March 1988 Report No. STAN-CS-88-1200

Parallel Approximation Algorithms for Bin Packing

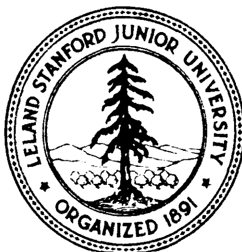
by

R. J. Anderson, E. W. Mayr, and M. K. Warmuth

Department of Computer Science

Stanford University

Stanford, California 94305



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

Manfred K. Warmuth
Department of Computer Science
University of California
Santa Cruz, California

Abstract

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 P-complete, and it is hence very unlikely that they can be efficiently parallelized. On the other hand, we exhibit an optimal 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.

[0]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.

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 $\in (0, 1)$, into a minimal number of unit capacity bins. For an instance I of the problem, $OPT(I)$ 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 remaining. It has been shown that the length of the packing generated by FFD is at most $\frac{11}{9}OPT(I) + 3$ [2][10]. The other approach for approximation algorithms is to look for algorithms with a performance bound of $(1 + \epsilon)OPT(I)$ [6][11]. 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 $(1 + \epsilon)OPT(I)$ algorithm due to [6].

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 multiprocessing 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 [7]. We consider a parallel algorithm to be fast if it is an NC algorithm [15], *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 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 ***P-complete algorithm.*** The proper interpretation of this is that deciding the value of a specified bit of the output of the algorithm is P-complete [1].

The main results of this paper are that the FFD heuristic is a p-complete algorithm, and that a packing that obeys the same performance bound as FFD can be computed by a fast parallel algorithm. The P-completeness result holds even if the problem is given with a unary encoding. This is interesting since most known p-complete number problems, such as Network Flow [8] and List Scheduling [9] can be solved by fast parallel algorithms if the numbers involved are small. A notable exception is Linear Programming which is also strongly ?-complete [5]. 0 ur algorithm for constructing a packing that obeys the same $\frac{11}{9}$ 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 $0(\log n)$ 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 $\epsilon > 0$. 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 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 P-complete in the strong sense, i.e., it is 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 [8][12] and list scheduling [9]. These are 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 P-complete to decide whether the FFD heuristic will pack the ith item into the bth 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 P-complete as can be seen by an easy log space reduction from the general monotone circuit value problem [13]. 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.

	bins	items
fan-out one:		
AND	$\delta_i, \delta_i+\delta_j-2\epsilon$	$\delta_i, \delta_i, \delta_i-2\epsilon, \delta_i-2\epsilon$
OR	$\delta_i+\delta_j-2\epsilon, \delta_i$	$\delta_i, \delta_i, \delta_i-2\epsilon, \delta_i-2\epsilon$
fan-out two:		
AND	$\delta_i, 2\delta_i-4\epsilon, \delta_i+\delta_j-3\epsilon, \delta_i+\delta_k-4\epsilon$	$\delta_i, \delta_i, \delta_i-2\epsilon, \delta_i-2\epsilon, \delta_i-2\epsilon, \delta_i-3\epsilon, \delta_i-4\epsilon$
OR	$2\delta_i-4\epsilon, \delta_i, \delta_i+\delta_j-3\epsilon, \delta_i+\delta_k-4\epsilon$	$\delta_i, \delta_i, \delta_i-2\epsilon, \delta_i-2\epsilon, \delta_i-2\epsilon, \delta_i-3\epsilon, \delta_i-4\epsilon$
gate β_n:	δ_n, δ_n	$\delta_n, \delta_n, \delta_n-2\epsilon, \delta_n-2\epsilon$

Table 1: Bins and item sizes for various types of gates

Let $\beta_1, \ldots, \beta_n$ be the gates of an n-gate monotone circuit, i.e., each β_i is either $\mathrm{AND}(i_1, i_2)$ or $\mathrm{OR}(i_1, i_2)$, 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 β_j, $j < i$. In our first construction, we transform the sequence $\beta_1, \ldots, \beta_n$ 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

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

The list segments for each gate are determined by Table 1 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.

