Parallel Sorting Algorithm by Median-Median

중위수의 중위수에 의한 병렬 분류 알고리즘

Yong Sik Min*
민 용 식*

Abstract

This paper presents a parallel sorting algorithm suitable for the SIMD multiprocessor. The algorithm finds pivots for partitioning the data into ordered subsets. The data can be evenly distributed to be sorted since it uses the prob ability theory. For n data elements to be sorted on p processors, when $n \geq p^2$, the algorithm is shown to be asymptotically optimal. In practice, sorting 8 million data items on 64 processors achieved a 48.43-fold speedup, while the PSRS required a 44.4-fold speedup. On a variety of shared and distributed memory machines, the algorithm achieved better than half-linear speedups.

요 약

본 논문은 SIMD 병렬 처리 컴퓨터에 적합한 병렬 분류 알고리즘을 제시키 위해서, 다음과 같이 수행이 된다. 첫째, 비순서화된 데이타 집합을 p개의 프로세서로 할당시킨후에 순차적 quicksort로 분류한다. 그 다음으로, 분류된 각 프로세서의 중위수값을 구한다음 이 값에 의해서 각 프로세서에 데이타 값을 할당시킨다. 각 프로세서에 할당된 데이타가 정확하게 분배가 되도록 중위수의 중위수 값을 구해서 각 프로세서에 적합한 데이타를 다시 할당 시키게 된다. 이때 각 프로세서가 지닌 데이타의 수는 확률이론을 이용하였다. 마지막으로, 각 프로세서에 할당된 데이타를 순차적 quicksort로 분류하면 된다. 여기서 분류될 데이타 n가 $n \geq p^2$일때 본 알고리즘은 최적이 되게됨을 볼 수가 있다. 실제적 구현에 있어서, 64개 프로세서를 이용해서 8백만개의 데이타를 분류할때 PSRS 방법의 speedup은 44.4인 반면에 본 알고리즘은 48.43이 된다. 즉, 다양한 공용과 분산 기억장치 기계에 관해서, 본 알고리즘의 speedup은 거의 절반 이상의 선형시간으로서 성취가 됨을 볼 수가 있다.

I. Introduction

With the growing number of areas in which computers are being used, there is an increasing demand for more computing power than today's machines can deliver. For many applications, extremely fast computers are being sought to process enormous quantities of data in reasonable amounts of time. One means of attaining very high computational speeds is to use a parallel computer : that is, one that has several processing units or processors [1].

Sorting is one of the most studied problems in computer science, because of its theoretical interest and practical importance. With the advent of parallel processing, parallel sorting has become an important area for algorithm research [2]. Most parallel sorts suitable for multiprocessor computers can be placed into one of two rough categories : (1) merge-based sorts, and (2) partition-based

*Hoseo Univ, Dept. of Computer Science
접수일자 : 1994년 8월 30일

sorts [1, 2]. Partition-based sorts consist of two phases : (1) partitioning the data set into smaller subsets such that all elements in one subset are no greater than any element in another, and (2) sorting each subset in parallel.

We now consider parallel sorting algorithms that were previously presented. One of these, Parallel Quicksort, has been a popular choice for research [1, 2, 4, 8, 9]. The basic result is that, given a sequence S of n distinct elements to be sorted, Quicksort starts by finding the median m of S. Element m is now placed in position n/2 of the sorted sequence. Then S is partitioned into two subsequences, S1 and S2, of elements smaller than and larger than m, respectively. The two subsequences, S1 and S2, are now executed simultaneously, since each of the two subproblems, S1 and S2, has the same structure as the original problem S and contains, at most, half as many points as S.

A similar effect happens to Evans and Yousif's two-way merge-based parallel sort [2, 3], as little parallelisms can be exploited in the last few phases of merging. Francis and Mathieson have noted this problem and proposed a Parallel Mergesort(PMS) which evenly partitions data to be merged among any number of processors [2, 7]. This sorting algorithm is merge-based, however : consequently, it involves too many data movements. Quinn has implemented a combination of Quicksort and Mergesort, Quickmerge, significantly reducing the amount of data movement [6]. Its execution time, however, is unstable in the sense that the pivots (or dividers) selected are not guaranteed to divide the data to be sorted into ordered subsets reasonably evenly. In the worst case, a single processor may have to perform a Mergesort on nearly all the data in the last phases, which makes linear speedup impossible.

