P-TREE STRUCTURES AND EVENT HORIZON: EFFICIENT EVENT-SET IMPLEMENTATIONS

Katerina Asdre
Department of Computer Science
University of Ioannina
GR-45110 Ioannina, GREECE

Stavros D. Nikolopoulos
Department of Computer Science
University of Ioannina
GR-45110 Ioannina, GREECE

ABSTRACT
This paper describes efficient data structures, namely the Indexed P-tree, Block P-tree, and Indexed-Block P-tree (or IP-tree, BP-tree, and IBP-tree, respectively, for short), for maintaining future events in a general purpose discrete event simulation system, and studies the performance of their event set algorithms under the event horizon principle. For comparison reasons, some well-known event set algorithms were also selected and studied; that is, the Dynamic-heap and the P-tree algorithms. To gain insight into the performance of the proposed event set algorithms and allow comparisons with the other selected algorithms, they are tested under a wide variety of conditions in an experimental way. The time needed for the execution of the Hold operation is taken as the measure for estimating the average time complexity of the algorithms. The experimental results show that the BP-tree algorithm and the IBP-tree algorithm behave very well with all the sizes of the event set and their performance is almost independent from the stochastic distributions.

1 INTRODUCTION
In a discrete event simulation system an event (or future event) is a collection of actions that are scheduled to be executed in a specific simulation time called event time. In such a system events are kept in objects known as event notices and maintained in a data structure known as event set. An event notice is represented by a record with two fields, t and a, where t is the scheduled time for its occurrence, and a is the activity which is scheduled in time t (Fishman 1973, Mitra 1982).

In a discrete-event simulation system based on the next-event time-advance approach, the next-event time-advance mechanism is responsible for the simulation clock; it initializes the simulation clock, and then it determines the event times of future events. The simulation clock is then advanced to the event time of the earliest event known as next event (i.e., the event with the minimum event time) and the system state is updated to account for the occurrence of this event. When the next event occurs, it is removed from the event set and the simulation clock is advanced to the time of the next event. The processing of this event may lead to the generation and scheduling of additional (new) future events. A new event is scheduled when its event time t becomes known. Then, an event notice is created and inserted into the event set in such a way that it is ensured that this event will occur at the scheduled time t. This type of simulation approach is referred to as discrete event-driven simulation.

The responsibility for the execution of these operations in a discrete event-driven simulation is due to an algorithm which is known as an event set algorithm (or event scheduling algorithm); that is, it

- scans the event set to determine the proper insertion position for the new event,
- removes the next-event from the event set, and
- advances the simulation clock to the time of the next-event.

It is obvious that being able to repeatedly select the event notice from the event set that has the minimum event time is essential. If all of the event notices in this event set are known in advance, and their event times remain unchanged, then the problem of determining the next event and updating the simulation clock is easily solved by sorting the event notices and retrieving them in order. In the simulation process discussed above, however, it is often necessary to insert new event notices into the event set as other events are being processed. This leads to the following set of priority queue operations:

- insert a new event notice into the event set (in a proper position according to its event time),
- find the event notice with the minimum event time, and
• remove the event notice with the minimum event
time from the event set.

The above priority queue operations are the most fre-
quent operations required by a discrete event simulation
system and they are involved in any event scheduling
algorithm. Thus, it is clear that the main factor that
affects the efficiency of an event scheduling algorithm
is the structure of the event set.

The most important requirements of an event
scheduling algorithm are speed of operation and storage
economy. Many researchers have extensively studied
this field and presented both analytical and empirical
results concerning the time and space performance of
many event scheduling algorithms. They use different
data structures for the simulation of the event sets; that is,
linear lists, special kinds of trees, time-indexed lists,
two-level structures and many other. Moreover, they
use different techniques for the operations performed
by the scheduling algorithms; see, (Brown 1988, Franta
and Maly 1975, Franta and Maly 1977, Jonassen and
Dahl 1975, Kaplan, Shafrir, and Tarjan 2002, McCor-
mack and Sargent 1981, Reeves 1984, Tan and Thng
2000) for an exposition of the main results.

