MasPar 머쉰상의 병렬 힙 병합 알고리즘

민 용 식†

요 약

본 논문은 크기가 n와 k인 *nheap*과 *kheap*을 병합시키기 위한 병렬 알고리즘을 제시함과 동시에 그들을 MasPar상에 실제로 구현하고자 하는데 그 주된 목적이 있다. 이때, EREW-PRAM(Exclusive-Read Exclusive-Write Parallel Random Acess Machine)상에서 $\max(2^{i-1}, \lceil(m+1)/4\rceil)$개의 프로세서를 이용해서 본 논문에 제시된 알고리즘의 시간 복잡도가 $O(\log(n/k)*\log(n))$임을 제시하였다. 여기서 i는 heap의 height를 뜻하며, m은 크기 n와 k의 합으로 구성된 것이다. 또한 이것을 MasPar 컴퓨터에 적용을 시켰을 때, 데이타의 양이 8백만개이고, 64개의 프로세서를 이용한 경우의 speedup을 33.934를 얻었다. 이때 적용된 데이타의 형태는 불완전 힙상에서 크기가 $k<n$를 지니는 경우의 처리이다. 그리고 이같이 제시된 알고리즘의 EPU(Effective Processor Utilization)을 계산하면 ≈ 1인 최적의 speedup율을 나타냄을 알수가 있다.

A Parallel Algorithm for Merging Heaps on MasPar Machine

Yong Sik Min†

ABSTRACT

In this paper, we suggest a parallel algorithm to merge priority queues organized in two heaps, *kheap* and *nheap* of sizes k and n, correspondingly. Employing $\max(2^{i-1}, \lceil(m+1)/4\rceil)$'s processors, this algorithm requires $O(\log(n/k)*\log(n))$ on an EREW-PRAM, where i is the height of the heap and m is the summation of sizes n and k. Also, when we run it on the MasPar machine, this method achieves a 33.934-fold speedup with 64 processors to merge 8 million data items which consist of two heaps of different sizes. So, our parallel algorithm's EPU is close to ≈ 1, which is considered as an optimal speedup ratio.

1. Introduction

Priority queues have traditionally been used for applications such as branch-and-bound algorithm, discrete-event simulation, shortest path algorithm, multiprocessor scheduling and sorting. A priority queue is an abstract data structure which allows deletion of the highest priority item and insertion of new items[4, 8]. A *heap*, which is a complete binary tree such that the priority of the item at each node is higher than that of the items at its children, 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 on O(log n) time on an n-item heap.

Many have studied about parallel heaps as follows: Pinotti and Pucci[8] developed the n-Bandwidth heap and this method employs O(n) processors. However, it is optimal only on the CREW-PRAM.

Das and Horng[4] have developed a parallel heap without dedicated maintenance processors. Also, Nao and Zhang[8] proposed the parallel heap running in O(n/p+log n) time with p * O(n/log (n)) processors on an EREW-PRAM.

In this paper, we propose the parallel algorithm of merging two heaps, nheap of size n and *kheap* of size k and implement them on an exclusive-read exclusive-write parallel random access machine(EREW-PRAM). Using $\max(2^{i-1}, \lceil(m+1)/4\rceil)$'s processors where i is the height of the heap

† 정 회 원 : 호서대학교 전자계산학과 부교수
논문접수 : 1994년 9월 30일, 심사완료 : 1995년 6월 5일

and m is the summation of sizes n and k, this algorithm presented in this paper requires time complexity of O(log(n/k)*log(n)). All the logarithms in this paper are assumed to be in base 2. Also, when we ran on the MasPar, we achieved a 33.934-fold speedup with 64 processors to use 8 million data which consist of two heaps of different sizes.

2. Merging heaps in parallel

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 *pheap* is rooted at p, similar to the subheap rooted at p. Let the *size*(*heap*) be the number of elements it contains, and the height be defined as $\lceil \log(\text{size(heap)}) \rceil$. We introduce a function *h*(*heap*) that returns the height of heap. We will define *slots* of those leaf positions in nheap which are to be filled by merging processes. We will say that a node *p* covers a group of slots if all slots are descendents of p.[7]

To evaluate a parallel algorithm for a given problem, we use the speedup which 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. Clearly, the larger the speedup, the better the parallel algorithm[1].

For the sake of clarity, we will develop the parallel algorithm by first showing how to merge two prefect heaps of equal sizes, then how to merge two heaps of different sizes.

