

Parallel Approximation Algorithms for Bin Packing

RICHARD J. ANDERSON

*Department of Computer Science, University of Washington,
Seattle, Washington*

ERNST W. MAYR

*Department of Computer Science, Stanford University,
Stanford, California*

AND

MANFRED K. WARMUTH

*Department of Computer Science, University of California,
Santa Cruz, California*

We study the parallel complexity of polynomial heuristics for the bin packing problem. We show that some well-known (and simple) methods like first-fit-decreasing are \mathcal{P} -complete, and it is hence very unlikely that they can be efficiently parallelized. On the other hand, we exhibit an optimal \mathcal{NC} algorithm that achieves the same performance bound as does FFD. Finally, we discuss parallelization of polynomial approximation algorithms for bin packing based on discretization.

© 1989 Academic Press, Inc.

1. INTRODUCTION

In this paper we investigate the parallel complexity of bin packing. Since bin packing is \mathcal{NP} -complete, there is little hope for finding a fast parallel algorithm to construct an optimal packing. However, quite a few efficient approximation algorithms have been developed for bin packing, so it is natural to ask if fast parallel algorithms exist that find provably good packings.

The bin packing problem requires to pack n items, each with size

The first two authors were supported in part by a grant from the AT&T Foundation, ONR contract N00014-85-C-0731, and NSF grant DCR-8351757; the third author acknowledges the support of ONR grant N00014-86-K-0454.

$\in (0, 1)$, into a minimal number of unit capacity bins. For an instance I of the problem, $\text{OPT}(I)$ will denote this number.

There have been two different approaches taken in studying sequential approximation algorithms for bin packing. One has been to look at simple heuristics and to analyze their behavior. A prominent example of such a heuristic is *first fit decreasing* (FFD). It considers the items in order of non-increasing size, and places each item into the first bin that has enough space remaining. It has been shown that the length of the packing generated by FFD is at most $\frac{11}{9} \text{OPT}(I) + 3$ (Baker, 1985; Johnson *et al.*, 1974). The other approach for approximation algorithms is to look for algorithms with a performance bound of $(1 + \varepsilon) \text{OPT}(I)$ (Fernandez de la Vega and Lueker, 1981; Karmarkar and Karp, 1982). Although these algorithms give an asymptotically better performance bound, the known algorithms of this type are complicated and have large runtimes. In this paper we are primarily concerned with parallel algorithms using the first approach, i.e., implementing simple packing heuristics that are relatively close to optimal. However, in the final section we briefly discuss a parallel implementation of the $(1 + \varepsilon) \text{OPT}(I)$ algorithm due to Fernandez de la Vega and Lueker (1981).

There are two reasons for investigating the extent to which simple bin packing algorithms can be implemented as fast parallel algorithms. The first reason is to develop good parallel algorithms for bin packing, with good time and processor bounds and close to optimal performance. Furthermore, if the analysis of the sequential algorithms carries through to the parallel case, we can avoid the monumental task of analyzing a bin packing algorithm from scratch. Bin packing is closely related to certain scheduling problems since the items can be viewed as tasks to be scheduled on a set of processors with the size of the items being interpreted as the processing time needed. Thus it is conceivable that an efficient parallel algorithm for bin packing could be of use for scheduling tasks on a multi-processing system.

The other reason for attempting to implement the simple bin packing heuristics as fast parallel algorithms is to investigate the nature of sequential algorithms versus parallel algorithms. A number of sequential algorithms, such as the greedy algorithms for computing a maximal independent set and computing a maximal path can be shown to be inherently sequential. The bin packing heuristics also seem quite sequential in nature, so it is important to examine to what extent this is inherent. The goal is to gain insight into what types of algorithms can be sped up substantially with parallelism and what algorithms probably cannot.

