On Sorting an Intransitive Total Ordered Set Using Semi-Heap

Jie Wu
Department of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL 33431
jie@cse.fau.edu

Abstract

[1] *The problem of sorting an intransitive total ordered set, a generalization of regular sorting, is considered. This generalized sorting is based on the fact that there exists a special linear ordering for any intransitive total ordered set. A new data structure called semi-heap is proposed to construct an optimal $\Theta(n \log n)$ sorting algorithm. Finally, we propose a cost-optimal parallel algorithm using semi-heap. The run time of this algorithm is $\Theta(n)$ with $\Theta(\log n)$ processors under the EREW PRAM model.*

1. Introduction

Sorting is one of the fundamental problems in computer science and many different solutions for sorting have been proposed [5, 6]. Basically, given a sequence of n numbers $(n_1, n_2, ..., n_n)$ as an input, a sorting algorithm generates a permutation (reordering) $(n_1^{'}, n_2^{'}, ..., n_n^{'})$ of the input sequence such that $n_1^{'} \geq n_2^{'} \geq ... \geq n_n^{'}$.

We consider a generalization of the sorting problem by replacing $\geq$ with $\succ$, where $\succ$ is a total order without the transitive property, i.e., it is intransitive. That is, if $n_i \succ n_j$ and $n_j \succ n_k$, it is not necessary that $n_i \succ n_k$. The total order requires that for any two elements n_i and n_j, either $n_i \succ n_j$ or $n_j \succ n_i$, but not both (antisymmetric).

The set N of n elements exhibiting intransitive total order can be represented by a directed graph, where $n_i \succ n_j$ represents a directed edge from vertex n_i to vertex n_j. The underlying graph is a complete graph. This graph is also called a *tournament* [2], representing a tournament of n players where every possible pair of players plays one game to decide the winner (and the loser) between them. Sorting on N corresponds to finding a Hamiltonian path in the tournament.

[1]This work was support in part by NSF grant CCR 9900646.

Hell and Rosenfeld [4] proved that the bound of finding a Hamiltonian path is $\Theta(n \log n)$, the same complexity as the regular sorting. They also considered bounds on finding some generalized Hamiltonian paths. It is easy to prove that many regular sorting algorithms can be used to find a Hamiltonian path in a tournament, such as bubble sort, insertion sort, binary insertion sort, and merge sort. Among parallel sorting algorithms, even-odd merge sort can still be applied. However, heapsort and quicksort cannot be used. Bar-Noy and Naor [1] studied different parallel solutions based on different models and the number of processors. They showed that under the CRCW PRAM model, the generalized sorting problem can be solved in $\Theta(\log n)$ using $\Theta(n)$ processors. Other fast parallel algorithms can be found in [7].

In this paper, we propose a new data structure called *semi-heap*, which is an extension of a regular heap structure. We introduce an optimal $\Theta(n \log n)$ algorithm to determine a Hamiltonian path in a tournament based on the semi-heap structure. Then, we propose a cost-optimal parallel algorithm based on the semi-heap structure that takes $\Theta(n)$ in run time using $\Theta(\log n)$ processors in the EREW PRAM model. An implementation of the cost-optimal parallel algorithm in the network model with a linear array of processors is also shown.

2. Semi-Heap Data Structure

In this section, we first show the existence of a Hamiltonian path in any given tournament and then propose the semi-heap data structure.

Proposition: *Consider a set N ($|N| = n$) with any two elements n_i and n_j, either $n_i \succ n_j$ or $n_j \succ n_i$, but not both. Then elements in N can be arranged in a linear order $n_1^{'} \succ n_2^{'} \succ ... \succ n_{n-1}^{'} \succ n_n^{'}$.*

The proposition states that a Hamiltonian path exists in any given tournament, but not necessary for a Hamiltonian circle. That is, we can always arrange n players in a linear

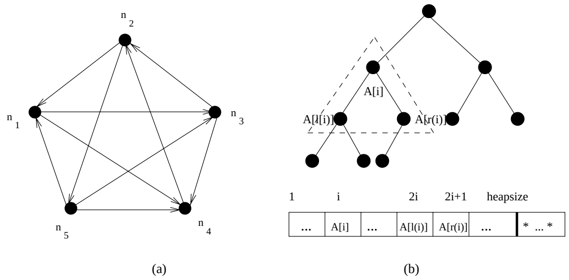

Figure 1. (a) A directed graph with a complete underlying graph. (b) A semi-heap structure as a set of overlapping triangles.

order from left to right such that each player beats the one to its right. Figure 1 (a) shows a directed graph with five vertices. One linear order is $n_3 \succ n_4 \succ n_2 \succ n_5 \succ n_1$. When $\succ$ is transitive, the linear order arrangement is reduced to a regular sorting problem. Unlike the regular sorting problem, more than one solution exists for the generalized sorting problem. For example, $n_1 \succ n_3 \succ n_2 \succ n_5 \succ n_4$ is another linear order for the example of Figure 1 (a).

