Sequential & Parallel Hybrid Approach for Non-Recursive Most Significant Digit Radix Sort

Ahmet Arif Aydin
Inonu University
14 PUBLICATIONS **59** CITATIONS
SEE PROFILE

Gita Alaghband
University of Colorado
71 PUBLICATIONS **392** CITATIONS
SEE PROFILE

SEQUENTIAL&PARALLEL HYBRID APPROACH FOR NON-RECURSIVE MOST SIGNIFICANT DIGIT RADIX SORT

Ahmet Arif Aydin[1] and Gita Alaghband[2]

[1]University of Colorado Boulder - Computer Science Department, 1111 Engineering Dr, Boulder, CO 80309

[2]University of Colorado Denver - Department of Computer Science and Engineering, Denver, CO, 80217

ABSTRACT

Sorting and parallelism are widely studied subjects and their combination is a challenging topic of importance in many scientific fields. Efficient sorting is important because of reduced time and energy consumption; implementation of efficient parallel sorting algorithms and utilizing multicore environment's power is the main focus of this article. In this study, traditional (recursive) sequential and alternative (non-recursive) sequential versions of the most-significant-digit radix sort algorithms are implemented, and their performance is compared. Furthermore, two hybrid sequential and parallel versions combining the most-significant-digit radix sort and Quicksort are implemented and evaluated. OpenMP API is utilized to implement the parallel algorithms. The resulting parallel versions outperform the well-known efficient Quicksort algorithm.

KEYWORDS

Hybrid Sort, Radix Sort, Quicksort, Parallel Sorting, OpenMP

1. INTRODUCTION

Fast and efficient sorting algorithms have always been studied by scientists from different areas focusing on less power consumption and fast turn-around time. Parallelism (Mollick 2006) is also a crucial topic to study because today's computers have been empowered by multiple cores which are available even in contemporary smart phones (Kim 2011). While many fast and efficient sequential sorting algorithms have been design and are employed by numerous applications, their parallelization to obtain better performance poses a major challenge. In this study, we focus on the important topics of sorting and parallelism.

Our first goal is to come up an alternative sequential most significant digit (MSD) radix sort algorithm that performs better than the traditional recursive version. The second goal is to implement the sequential and parallel versions of a hybrid combination of most significant digit radix sort and quicksort algorithms and study their performance.

A general overview of Radix sort and Quicksort algorithms are given in Section 2. Implementation details of traditional recursive, non-recursive, and hybrid versions of MSD radix sort algorithms are explained in Section 3, and performance benchmarking and runtime analysis of given algorithms are given in Section 4.

2. ALGORITHMS OVERVIEW

2.1 Radix Sort Algorithm

Radix sort is a well known and fast sorting algorithm. Previous studies have shown that radix sort is not limited to sort only integers (Han & Thorup 2002). It has been applied to strings (Zhang 2008) where they are sorted by their characters and binary numbers where they are sorted by their bits (Cheng 2003).

There are two main radix sorting strategies: the least-significant-digit and the most-significant-digit radix sorts. In this study, we focus on the most-significant-digit radix strategy with integer input. Some variations of the MSD Radix sort algorithm are Counting Sort (Cormen et al., 2009), Pigeonhole Sort (Pigeonhole Sort) and Postman's Sort, also known as Postal Sort (Ramey 1992).

The basic method is sorting the input values based on corresponding digits, characters or bits of each entry. Radix sort utilizes buckets for the sorting process; the number of buckets is determined based on the input type. If input type is binary, two buckets are needed for sorting because binary numbers consist of at most two distinct numbers 0 or 1's. Runtime complexity of radix sort is O (n*d) where n is size of input and d is iteration count over input and is defined by the digit count of the maximum number or length of the longest string in the input (Chang 2003).

2.2 Quicksort Algorithm

Quicksort is one of the most efficient known sequential sorting algorithms with an average running time of O(n * log n). Quicksort is a widely studied algorithm and is described in detail in (Cormen et al., 2009). In our hybrid algorithm, we take advantage of the efficiency of the sequential Quicksort and the parallelism inherent in MSD radix sort and combine them to come up with an efficient Hybrid MSD radix sort version. Quicksort implementation for the hybrid version is given in Section 3.

