ISSN:0975-9646

Parallel Histogram Sorting in MPI

Abhishek Pasari[#1], Pentyala Srinivasa Rao[#2]

[#]*Department of Applied Mathematics,*
Indian School of Mines,
Dhanbad, India

***Abstract*— Sorting is one of the most common and most important problems in the field of computer science. Parallel sorting has become very necessary in order to reduce the computation time for HPC applications which uses parallel sorting in each iteration. In this paper we present an approach to minimize the time required for sorting by distributing the array to a set of processors using MPI and output a global sorted array. We implement parallel histogram sort in MPI and then compare our performance with the NASA NAS benchmarks. In other words, we present a novel parallel sorting algorithm which is scalable, portable and optimal based on the type of counting based sort.**

***Keywords*— parallel histogram sort, parallel, HPC, MPI, OpenMPI, C++, sorting, parallel sorting, counting based sort.**

I. Introduction

Parallel sorting techniques despite being studied in large amount of literature, have not been sufficiently tested, optimized and scaled for high performance applications. With the change in hardware for high performance computing there is an inherent need for good scalable sorting algorithms. Examples of HPC applications which use sorting at every iteration is ChaNGa[1].

We propose a sorting algorithm where we start with an unknown unsorted array with an assumption that upper and lower bounds are known including the size of the array.

In this paper we describe a novel method of sorting in parallel. First we describe our serial implementation and then we proceed to describe the parallel implementation of histogram sort. In Section 2 we describe our serial algorithm, with an parallel algorithm in Section 3. In Section 4 tools which are required for our evaluation is described. Section 5 shows the various results based on our implementation. In the end Section 6 presents the conclusion with Section 7 describing the future work.

II. Sequential Algorithm

Here we first consider a unsorted array with two assumptions, firstly the lower bound of the keys inside an array and the upper bound of the range of keys inside an array is known. Secondly, the array must contain unsigned integers.

The sequential algorithm works as follows:

```
Input the unsorted array of size N_MAX
  Declare a histogram array of size MAX_KEY.
  Iterate over input array and add +1 to histogram array
       value.
  Construct sorted array through histogram generated.
Output the reconstructed sorted array.
```

In this above described algorithm we have taken **MAX_KEY** to be equal to the upper bound of the range of keys stored inside the unsorted array i.e. if 16 is the largest value in the range of number stored in an array then **MAX_KEY** is equal to 16.

III. Parallel Histogram Sort

The figure described here gives a working flow chart of the parallel histogram sorting algorithm implemented in this paper.

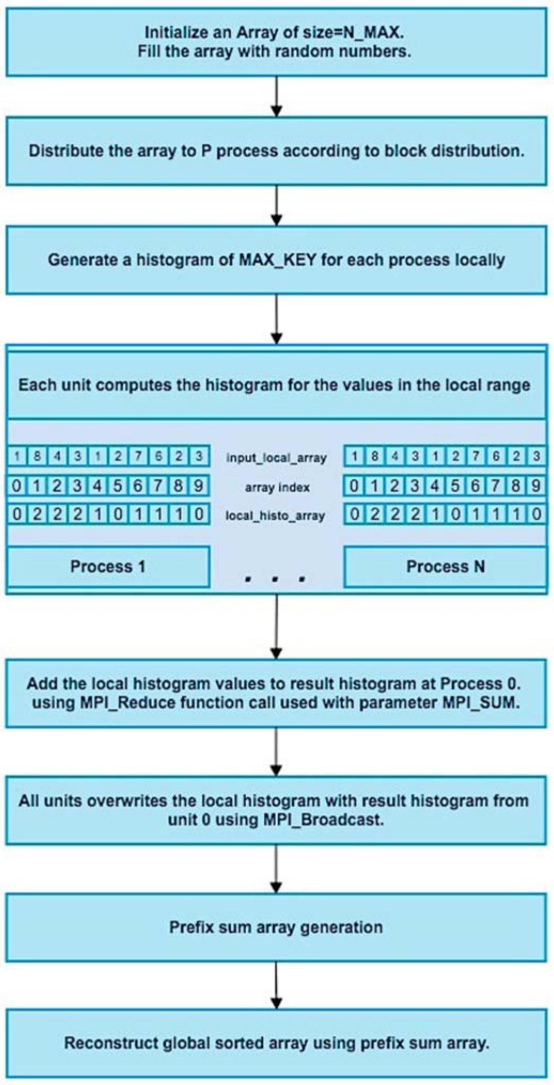
Initialize an Array of size=N_MAX.
Fill the array with random numbers.
Distribute the array to P process according to block distribution.
Generate a histogram of MAX_KEY for each process locally
Each unit computes the histogram for the values in the local range
1 8 4 3 1 2 7 6 2 3
0 1 2 3 4 5 6 7 8 9
0 2 2 2 1 0 1 1 1 0
input_local_array
array index
local_histo_array
1 8 4 3 1 2 7 6 2 3
0 1 2 3 4 5 6 7 8 9
0 2 2 2 1 0 1 1 1 0
Process 1
. . .
Process N
Add the local histogram values to result histogram at Process 0.
using MPI_Reduce function call used with parameter MPI_SUM.
All units overwrites the local histogram with result histogram from
unit 0 using MPI_Broadcast.
Prefix sum array generation
Reconstruct global sorted array using prefix sum array.

Fig. 1 Parallel histogram sort working diagram.

