TR-85-7

SORTING IN LINEAR EXPECTED TIME

M.T. NOGA and D.C.S. ALLISON

Lockheed Palo Alto Research Laboratory,
Palo Alto, CA 94304

and

Department of Computer Science
Virginia Polytechnic Institute and State University,
Blacksburg, VA 24061

ABSTRACT: A new sorting algorithm, Double Distributive Partitioning, is introduced and compared against Sedgewick's quicksort. It is shown that the Double Distributive Partitioning algorithm runs, for all practical purposes, in O(n) time for many distributions of keys. Furthermore, the combined number of comparisons, additions, and assignments required to sort by the new method on these distributions is always less than quicksort.

KEYWORDS: Statistical sorting, hybrid sorting, distributive partitioning, quicksort.

1. INTRODUCTION

In this paper we present an internal sorting method which apparently runs faster than quicksort [1] for many reasonable distributions of the items (keys). The outstanding feature of the new method is that for nearly every input vector approximately the same number of operations are required to sort. The method combines the idea of sampling with that of distributive partitioning. For reasons which will become clear we name the new method double distributive partitioning (DDP). To measure the speed of our algorithm against that of quicksort we have used the accepted standard of counting fundamental operations (those which take only one or a few cycle times to complete). The critical features of the algorithm are illustrated by segments of PASCAL code. Variable names were chosen so that a rather straightforward transcription into another high-level language should be possible.

2. HISTORICAL PERSPECTIVE

It has been known for some time that, besides being fast, quicksort performs remarkably well for almost any distribution of items. Consider Table 1 where the number of fundamental operations required by the 'median-of-three' hybrid quicksort of R. Sedgewick [2] are compared on several common distributions. (Sedgewick's implementation is known to be one of the fastest of the hybrid quicksort algorithms.) It can be seen that the number of operations involved is nearly identical. Analysis shows that quicksort has an O(nlogn)

expected-case behavior, where n is the number of items to be sorted. One slight drawback to using quicksort is that the worst-case behavior is $O(n^2)$. However, it can be argued that if care is exercised in choosing the partitioning element, then as n increases, the probability of the worst-case occurring is extremely small.

There is another sorting method called DPS [3,4,6,9,10,13] which has been shown to sort substantially faster that quicksort on uniform distributions; see Table 2 for some comparative performance statistics. In this method a linear formula is used to distribute the items into bins (buckets or boxes). The contents of any two bins i and j, $i < j$, are such that all items in bin i are less than those in bin j. After this initial distribution, the method may be recursively applied to each bin or comparison based methods can be used to finish the sort. A theoretical treatment of DPS by Devroye and Klincsek [4] asserts that performance will be O(n) for a wide range of distributions, i.e., those which have either exponentially dominating tails or compact support. Among these are the chi-squared, beta, normal, exponential, and rectangular distributions. Unfortunately, there is a rather wide range in performance depending upon which of these distributions is encountered during the sorting process [5]. The reason is that certain bins will contain a higher percentage of items than others. This means that if the distributive method is recursive, the probability is high that an item will have to be redistributed. On the other hand, if the DPS method is a

hybrid (like several of those listed in Table 2), additional time will be spent on the second pass (or third pass) sorting method (for example, bubblesort, insertion sort, quicksort, heapsort). Thus, it is questionable whether any of the existing DPS hybrids will run faster than quicksort for most non-uniform distributions except when n is quite large (n > 10000). Our research emphasis will be to find a distributive scheme which places approximately the same number of items into each bin regardless of the distibution of the items involved.

Although there are some other methods which work well in special cases, for example when the items are already nearly in order (insertion sort), or if the keys are of a fixed length (radix sort), we will be interested in more general methods where the keys are not necessarily of fixed size and there is no a priori knowledge concerning the original ordering of the keys.

Table 1. Comparing the performance of Sedgewick's quicksort on several different distributions. Entries are the number of operations (assignments + arithmetics + comparisons) for a PASCAL implementation. All operation counts were derived from an average of 25 runs.

n	Uniform	Normal	Gamma	Exponential
250	8028	8102	8071	8035
500	17698	17725	17910	17657
1000	38800	39213	38667	39008
2000	84524	84257	84640	84320
4000	182709	182503	181568	181914

Table 2. Previous tests of distributive sorting methods versus quicksort; n = 5000 items.

