IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 5, Issue 6, June 2016

A New Parallel Split and Concurrent Selection Sorting Algorithm based on Binary Search

Jagdeep Singh[1], **Alka Singh**[2]

M. Tech, Computer Science & Engineering, KNIT, Sultanpur, India[1]

Asst Professor, Dept of Computer Science & Engineering, KNIT, Sultanpur, India[2]

Abstract: We all know that the most important procedure in managing data is its sorting. To perform sorting, one can choose different sorting algorithmic methods. But, all the various sorting algorithms do not result the same speed, execution time and efficiency at a given set of inputs. Hence, it is necessary to know which algorithm can give better result for a given platform and pre-defined data sets. This paper presents reader an experimental study of performance comparison for various parallel sorting algorithms. Proposed algorithm shows up to 50 times speed up as compare to serial and two fold speedup as compare to parallel algorithm.

Keywords: Bitonic sort, odd-even merge sort, parallel merge sort, parallel rank sort, complexity.

I. INTRODUCTION

The process of rearranging data in a particular order that may be in a increasing or decreasing manner is termed as sorting. The data set may be alphabetical or numerical in nature. Every existing matter in this world has its few advantages as well as disadvantages. Similarly different sorting algorithms have their advantages and disadvantages. Sorting is used for performing search operation in an easier way saving time. But, sorting algorithms can't be used with same efficiency on all data-sets. Every method depending upon the data-set performs differently in different aspects with respect to processing time, speed and efficiency; few perform poorly whereas few give results in a short time. To understand the above phenomena in a better descriptive way, we in this paper will carry out an experimental study for an assumed data-set of various parallel sorting algorithms on the basis of performance and complexity.

PARALLEL SORTING METHODS

There are number of parallel sorting algorithms are available for different machines: Parallel odd-even transposition is an extension of bubble sort, operates in two alternate phases. First is called even-phase where even processors exchange values with right neighbours and second is odd-phase where odd processors exchange values with right neighbours. Parallel merge sort [1] is based on divide and conquers strategy. It assigns work to processors organized as a tree. First subdivides it in two parts and give it to the particular processors. Again apply the same method to each part. After that start merging between two processors element in sorted order, again apply the same method until they get the sorted data sequence [2]. Time complexity of parallel merge sort is 0 (n) but it has unbalanced processor [3] and load communication. Parallel sorting methods are classified on the following basis:

1. On the basis of memory usage:

Sorting method	Memory occupied
Odd – even sort	1
Parallel merge sort	n
Bitonic sort	n(log(logn))
Parallel rank sort	n(log(logn))
Parallel quick sort	logn

2. On the computational complexity basis:

Sorting method	Best case	Average case	Worst case
Odd – even sort	N	n^2	n^2
Parallel merge sort	nlogn	nlogn	nlogn
Bitonic sort	(log(logn))	(log(logn))	(log(logn))
Parallel rank sort	(log(logn))	(log(logn))	(log(logn))
Parallel quick sort	nlogn	nlogn	n^2

1. On the basis of recursion: Algorithms may be recursive or non-recursive in nature. Few may be both viz. parallel merge sort.

How much an algorithm's efficiency and performance can be improved by Parallelism? To answer this question, we have developed and executed three parallel sorting algorithms [4]. Execution time is the time required to execute an algorithm whether in sequential environment or parallel environment; Speed up is the ratio of the total time taken in the execution of an algorithm in sequential environment to that in a parallel environment [10].