In this paper we use the PRAM model of parallel computation (Fortune and Wyllie, 1978). We consider a parallel algorithm to be fast if it is an \mathcal{N}^c algorithm (Pippenger, 1979), i.e., if it runs in polylogarithmic time

using a polynomial number of processors. However, the main algorithm that we give will obey a far more reasonable bound, running in $O(\log n)$ time on an $n/\log n$ processor EREW (exclusive read, exclusive write) PRAM, and hence is asymptotically optimal. We say a problem is inherently sequential if it is \mathcal{P} -complete. This is relatively strong evidence that the problem is not in \mathcal{NC} , since if it were, then $\mathcal{P} = \mathcal{NC}$. We shall occasionally refer to an algorithm as being a \mathcal{P} -complete algorithm. The proper interpretation of this is that deciding the value of a specified bit of the output of the algorithm is \mathcal{P} -complete (Anderson and Mayr, 1987).

The main results of this paper are that the FFD heuristic is a \mathcal{P} -complete algorithm, and that a packing that obeys the same performance bound as FFD can be computed by a fast parallel algorithm. The \mathcal{P} -completeness result holds even if the problem is given with a unary encoding. This is interesting since most known \mathcal{P} -complete number problems, such as network flow (Goldschlager, Shaw, and Staples, 1982) and list scheduling (Helmhold and Mayr, 1987) can be solved by fast parallel algorithms if the numbers involved are small. A notable exception is linear programming which is also strongly \mathcal{P} -complete (Dobkin, Lipton, and Reiss, 1979). Our algorithm for constructing a packing that obeys the same $\frac{1}{9}$ bound as FFD, packs the large items (items of size $\geq \frac{1}{6}$) in the same manner as FFD and then fills in the remaining items. The algorithm runs in $O(\log n)$ time using $n/\log n$ processors. The packing algorithm generalizes to an algorithm that constructs an FFD packing in time $O(\log n)$ for all instances where all items are of size at least $\varepsilon > 0$. It can thus be viewed as an approximation scheme to FFD. As a subroutine, we also develop a new and optimal EREW-PRAM algorithm to match parentheses.

2. \mathcal{P} -COMPLETENESS PROOF FOR FFD

In this section, we prove that, in all likelihood, the FFD bin packing heuristic is not efficiently parallelizable. More formally, we show that the problem whether FFD places a distinguished item into a certain bin is \mathcal{P} -complete in the strong sense, i.e., it is \mathcal{P} -complete even if the items are given using a unary notation. Thus, to compute an FFD packing is difficult in parallel even for "small" item sizes, i.e., item sizes that are fractions with small integer numerators and denominators. This should be compared to the parallel complexity of other number problems (or problems involving numbers in an essential way), like network flow (Goldschlager *et al.*, 1982; Karp, Upfal, and Wigderson, 1985) and list scheduling (Helmhold and Mayr, 1987). These are \mathcal{P} -complete only in the weak sense and can be solved in \mathcal{NC} or \mathcal{RNC} if the numbers involved are small.

THEOREM 1. *Given a list of items, each of size between 0 and 1, in non-increasing order, and two distinguished indices i and b , it is \mathcal{P} -complete to decide whether the FFD heuristic will pack the i th item into the b th bin. This is true even if the item sizes are represented in unary.*

Proof. For the proof we use a reduction from the following variant of the monotone circuit value problem: a circuit consists of AND and OR gates whose fan-out is at most two. This restricted version is clearly \mathcal{P} -complete as can be seen by an easy log space reduction from the general monotone circuit value problem (Ladner, 1975). The details of the construction are omitted here.

Our reduction is described in two stages. We first reduce the restricted monotone circuit value problem to an FFD bin packing problem featuring bins of variable sizes. The construction is then modified to give an FFD packing into unit capacity bins.

Let β_1, \dots, β_n be the gates of an n -gate monotone circuit, i.e., each β_i is either AND(i_1, i_2) or OR (i_1, i_2), with i_1 and i_2 the inputs of the gate. Each input can be a constant (**true** or **false**), or the value of some other gate $\beta_j, j < i$. In our first construction, we transform the sequence β_1, \dots, β_n into a list of items and a list of bins. The list of item sizes will be non-increasing. For every gate β_i , we obtain a segment for each of the two lists. The segments for each list are concatenated in the same order in which the gates are given. For ease of notation, let

$$\delta_i = 1 - \frac{i}{n+1} \quad \text{and} \quad \varepsilon = \frac{1}{5(n+1)}.$$

The list segments for each gate are determined by Table I, where gate β_i is assumed to feed into gate β_j if it has just one output and into gates β_j and β_k otherwise.

