Relaxed min-max 힙을 병합하는 병렬 알고리즘

민 용 식†

요 약

본 논문에서는 relaxed min-max heap을 병합시키기 위하여 새로운 자료구조인 개선된 relaxed min-max-pair 힙을 제시함과 동시에, 두개의 relaxed min-max 힙, 즉 크기가 n인 relaxed min-max nheap과 크기가 k인 relaxed min-max kheap으로 구성된 우선 순위 큐를 병합시키기 위한 병렬 알고리즘을 제시하고자 한다. 본 논문에서는 [9]의 방법으로부터 relaxed min-max 힙을 병합 시키기 위해서 이용된 blossomed tree와 lazying 방법을 제거하여도 병합되는 새로운 방법을 제시하였다. 결과적으로 본 논문에 제시된 방법은 $\max(2^{h-1}, \lceil (m+1)/4 \rceil)$개의 프로세서를 이용할 경우, 시간 복잡도가 $O(\log(\log(n/k)) \times \log(k))$임을 볼 수가 있다. 그리고 크기가 서로 다른 두 개의 relaxed min-max heap으로 구성된 8백만개의 데이터를 병합시키기 위해서, MasPar 머신에서 64개의 프로세서를 이용하여 실행시킨 결과 35.205의 Speedup을 얻었다.

A Parallel Algorithm for Merging Relaxed Min-Max Heaps

Yong-Sik Min†

ABSTRACT

This paper presents a data structure that implements a mergable double-ended priority queue : namely an improved relaxed min-max-pair heap. By means of this new data structure, we suggest a parallel algorithm to merge priority queues organized in two relaxed heaps of different sizes, n and k, respectively. This new data-structure eliminates the blossomed tree and the lazying method used to merge the relaxed min-max heaps in [9]. As a result, employing $\max(2^{h-1}, \lceil (m+1)/4 \rceil)$ processors, this algorithm requires $O(\log(n/k) \times \log(n))$ time. Also, on the MasPar machine, this method achieves a 35.205-fold speedup with 64 processors to merge 8 million data items which consist of two relaxed min-max heaps of different sizes.

1. Introduction

Priority queues have traditionally been used for applications such as branch-and-bound algorithms, discrete-event simulation, shortest path algorithms, multiprocessor scheduling and sorting. A priority queue is an abstract data structure that easily allows deletion of the highest priority item and insertion of new items. A heap is a complete binary tree such that the priority of the item at each node is higher than that of the items of its children. A heap provides an optimal implementation of a priority queue on uniprocessor computers. Deletion of the highest priority item and insertion of a new item can each be accomplished in $O(\log n)$ time on an n-item

† 정 회 원 : 호서대학교 컴퓨터학부
논문접수 : 1997년 11월 4일, 심사완료 : 1998년 3월 14일

heap [4].

The merge operation of a heap is supported by the leftist heaps proposed by Crane, and by the binomial queues proposed by Vuillemen. Other priority queue implementations include the skew heap, the Fibonacci heap, the relaxed heap, and the pairing heap. Recently Olariu et al. [3] suggests a double-ended priority queue implementation called the min-max-pair heap, which supports sublinear time merging. Y. Ding and M.A. Weiss [9] gave a priority queue implementation called the relaxed min-max heap, which supports all the priority operations. The key idea of this method is that, by properly relaxing the order restrictions for min-max heaps, it can merge two regulary sized min-max heaps.

To make efficient use of a parallel computer for priority-queue-based applications, one needs a parallel version of the priority queue, which would allow multiple concurrent insertions and deletions. There have been various parallelization efforts on the heap operations for a shared-memory parallel computer [6]. Rao and Kumar proposed a practical scheme that allows $O(\log n)$ processors to be active simultaneously on a heap. Biswas and Browne present a scheme that can keep $O(\log n)$ processors active on a heap. Jones also presents a scheme to keep a similar number of processors active, but the heap access time in his scheme is $O(\log n)$ in only an amortized sense. Nao and Zhang [8] propose a parallel heap running in $O(A/p + \log n)$ time with $p \le O(n/\log(n))$ processors on an EREW-PRAM.