$$s(n,p) = \frac{T(n,1)}{T(n,p)}.$$

II. PARALLEL PROGRAMMING

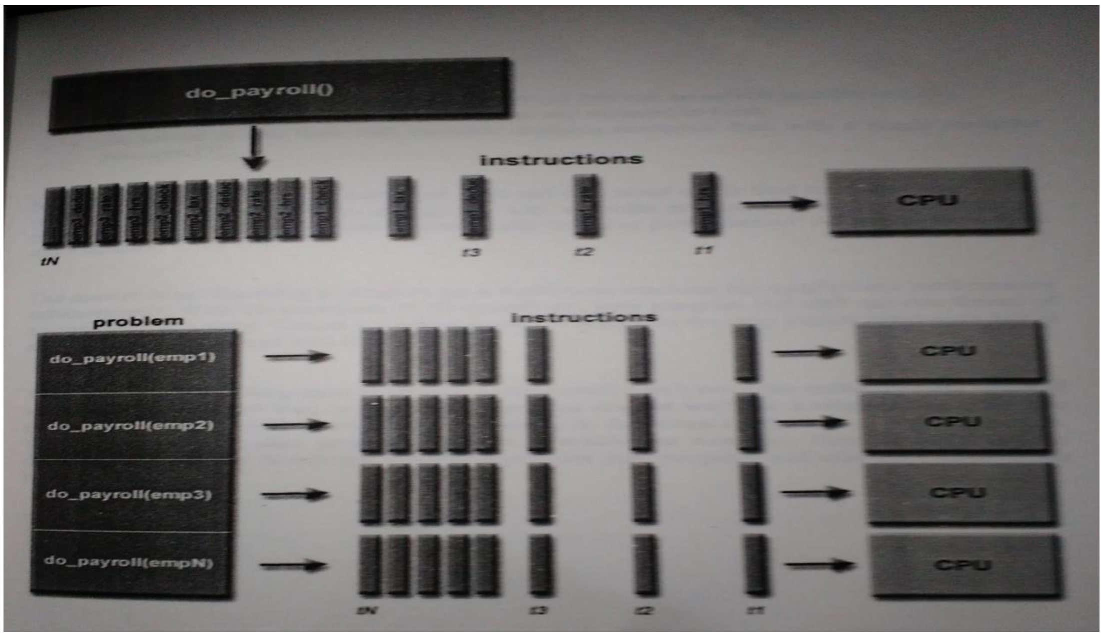


Figure 1- Parallel Execution of instructions

Parallel Computing Toolbox provides several high-level programming constructs that let you convert your applications to take advantage of computers equipped with multicore processors and MATLAB-PCT. Constructs such as parallel for-loops (parfor) and special array types for distributed processing and for GPU computing simplify parallel code development by abstracting away the complexity of managing computations and data between your MATLAB session and the computing resource you are using.

We can run the same application on a variety of computing resources without reprogramming it. The parallel constructs function in the same way, regardless of the resource on which your application runs—a multicore desktop (using the toolbox) or on a larger resource such as a computer cluster (using toolbox with MATLAB Distributed Computing Server).

SPMD (single program, multiple data)
It executes code in parallel on workers of parallel pool
Syntax
spmd, statements, end
spmd(n), statements, end
spmd (m,n), statements, end

Description
The general form of a spmd (single program, multiple data) statement is:

```
spmd
                    statements
end
```

spmd, statements, end defines a spmd statement on a single line. MATLAB® executes the spmd body denoted by statements on several MATLAB workers simultaneously. The spmd statement can be used only if you have Parallel Computing Toolbox. To execute the statements in parallel, you must first open a pool of MATLAB workers using parpool or have your parallel pretences allow the automatic start of a pool.

Inside the body of the spmd statement, each MATLAB worker has a unique value of labindex, while numlabs denotes the total number of workers executing the block in parallel. Within the body of the spmd statement, communication functions for communicating jobs (such as labSend and labReceive) can transfer data between the workers.

Values returning from the body of a spmd statement are converted to Composite objects on the MATLAB client. A Composite object contains references to the values stored on the remote MATLAB workers, and those values can be retrieved using cell-array indexing. The actual data on the workers remains available on the workers for subsequent spmd execution, so long as the Composite exists on the client and the parallel pool remains open.

By default, MATLAB uses as many workers as it finds available in the pool. When there are no MATLAB workers available, MATLAB executes the block body locally and creates Composite objects as necessary.

spmd(n), statements, end uses n to specify the exact number of MATLAB workers to evaluate statements, provided that n workers are available from the parallel pool. If there are not enough workers available, an error is thrown. If n is zero, MATLAB executes the block body locally and creates Composite objects, the same as if there is no pool available.Spmd(m,n), statements, end uses a minimum of m and a maximum of n workers to evaluate statements. If there are not enough workers available, an error is thrown. M can be zero, which allows the block to run locally if no workers are available.

