Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1992

An Application of Program Unification to Priority Queue Vectorization

Ling-Yu Chuang

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Aditya P. Mathur
Purdue University, apm@cs.purdue.edu

Report Number:
92-034

Chuang, Ling-Yu; Rego, Vernon J.; and Mathur, Aditya P., "An Application of Program Unification to Priority Queue Vectorization" (1992). *Department of Computer Science Technical Reports.* Paper 956.
https://docs.lib.purdue.edu/cstech/956

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

AN APPLICATION OF PROGRAM UNIFICATION TO PRIORITY QUEUE VECTORIZATION

Ling-Yu Chuang
Vernon Rego
Aditya Mathur

CSD-TR-92-034
May 1992

An Application of Program Unification to Priority Queue Vectorization*

Ling-Yu Chuang, Vernon Rego[†] and Aditya Mathur
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

In this experimental study, we examine the performance of a variety of unified priority queue implementations on a CRAY Y-MP. The scope of this study is restricted to determining if different implementations of priority queues exhibit markedly different performance characteristics under program unification. We found the answer to this question to be in the affirmative. In a larger view, this result has interesting consequences in the application of program unification to discrete event simulation applications - which is where our motivation lies. We find ordered lists and heaps to be promising priority queue data structures in the unified simulation paradigm; ordered lists and heaps work well with small/moderate-sized and large-sized event lists, respectively.

*This research was supported by the National Science Foundation under Grant No. ASC-9002225, and by NATO under Grant CRG 900108.

[†]Also supported in part by the Mathematical Sciences Section of Oak Ridge National Laboratory under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

1 Introduction

Priority queues are well-known discrete structures in computer science applications with uses as varied as in the ordering of states in enumeration problems, optimization in search problems such as graph traversals, sorting problems and task scheduling in operating systems [8]. Perhaps the most popular use of priority queues is in the area of event-scheduling in simulation applications, in particular, discrete event simulations. In a discrete event simulation, an analyst is typically interested in modeling a stochastic process with either a discrete or continuous parameter (time), over a discrete space which is made up of various types of events. How closely the simulation mimics the actual stochastic process is largely a function of the ability of the analyst to capture model essentials and the quality of the random number generator.

Discrete event simulations can be viewed as self-perpetuating programs in the following sense. A simulation which begins with one or a few scheduled events ensures that, as scheduled events occur in time, a sufficient number of new events are generated so that program execution can continue indefinitely. The scheduling mechanism that effects this behaviour does so by using a list, called an event list, to store events which are supposed to occur at some future time. These pending events remain in the list, in some data-structure dependent order, until the simulation program decides to extract an event at a time for processing. New events are scheduled and inserted into the list according to model specifications, and these are usually generated while some related event is being processed. The latter event, once removed from the list, represents the event whose time of occurrence coincides with the current time and is hence called the *current event.*

After processing the current event and possibly scheduling some new events, the simulation program retrieves from the event list an event with the highest priority. Because of this, a priority queue is used to implement the event list. The event with highest priority is typically one with the nearest scheduled time of occurrence, though, in certain situations (e.g., preemptive service in queuing systems) an event with a larger scheduled time of occurrence may have higher priority. For convenience, we will not concern ourselves with such a situation in this study. Additionally, though it is possible for two or more events to have the same priority (in which case the analyst is responsible for specifying how the program must break ties), we will assume that events occur at distinct times. Neither assumption affects the arguments or the results of this study.

Because of its importance in a variety of applications, a considerable amount of work has been done on priority queues. During the last decade, several new data structures have been proposed for the implementation of priority queues, including splay trees, skew heaps [13], skip lists [9], and an algorithm due to Henriksen [4]. Since discrete event simulation applications tend to require hours, and even days or weeks of processing time, and since a considerable portion of this time is consumed by event list processing, several researchers

have devoted effort towards either analyzing the algorithmic complexity (e.g., [7, 10]), or empirically measuring the execution time (e.g., Jones [5]) of various priority queue implementations. In an interesting study, Jones [6] proposes that some implementations, particularly skew heaps, lend themselves to efficient concurrent operations, thus indicative of potential for speedup in shared memory multiprocessor settings. In a theoretical vein, Olariu and Wen designed parallel initialization algorithms for priority queues [8].

In all previous studies of the efficiency of priority queue implementations, the focus has been either on scalar uniprocessing or scalar multiprocessing (in a shared memory system) of a single priority queue [6, 8]. This is a natural feature to study when the underlying application is assumed to be a single simulation, generating a single sample-path of the stochastic process of interest. In contrast, our focus is on using vector machines to generate several (usually, but not necessarily, independent) sample-paths of the process in parallel. The device used to achieve this is the technique of program unification [12] which allows for the algorithmic transformation of a sequential program into a unified, vector program. However, in order to obtain effective unified simulations, one requires an understanding of how different unified priority queue implementations perform on vector machines. Therefore, the purpose of this study is to conduct experiments through which we can compare various priority queue implementations and possibly identify one which exhibits good performance in the unified setting, so that we may proceed to use it for unified simulation applications. In this experimental study, we apply the program unification technique to five different kinds of priority queue implementations.

The following section contains a description of priority queues and how they may be unified for efficient execution on vector machines. For ease of explanation, this is done with the aid of examples. Section **3** contains a discussion on experimental methodology, and Section 4 contains the results of our experiments. In Section **5** we conclude the paper and outline some future work.

2 On Unifying Priority Queues

In this section we briefly describe the priority queue abstract data type, the required operations for insertion and deletion of items from a priority queue, and the application of unification to priority queues. Though there exist a number of specific implementations for a priority queue structure, for ease of explanation we will restrict our attention to a very special implementation called the *heap* or *implicit heap* [1, 5]. The heap data structure is an efficient implementation with a complexity of O(log n) for both insertion as well as deletion of items from an n item priority queue.

2.1 Priority Queues

The priority queue is an abstract data type based on the set model with the operations **insert** and **deletemin**, as well as the usual **makenull** for initialization of the data structure [1]. The operations **insert** and **deletemin** are sometimes also called **enqueue** and **dequeue** [6], respectively. Each item in a priority queue has a priority value. In discrete event simulations, an item is an event (typically a *tuple*) and its priority is its scheduled time of occurrence. Along with its priority, there will usually be other values of interest associated with each item. For example, in the simulation domain, at the very minimum, an event can be expected to have a *type* associated with it. The operation **deletemin** retrieves the item with highest priority from the queue; the operation **insert** places a newly generated item into the queue, in some priority-dependent position. As is usual in discrete event simulation applications, we will assume that for two distinct items the priority of the first is greater than the priority of the second if the first item has a smaller scheduled time of occurrence. This explains the usage of the term **deletemin** in describing the operation of retrieving the item with highest priority.

2.2 The Implicit Heap

A binary *heap* or *implicit heap* is an array implementation of a priority queue [1], essentially based on a binary tree structure. The structure is maintained by ensuring that the priority of a given node is equal to or higher than the priority of each of its children, given that each node may have one or at most two children. This is called the *heap property*. The **deletemin** operation proceeds by extracting the root node from the data structure, since the root is defined to be the node which contains the queue item with highest priority. Since this operation effectively breaks the structure apart, it is necessary to do some postprocessing in order to recreate an appropriate binary tree satisfying the heap property. This entails temporarily placing the last item of the heap (which is the rightmost leaf in the lowest level of the tree) in the root node. Next, a percolation process is initiated during which this item, in the root node, is compared with each of its children; a swap in position occurs, if necessary, for the heap property to be maintained. In this way, the displaced item percolates to an appropriate position in the binary tree while some other item moves up to occupy the root node and thus restore the binary tree structure.

In Figure 1 is shown the original source code (in Fortran) for the **deletemin** operation in a binary heap. Fortran was our language of choice because of its performance on vector machines like the CRAY-Y/MP and Alliant FX/80. Since a priority queue structure is typically used with only a single list of items, the data declarations allow for only one queue called *iheap*. The *event* and *clock* variables are used to contain the event code and the time of occurrence of the event that is currently being processed (i.e., current clock value),

```
        C* data declaration
1d.           real event
2d.           real clock
3d.           real iheap(1003, 2), tempiheap(1,2)
4d.           integer index
5d.           integer j, j2
        C*  Begin the deletemin operation
        C** Extract the root item from the heap
1.            event = iheap(1,1)
2.            clock = iheap(1,2)
        C** Temporarily place the last heap item in root
3.            iheap(1,1) = iheap(index,1)
4.            iheap(1,2) = iheap(index,2)
        C** Decrement the heap size accordingly
5.            index = index -1
        C** Begin the percolation phase
        C** starting at the root, working downwards
6.            j = 1
7.            j2 = 2*j
        C** Node j2 , j2+1 are the children of node j
        C** If we arrive at lowest level in tree, then stop
8.      2050  if( j .gt. (index / 2)) goto 2060
9.            if ((j2 .ne. index) .and.
10.               (iheap(j2,2) .gt. iheap(j2+1, 2))) then
11.                   j2 = j2+1
12.           endif
        C* Else, compare father and child nodes,
13.           if (iheap(j,2) .le. iheap(j2,2) ) goto 2060
        C* performing an item swap if necessary
14.           tempiheap(1,1) = iheap(j,1)
15.           tempiheap(1,2) = iheap(j,2)
16.           iheap(j,1) = iheap(j2,1)
17.           iheap(j,2) = iheap(j2,2)
18.           iheap(j2,1) = tempiheap(1,1)
19.           iheap(j2,2) = tempiheap(1,2)
20.           j = j2
21.           j2 = j*2
22.           goto 2050
23.     2060  continue
```

Figure 1: Code for the heap dequeue operation

respectively. In this example, each queue item is a two-tuple containing an event's code and its time of occurrence. The two-tuple *tempiheap* is used for temporarily storing event information while heap nodes swap their contents.

In Lines 1 and 2, the current event is extracted from the root of the heap and simulation time is advanced by advancing the clock to the time of the extracted event. Next, the rightmost leaf at the lowest level in the tree is brought into the root. Decrementing variable *index* accordingly decreases the size of the heap. Following Line 5, the percolation process is initiated for heap maintenance. The contents of the root may move down a number of levels, to settle at some appropriate position in the tree, while some other node's contents move into the root node. The number of steps required depends on the actual priority values compared, and ranges anywhere from one to a number equal to the height of the tree.

2.3 Heap Unification

Execution of a discrete event simulation program proceeds through a sequence of operations performed on its priority queue. An *instance* of the simulation program corresponds to execution of the program on particular input parameters; repeated execution typically involves varying input parameters. Instead of performing the operations sequentially, and repeatedly executing the program with different parameters, envision a single program which operates on a number of priority queues simultaneously. A single complete execution of the latter program would replace several repeated executions of the former. *Program unification* is a technique for source-to-source transformation of such programs. It exploits the parallelism that arises when multiple instances of a program are executed on simultaneously available, distinct data sets [12]. The term "unification" reflects the fact that multiple instances of a program are combined into a single unified program.