TABLE I
Bins and Item Sizes for Various Types of Gates

	bins	Items
Fan-out one		
AND	$\delta_i, \delta_i + \delta_j - 2\varepsilon$	$\delta_i, \delta_i, \delta_i - 2\varepsilon, \delta_i - 2\varepsilon$
OR	$\delta_i + \delta_j - 2\varepsilon, \delta_i$	$\delta_i, \delta_i, \delta_i - 2\varepsilon, \delta_i - 2\varepsilon$
Fan-out two		
AND	$\delta_i, 2\delta_i - 4\varepsilon, \delta_i + \delta_j - 3\varepsilon, \delta_i + \delta_k - 4\varepsilon$	$\delta_i, \delta_i, \delta_i - 2\varepsilon, \delta_i - 2\varepsilon, \delta_i - 2\varepsilon, \delta_i - 3\varepsilon, \delta_i - 4\varepsilon$
OR	$2\delta_i - 4\varepsilon, \delta_i, \delta_i + \delta_j - 3\varepsilon, \delta_i + \delta_k - 4\varepsilon$	$\delta_i, \delta_i, \delta_i - 2\varepsilon, \delta_i - 2\varepsilon, \delta_i - 2\varepsilon, \delta_i - 3\varepsilon, \delta_i - 4\varepsilon$
Gate β_n :	δ_n, δ_n	$\delta_n, \delta_n, \delta_n - 2\varepsilon, \delta_n - 2\varepsilon$

Let T_i denote any item of size δ_i , and F_i any item of size $\delta_i - 2\epsilon$. For every constant input of gate β_i , a T_i is removed from its list of items if the input is **false** and an F_i if it is **true**.

We claim that packing the list of items (which is clearly non-increasing) into the sequence of bins according to the FFD heuristic, emulates evaluation of the circuit in the following sense. Consider the bins in list order. When we start packing into the first bin of β_i 's segment, for $i = 1, \dots, n$, the remaining list of items starts with β_i 's segment, and two of the first four items in this segment have already been removed. The other two of these four items encode the values of the two inputs to gate β_i : a T_i stands for a **true** input, F_i for **false**. Suppose β_i is an AND-gate with fan-out two. Then β_i 's second bin receives a T_i if both of its inputs are **true**, and an F_i otherwise. In the first case, the second bin can further accommodate only the last item in β_i 's list, whereas in the second case, it has still room for the third-to-last item in the list. As a result, packing β_i 's items leaves space in the amount of $\delta_j - \epsilon$ and $\delta_k - \epsilon$ in β_i 's last two bins if β_i evaluates to **true**. If the output of β_i is **false**, the corresponding amounts are δ_j and δ_k . Thus, in the first case, F_j and F_k will also be packed into the last two of β_i 's bins since they are the largest items to fit. In the other case, T_j and T_k fit and will be packed. Therefore, after both inputs to β_j (similarly, β_k) have been evaluated, the two *remaining* of the first four items in β_j 's (resp., β_k 's) segment again properly reflect the values of the two inputs to the gate. Figure 1 shows the packings for two input combinations to a fan-out two OR-gate. The OR-gate functions quite similarly to the AND-gate just described, with the role played by the first two bins more or less reversed. The details of the simulations performed by the other types of gates listed in Table I are left to the reader.

In the second part of the construction, we show how to use unit size bins. Let u_1, \dots, u_q be the non-increasing list of item sizes, and let b_1, \dots, b_r

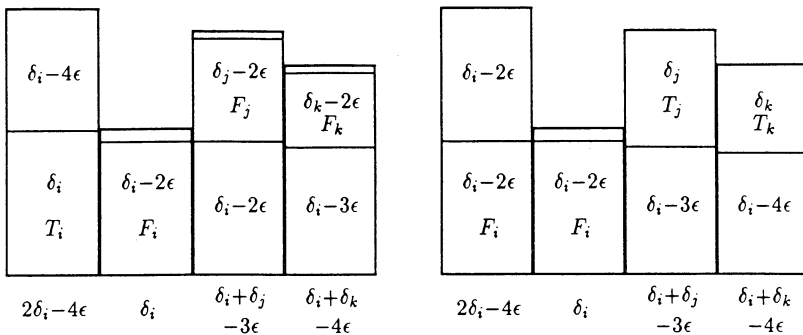


FIG. 1. Packing for OR gate with (a) one **true** input, (b) two **false** inputs.