The data structures used for the simulation of the
event set can generally be classified under three types;
that is, lists, tree structures and multi-lists. Lists are
structures that are based on the simple linear list. They
include doubly linked lists, indexed lists (Nikolopoulos
and MacLeod 1993), SPEEDES Queue which is based
on the event horizon technique (Steinman 1992, Stein-
man 1994, Steinman 1996) and many other. Trees are
structures that are based on the simple binary tree, and
include binary heaps (Andreou and Nikolopoulos 1998,
Franta and Maly 1975, Franta and Maly 1977, Hwang
and Steyaert, to appear), skip lists (Nikolopoulos and
MacLeod 1993), priority trees (Jonassen and Dahl 1975,
Lewis and Deneberg 1991, Nikolopoulos and MacLeod
1993) which are studied here as well. Finally, multi-
lists are structures that are the result of a combination
of several types of lists. This is done in order to combine
the merits of two structures that may not perform as
well when implemented separately. Such structures are
the calendar queue and the SNOOPY calendar queue
(Tan and Thng 2000).

This paper describes efficient data structures, name-
ly the Indexed P-tree, Block P-tree, and Indexed-Block
P-tree (or IP-tree, BP-tree, and IBP-tree, respectively;
for short), for the simulation event set. All the struc-
tures, combine the advantages of both the P-tree and
the static representation of the list. The combination
of the P-tree and the list provides efficient data struc-
tures for the simulation event set in the case where the
event horizon technique is applied. The main feature
of each of our event set algorithm is the efficiency of the
merging process in the event horizon technique; that is,
the process of sorting the event notices of the secondary
queue and inserting them back into the event set. We
point out that, in the horizon technique the most time
consuming operation performed by the event set algo-
rithm is the merging process of the secondary queue
back into the main event set.

To gain insight into the performance of the IP-tree,
the BP-tree and the IBP-tree, and allow comparisons
with other selected algorithms (i.e., Dynamic-heap and
P-tree), they are coded and tested under a wide variety
of conditions in an experimental way. The objective was
to estimate the average complexity of each algorithm.
For this purpose, we used a revised definition of com-
plexity. That is, for a given configuration of event set
and a given distribution providing the scheduled time,
we estimate the time expected to be needed for the ex-
ecution of the Hold procedure (or Hold model).

Two main parameters affect the execution time of
the above operations. They are (i) the schedule time
T, and (ii) the size N of the event set. The parameter
T, which is given by a stochastic distribution, deter-
nines how long an event will remain in the event set.
Six stochastic distributions are especially chosen which
are not only representative of typical simulation prob-
lems but also capable of showing the advantages and
limitations of each algorithm. The parameter N de-
defines the notion of the small and large event sets. Tests
were performed with values of N from 64 (small event
set) to 262144 (large event set). This range is represen-
tative of actual simulations and the behaviour of the
algorithms for N > 262144 can be extrapolated from
the results.

The results of this work show that the IP-tree algo-
rithm combines time performance, storage economy
and simplicity of coding. The BP-tree and the IBP-
tree algorithms outperform the IP-tree algorithm, and
the BP-tree algorithm has a slightly better performance
than the IBP-tree algorithm.

The paper is organized as follows. Section 2 presents
the main features of the Hold model and the event hori-
zon technique. Section 3 describes the P-tree structure
which is the structure that our approach is based on.
The IP-tree, the BP-tree and the IBP-tree structures
are described in Sections 4, 5 and 6, respectively. An
experimental evaluation of the algorithms is presented
in Section 7, where we also compare the performance of
the algorithms. Finally, Section 8 concludes the paper
with a summary of our results.

2 HOLD AND EVENT HORIZON

As already mentioned, the two basic operations per-
formed on the event set by an event set algorithm are
(i) insertion of a new event notice into event set, and (ii) determination and deletion of the notice of the next event. A standard metric for comparison of the performance of an event set algorithm is the time required for a Hold operation, which combines both insertion and deletion operations (Andreo and Nikolopoulos 1998, Franta and Maly 1975, Law and Kelton 2000, Mitranik 1982). Under the Hold model, event notices are repeatedly deleted and then re-inserted with a randomly reduced priority; this sequence of operations is known as a Hold operation. The hold operation works as follows:

1. Determine and remove the event notice with the minimum event time $T_{min}$ from the event set; that is, the current notice.

2. Increase the event time value of the current notice by $T$, where $T$ is a random variate distributed according to some distribution $F(t)$, and

3. Re-insert the new notice back into the event set; it now has $T_{new} = T_{min} + T$ event time.

The Hold model has two parameters: $N$, the number of notices in the event set, and $F$, the distribution used to determine the time an inserted event will occur. Thus, the model allows the average combined time for insertion and deletion to be measured as a function of the size of the event set and the stochastic distribution.