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 $i = 1, \ldots, n$, 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 accomodate 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

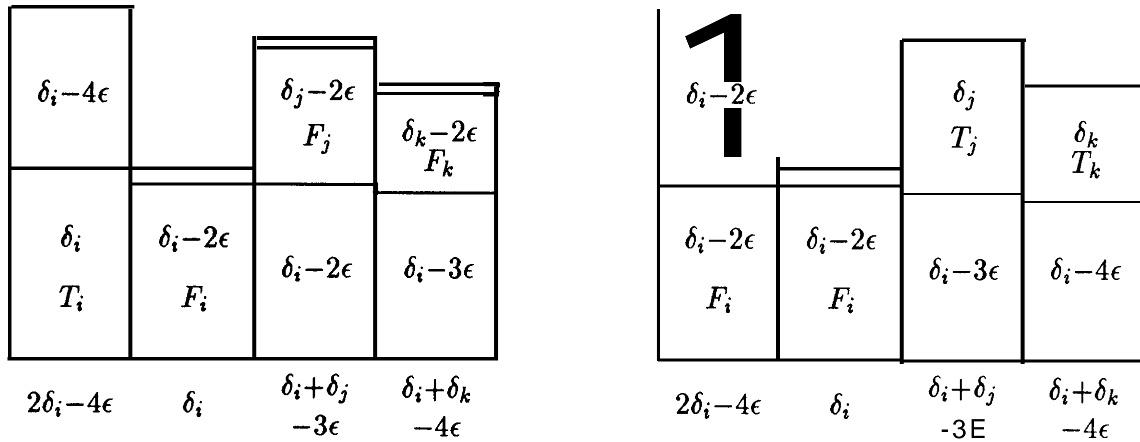


Figure 1: Packing for OR Gate with (a)one **true** input, (b) two **false** inputs

been evaluated, the two remaining of the first four items in β_j's (resp., β_k's) segment again reflect properly the values of the two inputs to the gate. Figure 2 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 1 are left to the reader.

In the second part of the construction, we show how to use unit size bins. Let $u_1, \ldots, u_q$ be the non-increasing list of item sizes, and let $b_1, \ldots, b_r$ 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 $v_1, \ldots, v_{2r}$ which when packed into r bins of size C leave space b_i in the ith 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 ith bin, thus leaving b_i empty space. Also note that v_{2r} is at least as large as u_1. Let $w_1, \ldots, w_{2r+q}$ 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 size 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 P-complete in the strong sense (under log space reductions), i.e., it remains 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 for a ***P-complete algorithm,*** a notion introduced in [1].

A number of other simple heuristics for bin packing can also be shown to be 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 left-over space.

3 A Parallel Alternative for FFD

Even though the FFD heuristic itself appears to be inherently sequential we are able to give an 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 NC-algorithm for instances where all items have size at least $\epsilon > 0$. 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 speed-up.

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***

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

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

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 $u_1, \ldots, u_r$ 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 [14]. We combine the set of items $\{u_k \mid \frac{i}{12} \leq s_k < \frac{i+1}{12}\}$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 $u_1, \ldots, u_q$ 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 $\frac{5}{6}$, $\lceil \frac{6U}{5} \rceil$ 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 ?-complete in the strong sense. Using a different approach to pack small items, however, still provides an asymptotically optimal NC-algorithm to achieve a packing with the same overall performance as FFD.

