PROBABILISTIC ANALYSIS
AND
PARALLELIZATION OF
BIN-PACKING ALGORITHMS

BY

CHANG EE CHIEN, B.Sc. (Hons), N.U.S.

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

INSTITUTE OF SYSTEMS SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

1993

Acknowledgments

I would like to thank my supervisors Dr Wang Weiguo and Dr Mohan S. Kankanhalli for their guidance, encouragement and support. Appreciation also goes to friends who show interest in this thesis and help me in one way or another.

Ee-Chien

April 1993

Abstract

Bin-packing problems appear in many applications. Since the general form of bin-packing problems is NP-hard, emphasis is placed on the study of heuristic algorithms. To compare the performance of heuristic algorithms, their average-case performance is an appropriate measure when the input of these algorithms are drawn from some known distributions.

In this thesis, we propose a heuristic algorithm, `HashPacking`. Based on this algorithm, we formulate algorithms for various settings and study their average-case performance. Furthermore, we give an expected work optimal parallel algorithm based on `HashPacking` for one-dimensional bin-packing. To obtain this optimal parallel algorithm, we develop a randomized version of Brent's Scheduling Principle, which under certain conditions, is able to transform a parallel algorithm to an expected work optimal randomized parallel algorithm.

Contents

Chapter 1

Introduction

1.1 Background

1.1.1 Motivation

Bin-packing problems arise in applications that require to pack or fit a collection of items into well-defined regions in a way that they do not overlap, and resource usage is minimized. For example, in stock-cutting problems, a collection of pieces is required to be cut from standardized stock in a way so as to minimize the wasted material. In packing problems, a collection of objects are required to be packed into standardized containers so as to minimize wasted space. By taking time as an additional dimension, the scheduling problem, where a collection of jobs is to be allocated to a set of machines so as to minimize the machines' idle time, can also be treated as a bin-packing problem.

In some cases, items have to be scheduled/packed/cut once they are received. That is, the decision regarding an item should be taken before the information on all the items is available. These problems can be formulated as the on-line bin-packing problem.

As most of the bin-packing problems are NP-complete, emphasis is placed on the study of heuristic algorithms. There are a few ways to measure the performance of a heuristic algorithm. If the input is drawn from a known distribution, then its average-case performance is an appropriate measure. In this thesis, we propose a heuristic algorithm for bin-packing problems and analyze its average-case performance. This algorithm is modified for different settings. Furthermore, we propose a parallelization of this algorithm.

1.1.2 Problem Definition

We consider *multi-dimensional bin-packing problem*, also known as *box-packing*[3] and *rectangle-packing*[8]. Given a list of d-dimensional items, the problem is to pack these items into minimum number of standard bins. A d-dimensional item $x = (x^1, x^2, .., x^d)$ where $x^1, x^2, .., x^d \in (0, 1]$, is the set $(0, x^1] \times (0, x^2] \times ... \times (0, x^d] \subset [0, 1]^d$. The bin, B is the item $(1, 1, .., 1)$. We define the operator $\oplus$ on $R \subset [0, 1]^d$ and $t \in [0, 1]^d$ as follows: $R \oplus t = \{y |$there exists a $y' \in R$ such that $y = y' + t$, where $+$ is the vector addition $\}$. In other words, operator $\oplus$ translates the set R by amount t. Given a list of n items $\langle R_1, R_2, .., R_n \rangle$, we say that these items can be packed in the bin B, if there exist $t_1, t_2, .., t_n \in [0, 1]^d$ such that

$$\bigcup_{j=1}^{n} R_j \oplus t_j \quad \subset \quad B, \text{ and} \tag{1.1}$$

$$(R_k \oplus t_k) \cap (R_i \oplus t_i) \quad = \quad \phi, \text{ for all } i, k \quad \text{such that } k \neq i \tag{1.2}$$

That is, the items are allowed to translate in the bin under the constraint that all items must be "contained" in the bin (equation (1.1)) and no "overlapping" of items is allowed in the bin (equation (1.2)).

In the two-dimensional bin-packing, items are rectangles and the bin is the unit square. A set of rectangles can be packed in a bin if, after proper translation, they can be contained in the square without overlapping. Under conditions (1.1) and (1.2), rotation of an item during packing is not allowed. For example, the item (1,0.5) cannot be packed with the item $(0.5, 1)$. This restriction is also known as the orthogonality restriction[1].

For a d-dimensional item, $(x^1, .., x^d)$, define its *volume* as the quantity $\prod_{j=1}^{d} x^j$.

By definition, the volume of a bin is $\prod_{j=1}^{d} 1 = 1$. Suppose that n items with volumes $v_1, .., v_n$ are packed in a bin, then the *wasted space* in this bin is

$$1 - \sum_{i=1}^{n} v_i.$$

If m bins are needed to pack n items with volume $v_1, .., v_n$, then we say that the wasted space of this packing is

$$m - \sum_{i=1}^{n} v_i.$$

Let l_n be a list of items $\langle x_1, x_2, ..., x_n \rangle$. Suppose A is an algorithm which on input l_n, produces a packing for the items. Let $A[l_n]$ be the number of bins used by A to pack the items in l_n. Let $OPT[l_n]$ be the minimum possible number of bins required to pack the items in l_n. Write $A^W[l_n]$ as the wasted space of the packing output by A on input l_n. Similarly, write $OPT^W[l_n]$ as the minimum possible wasted space produced in order to pack items in l_n.

Note that

$$\begin{aligned} A^W[l_n] &= A[l_n] - \sum_{i=1}^{n} v_i, \quad \text{and} && (1.3) \\ OPT^W[l_n] &= OPT[l_n] - \sum_{i=1}^{n} v_i, && (1.4) \end{aligned}$$

where v_i is the volume of the item x_i.

When the input is generated from a known distribution, the average case performance is a better measure of the efficiency of an algorithm. Let $X_i = (X_i^1, X_i^2, .., X_i^d)$ be the random variable for an item. The random variable for a list of n items is $L_n = \langle X_1, X_2, .., X_n \rangle$. Unless otherwise stated, we assume that all the X_i^j are independent and identically distributed random variables with uniform distribution, that is

$$X_i^j \sim U[0,1], \text{ for all possible } i, j.$$

We write $E[A[L_n]]$ as the expected number of bins used by algorithm A over L_n, and $E[OPT[L_n]]$ as the expected number of bins used in the optimal solution over input L_n. Similarly, we write $E[A^W[L_n]]$ and $E[OPT^W[L_n]]$ as the expected wasted space produced by algorithm A and the minimum possible wasted space respectively. From (1.3) and (1.4), we have

$$E[A^W[L_n]] = E[A[L_n]] - E[\sum_{i=1}^{n} \prod_{j=1}^{d} X_i^j], \quad \text{and} \tag{1.5}$$

$$E[OPT^W[L_n]] = E[OPT[L_n]] - E[\sum_{i=1}^{n} \prod_{j=1}^{d} X_i^j]. \tag{1.6}$$

We will describe certain quantities using the O, Ω and θ notation. Suppose $f(n)$ and $g(n)$ are two functions of interest. If there exists positive constants C_1 and C_2 such that

$$C_1 f(n) \leq g(n) \leq C_2 f(n)$$

with finitely many exceptions, then we write $g(n) = \theta(f(n))$ and say that $g(n)$ is of the order $\theta(f(n))$. If only the left inequality holds, that is

$$C_1 f(n) \leq g(n)$$

with finitely many exceptions, then we write $g(n) = \Omega(f(n))$. Similarly, if only the right inequality holds, then we write $g(n) = O(f(n))$.

To prove bounds on expectation, we often use the "high probability" argument. To pack n items, at most n bins are needed. If an event E has probability more than $1 - \frac{1}{n}$ of occurrence, and the expected number of bins used by an algorithm A to pack n items given that E occurred is $O(f(n))$, then, the expected number of bins used by A is

$$\left(1 - \frac{1}{n}\right) O\left(f(n)\right) + O(\frac{1}{n})n = O\left(f(n)\right).$$

In other words, if an event E has a high probability of occurrence (that is, $Pr(E) > 1 - \frac{1}{n}$), then we can assume that E has always occurred when the order of the expectation is of interest.

Since $X^1, X^2, .., X^d$ are independent and uniformly distributed, the expected volume of a d-dimensional item is $E[\prod_{j=1}^{d} X^j] = 2^{-d}$. The expected volume of n items is $n2^{-d}$. From (1.5),

$$E[A^W[L_n]] = E[A[L_n]] - \frac{n}{2^d}.$$

In addition, the algorithms described in the rest of the thesis will pack at most 2^d items in any single bin for d-dimensional bin-packing. Let n' be the number of bins containing strictly less than 2^d items in a packing produced by an algorithm A on input l_n. The total number of items in these bins is at most $n'(2^d-1)$ and at least n'. Since a single bin contains at most 2^d items, the total number of bins used, $A[l_n]$, is bounded as follow:

$$n' + \frac{n-(2^d-1)n'}{2^d} < A[l_n] < n' + \frac{n-n'}{2^d}.$$

Together with (1.5), we have the following bounds on the expected wasted space:

$$E[n'(\frac{1}{2^d})] \leq E[A^W[L_n]] \leq E[n'\left(1-\frac{1}{2^d}\right)]. \tag{1.7}$$

In other words, $E[n']$ is of the same order as $E[A^W[L_n]]$. Thus, we can estimate the expected wasted space by finding $E[n']$.

Algorithms for bin-packing can be divided into two classes, on-line and off-line. An on-line algorithm considers the items one at a time and each item considered must be packed before the next item is seen. Furthermore, once packed, the items cannot be moved, even within its bin. Off-line algorithms need not satisfy this constraint. That is, they have the access to all items before any decision is made.

There are two types of on-line algorithms, *closed-end* and *open-end*. For the closed-end on-line algorithms, the number of items to be packed is known before packing. For the open-end on-line algorithms, the number of items to be packed is not known before packing.

In average-case performance analysis of open-end on-line algorithms, since the number of items, N, is not known before packing, we assume that N follows a known distribution. We follow the model of open-end on-line bin-packing used in [13] where the inputs are drawn as follows:

For a given n,

1. Choose k from 1 to n at random.

2. Choose k items uniformly from $(0,1]^d$.
3. Feed these items to the algorithm one-by-one followed by a "stop" signal after the k-*th* item.

1.1.3 Related Work

The one-dimensional and multi-dimensional bin-packing problems are *NP-Hard* [6]. A lot of work has been done in approximation algorithms [1, 6, 7]. Two classical approximation algorithms are *Best Fit* (`BF`) and *First Fit* (`FF`). The `FF` algorithm works as follows: Suppose items $x_1, x_2, .., x_m$ are already packed in bins $b_1, b_2, .., b_{m'}$. The item x_{m+1} is then packed into the bin with least index into which x_{m+1} can be packed. If no such bin is available, then x_{m+1} is packed into a new bin $b_{m'+1}$. The algorithm `BF` differs from `FF` in the choice of the bin x_{m+1} is packed into. Instead of choosing the bin with the least index, `BF` packs x_{m+1} into the bin which has minimum wasted space and into which x_{m+1} can be packed. A version of `BF`, *Best Fit Decreasing* (`BFD`), sorts the items in the order of decreasing volume before the start of packing.