Examples
Perform a simple calculation in parallel, and plot the results:

```
parpool(3)
spmd
  % build magic squares in parallel
  q = magic(labindex + 2);
end
for ii=1:length(q)
  % plot each magic square
  figure, imagesc(q{ii});
end
delete(gcp)
```

Codistributed
It creates codistributed array from replicated local data
Syntax
C = codistributed(X)
C = codistributed(X,codist)
C = codistributed(X, lab,codist)
C = codistributed (C1, codist)

Description
C = codistributed(X) distributes a replicated array X using the default codistributor, creating a codistributed array C as a result. X must be a replicated array, that is, it must have the same value on all workers. Size(C) is the same as size(X).C = codistributed(X, codist) distributes a replicated array X using the distribution scheme defined by codistributor codist. X must be a replicated array, namely it must have the same value on all workers. Size(C) is the same as size(X). For information on constructing codistributor objects, see the reference pages for codistributor1d and codistributor2dbc.C = Codistributed (X, lab,codist) distributes a local array X that resides on the worker identified by lab, using the codistributor codist. Local array X must be defined on all workers, but only the value from lab is used to construct C. size(C) is the same as size(X).C = Codistributed (C1,codist) accepts an array C1 that is already codistributed, and redistributes it into C according to the distribution scheme defined by the codistributor codist. This is the same as calling C = redistribute (C1, codist). If the existing distribution scheme for C1 is the same as that specified in codist, then the result C is the same as the input C1.
Examples
Create a 1000-by-1000 codistributed array C1 using the default distribution scheme.

```
spmd
   N = 1000;
   X = magic(N);         % Replicated on every worker
   C1 = codistributed(X); % Partitioned among the workers
end
```

Create a 1000-by-1000 codistributed array C2, distributed by rows (over its first dimension).

```
spmd
   N = 1000;
   X = magic (N);
   C2 = codistributed(X, codistributor1d (1));
End
```

III. RELATED WORK

Our proposed parallel sorting algorithm on MATLAB is a hybrid algorithm, the combination of parallel radix sort and parallel selection sort . First it split the data sequence into several pieces then apply radix sort concurrently on all the pieces. After that it uses parallel selection sort to find the correct position of each element of a data sequence concurrently[5]. Then copy the elements of a data sequence to corresponding position to obtain the final sorted data sequence. The complete parallel sorting algorithm is called "split and parallel selection" algorithm composed of the above two stages. In which our parallel radix sort is similar as existing parallel radix sorting algorithm[9]. However its efficiency depends to a large extent on selection sort that finds the final position of elements in sorted data sequence.

Split and Concurrent Selection
Let us assume a shared memory multiprocessor with n processors, denoted by PI, P2........ Pn. Again, let us assume a data sequence D of size S which is initially unordered and one more data sequence d of size S which is initially empty. Proposed parallel sorting algorithm first split the data sequence D into subsequences of size Sin. Each subsequence is denoted by D and assigned to the processor P j where i is from 1 to n.

After splitting the data sequence D of size S equally to n processors, each processor Pi gets Sin elements and sort its assigned subsequence Di of size Sin by using a fast sequential radix sort parallely on each processor individually. An algorithm for proposed algorithm SCS is as follows:

Proposed Algorithm:
```
1).D ← Data Set, Mo ← 0; i ←1 to S/p
2).S ← Size, FST ← First element of corresponding individual lab, LST ← Last element of corresponding individual lab.
 3).For all processors P do in parallel
% initialising spmd tool to distribute jobs over
% all 4 labs for parallel processing
 4) spmd
   % distributing inputs to each lab
 5)p=codistributed(a);
 6)P=getLocalPart(p);
 7)len=length(P);

   % calling radixSort() method to sort elements
   % and storing result from individual lab
 8) Apply Radix sort on P parallelly.
```

```
9)end
10) if(Pi[k] < Po[FST]
11)break;
12)else(Pi[k] > Po[LST]) then
13) Mo ← Mo + total no. Of individual processor elements
14)Break
15) Elseif apply Binary Search to find the numbers of elements Mo are smaller than Pi[k] in po.
16) End if
17) Xi[k] = Xi[k] + Mo
18). Copy the elements of data sequence D to the corresponding position in data sequence.
19). Final sorted sequence obtained.
20). stop the algorithm
```