The idea is to replace scalar operations on a single queue structure by vector operations on a vector queue structure. While this may seem a trifle strange at first glance, our initial experimental work [11] has shown that there is considerable potential for speedup when one attempts to generate several trajectories of a stochastic process in parallel, instead of a single trajectory at a time. For simulation applications in particular, where a simulation is to be repeated several times in order for proper statistical results to be obtained, this makes immense sense. Further, when such an advantage can be had at little or no additional expense to the analyst, there seems to be no case to be made against attempting to exploit such parallelism - unless, of course, one can show that transformation related overheads outweigh any benefit to be had through transforming the code.

Based on the original program in Figure 1, if one must perform the **deletemin** operation on a given number, say *nprog*, of priority queues, one option is to execute the code in Figure 1 sequentially, a total of *nprog* times. Since most of the statements in this original program are

scalar statements, executing the code does not make efficient use of vector functional units in machines like the CRAY and the Alliant. There is little, if any, parallelism to exploit here. An alternative scheme is to exploit the fact that *nprog* pieces of identical code can be executed in parallel, even though each executes on different data. If we can transform the original scalar program into a vector program with *nprog* components, it is conceivable that, though the *nprog* programs may exhibit some data-dependent path divergence, overall the vector pipes will be used more efficiently and speedup will result.

A scheme for effecting precisely such a transformation is shown in Figure 2, for the **deletemin** operation. Observe that the data declarations have now been expanded to accommodate the vector structures. The dimensionality of each data structure in Figure 1 is increased by one through the transformation (see Figure 2). This merely reflects the fact that each priority queue works with a distinct data set, and these are generally independent. More sophisticated schemes for inducing correlations between simulation runs for the purpose of variance reduction will utilize data sets that are not independent; for ease of explanation, we exclude such issues from consideration here.

A more detailed look at the transformed program will show certain new entities. When the *nprog* distinct components of the transformed program execute their **deletemin** operations in parallel, all components must participate in the step involving the deletion of the root. In this way, each simulation obtains its current event. The variable *current*, set to 1 in this example, is used to select program components that must remain active. The j-th component of the *indic* array is used to indicate whether this component is *active* or *inactive*. A component is designated active if it is required to take part in the current computation; otherwise it is inactive. Initially, all components are made active so that each may extract its current event from the heap. In the example, this is achieved by ensuring that $indic(j)$ has the same value as *current* for each component j. In Line 1 of Figure 2, all *nprog* components begin to perform the root extraction in parallel. The initial loop completes this operation, and decrements the size of each heap by one. The loop at Line 12 effects the percolation process in parallel for all components. The *indic* array effectively masks out those components that terminate the percolation process ahead of others; the change in a component's status from active to inactive occurs at Line 26, where a component's *indic* value is set to zero if its percolation process has terminated. Swapping of node contents' takes place in the loop at Line 29 for all active components. The vector **deletemin** operation terminates at Line 42, when all components have completed their percolations. Factors that may affect the parallel operation negatively include the different stopping times for the different components' **deletemin** operations, and the overhead incurred in masking out inactive components from computations.

It can be seen from Figure 2 that the transformation restructures the original code into three blocks. Each block is a loop that executes a part of the original code a total of *nprog* times, once for each distinct data-set (i.e., priority queue). The first statement in each loop

```
      C* Data declarations
1d.           integer maxp
2d.           parameter (maxp = 500)
3d.           real event(maxp), clock(maxp)
4d.           real iheap(1003,2,maxp), tempiheap(1,2,maxp)
5d.           integer index(maxp)
6d.           integer j(maxp), j2(maxp)
      C*  Active/inactive status indicator array
7d.           integer indic(maxp)
      C* Define currently active components
1i            current = 1
2i            do 10 m = 1, nprog
3i                indic(m) = 1
4i      10    continue
      C* Begin the vector deletemin operation
      C* for all priority queues
1.            do 1210 m = 1,nprog
      C* If a queue is currently active then
2.                if( indic(m) .ne. current) goto 1210
      C** extract the root item,
3.                event(m) =  iheap(1,1,m)
4.                clock(m) =  iheap(1,2,m)
      C** move the last heap item into the root position, and
5.                iheap(1,1,m) = iheap(index(m),1,m)
6.                iheap(1,2,m) = iheap(index(m),2,m)
      C** decrease the size of the heap by one
7.                index(m) = index(m) -1
      C** Begin the percolation phase
8.                j(m) = 1
9.                j2(m) = j(m) * 2
10.     1210  continue
11.     1219  continue
      C** for all active priority queues
12.           do 12220 m = 1, nprog
      C* If the queue is active, and
13.               if(indic(m) .ne. current) goto 1220
      C** if we have arrived at the lowest level,
14.               if(j(m) .gt. (index(m)/2)) then
      C* make the queue inactive for the rest of the phase
15.                   indic(m) = 0
16.               endif
17.     1220  continue
```

Figure 2: Code for unified dequeue operation

```
     C** Else, continue the percolation process
18.            do 1230 m= 1, nprog
19.                if(indic(m) .ne. current) goto 1230
     C* If a queue is active and
20.                if((j2(m) .ne. index(m)) .and.
21.                   (iheap(j2(m),2,m).gt.iheap(j2(m)+1,2,m)))
22.                then
23.                    j2(m) = j2(m) + 1
24.                endif
    C** the heap property is satisfied at this level, then the
    C** phase is complete
25.                if(iheap(j(m),2,m).le.iheap(j2(m),2,m)) then
    C** So make the queue inactive for the rest of the process
26.                    indic(m) = 0
27.                endif
28.      1230   continue
      C* Otherwise, further item swaps are required.
29.            do 1240 m= 1, nprog
      C* If a queue is active, then
30.                if(indic(m) .ne. current) goto 1240
      C* perform a parent-child item swap,
31.                tempiheap(1,1,m) = iheap(j(m),1,m)
32.                tempiheap(1,2,m) = iheap(j(m),2,m)
33.                iheap(j(m),1,m) = iheap(j2(m),1,m)
34.                iheap(j(m),2,m) = iheap(j2(m),2,m)
35.                iheap(j2(m),1,m) = tempiheap(1,1,m)
36.                iheap(j2(m),2,m) = tempiheap(1,2,m)
37.                j(m) = j2(m)
38.                j2(m) = j(m) * 2
39.      1240   continue
     C* Finally, check if any queues are still active
40.            do 1250 m =1, nprog
     C* If a queue is active, continue the phase
41.                if(indic(m) .eq. current) goto 12219
42.      1250   continue
     C* otherwise, terminate the deletemin operation
```

Figure 2: Code for unified dequeue operation (continued)

attempts to determine the status of each component. Only the active components of the transformed program (i.e., ones with *indic* values equal to *current*) will execute the code in the loop. The other components remain inactive for the duration of the loop. The entire operation terminates when each component has finished working on the percolation phase in its own priority queue. The loop described in Lines 40 through 42 sends control of the transformed program to the next step of the percolation, so that active components may make node comparisons at the next level, if any component is found still active; otherwise, the **deletemin** operation terminates. In a simulation application, control is then transferred to the event processing part of the simulation's logic.

In the transformed program, code is broken into blocks simply to reduce complexity. If a block (i.e., a do loop in this case) contains too many *if* statements which make for a complicated branching situation within the block, then the vectorization of the block (i.e., loop) will be inhibited. Therefore, combined with complexity reduction in code within blocks, this only improves vectorization. In contrast, increasing the number of blocks in transformed programs will result in increased overheads due to an increased number of loop setups. Therefore, the blocks should be few in number, large enough, and yet simple enough to vectorize efficiently.

In this study, unification is applied to a number of other priority queue structures, including simple ordered lists, skew heaps, splay trees, and skip lists. The general principles behind unification remain the same in each case, though the control flows may be different. Some queues yield simple flows while others are more intricate. We found that it helps to pay special attention to how the *indic* array is used in conjunction with the *current* variable, both for correctness as well as for efficiency.

3 Analysis

In this section, we determine factors which affect execution speedup through a simple analysis. We also attempt to give a rough justification for why the implicit heap has a better performance overall than the skew heap. Besides the theoretical results, some experimental data is also given to support our arguments.

3.1 Speedup computation

We begin by examining the deletemin operation of the implicit heap. In the previous section, unified program code was broken into blocks. For analysis and comparison of code in a program and its unified counterpart, the original program code must also be divided into blocks. This division is not arbitrary, but based on unified program code blocks. In Figure 3 can be seen control flow between blocks in the program code for the implicit heap. Control starts at Block 1 and then moves to Block 2. Control may exit at Block 2, or