For the one-dimensional bin-packing, it is known that the expected minimum wasted space is bounded below by $\Omega(\sqrt{n})$. That is,

$$E[OPT^W[L_n]] = \Omega(\sqrt{n}).$$

It is known that `BFD` achieves $\theta(\sqrt{n})$ wasted space for one-dimensional bin-packing [8, 10]. In proving this bound, an algorithm `Match` is introduced. A description of `Match` is given in Appendix. For the d-dimensional bin-packing, Karp et. al. [8] give an algorithm which achieves expected wasted space of order $O(G(d,n))$ where

$$\begin{aligned} G(1,n) &= \sqrt{n}, \\ G(2,n) &= \sqrt{n}\log n, \text{ and} \\ G(d,n) &= n^{\frac{d-1}{d}} \quad \text{for } d \geq 3. \end{aligned}$$

For one dimensional open-end on-line bin-packing, it has been shown in [13] that the expected wasted space of all algorithms are bounded below by $\Omega(\sqrt{n \log n})$. There is an algorithm which achieves $\theta(n \log n)$ expected wasted space [14]. Furthermore, the algorithm `BF` achieves $\theta(\sqrt{n}\log^{\frac{3}{4}} n)$ expected wasted space and expected wasted space of the algorithm `FF` is bounded below by $\Omega(n^{\frac{2}{3}})$ and bounded above by $O((n \log n)^{\frac{2}{3}})$ [13]. For the one dimensional closed-end on-line bin-packing, `FF` can be modified to an algorithm which produces $\theta(\sqrt{n})$ expected wasted space [13].

1.2 An Overview of the Thesis

In this thesis, we propose a simple heuristic algorithm, `HashPacking`, for one-dimensional bin-packing. This algorithm is modified or generalized for different settings. We study the average-case performance of these algorithm and give a parallel algorithm for the one-dimensional setting.

In chapter 2, we describe `HashPacking`. Under the on-line setting, the expected wasted space is $O(n^{2/3})$. This algorithm is later modified to an off-line, linear time algorithm `ItHashPacking` which produces expected wasted space of order $O(\sqrt{n})$. A generalization, $\texttt{HashPacking}_d$, of `HashPacking` for d-dimensional on-line bin-packing is discussed in section 2.2. The expected wasted spaced produced by this algorithm is $O(n^{\frac{d+1}{d+2}})$.

In chapter 3, we give two parallelizations of of `ItHashPacking`. The first algorithm `PItHashPacking` uses $O(n)$ processors, $O(n^3)$ space and runs in $O(\log n)$ time. The second algorithm `PSortItHashPacking` incorporates an integer sorting algorithm. It uses $O(n)$ processors, $O(n)$ space and runs in expected $O(\log n)$ time. The expected work done of both algorithm are $O(n \log n)$. We propose a randomized version of Brent's scheduling principle which can be applied to the second algorithm to obtain an expected work optimal (expected work done of order $O(n)$) algorithm.

Figure 1.1 summarizes the organization of this thesis.

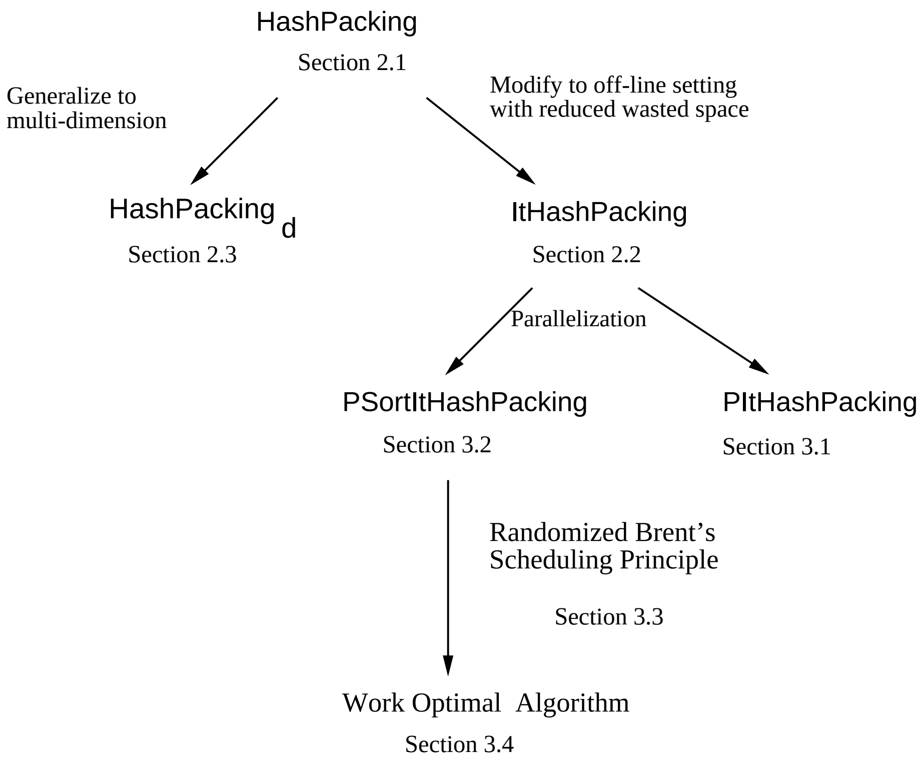

Figure 1.1: Overview

Chapter 2

Hash Packing Algorithm

2.1 One-dimensional Hash Packing

In this section we describe, and present average case analysis of two algorithms, $\texttt{HashPacking}_m$ and $\texttt{ItHashPacking}_m$, where m is a parameter.

The first algorithm, $\texttt{HashPacking}_m$ can be modified to an open-end on-line algorithm which produces an expected wasted space of $O(n^{\frac{2}{3}})$ when m is chosen to be $n^{\frac{1}{3}}$, that is, $E[\ \texttt{HashPacking}^{W}_{n^{\frac{1}{3}}}[L_n]] = O(n^{\frac{2}{3}})$. $\texttt{ItHashPacking}_m$ is a modification of $\texttt{HashPacking}_m$ with expected wasted space of $O(\sqrt{n})$, when m is chosen to be n^3. However, $\texttt{ItHashPacking}_m$ is not an on-line algorithm.

2.1.1 $\texttt{HashPacking}_m$

The idea of $\texttt{HashPacking}_m$ is to match as many pairs of items as possible and pack each pair in a bin. At the same time, it ensures that the wasted space in every bin will not exceed the given constant parameter.

Given input $l_n = \langle x_1, x_2, ..., x_n\rangle$. we classify the items into two types, big $(+1)$ and small (-1) as follows:

$$c_i = \begin{cases} +1, & x_i > \frac{1}{2}, \\ -1, & otherwise. \end{cases}$$

Define

$$e_i = \begin{cases} 1 - x_i, & x_i > \frac{1}{2}, \\ x_i, & otherwise. \end{cases}$$

Note that if x_i is a big item, then e_i indicates the volume of the largest item that can be packed with x_i in a bin.

Next, define a function, h, as

$$h(x_i) = \begin{cases} \lfloor 2e_i m \rfloor, & \text{if } c_i = +1, \\ \lceil 2e_i m \rceil, & otherwise. \end{cases}$$

The function h can be viewed as a hashing function whose range is $\{0, 1, 2, ..., m\}$. If x_1 is a big item, x_2 a small item and $h(x_1) = h(x_2)$, then both x_1 and x_2 can be packed in a bin with wasted space smaller than m^{-1}.

The algorithm $\texttt{HashPacking}_m$ is as follows:

Algorithm: $\texttt{HashPacking}_m$

---

1. Create $m + 1$ buckets labeled as $b_0, b_1, .., b_m$.
2. For each item x_i, store it in the bucket $b_{h(x_i)}$.
3. For each bucket, repeat packing two items of the different types in the bucket into a single bin, until no such pair can be found.
4. For the remaining items, pack each of them into a single bin.

If two items are packed together by $\texttt{HashPacking}_m$, we say that these two items are *matched*. If an item are not packed with other items, then we say that this item is *unmatched*.

2.1.2 Average-case analysis

We will first find a relation between the parameter m and the performance (that is, the expected number of bins used) of $\texttt{HashPacking}_m$. Then the number of bins used is minimized by choosing a suitable m.

Lemma 2.1 *For any $\alpha < 1$, $m = n^\alpha$, when n is large enough,*

$$E[\texttt{HashPacking}_m[L_n]] = \frac{n}{2} + \frac{n}{2m} + \theta(\sqrt{nm}).$$

Proof: Note that only big items will be hashed to the first bucket, b_0 and only small items will be hashed to the last bucket, b_m. Thus, all the items hashed to these two buckets will not be matched. We call b_0 and b_m *boundary buckets*, and the others *non-boundary buckets.*

Since the probability of an item being hashed to one of the two boundary buckets is $2(\frac{1}{2m}) = m^{-1}$, therefore, the expected number of items hashed to these buckets is nm^{-1}. Thus the expected number of bins used to packed these items is nm^{-1}.

Consider a non-boundary bucket, b_i. Let N_i be the random variable of the number of items hashed to it. The process of hashing n items can be viewed as a sequence of n Bernoulli trials with probability m^{-1} of success (being hashed to b_i). Thus, the expected number of items hashed to b_j is nm^{-1}. Furthermore, by Lemma A.1, together with the assumption that $m = n^\alpha$, where $\alpha < 1$, the probability that more than $2nm^{-1}$ items are hashed to b_j is

$$Pr(N_i > \frac{2n}{m}) < \left(\frac{e}{4}\right)^{\frac{n}{m}}.$$

By *Boole's inequality* (for any finite or countably infinite sequence of events $A_1, A_2, \ldots$, $Pr(A_1 \cup A_2 \cup \ldots) \leq Pr(A_1) + Pr(A_2) + \ldots$), the probability that at least one bucket has more than $2nm^{-1}$ items hashed to it is less than $m\left(\frac{1}{4}e\right)^{nm^{-1}}$ which is smaller than n^{-1} for large enough n and when $m = n^\alpha, \alpha < 1$. In other words, with high probability, every bucket has at most $2nm^{-1}$ items been hashed to it. Similarly, we can show that with high probability, every buckets has at least $n(2m)^{-1}$ items been hashed to it.

Let $n_j, j = 0, 1, .., m$ be the number of item in bucket b_j. By the high probability argument (1.1.2), we can assume that $n(2m)^{-1} < n_j < 2nm^{-1}$, that is, $n_j = \theta(nm^{-1})$. Let $C_1^j, C_2^j, ..., C_{n_j}^j$ be the random variables of the type of the items in the bucket b_j. It can be shown that these are independently and identical distributed random variables with each C_i^j having the same probability of being $(+1)$, or (-1). Let

$$C^j = \text{number of big items in the bucket } b_j.$$

Then $C^j \sim B(n_j, \frac{1}{2})$. The expected absolute difference between the number of big items and number of small items is $\sqrt{n_j}$. Therefore, the expected number of bins required to pack the items in b_j is $\frac{1}{2}(n_j - \sqrt{n_j}) + \frac{1}{2}\sqrt{n_j} = n_j + \frac{1}{2}\sqrt{n_j}$. Since $n_j = \theta(nm^{-1})$ with high probability, thus

$$\begin{aligned} E[\texttt{HashPacking}_m[L_n]] &= \frac{n}{m} + \sum_{j=1}^{m-1} \left(\frac{n_j}{2} + \theta\left(\sqrt{n_j}\right) \right) \\ &= \frac{n}{2m} + \frac{n}{2} + \theta(\sqrt{nm}). \end{aligned}$$

□

Lemma 2.1 indicates that the value of the parameter m affects the performance of the algorithm. The larger the value of m, the smaller the number of items which will be hashed to the boundary buckets. But then the number of bins containing only one item in the non-boundary buckets will be increased as it is more difficult to satisfy the requirement on wasted space. To minimize the number of bins used, choose m to be $n^{\frac{1}{3}}$. Together with (1.1.5), we have:

Theorem 2.2

$$E[\texttt{HashPacking}^W_{n^{\frac{1}{3}}}[L_n]] = \theta(n^{\frac{2}{3}}).$$

□

2.1.3 On-line setting

Note that the algorithm $\texttt{HashPacking}_m$ can be implemented as an on-line algorithm as follows:

---

1. Create $m+1$ lists $b_0, b_1, .., b_m$ of bins. These lists are initially empty. (Note that in HashPacking, $b_0, .., b_m$ are treated as buckets. In this algorithm, they are treated as lists of bins.)
2. For each incoming item x_i:
   (a) Compute its type and $h(x_i)$.
   (b) If there is a bin in the list $b_{h(x_i)}$ such that it contains an item of the different type from x_i, then pack x_i into it and remove it from $b_{h(x_i)}$. Otherwise, pack x_i into a new bin and append this bin to $b_{h(x_i)}$.