4 Parallel FFD for Big Items

Let $\epsilon > 0$ 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 $c_\epsilon \log n$ 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 $\lceil\log(\epsilon^{-1})\rceil$ 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 space 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 all bin sizes of the run are 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 ε oc
    S := (ρ0); co S holds a list of runs; the initial run ρ0 consists of n empty bins oc
    for (i := 0; 2^-i ≥ ε; i++) do
        L' := sublist of items in L with sizes ∈ (2^-(i+1), 2^-i];
        if L' = () then continue fi; co go to beginning of loop oc
        S' := ();
        repeat
            ρ := first run of the i + 1-projection of S;
            forward_pack(ρ, ψ); co ψ is a pre-run oc
            S'' = fill_in(ψ); co S" is a list of runs oc
            remove from runs in S" bins with less than ε 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 ≤ type of ρ then ψ := ρ; return fi;
    let L' = u1, ..., uℓ;
    let s1 ≤ s2 ≤ ... ≤ s|ρ| be the amounts of space available in ρ's bins;
    k := max{j | j ≤ |L'| and u1 + ... + uj ≤ s1}; co note that k > 0 oc
```

let r be minimal subject to

1. $r = \min\{|\rho|, \lceil \ell/k \rceil\}$; **or**
2. $(r+1)k < \ell$ **and** $u_{rk+1} + \ldots + u_{(r+1)k+1} \leq s_{r+1}$;

remove first r bins from p, put them into ψ;
if $\rho = ()$ then remove ρ from L' *fi;*
in parallel, add items $u_{(j-1)k+1}, \ldots, u_{jk}$ to jth 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 pl I ase 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 bin 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 a 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 items) 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 ith phase we are only concerned with the i + l-projection of the list of bins, each s-run has weight at least 2^{-2i}, and there are at most $2 \cdot 2^{2i}w_i$ runs for the algorithm to pack into. The number of runs in the last phase is therefore $O(1/\epsilon^3)$. Since the time requirement of the algorithm is clearly 0(log n) for every run generated, the claim follows. □

5 Packing Fill-in Items

In this section, we present asymptotically optimal EREW-PRAM algorithms for the following two problems:

1. merge two sorted lists of n elements each into a sorted list;

2. in a string of length n of opening and closing parentheses, find the matching pairs. This problem can also be phrased in terms of push and pop operations on a stack, with the goal to match pop's to pushes.

Since both problems can be solved sequentially in linear time, any optimal parallel algorithm must run in time O(log n) on an EREW-PRAM with $n/\log n$ processors. We first

describe the merge procedure. Note that for the fill-in packing we also require that the merging is done in such a way that all elements of a given value in the first sequence precede all elements of the same value from the second sequence. However, this can easily be taken care of, and we leave the corresponding details to the reader. For simplicity, we assume here that no element in the first sequence has the same value as an element in the second sequence.

From the two input sequences, we first select every $\lceil \log n \rceil$ th element, and merge the two selected subsequences. Viewing the first subsequence in increasing and the second in decreasing order results in a bitonic sequence. It can be easily sorted in $O(\log n)$ steps on $n/\log n$ processors by emulating the last stage of Batcher's bitonic sort [4][16]. Let $u^{(1)}, u^{(2)}, \ldots$ be the elements selected from the first sequence, and $v^{(1)}, v^{(2)}, \ldots$ those from the second. Also, let $U^{(i)}$ be the subsequence of the first sequence between $u^{(i)}$ and $u^{(i+1)}$, and let $V^{(i)}$ be defined accordingly for the second sequence.

Assume first that two or more selected elements $v^{(j)}, \ldots, v^{(k)}$ of the second sequence fall within $U^{(i)}$. We broadcast the elements in $U^{(i)}$ that are greater than $v^{(j)}$ and less than $v^{(k)}$ to $v^{(j)}, \ldots, v^{(k-1)}$. To do so, we assign one processor to each of $v^{(j)}, \ldots, v^{(k-1)}$, and use these processors to implement a balanced binary tree in such a way that each processor is responsible for at most two nodes (one leaf and possibly one internal node) in that tree. The elements in $U^{(i)}$ can be broadcast, along this tree, in a pipelined fashion, requiring $O(\log n)$ time. We then merge each $V^{(l)}$, for $l = j, \ldots, k-1$, with the sublist of $U^{(i)}$ between $v^{(l)}$ and $v^{(l+1)}$, using the processor responsible for $v^{(l)}$. All $U^{(i)}$ of this type are handled in this manner in parallel.

For- the second phase of the merge procedure, let $V^{(i)}$ be an interval unaffected above, and let j and k be maximal such that $u^{(j)} < v^{(i)}$ and $u^{(k)} < v^{(i+1)}$. The elements in $V^{(i)}$ are broadcast, as above, to $u^{(j)}, \ldots, u^{(k)}$, and the appropriate sublists are then merged with $U^{(j)}, \ldots, U^{(k)}$. Again, this can be achieved in $O(\log n)$ time using one processor per selected element. Since one $U^{(j)}$ may be affected by two adjacent $V^{(i)}$'s, we divide this second phase into two subphases, merging in each subphase only every other of the relevant $V^{(i)}$.