The event horizon is a fundamental concept that applies to both parallel and sequential discrete event simulations (Rao and Kumar 1988, Steinman 1992, Steinman 1994, Steinman 1996). Using event horizon one can improve the performance of several event sets; that is, priority queue data structures such as linked lists and various binary trees.

In order to exploit the event horizon for event set management algorithms, it is assumed that as new events are generated they are not inserted into the main priority queue data structure immediately; they are collected in an unsorted temporary (secondary) queue in such a way that one can always track the event with the earliest event time. As a result, when the event to be processed happens to be in the secondary queue, the queue is sorted and then it is “merged” back into the main priority queue.

The secondary queue is most frequently a linked list, providing the advantage of inserting a new event in constant time since the list is kept unsorted; it is sorted just before the merging process. Merging the two data structures, however, is not always a simple process. The main priority queue (event set) itself may be a very complicated data structure.

3 P-TREE

A Priority-tree (or P-tree) is either empty or it is a sorted, non-increasing sequence of nodes, the “left path”, such that to each node of the left path except the last one, is associated a P-tree (possibly empty), the “right subtree”. The nodes of the right subtree associated with a node $x$ on the left path, are ranked between $x$ and the left successor of $x$ (Johansen and Dahl 1975, Lewis and Deneberg 1991, Nikolopoulos and MacLeod 1993).

Neglecting node values, a binary tree is a P-tree if and only if each node having a right successor also has a left successor. The terminal node on the leftmost path is the element with the smallest key value. In order to insert a new element $x$ into a P-tree $T$, the algorithm P-insert below is applied recursively.

*P-insert $x$ into $T$:

1. If $T = \emptyset$ or $x.v \geq T.v$, let $x$ be the new root and $T$ its left subtree;

2. Otherwise search down the left path of $T$ for the first node $y$, if any, such that $y.v \leq x.v$;

   2.1 If none, append $x$ as the new left leaf;

   2.2 Otherwise $y.v \leq x.v \leq z.v$, where $z$ is the predecessor of $y (y = z.l)$. P-insert $x$ into the right subtree of $z$;

where $x$, $y$ and $z$ denote nodes, $u.l$ and $u.v$ denote the left subtree and the node value of $u$, respectively.

The detection of the event notice with the earliest time value can be performed in constant time provided that there is an additional pointer to the terminal notice on the left path. After the removal of this notice, the last right subtree, if it is not empty, is appended to the left path.

4 THE INDEXED P-TREE

An Indexed P-tree (or IP-tree) consists of a tree structure, the P-tree, and a static representation of a list structure, the $L$-list. The elements of the $L$-list point into specific event notices in the $P$-tree; see Figure 1. Using the event horizon technique, when the event horizon is crossed (i.e., when the event to be processed happens to be in the secondary queue), the secondary priority queue is sorted and then it is merged back into the main priority queue ($P$-tree).

We next describe the main operations performed in a discrete event simulation system using the $IP$-tree for the simulation of the event set.
Figure 1: An IP-tree structure; it consists of a list of pointers, called I-list, and a P-tree.

(i) **Insert operation:** According to the event horizon technique, a new event notice is inserted in the secondary queue, which is a linked list structure. As the secondary queue is kept unsorted, the insertion of a new event notice can be completed in constant time.

(ii) **Delete operation:** The deletion of the current event notice from the IP-tree structure can be implemented in constant time as it only involves deleting the current event notice from a P-tree structure.

(iii) **Merge operation:** Suppose that the event notices of the secondary queue have to be inserted into a standard P-tree structure. To this end, for each event notice the P-tree event set algorithm scans the whole P-tree, starting each time from the root of the tree, in order to determine its proper insertion position. The IP-tree event set algorithm takes advantage of the fact that the event notices in the secondary list are sorted in decreasing order and the I-list determines some specific subtrees of the P-tree; recall that every subtree of a P-tree is a P-tree itself. Thus, for each event notice the IP-tree algorithm scans the I-list and determines the proper insertion subtree. Then, it proceeds as the P-tree algorithm and completes the insertion operation. Thus, in order to insert the event notices of the secondary list into the IP-tree there is no need to scan the whole P-tree for each notice, meaning that the P-insert operation, as it was described before, is not necessary to start from the root of the P-tree.