(1) Merging two heaps of equal sizes

The process of merging two heaps of equal sizes in parallel is the same as the sequential method[7] which takes two heaps, *nheap* and *kheap*, each of size k(=n) and produces a new heap with 2k elements. With a single processor, it requires O(log k). At this time, two heaps are perfect heaps. This pseudo algorithm is as follows :

```
procedure merge_equal_perfect_heaps
(nheap, kheap)
  invoke simple_merge on nheap and kheap, taking
the last element of kheap as the new root.
end merge_equal_perfect_heaps

procedure simple_merge(nheap, kheap, newroot)
  copy nheap to temporary loction t
  place newroot at root of nheap
  copy t to leftson of nheap
  copy kheap to rightson of nheap
  sift-up(nheap)
end simple_merge
```

Lemma 1. Two perfect heaps of equal size k can be merged with O(log(k)) comparisons and O(k) data movement.[7]

(2) Merging two heaps of unequal sizes

We will consider the simple case of inserting a heap of k elements, *kheap*, into a heap of n elements, *nheap*. Without loss of generality, assume that $k < n$. We proceed with three phases. In the first phase, determining the level of the root of slots which has k by the merging process in *nheap*, we have to allocate the nodes of the level to each processor. Second, the *pheap*, that is, subheap of *nheap* which is allocated in the *nheap* and the *k'heap*, that is, subheap of *kheap* which is allocated in the *kheap* are merged. At the time, the new subheaps merged (*pheap*+*k'heap*) must be satisfied with the heap's order. In the last phase, the newly merged heap is connected to *nheap* which exists in the shared memory and construct the heap condition about 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 either a perfect heap or a nonperfect heap.

(a) perfect heap

In the process of selecting the location p, the heap has a characteristic to fill out from the leftmost node of *nheap* in order to fill out the nodes of *kheap* since *nheap* is a perfect heap. In this perfect heap, we have two steps in finding the location p. In the first step, we must determine the number of processors allocated in order to find the location p in each processor. However, the number of processors is equal to the number of leaves in *kheap*. In the second step, we select the locations to determine the number of processors. Since *nheap* is a perfect heap, the first processor is located to the leftmost leaf or subheap of *nheap*. Also, we store the number of slots which are filled in each processor. In order to store the number of slot, we use the global variable S(PE(i)), where i=1, 2, ···, 2^{l-1}, and l is the number of the level in *kheap* in shared memory. The process of finding p is as follows :

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

Theorem 1. In procedure parallel-perfect-level-find, it runs O(1).

proof. Since it is very trivial, we omit.

(b) nonperfect heap

If nheap is a nonperfect heap, there are three necessary steps to select the location p. In the first step, we have to 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 *nheap*(h(*nheap*)) and *kheap* (h(*kheap*)), we find the location which 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 determined 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 is the process of selecting the location p.

```
procedure parallel-nonperfect-level-find(nheap,
kheap, p) 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 perfect)
         then pl=2*pl+1 else pl=2*pl
    (e) go to step (2.b)
    (2.c) PE(1)=the location pl of nheap
     S(PE(1))=the number of slots in PE(1)
(3) /* allocate location of 2nd processor to pth
    node */
    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 2. The procedure to find the location p requires $O(2^{(i-1)})$'s processors.

proof. To determine the number of processors, we execute the instruction such as $p=2^{(h(kheap)-1)}$ in step 1 of the procedure parallel-nonperfect-level-find. Then the height of *kheap* corresponds to that level. Therefore, $2^{(h(kheap)-1)}$ mean $2^{(i-1)}$ which is the maximum number of nodes on one level in *kheap*.

Theorem 3. To execute to find the location p requires O(log(n/k)).

proof. The step 1 of the procedure parallel-nonperfect-level-find needs O(1), and so does in step 3. In step 2, (2.a) requires O(1) which is the difference between the height of *nheap* and *kheap*. Step(2.b) determines the location p of root node of slots in *nheap*. The process to determine the path from the root of *nheap* to location p is log(n)-log (k)=log(n/k). (2.c) requires O(1). Therefore, this procedure requires O(log(n/k)).

2.2 merging algorithm

Using the processor allocated in the above section, we suggest the 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 suggested the method to find the root of subheap in *n*heap. So, using this method, we execute the following instructions in order to move in to the local memory.