Fig. 1 explains the working of the algorithm where size of the input unsorted array is **N_MAX** and the size of number of process in the program is **P** with **MAX_KEY** equal to the biggest element in the input array. The input array is filled with random numbers and it is distributed in a block distribution pattern to all process with each process receiving **N_MAX/P** part of the array. After distribution, a local histogram is generated for each process. Each process computes is local histogram as shown in the figure where *input_local_array* is the part of distributed array to each process. *array_index* is the location of the array elements in both arrays. *local_histo_array* is the local histogram array for each process. The resulting local histogram are added together with **MPI_REDUCE** function to **Process 0. Process 0** after receiving the added local histogram broadcasts it to each process where they are able to generate the prefix sum array. After generation of prefix sum array each process constructs its part of the array resulting into a globally sorted array.

IV. Experimental Setup

A. OpenMPI

It is one of the most widely used and efficient model in parallel programming. In message passing approach a group of processes executes programs written in a programming language like C or C++ with calls of functions for sending and receiving messages. Open MPI is able to combine the expertise, technologies, and resources across the High Performance Computing community in order to build the best MPI library available.

B. NAS Benchmarks

The NAS Parallel Benchmarks[2] are a set of benchmarks designed to measure the performance of supercomputers. As there were only few benchmarks available NASA Advanced Supercomputing(NAS) Division in 1991 developed the benchmark in order to measure the performance of supercomputer. The benchmarks were extended and translated into different languages with currently the overall capacity of 11 different benchmarks. Out of them 8 were from the first version of the benchmarks. The benchmarks originally were written in Fortan and C as both the programming languages support OpenMP and MPI.

1.Integer Sort Benchmark: The IS (Integer Sort) NAS Parallel Benchmark using one of the class C which has over 2^{27} elements in its array measures the time needed for sorting using a histogram. The histogram sorting method is similar to the bucket sort algorithm with bucket size 1. In bucket sort algorithm it sorts the elements into each bucket according to their size and then sorts the elements in each bucket. Here in this benchmark we simply create a histogram array that counts the we simply create a histogram array that counts how often each number occurs in the key array. The histogram can then be used to print a sorted version of the key array. In other words, if the number 0 occurred 6 times according to the histogram, we simply write six zeros to the sorted array and then continue with the next number. In this benchmark the elements of the histogram arrays are added up to the right which is also called accumulated histogram. The accumulated histogram provides the same information as the regular histogram.

We compare our results implemented in MPI using OpenMPI with that of NAS benchmarks.

V. Results

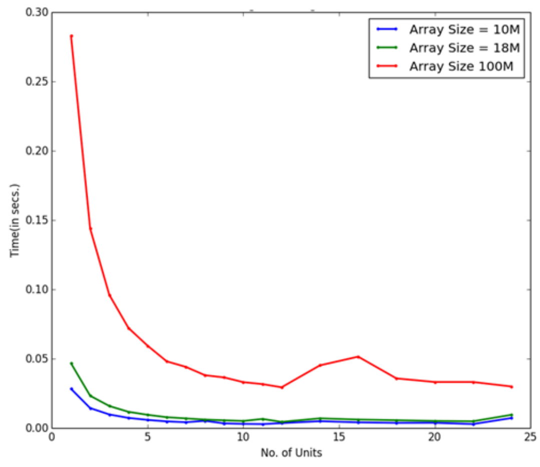

Fig. 2 Scaling of Parallel histogram sort

As you can see in the graph of Fig. 2 which shows how the behavior of a particular array size w.r.t time as the number of process increases the time taken by a particular array size decreases. The bumps shown in the figure is the cache effect by the hardware.

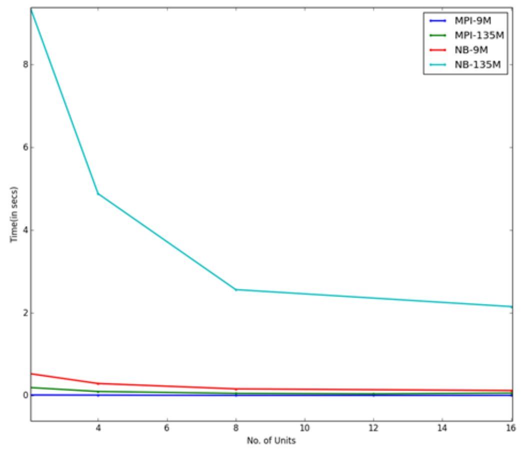

Fig. 3 Comparison of Parallel Histogram Sort with NAS benchmarks Integer Sort.

Here we can see the comparison between the parallel histogram MPI implementation vs Nasa Benchmark(NB) over the array size of 9 Million and 135 Million. we can see hat our implementation performs better than the NASA implementation under the process(units) size ranging from 4 to 16.

VI. Conclusions

The histogram sort has been successfully implemented in parallel using MPI in C++. We have been able to reduce the computation time drastically as shown in the above two graphs. Our algorithm still performs better when compared to NASA NAS benchmarks. Different array size can be sorted through our implementation if the maximum values to be sorted is known to the user with the values being integer.

VII. Future Work

In future this algorithm could be extended on a hybrid mechanism such as MPI+CUDA or OpenMP+ CUDA where we may see good scaling of our algorithm.

ACKNOWLEDGMENT

We would like to thank my guide Dr. P.S. Rao for his support as my guide and mentor throughout the work of this paper. Also, we would like to thank the Department of Applied Mathematics for the support throughout the work.

REFERENCES

1. Jetley, P., Gioachin, F., Mendes, C., Kale, L. V., & Quinn, T. (2008, April). Massively parallel cosmological simulations with ChaNGa. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on* (pp. 1-12). IEEE.
2. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S. and Simon, H.D., 1991. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, *5*(3), pp.63-73.