In the merging process, some of the subtrees that are not scanned during an insertion operation will not be scanned by the next insertion operation either, as the time-value of the event notice of the second operation is less or equal than the time-value of the event notice of the first operation. Taking advantage of this knowledge, the I-list is constructed in order to make the merging operation more efficient. In particular, if an event notice, say \( b \), is the next event to be inserted into the P-tree and the event notice which was last inserted, say \( a \), had a greater event time, we do not need to check a notice, say \( c \), that \( a \) was compared with and was found to be less than \( c \). Thus, we need to keep pointers to the event notices of the P-tree that \( a \) was compared with and then moved to a right subtree; see Figure 1. We shall call these specific notices I-notices.

An example of an IP-tree structure is presented in Figure 1. Let \( a \) be the last event notice which has been inserted into the P-tree and let \( 14 \) be its event time. The pointers of the I-list were pointed at notices with event times 18 and 17. Let \( b \) be the next event notice which has to be inserted into the P-tree and let \( 13 \) be its event time. Then, the time of the notice \( b \) is compared only with the times of the left children of the I-notices. Thus, a search is performed on the I-list and the I-notice that its left child has the greatest event time which is less than the time of \( b \) is determined; let \( c \) be such an I-notice. Then, the IP-tree algorithm \( P \)-inserts the notice \( b \) into the P-tree rooted at \( c \). Recall that every subtree of a P-tree is also a P-tree.

5 **THE BLOCK P-TREE**

The Block P-tree (or BP-tree) structure is a P-tree that consists of nodes containing an array of an initially fixed number of elements, say \( S \), which we call supernodes. The elements of every supernode are kept sorted in increasing order and the P-tree property is applied on the event with the earliest event time of each supernode. In other words, the position of a supernode in the BP-tree is determined by the earliest event time that it contains.

Inserting a new event notice is a very simple process. As the BP-tree algorithm takes advantage of the event horizon technique, the new event notice can be inserted in constant time in the secondary priority queue which is a static representation of the list structure.

The deletion of the event notice with the earliest event time is quite simple as well. The current supernode, that is the supernode containing the event
The IBP-tree algorithm takes advantage of the event horizon technique; that is, the new event notice can be inserted in constant time in the secondary priority queue which is a static representation of the list structure.

The deletion of the event notice with the earliest event time is quite simple as well and similar to the deletion operation of the BP-tree algorithm. The current supernode is tracked in constant time as there is a pointer variable pointing to it. Since the elements of the supernodes are sorted, the current event notice can be located in constant time. After extracting it, the BP-tree may need to be updated. This operation is similar to that performed by the BP-tree algorithm. Note that, despite the fact that the deletion operation may result to supernodes having less than S elements, it is essential that the supernodes cannot consist of more than S elements.

Using the event horizon technique, when the event horizon is crossed, the secondary priority queue, which is a static representation of a list structure, is sorted in increasing order. Then, the secondary queue forms a single supernode which is then reinserted back into the main priority queue (BP-tree). The advantage of this implementation is that only one supernode has to be inserted into the BP-tree. Note that, the number of the elements of each supernode can exceed the value of S, as the secondary queue can consist of more than S elements when the event horizon is crossed.

6 THE INDEXED BLOCK P-TREE

The Indexed-block P-tree (or IBP-tree) structure is based on the IP-tree and the BP-tree structures; it consists of an I-list and a BP-tree. Recall that the I-list is a static representation of the list structure and its elements point into specific event notices of the BP-tree; see Figure 2.

The insertion operation is a very simple process, since

Figure 2: An IBP-tree structure; it consists of an I-list, and a BP-tree with $S = 3$. 

The main motivation for the empirical studies performed so far comes from the fact that most of the
Table 1: The Six Distributions

(A) Unimodal
- EXP: Negative exponential (mean 1).
- U02: Uniform distribution over the interval [0, 2].
- U09: Uniform distribution over [0.9, 1.1].

(B) Bimodal
- BIM: 0.9 probability - uniform over [0, S].
- 0.1 probability - uniform over the interval [100S, 101S], where S is chosen to give
  the mixed distribution an average of unity.

(C) Discrete
- D1: T is constant with value of unity.
- D012: T is assigned the values 0, 1 or 2 with equal probabilities.

Theoretical performance bounds associated with the different event set algorithms hide significant constant factors. In addition, these results are usually expressed using different concepts such as expected case, worst case and amortized case bounds. Given the number of alternatives for implementing the event set and the need for solutions that are efficient in practice, the empirical studies have then arisen as an effective tool to evaluate the performance of an event set algorithm.