Consider three elements n_1, n_2, n_3 in N, denote $n_1 = \max\{n_1, n_2, n_3\}$ if $n_1 \succ n_2$ and $n_1 \succ n_3$. Note that in a total order without the transitive property, the maximum element may not exist among three elements. For example, if $n_1 \succ n_2$, $n_2 \succ n_3$, and $n_3 \succ n_1$, $\max\{n_1, n_2, n_3\}$ does not exist. Next we introduce a new concept of the maximum element based on $\succ$.

Definition 1: $n_1 = \max_{\succ}\{n_1, n_2, n_3\}$ *if both* $n_2 = \max\{n_1, n_2, n_3\}$ *and* $n_3 = \max\{n_1, n_2, n_3\}$ *are false.*

Note that when $n_i = \max\{n_1, n_2, n_3\}$ are false for all $i = 1, 2, 3$, every n_i is a maximum element.

A *semi-heap* is any array object that can be viewed as a complete binary tree, like a regular heap. A complete binary tree of height h is a binary tree that is full down to level $h - 1$, with level h filled in from left to right. However, the regular heap property is changed. Let $L(n^{'})$ and $R(n^{'})$ represent left and right child nodes of $n^{'}$, respectively. When a child, say $R(n^{'})$, does not exist, the relation $n^{'} \succ R(n^{'})$ automatically holds.

Definition 2: *A semi-heap for a given intransitive total order $\succ$ is a complete binary tree. For every node $n^{'}$ in the tree,* $n^{'} = \max_{\succ}\{n^{'}, L(n^{'}), R(n^{'})\}$.

When an array A is used to represent a semi-heap, $l(i)$ and $r(i)$ are used as indices of the left and right child nodes of i; they can be computed simply by $l(i) = 2i$ and $r(i) = 2i + 1$. Figure 1 (b) shows a semi-heap with 10 elements. A semi-heap can be viewed as a set of overlapping triangles, with each triangle consisting of $A[i]$, $A[l(i)]$, $A[r(i)]$. Figure 2 shows four possible configurations of a triangle under relation $\succ$. In this figure, if $A[i] \succ A[l(i)]$ is true, a directed edge is drawn from $A[i]$ to $A[l(i)]$. Note that $A[i] = \max_{\succ}\{A[i], A[l(i)], A[r(i)]\}$ for all cases. In cases (a) and (b) condition $A[i] = \max\{A[i], A[l(i)], A[r(i)]\}$ also holds.

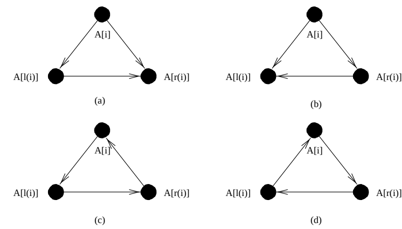

Figure 2. Four possible configurations of a triangle in a semi-heap.

To simplify the presentation, we fill in a special symbol $*$, with a smaller value than any one in the semi-heap for entries that are outside the semi-heap. That is, $A[i] \succ A[j]$ is true for all i inside the semi-heap and all j outside the semi-heap. Specifically, $A[i]$ is an element of the semi-heap if $1 \leq i \leq heapsize$. $A[j]$ is an element outside the semi-heap if $j > heapsize$.