be the list of variable bin sizes obtained in the first part. Define B to be the maximum of the b_i , and let $C = (2r + 1)B$. We construct a list of decreasing items v_1, \dots, v_{2r} which when packed into r bins of size C leave space b_i in the i th bin. Let

$$v_i = \begin{cases} C - iB - b_i, & \text{if } i \leq r; \\ C - iB, & \text{if } i > r. \end{cases}$$

When these items are packed according to the FFD heuristic, items v_i and v_{2r+1-i} end up in the i th bin, thus leaving b_i empty space. Also note that v_{2r} is at least as large as u_1 . Let w_1, \dots, w_{2r+q} be the list of item sizes obtained by concatenating the v - and u -lists, and normalizing the sizes by dividing each of them by C . Assume without loss of generality that the output gate β_n of the given circuit is an AND-gate. An FFD packing of the items in the w -list into unit bins will place the item corresponding to the second T_n in β_n 's list into the last bin iff the output of the circuit is **true**.

The two parts of the construction described above can clearly be carried out on a multitape Turing machine using logarithmic work space. Since all numbers involved in the construction are bounded in value by a polynomial in the size of the circuit, we have shown that FFD bin packing is \mathcal{P} -complete in the strong sense (under log space reductions), i.e., it remains \mathcal{P} -complete even if numbers are represented in unary (with fractions given by a pair of integers). ■

FFD is a rather simple sequential algorithm to achieve bin packings relatively close to optimal. As we have just seen, however, it is another example of a \mathcal{P} -complete algorithm, a notion introduced in (Anderson and Mayr, 1987).

A number of other simple heuristics for bin packing can also be shown to be \mathcal{P} -complete, e.g., *best fit decreasing* (BFD). The BFD heuristic considers items in order of non-increasing size. It places each item into a bin in such a way as to minimize the leftover space.

3. A PARALLEL ALTERNATIVE FOR FFD

Even though the FFD heuristic itself appears to be inherently sequential we are able to give an \mathcal{NC} -algorithm for bin packing that achieves the same overall performance bound as FFD. This algorithm works in two stages. The first stage relies on

THEOREM 2. *The packing obtained by the FFD heuristic can be computed by an \mathcal{NC} -algorithm for instances where all items have size at least $\varepsilon > 0$. On a sorted list of n items, the algorithm uses $n/\log n$ processors and runs in time $O(\log n)$.*

The proof of this theorem will be given in the next two sections. Here, we show how to apply it to get a good parallel alternative for FFD. Our two-stage algorithm first packs all items of size at least $\frac{1}{6}$ according to FFD, using the above algorithm. The second stage uses the remaining items to fill bins up in a greedy fashion. It makes sure that each bin is filled to at least $\frac{5}{6}$ before it proceeds to the next. We call the resulting packing a *composite* packing. There are a number of possible algorithms to use for the second stage. One possibility is to use the *first-fit-increasing* heuristic (FFI). An FFI packing can be computed by an \mathcal{NC} -algorithm, but it is not known how to do so for variable size bins with a linear number of processors. Below, we give a different method which can be implemented with optimal speedup.

The following lemma establishes that the composite packing is within a factor of $\frac{11}{9}$ of optimal. Variants of this lemma have been used extensively in the analysis of bin packing algorithms.