7.1 Test Conditions

Most of the research performed to date uses the Hold procedure to estimate the average time complexity of an event set algorithm. The time needed for the execution of the Hold operations is the measure for estimating the average time complexity. Obviously, the data structure chosen to simulate the event set as well as the size of the event set affect the processor time required for the Hold operations (insertion and deletion). Tests were performed for $N = 2^k$, $k = 6, 7, \ldots, 18$, where $N$ is the size of the event set; that is, the number of event notices in the event set.

A crucial step in designing the tests lies in the selection of the stochastic distribution which provides the event time $T$; that is, the parameter for the Hold procedure that determines how long an event notice remains in the event set. Six distributions have been chosen because they differ in their characteristics and reveal the advantages and the disadvantages of an algorithm; see Table 1. Each test includes the following operations:

1. generate $N$ event notices with each one having event time that is generated by the distribution $F$ and insert them into the event set.
2. without counting time, execute $1.6 \times 10^6$ times the Hold procedure with the distribution $F$.

Table 2: Dynamic-heap Algorithm: Test Results

<table>
<thead>
<tr>
<th>$N$</th>
<th>U02</th>
<th>U01</th>
<th>EXP</th>
<th>BIM</th>
<th>D1</th>
<th>D012</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>5.11</td>
<td>5.13</td>
<td>5.42</td>
<td>6.68</td>
<td>5.10</td>
<td>5.13</td>
</tr>
<tr>
<td>256</td>
<td>5.85</td>
<td>5.90</td>
<td>6.20</td>
<td>7.42</td>
<td>5.88</td>
<td>5.87</td>
</tr>
<tr>
<td>1024</td>
<td>6.60</td>
<td>6.58</td>
<td>6.88</td>
<td>8.14</td>
<td>6.51</td>
<td>6.57</td>
</tr>
<tr>
<td>4096</td>
<td>7.41</td>
<td>7.42</td>
<td>7.71</td>
<td>8.95</td>
<td>7.29</td>
<td>7.37</td>
</tr>
<tr>
<td>16384</td>
<td>8.36</td>
<td>8.35</td>
<td>8.76</td>
<td>9.95</td>
<td>8.00</td>
<td>8.11</td>
</tr>
<tr>
<td>65536</td>
<td>9.35</td>
<td>9.33</td>
<td>9.62</td>
<td>10.93</td>
<td>8.83</td>
<td>8.94</td>
</tr>
<tr>
<td>262144</td>
<td>10.53</td>
<td>10.54</td>
<td>10.89</td>
<td>12.12</td>
<td>9.54</td>
<td>9.74</td>
</tr>
</tbody>
</table>

Table 3: $P$-tree Algorithm: Test Results

<table>
<thead>
<tr>
<th>$N$</th>
<th>U02</th>
<th>U01</th>
<th>EXP</th>
<th>BIM</th>
<th>D1</th>
<th>D012</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>3.27</td>
<td>2.67</td>
<td>3.78</td>
<td>7.87</td>
<td>2.33</td>
<td>3.38</td>
</tr>
<tr>
<td>256</td>
<td>4.02</td>
<td>2.99</td>
<td>4.48</td>
<td>16.15</td>
<td>2.33</td>
<td>6.80</td>
</tr>
<tr>
<td>1024</td>
<td>5.33</td>
<td>3.50</td>
<td>5.42</td>
<td>31.42</td>
<td>2.33</td>
<td>17.77</td>
</tr>
<tr>
<td>4096</td>
<td>7.82</td>
<td>4.42</td>
<td>6.41</td>
<td>58.61</td>
<td>2.35</td>
<td>73.14</td>
</tr>
<tr>
<td>16384</td>
<td>12.66</td>
<td>6.10</td>
<td>7.83</td>
<td>102.90</td>
<td>2.36</td>
<td>308.95</td>
</tr>
<tr>
<td>65536</td>
<td>22.00</td>
<td>9.26</td>
<td>9.52</td>
<td>141.88</td>
<td>2.39</td>
<td>1253.46</td>
</tr>
<tr>
<td>262144</td>
<td>54.41</td>
<td>20.78</td>
<td>13.69</td>
<td>280.26</td>
<td>2.42</td>
<td>4905.45</td>
</tr>
</tbody>
</table>

3. execute $1.6 \times 10^6$ Hold operations and count the total processor time (CPU time) needed to complete them.