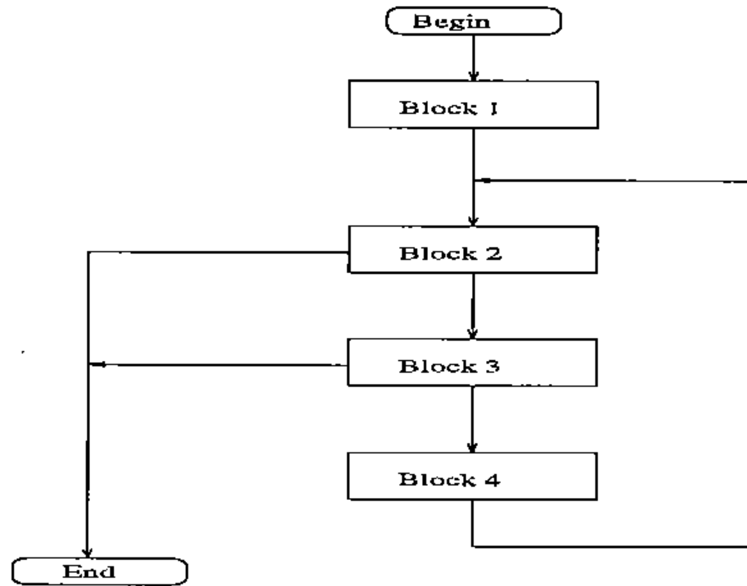

Figure 3: Control flow for the original code of the implicit heap

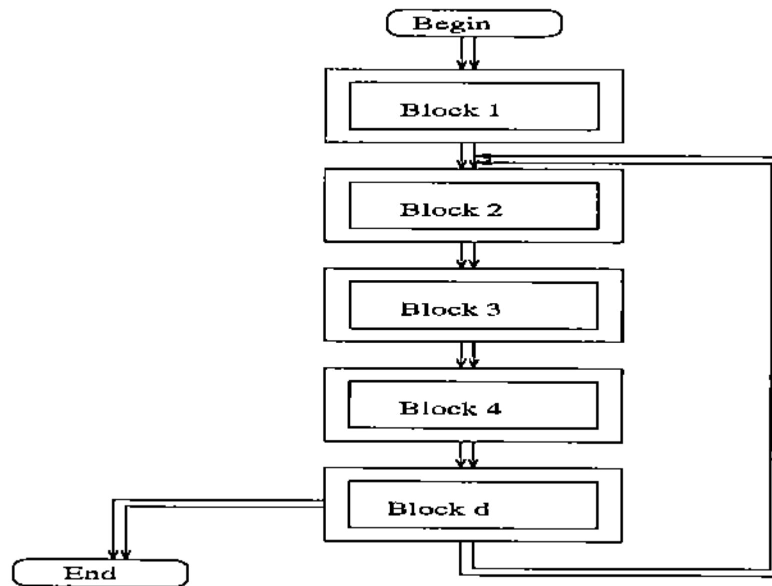

Figure 4: Control flow for the unified code of the implicit heap

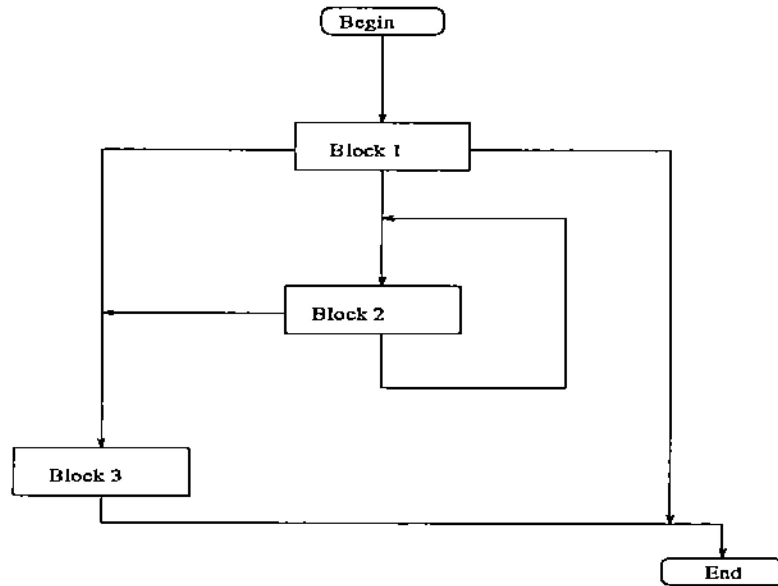

Figure 5: Control flow for the original code of the skew heap

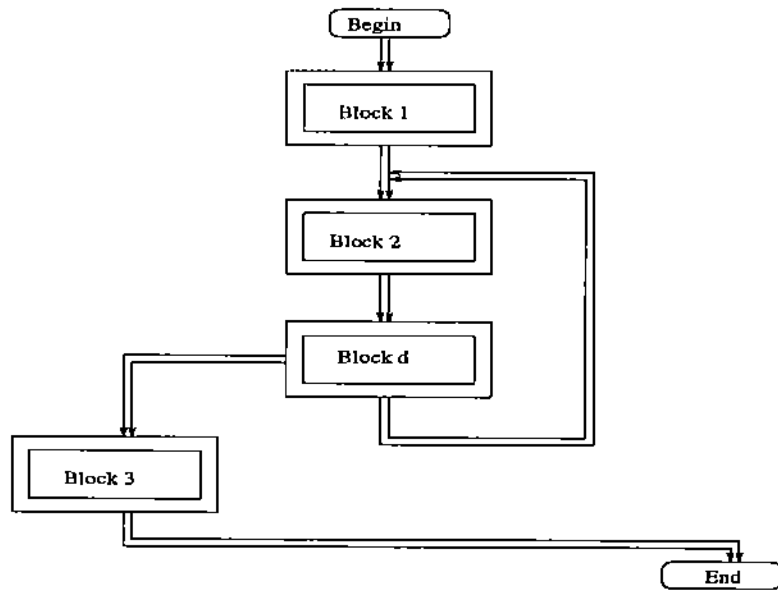

Figure 6: Control flow for the unified code of the skew heap

continue on into Block 3. At Block 3, if control does not satisfy criteria for stopping, it enters Block 4. After Block 4, control returns to Block 2 and the process is repeated. For a deletemin operation to complete, Block 1 needs to be executed once, and the loop around Block 2, 3, and 4 is repeatedly executed until the stopping condition is satisfied at either Block 2 or Block 3. Since there are two exit points for the deletemin operation execution may terminate at either block.

Lets assume that N instances of a simulation program are to be executed. If the program uses an implicit heap for its priority queue, then N independent priority queues are to be operated upon, each a certain (possibly different) number of times. In terms of Figure 3, each instance executes the loop around Block 2, 3, and 4 some instance-data dependent number of times. Suppose that instance i executes Block 2 precisely $E_i(k)$ times, where k is the size of the priority queue. If instance i exits at Block 2, it must execute Block 3 one time less than the number of times it executes Block 2, i.e., $(E_i(k)-1)$ times. On the other hand, if instance i exits at Block 3, it must execute Block 3 as many times as it executes Block 2, i.e., $E_i(k)$ times. It executes Block 4 exactly $(E_i(k)-1)$ times regardless of where it exits.

Let t_i indicate the average amount of time needed for any instance to execute Block i once. If it exits at Block 2, the time required by instance i to finish its deletemin operation on the heap is given by

$$T_i = t_1 + t_2 * E_i(k) + t_3 * (E_i(k) - 1) + t_4 * (E_i(k) - 1) \tag{3.1}$$

If instance i exits at Block 3, the time required for it to finish its deletemin operation on the heap is

$$T_i = t_1 + t_2 * E_i(k) + t_3 * E_i(k) + t_4 * (E_i(k) - 1) \tag{3.2}$$

If R instances exit at Block 2 and $(N-R)$ instances exit at Block 3, the total time required for N instances to complete their deletemin operations *sequentially* is given by

$$T_S(N) = \sum_{i=1}^{N}(t_1 + t_2 * E_i(k) + t_3 * E_i(k) + t_4 * (E_i(k) - 1)) - t_3 * R \tag{3.3}$$

An analysis of the unified code can be done in a similar manner. The control flow for the unified deletemin operation is shown in Figure 4. Observe that each of the N instances referred to above now becomes a *component* of an N-component unified program. During unified program execution, program components are either masked out of program block execution, or join a subset of components to execute a particular program block in unison When the unified code begins execution all N components execute Block 1 simultaneously. Next, all components simultaneously execute Blocks 2, 3, and 4 in sequence. Following this, they all arrive at a block, called Block d, which does not have a counterpart in the

nonunified code. This is a decision block that contains code which decides when the unified operation is to terminate. Although some of the components in the unified program will have completed their deletemin operation ahead of others, the unified program will not terminate until all program components terminate. Control flow either exits at Block d or returns to Block 2 to execute the loop again. Thus, the loop around Block 2, 3, 4, and d will be executed a total of $\mathcal{E}(N,k)$ times, where

$$\mathcal{E}(N,k) = MAX_{1\leq i\leq N}(E_i(k)) \tag{3.4}$$

In a vector machine, the time required to execute a unified program block is a function of N. Suppose that $T_i(N)$ is the average time needed to execute Block i with N program components. The total time required to finish the unified deletemin operation will be

$$T_U(N) = T_1(N) + (T_2(N) + T_3(N) + T_4(N) + T_d(N)) * \mathcal{E}(N,k) \tag{3.5}$$

Defining the *inverse block speedup* coefficient $\alpha_i(N)$ (see [11, 12]) as

$$\alpha_i(N) = \frac{T_i(N)}{N * t_i} \tag{3.6}$$

and correspondingly, the program *block speedup* coefficient, $\gamma_i(N)$, as

$$\gamma_i(N) = \frac{1}{\alpha_i(N)} = \frac{N * t_i}{T_i(N)} \tag{3.7}$$

we can use Equation (3.7) to estimate the average time required by the nonunified code to execute block i as

$$t_i = \frac{T_i(N) * \gamma_i(N)}{N} \tag{3.8}$$

By substituting this estimate into Equation (3.3) we obtain

$$\begin{aligned} T_S(N) = \quad & \textstyle\sum_{i=1}^{N}\Big(\frac{T_1(N) * \gamma_1(N)}{N} + \frac{T_2(N) * \gamma_2(N)}{N} * E_i(k) \\ & + \frac{T_3(N) * \gamma_3(N)}{N} * E_i(k) + \frac{T_4(N) * \gamma_4(N)}{N} * (E_i(k) - 1)\Big) \\ & - \frac{T_3(N) * \gamma_3(N)}{N} * R \\ = \quad & N * \frac{T_1(N) * \gamma_1(N)}{N} + \frac{T_2(N) * \gamma_2(N)}{N} * \sum_{i=1}^{N} E_i(k) \\ & + \frac{T_3(N) * \gamma_3(N)}{N} * (\sum_{i=1}^{N} E_i(k) - R) + \frac{T_4(N) * \gamma_4(N)}{N} * \sum_{i=1}^{N}(E_i(k) - 1) \end{aligned} \tag{3.9}$$

$$
\begin{aligned}
= \quad & T_1(N) * \gamma_1(N) + T_2(N) * \gamma_2(N) * \overline{E(k)} \\
& + T_3(N) * \gamma_3(N) * (\overline{E(k)} - \frac{R}{N}) + T_4(N) * \gamma_4(N) * (\overline{E(k)} - 1) \\
= \quad & T_1(N) * \gamma_1(N) - T_3(N) * \gamma_3(N) * \frac{R}{N} - T_4(N) * \gamma_4(N) \\
& + \overline{E(k)} * (T_2(N) * \gamma_2(N) + T_3(N) * \gamma_3(N) + T_4(N))
\end{aligned}
\tag{3.10}
$$

Using this, we compute speedup as

$$
S(N) = \frac{T_S(N)}{T_R(N)} = \frac{C_1(N) + \overline{E(k)} * C_2(N)}{C_3(N) + \mathcal{E}(N,k) * C_4(N)}
\tag{3.11}
$$

where

$$
\begin{aligned}
C_1(N) = \quad & T_1(N) \quad * \gamma_1(N) - T_3(N) * \gamma_3(N) * \frac{R}{N} - T_4(N) * \gamma_4(N) \\
C_2(N) = \quad & T_2(N) \quad * \gamma_2(N) + T_3(N) * \gamma_3(N) + T_4(N) * \gamma_4(N) \\
C_3(N) = \quad & T_1(N) \\
C_4(N) = \quad & T_2(N) + \quad T_3(N) + T_4(N) + T_d(N)
\end{aligned}
$$

As $\overline{E(k)}$ and $\mathcal{E}(N,k)$ take on large values, the contribution of $C_1(N)$ and $C_3(N)$ become small. We can thus approximate speedup by

$$
S(N) \approx \frac{\overline{E(k)} * (T_2(N) * \gamma_2(N) + T_3(N) * \gamma_3(N) + T_4(N) * \gamma_4)}{\mathcal{E}(N,k) * (T_2(N) + T_3(N) + T_4(N) + T_d(N))}
\tag{3.12}
$$

Finally, since Block d is a relatively small block with only a single statement, speedup can further be approximated as

$$
S(N) \approx \frac{\overline{E(k)} * \Gamma(N)}{\mathcal{E}(N,k)}
\tag{3.13}
$$

where $\Gamma(N)$ is a weighted average of $\gamma_2(N)$, $\gamma_3(N)$, and $\gamma_4(N)$. From Equation (3.13), its clear that $\overline{E(k)}$ and $\mathcal{E}(N,k)$ have a significant influence on speedup.

To determine how well the approximation compares with the real equation, we obtained several execution profiles for the heap deletemin operation. Table 2 exhibits speedup computed using Equation (3.11), and Table 3 exhibits speedup computed via the approximation in 3.13. Note that our approximation hinges on two assumptions, namely, (1) the value of T_d is relatively small, and (2) the values of C_1 and C_3 have little effect when priority queue sizes become large. It appears that the approximation works well.