Recall that there are two types of on-line algorithms, open-end and closed-end. For closed-end on-line algorithms, the number of items to be packed is known before packing, whereas for open-end on-line algorithms, the number of items to be packed is not known before packing.

Clearly, the expected wasted space produced by $\texttt{HashPacking}_{n^{\frac{1}{3}}}$ as a closed-end on-line algorithm is same as the result proven in the previous section, that is, $\theta(n^{\frac{2}{3}})$.

Consider the open-end on-line bin-packing. If the number of items to be packed is k, then the expected wasted space produced by $\texttt{HashPacking}_m$ in the open-end setting is $E[\texttt{HashPacking}_m^W[L_k]]$. The expected wasted space produced by $\texttt{HashPacking}_m$ in the open-end setting is:

$$\sum_{k=1}^{n} Pr(N=k)E[\texttt{HashPacking}_m^W[L_k]], \tag{2.1}$$

where N is the random variable for the number of items to be packed, and n is the maximum number of items to be packed. During the packing, for each incoming item, the probability that the addition of a new item will reduce the number of bins containing one item by one is less than $\frac{1}{2}$. On the other hand, the probability that it will increase the number by one is more than $\frac{1}{2}$. Thus $E[\texttt{HashPacking}_m^W[L_k]] < E[\texttt{HashPacking}_m^W[L_{k+1}]]$. Therefore, (2.1) is bounded by $E[\texttt{HashPacking}_m^W[L_n]]$. Choosing $m = n^{\frac{1}{3}}$, (2.1) is bounded by $O(n^{\frac{2}{3}})$. In particular, if the number of items to be packed is chosen uniformly from 1 to n, then the expected wasted space is:

$$\begin{aligned}\sum_{k=1}^{n} Pr(N = k)E[\texttt{HashPacking}_{n^{\frac{1}{3}}}^W[L_k]] &= \sum_{k=1}^{n} \frac{1}{n}(\frac{k}{2n^{\frac{1}{3}}} + \theta(n^{\frac{2}{3}})) \\ &= \theta(n^{\frac{2}{3}}).\end{aligned}$$

2.2 Iterative One-dimensional Hash Packing

The expected wasted space produced by $\texttt{HashPacking}_{n^{\frac{1}{3}}}$ is larger than that produced by `Match` which is of the order $\theta(\sqrt{n})$. To improve the performance of $\texttt{HashPacking}_m$, we study the distribution of the unmatched items in the output of $\texttt{HashPacking}_m$. A large proportion of these unmatched items can be further matched. They are not matched by $\texttt{HashPacking}_m$ because the wasted space produced by matching them is greater than the pre-defined requirement, $\frac{1}{m}$. Thus, if $\texttt{HashPacking}_{m'}$ is applied on those unmatched items with $m' < m$, more items will be matched. This observation motivates the formulation of the following algorithm, $\texttt{ItHashPacking}_m$.

2.2.1 $\texttt{ItHashPacking}_m$

The idea is to apply $\texttt{HashPacking}_m$ on the input iteratively. Each iteration reduces the number of buckets, m, by half.

The idea of Iterative Hash Packing on input l_n:

---

1. Let $m' \leftarrow m$ and $l'_n \leftarrow l_n$.

2. Apply $\texttt{HashPacking}_{m'}[l'_n]$.

3. Let $m' \leftarrow \lceil \frac{m'}{2} \rceil$. Let l'_n be the set of unmatched items after line 2.

4. Repeat line 2 through line 3 until m' equal 1.

Note that when $m' = 1$, there are two buckets b_0 and b_1 (line 4).

Since in every iteration, the number of buckets is reduced by half, there is a simple implementation of the above idea, $\texttt{ItHashPacking}_m$. The algorithm $\texttt{ItHashPacking}_m$ consists of two parts, `Hashing` and `Iteration`. The first part hashes the items to the buckets. The second part consists of a number of iterations. During each iteration, consecutive buckets are paired up. Each pair of buckets are either *merged* or *joined* into one bucket. The iteration is repeated until there is only two buckets left ($m' = 1$).

The merging and joining operations involve three buckets. Suppose that b_i and b_{i+1} are the two buckets in the pair, and that, c_i and c_{i+1} are the type of items in the respective buckets (recall that all items in a bucket are of the same type). The joining operation transfers the items in both buckets, b_i and b_{i+1} to another bucket, b_j. This operation is only possible if $c_i = c_{i+1}$. The merging operation is possible only if $c_i = -1$ and $c_{i+1} = +1$ (that is, b_i contains small items and b_{i+1} contains big items). This operation takes each small item from b_i and packs it with a big item from b_{i+1} into a bin. The remaining items (can be big or small items) are stored into another bucket, b_j.

The algorithm $\texttt{ItHashPacking}_m$ is as follows:

Algorithm: $\texttt{ItHashPacking}_m$

For convenience, assume $m = 2^k$ for some positive integer k.

1. **Hashing**

   Create m+1 buckets, $b_0^0, b_1^0, .., b_m^0$. Store each item x_i into the bucket $b_{h(x_i)}$. If it can be matched with another item in the same bucket, then they are packed in a bin and removed from the bucket.

   Let $j \leftarrow 0$ and $m' \leftarrow m$.

2. **Iteration**

   (a) **Phase I**

   For each odd labeled buckets, $b_i^j, i = 1, 3, .., m' - 1$. If b_i^j and b_{i+1}^j contain the same type of items, then join b_i^j and b_{i+1}^j into b_{i+1}^j. Otherwise, if b_i^j contains small items and b_{i+1}^j contains big items, then b_i^j is merged with b_{i+1}^j into b_{i+1}^j.

   (b) **Phase II**

   Create $\frac{m'}{2} + 1$ buckets, $b_0^{j+1}, ..., b_{\frac{m'}{2}}^{j+1}$. For each even labeled buckets, $b_i^j, i = 0, 2, .., m' - 2$. If b_i^j and b_{i+1}^j contain same type of items, then join b_i^j and b_{i+1}^j into $b_{\frac{i}{2}}^{j+1}$. Otherwise, if b_i^j contains small items and b_{i+1}^j contains big items, then b_i^j is merged with b_{i+1}^j into $b_{\frac{i}{2}}^{j+1}$. Note that cases where b_i^j contains big items and b_{i+1}^j contains small item never occurred.

   Transfer items in b_0^j to b_0^{j+1} and $b_{m'}^j$ to $b_{\frac{m'}{2}}^{j+1}$.

   (c) Let $m' \leftarrow \frac{m'}{2}$, and $j \leftarrow j + 1$.

   (d) Repeat line 2(a) through line 2(c) until there are only two buckets left, that is, $m' = 1$.

   (e) Pack each item in the two buckets,b_0 and b_1, into a single bin.

A sample run with $m = 8$ and $n = 16$ is shown in Figure 2.1.

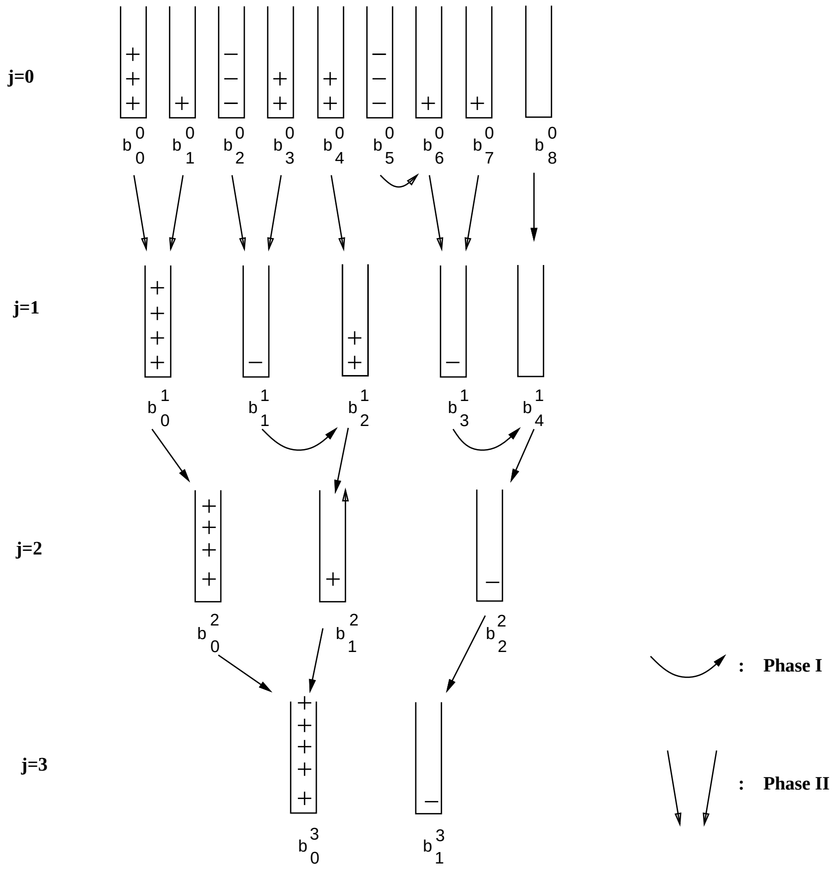

Figure 2.1: Sample run of $\texttt{ItHashPacking}_m$

2.2.2 Average case analysis

Theorem 2.3

$$E[\texttt{ItHashPacking}^W_{n^3}[L_n]] = \theta(\sqrt{n}).$$

Proof Let $m = n^3$. We will view the function h in a different way. First, $m + 1$ equi-distant grid points at $0, \frac{1}{2m}, \frac{1}{4m}, .., \frac{1}{2}$, are marked on the interval $[0, \frac{1}{2}]$. These grid points correspond to the buckets. Then, all the items are marked on this interval according to their e-value. A big item is marked as "+" while a small item is marked as "–". The function h can be viewed as a mapping which maps a big item to its nearest grid point to the left and a small item to its nearest grid point to the right (Figure 2.2). After the mapping, the initial order of "+"s and "–"s may change. For example, consider the third "–" from the left in Figure 2.2. The "–" which is originally to the left of a "+" is now to the right after the mapping. We call such case a *swap*.

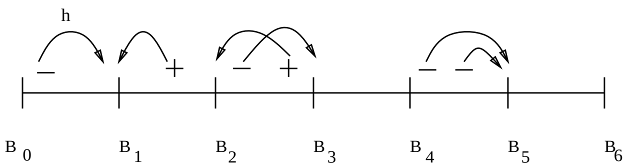

Figure 2.2: Swap of items during hashing

We will first show that with the choice of $m = n^3$, after the mapping, no swap will occur with high probability. Next we show that if no swap occurs, the wasted space produced by `ItHashPacking`$_{n^3}$ is the same as `Match` (Remark A.3), which is bounded by $O(\sqrt{n})$.

Note that if every sub-interval between two consecutive grid points contains only one item, then no swap will occur. Let E be the event that every sub-interval contains only one item. Since $m = n^3$,

$$
\begin{aligned}
Pr(E) &= (\frac{n^3}{n^3})(\frac{n^3-1}{n^3})(\frac{n^3-2}{n^3})...(\frac{n^3-n+1}{n^3}) \\
&= (1-\frac{1}{n^3})(1-\frac{2}{n^3})...(1-\frac{n-1}{n^3}) \\
&> (1-\frac{n}{n^3})(1-\frac{n}{n^3})...(1-\frac{n}{n^3}) \\
&= (1-\frac{1}{n^2})^{n-1} \\
&> 1-\frac{1}{n}.
\end{aligned}
$$

That is, with high probability, event E happens. Recall that if $Pr(E) > 1 - \frac{1}{n}$, and only the order of expected wasted space is of interest, then we can assume that event E always occurs.