Operation (1) initializes the system while operation (2) allows it to reach a steady state. Operation (3) yields a measure of the complexity of the tested algorithms. The algorithms were coded in C programming language and the experimental results were taken from Sun-Blade-1000, 2 x 750 MHz Ultrasparc-III processors (8MB cache), 512MB RAM.

The IP-tree, the BP-tree and the BIP-tree algorithms were coded and run to collect evidence of their performance under realistic conditions. For comparison reasons, well-known event set algorithms were also coded and run under the same conditions; that is, the Dynamic-heap and the $P$-tree algorithms. To gain insight into the performance of the proposed algorithms and allow comparisons with the other event set algorithms, they were tested under a wide variety of conditions in an experimental way. The experimental results for each algorithm (that is the time in seconds needed to complete each algorithm) are represented in the form of tables and graphs; see, Tables 2–6 and Figures 3–8.

7.2 Dynamic-heap and $P$-tree: Hold model

As expected, the Dynamic-heap algorithm provides a very good time performance. The performance results are given in Table 2. What is observed is the expected logarithmic behavior of a heap data structure and
the fact that the time performance of the event set algorithm is almost the same with all the distributions. We note that the Static-heap algorithm has the same performance.

One can easily observe (see Table 3), that the performance of the $P$-tree is not as good as the performance of the heap algorithm. Its CPU times increase with the variance of the scheduling distribution. It is remarkable that the $P$-tree is efficient under constant values (D1 distribution). This performance was expected because each new event notice becomes the new root of the $P$-tree, which in this case is a sorted linked list, and thus the new event is inserted in constant time. Its performance is extremely worst with the discrete D012 distribution. Furthermore, the $P$-tree algorithm becomes even more inefficient as the size of the event set increases. The experimental results showed that the performance of the $P$-tree algorithm cannot be improved by applying the event horizon technique.

### 7.3 IP-tree, BP-tree and IBP-tree: Event Horizon

The experimental results of the performance of the IP-tree algorithm are presented in Table 4. These show the superiority of the IP-tree compared to the $P$-tree algorithm and its excellent performance with all the sizes of the event set and all the stochastic distributions. Specifically, we observe that the CPU time for the D012 distribution is extremely decreased compared with the results taken by the $P$-tree algorithm. In addition, one can observe that the CPU times for all the distributions are almost the same.

The experimental results of the performance of the BP-tree algorithm are presented in Table 5. The results show the superiority of the algorithm compared to the heap algorithm, and also to the $P$-tree and IP-tree algorithms. Furthermore, its performance is slightly better than the performance of the IBP-tree, apart from the Exponential and the discrete D012 distributions when $N > 65536$. Note that the value of the parameter $S$ is equal to the size of the event set; that is $S = N$. Recall that the parameter $S$ determines only the initial size of the supernodes because the size changes as the secondary queue becomes a supernode every time the event horizon is crossed. The experimental results showed that the algorithm performs better when the BP-tree has initially only one supernode which contains $N$ elements; that is when $S = N$.

Table 6 presents the experimental results of the performance of the IBP-tree algorithm. Its excellent performance, regardless of the size of the event set or the stochastic distribution, show the superiority of the algorithm over the IP-tree and the heap algorithms. The IBP-tree algorithm outperforms the IP-tree algorithm because it takes advantage of the properties of the latter and, in addition, the size of the event set can be considered to be smaller as the event notices form supernodes. Thus, if the size of the event set is equal to $N$, the IP-tree algorithm produces a $P$-tree containing $N$ nodes, while the IBP-tree algorithm can produce a BP-tree much smaller, having $N/S$ nodes, if the value of $S$ is sufficiently large. The experimental results show that the IBP-tree algorithm has the best performance when $S \approx 3000$. Consequently, when $N < S$ the IBP-tree algorithm behaves as the BP-tree algorithm.