3. Generalized Sorting Using Semi-Heap

Although a semi-heap resembles a heap, the traditional heapsort algorithm cannot be directly applied to a semi-heap to generate a generalized sorted sequence. Recall that with the transitive property, root $A[1]$ of the heap is always the maximum element in the heap, i.e., the player at the root "beats" all the other players in the tournament. When we "discard" the root, it is "replaced" by the last element $A[n]$ in the heap, and then, the heap is reconstructed by pushing $A[n]$ down in the heap, if necessary, so that the new root is the maximum element among the remaining elements. However, in a semi-heap, we may face a situation in which $A[n]$ beats all $A[1]$, $A[2]$, and $A[3]$, which is an impossible situation in a regular heap. $A[n]$, the new root, cannot be selected (and is removed from the semi-heap) in the next round to be placed after $A[1]$, the previously selected element, because $A[n]$ beats $A[1]$. On the other hand, because $A[n]$ beats $A[2]$, its left child, and $A[3]$, its right child, $A[n]$ cannot be pushed down in the semi-heap. Therefore, a different strategy has to be developed for semi-heap.

We follow closely the notation used in Cormen, Leiserson, and Rivest's book [3]. The sorting using semi-heap consists of four modules: SEMI-HEAPIFY(A, i), BUILD-SEMI-HEAP(A), REPLACE(A, i), and SEMI-HEAP-SORT(A). SEMI-HEAPIFY(A, i) constructs a

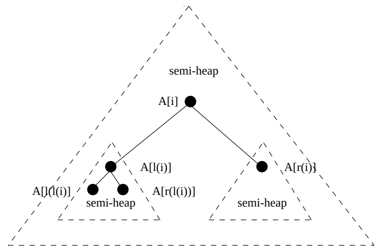

Figure 3. The construction of a semi-heap using SEMI-HEAPIFY.

semi-heap rooted at $A[i]$, provided that binary trees rooted at $A[l(i)]$ and $A[r(i)]$ are semi-heaps (see Figure 3). The cost of SEMI-HEAPIFY is the height of node $A[i]$, measured by the number of edges on the longest simple downward path from the node to a leaf. That is, the cost of SEMI-HEAPIFY is $\Theta(\log n)$, where $n = heapsize$. BUILD-SEMI-HEAP uses the procedure SEMI-HEAPIFY in a bottom-up manner to convert an arbitrary array A into a semi-heap. The cost of BUILD-SEMI-HEAP is $\Theta(n)$, which is the same cost of building a regular heap.

Generalized sorting is done through SEMI-HEAP-SORT by repeatly printing and removing the root of the binary tree (which is initially a semi-heap). The root is replaced by either its leftchild or rightchild through REPLACE. The selected child is replaced by one of its child nodes. The process continues until reaching one of the leaf nodes and the entry for that leaf node is replaced by $*$, i.e., that leaf node is removed from the tree. A new tree derived is no longer a semi-heap; however, each overlapping triangle in the tree still meets the maximum element requirement in Definition 2. The cost of REPLACE is the height of the current tree, which is bounded by the height of the original semi-heap, $\Theta(\log n)$. Therefore, the cost of SEMI-HEAP-SORT is $\Theta(n \log n)$. Without loss of generality, we assume that $n \geq 1$.

SEMI-HEAPIFY(A, i)
if $A[i] \neq \max_{\succ}\{A[i], A[l(i)], A[r(i)]\}$
  then find $winner$ such that
    $A[winner] \longleftarrow \max\{A[i], A[l(i)], A[r(i)]\}$
    exchange $A[i] \longleftrightarrow A[winner]$
    SEMI-HEAPIFY$(A, winner)$

BUILD-SEMI-HEAP(A)
for $i \longleftarrow \lfloor \frac{heapsize}{2} \rfloor$ **downto** 1
  do SEMI-HEAPIFY(A, i)

REPLACE(A, i)
if $(A[l(i)] = *) \wedge (A[r(i)] = *)$
  then $A[i] \longleftarrow *$
else if $(A[i] \succ A[l(i)]) \wedge (A[l(i)] \succ A[r(i)])$
  then $A[i] \longleftarrow A[l(i)]$
    REPLACE$(A, l[i])$
  else $A[i] \longleftarrow A[r(i)]$
    REPLACE$(A, r[i])$

SEMI-HEAP-SORT(A)
BUILD-SEMI-HEAP(A)
while $(A[l(1)] \neq *) \vee (A[r(1)] \neq *)$
  do **print**$(A[1])$
    REPLACE$(A, 1)$
print$(A[1])$

Theorem 1: *BUILD-SEMI-HEAP constructs a semi-heap for any given complete binary tree.*

Proof: The procedure BUILD-SEMI-HEAP goes through nodes that have at least one child node and runs SEMI-HEAPIFY on these nodes. The order in which these nodes are processed guarantees that the subtrees rooted at child nodes of $A[i]$ are semi-heap before SEMI-HEAPIFY runs at $A[i]$.

