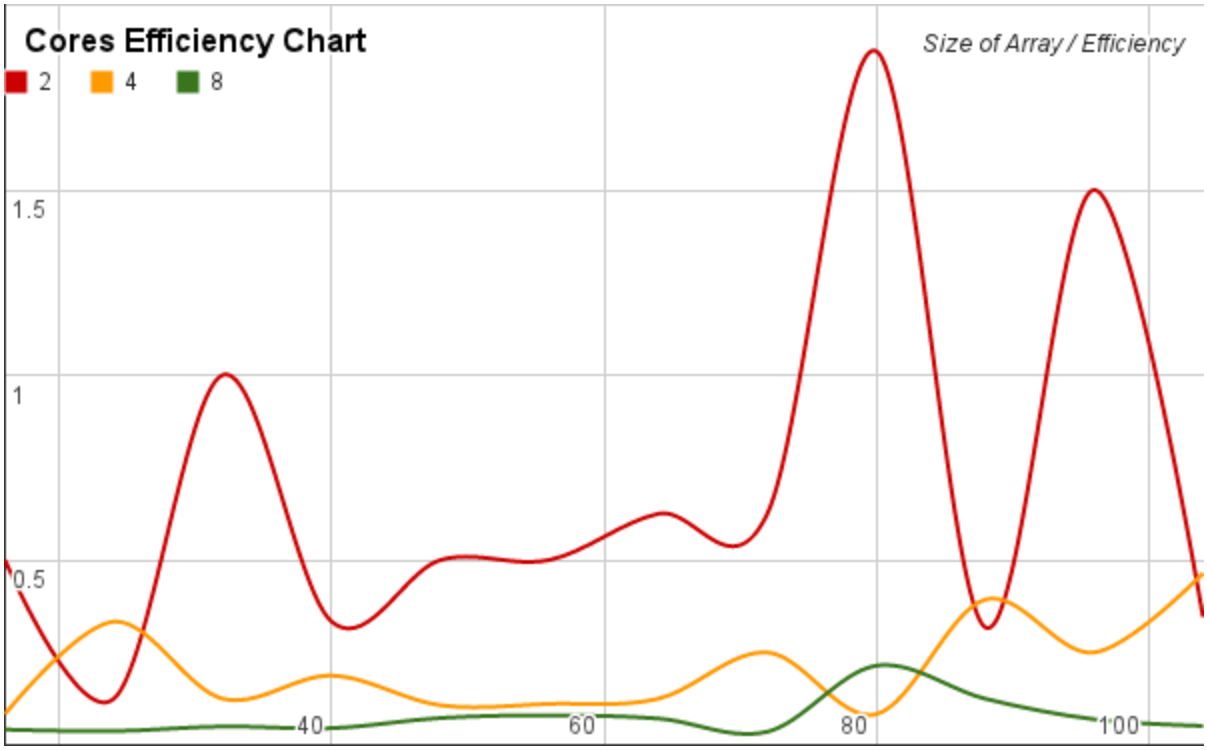
Cores Time Data

	16	24	32	40	48	56	64	72	80	88	96	104
1	0.000001	0.000001	0.000004	0.000002	0.000003	0.000004	0.000005	0.000005	0.000015	0.000007	0.000015	0.000009
2	0.000001	0.000004	0.000002	0.000003	0.000003	0.000004	0.000004	0.000004	0.000004	0.000011	0.000005	0.000013
4	0.000003	0.000003	0.000004	0.000004	0.000007	0.000009	0.000008	0.000004	0.000012	0.000007	0.000005	0.000007
8	0.000009	0.00001	0.00001	0.000011	0.000012	0.000014	0.000014	0.000014	0.000007	0.000007	0.000009	0.000017



Cores Efficiency Data

	16	24	32	40	48	56	64	72	80	88	96	104
2	0.5	0.125	1	0.33	0.5	0.5	0.625	0.625	1.875	0.318	1.5	0.34615
4	0.08	0.33	0.125	0.18	0.10714	0.11111	0.125	0.25	0.08333	0.39285	0.25	0.46428
8	0.04	0.0375	0.05	0.05	0.07291	0.08035	0.07142	0.03571	0.21428	0.125	0.06944	0.05147



Conclusion

The Core Time Comparison Chart indicates that my hypothesis was incorrect. Having 8 cores get us the worst times from 16 to 72 randomly selected numbers. I think this happened because receiving blocks the process till it receives, and then the time is calculated based on the slowest process, and having 8 cores means it has the most message passing. However from 80 to 88 random numbers, the computation time is the quickest amongst 2 and 4 processors which was my initial hypothesis. I think this happened simply because we got lucky and got a best case sequence of numbers. Having 2 cores had the best times according to my data, this was due to the fact that only one message pass was every required. I did not think that 2 would have the quickest time initially because the local sort would have taken a long time because each core would have to sort n/2 numbers.

In terms of efficiency my hypothesis was spot on, having 2 cores does in fact give very good efficiencies across the board.

Problems

My program does not accurately output the local buffers of each core or the sorted list if the size of the array typed in by the user is not a multiple of the number of cores, also typed in by the user. This is because the problem specifically states that each processor generate n/comm_sz random numbers, which would cause problems if n is not evenly divisible by comm_sz.

Run Instructions

The output of the program only gives the final sorted list of random numbers and a run time. To run the program type “mpiexec -n <number of cores> ./parallelMergesort <size of array>” where you would replace anything that starts with < and > with your intended number. The script was ran to get the numbers I presented. Run the script by typing “./script”.