3. SORTING ALGHORITHMS

Radix sort uses buckets for the sorting process. The number of buckets is determined by the type of input. In this study, where the input type is integer all possible integers can be formed by the ten digits: 0, 1, 2…, 9. Therefore, a total ten buckets each representing all possible values of a digit is needed. The data set (input) consists of randomly generated numbers with varying range and size. Input size is at most 4 million and numbers are at most 10 digits long.

The remainder of this section presents our implementations of a number of sorting algorithms for our comparative study in the following order: the traditional recursive and non-recursive MSD radix sort, sequential Quicksort to be used in the hybrid versions, two sequential and two parallel versions of the Hybrid MSD radix sort. OpenMP API (OpenMP Architecture Review Board 2008) (Ma et al. 2009) (Sato 2002) is used to implement the shared memory MIMD parallel versions of the algorithms.

3.1 Traditional (Recursive) MSD Radix Sort Implementation

```
Recursive MSD(& data,digit)
{
    // recursion stop condition
    if (data size== 1 or digit<1)
        return

    create buckets and insert numbers

    // decrease the digit
    digit--

    // operate on each bucket
    for (i=0;i<10;i++)
        Recursive MSD(i th bucket, digit )
        //recursive call for each bucket
}
```

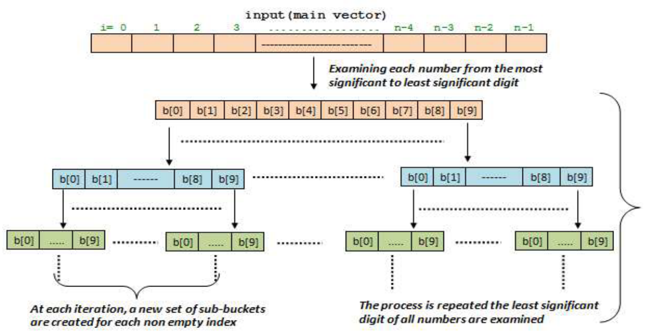

input(main vector)
i= 0
1
2
3
n-4
n-3
n-2
n-1
Examining each number from the most significant to least significant digit
b[0] b[1] b[2] b[3] b[4] b[5] b[6] b[7] b[8] b[9]
b[0] b[1] b[8] b[9]
b[0] b[9]
At each iteration, a new set of sub-buckets are created for each non empty index
The process is repeated the least significant digit of all numbers are examined

Figure 1. Recursive MSD radix sorting process and pseudo code

MSD radix sort is named after its sorting operation. Numbers are examined starting from the most significant to the least significant digits. A high level pseudo code and diagram for the recursive MSD radix sort algorithm sorting are presented in Figure 1. In the figure, b[i] refers to a bucket corresponding to digit value i and holds a pointer to a linked list of numbers whose digit value of the radix being examined equals i.

The sorting process begins with passing the entire input (main vector) to the *Recursive MSD* function by reference. This allows the function to make changes directly to the input. Numbers from the input vector are placed into buckets based on the value of the corresponding digits being examined (most to the least significant digit order). Each of these buckets is sent to the Recursive MSD function for further execution. There are two conditions to stop the recursive call: when the size of the bucket is equal to one, and if the digit count is smaller than one. The latter indicates that prior call has worked on the least significant digit, and there are no more digits left. Since buckets from recursive calls are sorted, they are merged into the main input.

3.2 Non-recursive MSD Radix Sort Implementation

The fact that MSD radix sort begins its sorting operation with examining numbers from the most significant digit, provides the opportunity to allocate each of the numbers in their final buckets after the first iteration. Therefore, each bucket can be considered as an independent vector to be sorted in this range. Non-recursive MSD radix sort's diagram is illustrated in Figure 2. First_bucket and second_bucket are two complete bucket data structures implemented as a two dimensional vector. We also use a middle_bucket data structure to sort individual vectors belonging the first_bucket (and/or second_bucket) for the subsequent digits as described below. Before sorting process begins, maximum number of input is found and its digit count is calculated. Digit count determines the number of iterations of MSD radix sort over the input.