This paper provides a new data structure — the improved relaxed min-max-pair heap — that efficiently supports the merge operation of the priority queue. This new data structure eliminates the blossomed tree and lazy merging method used to merge relaxed min-max heaps in [9], and proposes a parallel algorithm to efficiently merge two relaxed heaps of different sizes. Using $\max(2^{i-1}, \lceil (m+1)/4 \rceil)$'s processors, the new algorithm requires $O(\log(n/k) \times \log(n))$ time. We achieved a 35.205-fold speedup, with 8 million data items on the 64 processors MasPar.

The rest of this paper is organized as follows. Section II describes the parallel algorithm for merging two relaxed min-max heaps of size n and k. Section III discusses the results of this method and Section IV gives conclusions.

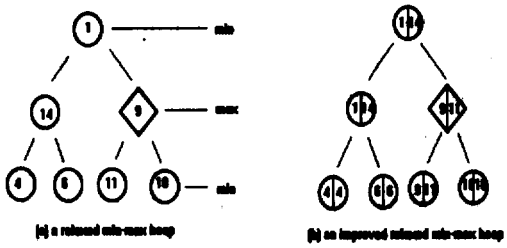
2. Merging Relaxed Heaps in Parallel

We first give definitions related to merging the relaxed min-max heaps and then define a new data structure; namely, the improved relaxed min-max-pair heaps. Here, the relaxed min-max nheap is regarded as an nheap and the relaxed min-max kheap as a kheap. We define a perfect heap as a heap with $2^i - 1$ elements, in which all leaves are on the same level; otherwise, the heap is non-perfect. The relaxed min-max pheap is rooted at p, similar to the subheap rooted at p. We call it a pheap. Let the size(heap) refer to the number of elements the heap contains, and the height be defined as $\lfloor \log(size(heap)) \rfloor$. We introduce a function h(heap) which returns the height of the heap. We will define slots of those leaf positions in nheap which are to be filled by merging processes [7].

A relaxed min-max tree T is a min-max labeled tree such that, for any node v in T, neither children(v) nor grandchildren(v) may contain more than one relaxed nodes. Clearly, any subtree of a relaxed min-max tree is also a relaxed min-max tree. This example is shown in Fig. 1(a). A min-max-pair heap is a binary tree H featuring the relaxed heap-shape

property, such that every node in H has two fields (called the min field and the max field) and such that H has a min-max ordering. For every i(1≤i≤n), the value is stored in the min field of H[i]; similarly, the value stored in the max field of H[i] is the largest key stored in the subtree of H rooted at H[i].

We describe the new improved relaxed min-max-pair heap's(T) property as follows: (1) T has the heap-shape; (2) T is a relaxed min-max tree; and (3) T is the min-max-pair ordered. But, in order to give the min-max value of each level, we proceed from the root to leaves. And so, in the present level and the following level, we choose the min-max-pair. When the node chosen becomes the root node, its child nodes must be processed until the leaf nodes have been reached according to the relaxed min-max-heap. The improved relaxed min-max-pair heap is developed and used only to merge relaxed min-max heaps. An example of the new, improved relaxed min-max-pair heap is shown in Fig. 1(b).



(Fig. 1) Comparison between a relaxed min-max heap and an improved heap.

Next, we consider a merging method conce -rning the relaxed min-max heaps. For the sake of clarity, we first show how to merge two perfect relaxed min-max heaps of an equal size. We show how to merge two relaxed min -max heaps of different sizes. In the case of merging two perfect relaxed min-max heaps of an equal size, however, this method's process is the same as the sequential one[7] which tak es two heaps, an nheap and a kheap, each of s ize k(=n), and produces a new heap with 2k e lements. With a single processor, it runs in O (log k) time.

We now consider a simple case of inserting a relaxed min-max heap of k elements, kheap, into a relaxed min-max heap of n elements, nheap. Without loss of generality, assume that k 〈 n. We proceed with three phases of merging the relaxed min-max heaps as follows. In the first phase, determining the level of the root of slots that have k by the merging process in nheap, we have to allocate the nodes of the level to each processor. Second, the pheap (that is, the subheap of nheap that is allocated in the nheap) and the k'heap (that is, the subheap of kheap that is allocated in the kheap) are being merged. At this point, the merged new subheaps(pheap +k'heap) satisfies the relaxed min-max-pair heap's condition. In the last phase, we connect the newly merged heap to a nheap, which exists in the shared memory, and construct the relaxed min-max heap condition on the nheap.