Together, we have just established

Theorem 4 ***Two sorted lists of length n each can be merged on an EREW-PRAM with $n/\log n$ processors in time $O(\log n)$. This result is asymptotically optimal.***

Proof: □

The second problem considered in this section concerns simulating a pushdown stack or matching parentheses. We use the second picture. Let an arbitrary string of n opening and closing parentheses be given. First, we employ an optimal parallel prefix routine to find and remove all those (opening or closing) parentheses that are not matched. For the remaining parentheses, we use parallel prefix once more to assign a level to each parenthesis, in the standard manner. The first (opening) parenthesis is assumed to be assigned level 1. The problem now becomes finding, for each opening parenthesis, the first closing parenthesis following it in the string and having the same level.

Imagine $n/\log n$ processors of an EREW-PRAM arranged in form of a balanced binary tree, with each leaf processor responsible for an interval of roughly $2 \log n$ parentheses. For

convenience we refer to the nodes of the tree by their inorder number, and we assume that every processor knows the inorder number of its node. First, the leaf processors find all matching pairs of parentheses within their respective interval. The unmatched parentheses at every leaf form a subsequence of closing parentheses followed by a subsequence of opening parentheses. Next, each processor in the tree, from the leaves towards the root, computes a triple (c, m, o). Here, m is the number of matching pairs, with the opening parenthesis in the left and the closing parenthesis in the right subtree of the node assigned to the processor; c and o are the number of unmatched closing respectively opening parentheses in the subtree rooted at the node. Each processor at an internal node of the tree can compute its triple from those of its two children as follows:

$$(c, m, o) = (lc + \max(0, rc - lo), |lo - rc|, ro + \max(0, lo - rc)),$$

where lc and lo are the c- and o-value of the left child, and rc and ro corresponingly for the right child. This computation proceeds level by level, and takes $O(\log n)$ time. Using an optimal routine for parallel prefix computation, we also compute $b(v) = \sum_{w<v} m(w)$ for every node v in the tree.

Every pair of matching parentheses can now be assigned a uniquely determined index (b, i). Consider a pair matched at node v. Then $b = b(v)$, and i gives the nesting depth of the pair in the subsequence of pairs matched at v. Thus, the outermost pair of parentheses being matched at v has index $(b(v), 0)$, the innermost $(b(v), m(v) - 1)$ where $m(v)$ is the m-value in v's triple computed above. Originally, the index of a matching pair of parentheses is known at the node in the tree where the pair matches.

The goal of the next stage of the algorithm is to communicate its index to every parenthesis in the string that is left after the preprocessing. Consider node v in the tree. It matches an interval of $m(v)$ opening parentheses which it received from its left child with an interval of $m(v)$ closing parentheses received from its right child. The processor at v sends the indices describing the endpoints of each part to the corresponding child, together with a parameter describing the position of the interval in the sequence of parentheses originally passed up from that child. Upon receiving this information from its parent, the processor at a (non-leaf) descendant node can break the corresponding interval into two intervals, one that came from its left child, and one from its right child, and send the appropriate information on to its children. Leaf processors distribute index interval information to the corresponding parentheses in their subinterval. With some care in the implementation, each leaf processor requires only $O(\log n)$ time. Therefore, if all processors start out simultaneously to propagate the index information for the intervals of parentheses they match, the whole stage obviously takes time $O(\log n)$.

Finally, all opening parentheses in parallel write their address to position $b + i$ of some global array of length n, where (b, i) is the index received by the parenthesis. In the following step, all closing parentheses can read the cell of the array given in the same manner by their index, and in this way find their matching opening parenthesis. Since all sums $b + i$ are distinct, no write or read conflicts will occur.

***Theorem 5** All matching pairs in an arbitrary string of n parentheses can be found in time $O(\log n)$ on an n processor EREW-PRAM.* □

We remark that a (completely different) CREW-PRAM algorithm for this problem obeying the same asymptotic resource bounds has been given in [3].

6 Parallel Approximation by D iscret izat ion