LEMMA 3.1. *The length of the composite packing $L_c(I)$ satisfies*

$$L_c(I) \leq \max\{L_{\text{ffd}}(I), \frac{6}{5} \text{OPT}(I) + 1\} \leq \frac{11}{9} \text{OPT}(I) + 4.$$

Proof. Let L be the length of the FFD packing of the items with size at least $\frac{1}{6}$. Clearly $L \leq L_{\text{ffd}}(I)$, so if all the items packed by the second stage of the algorithm are placed into the first L bins, then $L_c(I) \leq L_{\text{ffd}}(I)$. If more than L bins are used, then all bins except possibly the last one are filled to at least $\frac{5}{6}$, so $L_c(I) \leq \frac{6}{5} \text{OPT}(L) + 1$. ■

We now describe the second stage of the algorithm for constructing the composite packing. It runs in $O(\log n)$ time and uses $n/\log n$ processors. Let u_1, \dots, u_r be a list of items, all of size less than $\frac{1}{6}$. The first step is to combine these items into chunks so that all chunks (except possibly the last) have size between $\frac{1}{24}$ and $\frac{1}{6}$. The items of size at least $\frac{1}{24}$ are big enough, and each is put into a chunk by itself. For the remaining items, the partial sums $s_k = \sum_{1 < j \leq k} u_j$ are determined using optimal prefix summation (Ladner and Fischer, 1980). For each i , we combine the set of items $\{u_k \mid i/12 \leq s_k < (i+1)/12\}$ to form a chunk. Since the items have size less than $\frac{1}{24}$, each chunk will have a size between $\frac{1}{24}$ and $\frac{3}{24}$.

The bins packed by the FFD algorithm with items of size at least $\frac{1}{6}$ can now be filled in. We have, in parallel, each bin filled to less than $\frac{5}{6}$ pick a distinct chunk to add to the bin. Since the sizes are at least $\frac{1}{24}$, only a constant number of passes is needed. Each pass can be implemented using parallel prefix computation.

If there are left over items, they are packed in new bins. The algorithm is similar to the one just used for filling up the bins partially packed by the FFD algorithm, except that we do not know the number of bins to use. Let

u_1, \dots, u_q be the list of left over items (chunks), each of size between $\frac{1}{24}$ and $\frac{1}{6}$, and let $U = \sum_{j=1}^q u_j$. Since each bin can be filled to at least $\frac{5}{6}$, $\lceil 6U/5 \rceil$ bins will certainly suffice. We start our iterative packing with this number of active bins, arranged in an array. In a pass, each active bin determines how many bins to its left (including itself) are filled to less than $\frac{5}{6}$, and how many items are currently packed in bins to its right. Two parallel prefix computations are used to find these numbers. Then the largest index is determined such that, to its right, there are enough items to satisfy the requests up to and including the bin given by the index. The items currently stored in the rightmost bins are used to fill up, one item per bin, the underfull bins to the left of or at the index. Bins that are emptied by this process become inactive. Since the items have size at least $\frac{1}{24}$ a constant number of passes suffices. As above, each pass can be executed in $O(\log n)$ time on an $n/\log n$ processor EREW-PRAM.

The results presented in this and the previous section show that it is the *small* items that make FFD hard to parallelize. Here, *small* need not even be "very small" since, as we have seen, FFD is \mathcal{P} -complete in the strong sense. Using a different approach to pack small items, however, still provides an asymptotically optimal \mathcal{NC} -algorithm to achieve a packing with the same overall performance as FFD.

4. PARALLEL FFD FOR BIG ITEMS

Let $\varepsilon > 0$ be fixed. In this section, we describe our main algorithm. It constructs an FFD packing for lists of items whose size is bounded below by ε . The algorithm runs in time $c_\varepsilon \log n$, where c_ε is a constant depending on ε . The algorithm can be implemented using $n/\log n$ processors on an EREW-PRAM, provided that the input list of items is given in non-increasing order. Otherwise, we have to sort the list first, which, for the stated time bound, requires a linear number of processors.

Performing an FFD packing on a non-increasing list of items can be viewed in two ways. The first is to consider the items in order, move each one down the list of (partially filled) bins and place it into the first bin it fits. An alternate way is to consider the bins one after another, have each move down the list of items and pick up and pack any item that fits into the available space. These two viewpoints lead to two different ways of decomposing the initial problem into simpler parts, and we shall use both methods. We first subdivide the list of items into contiguous sublists in such a way that the item sizes within any sublist are within a factor of two. This can be done generating at most $\lceil \log(\varepsilon^{-1}) \rceil$ sublists. The sublists are packed sequentially since there is only a constant number of them.

Accordingly, the algorithm is subdivided into *phases*, packing in phase i the items with size in $(2^{-(i+1)}, 2^{-i}]$.