When SEMI-HEAPIFY is called at $A[i]$, if $A[i]$ is the maximum element based on among $A[i]$, $A[l(i)]$, and $A[r(i)]$ based on $\succ$, the binary tree rooted at $A[i]$ is automatically a semi-heap. Otherwise and without loss of generality, one of the child nodes, say $A[l(i)]$, is the winner among three, i.e., $A[l(i)]$ beats both $A[i]$ and $A[r(i)]$. In this case, $A[l(i)]$ is swapped with $A[i]$, which ensures that node $A[i]$ and its child nodes satisfy the semi-heap property. However, node $A[l(i)]$ now has the original $A[i]$ and thus the subtree rooted at $A[l(i)]$ may violate the semi-heap property. Therefore, SEMI-HEAPIFY must be called recursively on that subtree.

A new problem (that does not appear in the original heap structure) is how to ensure that the resultant root $A[l(i)]$, after applying SEMI-HEAPIFY at $A[l(i)]$, will not violate the semi-heap property among $A[i]$, $A[l(i)]$, and $A[r(i)]$. In a regular heap, $A[i]$ is the maximum element in the tree rooted at $A[i]$, the heap property among $A[i]$, $A[l(i)]$, and $A[r(i)]$ automatically holds. In a semi-heap, we need to prove that the newly selected root $A[l(i)]$ (other than the original value $A[i]$), which is either $A[l(l(i))]$ or $A[r(l(i))]$ in the original tree, cannot beat both $A[i]$ (the original $A[l(i)]$) and $A[r(i)]$. In fact, we prove that $A[i]$ (the original $A[l(i)]$) always beats the newly selected $A[l(i)]$ (the original $A[l(l(i))]$ or $A[r(l(i))]$). We consider the following two cases in the original tree with a semi-heap rooted at $A[l(i)]$ (see Figure 3):

- If $A[l(i)]$ beats both $A[l(l(i))]$ and $A[r(l(i))]$. The problem is solved, because in the resultant tree, node $A[l(i)]$ becomes $A[i]$ and either $A[l(l(i))]$ or $A[r(l(i))]$ becomes $A[l(i)]$.

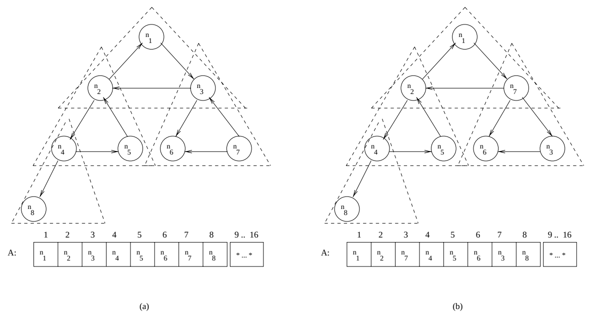

Figure 4. An example tree: (a) the initial configuration, (b) the semi-heap configuration, after applying BUILD-SEMI-HEAP.

- If $A[l(i)]$ beats only one child node, then without loss of generality, we assume that $A[l(i)]$ (which is now $A[i]$) beats $A[l(l(i))]$, $A[l(l(i))]$ beats $A[r(l(i))]$, and $A[r(l(i))]$ beats $A[l(i)]$. To select a winner among the original $A[i]$ (now $A[l(i)]$), $A[l(l(i))]$, $A[r(l(i))]$, other than $A[l(i)]$, $A[l(l(i))]$ is the only choice (since $A[r(l(i))]$ has lost to $A[l(l(i))]$). Consequently, $A[l(l(i))]$ becomes the newly selected root of the left subtree of $A[i]$, based on the assumption, $A[i]$ (the original $A[l(i)]$) beats $A[l(i)]$ (the original $A[l(l(i))]$) in the resultant tree. ■

Consider a complete binary tree with eight vertices, i.e., $heapsize = 8$. The initial configuration of array A is n_1, n_2, n_3, n_4, n_5, n_6, n_7, and n_8. The tournament is represented by an 8×8 matrix M given below, where $M[i, j] = 1$ if n_i beats n_j (i.e., $n_i \succ n_j$) and $M[i, j] = 0$ if n_i is beaten by n_j (i.e., $n_j \succ n_i$). $M[i, i] = -$ represents an impossible situation. Note that $M[i, j] = 1$ if and only if $M[j, i] = 0$.