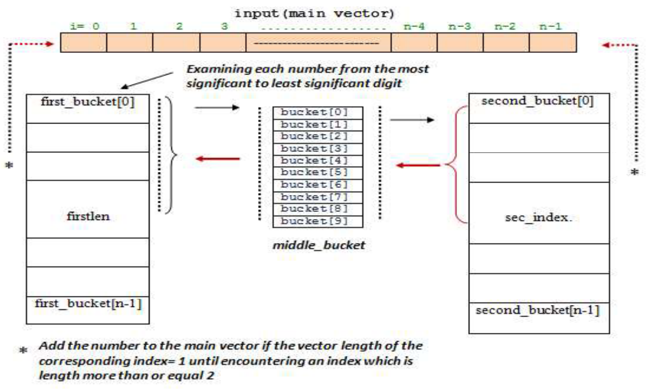

input(main vector)
i= 0 1 2 3 .... n-4 n-3 n-2 n-1
Examining each number from the most significant to least significant digit
first_bucket[0]
firstlen
first_bucket[n-1]
bucket[0]
bucket[1]
bucket[2]
bucket[3]
bucket[4]
bucket[5]
bucket[6]
bucket[7]
bucket[8]
bucket[9]
middle_bucket
second_bucket[0]
sec_index
second_bucket[n-1]
*
*
* Add the number to the main vector if the vector length of the corresponding index= 1 until encountering an index which is length more than or equal 2

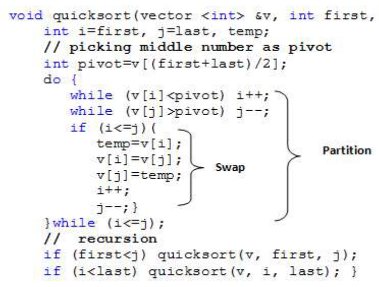
Figure 2. Non-Recursive MSD radix sort algorithm sorting process diagram

The sorting process begins by calculating the most significant digit of each number in the main vector, it then adds each number with the corresponding digit value i (i= 0…9) in ith bucket of the first_bucket. After the first iteration, all numbers are written to first_bucket vector. Each index in this structure is considered an individual vector which is sorted for the next most significant digit using the middle_bucket vector. The content of the middle_bucket is added to the second_bucket structure, and the process is repeated until all indices of the first_bucket are processed and the first_bucket is empty. During the next iteration, each index of the second_bucket is processed based on the third most significant digit and numbers are sent from second_bucket to first_bucket using the middle_bucket in reverse order.

During this sorting process, numbers can be either in the first_bucket or the second_bucket. At each iteration, the index length is checked. If length is 1(there is no need to use the middle_bucket) the index can be added to main vector until reaching an index that's length is greater than 1. This technique decreases size of the input for further iterations.

3.3 Quicksort Implementation

Quicksort is a state of the art sorting algorithm. Generally, the "*partition*" part of the Quicksort is implemented as an individual function (Cormen et al., 2009). Figure 3 represents our implementation of the Quicksort with where partition section included in the main procedure. While the selection of the pivot element is an important factor in performance tuning of the Quicksort, we use the middle element as the pivot in all our test cases. This is due the fact that any performance improvements in the Quicksort will also improve the performance of all of our hybrid algorithms.



```
void quicksort(vector <int> &v, int first, int last){
    int i=first, j=last, temp;
    // picking middle number as pivot
    int pivot=v[(first+last)/2];
    do {
       while (v[i]<pivot) i++;
       while (v[j]>pivot) j--;
       if (i<=j){
          temp=v[i];
          v[i]=v[j];
          v[j]=temp;
          i++;
          j--;}
    }while (i<=j);
    // recursion
    if (first<j) quicksort(v, first, j);
    if (i<last) quicksort(v, i, last); }
```

Figure 3. Quicksort algorithm implementation

3.4 Hybrid MSD Radix Sort Versions

The hybrid versions use a combination of MSD radix sort and Quicksort algorithms. MSD radix sort is deployed to create independent vectors, and Quicksort is applied to sort each of the generated vectors efficiently. As described earlier, digit count of the maximum number in the input vector determines the number of iterations needed in the complete MSD radix sort. When the digit count = d, the input will be sorted by MSD radix sort after d iterations.