Now, we assume that event E always occurs. In addition, we relabeled the items such that for any two items, x_k and $x_{k'}$, if $k > k'$, then $h(x_k) > h(x_{k'})$. For the algorithm `Match`, the number of of unmatched big items, W, (remark A.3) is

$$W = \max_{1 \le j \le n} \left\{ \sum_{i=1}^{j} c_i \right\}.$$

Recall that c_i is defined as follows:

$$c_i = \begin{cases} +1, & x_i > \frac{1}{2}, \\ -1, & otherwise. \end{cases}$$

We observe that for any k, if $c_k = -1$ and $c_{k+1} = +1$,

$$\begin{aligned} & \max_{1 \le j \le n} \left\{ \sum_{i=1}^{j} c_i \right\} \\ = & \max_{1 \le j \le n} \left\{ \sum_{1 \le i \le j, i \ne k, i \ne (k+1)} c_i \right\}. \end{aligned}$$

That is, by removing any two consecutive values of -1 and $+1$ in the sequence, the value W remains unchanged. Thus, W can be calculated by recursively discarding any two consecutive values of -1 and $+1$, until it is not possible to do so. The final sequence of c_i is of form, $+1, +1, +1, .., +1, -1, -1, ..., -1$. The number of $+1$'s in the final sequence is W.

Note that $\texttt{ItHashPacking}_{n^3}$ packs the input by continuously packing a small item x_k and a big item $x_{k'}$ in the same bucket. Since the two items are drawn from the same bucket, and the algorithm packs all possible pairs in a bucket, thus, we can assume that $k' = k + 1$. Therefore, the number of big items unmatched by $\texttt{ItHashPacking}_{n^3}$ is equal to W. Thus the expected wasted space produced by $\texttt{ItHashPacking}_{n^3}$ is the same as that of `Match` which is of the order $O(\sqrt{n})$. $\square$

2.2.3 Remark

If m is large, for example, $m = n^3$, then after `Hashing`, the buckets are "sparse" in the sense that most buckets are empty. We can avoid having empty buckets through sorting. Specifically, as follows:

1. `Hashing:` The buckets number $h(x_i)$ is calculated for each item x_i.
2. `Sorting:` The items are sorted by using the bucket number as the key. Suppose the sorted list is $x_{i_1}, x_{i_2}, ..., x_{i_n}$. Then the item x_{i_j} is stored in bucket b_j for $1 \leq j \leq n$.
3. `Iteration:` The routine `Iteration` (of `ItHashPacking`$_m$) is then applied.

Note that only n buckets are needed. The output of this version is the same as `ItHashPacking`. Furthermore, for this version, we are able to prove that at every iteration, the number of unpacked items is expected to decrease by a factor of $\sqrt{2}$ (section 3.2) which is a useful result for obtaining a work optimal parallel algorithm. In the next chapter, we will exploit this fact to obtain the second parallel algorithm.

2.3 Multi-dimensional Hash Packing

This section gives an algorithm for multi-dimensional bin-packing. Each item is a d-dimensional box, and the bin is a unit d-dimensional unit cube.

2.3.1 `HashPacking`$_{d,m}$

For an item $x = (x^1, x^2, ..., x^d)$, where $x^i \in (0, 1]$, define its type, $c = (c^1, c^2, ..., c^d)$ where,

$$c^i = \begin{cases} +1, & x^i > \frac{1}{2} \\ -1, & otherwise. \end{cases}$$

As in the one-dimensional case, define the e-value of an item, $(e^1, e^2, ..., e^d)$, where,

$$e^i = \begin{cases} 1 - x^i, & x^i > \frac{1}{2} \\ x^i, & otherwise. \end{cases}$$

Define the function, h, on items as follows,

$$\begin{aligned} h((x^1, ..., x^d)) &= (f^1, ..., f^d) \quad \text{where} \\ f^i &= \begin{cases} \lfloor 2e^i m \rfloor, & c^i = +1 \\ \lceil 2e^i m \rceil, & otherwise. \end{cases} \end{aligned}$$

Taking the two-dimensional bin-packing as example, an item (0.8,0.8) is classified as type (+1,+1), and an item (0.1,0.8) is classified as (−1,+1). The respective e-values are (0.2,0.2) and (0.1, 0.2). The function h is illustrated in Figure 2.3.

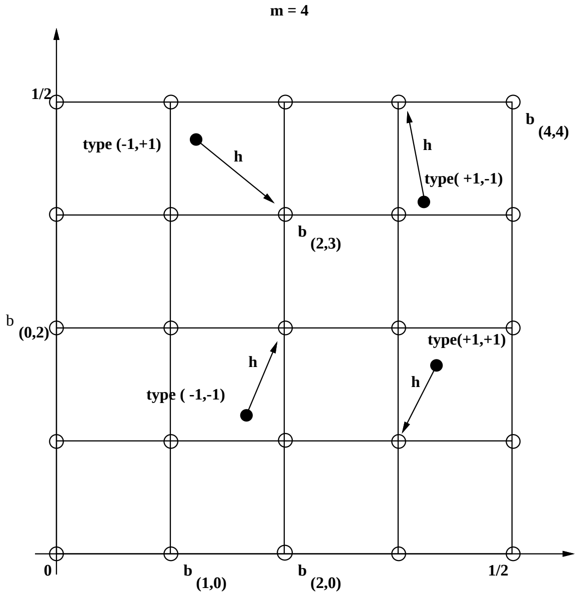

Figure 2.3: Effect of function h on two-dimensional items

Algorithm: $\texttt{HashPacking}_{d,m}[l_n]$:

---

1. Create m^d buckets and label them as $b_{(i^1,i^2,..,i^d)}$ where $i^1, i^2, .., i^d$ are integers between 0 and m.

2. For each item, x_i, store it into the bucket $b_{h(x_i)}$.

3. For each bucket, partition the items in the bucket into maximum number of groups such that all groups except one contain 2^d items of distinct types. For each group that contains all distinct types, use a single bin to pack the items in the group. For each item in the group that does not contain all distinct types, use a single bin to pack it.

The following lemma states that items of pairwise distinct type which are hashed to the same bucket can be packed in a single bin. This implies the correctness of `HashPacking`$_m$.

Lemma 2.4 *Given 2^d items $x_1, x_2, ..., x_{2^d}$. Suppose that all the items are of pairwise distinct types, and that for all $i, j = 1, 2, ..., 2^d$, $h(x_i) = h(x_j)$, then all the items can be packed in a single bin.*

Proof We will prove by induction on t the following statement. Note that this statement directly implies the claim.

Given a list S of 2^t items $x_1, ..., x_{2^t}$ that are of pairwise distinct type, and for all i,j, $h(x_i) = h(x_j)$. Furthermore, all the items are of the form:

$$x_i = (x_i^1, x_i^2, .., x_i^t, 1, 1, .., 1),$$

where $x_i^1, x_i^2, .., x_i^d \in (0, 1]$. Then all these items can be packed in a single bin.

Basis

When t=0, there is only one item (1,1,..,1,1).

Induction

Now assume that the statement holds for $t = k$. Consider the case where $t = k + 1$. Then S contains 2^{k+1} items. These items are of the form:

$$x_i = (x_i^1, x_i^2, .., x_i^{k+1}, 1, 1, .., 1).$$

Note that the type of an item is of the form: $(c_i^1, c_i^2, .., c_i^{k+1}, 1, 1, ..., 1)$. Since they are of pairwise distinct type, half of them have their type at $(k+1)$-th co-ordinate as $+1$ and another half as -1. Furthermore, for each item x_i in S, there exists one and only one item in S, x_j, such that both x_i and x_j have the the same type on all co-ordinate except the $(k+1)$-th. That is, $c_i^s = c_j^s$ for all s except $s = k + 1$.

Since $h(x_i) = h(x_j)$, thus, both items can be packed in the following bin:

$$(\max(x_i^1, x_j^1), \max(x_i^2, x_j^2), ..., \max(x_i^k, x_j^k), 1, 1, ..., 1). \tag{2.2}$$

Pair the items in S up as above and treat each pair as a single item given by (2.2). Now, there are 2^k items which satisfy the condition of the statement for $t = k$. Therefore, the list of 2^{k+1} items can be packed in a unit bin. $\square$

2.3.2 Average-case analysis

As in the one-dimensional case, we will first find a relation between the parameter m and the expected performance (in terms of the number of bins used) of `HashPacking`$_{d,m}$. Then the number of bins used is minimized by choosing a suitable m.

Lemma 2.5 *Let $Y_1, .., Y_n$ be n independently and identical distributed random variables, where*

$$Pr(Y_i = y) = \frac{1}{2^d}, \ \textit{for } y = 1, 2, .., 2^d.$$

Let $X_1, .., X_{2^d}$ be random variable where each X_i is the total number of Y_j, $1 \leq$

$j \leq n$ having the value i. Then

$$E[\max_{1 \leq i \leq 2^d}(X_i) - \min_{1 \leq i \leq 2^d}(X_i)] = \theta(\sqrt{n}).$$

Proof Suppose $W = \{x_1, x_2, .., x_{2^{d'}}\}$ is a set of integers, define $ABS(W) = \max(W) - \min(W)$. That is, $ABS(W)$ is the difference between the maximum and minimum integer in the set W. We prove this lemma by induction on d.

Basis

When $d = 1$, then the expected absolute difference between X_1 and X_2 is $\theta(\sqrt{n})$.

Induction hypothesis

Suppose $\sum_{i=1}^{2^{d-1}} X_i = n$, then

$$E[ABS(\{X_1, ..., X_{2^{d-1}}\})] = \theta(\sqrt{n}).$$

Induction Step

Suppose that $\sum_{i=1}^{2^d} X_i = n$. Let

$$\begin{aligned} Z_1 &= \sum_{i=1}^{2^{d-1}} X_i, \quad \text{and} \\ Z_2 &= \sum_{i=2^{d-1}+1}^{2^d} X_i. \end{aligned}$$

The expected absolute different between Z_1 and Z_2 is $\theta(\sqrt{n})$, since $Z_1 \sim B(n, \frac{1}{2})$.

Consider $X_1, ..., X_{2^{d-1}}$, by induction hypothesis, the expected $ABS(\{X_1, ..., X_{2^{d-1}}\})$ is $\theta(\sqrt{Z_1})$. However, $Z_1 \leq n$, thus

$$\begin{aligned} E[ABS(\{X_1, ..., X_{2^{d-1}}\})] &= O(\sqrt{n}), \text{ and} \\ E[ABS(\{X_{2^{d-1}+1}, ..., X_{2^d}\})] &= O(\sqrt{n}). \end{aligned}$$

Since $ABS(\{X_1, X_2, .., X_{2^d}\}) \leq ABS(\{X_1, ..., X_{2^{d-1}}\}) + ABS(\{X_{2^{d-1}+1}, ..., X_{2^d}\})$ therefore,

$$E[ABS(\{X_1, X_2, .., X_{2^d}\})] = O(\sqrt{n}).$$

Furthermore since

$$ABS(\{X_1, X_2, .., X_{2^d}\}) \geq ABS(\{X_1, ..., X_{2^{d-1}}\})$$

therefore, we have

$$E[ABS(\{X_1, X_2, .., X_{2^d}\})] = \theta(\sqrt{n}).$$

$\square$

Lemma 2.6 *For every $\alpha < 1$, $m = n^\alpha$, when n is large enough,*

$$E[\texttt{HashPacking}_{d,m}[L_n]] = \frac{n}{2^d} + \theta(\frac{n}{m}) + \theta(\sqrt{nm^d}).$$

Proof This proof is a generalization of the proof of Lemma 2.1.

We count the number of bins required to pack the items in *boundary buckets* (defined later in the proof) and *non-boundary buckets* separately.

