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, heap and heap of sizes k and n, correspondingly. Employing max(2^{-i}, [(m+1)/4])'s processors, this algorithm requires $O(\log(n/k) \times \log(n))$ 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, the 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 \(\approx 1\), 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 $O(n/p \times \log n)$ time with $p \times O(n/\log (n))$ 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}, [(m+1)/4])'s processors where i is the height of the heap.
and \( m \) is the summation of sizes \( n \) and \( k \), this al-
gorithm presented in this paper requires time
complexity of \( O(\log(n/k) \times \log(n)) \). All the log-
arithms 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 dif-
ferent sizes.

2. Merging heaps in parallel

We define a perfect heap as a heap with \( 2^e - 1 \)
elements, in which all leaves are on the same
level, otherwise the heap is non-perfect. The
\( p \)-heap is rooted at \( p \), similar to the subheap root-
ed at \( p \). Let the \( \text{size}(p) \) be the number of ele-
ments it contains, and the height be defined as
\( \lceil \log(\text{size}(p)) \rceil \). We introduce a function \( \text{h}(p) \)
that returns the height of heap. We will
define slots of those leaf positions in \( p \)-heap 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 algo-

For the sake of clarity, we will develop the
parallel algorithm by first showing how to merge
two perfect 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, \( p \)-heap and
\( k \)-heap, 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 fol-
lows:

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

procedures simple_merge(nheap, kheap, newroot)
  copy nheap to temporary loction t
  place newroot to right son of nheap
  copy t to left son of 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, \( k \)-heap, into a heap of \( n \) ele-
ments, \( n \)-heap. 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
\( n \)-heap, we have to allocate the nodes of the level
to each processor. Second, the \( p \)-heap, that is,
subheap of \( n \)-heap which is allocated in the \( n \)-heap
and the \( k \)-heap, that is, subheap of \( k \)-heap which is
allocated in the \( k \)-heap are merged. At the time,
the new subheaps merged (\( p \)-heap + \( k \)-heap) must
be satisfied with the heap’s order. In the last
phase, the newly merged heap is connected to
\( n \)-heap which exists in the shared memory and
construct the heap condition about \( n \)-heap.

2.1 level-find algorithm

In the first phase of the merging process, in
order to determine the location \( p \)-node in each
processor, we use the level of \( n \)-heap. Then we
classify \( n \)-heap 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^i, 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 processor has */
(1) /* determine the number of processors */
   p=2^k(hheap−1)
(2) /* determine the location of PE(i) in nheap */
   for all i(1 ≤ i ≤ p) do in parallel
   i=2^k(hheap−1)+i−1
   PE(i)=the i-th 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^k(hheap−1)
(2) /* determine the location of the first processor */
   (2a) level=h(nheap)−h(kheap)+1
   if(level≤0) then level=0
   p=2^level /* calculate the current level */
   (2b) if(nheap is not a leaf)
      (a) if(the subheap of pl is a perfect heap)
         then pl=pl+1; go to step (2b)
      (b) LD=size(pl's subheap)−size(pl's slot)
      (c) if(LD≤1) then go to step (2c)
      (d) if(the subheap of 2*pl is perfect)
         then pl=2*pl+1 else pl=2*pl
      (e) go to step (2b)
   (2c) 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 \( O(2^{r-1}) \)'s processors.

proof. To determine the number of processors, we execute the instruction such as \( p=2^{k_{\text{heap}}(n/k)} \) in step 1 of the procedure parallel-nonperfect-level-find. Then the height of \( k_{\text{heap}} \) corresponds to that level. Therefore, \( 2^{k_{\text{heap}}(n/k)} \) mean \( 2^{r-1} \) which is the maximum number of nodes on one level in \( k_{\text{heap}} \).

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 \( n_{\text{heap}} \) and \( k_{\text{heap}} \). Step\((2.b)\) determines the location \( p \) of root node of slots in \( n_{\text{heap}} \). The process to determine the path from the root of \( n_{\text{heap}} \) 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 \( k_{\text{heap}} \) which is the subheap of \( k_{\text{heap}} \) and \( p_{\text{heap}} \), the subheap of \( p_{\text{heap}} \).