2.1 Level-Find Algorithm

In the first phase of the merging process, in order to determine the location p's node in each processor, we use the level of nheap; then, we classify nheap as a perfect heap or nonperfect heap.

a. A Perfect Heap

In the process of selecting the location p of the node, we must fill out a characteristic of the heap from the leftmost node of nheap in order to fill out the node of kheap, since an nheap is a perfect heap. In this perfect heap, there are two steps to find the location p of the node. In the first step, we determine the number of processors allocated to find the

location p in each processor. The number of processors, however, is equal to the number of leaves in kheap. In the second step, we select the locations as the number of processors determined. Since an nheap is a perfect heap, the first processor is located to the leftmost leaf or subheap of the nheap. Also, each processor must know the number of slots: a global variable S(PE(i),i=1,2,...,2^{l-1}, where l is the number of levels in kheap) is used to store the number of slots. This procedure for finding p takes O(1) time.

```
procedure parallel-perfect-level-find
/* p:the number of processors,
   PE(i):ith processor(1≤i≤p),
   S(PE(i)): the number of slots which ith
   processor has */
(1) /* determine the number of processors */
    p=2^(h(kheap)-1)
(2) /* determine the location of PE(i) in an
   nheap */
   for all i(1≤i≤p) do in parallel
       i = 2^(h(nheap)-1) + i - 1
       PE(i) = the ith location of an nheap
   allfor
(3) /* determine the number of slots which PE(i)
   has */
   for all i (1≤i≤p) do in parallel
       S(PE(i)) = the number of slots in PE(i)
   allfor
end
```

b. A Nonperfect Heap

If nheap is a nonperfect heap, three steps are necessary to select the location p of the node. In the first step, we determine the number of processors to allocate the location p to each processor. In the second step, we determine the location of the determined processor. Then, using the difference of the height between an nheap (h(nheap)) and a kheap (h(kheap)), we find the location that is the root of the subtree that is not a first complete binary tree from each subtree of the level determined. Then, if the difference between the size of the subtree and the slot is not less than 1, we find the lower subtree and select the nonperfect heap which has the difference of 1. We allocate the selected location to the first processor.

In the second step, we allocate the next location determined to the next processor and so on. Then, the number of processors allocated is equal to the number of p determined in the first step. The number of slots allocated in each processor is set to S(PE(i)) such as in a perfect heap. The following pseudo algorithm shows how to select the location p.

```
procedure parallel-nonperfect-level-find
begin
(1) /* determine the number of processors */
   p = 2^(h(kheap)-1)
(2) /* determine the location of the first
   processor */
   (2.a) level = h(nheap) - h(kheap) -1
       if(level(= 0) then level = 0
       pl = 2^(level) /* calculate the current level
       */
   (2.b) if(nheap is not a leaf)
       (a)if(the subheap of pl is a perfect heap)
          then pl=pl+1;go to step (2.b)
       (b)LD=size(pl's subheap)-size(pl's slot)
       (c)if(LD≤1) then go to step 2.c
       (d)if(the subheap of 2*pl is a perfect)
          then pl=2*pl+1 else pl=2*pl
       (e)go to step 2.b
   (2.c) PE(1)=the location pl of an nheap
       S(PE(1))=the number of slots in PE(1)
   (3) /* allocate the location from 2nd processor
      to pth */
      for all i ( 2≤i≤p) do in parallel
```

PE(i)=pl of nheap + (i-1)'s location
S(PE(i))=the number of slots in PE(i)
allfor
end

Theorem 1. The above procedure for finding the location p requires $O(2^{i-1})$ processors.

Proof. To determine the number of processors, we execute the instruction such as $p=2^{(h(kheap)-1)}$ in step 1 of the above procedure. Then the height of kheap corresponds to the level. Therefore, $2^{(h(kheap)-1)}$ means $2^{(i-1)}$, which is the maximum number of nodes in one level of kheap.■

Theorem 2. It takes $O(\log(n/k))$ time to execute finding location p.