To overcome the difficulty of data distribution, Huang and Chow proposed extracting a random sampling from the data and using the order information of the sample to help the partitioning. Their method is called Parallel Sorting by Sampling (PSS) [5]. To distribute the data evenly, Shi and Schaeffer proposed Parallel Sorting by Regular Sampling(PSRS) [12]. This method finds pivots for partitioning the data into ordered subsets of approximately equal size by using a regular sample from sorted sublists of the data.

This paper describes a parallel sorting algorithm suitable for a variety of multiprocessor architectures. Parallel Sorting by Median-Median (PSMM) finds pivots for partitioning the data into ordered subsets. The method evenly distributes the data to be sorted since it uses probability theory. In practice, when PSMM runs on the MasPar machine, sorting of 8 million data items on 64 processors achieved a 48.43-fold speedup, while the PSRS required a 44.4-fold speedup.

Ⅱ. Parallel Sorting by Median-Median

Let the data set to be sorted on a p-processor EREW-PRAM multiprocessor be denoted by X and the size of X be n. Let X[i : j] be {X[i], X[i+1],...,X[j]}, while $0 \le i \le j < n$. For simplicity in the analysis of the algorithm, we assume X[i] = X[j], where i = j. Also, p/2 refers to ⌊p/2⌋.

Parallel Sorting by Median-Median (PSMM) has three phases. The psuedo-code for the algorithm is shown in Fig. 1. In the first phase, each of the p processors sorts a contiguous list of size w = ⌊n/p⌋ using sequential Quicksort. More precisely, each processor i($1 \le i \le p$) sorts a list X [(i-1)w : iw-1] with Quicksort. After this phase, each processor has acquired data items to be sorted. In the second phase, we choose the median value of each processor. The set Y is the median value yielding a list Y[1], Y[2],...,Y[p]. This median value of each processor is a pivot element. Choosing the median value of each processor is very simple. That location is a half-position of each processor. The partitioning of X is accomplished

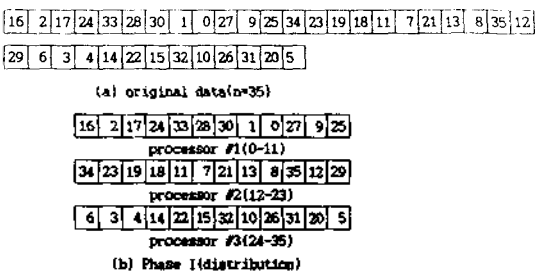
as follows. Each processor finds where each of p-1's pivot divides its list, using a binary search. More precisely, each processor i($1 \le i \le p$) finds the index of the largest element no larger than the jth pivot, $j = 1, 2, \ldots, p-1$. At this point, we calculate the size of the sublist each processor will have. If the size of the sublist does not satisfy the range around 80% to 120%, we reapply the sublist of the processor that was not satisfied using the same method described above in the second step until that list satisfies the condition. After doing this, all processors are synchronized. At this point, each of the p sorted lists of X has been divided into p sorted sublists with the property that every item in every list's ith sorted sublist is greater than any item in any list's (i-1) th sorted sublist for $2 \le i \le p$.

In the last phase, each processor i($1 \le i \le p$) performs a sequential quicksort to require all the ith sorted sublists of the p lists. After this, all processors are synchronized and X is sorted.