Boundary Buckets:

Consider the function, h, and the first co-ordinate of the item's size. By definition of this function, buckets in the form $b_{(0,i^2,i^3,...,i^d)}$, where $0 \leq i^2, ..., i^d \leq m$, do not contain any items of type $(-1, c^2, c^3, ..., c^d)$ where $c^2, .., c^d$ are either $+1$ or -1. Similarly, buckets of the form $b_{(m,i^2,i^3,...,i^d)}$ do not contain any items of type $(+1, c^2, c^3, ..., c^d)$. Therefore, items in these buckets can not form a set of all types. Thus, the algorithm will pack each item in a single bin.

In general, for every j, buckets of the form, $b_{(i^1,i^2,...,i^{j-1},0,i^{j+1},...,i^d)}$ and $b_{(i^1,i^2,...,i^{j-1},m,i^{j+1},...,i^d)}$ do not contain any items of type $(c^1, ..., c^{j-1}, -1, c^{j+1}, .., c^d)$ and $(c^1, ..., c^{j-1}, 1, c^{j+1}, .., c^d)$ respectively, where $0 \leq i^1, i^2, ..i^d \leq m$ and $c^1, c^2, .., c^d \in \{-1, +1\}$. Let us call these buckets as the *boundary buckets*.

Let M be the total number of boundary buckets and H be the total number of items hashed to the boundary buckets. Since all the non-boundary buckets are of the form $b_{(i^1,i^2,..,i^d)}$ where $1 \le i^1, i^2, .., i^d \le m-1$, thus $M = m^d - (m-1)^d = \theta(m^{d-1})$. The probability that an item is hashed to a certain boundary bucket is not necessarily the same for all boundary buckets. For example, the probability that an item is hashed to the boundary bucket $b_{(0,0)}$ is $(4m^2)^{-1}$ while the probability that it is hashed to $b_{(1,0)}$ is $1(2m^2)^{-1}$. In general, the probability that an item is hashed to a boundary bucket is between m^{-d} and $(2m)^{-d}$. Therefore the expected number of items in a boundary bucket is of the order $\theta(nm^{-d})$. Thus,

$$E[H] = M\theta(\frac{n}{m^d}) = \theta(m^{d-1})\theta(\frac{n}{m}) = \theta(\frac{n}{m}). \tag{2.3}$$

Non-boundary Buckets:

Now, consider the non-boundary buckets. In a way similar to the proof for one-dimensional case, we can show that with high probability, every bucket has $\theta(nm^{-d})$ items hashed to it.

Now, we assume that every bucket has $\theta(nm^{-d})$ items hashed to it and will prove that expected number of bins needed to pack all the items in the non-boundary buckets is $n2^{-d} + \theta(\sqrt{nm^{-d}})$. Recall that in the algorithm, items are partitioned into maximum number of groups such that each group (excluding one) has 2^d items of distinct types. Take any non-boundary bucket, say b_j. let $T^j_{(c^1,..,c^d)}$ be the random variable for the number of items of type $(c^1, .., c^d)$ in bucket b_j.

Let

$$\begin{aligned} N_j &= \sum_{c^1,\ldots,c^d \in \{-1,1\}} T^j_{(c^1,\ldots,c^d)}, \\ T^j_{min} &= \min_{c^1,\ldots,c^d \in \{-1,1\}} T^j_{(c^1,\ldots,c^d)}, \quad \text{and} \\ T^j_{max} &= \max_{c^1,\ldots,c^d \in \{-1,1\}} T^j_{(c^1,\ldots,c^d)}. \end{aligned}$$

The number of groups containing all distinct types is T^j_{min}. For those items which are unable to form a group of distinct types, each one of them is packed in a separate bin. Thus the number of bins required to pack all the items in b_j is

$$(T^j_{min}) + (N_j - 2^d T^j_{min}).$$

Note that,

$$\begin{aligned}(N_j - 2^d T^j_{min}) &< (2^d T^j_{max} - 2^d T^j_{min}) \\ &= 2^d (T^j_{max} - T^j_{min}).\end{aligned}$$

By Lemma 2.5, $E[(T^j_{max} - T^j_{min})] = \theta(E[\sqrt{N_j}])$. The expected number of bins required to pack the items in all the non-boundary buckets is

$$\begin{aligned}&E[\sum_{j\in[1,m-1]^d} T^j_{min} + N_j - 2^d T^j_{min}] \\ &= \sum_{j\in[1,m-1]^d} (E[T^j_{min}] + E[N_j - 2^d T^j_{min}]) \\ &\leq \sum_{j\in[1,m-1]^d} (E[\frac{N_j}{2^d}] + \theta(\sqrt{\frac{n}{m^d}})) \\ &= \frac{n}{2^d} + \theta(\sqrt{m^d n}).\end{aligned}$$

Therefore, the expected number of bins required by $\texttt{HashPacking}_{d,m}$ is

$$\frac{n}{2^d} + \theta(\frac{n}{m}) + \theta(\sqrt{nm^d}) = \frac{n}{2} + \theta(\frac{n}{m}) + \theta(\sqrt{nm^d}).$$

□

To minimize the number of bins, choose m as $n^{(d+2)^{-1}}$.

Theorem 2.7

$$E[\texttt{HashPacking}^W_{d,n^{(\frac{1}{d+2})}}[L_n]] = \theta(n^{(\frac{d+1}{d+2})})$$

□

Chapter 3

Parallelization

In this chapter, we describe the parallelization of an one-dimensional bin-packing algorithm, `ItHashPacking`$_m$ where m is chosen to be n^3 (in the rest of this chapter, m is chosen to be n^3 unless otherwise stated).

Two parallel algorithms are presented here, `PItHashPacking` and `PSortItHashPacking`. Both algorithms assume a CRCW PRAM (concurrent read, concurrent write, parallel random-access machine) model [9]. The first algorithm, `PItHashPacking`, uses n processors, $O(n^3)$ memory space, and runs in $O(\log n)$ time. The second algorithm, `PSortItHashPacking`, uses n processors, $O(n)$ memory space, and runs in expected $O(\log n)$ time. The work done of both algorithms is $O(n \log n)$. By applying the randomized Brent's scheduling principle given in section 3.3, the second algorithm can be improved to achieve expected $O(n)$ work done which is optimal and expected $O(\log^2 n)$ time complexity, assuming that the input follows the uniform distribution. The output of both algorithms is the same as the output of `ItHashPacking` which produces an expected wasted space of $\theta(\sqrt{n})$.

3.1 PItHashPacking

This algorithm assumes a CRCW model which allows more than one processor simultaneous read and write access to the same memory location and if m processors simultaneously write to a location, then exactly one (an arbitrary one) will succeed.

The main idea of this algorithm is to parallelize `ItHashPacking` (refer to section 2.2).

We will present `PItHashPacking` by first illustrating the difficulties which lie in the merging and joining operations of `ItHashPacking`. Next, we describe in detail the method used to overcome these difficulties. Then we describe `PItHashPacking` followed by a sample run of `PItHashPacking`.

3.1.1 Basic Operations: Merging and Joining operations

Before describing the algorithm, we make the following observations. Recall that in `ItHashPacking`, $O(\log n)$ iterations are needed. In each iteration, two phases are carried out to combine two neighboring buckets into one. There are two ways to combine two neighboring buckets, merging and joining. After each iteration, the number of buckets is approximately reduced by half.

Note that for `ItHashPacking`, an iteration can be carried out only if the previous iteration is completed. Thus, the parallel version of this algorithm has running time of $\Omega(\log n)$ since there is a total of $\log n$ iterations. It seems that one "natural" parallelization scheme is to assign a processor to each bucket to maintain the items in it in every iteration. However, this will not achieve $O(\log n)$ time as explained below. When merging two neighboring buckets, it takes a number of sequential steps, proportional to the number of items in the buckets, to complete the merging. In addition, the expected number of items in the buckets is not constant. (Consider the last iteration, the expected number of items in a bucket is of the order $\theta(\sqrt{n})$). Hence, the total running time of the parallel algorithm will not be bounded by $O(\log n)$.

We overcome this problem by assigning a processor to each item. Each processor will maintain the status of the item assigned to it and will carry out the merging and joining operations needed to be done on this item. The details of the operations is described below.

On input $\langle x_1, x_1, .., x_n \rangle$, the item x_i is assigned to the processor p_i. Two arrays, B and D, are needed to store information about the buckets and items. To represent the fact that a bucket, b_k contains items $x_{i_1}, x_{i_2}, ..., x_{i_m}$, a pointer pointing to the head of a list formed by $x_{i_1}, x_{i_2}, ..., x_{i_m}$, and a value indicating the number of elements in this list are stored in the array cell B_k. Items are linked together by having their indices stored in neighboring cells in array D. Furthermore, each processor is given a private memory area to store information about the item assigned to it and the the list that contains this item. This information includes:

- which item is assigned to the processor,
- the location of the item in array D, and
- which bucket the item is in.

Note that other information about the list like the location of the head and tail of list can be obtained from the above information. For convenience, the head of a list containing big items is always on the left (an item x_i is said to be on the left of x_j if they are stored in $D_{i'}$ and $D_{j'}$ respectively where $i' < j'$), and is on the right for a list containing small items.

Figure 3.1 shows an example in which:

1. Bucket b_1 contains item x_1,
2. Bucket b_2 is empty,
3. Bucket b_3 contains items x_2, x_4 and x_5 (since the head of the list is on the left, x_2, x_4 and x_5 are big items), and
4. Item x_3 is not in any one of the buckets.

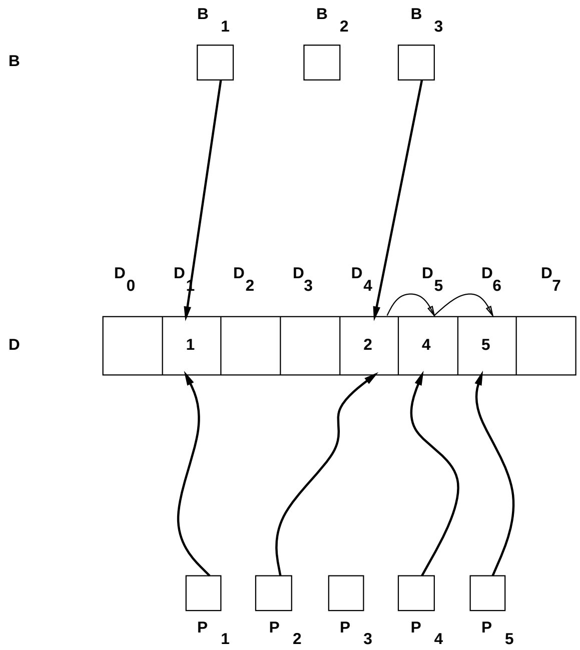

Figure 3.1

Suppose that two buckets (lists) are to be merged. Consider an item x_i in one of the list and suppose that x_i is assigned to the processor p_i. If x_i is the k-th item from the tail of the list, then it should be packed with the k-th item from the tail of the other list. Assume that p_i knows which lists are to be merged (in section 3.1.2, we will explain how p_i obtains this information). Since p_i can computes the location of both the list's tails, it can decide in constant time, which item x_i is to be packed with. Figure 3.2 illustrate the merging operation.

Merging:

Variables H_1 and H_2 are the respective indices (in D) of the heads of the left and right list. Similarly, T_1 and T_2 are the respective indices of the tail of the left and right list. The variable i is the location in D where the item x_k (which is assigned to the processor p_k) is stored (Figure 3.2).