Proof. The step 1 and 3 of procedure parallel-nonperfect-level-find needs O(1) time. In step 2, (2.a) takes O(1) time, which is the difference between the height of nheap and kheap, and (2.b) determines the location p of the root node of slots in nheap. The process to determine the path from the root of a nheap to location p is $\log(n)-\log(k)=\log(n/k)$ time. (2.c) takes O(1) time. This procedure, therefore, takes $O(\log(n/k))$ time.■

2.2 Merging Algorithm

Using the processor allocated in the above section, we suggest a merging method between k'heap which is the subheap of kheap and pheap, the subheap of pheap.

a. The Allocation Method of Subheap in PE(i)

We have suggested a method to point out the root of a subheap in an nheap. Using this method, we execute the following instructions in order to move the subheap which is a part of nheap to the local memory.

for all PE(i) ($1\le i\le p$) do in parallel
repeat
move the subheap which is constructed with nheap(the location of PE(i)) to the pheap of each processor
until(nheap's last node)
allfor

In order to merge k'heap with pheap, we have to select k'heaps from kheap in the shared memory and move them to processors. Then, the number of k'heaps that are to be assigned to a processor depends on the number of slots in that processor. To execute this, we point out the location of kheap, which has the ith location indicated by the total number of slots already assigned to the previous processors. Then, from the location of kheap determined, we create the merged k'heap. The merged k'heap is constructed with the number of slots to be assigned at the local memory of each processor .

For example, in Fig. 2.(c), we assume that a circle represents an internal node. We store the number of slots of each processor to S(PE(i)). To determine the location of kheap indicated in each processor, we select the location of kheap using X(PE(i)). Then each X(PE(i)) is initialized to 1. If S(PE(i)) = {2,2,1}, it represents the value of S(PE(i)) in Fig. 2. Since the number of slots in the first processor is S(1) = 2 and the location indicated in kheap of the first processor is 1(X(1) = 1), the number of slots indicated by the first processor is 2(S(1) = 2) for the first location of kheap.

Since the number of slots in the second processor, PE(2), is S(2) = 2 and the location indicated in kheap of the second processor is 3 (which is the sum of S(1) = 2 and 1), it points out two nodes from the third location of kheap which has the same number of slots (that is, 2(S(2) = 2) as the second processor. Also, the third processor PE(3) has the

number of slots S(3) = 1 and X(3) = 5(2(S(1))+2(S(2))+1): that is, it indicates only one node, since the third processor points out the number of slots(S(3)=1) in the 5th location of kheap. The following pseudo-algorithm describes the above things.

```
procedure selection-kheap's point
begin
int X[1:n]
for all PE(i)(1≤i≤p) do in parallel
  (1) /* determine the location in the kheap
      which each processor has */
    (1.a) for all i(1≤i≤p) do in parallel
            sum=0
    (1.b)   for j=1 to (i-1) do
              sum=sum+S[j-1]
            endfor
            X[i]=x[i]+sum
          allfor
  (2) /* move the number of slots determined
      from the kheap */
    (2.a) for all i ( 1≤i≤p) do in parallel
    (2.b)   for j=X[i] to (X[i]+S[i]-1) do
              move the j-th location of kheap
              to proper PE(i)
            endfor
          allfor
end
```

Theorem 3. The above procedure runs in O(*max(p, the number of slots which one processor has)*) time.

Proof. In the above procedure, X(1.a) runs in O(1) time and (1.b) runs in O(*p*) time. (2.a) runs p-1 times using p-1 processors. first step runs in O(p) time. (2.a) takes time, and (2.b) takes O(the number of slots which one processor has) time since it occurs as the number of slots from the proper location of kheap. As a result, it runs in O(*max(p, the number of slots which one processor has)*) time.■

Next, we consider that a pheap and a k'heap are being merged.[see Fig. 2.(d)] The following pseudo algorithm describes this:

```
procedure parallel-union-heaps(pheap, k'heap,
newheap)
begin
for all i(1≤i≤p) do in parallel
  (1) if(size(pheap) ) size(k'heap))
        then newroot = { last element in pheap }
                change the location of pheap
                and k'heap
        else newroot = { last element in k'heap}
  (2) distribute pheap to temporary location t
  (3) place newroot at pt
  (4) copy to to leftson of newheap(pt)
  (5) copy k'heap to rightson of newheap(pt)
  (6) sift-up(newheap)
allfor
end
```