Fig. 2 illustrates how the PSMM works for $n = 36$ and $p = 3$. In phase 1, each processor is assigned $w = n/p = 12$ contiguous elements to sort. In the second phase, we choose the median value of each processor. We will call it a pivot element. In Fig. 2, the pivot elements are {24, 19, 15}. From this, we require $p-1 = 2$ pivots elements in each processor using a binary search, resulting in Fig. 2(e). However, at this time, we check the probability that each processor has. The median of the first processor is {2, 6, 11}. From this, we require the $p-1 = 2$ pivots (the total pivot is 4) using a binary search. We also check the probability. At this time, in order to be satisfied, we sort them using a sequential Quicksort algorithm. In Fig. 2(h), the first processor has 14 elements, a second processor has 10 elements, and a third has 12 elements. Once each processor receives its portion of data from the other processor, it can merge the results into the final sorted array. In this final partition phase, this method requires a bound of $2\lfloor n/p \rfloor$. In this paper, it is shown that the PSMM tries to achieve almost a load balancing in practice using the probability condition.

```
    procedure PSMM(X, n, p)
   /* X[0 n-1]  array to be sorted, n: the size of array,
      p: the number of processors   */
    begin
    /* phase I: divide the array into p contigous list and sort
                each in parallel */
    size=n/p
(1) for i=0 to p-1 do in parallel
      start= i * size
      end= start + size -1
      if(end >= n) then end = n - 1
      quicksort(X, start, end)
    endfor
    /* phase II:
       (a) calculate the median value of each processor
       (b) divide the array by binary search and
       (c) apply the proper size by probability
       (d) if not satisfied (c), reapply to (b)
              in only unsatisfied processor */
(2) for i=0 to p-1 do in parallel
      Y[i]=X[(start+end)/2]
    endfor
    /* divide the array by binary search */
(3) for j=0 to p-1 do in parallel
(4)   for i=0 to p-1
        z[i]=the median value found in binary search using Y
        subsize[j]=z[i-1]+i
      endfor
    endfor
    /* in parallel, count the size of each processor and
       bucksize's value is initially zero */
(5) for i=0 to p-1 do
   (5.a) for j=0 to p-1 do
           bucksize[j]=bucksize[j]+subsize[j+i]-subsize[j]
         endfor
   (5.b) if(the size of bucksize[i] is notsatisfied around 80%
              to 120%)
           then  we apply only the processor which is not satisfied
                   in conition in (2)
    endfor
    /* phase III: sort using sequential quicksort */
(6) for i=0 to p-1 do in parallel
      start=z[i-1]+i
      end=z[i]
      quicksort(X, start, end)
    endfor
   end
```

Fig. 1 pseudo code for PSMM



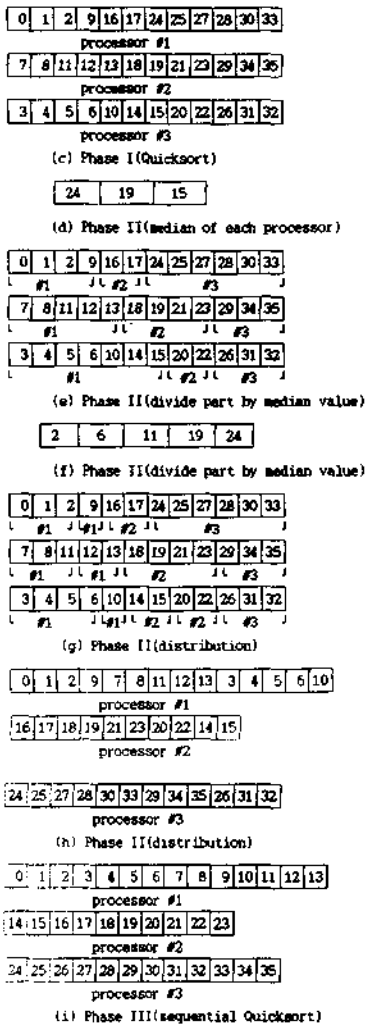


Fig. 2 PSMM example

Ⅲ. Complexity Analysis

In phase one and part of phase two of the PSMM, all processors have roughly the same amount of work to do. In phase two (some part) and phase three, it is not obvious how evenly the work is divided because this depends on how well the data has been partitioned.

Theorem 1. The time complexity of Quicksort requires O(n log n) in the average case. [1]

Theorem 2. The time complexity of binary search problem requires at most O(log n) [1].

Lemma 1. In phase three of the PSMM, each processor merges less than 2w elements. [1]

Proof. Consider any processor i, where $1 \le i \le p$. There are three cases :

(1) In case of $i=1$. All the data to be merged by processor 1 must be $\le y_1$. Since there are $p^2-p-p/2$ elements which are $> y_1$, there are at least $(p^2-p-p/2)w/p$ elements of X which are $> y_1$. In other words, there are at most $n-(p^2-p-p/2)w/p=(p+p/2)w/p <$ 2w elements of X which are $\le y_1$.