In the hybrid case, MSD radix sort is applied to input a number of times, referred to as *step* ($1 \leq$ *step* $\leq$ d-1) to create a number of independent vectors. After each case, Quicksort is used to sort individual vectors without any dependencies. Two sequential and two parallel versions of the hybrid MSD radix sorts are implemented and are described in details next. The impact of applying different number of MSD steps before Quicksort on running time is analyzed.

3.4.1 Sequential Hybrid1 MSD Version (Hybrid1)

In the hybrid versions, the value of "*step*" is determined before the sorting process begins. "*step*" indicates the number of times MSD Radix sort is applied to the input before Quicksort. This version is exactly as described above using non-recursive MSD radix sort (Figure 2) and Quicksort.

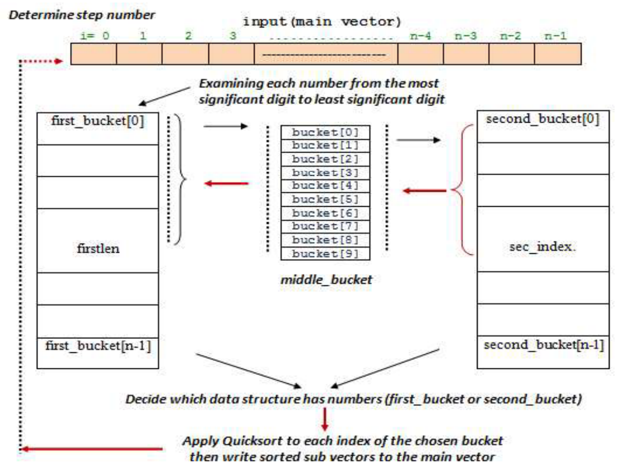

Determine step number
input (main vector)
i= 0 1 2 3 n-4 n-3 n-2 n-1
Examining each number from the most significant digit to least significant digit
first_bucket[0]
firstlen
first_bucket[n-1]
bucket[0]
bucket[1]
bucket[2]
bucket[3]
bucket[4]
bucket[5]
bucket[6]
bucket[7]
bucket[8]
bucket[9]
middle_bucket
second_bucket[0]
sec_index
second_bucket[n-1]
Decide which data structure has numbers (first_bucket or second_bucket)
Apply Quicksort to each index of the chosen bucket then write sorted sub vectors to the main vector

Figure 4. Sequential Hybrid1 MSD radix sort sorting process diagram

First, input is processed with MSD radix sort "step" times, after determining the individual vectors storing the input numbers according to their most significant digit orders, Quicksort is applied to sort them as illustrated in Figure 4. At the end, independently sorted vectors are written to the main vector in order.

3.4.2 Sequential Hybrid2 MSD Version (Hybrid 2)

Hybrid 2 version differs from Hybrid 1 in two ways. First, it uses an additional data structure, middle_bucket1 vector, in the first iteration of MSD radix sort to split the input based on the most significant digit, and uses the index (digit value) of the middle_bucket1 for subsequent iterations. Second, in Hybrid1, MSD radix sort is applied to the entire input followed by the Quicksort to sort individual indices (buckets). While in Hybrid 2, the sorting is completely applied to individual vectors corresponding to indices of the middle_bucket1 one-at-a-time. This process is equivalent to applying Hybrid1 MSD to each of the vectors in middle-bucket1. Therefore Hybrid1 MSD sort is used a maximum of ten times on shorter input values stored in buckets of middle-bucket1. This sorting diagram of Hybrid MSD Radix sort version2 is given at Figure 5.