In Table 4 can be seen values for Γ, and in Table 5 can be seen values for $\overline{E}/\mathcal{E}$, both relating to the heap's deletemin operation. Though the number of components N and

priority queue size k change, the ratio $\overline{E}/\mathcal{E}$ tend to remain constant. However, the Γ values increase with increasing N. This is because the vector sizes become larger with increasing N. It is precisely this fact which enhances vectorization in unified programs. Thus, Equation (3.13) and Tables 4 and 5 suggest that speedup values for the unified code will increase with increasing N.

The same technique can be used to analyze the deletemin operation for the skew heap. Using the control flow graphs for the deletemin operation on the skew heap (show in Figures 5 and 6), speedup can be obtained as

$$S'(N) = \frac{T'_S(N)}{T'_R(N)} = \frac{D_1(N) + \overline{E'(k)} * D_2(N)}{D_3(N) + \mathcal{E}'(N,k) * D_4(N)} \tag{3.14}$$

where,

$$\begin{aligned} D_1(N) &= T'_1(N) * \gamma'_1(N) + T'_3(N) * \gamma'_3(N) * \frac{Q}{N} \\ D_2(N) &= T'_2(N) * \gamma'_2(N) \\ D_3(N) &= T'_1(N) + T'_3(N) \\ D_4(N) &= T'_2(N) + T'_d(N) \end{aligned}$$

and $(N - Q)$ is the number of the components which exit at Block 1. Using arguments similar to our previous simplifying arguments, we obtain the speedup approximation

$$S'(N) \approx \frac{\overline{E'(k)} * (T'_2(N) * \gamma'_2(N))}{\mathcal{E}'(N,k) * (T'_2(N) + T'_d(N))} \approx \frac{\overline{E'(k)} * \Gamma'(N)}{\mathcal{E}'(N,k)} \tag{3.15}$$

3.2 Speedup comparison

From the speedup functions for the implicit and skew heaps, it is clear that the quantities $\Gamma(N)$ and $\overline{E(k)}/\mathcal{E}(N,k)$ play a key role in determining speedup. Comparing speedups for the two data structures reduces to a comparison of these quantities. We make the following observations.

1. The unified code for the skew heap has a smaller loop-building overhead since it has fewer loops than the unified code for the implicit heap. This suggests a larger value of Γ for the deletemin operation on a skew heap.

2. How well the unified code vectorizes is closely related to the value of Γ. Using the MFLOP rating on the CRAY Y-MP, define

$$\mathcal{M} = \frac{M_S}{M_U} \tag{3.16}$$

size/nprog	500	300	100
300	0.856927	0.857215	0.857400
200	0.892860	0.892850	0.892963
100	0.883697	0.884000	0.884371

Table 1: $\overline{E}/\mathcal{E}$

size/nprog	500	300	100
300	2.12099	1.97271	1.72339
200	2.20281	2.17720	2.05532
100	2.11892	2.20874	2.01045

Table 2: Speedup for heap deletemin (3.8)

size/nprog	500	300	100
300	2.16442	2.02061	1.79191
200	2.25607	2.23649	2.14522
100	2.18264	2.28232	2.10860

Table 3: Speedup for heap deletemin (3.10)

size/nprog	500	300	100
300	2.52579	2.35718	2.08994
200	2.52679	2.50489	2.40236
100	2.46990	2.58181	2.38429

Table 4: Γ values for heap deletemin

size/nprog	500	300	100
300	0.856927	0.857215	0.857400
200	0.892860	0.892850	0.892963
100	0.883697	0.884000	0.884371

Table 5: $\overline{E}/\mathcal{E}$ values for heap deletemin

size/nprog	500	300	100
300	0.525554	0.545163	0.573918
200	0.520985	0.534492	0.568603
100	0.499501	0.512477	0.546497

Table 6: $\overline{E}/\mathcal{E}$ values for skew heap deletemin

where M_R, M_U are the MFLOP values for the nonunified and unified programs, respectively. The data suggests that the skew heap has slightly higher value of $\mathcal{M}$ than the implicit heap. This also suggests a larger value of Γ for the skew heap.

3. Both data structures have O(log k) complexity in the average case. However, the skew heap has a worst case behaviour of O(k) [10] while the implicit heap has a worst case behaviour of O(log k). Although the worst case behaviour of the skew heap may not arise often, a single component exhibiting worst-case like behavior will make the unified program exhibit worst-case like behaviour. Therefore, the implicit heap will have larger value of $\overline{E(k)}/\mathcal{E}(N,k)$ than the skew heap.

Though the first two observations favor the skew heap for speedup, they have a relatively small influence compared to the last observation which favors the implicit heap. In particular, for N large, the last observation is critical. As N becomes large, the chances of having worst-case like behaviour tend to be large for the skew heap, because a single straggling component forces the rest to wait.

In Table 5 and Table 6 can be seen the $\overline{E(k)}/\mathcal{E}(N,k)$ ratios for the deletemin operation of the heap and of the skew heap, respectively. Clearly, the $\overline{E(k)}/\mathcal{E}(N,k)$ ratios for the skew heap tend to be smaller than the corresponding ratios for the implicit heap. Moreover, the values tend to decrease as N increases.

Although we only analyzed the deletemin operation, the structure and the analysis of the insert operation is similar, for both data structures. Therefore, we expect that the implicit heap will have a better speedup than the skew heap for both operations.

4 Description of Experiments

Performance studies of specific implementations of priority queues are, generally speaking, fairly difficult to conduct. Analytic methods tend to require considerable model simplification before any reasonable insights into performance can be gleaned [7]. On the other hand, measurements made by executing code require few assumptions, yield plenty of numbers, but are often difficult to interpret in general contexts. Usually, one is forced to study performance in a limited context. A thorough study, in this spirit, was conducted by Jones [5]. In this section we outline how our experiments are conducted using this approach.

In an early study of priority queues, Vaucher and Duval [2] proposed a model called the *hold model* for the purpose of comparing the performance of various queue implementations. Since then, this model has become a widely accepted standard benchmark in empirical comparisons [2]. The advantage of the hold model for priority queue studies is its simplicity and ease of use. The disadvantage of the model lies in potential differences between its behaviour and the behaviour of priority queues within simulations.

In a discrete event simulation, a stochastic process is taken through a particular realization of its space by moving the system from one event to another event in time, maintaining causality throughout the execution sequence. While the current event is being processed, new events (future events) may be generated and scheduled for some appropriate times in the future. Scheduling future events requires the insertion of items into the event list. Similarly, determining the current event and its time requires deleting the highest priority item from the event list. In general, for each item deleted, an average of one item must be inserted so that the simulation can continue indefinitely. An average of less than one insertion would lead to an empty event queue, and an average of greater than one insertion would lead to a queue that grows without bound. Hence, for each deletion, a random number of insertions (averaging one) may ensue.

In discrete event simulations, when an item is inserted into the event list, the time until its occurrence is the single factor determining its priority in the list. In general, the distribution of this time is a complicated, model-dependent quantity. Further, since each item is associated with an event type through an integer code, each event type may be associated with a time-to-occurrence that is different in distribution from other event types. The hold model for priority queues assumes that all events are of the same type, the time-to-occurrence random variable for each event has the same distribution, and inter-event times are independent. As explained above, in a general discrete event simulation, the **insert** and **deletemin** operations may occur in complicated sequences, where each type of operation may depend on a subsequence of preceding types. Thus, while it is easy to show that the hold model assumptions are not true in general models, it is still a useful device in that it enables us to obtain measurements in a simple but meaningful framework incorporating some reasonable amount of randomness; random enough to be useful, but yet not random enough to make the model results too complicated to interpret.

In Figure 7 is shown the pseudo-code that we use for implementing the hold model. The basic operations used in the hold model are **deletemin**, **generate** a new event, and **insert**. A set of operations in precisely this sequence is defined as a *hold operation.* The model ensures that the priority queue eventually settles at some reasonable size, even if it is allowed to increase in size in a random manner initially. Being able to control the size of the priority queue allows us to control a key variable in the experiment. Also, the time-to-occurrence value used in scheduling an event, or equivalently inserting an item into the priority queue, is a random variate independently sampled from a given distribution which remains fixed for the duration of an experiment.

We selected five different algorithms for maintaining priority queues. These include an ordered list, a heap, the top-down version of a skew heap, a splay tree and a skip list. The ordered-list [1] is a simple and well-known structure which maintains items in order in a list. The heap [1], described in the previous section, is an important and also well-recognized structure. Less well-known are the skew heap [13] and the splay tree [13]. We also decided

```
C* Initialize the queues
     q = nil
     do i = 1, size
         r = obtain_a_random_variate_from_a_given_distribution(d)
         call insert(q,r)
     endo
C* Perform the hold a total of "trial' times by
     tri = 0
10   tri = tri + 1
C* doing a deletemin operation, followed by
     call deletemin(q,notice)
     event = notice_type(notice)
     clock = notice_time(notice)
C* an insert operation
     r = obtain_a_random_variate_from_a_given_distribution(d)
     clock = clock + r
     insert(q,clock)
     if( tri .lt. trial) goto 10
```

Figure 7: Code for the hold model

to include a newer structure called the skip list [9] because it has not been studied in the simulation context before, and we had no idea how it would perform in our application. Skew heaps and splay trees were chosen because they belong to a class of "nearly optimal" implementations for scalar machines, as identified by [5, 6].

The two key parameters that affect the performance of the hold model are, predictably, the size of the priority queue, and to a less well-understood extent, the probability distribution used to generate the times-to-occurrence for events. We decided to study the effects of different distributions, and in particular, to utilize distributions with very different standard-deviation to mean ratios CV [3] (i.e., coefficient of variation, where $CV \equiv \sigma / \mu$). In particular, we chose the popular Exponential distribution ($CV = 1$), the distribution obtained as a mixture of two exponentials, or Hyperexponential ($CV > 1$), and the distribution obtained as a sum of four exponentials, or four-stage Erlang ($CV < 1$).

5 Empirical Results

Given that all events are scheduled using the same time-to-occurrence random variable (i.e., the distribution is fixed), the key parameters in the experiments include the size of each priority queue, and the number of program components making up the unified program. In the following experiments we keep one parameter fixed while the other is varied, reporting

both execution timings as well as speedups. For an experiment in which the number of program components is the free variable, all priority queues are forced to have the same size; this is to eliminate additional parameters in the model. Observe that the number of program components is equal to the number of priority queues used, since each component works on a distinct priority queue. This parameter can be expected to have significant impact on timings and hence speedup, since it will affect the number of times each loop in each block of the transformed program is executed. This number is essentially the number we attempt to exploit through vectorization.

Each of the following experiments falls into one of two categories. In one category, the number of program components is kept fixed at 500 while the size of each priority queue data structure is varied from 1 to 1000. The component count was set at 500 and the queue size limit was set at 1000 in order to reduce machine related overhead time used. With the same reasoning, in the other category, the size of each priority queue was kept fixed at 100 while the number of program components is varied from 1 to 500. In each case, a total of 100 hold operations (one insert and one deletemin for every hold operation) is executed. This number was chosen after we ran a sufficient number of pilot experiments to determine that the variance in execution times with this many hold operations is negligible.