$$M = \begin{pmatrix} - & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & - & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & - & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & - & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & - & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & - & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & - & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & - \end{pmatrix}$$

Figure 4 (a) shows the initial configuration of this complete binary tree in array A, where the corresponding tree structure is represented by a set of overlapping triangles. Three edges among three vertices in each triangle represent tournament results between three pairs of players in the triangle. That is, an edge directed from n_i to n_j exists if $M[i, j] = 1$ in matrix M. Relationships between two vertices from different triangles are not shown in the figure. Figure 4 (b) shows the resultant semi-heap after applying BUILD-SEMI-HEAP. $A[j]$ is filled with $*$ for $j \geq 8$. Actually, it is sufficient to define the size of A to be $2 \times heapsize$. A step-by-step application of REPLACE($A, 1$) to the example of Figure 4 is shown in Figure 5, where the selected (printed) elements are placed beside the root in a left-to-right order. In this example, the final output sequence is $n_1 \succ n_7 \succ n_3 \succ n_2 \succ n_4 \succ n_5 \succ n_8 \succ n_6$. Once all elements are printed, all entries in array A are filled with $*$. The correctness of this result can be easily verified through the given matrix M.

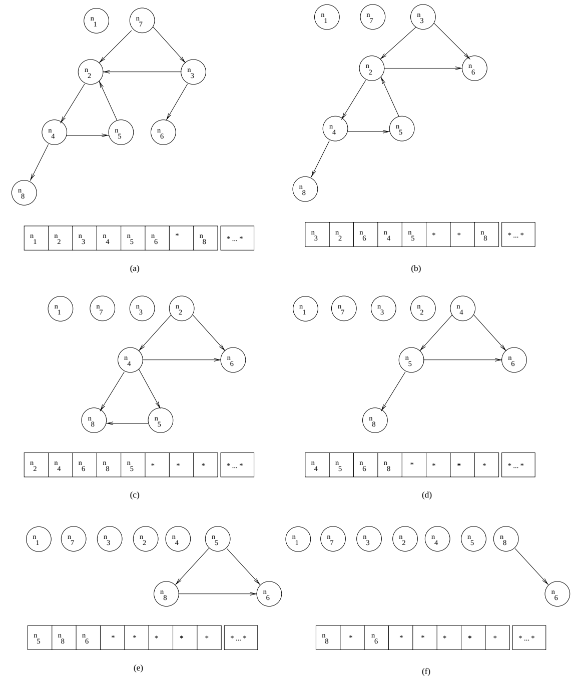

Figure 5. A step-by-step application of REPLACE(A, i) in the example of Figure 4.

Note that although the REPLACE process destroys the semi-heap structure (since the resultant tree is no longer a complete binary tree), each overlapping triangle in the corresponding binary tree still maintains one of the four possible configurations of a semi-heap as shown in Figure 2. Therefore, it always generates a generalized sorted sequence for any given semi-heap.

Theorem 2: *For any given semi-heap, SEMI-HEAP-SORT generates a generalized sorted sequence.*

Proof: It suffices to show that REPLACE always replaces the current root by an element beaten by the root. In addition, each overlapping triangle in the binary tree is still one of the four possible configurations of a triangle in a semi-heap, i.e., the root of each triangle is the maximum element based on $\succ$ in the triangle. Based on the definition of REPLACE, the current root $A[i]$ is replaced by $A[l(i)]$ for cases (a) and (c) and by $A[r(i)]$ for cases (b) and (d) of Figure 2. The replacing element, say $A[l(i)]$, is itself replaced by an element in the triangle rooted at $A[l(i)]$. This process continues iteratively down the semi-heap. In addition, the new root $A[i]$ beats both of its child nodes (if any). This property ensures when a child node is missing (i.e., the corresponding triangle contains only two nodes), $A[i]$ can still be replaced by another child node without causing any problem. Therefore, the root of each triangle is still the maximum element based on $\succ$ in the triangle. ■

4. Parallel Generalized Sorting Using Semi-Heap

We introduce in this section a cost-optimal parallel sorting algorithm using semi-heap in the EREW PRAM model. A sorting algorithm is *cost-optimal* if the product of run time and the number of processors is $\Theta(n \log n)$, the bound for sequential solutions. Specifically, the pipeline technique is used to reduce the run time of the sequential algorithm from $\Theta(n \log n)$ to $\Theta(n)$ using $\Theta(\log n)$ processors with different processors handling activities of different levels of the heap.