This procedure constructs a heap as we treat the last node of the higher heap among two heaps (that is, pheap and k'heap) as the root of a newheap. We construct the newheap (that is relaxed min-max heap using creation function [9]) according to the minmax condition. At this time, we change the slot's min-max value from the k'heap which is merged: that is, we change the min-max value of the slot's elements to its min-max level's value. Changing its value changes the kheap's remaining elements. We require the resultant relaxed min-max heap while the newheap moves to the proper location of the nheap that is located in shared memory. But, if the root of the newheap is changed, an nheap does not satisfy the relaxed min-max heap condition. To solve this problem, we use a local variable cv. If the root

of the k'heap is changed after merging the pheap and the k'heap, we set the value of cv to 1. Otherwise, the value of cv is 0. To make the relaxed min-max heap condition, we then use the creation function, and we repeat this method until all cv's in each processor are equal to 0.

```
procedure construct-twoheaps
begin
  (1) n=log⌊size(nheap after merging)⌋
  (2) for l=n down to 1 do
        k=2^(h(nheap)-1)
        s=0
  (3)   m=min(⌊size(nheap after merging)⌋/2,
        2*k-1)
  (4)   for all j(k≤j≤m) do in parallel
          p=2*j
          if(p(⌊size(nheap after merging)⌋) and
            (nheap(p))nheap(p+1)) then p=p+1
          if(nheap(p)⟨nheap(j))
            then exchange(nheap(p),nheap(j))
          lock s
            s=s+1
          unlock s
        allfor
  (5)   while(s⟨(n-k+1))do
            wait
        endwhile
      endfor
  (6) for all i(1≤i≤p) do in parallel
        creation(PE(i)'s subheap)
        if(the subheap's root node is exchanged)
          then cv=1 else cv=0
      allfor
  (7) for all cv in PE(i)=0 do in parallel
        return(relaxed min-max heap)
      allfor
  allfor
end
```

Theorem 4. The above procedure requires ⌈(m+1)/4⌉ processors.

Proof. The number of nheap after merging is m(=n+k) and the height is l (=h(nheap)+1). The maximum number of nodes in this heap is $\sum 2^{l-1} = 2^{l}-1 \le m$.

The maximum number of processors needed to process in parallel is 2^{l-2} since the number of nodes in level l-1 requires them; so, it is $2^{l-2} \le \lceil (m+1)/4 \rceil$. Therefore, the number of processors needed is ⌈(m+1)/4⌉.■

Theorem 5. The above procedure has the time compexity of $O(\log(n/k)\times\log(n))$.

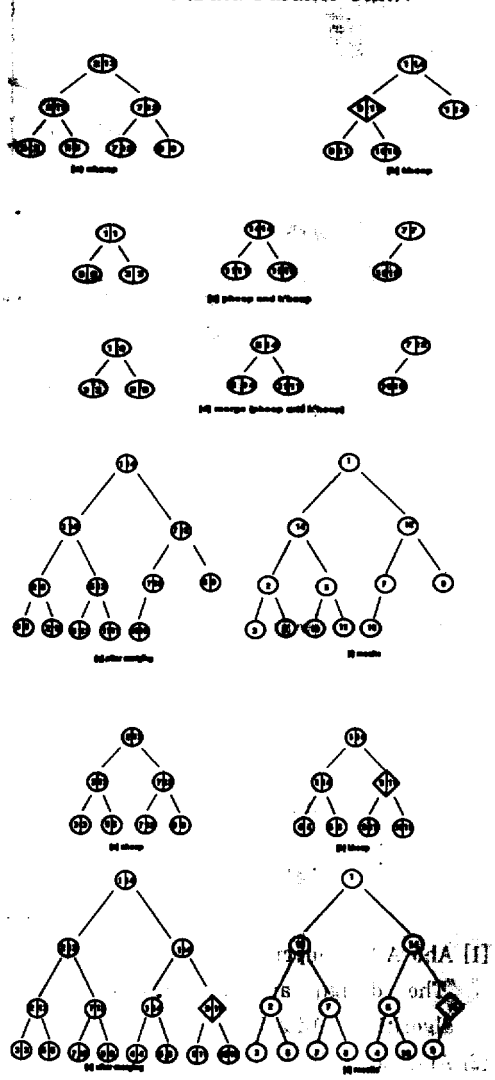
Proof. In the above procedure, step 4 takes O(1) time and steps 2 through 5 take O(log n) time. Further, since step 6 takes log(n) time as a sift-up function, it runs in $O(\log(p+k'))$ time because the subheap is pheap and k'heap. Steps 1 through 6 need the value of cv≠0; that is, it does not change the root node of the subheap in all processors. This is what we indicated from the root of the nheap to location p which the processor pointed out. The time complexity is $O(\log(n/k))$ as seen in theorem 2. Therefore, it runs in $O(\log(n/k)\times\log(n))$ time.■