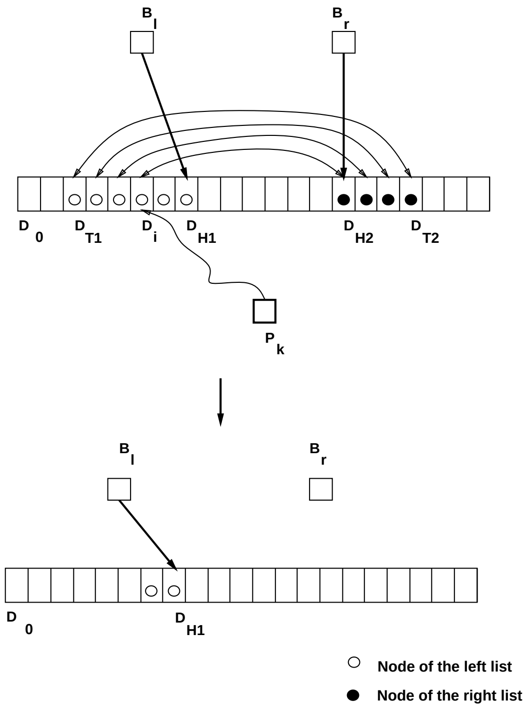

Figure 3.2: Merging two lists

For all processors p_k,

If item x_k is in the left list, **then**

begin

If $T_2 - (i - T_1) \geq H_2$, **then**

begin

Pack x_k with $x_{D_{T_2-(i-T1)}}$

$D_k \leftarrow 0$

end

If x_k is the head of the list,

then update array B for the change of the list size.

Update the private memory for the change of location of x_k in array D.

end

If item x_k is in the right list, **then**

begin

If $T_1 + (T_2 - i) \leq H_1$, **then**

begin

Pack x_k with $x_{D_{T_1+(T_2-i)}}$

$D_k \leftarrow 0$

end

If x_k is the head of the list,

then update array B for the change of the list size.

Update the private memory for the change of location x_k in array D.

end

Similarly, two lists of small items can be joined in constant time by the following algorithm.

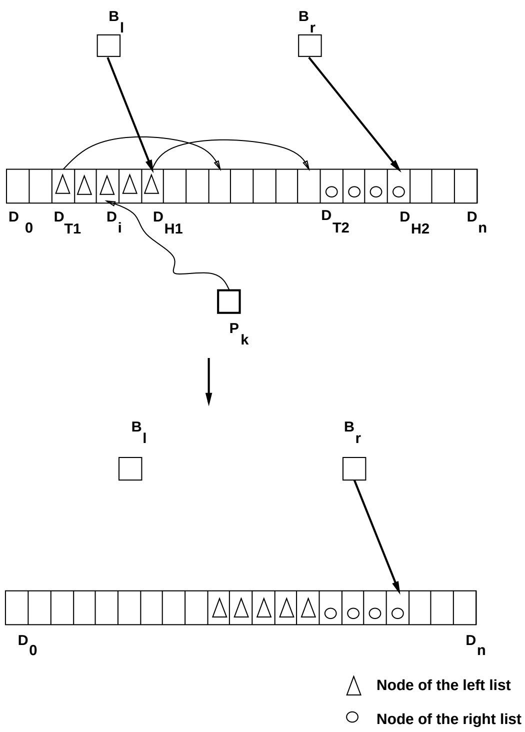

Figure 3.3: Joining two lists

Joining:

Variables H_1 and H_2 are the respective indices (in D) of the heads of the left and right list. Similarly, T_1 and T_2 are the respective indices of the tail of the left and right list. The variable i is the location in D where the item x_k (which is assigned to the processor p_k) is stored (Figure 3.3).

For all processors p_k, $1 \leq k \leq m$ do

    If item x_k is in the left list, **then**

        begin

            $Next \leftarrow i + T_2 - H_1 - 1$

            $D_i \leftarrow 0$

            $D_{Next} \leftarrow k$

            If x_k is the head of the list,

                then update array B for the change of the list size.

            Update the private memory for the change of location of x_k in array D.

        end

    If item x_k is in the right list, **then**

        If x_k is the head of the list, **then** update array B for the change of the list size.

The joining operation on two lists of big items is similar to the above except that the heads of the lists are on the left.

3.1.2 Description of the Algorithm

The parallel algorithm consists of two parts: `Hashing` and `Iteration`.

First part :`Hashing`

Each processor p_i, computes the bucket to which the item x_i is hashed to. The following is the initialization of the array D and B.

For each processor, $p_i, 1 \leq i \leq n$, do

$$D_{h(x_i)} \leftarrow i$$

$$B_{h(x_i)} \leftarrow h(x_i)$$

Note that there are cases where more than one item, for example, items x_j and x_k, can have the same hash value, that is, $h(x_j) = h(x_k)$. This results in two processors, p_j and p_k, trying to write to the same memory location, $D_{h(x_j)}$. In this case, only one of the processors will succeed. The other processor, after knowing that it has failed to write, will pack the item assigned to it into a bin by itself. Similarly, for more than two processors, only one will succeed.

Second part : `Iteration`

First, every processor, p_i, is assigned to the item x_i. The iteration described below is repeated until only two buckets left. In each iteration, there are two phases.

1. **First Phase**

   Group every odd-labeled bucket with next higher labeled bucket. Then each processor p_i will first determine which bucket b_i the item x_i is in, and the type c_i of this item. Then it checks the type, c, of the items in the bucket that is grouped with b_i. If $c_i = c$, then a joining operation is needed. If $c_i = -1$ and $c = +1$, then a merging operation is needed. If $c_i = +1$ and $c = -1$, then nothing is to be done. By the above checking, p_i can decide which operation, whether a merge or join operation is to be done on the two buckets. Using the merging and joining operation described above, this phase can be done in constant time.

   Note that when the first phase is completed, the number of buckets still remains the same. Furthermore, during the first phase, there are cases where a bucket containing big items is grouped with a bucket containing small items. Since it is not possible to pack a big item with a small item on the right, no list operation is done in this case.

2. **Second Phase**

   The second phase is similar to the first phase except that it groups each even-labeled bucket with the next higher labeled bucket.

   Note that after the two phases are completed, it is impossible for any even-labeled bucket and the next higher labeled bucket both to be non-empty. Now, if an odd-labeled bucket is non-empty, transfer its content to the next lower labeled bucket (by just copying the list header). Next, all the odd-labeled buckets are removed. The remaining buckets are then relabeled as $0, 1, 2, 3, ..., m/2$. The process of relabeling may be costly. However, in this case, since the new labels are exactly half of the previous labels, hence new labels can be computed in parallel in constant time. After computing the new labels, updating need to be done on array B and the private memory of all the processors. These can be done in constant time. The detail is left out for conciseness.

3.1.3 An Example

We describe a sample run of this algorithm to further illustrate the algorithm. Figure 3.4 `(0)` shows the initial setting. Figure 3.4 `(1a)`, `(1b)` show respectively the configuration after the completion of the first and second phase of the first iteration. Similarly Figure 3.4 `(2a)`, `(2b)`, `(3a)`, `(3b)` show the resulting configuration of the two phases during the second and third iteration.

For the initial setting (`0`), processors p_i are assigned to items x_i and bucket b_i contains item x_i for $0 \leq i \leq 16$. Consider `1a` (completion of the first phase of the first iteration),

1. items in b_1 and b_2 are joined (`A*`),

2. items in b_5 and b_6 are merged (`B*`), and

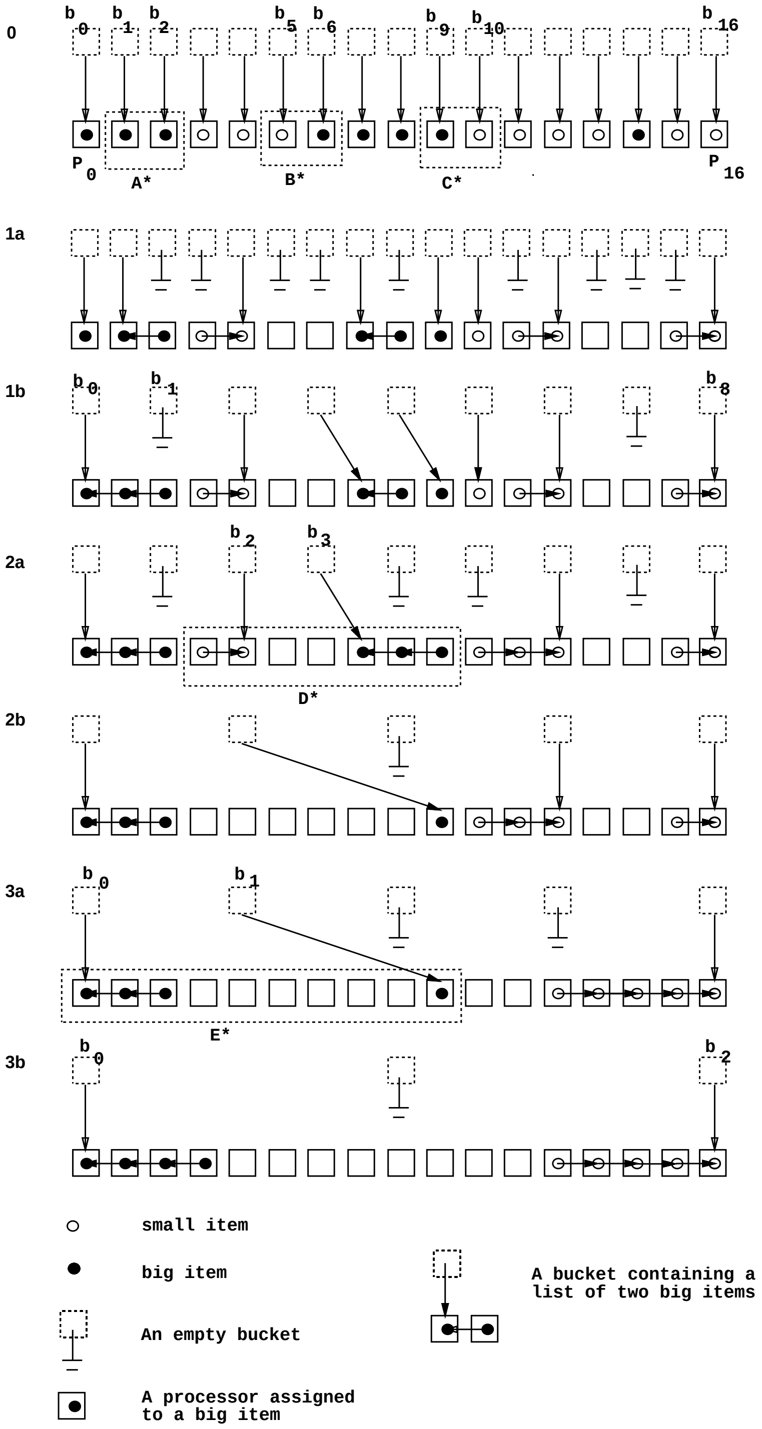

Figure 3.4: Sample run

3. no list operation is done for items in b_9 and b_{10} as it is impossible to pack a big item with a small item to its right (`C*`).

Consider `2b` (completion of the second phase of the second iteration), the two lists in b_2 and b_3 are merged (`D*`).

Consider `3b`, the two lists in b_0 and b_1 are joined (`E*`).

3.1.4 Complexity of `PItHashPacking`

Since every iteration can be completed in a constant time, and there are $\log n$ iterations, thus `PItHashPacking` runs in $\theta(\log n)$ time. In addition, `PItHashPacking` use n processors, thus the work done of `PItHashPacking` is $O(n \log n)$. This algorithm uses $O(n^3)$ of memory space.

3.2 `PSortItHackPacking`

This algorithm needs n processors and runs in expected $O(\log n)$ time. It can be modified using a randomized Brent's scheduling principle (to be discussed in section 3.3) to a randomized algorithm which uses $O(n \log^{-2} n)$ processors and runs in expected $O(\log^2 n)$ time if the input follows the uniform distribution.

The idea of `PSortItHashPacking` is to parallelize the modified `ItHashPacking` described in section 2.2.3 which consists of three parts: hashing, sorting and iterations. Similarly, the parallel algorithm consists of three parts, `Hashing, Sorting and Iteration.`

- `Hashing`: for each $i = 1, 2, .., n$, p_i picks item x_i and calculates the bucket number, $h(x_i)$.

- `Sorting`: the n items are sorted using bucket number as key. Since the number of items is n and the key range is $[0, n^3]$, the randomized parallel integer sorting algorithm presented in [12] can be used. This integer sorting algorithm needs $n \log^{-1} n$ processors and runs in expected $O(\log n)$ time.