```
for all PE(i) (1 ≤ i ≤ 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
```

To select the *k'heap* which is merged to *pheap*, we have to move kheap which is in the shared memory to the local memory of each processor the number of slots that are assigned to the 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 *k* heap determined, we created the merged *k'heap* which constructs the local memory of each processor the number of slots that are assigned to the processor. The following pseudo algorithm describes the above things.

```
procedure selection-kheap-point(kheap, p)
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
  alfor
(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 jth location of kheap to proper PE(i)
      endfor
  allfor
end
```

Theorem 4. In procedure selection-*kheap*-point, it runs O(max(*p*, the number of slots each processor has)).

proof. In procedure selection-*kheap*-point, (1.a) requires O(1) and (1.b) runs O(p) since it runs p-1 times using p processors. So, the first step runs O(p). (2.a) requires O(1) and (2.b) requires O(the number of slots which one processor has) since it occurs as the number of slot as the proper location of *k*heap. As a result, it runs O (max(*p*, the number of slots which one processor has)).

Next, we consider to merge *pheap* and *k'heap*. 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 t 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 newheap. Since newheap must satisfy the condition of heap, we use the sift-up function. We acquire the resultant heap while newheap moves the proper location of *nheap* which is located in shared memory. But, if the root of newheap is changed, *nheap* doesn't satisfy the heap-ordered condition. To solve this problem in this paper, we use the local variable *cv*. If the root of *k'heap* is changed after merging the *pheap* and the *k'heap*, we set the value of *cv* as 1. Otherwise, the value of *cv* is 0. Then to make the heap-ordered, we use the sift-up function and we repeat this method until all cvs for each processor is equal to 0.