### 7.4 A Comparison of the Algorithms

The experimental results show that the IP-tree algorithm has an extremely better performance than the $P$-tree algorithm. The latter becomes very inefficient as the size of the event set increases, especially with the D012 distribution. The BP-tree and IBP-tree al-

---

**Table 4: IP-tree Algorithm: Test Results**

<table>
<thead>
<tr>
<th>$N$</th>
<th>U02</th>
<th>U91</th>
<th>EXP</th>
<th>BIM</th>
<th>D1</th>
<th>D012</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4.04</td>
<td>3.81</td>
<td>4.39</td>
<td>5.26</td>
<td>3.46</td>
<td>3.72</td>
</tr>
<tr>
<td>256</td>
<td>4.46</td>
<td>4.17</td>
<td>4.88</td>
<td>6.37</td>
<td>3.66</td>
<td>4.01</td>
</tr>
<tr>
<td>1024</td>
<td>4.89</td>
<td>4.52</td>
<td>5.36</td>
<td>7.00</td>
<td>3.88</td>
<td>4.32</td>
</tr>
<tr>
<td>4096</td>
<td>5.43</td>
<td>5.02</td>
<td>5.98</td>
<td>7.79</td>
<td>4.13</td>
<td>4.72</td>
</tr>
<tr>
<td>16384</td>
<td>6.12</td>
<td>5.65</td>
<td>6.76</td>
<td>8.36</td>
<td>4.48</td>
<td>5.28</td>
</tr>
<tr>
<td>65536</td>
<td>6.86</td>
<td>6.21</td>
<td>7.50</td>
<td>8.52</td>
<td>4.74</td>
<td>5.72</td>
</tr>
<tr>
<td>262144</td>
<td>9.08</td>
<td>8.50</td>
<td>9.97</td>
<td>9.96</td>
<td>5.15</td>
<td>7.78</td>
</tr>
</tbody>
</table>

**Table 5: BP-tree Algorithm: Test Results**

<table>
<thead>
<tr>
<th>$N$</th>
<th>U02</th>
<th>U91</th>
<th>EXP</th>
<th>BIM</th>
<th>D1</th>
<th>D012</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>3.92</td>
<td>3.45</td>
<td>4.38</td>
<td>4.73</td>
<td>3.26</td>
<td>3.42</td>
</tr>
<tr>
<td>256</td>
<td>4.44</td>
<td>3.78</td>
<td>4.96</td>
<td>5.29</td>
<td>3.47</td>
<td>3.67</td>
</tr>
<tr>
<td>1024</td>
<td>4.98</td>
<td>4.13</td>
<td>5.58</td>
<td>6.15</td>
<td>3.66</td>
<td>3.92</td>
</tr>
<tr>
<td>4096</td>
<td>5.60</td>
<td>4.55</td>
<td>6.30</td>
<td>6.96</td>
<td>3.95</td>
<td>4.23</td>
</tr>
<tr>
<td>16384</td>
<td>6.30</td>
<td>5.08</td>
<td>7.09</td>
<td>7.41</td>
<td>4.24</td>
<td>4.66</td>
</tr>
<tr>
<td>65536</td>
<td>7.10</td>
<td>5.57</td>
<td>8.02</td>
<td>7.99</td>
<td>4.46</td>
<td>5.03</td>
</tr>
<tr>
<td>262144</td>
<td>8.52</td>
<td>7.69</td>
<td>9.94</td>
<td>9.94</td>
<td>4.77</td>
<td>6.74</td>
</tr>
</tbody>
</table>

**Table 6: IBP-tree Algorithm: Test Results**

<table>
<thead>
<tr>
<th>$N$</th>
<th>U02</th>
<th>U91</th>
<th>EXP</th>
<th>BIM</th>
<th>D1</th>
<th>D012</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>4.07</td>
<td>3.50</td>
<td>4.58</td>
<td>5.76</td>
<td>3.41</td>
<td>3.54</td>
</tr>
<tr>
<td>256</td>
<td>4.52</td>
<td>3.81</td>
<td>5.08</td>
<td>6.59</td>
<td>3.58</td>
<td>3.73</td>
</tr>
<tr>
<td>1024</td>
<td>5.03</td>
<td>4.15</td>
<td>5.65</td>
<td>7.30</td>
<td>3.76</td>
<td>3.96</td>
</tr>
<tr>
<td>4096</td>
<td>5.65</td>
<td>4.59</td>
<td>6.33</td>
<td>7.84</td>
<td>4.04</td>
<td>4.25</td>
</tr>
<tr>
<td>16384</td>
<td>6.35</td>
<td>5.11</td>
<td>7.14</td>
<td>8.05</td>
<td>4.35</td>
<td>4.68</td>
</tr>
<tr>
<td>65536</td>
<td>7.14</td>
<td>5.59</td>
<td>8.06</td>
<td>8.25</td>
<td>4.56</td>
<td>5.01</td>
</tr>
<tr>
<td>262144</td>
<td>8.66</td>
<td>7.66</td>
<td>9.91</td>
<td>9.45</td>
<td>4.91</td>
<td>6.63</td>
</tr>
</tbody>
</table>
algorithms are even more efficient than the \textit{IP}-tree algorithm as their performance is excellent regardless of the size of the event set or the distribution that is used.