(2) In case of $i=p$. All the data to be merged by processor p must be $> y_{p-1}$. There are $(p-2)p+p/2$ elements which are $\le y_{p-1}$. That is there are at least $(p^2-p-p/2)w/p$ elements of X which are $\le y_1$, or there are at most $n-((p^2-2p-p/2)w/p,2w$ elements of X which are $> y_{p-1}$.

(3) In case of $1 < i < p$. All data to be merged by processor i must be $> y_{i-1}$ and $\le y_i$. There are $(i-2)p+p/2$ elements which are $\le y_{i-1}$, implying $lb=((i-2)p+p/2)w/p$ elements of X. On the other hand, there are $(p-i)$ $p-p/2$ elements which are $> y_i$. As well, there are w/p-1 elements that fall between zi and the next highest element in the median-median. Since the size of X is n, there are at most 2w-w/p+1<2w elements of X for processor i to merge.

In conclusion No processor merges more than $2w=2n/p$ elements in the last phase of the PSMM. If p^2 does not divide n evenly, it is easy to prove that no processor merges more than 2 n/p elements.

What is the time complexity of PSMM? The analysis for phase one is easy. (see Theorem 3)

Theorem 3. In the phase one of PSMM, the time complexity is O(w log w).

Proof. The initial Quicksort takes O(w log w), representing the time consumed by each processor to sort $w(=n/p)$ data using a sequential quicksort. (see theorem 1)

Theorem 4. In the PSMM algorithm, the time complexity is $O(w \log w + p^2 \log w)$.

Proof. The analysis is presented by phase.

(1) In step 2 of phase II, it takes O(1).

(2) In step 3 of phase II, it requires $O(1*(p \log w))$ since step 4 of phase II requires $O(p \log w)$.

That is, steps 2 and 4 of phase II require $O(p^2 \log w)$.

(3) In step 5 of phase II, while (5.a) requires O(p) and (5.b) requires $O(p \log w)$ that applies recursively each processor in the worst case, it needs $O(p*p \log w) = O(p^2 \log w)$.

(4) Step 6 of phase III requires $O(2w \log w)$ since the size of data to be merged by any processor is always less than 2w.

The summation of the time of all three phases gives a time complexity for the PSMM of $O(w \log w + p^2 \log w + 2w \log w)$, which is asymptotic to $O(w \log w) = O((n/p) \log n)$ when $n \geq p^2$. ■

The bound on the size of the data to be merged in the PSMM is an important difference which other partitioned-based sorts, such as the PSRS, do have while Quickmerge and PSS do not. Theoretically, the PSMM is optimal when $n \geq p^2$,

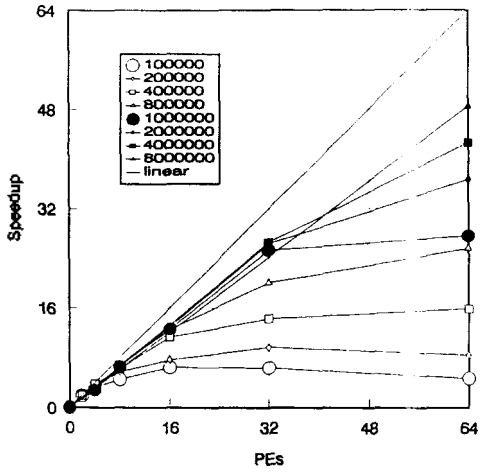


Fig. 3 Speedup of PSRS

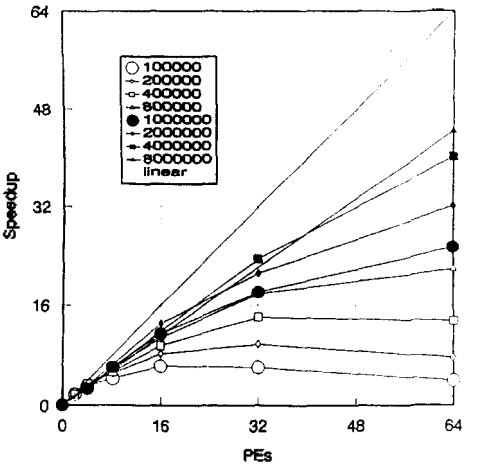


Fig. 4 Dynamic Huffman Method

regardless of the distribution of the original data. Two final points need to be addressed. First, the algorithm and its analysis are based on there being no duplicate elements in the list to be sorted. Second, it is possible to use more than p (p-1) elements in this method for choosing the median.