Test Authors	DPS method and time	Quicksort method and time	Computer and Language
Meijer and Akl [6]	Distributive Pass, Heapsort Pass. 1.970sec	Non-recursive version due to S. Baase [7]. 6.301sec	Burroughs B6700, PASCAL
Dobosiewicz [3]	Find median, split vector, distributive pass on each subvector. Method is recursive. 1.264ms	Non-recursive version due to R. Loeser [8]. 2.072ms	CDC 6400, FORTRAN (FTN 4.1, OPT = 1)
van der Nat [9]	Partition vector into two groups of size n/2, distributive pass on each subvector, merge sort two groups. Method is recursive. 3.448ms	Median-of-three due to R. Sedgewick [2]. 5.262ms	IBM 370/158, ALGOL 60
Kowalik and Yoo [10]	Distributive Pass. Method is recursive. 122ms	Non-recursive version due to R. Singleton [11]. 168ms	Amdahl 470 V/6, FORTRAN

3. DPS ON NON-UNIFORM DISTRIBUTIONS

Figure 1a illustrates what can happen when we use DPS on any distribution that is non-uniform; some bins will contain more items on the average than other bins. In the illustration the curve represents an exponential distribution. Overlaying the distributive bins on top of the curve clearly indicates that in the expected sense bin 1 will contain more items than bin 2, bin 2 more items than bin 3, bin 3 more items than bin 4, and so forth for each pair of consecutive bins. For a different distribution a completely different scenario might present itself. As an example, consider the standard normal distribution; Fig. 1b. The middle bins would become the most overpopulated while outlier bins, such as bins 1 and 2, would contain very few (if any) items.

All published DPS sorts employ a linear distribution formula under the assumption that the data is nearly uniformly distributed. Historically, there have been several reasons for adopting this strategy. First, if no information is known about the distribution then it is usually best to assume uniformity. Second, gathering statistics in the hope of using this information to derive a non-linear distribution formula could be quite costly. Third, even if gathering statistics was relatively inexpensive, the number of operations needed to implement such a formula (multiplications, additions, special functions, etc.) would most likely be prohibitive [6].

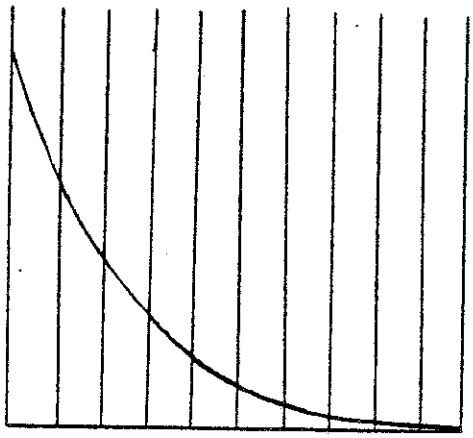

Fig. 1a

Linear distribution scheme: Distributive bins superimposed over exponential curve show that leftmost bins will contain highest percentage of items.

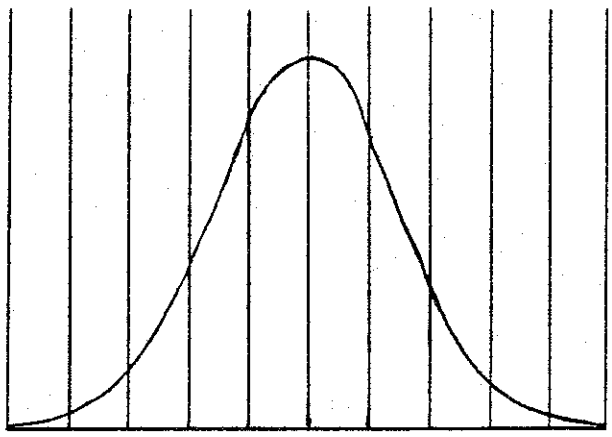

Fig. 1b

For a normal distribution center bins will contain highest percentage of the items.

Another subtle reason for using a linear distribution formula can be deduced from Figures 1a and 1b. After the first distributive pass, the items in each bin will be much closer to a uniform distribution than the original set of items. Therefore, it should take only one more distributive pass on the average to approximately sort all the bins. After this a simple sorting technique, such as bubblesort or insertion sort, can be used to complete the sorting process.