In phase i , we can disregard all bins that have space $2^{-(i+1)}$ or less available. Omitting these bins, we obtain a subsequence of bins called the $i+1$ -*projection* of the original list. To pack the sublist of items in phase i , we divide the $i+1$ -projection of the list of (partially packed) bins into *runs*. A *run* is a contiguous segment of bins whose length is maximal subject to the following two conditions.

1. The available space is non-decreasing.
2. There is an integer t , called the *type* of the run, such that the available space in each bin of the run is in the interval $(2^{-(t+1)}, 2^{-t}]$.

A sublist of bins satisfying just the first of these two conditions is called a *pre-run*.

Packing a sublist (or as much of it as fits) into a run is achieved by alternating two routines, *forward_pack* and *fill_in* until no more items fit into bins of the run, or all items in the sublist have been packed. The *forward_pack* routine determines how many consecutive items at the beginning of the list will fit into the first bin of the run. Let this number be k . The routine then determines how many consecutive chunks of k items each can be packed into consecutive bins, following the FFD heuristic. To do so, it checks which bin could actually accommodate the first $k+1$ item chunk. Finally, *forward_pack* packs, in parallel, the chunks of k items into the appropriate number of leading bins of the run, removes these bins from the run, and returns them as a pre-run.

algorithm FFD_pack (L, ε);

```

co  $L$  is a sorted list of  $n$  items to be packed according to FFD; each item has size at least
     $\varepsilon$  oc
 $S := (\rho_0)$ ; co  $S$  holds a list of runs; the initial run  $\rho_0$  consists of  $n$  empty bins oc
for ( $i := 0$ ;  $2^{-i} \geq \varepsilon$ ;  $i++$ ) do
     $L' :=$  sublist of items in  $L$  with sizes  $\in (2^{-(i+1)}, 2^{-i}]$ ;
    if  $L' = ()$  then continue fi; co go to beginning of loop oc
     $S' := ()$ ;
    repeat
         $\rho :=$  first run of the  $i+1$ -projection of  $S$ ;
        forward_pack ( $\rho, \psi$ ); co  $\psi$  is a pre-run oc
         $S'' =$  fill_in ( $\psi$ ); co  $S''$  is a list of runs oc
        remove from runs in  $S''$  bins with less than  $\varepsilon$  space;
        append  $S''$  to  $S'$ 
    until  $L' = ()$ ;
    append the unused portion of the  $i+1$ -projection of  $S$  to  $S'$ ;
     $S := S'$  with the bins not in the  $i+1$ -projection of  $S$  merged back in
od
end.

```

procedure forward_pack (ρ, ψ);
if $i \leq \text{type of } \rho$ **then** $\psi := \rho$; **return fi**;
 let $L' = u_1, \dots, u_j$;
 let $s_1 \leq s_2 \leq \dots \leq s_{|L'|}$ be the amounts of space available in ρ 's bins;
 $k := \max\{j \mid j \leq |L'| \text{ and } u_1 + \dots + u_j \leq s_1\}$; **co** note that $k > 0$ **oc**
 let r be minimal subject to
 1. $r = \min\{|\rho|, \lceil l/k \rceil\}$; or
 2. $(r+1)k < l$ and $u_{rk+1} + \dots + u_{(r+1)k+1} \leq s_{r+1}$;
 remove first r bins from ρ , put them into ψ ;
if $\rho = ()$ **then** remove ρ from S **fi**;
 in parallel for $j = 1, \dots, r$, add items $u_{(j-1)k+1}, \dots, u_{jk}$ to j th bin in ψ ;
return
end.

The pre-run returned by *forward_pack* is subject to fill-in packing. Here, smaller items further down in the list are packed into the space left after the forward packing. The function *fill_in* first breaks the pre-run into runs. If all bins in the pre-run were actually filled by the forward packing (that is, the number r of bins in the pre-run was determined by the second condition for r in procedure *forward_pack*), these runs are all of type greater than the phase number i , and no more items can be packed into them in phase i . Otherwise, if the pre-run contains a run of type i (possibly since *forward_pack* did not pack the run since it was of type i), *fill_in* tries to pack more items into the bins of the run. Due to the constraints on the amount of space left in type i bins and the size of items packed in phase i , at most one additional item per bin can be packed by *fill_in*.