It is natural to ask if it is possible to do better than FFD -with a parallel approximation algorithm for bin packing. The answer is yes, since it is possible to implement the algorithm in [6] as a fast parallel algorithm. This algorithm constructs a packing that is within a factor of $1 + \epsilon$ of the optimum for any fixed ϵ in $O(n)$ time. The run time for the algorithm is enormous, having a constant term which is exponential in $\frac{1}{\epsilon}$.

The basic idea of the algorithm in [6] is to first consider a packing problem where the number of item sizes is fixed and the size of the smallest item is bounded below by a constant. They show that such a packing problem can be solved to within an additive constant in ***constant'*** time. The algorithm reduces the packing problem to the restricted version by dividing elements into a number of groups and then rounding the size of the elements in a group up to the same value. They also show that this packing gives a good approximation to the original packing problem. There are no obstacles to implementing this as an $\mathcal{NC}$-algorithm, using, among other things, some of the techniques presented in section 3. Further details involved in the construction of the packing are left to the reader.

7 Conclusion

We have seen that some very simple sequential bin packing heuristics are P-complete, and hence in all likelihood not efficiently parallelizable. With FFD, we have established one of the first number problems (other than LP) known to be P-complete in the strong sense. Interestingly enough, however, we have also been able to present an NC-algorithm that can be viewed as a parallel approximation scheme for FFD.

While there exist polynomial time and $\mathcal{NC}$ approximation schemes for the $\mathcal{NP}$-complete problem of bin packing, the constants involved in these algorithms are prohibitively large. An interesting open problem is whether more efficient sequential approximation schemes can be parallelized. More generally, one might ask whether there are natural *parallel* approximation schemes for bin packing, i.e., schemes not derived from sequential ones.

Another interesting question is to study the application of parallel approximation techniques to scheduling problems, some of which are very closely related to bin packing. For instance, there are many sequential heuristics based on list schedules, the parallel complexity of these methods, however, is largely unknown.

'The number of arithmetic operations in this algorithm is constant. The numbers involved are not large, so the number of bit operations used is polynomial in n. Using parallel algorithms for the arithmetic operations, this can safely be considered an $O(\log n)$ time parallel algorithm.

References

[1] R. Anderson and E. Mayr. Parallelism and greedy algorithms. In F. P. Preparata, editor, ***Advances in Computing Research; Parallel and Distributed Computing,*** pages 17-38, JAI Press, 1987.

[2] B. Baker. A new proof for the first-fit decreasing bin-packing algorithm. ***Journal of Algorithms,*** 6(1):49–70, 1985.

[3] I. Bar-On and U. Vishkin. Optimal parallel generation of a computation tree form. ***ACM Transactions on Programming Languages and Systems,*** 7(2):348–357, ***1985.***

[4] K. Batcher. Sorting networks and their applications. In ***Proceedings of AFIPS Spring Joint Comp. Conf.,*** pages 307-314, 1968.

[5] D. Dobkin, R. Lipton, and S. Reiss. Linear programming is log-space hard for $\mathcal{P}$. ***Information Processing Letters,*** 8(2):96–97, ***1979.***

[6] W. Fernandez de la Vega and G. Lueker. Bin packing can be solved within $1 + \epsilon$ in linear time. *Combinatorica*, 1(4):349–355, 1981.

[7] S. Fortune and J. Wyllie. Parallelism in random access machines. ***In Proceedings of the 10th Ann. ACM Symposium on Theory of Computing (San Diego, CA),*** pages 114–118, ACM, 1978.

[8] L. Goldschlager, R. Shaw, and J. Staples. The maximum flow problem is log space complete for P. ***Theoretical Computer Science,*** 21(1):105–111, 1982.

[9] D. Helmbold and E. Mayr. Fast scheduling algorithms on parallel computers. In F. P. Preparata, editor, ***Advances in Computing Research; Parallel and Distributed Computing,*** pages 39-68, JAI Press, 1987.

[10] D. Johnson, A. Demers, J. Ullman, M. Garey, and R. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. ***SIAM J. on Comput.,*** 3:299–326, 1974.

[11] N. Karmarkar and R. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In ***Proceedings of the 23rd Ann. IEEE Symposium on Foundations of Computer Science (Chicago, IL),*** pages 312-320, IEEE, 1982.

[12] R. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random $\mathcal{NC}$. In ***Proceedings of the 17th Ann. ACM Symposium on Theory of Computing (Providence, RI),*** pages 22-32, ACM, 1985.

[13] R. Ladner. The circuit value problem is log-space complete for $\mathcal{P}$. ***SIGACT News,*** 7(1):583–590, 1975.

[14] R. Ladner and M. Fischer. Parallel prefix computation. *J.ACM*, 27(4):831–838, 1980.

[15] N. Pippenger. On simultaneous resource bounds. In ***Proceedings of the 20th Ann. IEEE Symposium on Foundations of Computer Science (San Juan, PR),*** pages 307–311, IEEE, 1979.

[16] H. Stone. Parallel processing with the perfect shuffle. ***IEEE Trans. on Computers,*** C-20(2):163-271, 1971.