IV. COMPUTATIONAL COMPLEXITY

A. Complexity Analysis DISCS Algorithm

This section presents detail analysis of the computational complexity of proposed algorithm. This analysis is divvied in two parts: First, parallel radix sort phase complexity and second selection sort based on binary search[6].

B. Parallel Radix Sort Phase Complexity

Radix sort is one of the oldest and well- known sorting algorithms on sequential machines. It is the most efficient sorting algorithm for small element. It assumes that the elements are d digit numbers and sort one digit of element at a time, from least to most significant bit. In sequential radix sort, for a fixed element size d, the complexity of n records is 0 (n). In our proposed algorithm, radix sort is used in parallel to sort the each subsequence of size Sin, so that computational complexity of first phase is O(S/n).

C. Selection Sort Based On Binary Search Complexity

This phase is used to find out the sorted position of elements by using binary search algorithm. Binary search is the best searching algorithm for sorted elements. The complexity of binary search for n record is 0 (1) in best case and 0 (logn) in worse case. In the proposed algorithm binary search is used to find out the sorted position of an element in subsequence of size Sin and find the position in all S subsequence[7]. So, the computational complexity of second phase is 0 (n*logS/n).

D. SCS Complexity

According to the analysis in above subsections, the computational complexity of SCS parallel sorting algorithm is 0 (S/n) + 0 (nlogS/n).

V. IMPLEMENTATION

To implement this algorithm in MATLAB-PCT we made two kernels First kernel is used to divide the data sequence into subsequences and sort the each subsequence Di through corresponding thread Ti by using sequential radix sort. Second kernel uses the results of first kernel. Each thread Ti applies selection sort on sorted Di in which it uses binary search to fmd out the exact sorted position of each element of data sequence D. After that, copy the corresponding element of the obtained position in data sequence d. Finally we get the sorted output in data sequence d.

Here we have evaluated the speed up of parallel selection sort on MATLABs with parallel sorting algorithm based on odd-even merge sort and sequential quick sort. Sequential quick sort is implemented in c and the parallel odd-even merge sort is implemented in MATLAB-PCT. We have evaluated the performance of all implemented algorithms on a large data sequence having number of elements from lK to 100M.

Result shows that Parallel Proposed sort is better than others for large data sequence. It gives almost two times speedup than parallel odd-even merge sort. For small size of data sequence like has less than 10K elements sequential quick sort is better than others. But more than 10 K elements of data sequence parallel selection sort takes less execution time.

Table I shows the execution time of sequential quick sort and parallel selection sort. It shows that sequential quick sort gives better performance for small data sequence which has less than 10K elements. Parallel selection sort gives almost 300X speed up than sequential quick sort for large data sequence which has more than 1 M elements. This is just because of number of threads work independently and simultaneously to sort the large data sequence.

Table.1 the execution time of sequential quick sort and proposed sort

Numbers of Elements	Sequential Quick sort (in milliseconds)	Proposed sort(in milliseconds)
1K	20.0	41.34
10K	170.0	239.47
20K	390.0	280.80
50K	1300.0	363.15
100K	5700.0	638.90
1M	131000.0	2548.40