All programs were written in basic Fortran. The tree structures and pointers were simulated in Fortran using arrays. The original or nonunified programs for priority queue manipulation were written without any special attempt to exploit vectorization; indeed this is difficult to do for most structures, with the exception of the ordered list which lends itself to vectorization in a natural manner. To exploit the most out of vectorization, we refrained from using subroutine calls or function calls. All experiments were run using one processor on the CRAY-Y/MP at the Supercomputing Center at the University of Illinois. Programs were compiled first by the preprocessor so as to add vectorization directives, and then fed to the compiler. Each of the five implementations contained different amounts of code that vectorized, both for nonunified code and for unified code. We used subroutine *etime* to obtain timings. Execution time for the initialization routine was eliminated by deleting its contribution to the overall running time.

The first six graphs (Figures 8 though 13) exhibit timings measured via the hold operation on each of the five priority queue structures for different distributions *without* using unification. That is, the execution times of the original nonunified programs are measured. It is instructive to note that this experiment is different from the one conducted by Jones [5] simply because the underlying machine now has the potential to vectorize code. The experiments conducted by Jones [5] were based on scalar processor timings. Consequently, one might expect the results to differ; indeed, our experiments confirm that they do. In Figure 8 is shown a graph of execution time versus priority queue size, for a number of program components fixed at 500. In Figure 9 is shown a graph of execution time versus number of program components, for a priority queue size fixed at 100. In both cases, the

time-to-occurrence random variable used was an Exponential random variable. This pair of experiments is repeated using the four-stage Erlang, and the two-component Hyperexponential distributions, with the results displayed in Figures 10 and 11, and Figures 12 and 13, respectively.

Among the five data structures used, the ordered list is the only structure in which the **insert** and **deletemin** operations have an average and worst-case complexity of $O(n)$, where n is the size of the queue. The complexity of these operations in each of the other structures is $O(\log n)$. It is of interest to observe that while the ordered list can be expected to perform terribly on a scalar machine when n is large, (for example, even when $n = 50$, Jones [5] concludes that the ordered list is uniformly the worst among all data structures compared), its performance improves drastically on a vector machine. In addition, its inherent simplicity allows it to utilize vectorization to a level that cannot be achieved by the other structures which are considerably more intricate both in structure and operation. As a result, the ordered list outperforms the four other priority queues for all distributions used, for reasonable queue sizes (i.e., less than 200). For queue sizes between 200 and 300, the ordered list still outperforms the splay tree and the skip list. Unfortunately, for larger queue sizes, the $O(n)$ algorithmic complexity begins to dominate, making the $O(\log n)$ schemes more competitive and hence more attractive.

In like fashion, it is of interest to observe that while the intricacies of the other data structures do not aid in vectorization, they are not an obstruction either. In each case, these data structures retain their $O(\log n)$ complexity (seen in the graph as approximately straight lines, because of the logarithmic scale on the horizontal axes representing various queue sizes). From the graphs it can be seen that the heap and the top-down version of the skew heap exhibit uniformly better performance than the splay tree and the skip list. The skip list exhibits the worst performance for the *Exponential* and $Erlang_4$ distributions while its performance is very similar to that of the splay tree for the $Hyperexponential_2$ distribution.

The next six graphs (Figures 14 through 19) exhibit speedups obtained via the hold model on each of the five priority queue structures, for different distributions. Since we are interested in obtaining speedup over a number of independent simulation runs, say n, speedup with n program components is computed as the ratio of the time taken by an n-component unified program to operate on n priority queues in parallel, to the total time required for all n programs to operate on their priority queues sequentially. That is, execution times of the unified programs are now measured. In Figure 14 is shown a graph of unified-program speedup versus priority queue size, for a number of program components fixed at 500. In Figure 15 is shown a graph of unified-program speedup versus number of program components, for a priority queue size fixed at 100. In both cases, the time-to-occurrence random variable used was an Exponential random variable. This pair of experiments is repeated using the four-stage Erlang, and the two-component Hyperex-

ponential distributions, with the results displayed in Figures 16 and 17, and Figures 18 and 19, respectively.

Because the nonunified program for the ordered list structure vectorizes well, there is little for it to gain from unification. On the other hand, unification contributes towards a considerable amount of overhead. Hence, the speedup of the ordered list remains constant and even becomes less than 1, as the size of the priority queue is varied.

With the exception of the ordered list, for all other priority queue implementations speedup increases as the number of program components increases; and this is only to be expected. The heap and the skew heap begin by performing poorly when the number of program components is small, and then both of these structures outperform the others after a point. This is indicative of the fact that unification aids in their vectorization, adding little damaging overhead in the process. The splay tree structure performed uniformly fairly in all situations, suggesting that unification related overhead was greater in this case. With the exception of the experiment in which the Hyperexponential distribution was used, the skip list structure started out well but completed only fairly. Nevertheless, in peak speedup, it outperformed the splay tree; it is clear that unification is not an aid to skip list vectorization.

When the number of components was kept fixed while the size of each priority queue structure was varied, the speedup obtained for each unified structure exhibited a uniform pattern. Except for the ordered list which gave uniformly poor speedup throughout, the others started out very well, with high speedups, and fell almost exponentially with increasing queue size. While this is in keeping with the fact that increased vector sizes yield diminishing returns, it is largely due to the variation in phase termination for each structure. That is, though some components finish operating on their priority queues, they nevertheless have to wait until all components terminate, and this causes an increase in execution time and a decrease in speedup. In all situations, the heap structure uniformly exhibited the best speedup, and the splay tree the worst speedup.

Usually, when the vector length (i.e., N) increases, better speedup can be expected. Since every vector computer has a vector size for peak performance, unified program execution will be best for N equal to this vector size, or a multiple of this vector size. For the CRAY Y/MP (which has a vector size of 64), speedup with $N = 60$ can be expected to be better than speedup with $N = 70, 80$ or 90. This is witnessed in the performance degradation shown in Figures 15, 17, and 19. This degradation is also present when N exceeds a small multiple of 64, such as at N greater than 128, and N greater than 192. Since our data points after 100 are at 200, 300, 400, and 500, the degradation after $N = 100$ is not visible in the graphs.

Figures 20 and 21, exhibit the MFLOP rates obtained by each of the five data structures on the CRAY Y/MP. In both cases, the *Exponential* distribution was used. In Figure 20 can be seen the MFLOP rates for the nonunified programs. As expected, the ordered list

vectorizes well and shows the best use of the machine. Its peak MFLOP rating is 19.75, while the other four don't exceed 9.00.

The MFLOP rates for the unified programs are shown in the Figure 21. The skew heap has best machine utilization in this case. The splay tree exhibits second best utilization. Though the implicit heap has the best overall speedup, its machine utilization is not as favourable. All three data structures show tremendous improvement in MFLOP rates as a consequence of unification. Clearly, as N becomes larger, the MFLOP rating will further increase. The ordered list and skip list do not show improved MFLOP ratings because their respective nonunified codes vectorized sufficiently well to make unification not beneficial.