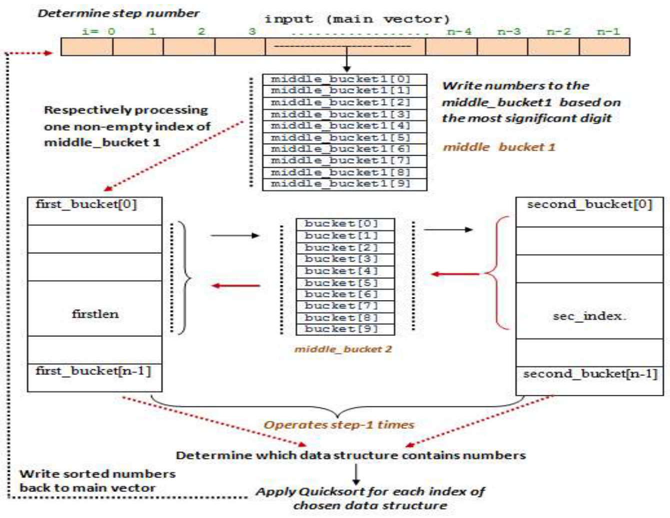
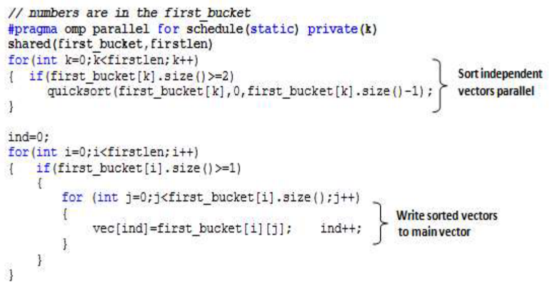


Figure 5. Sequential Hybrid2 MSD radix sort sorting process diagram

3.4.3 Parallel Hybrid1 MSD Radix Sort Version (P1)

Parallelizing sequential Hybrid1 MSD radix sort using OpenMP API is relatively straight forward. After application of the MSD Radix Sort the individual vectors generated by the first_bucket and the second_bucket respectively can be sorted independently and in parallel using the Quicksort presented in Section 3.3. The implementation for this case for the first_bucket is given at Figure 6-a.



```
// numbers are in the first_bucket
#pragma omp parallel for schedule(static) private(k)
shared(first_bucket,firstlen)
for(int k=0;k<firstlen;k++)
{   if(first_bucket[k].size()>=2)
       quicksort(first_bucket[k],0,first_bucket[k].size()-1);
}

ind=0;
for(int i=0;i<firstlen;i++)
{    if(first_bucket[i].size()>=1)
     {
         for (int j=0;j<first_bucket[i].size();j++)
         {
             vec[ind]=first_bucket[i][j];    ind++;
         }
     }
}
```

Sort independent vectors parallel
Write sorted vectors to main vector

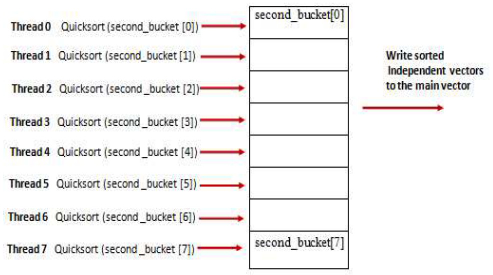

Thread 0 Quicksort (second_bucket [0])
Thread 1 Quicksort (second_bucket [1])
Thread 2 Quicksort (second_bucket [2])
Thread 3 Quicksort (second_bucket [3])
Thread 4 Quicksort (second_bucket [4])
Thread 5 Quicksort (second_bucket [5])
Thread 6 Quicksort (second_bucket [6])
Thread 7 Quicksort (second_bucket [7])
second_bucket[0]
second_bucket[7]
Write sorted Independent vectors to the main vector

Figure 6. Hybrid1 MSD radix sort parallel region implementation

Figure 7. Parallel threads representation

3.4.4 Parallel Hybrid2 MSD Radix Sort Version (P2)

Parallel Hybrid 2 version is implemented using the sorting structure of sequential Hybrid2 MSD radix sort described in Figure 5. The parallel section of this algorithm is again the application of Quicksort to independent buckets as demonstrated at Figure 6. Figure 7 demonstrates an example of parallel threads (in this case 8) applying Quicksort to different buckets in parallel in second_bucket. After applying Quicksort to each independent vector, the sorted vectors will be added to the main vector in order.