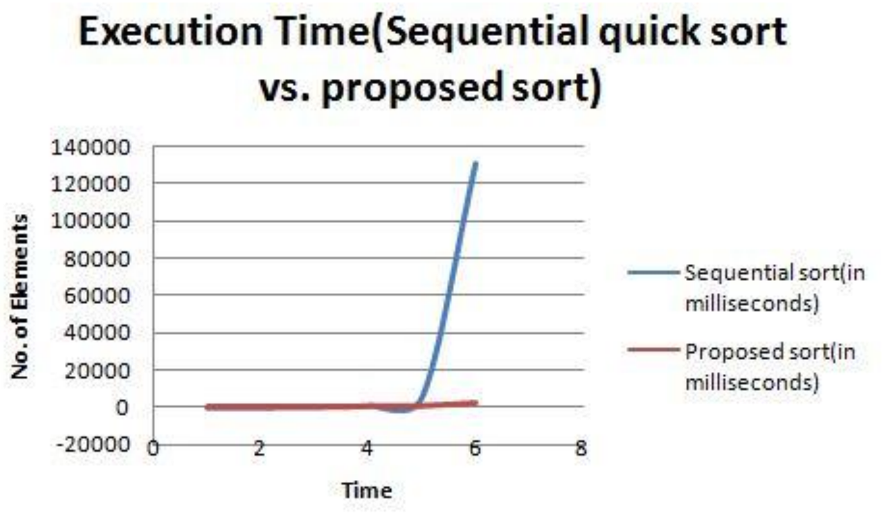


Fig.2 Performance comparison of sequential quick sort and proposed sort

Table II shows the execution time of parallel odd- even merge sort and Parallel selection sort on MATLAB-PCT with 100 elements in each threads. After analysing the results of both parallel algorithm we can say that our proposed algorithm gives better performance than parallel odd-even merge sort[8].

Table.2 the execution time of Parallel odd even merge sort and proposed sort

Numbers of Elements	Parallel Odd-even Merge sort (in milliseconds)	Proposed sort(in milliseconds)
1K	70.99	41.34
10K	287.43	239.47
20K	387.80	280.80
50K	521.27	363.15
100K	956.72	638.90
1M	8426.48	4348.40
10M	142867.89	84850.95
20M	307152.90	113570.86
50M	1023964.35	432653.21

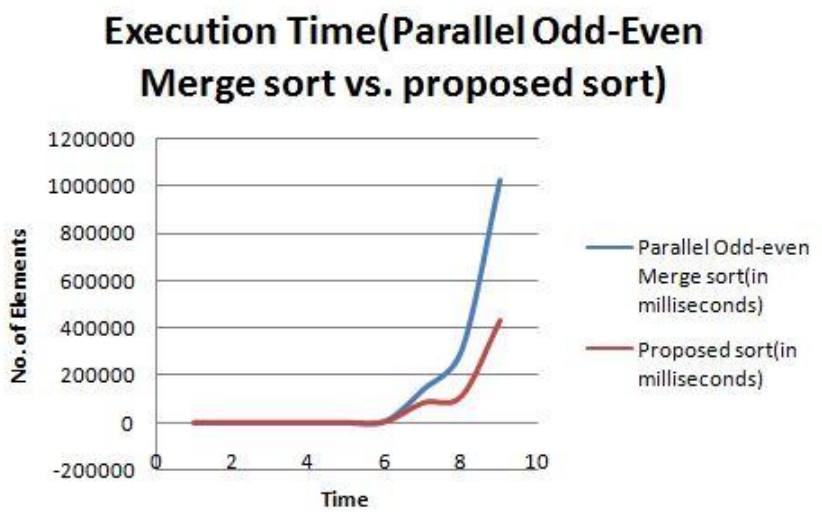
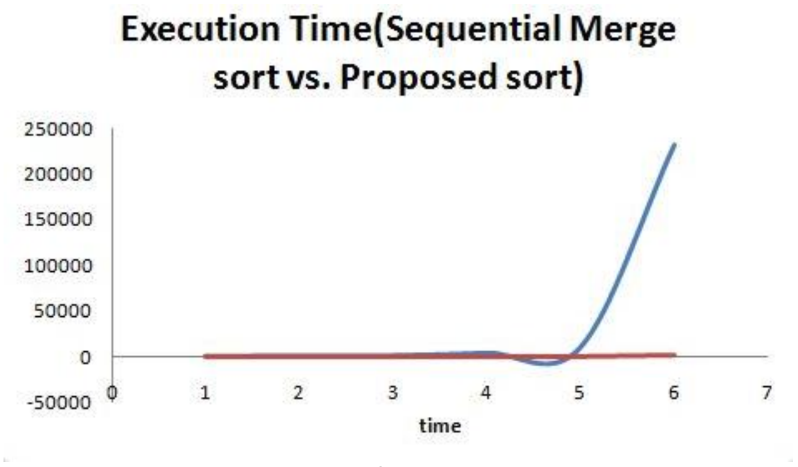