What is also remarkable is the fact that the \textit{BP}-tree and the \textit{IBP}-tree algorithms provide results which are better even than the results of the well known efficient Dynamic-heap algorithm. The latter performs, as expected, better than the \textit{P}-tree algorithm regardless of the distribution which is used. Furthermore, the \textit{BP}-tree and the \textit{IBP}-tree algorithms outperform all the
other algorithms and their superiority can easily be concluded. Figures 3–8 present the performance of each one of the algorithms under the six distributions.

We would like to comment on the logarithmic behaviour of the BP-tree and the IBP-tree algorithms. One can easily observe from Figures 3–8 that the two algorithms behave like the Dynamic-heap algorithm. Furthermore, the performance of the IP-tree algorithm resembles the performance of the P-tree algorithm.

What should also be pointed out is that the event horizon technique, when applied to the Dynamic-heap algorithm or the P-tree algorithm, does not result to a better performance. Applying the event horizon principle to the heap algorithm involves using a heap structure (static representation) as a main event set and a list data structure (unsorted array) as a secondary event set. When the minimum event time is found to be in the secondary list, its elements are merged back into the main priority queue data structure (merge operation). The array is kept unsorted because experimental results showed that the performance of the Static-heap is not improved in the case where the secondary list is sorted either in increasing or decreasing order. Furthermore, it was observed that the event horizon technique does not affect the performance of the Dynamic-heap algorithm; that is, the performance of the algorithm with the event horizon technique is almost the same as without it.

In the P-tree algorithm the secondary data structure is an unsorted linked list. When the next-event to be processed (event notice with the minimum event time) happens to be in the secondary list, the latter is sorted in an increasing order and its elements are placed back into the main event set. The experimental results showed that when we apply event horizon the CPU times taken by the P-tree algorithm are slightly increased for all the distributions except for the D012 distribution.

8 CONCLUDING REMARKS

The P-tree structures proposed in this paper could usefully replace the classic P-tree structure, as well as the heap structure, for the simulation event set in a general purpose discrete event simulation system. The processor time obtained with the IP-tree, the BP-tree and the IBP-tree algorithms is relatively insensitive to variations in the scheduling distributions or the number of event notices in the event set, and points to their superiority over the P-tree structure, and also over the Dynamic-heap. Our proposed structures provide time efficiency, size flexibility and space economy.

Future work might involve how the BP-tree or IBP-tree algorithms can be efficiently parallelized. Furthermore, it would be interesting to study the performance of algorithms that use other tree-like data structures under the event horizon technique and/or the I-list technique.

In closing, we point out that the results of this work prompts us to suggest the BP-tree and the IBP-tree as efficient data structures for the simulation event set in a general purpose discrete event simulation system.

REFERENCES


Reeves, C. M. 1984. Complexity analyses of event set

AUTHOR BIOGRAPHIES

KATERINA ASDRE received the B.Sc degree (with Honours) and the M.Sc degree in Computer Science from the University of Ioannina, Greece, in 1999 and 2001, respectively. She is currently a Ph.D student in the Department of Computer Science at the University of Ioannina, Greece. Her research interests are in parallel algorithms, data structures and algorithms, graph algorithms, simulation techniques, discrete event systems. Her e-mail address is <katerina@cs.uoi.gr>.

STAVROS D. NIKOLOPOULOS received the B.Sc degree (with Honours) in Mathematics from the University of Ioannina, Greece, in 1982, the M.Sc degree in Computer Science from University of Dundee, Scotland, in 1985, and the Ph.D degree in Computer Science from the University of Ioannina, in 1991. He was a Research Assistance in the Sacylant Undersea Research Centre, Italy, in 1990-91. From 1992 to 1996 he was a Lecturer in the Department of Computer Science, University of Cyprus, Nicosia. In 1996 he joined the Department of Computer Science, University of Ioannina, Greece, where he is currently an Associate Professor. His research interests focus on parallel algorithms, data structures and algorithms, graph theory, graph modelling, combinatorial and graph algorithms, combinatorial mathematics, simulation techniques, discrete event systems. His e-mail address is <stavros@cs.uoi.gr>, and his web page is <www.cs.uoi.gr/stavros>.