4. TOWARDS A NEW DISTRIBUTIVE METHOD

The effect of using a non-linear distribution formula is illustrated in Figure 2. The width of the bins is smaller when there are more items and larger when there are less items. Therefore, each bin will contain approximately the same number of items. Can we sort faster than quicksort by gathering statistical information and using this information to distribute the items in a non-linear fashion? Surprisingly, there has been little research directed towards answering this seemingly straightforward question. Only in [6] has it been mentioned that this course of attack might yield a fast method. Our goal is set -- we will attempt to develop a O(n) statistical sorting method which outperforms quicksort for nearly all distributions.

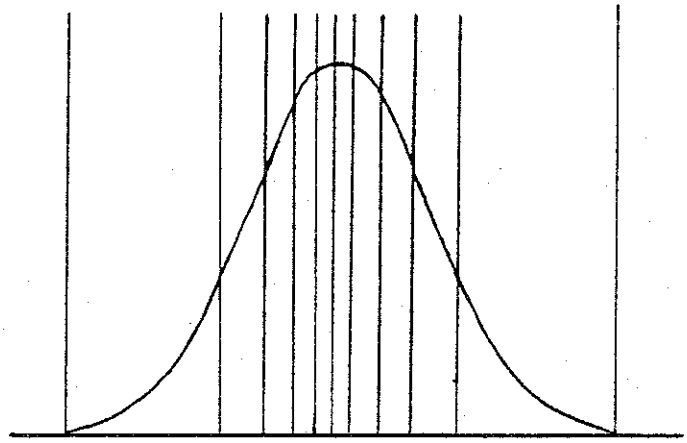

Fig. 2

Non-linear distribution scheme: Each bin will contain approximately the same number of items. Bins will vary in width depending upon shape of distribution curve.

The new method will clearly require two sub-tasks, one to gather the statistics, and another to derive the correct formulae needed to distribute the items into the bins. Hopefully, the end result of the distributive pass will be such that each bin will contain approximately the same number of items.

5. GATHERING STATISTICS

To insure speed, we have chosen to systematically sample items 1, 1+k, 1+2k, 1+3k, ... 1+mk. k is the period of the sample, meaning that every kth element is to be sampled. m is chosen so that 1+mk <= n < 1+(m+1)k. First, we locate the minimum and maximum items of the sample and divide the range into a number of equally sized intervals. All of the sampled items are then linearly distributed to obtain a relative frequency count for each interval. The computation can be carried out in O(m+1) <= O(n) time, where m+1 is the number of samples, by using a simple distributive pass. The following (commented) PASCAL code illustrates the basic mechanism, where a is the array to be sorted:

```
{ Find min and max items in sample }

samplemin := a[1];
samplemax := a[1];
i := 1 + k;
WHILE i <= n DO
  BEGIN
    IF      a[i] < samplemin THEN samplemin := a[i]
    ELSE IF a[i] > samplemax THEN samplemax := a[i];
    i := i + k
```

```
  END;

{ Divide the range [samplemin,samplemax] into
  a number of equally spaced intervals (nintervals)
  and obtain a frequency count of the number of sampled
  items in each interval.  The code is a little tricky
  in the sense that we want all the items distributed
  into intervals 0 through (nintervals - 1).  However,
  because of computer arithmetic (some machines round,
  others chop), the computation of j can be as large
  as nintervals.  The remedy is to add the contents
  of the frequency counts for intervals
  (nintervals - 1) and nintervals and store the
  result in freq[nintervals - 1]. }

IF n < 1000 THEN nintervals = 10
            ELSE nintervals := round(n/100.0);
sampleconstant := nintervals/(samplemax - samplemin);
FOR i := 0 TO nintervals DO freq[i] := 0;
i := 1;
REPEAT
  j := trunc((a[i] - samplemin) * sampleconstant);
  freq[j] := freq[j] + 1;
  i := i + k
UNTIL i > n;
freq[nintervals - 1] := freq[nintervals - 1]
                              + freq[nintervals];
```

We believe that k should be rather small (say around 5), thus yielding a very large sample. There are two reasons. First, the expense of taking a large sample will be quite small when compared with the expense of distributing all n items, which will have to be carried out later in a distributive pass. Second, there is no reliable method for estimating the standard error of the mean when we use a systematic sampling scheme. However, a larger sample size tends to nullify a periodic variation in the data, except if the data has a period of exactly k. It is clear that systematic sampling is a very good method when the data are randomly distributed, approximately in order, or approximately in reverse order [12].