Because procedure BUILD-SEMI-HEAP(A) takes only $\Theta(n)$, no speed up is necessary for this part. Procedure SEMI-HEAP-SORT can be improved by assigning one processor to each level of the binary tree, which initially is a semi-heap. REPLACE($A, 1$) is pipelined level to level and this procedure is called at every other step, because each node is shared by two processors at adjacent levels, a passive step is inserted between two calls. The run time of SEMI-HEAP-SORT is reduced to $\Theta(n)$ using $\Theta(\log n)$ processors. This parallel algorithm runs on the CREW PRAM model, since two adjacent processors may access (read) vertices in two overlapping triangles of the tree. However, simultaneous accesses can be avoided by creating a copy of each vertex that appears in two overlapping triangles. The enhanced version runs on the EREW PRAM model.

We use the network model to illustrate the parallel algorithm. The *network model* [8] can be viewed as a graph where each node represents a processor, and each directed edge (P_i, P_j) represents a two-way communication link between processors P_i and P_j. It is easy to convert the algorithm back to the EREW PRAM model by replacing send and receive commands in the network model by read and write commands in the EREW PRAM model. Shared elements are duplicated and stored in local memory of adjacent processors. Processors are connected as a linear array, where each processor communicates with up to two adjacent processors.

The *level* of each node in the semi-heap is its distance to the root. Clearly, $h = \lceil \log(n+1) \rceil$ is the maximum level and is called the *depth* of the semi-heap. A linear array of h processors are used which are labeled as $P_0, P_1, ..., P_{h-1}$. Processor P_i has a copy of elements in levels i and $i+1$ of the semi-heap. In general, P_i is assigned with 2^i triangles (i.e., 3×2^i consecutive elements in array A).

In the proposed parallel algorithm, each processor alternates between an *active step* and a *passive step*. Processors with even ID's take active steps in even steps, while with odd ID's take active steps in odd steps. That is, at an even step, processors P_0, P_2, P_4, ... take the active step and processors P_1, P_3, P_5 ... take the passive step. The role of active and passive among these processors exchanges in the next step, which is an odd step. Active and passive steps include the following activities: At an active step, each processor performs local update and sends relevant messages to two adjacent processors (if they exist). At a passive step, each processor receives messages from two adjacent processors (if they exist) and saves them.

In the implementation using the network model, processor P_0 initiates the sorting process and the rest P_i's are activated in sequence. Processor P_0 also generates a termination signal which is passed down the linear array of processors once the job is completed. To make our algorithm more general, some activities are not ordered within a step.

P_0 at an active step (starts from step 0):

1. Prints root $A[1]$.
2. If both child nodes are $*$, $A[1]$ is replaced by $*$, and then, P_0 sends a termination signal to P_1 and stops.

   If at least one child node is not $*$, replaces $A[1]$ by one of two child nodes, $A[2]$ or $A[3]$, following the rule in REPLACE. If A[2] is selected, P_0 sends $id = 2$ to processor P_1; otherwise, $id = 3$ is sent. In the next step (a passive step), P_0 receives $(id, replacement)$ from P_1, and then, performs the update $A[id] := replacement$.

$P_i, i > 0$, in a passive step:

If P_i receives $(id, replacement)$ from P_{i+1}, it performs the update $A[id] := replacement$.

If P_i receives signal $id = i$ from P_{i-1}, it performs the following activities in next active step:

1. If both child nodes are $*$, $A[i]$ is replaced by $*$; otherwise, $A[i]$ is replaced by either $A[2i]$ or $A[2i+1]$ based the replacement rule.
2. Send $(i, A[i])$ to P_{i-1}.
3. If either $A[2i]$ or $A[2i+1]$ is selected to replace $A[i]$, the corresponding id ($2i$ or $2i+1$) is sent to P_{i+1}, provided P_i is not the last processor (i.e., $i \neq h-1$); otherwise, the selected element is replaced by $*$.

If P_i receives the termination signal from P_{i-1}, it forwards the termination signal to the next processor P_{i+1} (if it exists) in the next active step, and then, P_i stops.

Note that in the above algorithm, although each processor is assigned a different number of triangles, its workload stays the same: each processor operates on at most two triangles in a passive step and at most one triangle in an active step. When a child node exceeds the boundary of the semi-heap, it has a default value of $*$ and no replacement is needed. The step-by-step illustration of the above algorithm is shown in Figure 6 for the first three steps of Figure 5, where the semi-heap is represented as a tree structure without showing the detail orientation of each triangle. In this example, each step of Figure 5 corresponds to two steps in Figure 6. Replacement activities are shown using dashed lines.