Ⅳ. Experimental Results

To implement the PSMM on the MasPar using randomly generated data, 32-bit integers with various distributions were tested. No tests were made for duplicate elements, of which there were undoubtedly a few. The size of the array to be sorted ranged from 0.1 million to 8 million elements. Experiments were done using 2, 4, 8, 16, 32, 64 processors on the MasPar machine. Each data point presented in this section was obtained as the average of 20 program executions, each on a different set of test data.

The experiement used the sequential Quicksort time from the PSRS [2], which provides the optimal sequential sort. It needs the speedups which evaluate a new parallel algorithm for some problem. Speedup is defined as the time required

to solve the problem; that is, the time elapsed from the moment the algorithm starts to the moment it terminates [1].

It is reasonable to assume that the time of sorting n integers using the improved Quicksort on one PE is

$$t_{pe}(n) = c\, n \log n,$$

where c is a constant independant of size n. Sequential times for lists of more than 0.2 million elements were calculated using the formular [2]:

$$t_{pe}(n) = \frac{n \log n}{100{,}000 \log 100{,}000} \times t_{pe}(100{,}000),$$

where 0.4 million $\leq n \leq 8$ million and $t_{pe}(100{,}000) = 6.63$ seconds. Note that if one used this formula to compute $t_{pe}(200{,}000)$, the result is almost a perfect match with the corresponding experimental time.

Table 1. Sorting times of PSMM

n \ PE	1	2	4	8	16	32	64
100,000	6.63	3.48	2.04	1.48	1.04	1.05	1.44
200,000	14.00	7.25	3.86	2.48	1.84	1.45	1.68
400,000	19.71	15.19	7.91	4.86	2.62	2.08	1.88
800,000	62.62	–	20.28	9.62	4.99	3.11	2.45
1,000,000	79.56	–	28.53	12.18	6.28	3.14	2.89
2,000,000	167.56	–	–	–	12.86	6.34	4.56
4,000,000	350.17	–	–	–	–	13.17	8.21
8,000,000	732.28	–	–	–	–	–	15.12

Table 2. Sorting times of PSRS

n \ PE	1	2	4	8	16	32	64
100,000	6.63	3.86	2.11	1.56	1.06	1.09	1.66
200,000	14.00	8.08	4.20	2.74	1.71	1.43	1.83
400,000	19.71	16.77	8.60	5.58	3.13	2.11	2.20
800,000	62.62	–	21.51	10.67	5.84	3.50	2.86
1,000,000	79.56	–	29.88	13.13	6.97	4.38	3.12
2,000,000	167.56	–	–	–	13.73	7.87	5.19
4,000,000	350.17	–	–	–	–	14.83	8.70
8,000,000	732.28	–	–	–	–	–	16.49

Table 3. RDFAs of PSMM

n \ PE	2	4	8	16	32	64
100,000	1.003	1.025	1.089	1.245	1.368	
200,000	1.002	1.010	1.097	1.252	1.383	
400,000	1.004	1.108	1.212	1.315	1.443	–
800,000	–	1.132	1.245	1.353	1.463	1.072
1,000,000	–	2.001	2.002	1.896	1.902	1.945
2,000,000		–	–	1.578	1.998	2.005
4,000,000			–	–	2.065	2.076
8,000,000	–	–	–	–	–	2.047

Table 4. RDFAs of PSMM

n \ PE	2	4	8	16	32	64
100,000	1.001	1.008	1.030	1.074	–	
200,000	1.002	1.003	1.012	1.032	1.043	–
400,000	1.001	1.002	1.008	1.017	1.044	–
800,000		1.002	1.005	1.017	1.026	1.062
1,000,000	–	1.001	1.004	1.010	1.021	1.047
2,000,000	–	–	–	1.009	1.016	1.045
4,000,000		–	–	–	1.001	1.026
8,000,000	–	–		–	–	1.017

Table I shows the time required to sort using the PSMM, and Fig. 3 plots the speedups achieved. As the problem size increases, the task granularity increases, offsetting the overheads of the algorithms and resulting in better speedups. Sorting 8 million items with 64 processors gave a 48.43-fold speedup.