The number of sample intervals (nintervals) does not have to be large. A good choice is to let this quantity be 1 percent of the number of items. However, if n is less than 1000, the number of sample intervals should not go below about 10 [12].

6. NON-LINEAR DISTRIBUTION SCHEMES

One idea is to somehow match the statistical data against some set of standard distributions. The cumulative distribution function (CDF) corresponding to the closest match could then be used to distribute the items into O(n) bins. Unfortunately, there is no known fast heuristic which can carry out the match with any reliability. Even if it were possible to quickly and reliably estimate the distribution type, a significant number of arithmetic operations would be necessary (in many cases) to implement the CDF.

A second idea is to use curve fitting techniques, such as the method of least-squares, to approximate the CDF based upon the statistical data. We have actually implemented this idea with some success. The key is that the polynomial which approximates the curve must be of a rather low degree; otherwise the number of multiplications and additions needed to find the polynomial as well as carry out the distributive pass would make sorting by this method more time consuming than quicksort. With this restriction in mind, we employed the method of least-squares to fit a third degree polynomial to the accumulated sample data. This polynomial (factored by Horner's

method) was then used to distribute the unsorted vector. We found this method to be slightly faster than quicksort for n < 2000. We tested both the uniform and standard normal distributions. However, for large n it became necessary to use more intervals to gather the statistical data. It is a well known result that with additional data points it becomes harder for a least-squares curve of a fixed degree to exactly represent the data. Therefore, the variance in the population size of the distributive bins began to increase dramatically as n grew larger and larger. Outlier bins near the minimum and maximum of the sampled data tended to become very overcrowded. This is the main reason this method was abandoned.

The failure of the CDF least-squares method points the way to a third distributive method which is both elegant and straightforward. The idea is to use a piecewise linear approximation, where the number of bins is directly proportional to the number of samples falling into each particular interval. Thus, each item must pass through two distributive formulae to locate its destination bin. The second distribution formula will be specific to the particular statistical interval concerned. We illustrate the idea by giving an example.

Suppose that n = 1000 and we are taking a sample size of 10 percent. We will use 10 sample intervals. After sampling let us assume that the following frequencies were recorded for each of the intervals 0 through 9:

interval	0	1	2	3	4	5	6	7	8	9
frequency	2	8	6	10	13	12	12	10	13	14

Furthermore, let us assume that the sample min was found to be 0, and the sample max 100. If n bins were to be used to distribute the items over the entire range then we would break interval 0 into 20 bins, interval 1 into 80 bins, interval 2 into 60 bins, ..., and interval 9 into 140 bins. The 20 bins of interval 0 would cover the range [0,10), the 80 bins of interval 1 would cover [10,20), ..., and the 140 bins of interval 9 would cover [90,100]. Any item less than the sample min would go into bin 0, and any item greater than the sample max would go into bin 1000. The distribution formula for (say) interval 2 would be as follows:

$$(x - 20)/(30 - 20) * 60 + 100$$

where x is an item that falls into interval 2. Code excerpts in the next section illustrate how it is possible to efficiently compute the distribution constants for each interval.

It should now be clear why we have chosen the name double distributive partitioning. As we shall see, not only does this method approximate the CDF over the sample range, it has the further advantage of being computationally robust unlike the least-squares method.