4. ANALYSIS

4.1 Coding and Execution Environment

The code is written in C++ and OpenMP. We have tested our implementation on a 12-core shared memory compute node of the 192-core cluster, Hydra, available in the Parallel Distributed System lab (PDS Lab 2013).

4.2 Sequential MSD Radix Sort Versions Analysis

Performance benchmarking of the recursive MSD radix sort, non-recursive MSD radix sort, and Quicksort is represented at Figure 8. In this case, input consists of 2 million randomly generated numbers. These numbers are of varying ranges 3D,4D.., 10D, where *"D"* represents the digit count of the input numbers.

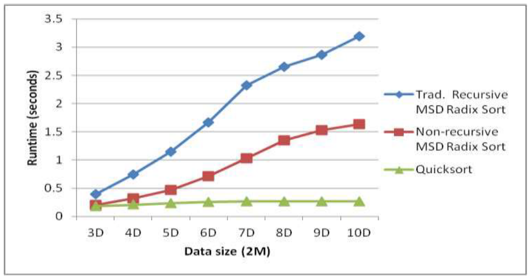


Figure 8. Performance comparison of sequential algorithms

Figure 8 shows that as the length of numbers increases from 3D to10D, runtime of MSD radix sort versions increases because their sorting process is related to the digit count of the input numbers. Moreover, we observe that Quicksort's performance is not significantly changed for different digit-count values making it an excellent choice for sorting the individual vectors produced after applying MSD for several steps.

In Figure 8, the non-recursive MSD radix sort version runs 1.74 to 2.56 times faster than the traditional recursive MSD radix sort.

4.3 Sequential Hybrid MSD Radix Sort Versions Analysis

We experimented by collecting runtime results of sequential Hybrid radix sort versions for 0.5M, 1M, 2M and 4M data sizes. We analyzed the data size for several distinct step numbers (S) to determine effects of various step values on performance. 'S' indicates how many times MSD radix sort is processed over input before Quicksort is applied. Runtime benchmarking of sequential Hybrid1 and Hybrid 2 versions is demonstrated at Figure 9-a. In this case, length of input is 4 million and it consists of at most 7D numbers. Sequential Hybrid2 MSD radix sort always performs better than sequential Hybrid1 MSD radix sort version.

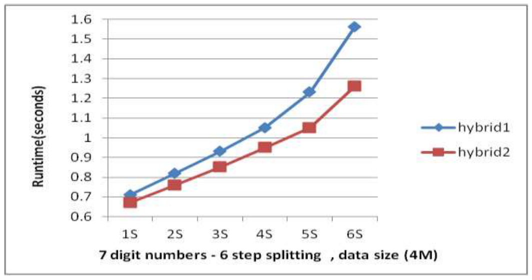


Figure 9-a. Sequential Hybrid versions

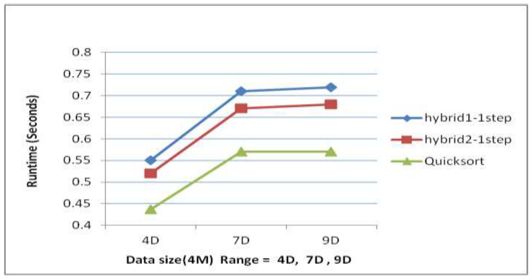
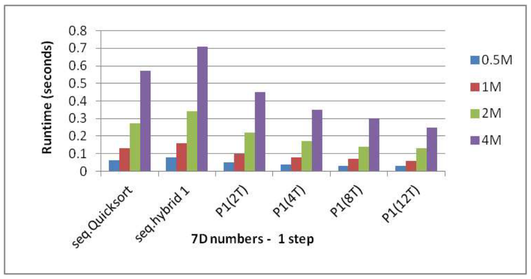