Next, we consider the space complexity of the algorithm to merge two relaxed min-max heaps in parallel. First, we consider the size of nheap and kheap, which has 2n and 2k. So, it is $O(2n+2k)$ space. Second, when we think of the local memory that each processor has, it requires $O(p+k')$ since it needs the pheap(size is p) and the k'heap(size k'). Therefore, the total space needed is $O(n+k+$*the number of processor*$\times(k'+p))$.

3. Experimental Results

The MasPar MP-1 system, which was developed by MasPar Computer Co. in 1990, is an SIMD-SM machine with 8K processors. Each processor has a local memory of 64K bytes. The control unit of each PE(Parallel

Processor) is called ACU(Arithmetic Control Unit). The ACU and the PEs together are known as the DPU(Data Parallel Unit).



(Fig. 2) The example of relaxed min-max heaps with different sizes.

It is attached to a DEC 5000(known as the Front End) to allow user interface.

All compilations are carried out on the Front End. The DPU processors communicate with each other using two mechanisms : X-net and Router. This mechanism is supplied by MPL(MasPar Programming Language) which is an extension to the programming language C. The MasPar system provides a transparent control mechanism that automatically schedules parallel tasks on PEs, optimizes the use of hardware resources and manages all data motions.[11]

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

We developed a sequential program for merging relaxed heaps, which provides an optimal sequential merging for the relaxed heaps. We use the speedups to evaluate a new parallel algorithm for merging relaxed heap's. Speedup is defined to be the time elapsed from the moment the algorithm begins to the moment it terminates [2]. It is reasonable to assume that the time for merging relaxed heaps using the sequential method with a PE is as follows:

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

where c is a constant indepent of size n. Sequential times for the lists with more than 0.2 million elements were calculated using the formula:

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

where the size of k simply represents half of the total size. This size's value is different, with the actual's size and 0.02 million ≤ n ≤ 8 million and $t_{pe}(100,000) = 5.304$ seconds. Note

that if one uses this formula to compute t(200,000), the result is almost a perfect match with the corresponding experimental time.

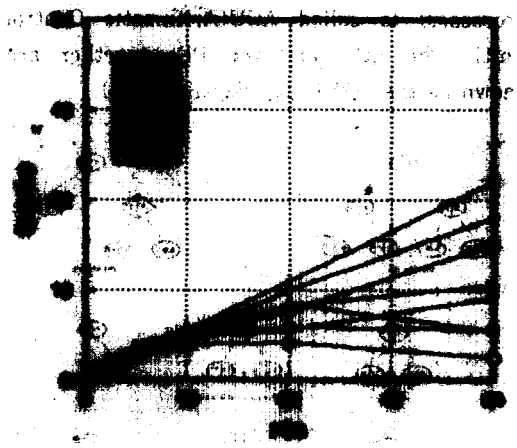
Table 1 shows the time required to merge. Fig. 3 plots the speedups achieved. As the problem size increases, task granularity increases. So, offsetting the overheads of the algorithms results in better speedup. Merging two relaxed heaps of 8 million elements with 64 processors yielded a 35.205-fold speedup over the use of one processor. This method was implemented in each processor's local memory. Global memory was used to communicate the code.