7. IMPLEMENTATION AND STORAGE REQUIREMENTS

Using the information gained from the frequency count of the sample we can conveniently compute the necessary constants needed for the piecewise distribution formulae. One distribution formula is computed for each sample (or statistical) interval. The following PASCAL excerpt demonstrates the actual mechanism. The parameter c is used to vary the total number of bins that will be used over all intervals of the double distributive pass. For example, if c = 0.5 then n/2 bins will be used over the entire sample range. offset[i] holds the number of bins used over all previous sample intervals; that is, over statistical intervals 0 through (i-1), i > 0. The array intervalmin holds the minimum allowable value for each statistical interval, and nbins_in_interval is the number of distributive bins that will be employed over a particular interval.

```
kc := k * c;
intervalwidth := (samplemax - samplemin)/nintervals;
offset[0] := 0;
nbins_in_interval := round(kc * freq[0]);
intervalmin[0] := samplemin;
constant[0] := nbins_in_interval / intervalwidth;
FOR i := 1 TO nintervals-1 DO
  BEGIN
    offset[i] := offset[i-1] + nbins_in_interval;
    nbins_in_interval := round(kc * freq[i]);
    intervalmin[i] := intervalmin[i-1] + intervalwidth;
    constant[i] := nbins_in_interval / intervalwidth
  END;
extrabin := offset[nintervals-1] + nbins_in_interval;
```

The next section of PASCAL code contains the double distributive pass. A singly linked-list is used to represent the items of each bin. In this way, we avoid having to physically move the elements during the distributive pass. An extra bin (calculated above) is used to catch those items which will overflow the computation of j2 in the last sample interval. Again, this is necessary since certain machines (such as the VAX 11/780) round floating point computations. Below, the variable sampleconstant (see code excerpt in section 5) is used to compute the statistical interval to which each item belongs.

```
FOR i := 0 TO extrabin DO lhead[i] := 0;
FOR i := 1 TO n DO
  BEGIN
    IF (a[i] <= samplemin)
      THEN j2 := 0
    ELSE IF (a[i] >= samplemax)
      THEN j2 := extrabin
    ELSE
      BEGIN
        j1 := trunc((a[i] - samplemin)
                * sampleconstant);
        j2 := trunc((a[i] - intervalmin[j1])
                * constant[j1] + offset[j1])
      END;
    link[i] := lhead[j2];
    lhead[j2] := i
  END;
```

At the completion of the above computation each empty list will have a list head equal to zero, and each non-empty list will have a terminating zero link. By making one pass through all the lists, the contents of the bins may be quickly rearranged into a destination array b. An insertion sort may then be employed to complete the sorting process [13]. The

following PASCAL code completes the algorithm.

```
{ dump linked-list into b }

i := 0;
FOR j2 := 0 TO extrabin DO
  IF lhead[j2] <> 0
    THEN
      BEGIN
        next := lhead[j2];
        i := i + 1;
        b[i] := a[next];
        WHILE link[next] <> 0 DO
          BEGIN
            next := link[next];
            i := i + 1;
            b[i] := a[next]
          END
      END;

{ insertion sort }

b[n+1] := infinity;  { infinity is a large real number }
FOR i := n-1 DOWNTO 1 DO
  IF b[i] > b[i+1] THEN
    BEGIN
      temp := b[i];
      j := i + 1;
      REPEAT
        b[j-1] := b[j];
        j := j + 1
      UNTIL b[j] > temp;
      b[j-1] := temp
    END
```

The amount of storage required to implement the algorithm is only slightly greater than that used by most of the hybrid DPS algorithms. Assuming that items and linked-list pointers each require one word of storage, the total array storage required is (3n + cn) words (c is the constant that determines the total number of bins used in the double distributive pass).

8. AVERAGE AND WORST CASE ANALYSIS

We will assume that the items consist of a set of n independent, identically distributed random variables with density function f. As we have noted, the sampling process takes O(m+1) < O(n) time. To distribute the items and dump them into array b also takes O(n) time. The time for the insertion sort can be calculated by considering the time it takes to carry out an insertion sort on each individual bin. To sort bins 1 through (cn-1) will take O(n) time for the following density functions: exponential, gamma, beta, binomial, uniform, normal, plus all other densities which have either exponentially vanishing tails or compact support (without strong peaks). The proof follows immediately from the main result given by Devroye and Klincsek in [4], where they show that it takes O(n) time to sort using a distributive pass followed by a bubblesort pass. Essentially, our result follows because the systematic samples gathered from the random distribution are also inherently identically distributed random variables. Thus, the analysis of the distributive pass over each statistical interval reduces to the one given by Devroye and Klincsek for their theoretical sort over the entire range of the items.