Figure 9-b. Quicksort and Hybrid versions

As we indicated in section 3.4.2, hybrid 2 MSD radix sort has an additional data structure to split input into buckets and work on a small portion of input. This property of Hybrid 2 yields performance gains over Hybrid1. From Figure 9-a, we can see that when step number (S) increases, overall performance of both hybrid versions decreases. Figure 9-b shows performance results of Quicksort over Hybrid MSD versions (for S=1) for different ranges.

4.4 Parallel Hybrid MSD Radix Sort Versions Analysis

Parallel Hybrid1 (P1) and Parallel Hybrid2 (P2) versions are analyzed using on different data size, range, and step values. Performance comparison of sequential and parallel versions of Hybrid MSD radix sort is presented at Figure 10 (a and b). In this figure, step = 1 and digit count is at most 7D. As the figure shows, both parallel hybrid versions perform faster than their sequential counter parts. For instance; P1 (2T: two threads) is 1.58 times and P1 (12T) 2.63 times faster than its sequential version. Also, P2 (2T) is 1.75 times and P2 (12T) 3.5 times faster than its sequential version.



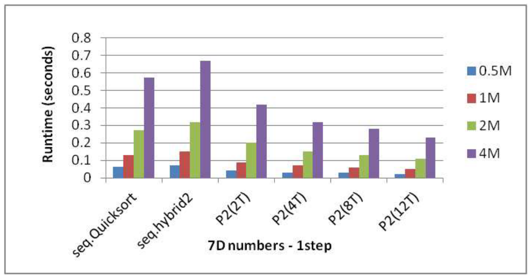
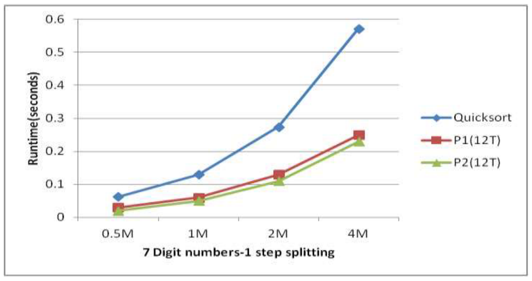


Figure 10-a. Sequential and parallel Hybrid1

Figure 10-b. Sequential and parallel Hybrid2

The impact of step number 'S' on performance is also analyzed for the parallel versions. Figures 11 (a and b) and 12 (a and b) show that when step number increases from 1 to 2, the overall performance decreases. This is due to the fact that the MSD steps are applied sequentially. As we increase the number of sequential steps that the MSD sort is applied, the amount of sequential work increases the overall execution time. In this implementation, S=1 produces the best overall execution time.



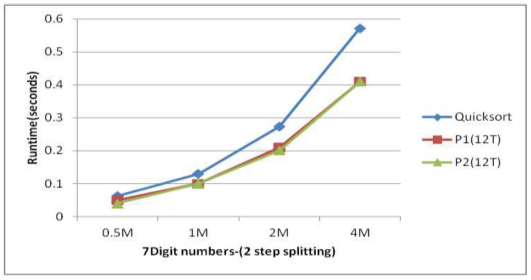
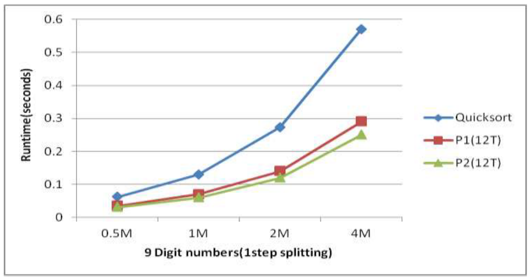


Figure 11-a. Quicksort P1&P2 MSD (7D-1step)

Figure 11-b. Quicksort P1&P2 MSD (7D-2step)