```
procedure construct-twoheaps(nheap)
begin
(1) n=log  size(nheap after merging)
(2) for l=n down to 1 do
    k=2^(l-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
      sift-up(PE(i)'s subheap)
      if(the root node is exchanged)
        then cv=1 else cv=0
    allfor
(7) for all cv in PE(i)=0 do in parallel
      return(nheap)
    allfor
  allfor
end
```

In order to calculate the number of processors, we need to merge two heaps. The method is described below.

Theorem 5. In procedure construct-twoheaps, it requires $\lceil (m+1)/4 \rceil$ processors.

proof. The number of *n*heap after merge is $m(=n+k)$ and the height is $l(=h(nheap)+1)$. The maximum number of nodes in this heap is $2^{l-1}=2^l-1 \le m$. The maximum number of processors needed to process in parallel is $2^{(l-1)}$ since the number of nodes in level l-1 requires them. So, it is $2^{(l-2)} \le \lceil (m+1)/4 \rceil$. Therefore we need $\lceil (m+1)/4 \rceil$ processors.

Theorem 6. In procedure construct-twoheaps, its time complexity is $O(\log(n/k)*\log(n))$.

proof. In procedure construct-twoheaps, step (4) requires O(1) and steps (2) through (5) require O(log n). Further, since the step (6) requires log(n) as sift-up function, it runs O(log(p +k')) because the subheap is *pheap* and *k'heap*. Steps (1) through (6) need the value of $cv \le 0$, that is, it doesn't change the root node of the subheap in all processors. This indicates that nodes from the root of *nheap* to location *p* must be changed. This time complexity is O(log(n/k)) as we see in the theorem 3. Therefore, it runs O (log(n/k)).

3. Experimental Results

The MasPar MP-1 system, which developed at MasPar Computer Co. in 1990, is a SIMD-SM machine with 8K processor. Each processor has a local memory of 64K bytes. The control unit for these PEs(Parallel Processor) is called ACU (Arithmetic Control Unit). The ACU and the PEs together are known as the DPU(Data Parallel Unit). It is attached to a DEC 5000(known as the Front End) to allow user interface.

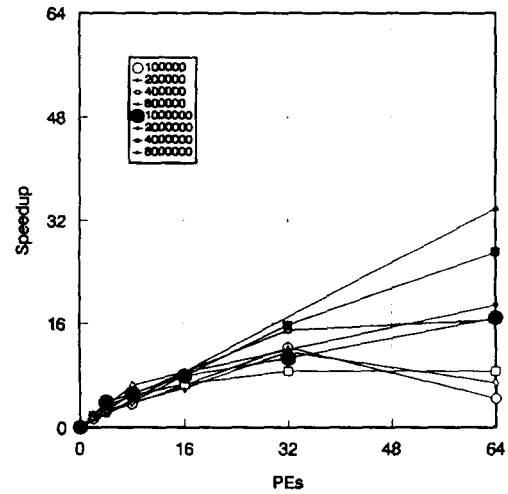
All compilations is carried out on the Front End. The DPU processors communicate with each other using two mechanism : X-net and Router. This mechanism is supplied by MPL(MasPar Programming Language) which is an extension to the programming language C. The MasPar system provides a transparent control mechanism that automatically schedules parallel tasks on PEs, optimizes the use of hardware resources and manages all data motion.[9]

Our algorithm was implemented in MPL to run on a 64-processor MasPar system. Also, to implement this algorithm on the MasPar, we tested randomly generated 32-bits integers with various distributions. No tests were made for duplicate elements, of which there were undoubtedly a few. The size of array 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 points presented in this section was obtained as the average of 20 program executions, each on a different set of test data which has a nonperfect heap.

〈Table 1〉 shows the time required to merge, and (Fig. 1) 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 heaps of 8 million with 64 processors yields a 33.934−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 heaps in parallel(unit: second)

n PE	1	2	4	8	16	32	64
100,000	6.384	4.711	2.732	1.758	0.9855	0.521	1.457
200,000	12.495	7.498	4.855	3.251	2.076	1.084	1.843
400,000	26.604	14.955	8.136	5.478	4.043	3.093	3.102
800,000	58.12	29.034	17.42	9.045	6.89	3.902	3.513
1,000,000	76.34		19.99	14.67	9.761	7.231	4.543
2,000,000	152.85			29.995	18.359	12.885	8.125
4,000,000	312.51					19.893	11.531
8,000,000	683.86						20.153



(Fig. 1) Speedup in heaps of different sizes

4. Conclusion

In this paper, we suggested the parallel algorithm to merge two heaps on an EREW-PRAM and implemented them on the MasPar Machine. As a result, we showed that the algorithm requires time complexity of $O(\log(n/k)*\log(n))$ and space complexity of $O(n+k+$the number of processor$*(k'+p))$. The detailed complexities are shown in 〈Table 2〉. Also, when we ran the algorithm on the MasPar machine, we achieved a 33.934-fold speedup using 64 processors to merge 8 million data, which consist of two heaps of different sizes.

In 〈Table 2〉, if the size of two heaps are equal and two heaps are perfect heaps, we use only one processor since its time complexity is O (log(k)) and is better than using two or more processors. In this method, we found the speedup is $S=T(1)(n)/T(k)(n)=(\log(\log(n/k))*\log(k))/(1/p)*(\log(n/k)*\log(n)*p)\fallingdotseq 1$. This method improves the performance of merging two heaps because EPU is almost equal to 1.

〈Table 2〉 The result of merging two heaps in parallel

	number of processors	space complexity	time complexity
perfect heap (equal size)	1	2k	O(log(k))
perfect heap (different size) & nonperfect heap	max(2^{m}, (m+1)/4)	n+k+number of processor *(k'+p)	O(log(n/k)) **log(n)

References

[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] Dekel, E. and Ozsvath, Istvan, "Parallel External Merging", IEEE, pp. 921-923, 1986.

[4] Deo, N and Prasad S., "Parallel Heap", Proceedings of the 1990 International Conference on Parallel Processing, pp. 169-172, Aug, 1990.

[5] Gonnet, G. H. and Nunro, J. I., "Heaps on Heaps", SIAM Journal of Computing, pp. 964-971, Vol. 15, No. 4, Dec. 1986.

[6] Stasko, J. T. and Vitter, J. S., "Pairing Heaps : Experiments and Analysis", Communication of ACM, Vol. 30, No. 2, pp. 234-249, Mar. 1987.

[7] Strothotte, Thomas 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] —, "MasPar System Overview", MasPar Computer Co., 1992.

민 용 식

1981년 광운대학교 전자계산학과 졸업(이학사)
1983년 광운대학교 대학원 전자계산학과 졸업(이학석사)
1991년 광운대학교 대학원 전자계산학과 졸업(이학박사)
1984년~87년 송원실업전문대학 전자계산학과 전임강사
1987년~현재 호서대학교 전자계산학과 부교수
1993년~94년 미국 LSU 전자계산학과 객원교수
관심분야 : 병렬 알고리즘 설계 및 분석, 코딩 이론, 유전자 알고리즘, 컴퓨터 그래픽스.