- `Iteration` is the same as `Iteration` of `PItHashPacking`.

The running time for `Hashing`, is clearly constant. the expected running time for `Sorting` is $O(\log n)$ and running time for `Iteration` is $O(\log n)$. Thus, the expected running time of the second algorithm is $O(\log n)$.

The `Hashing`, and `Sorting`, are clearly work optimal. However `Iteration` is not work optimal as there is a linear time algorithm for a single processor to complete the work. This algorithm treats items in each bucket as a linear linked list. Thus the joining operation can be done in constant time. Furthermore, there are only $O(n)$ joining operations needed to be done in `Iteration`. A single merging operation can not be done in constant time. However, the total time needed for all merging operations is bounded by $O(n)$ since there are at most n items needed to be packed. Thus this algorithm runs in linear time.

In the next section, we will give a randomized method, which under certain conditions, is able to transform a parallel algorithm to a randomized algorithm whose expected work done is reduced. In section 3.4, we will prove that `PSortHashPacking` satisfies the conditions stated in section 3.3 and will show that by applying this randomized method, `PSortHashPacking` can be transformed into a work optimal algorithm.

3.3 Randomized Brent's Scheduling Principle

Given a parallel algorithm A, we propose a method to simulate A such that under certain conditions, the expected work done of the simulation is less than the expected work done of A. To avoid confusion, we used the term *virtual processors* $p_i, i = 1, 2, ...$, when referring to the processors needed by the original algorithm and the term *simulator* $s_i, i = 1, 2, ..$, when referring to the processors used to simulate the original algorithm. Suppose that on input of length n, a parallel algorithm A takes $T_v(n)$ steps by using $U_v(n)$ processors, $p_i, i = 1, 2, .., U_v(n)$. At the t-th step, it may not be necessary that all the $U_v(n)$ processors are needed. Let $u_t(n)$ be the number of necessary processors during the t-th step of computation. If $u_t(n)$ decreases fast as t increases, then most of the processors will be idle for a significant portion of the time. However, the idling time is also included in the calculation of work done. The Brent's scheduling principle gives a method which reduces the proportion of idle processors with the cost of increasing the number of steps by a constant factor [9]. Consider the parallel algorithm A, the work done is the product $T_v(n)U_v(n)$. Brent's scheduling principle gives a way to simulate A with $U_s(n)$ processors, where $U_s(n) < U_v(n)$, for all n. To simulate the t-th step computation, the jobs of the $u_t(n)$ virtual processors are evenly distributed to the simulators. Hence, the number of steps required for the simulation of the t-th step computation is $\lceil u_t(n)/U_s(n)\rceil \leq (u_t(n)/U_s(n))+1$. Therefore, the total parallel steps required for the simulation is less than $\left(\sum^t u_t(n)\right)/U_s(n)+T_v(n)$. In some cases, by choosing a suitable $U_s(n)$, the work done decreases but the running time still remains in the same order of magnitude. For example, to sum n integers, by assigning a processor to each integer, it is easy to obtain a parallel algorithm which runs in $\log n$ steps (Figure 3.5). In this case, $u_t(n) = O(n2^{-t})$, $U_v(n) = n$, $T_v(n) = O(\log n)$, and the work done is $O(n \log n)$. By using $U_s(n) = O(n \log^{-1} n)$ simulators, the time taken for the simulation is $O(\log n)$ and the work done is $O(n)$.

Note that Brent's scheduling principle assumes that there is a way to distribute the $U_v(n)$ virtual processors evenly to the $U_s(n)$ simulators before the actual simulation, which is not always possible. We propose a randomized version of Brent's scheduling, which solve this problem by randomly allocating the virtual processors to the simulators.

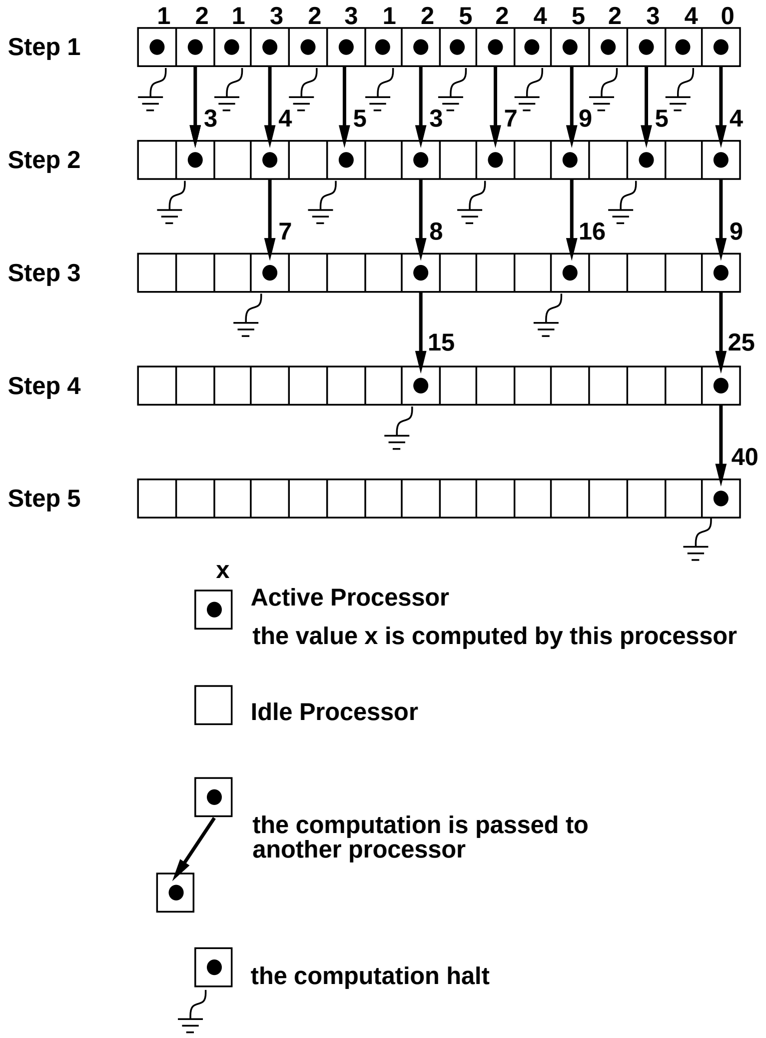

Figure 3.5: Parallel computation of summation

Randomized Brent's Scheduling

During the execution of a parallel algorithm A, we say that a virtual processor is *active* at step t if it is needed at the t-th step of the computation, otherwise, it is said to be idle at the t-th step (Figure 3.6). Consider a virtual processor p_i at step t. After the completion of step t, the computation is passed to a set of processors which does not necessarily contain p_i. We assume that the number of virtual processors in this set does not exceed a constant independent to the input length. This is a fair assumption because if this number depends on some variable, then the running time of the computation of one step of A is not constant. If p_i passes its computation to p_j after the t-th step, we say that p_i activates p_j at step $t+1$ (Figure 3.6).

To simulate the t-th step of A, the active virtual processors are first determined and then randomly assigned to the simulators. After the assignment, the simulators start to simulate the virtual processors assigned to them. Thus the amount of time taken by a simulator depends on the number of virtual processors assigned to it. Furthermore, the simulation of the next step, $(t+1)$-th step of A, can be started only after all the simulators complete their task. That is, even if a particular simulator completes its task fast, it still has to wait for all the simulators completing their task in order to carry out the next simulation step. Therefore, the time taken to complete the simulation of one step of A depends on maximum number of virtual processors assigned to a simulator. There are algorithms that need not to have this constraint. We will explain more on this point later.

The simulation of A is as follows:

---

1. For each simulator s_i, $1 \leq i \leq U_s(n)$, do

   Let K_i be the set $\{ p_j | j = q(i-1)+1, q(i-1)+2, ..., qi \}$,

   assuming $q = (U_v(n)/U_s(n))$ is an integer.

2. For each simulator s_i, $1 \leq i \leq U_s(n)$, do

   Simulate the virtual processors in K_i for one step (of A).

   Let K_i' be the virtual processors activated by the

   virtual processors in K_i during the above simulation.

   $K_i \leftarrow \phi$.

3. For each simulator s_i, $1 \leq i \leq U_s(n)$, do

For each $p_j \in K_i'$, do

a. randomly choose an integer m from interval $[1, .., U_s(n)]$.

b. $K_m \leftarrow K_m + \{p_j\}$.

c. If s_i fails to add p_j to the set K_m, then goto line 3.a.

4. Repeat line 2 and 3 until all $K_i = \phi$.

In line 3.a, more than one simulator, say $s_{i_1}, .., s_{i_r}$, may at the same time, choose the same value for m. That is, these simulators trying to add an element to the set K_m at the same time. This can be viewed as these simulators trying to write to a location. In this case, only one of them will succeed (CRCW PRAM). Those simulators which fail, after knowing that they have failed, will try again.

Let S_i^t be the random variable denoting the number of virtual processors assigned to the simulator s_i at the step t of the simulation. Note that $S_1^t + S_2^t + .. + S_{U_s(n)}^t = u_t(n)$. Hence, the random variables $S_1^t, .., S_{U_s(n)}^t$ follow the multi-nomial distribution [5]. Let

$$S^t = \max(S_1^t, S_2^t, ..., S_{U_s(n)}^t).$$

Consider the simulation of the t-th step of A. Let T_i^t be the running time of s_i for the t-th step and

$$T^t = \max(T_1^t, T_2^t, ..., T_{U_s(n)}^t).$$

The total running time for the simulation is:

$$T_s = \sum_{t=1}^{T_v(n)} T^t.$$

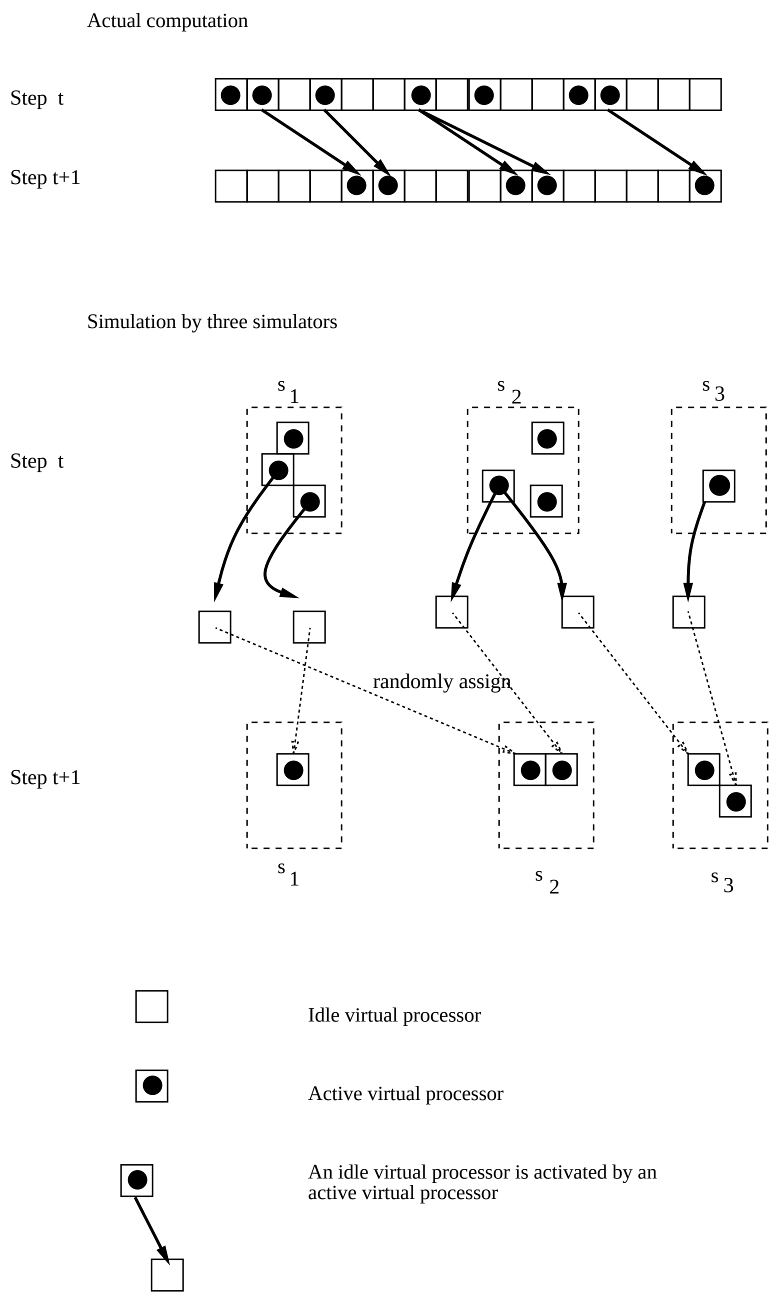

Figure 3.6: Randomized Brent's scheduling

The followings lemma states the relation between T^t and S^t.