(a) the allocation method of subheap in \( n_{\text{heap}} \)

We suggested the method to find the root of subheap in \( n_{\text{heap}} \). So, using this method, we execute the following instructions in order to move in to the local memory.

for all \( PE(i) \) \((1 \leq i \leq p)\) do in parallel
repeat
move the subheap which is constructed with \( n_{\text{heap}} \)
(the location of \( PE(i) \)) to the \( p_{\text{heap}} \) of each processor
until(\( n_{\text{heap}} \)'s last node)
endfor

To select the \( k_{\text{heap}} \) which is merged to \( p_{\text{heap}} \), we have to move \( k_{\text{heap}} \) 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 \( k_{\text{heap}} \) which has the \( i \)th location indicated by the total number of slots already assigned to the previous processors. Then, from the location of \( k_{\text{heap}} \) determined, we created the merged \( k_{\text{heap}} \) 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-\( k_{\text{heap}} \)-point(\( k_{\text{heap}} \), \( p \))
begin
\( \text{int} [X[1:n]] \)
for all \( PE(i) \) \((1 \leq i \leq p)\) do in parallel
\( (1.a) \) determine the location in the \( k_{\text{heap}} \) which each processor has */
(1.b) for all \( i(1 \leq i \leq p) \) do in parallel
\(\text{sum}=0 \)
\( (1.b)\) for \( j=1 \) to \((i-1)\) do
\(\text{sum} = \text{sum} + [X[j-1]] \)
endfor
\( X[i]=X[i]+\text{sum} \)
endfor
endfor
(2) */ move the number of slots determined from the \( k_{\text{heap}} \) */
(2.b) for all \( i(1 \leq i \leq p) \) do in parallel
(2.b) for \( j=X[i] \) to \((X[i]+SI[i]-1)\) do
move the \( j \)th location of \( k_{\text{heap}} \) to proper \( PE(i) \)
endfor
endfor

Theorem 4. In procedure selection-\( k_{\text{heap}} \)-point, it runs \( O(\max(p, \text{the number of slots each processor has})) \).

proof. In procedure selection-\( k_{\text{heap}} \)-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 \( O(\text{the number of slots which one processor has}) \) since it occurs as the number of slot as the proper location of \( k_{\text{heap}} \). As a result, it runs \( O(\max(p, \text{the number of slots which one processor has})) \).
Next, we consider to merge \texttt{pheap} and \texttt{kheap.}

The following pseudo algorithm describes this:

\begin{verbatim}
procedure parallel-union-heaps(pheap, kheap, newheap)
begin
    for all \(i (1 \leq i \leq p)\) do in parallel
    (1) if(size(pheap) > size(kheap))
        then newroot = (last element in pheap)
        change the location of pheap and kheap
    else newroot = (last element in kheap)
    (2) distribute pheap to temporary location \(t\)
    (3) place newroot at \(pt\)
    (4) copy \(t\) to leftson of newheap(\(pt\))
    (5) copy kheap to rightson of newheap(\(pt\))
    (6) sift-up(newheap)
    alfor
end
\end{verbatim}

This procedure constructs a heap as we treat the last node of the higher heap among two heaps, that is, \texttt{pheap} and \texttt{kheap} 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 \texttt{nheap} which is located in shared memory. But, if the root of newheap is changed, \texttt{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 \texttt{kheap} is changed after merging the \texttt{pheap} and the \texttt{kheap}, 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 \(cv\)'s for each processor is equal to 0.

\begin{verbatim}
procedure construct-twoheaps(nheap)
begin
    (1) \(n=\log \text{ size(nheap after merging)}\)
    (2) for \(i=n\) down to 1 do
            \(k=2^{i-1}\)
            \(s=0\)
            \(3) m=\min(\text{size(nheap after merging))/2, 2*2^{k-1}\)
    (4) for all \(j, k \leq j \leq m\) do in parallel
            \(p=2*j\)
            if \((p < \text{ size(nheap after merging)}\) and
            and \(nheap(p) > nheap(p+1)\) then \(p=p+1\)
            if \((nheap(p) < nheap(j))\)
                then exchange(nheap(p), nheap(j))
                lock s
            unlock s
        (5) while \((s < (n-k+1))\) do
                wait
            endwhile
        endfor
    (6) for all \(i (1 \leq i \leq p)\) do in parallel
            sift-up(PE(i)'s subheap)
            if (the root node is exchanged)
                then \(cv=1\) else \(cv=0\)
            alfor
    (7) for all \(cv\) in PE(i)=0 do in parallel
            return(nheap)
            alfor
        end
end
\end{verbatim}