〈Table 1〉 Time to merge two relaxed min-max heaps(units:seconds).

n+k PE	1	2	4	8	16	32	64
100,000	5.304	3.1017	1.723	1.2578	0.8566	0.8021	1.357
200,000	11.359	6.4908	3.508	2.5061	1.6707	0.848	1.304
400,000	24.4066	13.569	7.136	4.8547	3.034	2.593	2.729
800,000	52.012	27.374	15.916	8.6029	5.8523	3.902	3.0664
1,000,000	66.3	–	18.41	13.15	8.221	7.4736	4.529
2,000,000	140.58	–	–	28.116	16.886	11.463	5.813
4,000,000	297.1305	–	–	–	–	18.57	10.317
8,000,000	627.6686	–	–	–	–	–	17.629

4. Conclusion

This paper has presented an improved relaxed min-max-pair heap, a data structure that implements a mergeable double-ended priority queues. Among other operations, it supports very efficient merging. Especially, the new data structure suggested in this paper eliminates the blossomed tree and lazy merging method to merge the relaxed heaps method suggested in [9]. As a result, employing $\max(2^{i-1}, \lceil (m+1)/4 \rceil)$ processor, the time complexity is $O(\log(n/k)\times\log(n))$ and the space complexity is $O(n+k+$*the number of processor*$\times(k'+p))$. Also on the MasPar machine, a 35.205-fold speedup is achieved, using 64 processors to merge 8 million data which consist of two relaxed heaps of different sizes.



(Fig. 3) Speedups of the relaxed min-max heaps of different sizes.

Acknowledgement

I thank Dr. Kundo and Dr. Zheng for their insighful comments, which have helped to improve the results and the presentation of the paper. I also thank Dr. Prasad who has supplied more data to complete the paper.

참 고 문 헌

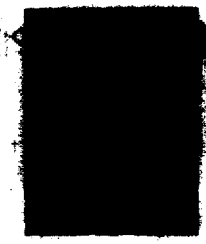
[1] Aho A.V., Hopcroft J.E. and Ullman J. D., "The design and analysis of computer algorithm", Addision-Wesely, 1974.

[2] Akl, Selim D., "The design and analysis of parallel algorithms",Prentice-hall, 1989.

[3] Olarin S., Overstreet C.M. and Wen Z., "A mergeable double-ended priority queue", Computer Journal, Vol. 34, No. 5, pp. 423-427, 1991.

[4] Deo N. and Prasad S., "parallel heap", Proceedings of the 1990 int'l conference on parallel processing, pp. 169-172, Aug. 1990.

[5] Gonnet G.H. and Munro J. I., "Heaps on heaps", SIAM Journal of Computing, Vol.15,No. 4,pp. 964-971, Dec. 1986.
[6] Deo N. and Prasad S., "Parallel heaps: An Optimal Parallel priority queue", Journal of Supercomputing, 6, pp. 87-98, 1992.
[7] Strothotte Th. and Sack J.R.,"An algorithm for merging heaps", Acta informatica 22, pp.171-186,1985.
[8] Zhang W. and Korf R.E., "Prallel Heap Operations on EREW-PRAM:summary of results", 6th IPPS, pp.315-318, 1993.
[9] Ding Y., Weiss M.A., "The relaxed min-max heap. A mergeable double-ended priority queue", Acta informatica 30, pp. 215-232, 1993.
[10] Prasad S. K., "Efficient parallel algorithms and data structures for discrete-event simulation", Ph.D dissertation, Univ. of Central Florida, Dec. 1990.
[11] —,"MasPar System Overview", MasPar Computer Co., 1992.



민 용 식

1981년 광운대학교 전자계산학과 졸업(이학사)
1983년 광운대학교대학원 전자계산학과 졸업(이학석사)
1991년 광운대학교대학원 전자계산학과 졸업(이학박사)
1984년~1987년 송원실업전문대학 전자계산학과 전임강사
1987년~현재 호서대학교 컴퓨터학부 게임공학전공 교수
1993년~1994년 미국 LSU 교환교수
관심분야 : 병렬 알고리즘, 가상현실, 계산기하학, 게임공학