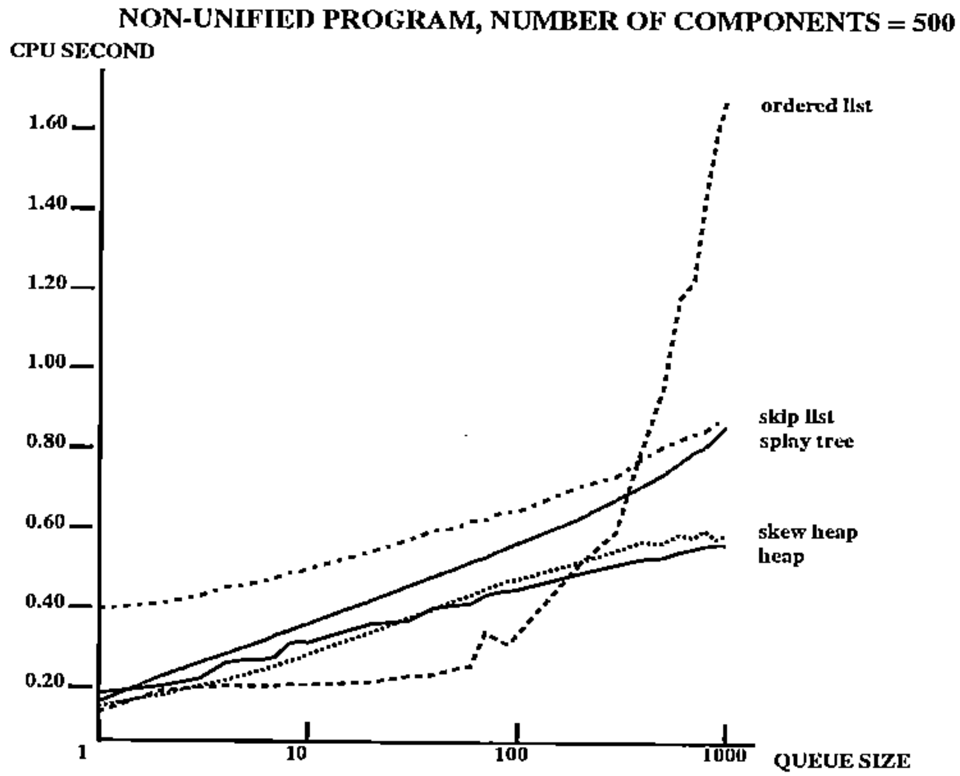

Figure 8: *Exponential* Distribution

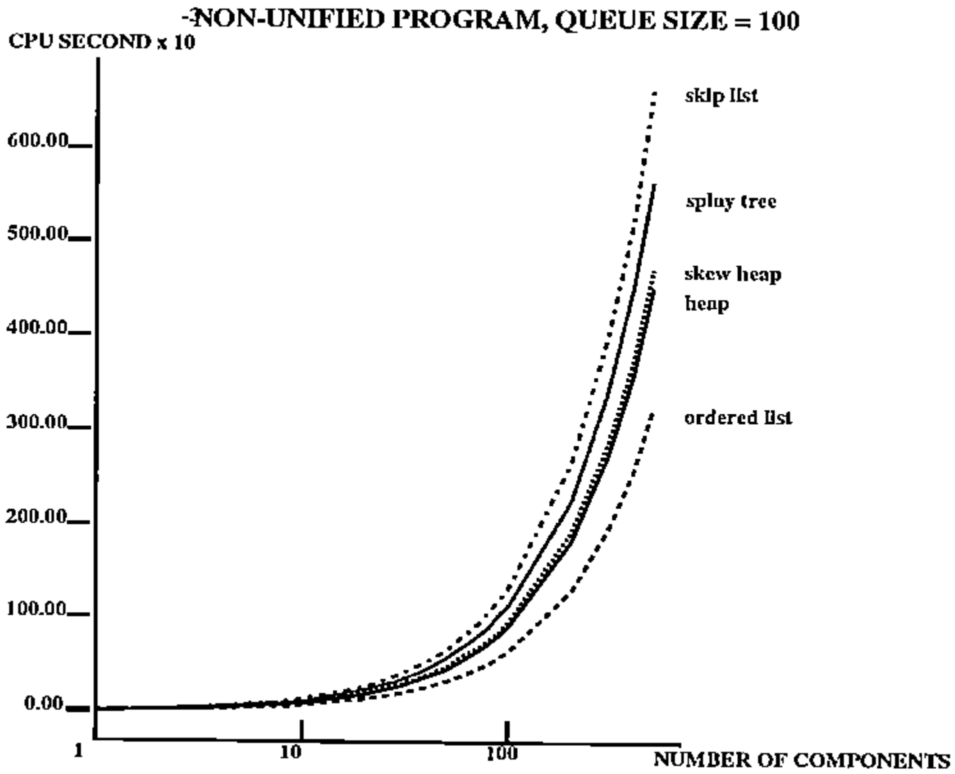

Figure 9: *Exponential* Distribution

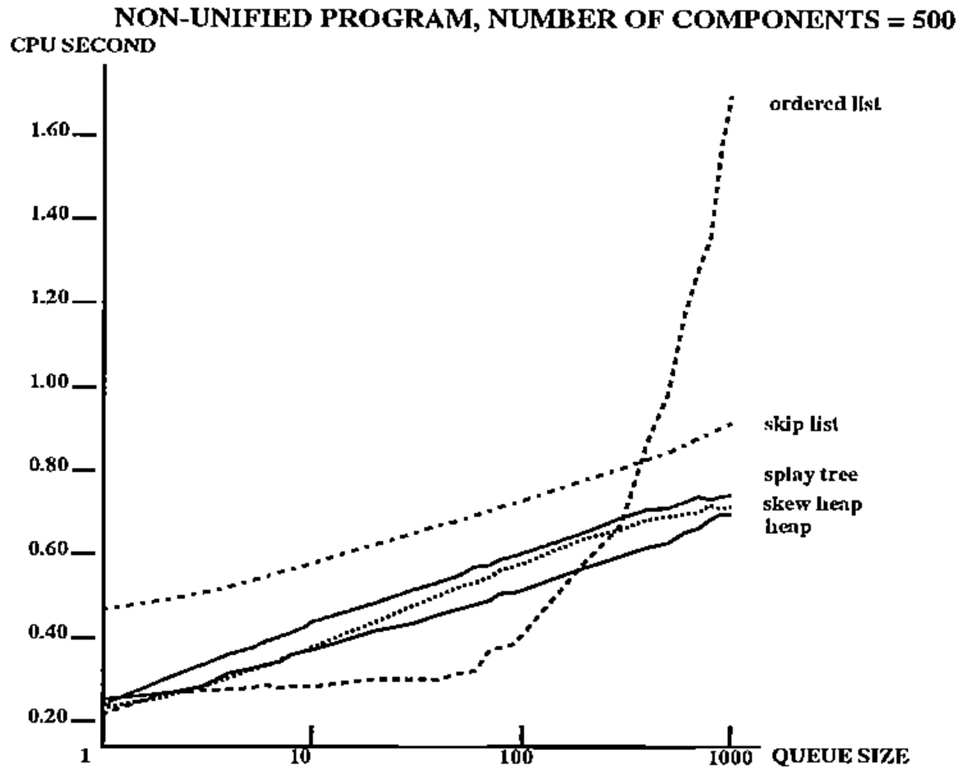

Figure 10: $Erlang_4$ Distribution

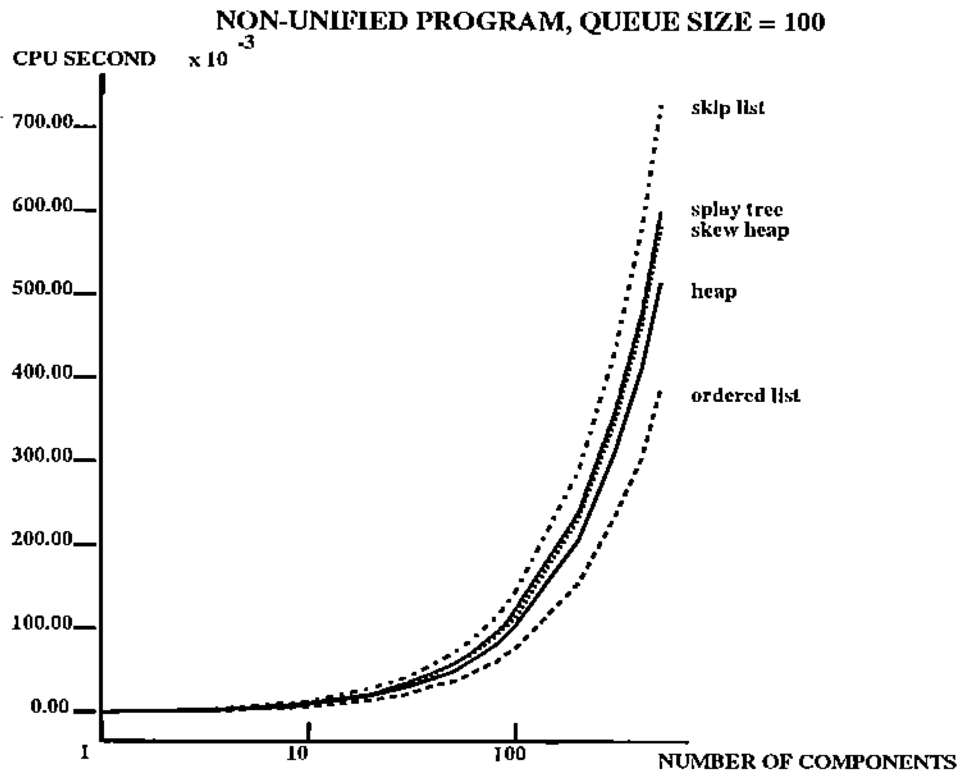

Figure 11: $Erlang_4$ Distribution

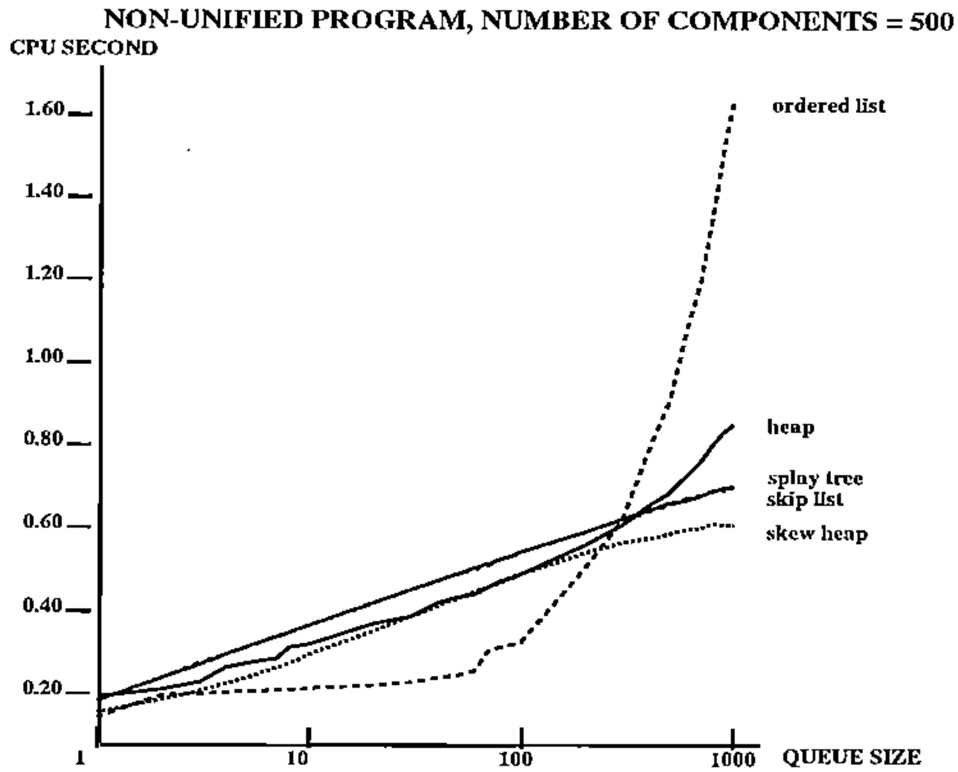

Figure 12: $Hyperexponential_2$ Distribution

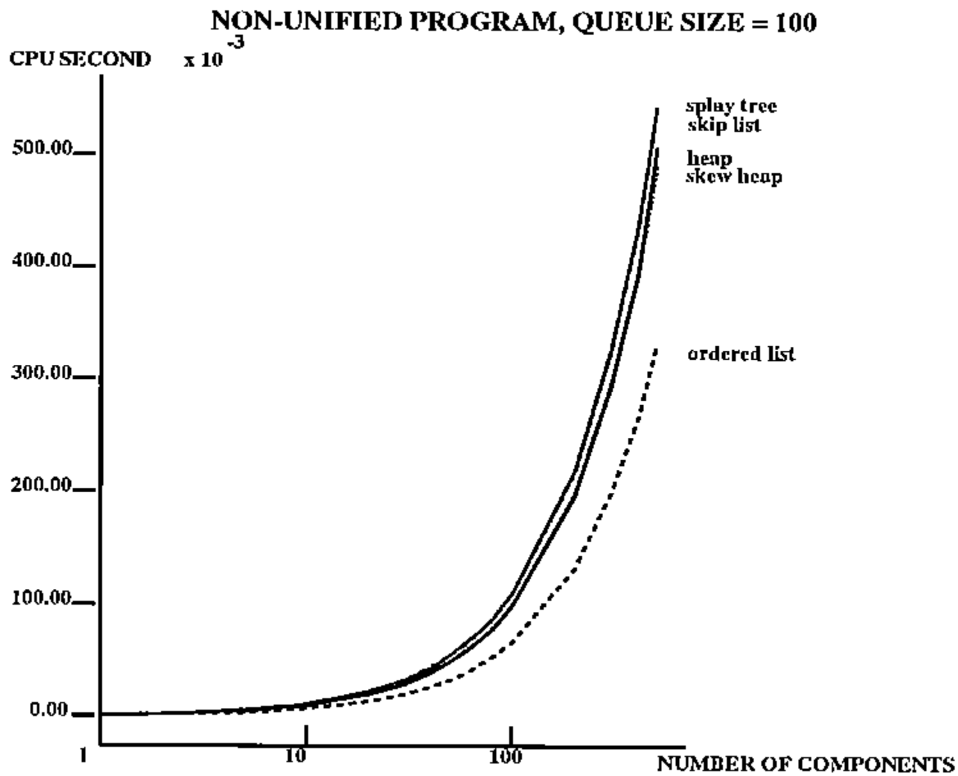

Figure 13: $Hyperexponential_2$ Distribution

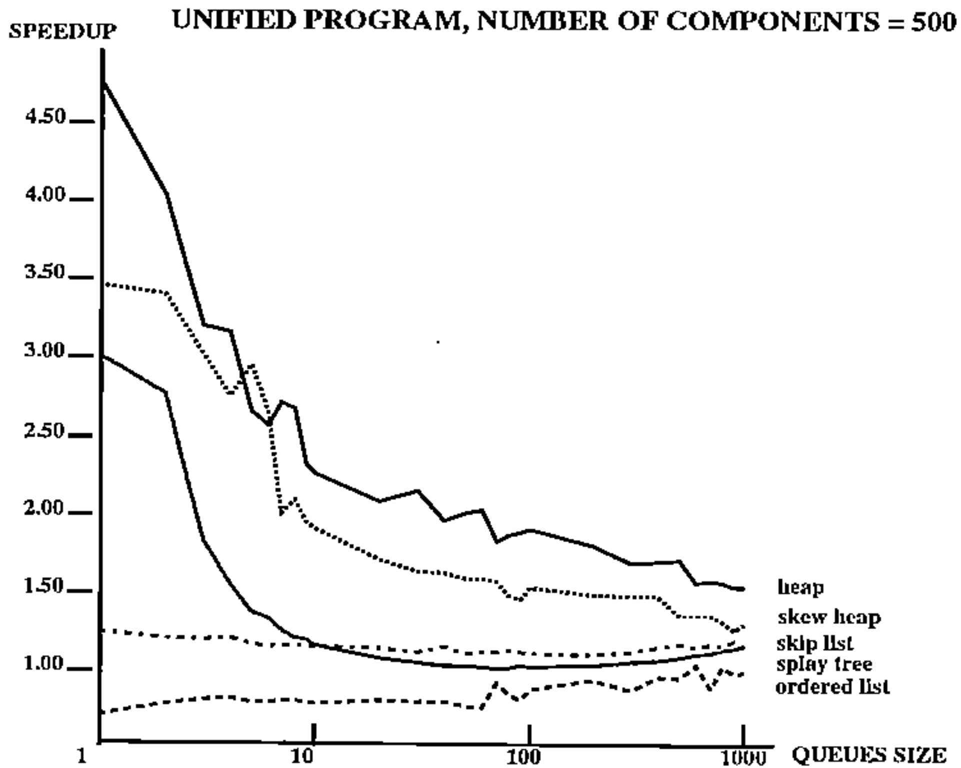

Figure 14: *Exponential* Distribution

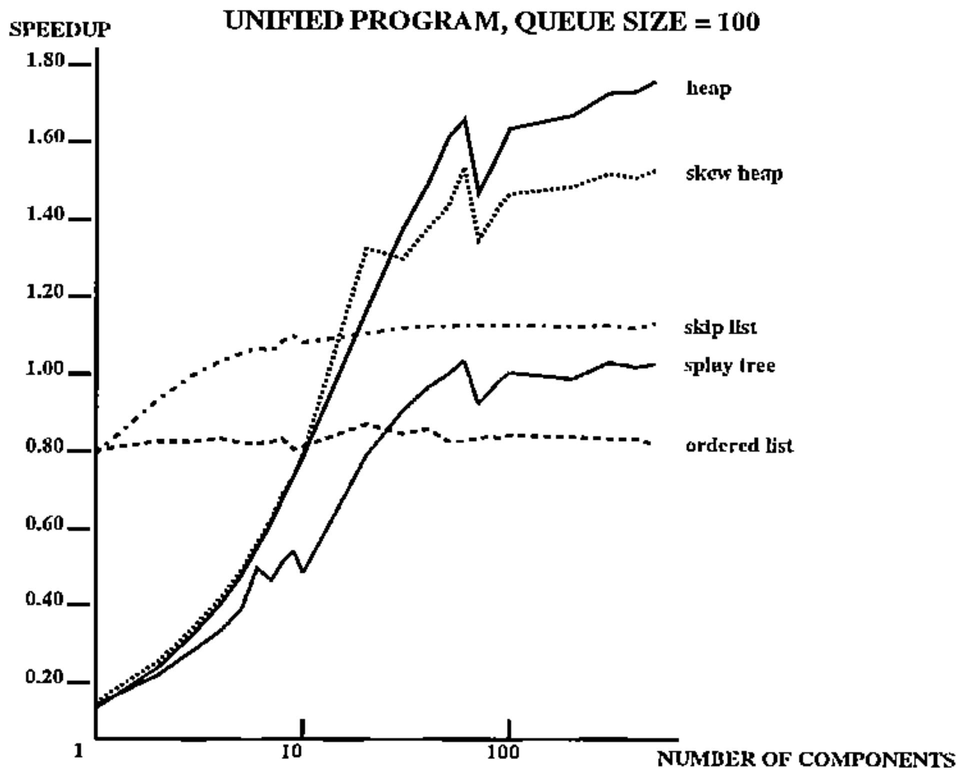

Figure 15: *Exponential* Distribution

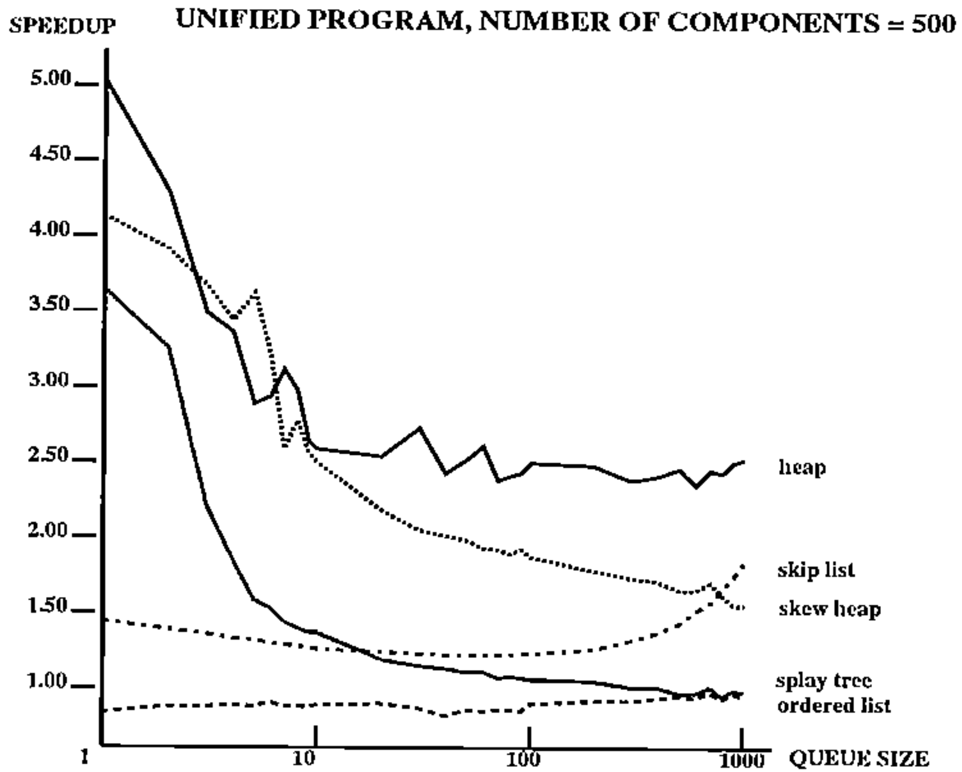

Figure 16: $Erlang_4$ Distribution

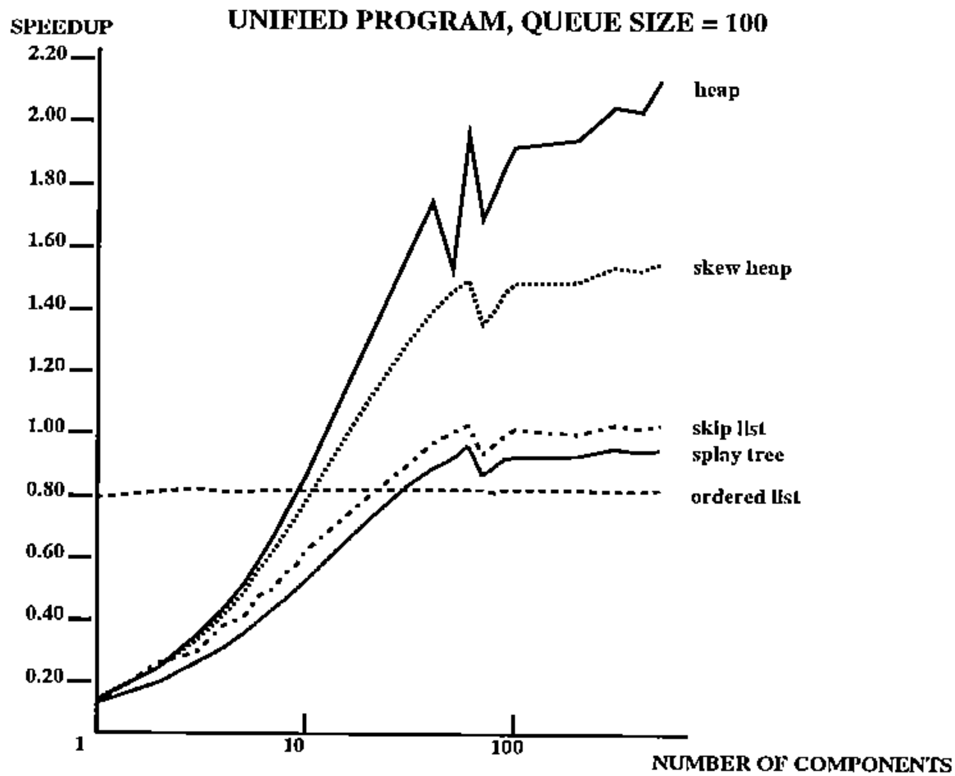

Figure 17: $Erlang_4$ Distribution

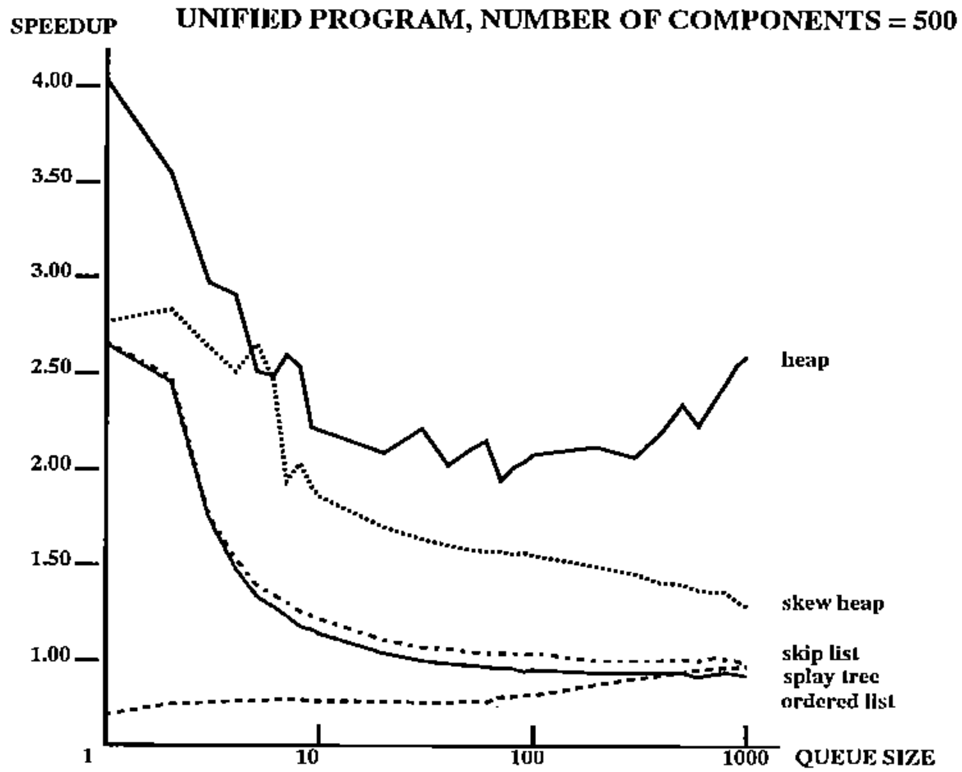

Figure 18: $Hyperexponential_2$ Distribution

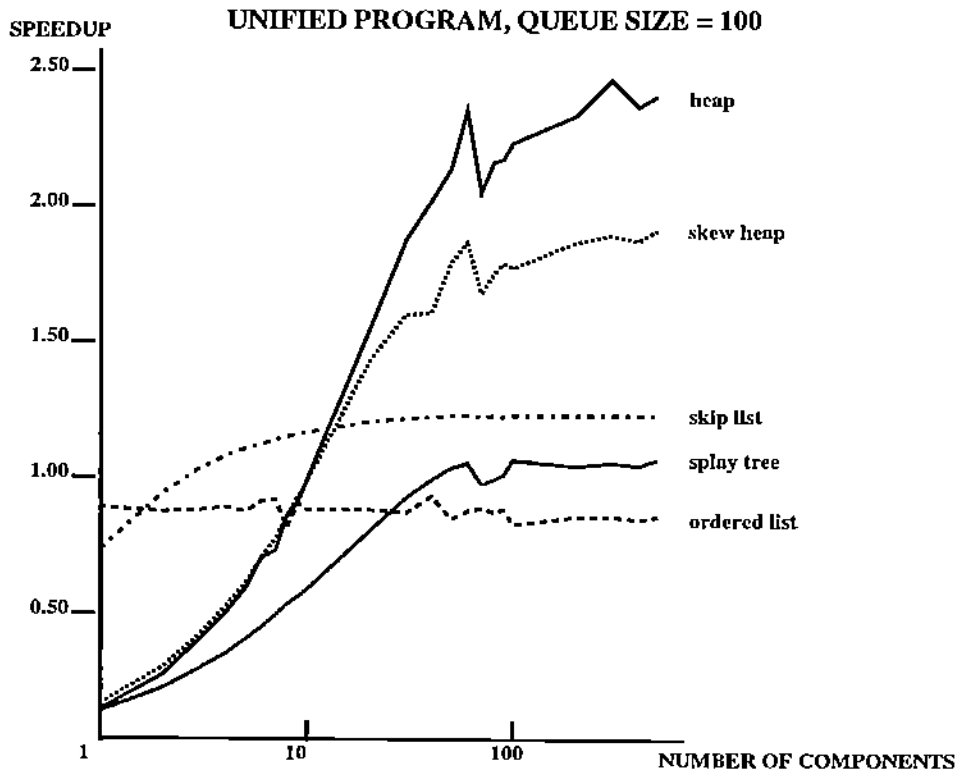

Figure 19: $Hyperexponential_2$ Distribution

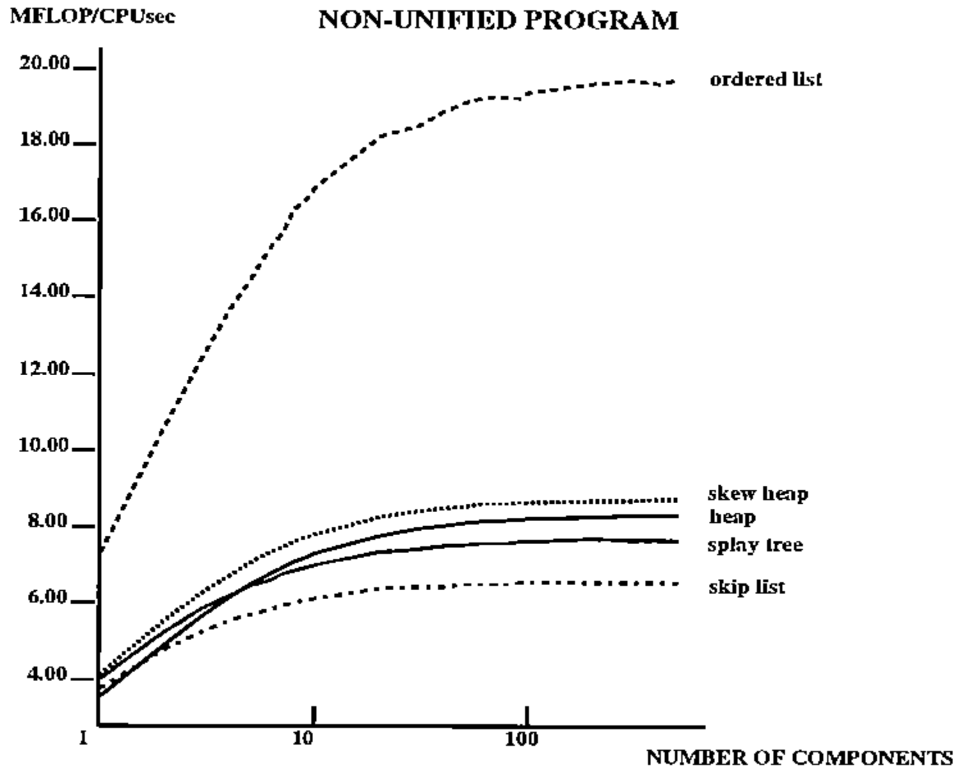

Figure 20: *Exponential* Distribution

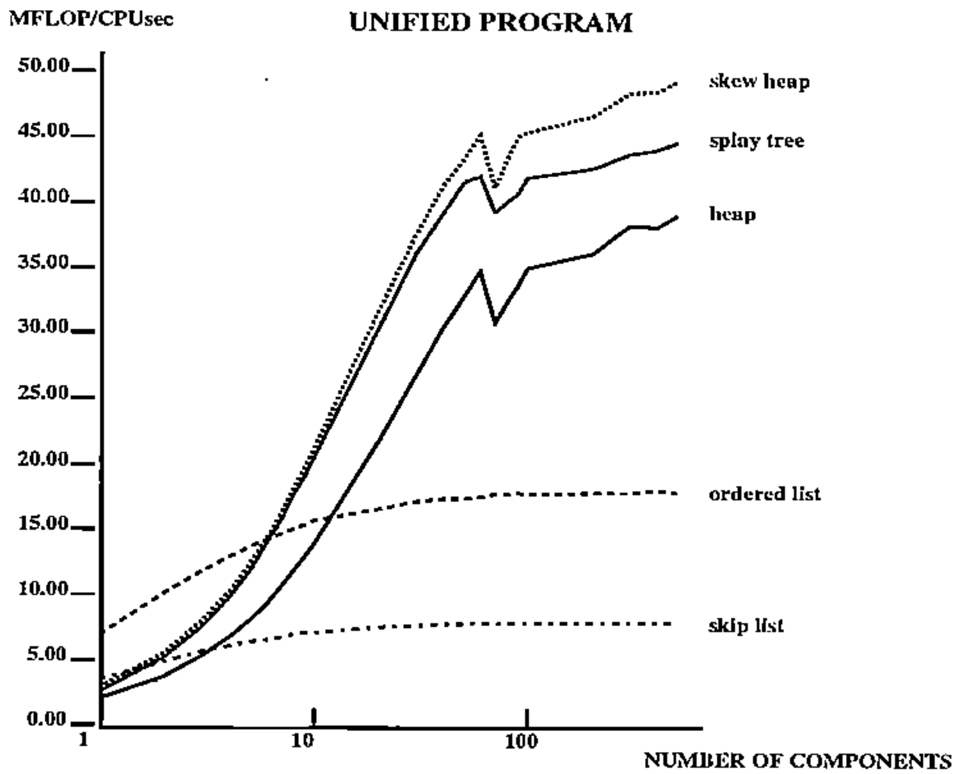

Figure 21: *Exponential* Distribution

6 Discussion and Future Work

We undertook this project mainly to determine if any one particular priority queue implementation would perform uniformly better than the others through program unification on a vector machine. While we did not expect the unified version of the ordered list to perform well, simply because of its algorithmic complexity, we were quite surprised when the non-unified version performed well. While this is understandable, in retrospect, it also makes a good case for experimental research where one sometimes stumbles upon something that one would not ordinarily think of