Fig.3 the execution time of Parallel odd even merge sort and proposed sort

Table III shows the execution time of Sequential merge sort and Proposed sort on MATLAB-PCT with 100 elements in each threads. After analysing the results of both parallel algorithm we can say that our proposed algorithm gives better performance than merge sort.

Table.3 the execution time of merge sort and proposed sort

Numbers of Elements	Sequential Merge sort (in milliseconds)	Proposed sort(in milliseconds)
1K	28.0	41.34
10K	176.0	239.47
20K	590.0	280.80
50K	3300.0	363.15
100K	9700.0	638.90
1M	231000.0	2548.40



(Fig.4 the execution time of merge sort and proposed sort.)

VI. CONCLUSION

We have shown that how after decreasing number of elements per thread proposed algorithm is giving better performance. Proposed parallel algorithm gives almost 50x speed-ups than sequential quick sort for a large data sequence. It gives almost 3.5x speed-ups than parallel odd-even merge sort for our MAT LAB-PCT based machine. In future we can improve the performance of parallel section sort by minimising the number of elements with any other elements is compared to find its correct position in list.

REFREENCES

[1] Kalim Qureshi and Haroon Rashid,"A Practical Performance Comparison of Parallel Matrix Multiplication Algorithms on Network of Workstations.", IEEE Transaction Japan, Vol. 125, No. 3, 2005.

[2] Kalim Qureshi and Haroon Rashid," A Practical Performance Comparison of Two Parallel Fast Fourier Transform Algorithms on Cluster of PCS", IEEE Transaction Japan, Vol. 124, No. 11, 2004.

[3] Kalim Qureshi and Masahiko Hatanaka, "A Practical Approach of Task Partitioning and Scheduling on Heterogeneous Parallel Distributed Image Computing System," Transaction of IEEE Japan, Vol. 120-C, No. 1, Jan., 2000, pp. 151-157.

[4] K. Sado, Y. Igarashi, Some Parallel Sorts on a Mesh-Connected Processor Array and Their Time Efficiency, Journal of Parallel and Distributed Computing, 3, pp. 398- 410, 1999.

[5] D. Bitton, D. DeWitt, D.K. Hsiao, J. Menon, A Taxonomy of Parallel Sorting, ACM Computing Surveys, 16,3,pp. 287-318, September 1984. [1] S. Huba and K. Dzmitry, "Parallel merge sort," Published by Chapman University, California, USA, March 1, 2011.

[6] E. Sintorn and U. Assarsson, "Fast parallel GPU-sorting using a hybrid algorithm," Published by Chalmers University Of Technology, Gothenburg, Sweden, 2007.

[7] M. Ratnayake and K. Amer, "An FPGA architecture of counting-sorting on a large data Volume: Application to video signals," IEEE 41st Annual Conference on Information Science and Systems, Baltimore, MD, Mar. 2007, pp.431-436.

[8] L. Ha, J. Kruger and T. Silvay, "Fast 4-way parallel radix sorting on GPUs," University of Utah, submitted to Computer Graphics Forum, (2/ 2009).

[9] D. Z. Chen., "Efficient parallel binary search on sorted arrays with applications," IEEE Trans. on Parallel and Distributed Systems, Vol. 6. Pp.440-445, Apr. 1995.

[10] H. Peters, O. Schulz-Hildebrandt, and N. Luttenberg "parallel External Sorting for CUDA- enabled GPUs with load balancing and low transfer overhead," IEEE International Symposium on Parallel & Distributed processing, Workshops and Phd Forum (IPDPSW), Atlanta, GA, 19-23 April 2010, pp.I-8.