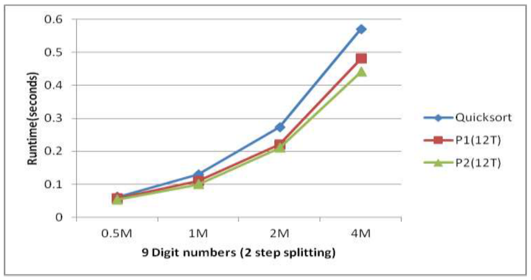


Figure 12-a. Quicksort P1&P2 MSD (9D-1step)

Figure 12-b. Quicksort P1&P2 MSD (9D-2step)

In all cases presented above, parallel Hybrid2 outperforms Parallel Hybrid1 and they both significantly outperform the very efficient Quicksort algorithm. For example, P2 performs 2.28 times significantly faster than Quicksort with 4M data size, Figure 12-a.

4.5 Worst Case Memory Analysis

The hybrid sorts use additional memory compared to traditional Quicksort and in-place radix sorts. The worst case memory consumption in our implementation for the sequential and parallel Hybrid1 MSD and Hybrid2 MSD sorts is 4n + constant and 5n+ constant respectively, where constant refers to the memory for the bucket structure and book-keeping and n is the size of input vector. The worst case size for each of the bucket structures is n+10+B, where 10 refers to the 10 header locations for each bucket vector and n is the number elements being stored in the 10 buckets.

5. CONCLUSION

An efficient sequential non-recursive MSD radix sort version is implemented as an alternative version to the recursive one. Results presented, show the non-recursive MSD radix sort performs remarkably better than the traditional recursive MSD radix sort. Moreover, the non-recursive MSD radix sort and the efficient Quicksort methods are used to implement two versions of sequential and their corresponding parallel Hybrid MSD radix sort versions. Performance benchmark of the sequential and parallel Hybrid MSD radix sort algorithms demonstrates that both parallel hybrid versions perform faster than their sequential counterparts and that they yield significant performance gain over Quicksort.

REFERENCES

Book

Chang, SK., 2003. *Data Structures and Algorithms*, World Scientific Publishing. Singapore.

Cormen, T. et al ,2009. *Introduction to Algorithms* 3rd edition. The MIT Press, London, England.

Journal

Mollick, E., 2006. Establishing Moore's Law. *IEEE Annals Hist. Computing*, Vol. 28, No.3, pp. 62-75.

Ramey, R., 1992. The Postman's Sort. *C Users Journal*, Vol.10, pp. 59-76.

Conference paper or contributed volume

Cheng, SW., 2003. ARBITRARY LONG DIGIT INTEGER SORTER HW/SW CO-DESIGN. *Proceedings of the ASP-DAC.* Taiwan, Taipei, pp- 538 – 543.

Han,Y. and Thorup, M., 2002. Integer Sorting in O($n\sqrt{\log\log n}$) Expected Time and Linear Space. *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science.* pp-135-144.

Kim , KJ. et al, 2011. Parallel Quick Sort Algorithms Analysis Using OpenMP 3.0 in Embedded System. *Proceedings of 11th International Conference on Control, Automation and Systems.* Gyeonggi-do, Korea, pp. 757-761

Ma,H. et al, 2009. Barrier Optimization for OpenMP Program. *Proceedings of 10th ACIS International Conference on Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing*. Daegu, pp. 495-500.

OpenMP Architecture Review Board, 2008. OpenMP Application Program Interface Version 3.0.

PDS Lab 2013. Available from < http://pds.ucdenver.edu/>

Pigeonhole Sort, Available from < http://xlinux.nist.gov/dads//HTML/pigeonholeSort.html>

Sato, M., 2002. OpenMP: Parallel Programming API for Shared Memory Multiprocessors and On-chip Multiprocessors. *Proceedings of 15th International Symposium System Synthesis*. Kyoto, Japan, pp.109-111.

Zhang, Y., 2008. Radix Plus Length Based Insert Sort. *Proceedings of Seventh IEEE/ACIS International Conference on Computer and Information Science.* Portland. pp. 61-66.