For comparison purposes, the PMS, the Quickmerge, the PSS, and the PSRS were implemented. For brevity, only the PSRS results are shown, as they are significantly better than those of the PMS and the PSS. Table II shows the time required to sort the elements using the PSRS, and Fig. 4 plots the speedups achieved.

The RDFA (Relative Derivation of the size of the largest partition From the Average size) measures for the PSMM are shown in Table III and the PSRS in Table IV. For the PSMM, the data clearly shows that, as the number of processors

increases, so does the RDFA metric. However, the PSRS is the claim that the data is evenly partitioned ; so, the RDFA is almost constant.

From the data obtained, a few observations can be made on the performance of the PSMM.

(1) As the number of PEs increases, the speedups are good. For example, when sorting 8 million 32-bit integers with 64 PEs, the PSMM has a speedup of a 48.43-fold while PSRS has a 44.4-fold speedup.

(2) Although Table IV shows that the PSRS RDFAs are consistently close to 1.0, the PSMM RDFAs are not close to 1.0, but are almost near to 1.0.

In general, the speedup relationships of the five algorithms implememted are as follows :

PMS ≤ Quickmerge ≤ PSS ≤ PSRS ≤ PSMM.

The speedups of the PMS are found to be rather poor. We cannot confirm Francis and Masthieson's claim that the PMS has a linear speedup [2]. The PSMM has been implemented on the shared memory MasPar machine. A 48.43-fold speedup was achieved sorting 8 million data items on 64 processors. The sorting was done in each processor's local memory, while the global memory was used to communicate data.

V. Conclusion

Long memory latency and the overhead of scheduling and synchronization are two critical factors that greatly affect the speedup of a parallel algorithm on the present parallel machine [1, 2]. Parallel Sorting by Median-Median is intended to minimize both. Experiments on the MasPar machine (SIMD machine) make it clear that good speedups for parallel sorting is indeed achievable. The results reported here are quite encouraging, considering that sorting is generally believed a hard problem to parallelize. In the future, we think about distributing data evenly in the process of sorting.

Acknowledgements

Dr. Zheng supplies me with constructive suggestions for improving this paper. And LSU provided me with using MasPar machine.

References

1. Selim G. Akl. Parallel Sorting Algorithms, academic press, 1985
2. Hanmao Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," Journal of Distributed and Parallel computing, pp. 361 - 372, 1992
3. Evan, D. J. and Yousif, N. Y., "The Parallel Neighbor Sort for Shared Memory Multiprocessor," IEEE trans. compu., 37, 12(1988), pp. 1619-1626
4. Evans, D. J. and Yousif, N. Y., "Analysis of the Performance of the Parallel Quicksort Method," Bit 25(1985), pp. 106-112
5. Huang, J. S. and Chow, Y. C., "Parallel Sorting and Data Partitioning by Sampling," 7th int'l computer software and application conference, 1983, pp. 627-631
6. Quinn, M. J., "Parallel Sorting Algorithms for Tightly Coupled Multiprocessor," parallel comput., 6(1988), pp. 349-357
7. Francis, R. S., and Mathieson, I. D., "A Benchmark Parallel Sort for Shared Memory Multiprocessors," IEEE Trans. Comput., 37, 12 (1988)
8. Shim, J. H and Hong, S. S., "The Optimal Algorithm without a Memory Conflicts," Journal of the Korea Information Science, Vol. 16, No. 1, pp. 28-37, Feb. 1989
9. Han, T. D., "The Design and Analysis of Parallel Algorithm," Proceedings of Parallel Processing System, Vol 1, No. 1, Dec. 1989

▲**Yong Sik Min**(Regular Member)

1981년 2월 : Dept. of Computer Science, Kwangwoon Univ.(B. S.)

1984년 2월 : Dept. of Computer Science, Kwangwoon Univ.(M. S.)

1991년 2월 : Dept. of Computer Science, Kwangwoon Univ.(Ph. D)

1984년 3월 ~ 1987년 2월 : Full-time lecturer, Songwon Junior College Dept. of Computer Science

1987년 3월 : present : Associate Professor, Hoseo Univ. Dept. of Computer Science

1993년 8월 ~ 1994년 8월 : Visiting Professor, Lousiana State Univ. Dept. of Computer Science