To approximate the time to sort bins 0 and cn will require an estimate of how many items fall into these bins. An expression for the expected number of items in these two intervals is

$$\text{expected in bin cn} = n\int_{\text{samplemax}}^{\max} xf(x)dx < \alpha_1 n,$$

$$\text{expected in bin 0} = n\int_{\min}^{\text{samplemin}} xf(x)dx + \beta < \alpha_2 n.$$

The constant β is the number of items in statistical interval 1 falling into bin 0. That β is a constant follows from the assumption that the n items are random variables. It turns out that the integral in both expressions can be bounded above (as n gets large) by positive constants α_1 and α_2 for the set of density functions given above. Thus the total time taken by the algorithm is

$$T(n) \leq (\alpha_1 n)^2 + O(n) + (\alpha_2 n)^2 + 2\alpha_2\beta + \beta^2.$$

In most instances α_1 and α_2 will be quite small (< 0.1) and thus the running time exhibited by the algorithm will be dominated by the O(n) term.

The worst case of the algorithm is $O(n^2)$ since it is possible that after the distributive pass one bin may contain as many as (n-1) items. The result follows immediately from the fact that insertion sort has a worst case $O(n^2)$ running time.

We note that the worst case can be improved by making a slight modification to the algorithm. After the items are distributed a check of the number of items in each bin can be made. If a bin contains more than (say) 10 items, heapsort [14] can be employed to finish ordering it. Since heapsort always requires O(nlogn) time to sort a vector of size n, the worst case of the modified DDP sort would then be O(nlogn).

Note also that the average T(n) in equation (1) will always be less than or equal to O(nlogn) if heapsort is used on bins 0 and cn.

9. TEST RESULTS

PASCAL implementations of DDP and quicksort were compared on a DIGITAL VAX 11/780. The tests were conducted with 0.01n sample intervals and n/2 bins (c = 0.5) for DDP and a smallest partition size of 8 (in lieu of insertion sort) in Sedgewick's quicksort. Tests were restricted to four distribution types: uniform, standard normal, gamma (alpha = 2.0, beta = 1.0), and exponential. IMSL subroutines GGUBS, GGNML, and GGAMR were used for random number generation [15]. Instead of obtaining CPU timings, counts of certain fundamental operations were obtained by placing software counters into the DDP and quicksort codes. The set of tallied operations included:

1. Arithmetics: Addition, subtraction, multiplication, division.
2. Assignment: Replacing the value of a variable by the value of another variable, or the value of a constant.
3. Comparison: Comparing the values of two variables, a variable and a constant, or two constants.

No distinction was made between the use of simple variables and array variables or their types (real, integer, boolean) in any of the operation counts. In some cases the code contained composite statements involving several

operations. These were broken into the requisite number of assignments, comparisons, and arithmetics. Exceptions were arithmetic statements such as

```
i := i + k.
```

These were counted as only one arithmetic; the assignment involved was considered to be implicit and was not added to the assignment counter. FOR loop constructs with n repetitions counted as an additional n comparisons and n arithmetics.

A summary of the test results can be found in the graph of Figure 3. The two curves in the graph represent the average of the total number of operations (arithmetics + assignments + comparisons) taken by DDP and quicksort on the set of four distributions. Twenty five test runs were made for each distribution type. One reason we chose not to graph the results of each distribution separately is that the standard deviation amongst the total number of operations for the four types was very small. For example, for n = 32000, these were 78.4 (DDP) and 4376.0 (quicksort). The standard deviation divided by the average number of operations was consistently better for the DDP sort. These results, along with the fact that DDP always sorted with fewer operations than quicksort, indicate the superiority of double distributive sorting.

Additional statistics were gathered for DDP to determine the average number of items falling into the set of non-empty bins, the average number of bins containing more than 8 items, and to determine the size of the largest bin; see Table 3. (The extra operations needed to gather these statistics were not included in the DDP operation counts.) Note the stability in the average size of the non-empty bins, the expected doubling in the number of bins containing more than 8 items, and small growth size of the most populous bin. (The largest bin was the largest one obtained during any one of the 25 test runs.)