otherwise. The moral of this story is that (non-unified) discrete event simulations utilizing ordered lists for priority queue implementations of event lists can be expected to perform very well on vector machines for reasonable event list sizes. On the other hand, one also comes to the surprising conclusion that for unified discrete event simulations, the implicit heap performs very well on vector machines. Combined with its simple structure and ease of programming (compared to the more complicated priority queue structures), the implicit heap is certainly the structure of choice.

Since this is only part of a larger study, we temper the above conclusions by saying there are several other things that should be considered as well. These range from quantities as difficult to model as **insert** and **deletemin** sequences and time-to-occurrence random variables, to application specific details involving the number of event types, and contribution of priority queue processing time to the overall simulation time (which, in rare situations, may be mainly event processing time). Clearly, the role of unified priority queues is inherently application dependent, and consequently speedup performance will also be application dependent.

In our future work, we plan to show how such unified programs can be created in fairly effortless ways, and often with good speedup characteristics. Some initial work in this respect has already been accomplished [11]. We expect this work in priority queue unification to be an aid in taking this work to a more mature level, where starting with the priority queue, one unifies the entire simulation application. Our current work involves extensions of these ideas to the CM and MasPar architectures.

References

[1] A. V. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass, 1974.

[2] J. B. Evans. *Structures of Discrete Event Simulation.* Ellis Horwood Limited, Market Cross House, Cooper St. Chichester, West Sussex, PO19 1EB, England, 1988.

[3] D. Gross and C. M. Harris. *Fundamentals of Queuing Theory.* John Wiley and Sons, 605 Third Avenue, New York, NY. 10158, 1985.

[4] J. O. Henriksen. An improved event list algorithm. In *Winter Simulation Conference*, Piscataway, N.J., 1983. IEEE.

[5] D. W. Jones. An empirical comparison of priority-queue and event-set implementation. *Communications of the ACM*, 29(4), April 1986.

[6] D. W. Jones. Concurrent operations on priority queues. *Communications os the ACM*, 32(1), January 1989.

[7] J. H. Kingston. Analysis of tree algorithms for the simulation event list. *Acta Inf.*, 22(1), April 1985.

[8] S. Olariu and Z. Wen. Optimal parallel initialization algorithms for a class of priority queues. *IEEE Transactions on Parallel and Distributed Systems*, 2(4), October 1991.

[9] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6), June 1990.

[10] C. M. Reeves. Complexity analyses of event set algorithms. *The Computer Journal*, 27(1), 1984.

[11] V. Rego, L. Chuang, and A. Mathur. Concurrent stochastic simulations: Experiments with unification. In *Fifth Annual Canadian Supercomputing Symposium*, Fredericton, N. B., Canada, 1991.

[12] V. Rego and A. P. Mathur. Exploiting parallelism across program execution: a unification technique and its analysis. *IEEE Transactions on Parallel and Distributed Systems*, October 1990.

[13] D. D. Sleator and R. E. Tarjan. Self-adjusting binary trees. In *In proceedings of the ACM SIGACT Symposium on Theory of Computing*, New York, 1983. ACM.