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 \(nheap\) after merge is \(m(=n+k)\) and the height is \(l(=h(nheap)+1)\).

The maximum number of nodes in this heap is \(2^{l-1}=2^{l-1} \leq m\). 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 \(2^{l-2} \leq \lceil (m+1)/4 \rceil\). Therefore we need \(\lceil (m+1)/4 \rceil\) processors.

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

proof. In procedure construct-twoheaps, step (4) requires \(O(1)\) and steps (2) through (5) require \(O(n/\log(n))\). Further, since the step (6) requires \(O(n/\log(n))\) as sift-up function, it runs \(O(n/\log(n))\) because the subheap is \texttt{pheap} and \texttt{kheap}. Steps (1) through (6) need the value of \(cv \leq 0\), that is, it doesn't change the root node of the subheap in all processors. This indicates that nodes from the root of \texttt{nheap} to location \(p\) must be changed. This time complexity is \(O(n/\log(n))\) as we see in the theorem 3. Therefore, it runs \(O(n/\log(n))\).
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>
<thead>
<tr>
<th>n</th>
<th>PE</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>100,000</td>
<td>6.384</td>
<td>4.711</td>
<td>2.732</td>
<td>1.768</td>
<td>0.9665</td>
<td>0.521</td>
<td>1.457</td>
<td></td>
</tr>
<tr>
<td>200,000</td>
<td>12.495</td>
<td>7.488</td>
<td>4.855</td>
<td>3.351</td>
<td>2.076</td>
<td>1.254</td>
<td>1.843</td>
<td></td>
</tr>
<tr>
<td>800,000</td>
<td>58.12</td>
<td>29.034</td>
<td>17.42</td>
<td>9.045</td>
<td>6.89</td>
<td>5.002</td>
<td>3.513</td>
<td></td>
</tr>
<tr>
<td>1,000,000</td>
<td>76.34</td>
<td>39.999</td>
<td>14.67</td>
<td>9.761</td>
<td>7.221</td>
<td>4.543</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2,000,000</td>
<td>152.85</td>
<td>29.986</td>
<td>18.359</td>
<td>12.886</td>
<td>8.126</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4,000,000</td>
<td>312.51</td>
<td>26.218</td>
<td>18.359</td>
<td>12.886</td>
<td>8.126</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8,000,000</td>
<td>685.86</td>
<td>22.321</td>
<td>18.359</td>
<td>12.886</td>
<td>8.126</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(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 \(O(\log(n/k) \times \log(n))\) and space complexity of \(O(n+k+\text{the number of processor} \times (k' + p))\). 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 \(O(\log(n))\) and is better than using two or more processors. In this method, we found the speedup is \(S = T(1)/T(2) = (\log(n/k)) \times \log(k)\) \(/(1/p) \times (\log(n/k) \times \log(n) \times p) = 1\). 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

<table>
<thead>
<tr>
<th>number of processors</th>
<th>space complexity</th>
<th>time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfect heap</td>
<td>1</td>
<td>(2k)</td>
</tr>
<tr>
<td>(equal size)</td>
<td></td>
<td>(O(\log(n)))</td>
</tr>
<tr>
<td>perfect heap</td>
<td>(\max(2^{-i} \times (m+1)/4, n+k+number)</td>
<td>(O(\log(n/k)) * \log(n))</td>
</tr>
<tr>
<td>(different size)</td>
<td></td>
<td>time complexity</td>
</tr>
<tr>
<td>&amp; nonperfect heap</td>
<td></td>
<td>(1/p)</td>
</tr>
</tbody>
</table>

References