Theorem 3: *The proposed parallel implementation is cost-optimal with a run time of* $\Theta(n)$ *using* $\Theta(\log n)$ *processors.*

Proof: It is clear that $\Theta(\log n)$ processors are used. Also, one element is selected (printed) in every other step and all n elements are printed in $2n$ steps, and hence, the run time is $\Theta(n)$. Because the product of run time and the number of processors used matches the lower bound $\Theta(n \log n)$ for a sequential algorithm, the proposed parallel implementation is cost-optimal. ■

The proposed implementation can be extended without having to identify the last processor. This extension can be done by adding one extra processor P_h which handles the last level of the semi-heap (this last level is also duplicated). Clearly, each child node of any element in the last level is an $*$. Therefore, no other processor will be activated by P_h. Also, each processor can terminate itself without using a termination signal originated from P_0. P_i terminates itself once all 3×2^i elements (that it controls) become $*$; however, the bookkeeping process is more complicated than the one in the original design.

5. Conclusions

We have proposed a data structure called semi-heap which is a generalization of the traditional heap structure. The semi-heap structure is used to solve a generalized sorting problem. We have shown that the generalized sorting problem can be solved optimally using semi-heap. The solution can be easily extended to a cost-optimal EREW PRAM algorithm with $\Theta(n)$ in run time using $\Theta(\log n)$ processors. An implementation of this parallel algorithm under the network model is shown, where processors are connected as a linear array. We are currently studying the problem of generalized merging, where the relation between elements does not have the transitive property. The result of this study will be reported in a separate paper [9].

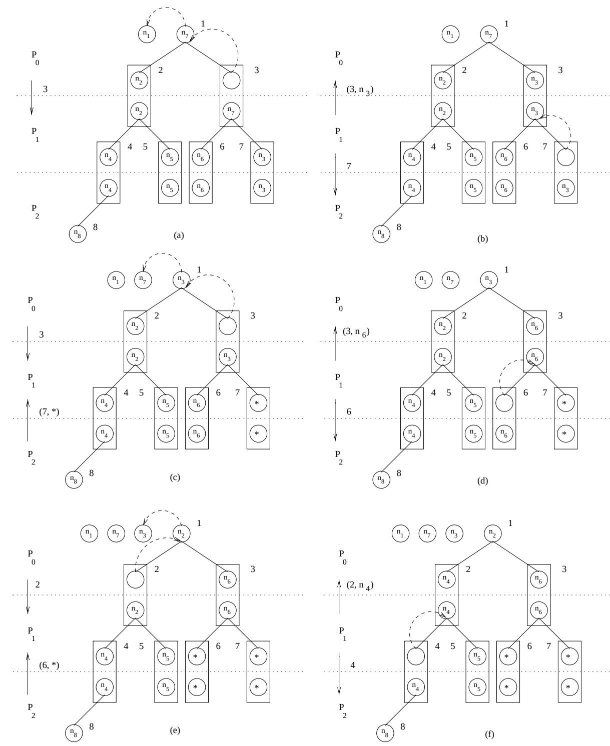

Figure 6. A step-by-step illustration of the first three steps of Figure 5.

References

[1] A. Bar-Noy and J. Naor. Sorting, minimal feedback sets, and Hamilton paths in tournaments. *SIAM Journal of Discrete Mathematics*. 3, (1), Feb. 1990, 7-20.

[2] J. A. Bondy and U.S.R. Murthy. *Graph Theory and Applications*. The Macmillan Press. 1976.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press. 1994.

[4] P. Hell and M. Rosenfeld. The complexity of finding generalized paths in tournaments. *Journal of Algorithms*. 1983, 4, 303-309.

[5] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company. 1992.

[6] D. Knuth. *The Art of Computer Programming, Vol 3, Sorting and Searching*. Addison-Wesley Publishing Company, second edition. 1998.

[7] D. Soroker. Fast parallel algorithms for finding Hamilton paths and cycles in a tournament. *Journal of Algorithms*. 1988, 276-286.

[8] J. Wu. *Distributed Systems Design*. The CRC Press. 1999.

[9] J. Wu and S. Olariu. On optimal merge of two intransitive sequences. in preparation.