The relationship between the number of specific operations required by each algorithm varied with the size of n. For n = 1000, (DDP:quicksort), the ratios were (12.7:14.1) for arithmetics, (5.7:10.2) for assignments, and (8.4:14.5) for comparisons. For n = 32000, the respective ratios were (10.1:16.1), (18.2:43.9), and (26.9:69.3). Obviously, the widening performance gap as n grows larger can be attributed to the expected complexity of each method.

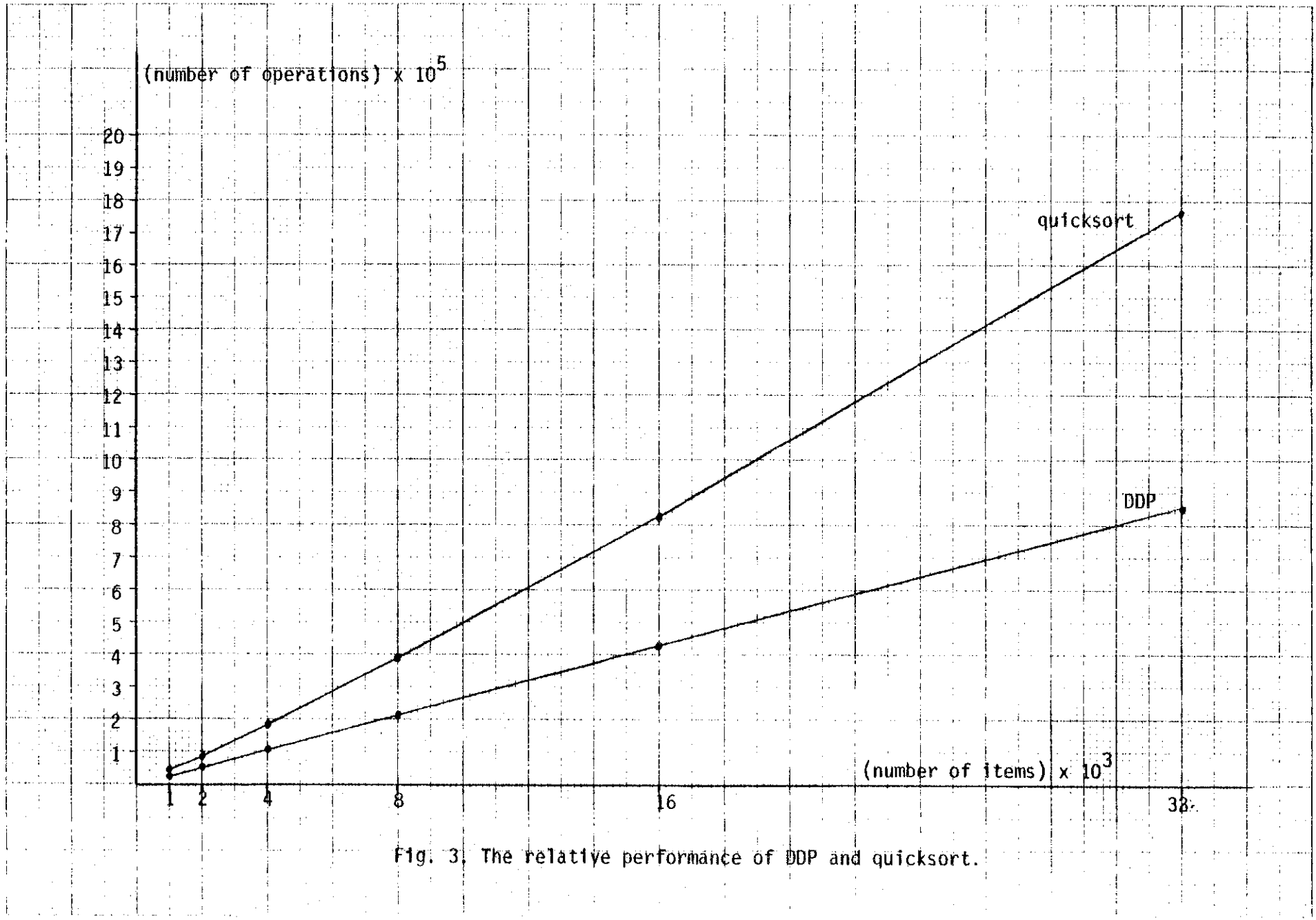

Fig. 3. The relative performance of DDP and quicksort.