We can compute a fill-in packing by first merging the reversal (which is non-increasing) of the list of amounts of space left in the bins of the run with the list of item sizes. When merging the two lists, we take care that all bins precede all items of the same size. We then interpret the combined list as a string of parentheses, with each bin corresponding to an opening, and each item to a closing parenthesis. The natural matching of the parentheses can be seen to give the assignment of items to bins as obtained by FFD, since every item goes into the smallest possible (and hence last in the reversed list) bin still available, and the items are considered in decreasing order.

The details of the implementation of *fill_in* will be given in the next section where we show that it can be made to run in time $O(\log n)$ on an EREW-PRAM with $n/\log n$ processors.

Assuming these resource bounds, we state

THEOREM 3. *Algorithm FFD_pack(L, ϵ) runs in time $c_\epsilon \log n$ on an $n/\log n$ processor EREW-PRAM. The constant c_ϵ is polynomial in $1/\epsilon$.*

Proof. To analyze the complexity of *FFD_pack* we introduce a generalization of the concept of a run: A *stacked run* or *s-run* of type j is a run of type j obtained from the $j+1$ -projection of the list of bins. As a

consequence, an s -run of type j may be composed of several runs of type j separated, in the original list, by runs of higher types. Because of this, the number of runs can be larger than the number of s -runs, but at most by a factor of two. To every s -run of type j , we assign a weight of 2^{-2j} . The weight of a list of bins is the sum of the weights of all its s -runs.

Consider the effect of forward packing items in phase i into bins of an s -run of type j , $j < i$. Note that the items packed by the forward packing are not necessarily a contiguous sublist since some of the items may be used as fill-ins. For the moment, we assume that enough items are available to fully pack all bins in the s -run in the forward packing. Disregarding fill-in items, the forward packing of the s -run can create at most $2^{-j}/2^{-(i+1)} - 2^{-(j+1)}/2^{-i} = \frac{3}{2} 2^{i-j}$ pre-runs which all decompose into runs of type $i+1$ or higher (at most one run of any type per pre-run). Thus, the weight of the s -runs resulting from forward packing to capacity one s -run of type j (and disregarding fill-in terms) in phase i is bounded by

$$\frac{3}{2} 2^{i-j} \sum_{k>i} 2^{-2k} < 2^{-2j}.$$

The forward packing routine may also leave a partially filled bin or fail to pack a whole s -run to capacity when it runs out of items. Since at most one s -run of every type can be only partially packed in this way, this adds, for the whole phase, a weight bounded by

$$\sum_{k \geq 0} 2^{-2k} = \frac{4}{3}.$$

Next, we consider the effect of the fill-in routine on the weight of s -runs. In phase i , *fill_in* is going to affect only s -runs of type i . Suppose when filling in an s -run of type i *fill_in* creates two new s -runs of some type $j > i$. Then all items added to the bins in the first s -run come after the items of the second s -run in the item list. Let u be the size of the fill-in item packed into the first bin of the first new s -run, and v the size of the fill-in item in the second s -run. Since the item of size v came earlier in the item list, it did not fit into the first bin of the first run. After the item of size u is packed into this bin, there is still an amount of space larger than $2^{-(j+1)}$ left since the s -run is of type j . Hence, $v > u + 2^{-(j+1)}$. We conclude that every s -run of type j generated by *fill_in* except the last one accounts for a drop of at least $2^{-(j+1)}$ in item size. Since all item sizes in phase i are in $(2^{-i+1}, 2^{-i}]$ at most 2^{j-i} s -runs of type j can be created, causing an additional weight increase of $\leq \sum_{j>i} 2^{-2j} \cdot 2^{j-i} = 2^{-2i}$.

Let w_i be the total weight of the list of bins at the beginning of phase i . Then $w_{i+1} \leq 2w_i + \frac{7}{3}$ and $w_0 = 1$. From this, we obtain $w_i = O(2^i)$. Since in the i th phase we are only concerned with the $(i+1)$ -projection of the list of bins, each s -run has weight at least 2^{-2i} , and there are at most $2 \cdot 2^{2i} w_i$