Table 3. Some statistics gathered from tests on the DDP sort: Average size of non-empty bins (non-empty), average number of bins containing more than 8 items (big 8), and size of largest bin during each test (largest).

n		UNIFORM	NORMAL	EXPO-NENTIAL	GAMMA
1000	non-empty	2.321	2.339	2.352	2.325
	big 8	1.440	1.480	1.840	1.600
	largest	18	26	22	20
2000	non-empty	2.325	2.329	2.335	2.328
	big 8	2.240	3.560	2.760	2.320
	largest	19	23	28	36
4000	non-empty	2.322	2.335	2.321	2.329
	big 8	4.160	4.400	4.680	5.160
	largest	18	19	22	25
8000	non-empty	2.325	2.321	2.325	2.325
	big 8	7.800	8.640	8.240	8.960
	largest	15	26	33	27
16000	non-empty	2.322	2.325	2.325	2.321
	big 8	15.560	18.240	18.080	16.200
	largest	33	28	30	28
32000	non-emtpy	2.327	2.326	2.320	2.323
	big 8	30.000	33.360	30.520	32.200
	largest	20	31	30	44

10. CONCLUSIONS

For approximately 20 years it has been the goal of computer scientists to find a sorting method which runs faster than quicksort on a real machine, independent of the distribution of items involved in the sort. We believe that the method presented, which has linear expected time complexity, realizes this goal.

Quicksort still possesses some advantages over DDP, such as a smaller storage requirement and superior locality of reference, important issues in the paging environments of modern mainframe computers. However, in most cases, we believe that DDP will outperform quicksort when a large amount of real store is available to handle the sort computation.

Two issues unresolved by the present research involve the adaptability of DDP to parallel computation and the search for a more robust sampling scheme that might allow DDP to perform with even greater consistency over a wider class of inputs.

11. ACKNOWLEDGEMENTS

We would like to thank Pete Mancuso for several helpful discussions concerning statistical distribution theory, Carolyn Bjorklund for reading an earlier version of the manuscript, and a referee for suggesting some modifications to the DDP code which resulted in a more efficient implementation.

REFERENCES

[1] C.A.R. Hoare, Quicksort (Algorithm 64), CACM 4, no. 7 (1961), p. 321.

[2] R. Sedgewick, Implementing quicksort programs, CACM 21, no. 10 (1978), pp. 847-856.

[3] W. Dobosiewicz, Sorting by distributive partitioning, Info. Proc. Lett. 7, no. 1 (1978), pp. 1-6.

[4] L. Devroye and T. Klincsek, Average time behavior of distributive sorting algorithms, Computing 26, no. 1 (1981), pp. 1-7.

[5] M.T. Noga, Fast Geometric Algorithms, Ph.D. Thesis, Dept. of Comp. Sci., Virginia Polytechnic Institute and State University, Blacksburg, VA 24061 (1984).

[6] H. Meijer and S.G. Akl, The design and analysis of a new hybrid sorting algorithm, Info. Proc. Lett. 10, no. 4-5 (1980), pp. 213-218.

[7] S. Baase, Computer Algorithms: Introduction to Design and Analysis, Addison-Wesley (1978).

[8] R. Loeser, Some performance tests of "quicksort" and descendants, CACM 17 (1974), pp. 143-152.

[9] M. van der Nat, A fast sorting algorithm, a hybrid of distributive and merge sorting, Info. Proc. Lett. 10, no. 3 (1980), pp 163-167.

[10] J.S. Kowalik and Y.B. Yoo, Implementing a distributive sort program, Journal of Information and Optimization Sciences 2, no. 1 (1981), pp. 28-33.

[11] R.C. Singleton, An efficient algorithm for sorting with minimal storage, CACM 12, no. 3 (1969), pp. 185-186.

[12] G.W. Snedecor and W.G. Cochran, Statistical Methods, Iowa State Univ. Press (1967).

[13] D.C.S. Allison and M.T. Noga, Usort: an efficient hybrid of distributive partitioning sorting, BIT 22 (1982), pp. 135-139.

[14] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley (1974).

[15] International Mathematical and Statistics Library, Edition 8, (1980).