Lemma 3.2 *Suppose $U_s(n) = n \log^{-2} n$. If $S^t \geq \log n$, then with high probability,*

$$T^t = O(S^t).$$

Otherwise, with high probability,

$$T^t = O(\log n).$$

Proof Consider the running time of a simulator s_i for the simulation of the t-th step of A. The running time of s_i for line 1 is $O(|K_i|) = O(S^t)$. The running time of s_i for line 2 depends on the size of K_i and K_i'. Since each virtual processor can only activate at most a constant number of processors, thus $|K_i'| < C|K_i|$ for some constant C. Therefore, the running time for line 2 is $O(|K_i|) = O(S^t)$.

The running time for line 3 depends on the set K_i' and the number of times s_i fails to execute line 3(b). Let T_i' the number of tries s_i takes to execute line 3(b). Furthermore, let

$$T' = \max(T_1', T_2', ..., T_{U_s(n)}').$$

The process of assigning the elements in K_i' can be viewed as a sequence of T_i' Bernoulli trail. A success occurs when s_i succeeds in carrying out line 3(b). The probability of a success is $((U_s(n)-1)(U_s(n))^{-1})^{U_s(n)-1} < 2e^{-1}$ for large enough n. Recall that $|K_i'| = O(S_i^t)$. If $S_i^t \geq \log n$, then there is a constant D such that[4],

$$Pr(T_i' > DS_i^t) < \frac{1}{n^2}.$$

If $S_i^t < \log n$, then

$$Pr(T_i' > D\log n) < \frac{1}{n^2}.$$

By *Boole's inequality*, if $S^t \geq \log n$,

$$\begin{aligned} Pr(T' < DS^t) &\leq Pr(T_1' < DS^t, T_2 < DS^t, .., T_{U_s(n)} < DS^t) \\ &< \frac{1}{n^2}U_s(n) < \frac{1}{n}, \end{aligned}$$

and if $S^t < \log n$,

$$\begin{aligned} Pr(T' < D\log n) &\leq Pr(T_1' < D\log n, T_2' < D\log n, .., T_{U_s(n)} < \log n) \\ &< \frac{1}{n}. \end{aligned}$$

$\square$

We divide the simulation into two parts: for $1 \leq t \leq B \log\log n$ and $B \log\log n < t \leq \log n$, where B is a positive constant whose value will be determined in the proof of Lemma 3.3.

Lemma 3.3 *Suppose $u_t(n) = nc^{-t}$, where $c > 1$, $T_v(n) = \log n$, and $U_s(n) = n \log^{-2} n$. There exists a positive constant B such that for $1 \leq t \leq B \log\log n$,*

$$S^t = O(E[S_1^t]),$$

with high probability, and for $B \log\log n < t \leq \log n$,

$$S^t = O(\log n),$$

with high probability.

Proof Note that in general,

$$\begin{aligned}
Pr(S^t = g) &< Pr(S_1^t = g \text{ or } S_2^t = g \text{ or } \ldots \text{ or } S_{U_s(n)}^t = g) \\
&< \sum_{i=1}^{U_s(n)} Pr(S_i^t = g), \\
Pr(S^t \geq g) &= \sum_{k=g}^{n} Pr(S^t = k) \\
&< \sum_{k=g}^{n} \sum_{i=1}^{U_s(n)} Pr(S_i^t = k) \\
&= \sum_{i=1}^{U_s(n)} \sum_{k=g}^{n} Pr(S_i^t = k).
\end{aligned}$$

Therefore, we have

$$Pr(S^t \geq g) < \sum_{i=1}^{U_s(n)} Pr(S_i^t \geq g). \tag{3.1}$$

Let $E_i^t = E[S_i^t]$ (the expected number of virtual processors distributed to the simulator s_i in the t-th step). We divide the proof into two parts: for $1 \leq t \leq B \log\log n$ and $B \log\log n < t \leq \log n$, where $B = \log^{-1} c$.

part 1: $1 \leq t \leq B \log\log n$:

Note that, for any i, $E_i^t = u_t(n)/U_s(n) = c^{-t}\log^2 n$. We will prove that $Pr(S^t < D_1 E_i^t) > 1 - \frac{1}{n}$ where D_1 is a positive constant, $D_1 = e^2$. By Lemma A.1,

$$Pr(S_i^t > D_1 E_i^t) \quad < \quad e^{D_1 E_i^t - E_i^t} . \left(\frac{E_i^t}{D_1 E_i^t}\right)^{D_1 E_i^t} . \tag{3.2}$$

Since $t \leq B \log\log n$, we have

$$\begin{aligned} D_1 E_i^t &= \frac{D_1 \log^2 n}{c^t} \\ &\geq D_1 \frac{\log^2 n}{c^{B \log\log n}} \\ &= D_1 (\log n)^{2 - B\log c} \\ &= D_1 \log n \quad (\text{since} \quad B = \log^{-1} c). \end{aligned}$$

From (3.2),

$$\begin{aligned} Pr(S_i^t > D_1 E_i^t) &< e^{D_1 E_i^t - E_i^t} \left(\frac{E_i^t}{D_1 E_i^t}\right)^{D_1 E_i^t} \\ &< e^{D_1 E_i^t - E_i^t} \left(\frac{1}{D_1}\right)^{D_1 \log n} \\ &< \left(\frac{e}{D_1}\right)^{D_1 \log n} = e^{-e^2 \log n} = n^{-e^2} \\ &< \frac{1}{n^3}. \end{aligned}$$

By (3.1),

$$\begin{aligned} Pr(S^t > D_1 E_i^t) &< U_s(n) \frac{1}{n^3} \\ &< \frac{1}{n}. \end{aligned}$$

part 2: $B \log\log n < t \leq \log n$:

We will prove that $Pr(S^t < D_2 \log n) > 1 - \frac{1}{n}$, for some positive constant D_2. Since $t > B \log\log n$, $E_i^t < \log n$. By Lemma A.1, and choosing a large enough constant

D_2, say $D_2 = e^2$,

$$
\begin{aligned}
Pr(S_i^t > D_2 \log n) &< e^{D_2 E_i^t - E_i^t} \left(\frac{E_i^t}{D_2 E_i^t} \right)^{D_2 E_i^t} \\
&< \left(\frac{e \log n}{D_2 \log n} \right)^{D_2 \log n} = e^{-e^2 \log n} \\
&< \frac{1}{n^3}.
\end{aligned}
$$

By (3.1),

$$
Pr(S^t > D_2 \log n) < U_s(n) \frac{1}{n^3} < \frac{1}{n}.
$$

□

Theorem 3.4 *Suppose $u_t(n) = nc^{-t}$, where $c > 1$, $T_v(n) = \log n$, and $U_s(n) = n \log^{-2} n$, then*

$$
E[T_s] = O(\log^2 n).
$$

Proof By Lemma 3.3, for $1 \le t \le B \log \log n$, $S^t = O(E_1^t)$ with high probability. Since $E_1^t > \log n$, by Lemma 3.2, $E[T^t] = O(E_1^t)$. Similarly, for $B \log \log n < t \le \log n$, $S^t = O(\log n)$ with high probability. By Lemma 3.2, $E[T^t] = O(\log n)$. We have

$$
\begin{aligned}
O(E[\sum_{t=1}^{T_v(n)} T^t]) &= O(E[\sum_{t=1}^{B \log \log n} T^t]) + O(E[\sum_{t=B \log \log n+1}^{\log n} T^t]) \\
&= O(\sum_{t=1}^{B \log \log n} E_1^t) + (\log n) O(\log n) \\
&= O\left(\log^2 n\right) + (\log n) O(\log n) \\
&= O\left(\log^2 n\right).
\end{aligned}
$$

□

Remark

Note that T^t is the time to simulate the t-th step of A. Whereas T_i^t is the time taken by a simulator s_i in simulating the t-th step of A. Recall that $T^t = \max(T_1^t, T_2^t, ..., T_{U_s(n)}^t)$.

The reason that T^t is the maximum of T_i^t is that in order for a simulator to start simulating the $(t+1)$-th step of A, all simulators must complete their simulation of the t-th step. However, there are algorithms that need not satisfy this constraint. For example, the list ranking algorithm [9]. Without this constraint, it is possible to transform the non-work optimal list ranking algorithm to a work optimal randomized algorithm which runs in expected $O(\log n)$ time [9]. Note that by applying the randomized Brent's scheduling, although a work optimal algorithm is obtained, the expected running time is of the order of $O(\log^2 n)$.

3.4 Work optimal algorithm

This section shows that the randomized Brent's scheduling can be applied to `PSortHashPacking` to obtain a work optimal parallel algorithm.

The following lemma states that during running of `PSortItHashPacking`, the number of items which remain unpacked decreases by a constant factor with high probability after each iteration.

Lemma 3.5 *Consider* `PSortItHashPacking`*. Suppose that n items are in n buckets where each bucket contains exactly one item and the procedure* `Iteration` *is then applied. Let N_i be the number of unpacked items after the i-th iteration. Then there exists a constant $D > 1$, such that*

$$Pr[N_i > \frac{n}{D^i}] < \frac{1}{n} \quad \textit{for sufficiently large } n.$$

Proof Assume that n is a power of 2, $n = 2^k$. Let i be some integer such that $i < k$. Furthermore, let x_i be the item in the i-th bucket and X_i be the random variable for x_i. For the purpose of this proof, the items are divided into groups of 2^i items in the following manner:

$$G_1 = \langle x_1, x_2, .., x_{2^i} \rangle,$$

$$
\begin{aligned}
G_2 &= \langle x_{2^i+1}, x_{2^i+1}, ..., x_{2^{i+1}} \rangle, \\
&\cdots \\
G_{2^{k-i}} &= \langle x_{2^k-2^i+1}, x_{2^k-2^i+2}, ..., x_{2^k} \rangle.
\end{aligned}
$$

Consider G_j, $1 \leq j \leq 2^{k-i}$. We compare how items in G_j are packed by the first i iterations when all the n items are taken as input, with how they are packed when only items in G_j are taken as input. Note that they are packed in a similar way except that items unpacked in the latter case may be packed in the former. Thus the total number of unpacked items after the first i iterations when all the n items are taken as input is at most $\sum_{j=1}^{2^{k-i}} M_j$, where M_j is the number of unpacked items when G_j is the input to `Iteration`, for $j = 1, .., 2^{k-i}$.

Now, $G_1, G_2, .., G_{2^{k-i}}$ are treated as 2^{k-i} independent inputs. Each $G_j, 1 \leq j \leq 2^{k-i}$ contains 2^i items.

In the proof of Theorem 2.3, we have shown that the number of unpacked items produced by $\texttt{ItHashPacking}_{n^3}$ is same as that of `Match`. Using the same technique in the proof of Theorem 2.3, we can show that the number of unpacked items produced by `Iteration` on input G_j is same as that of `Match` on input G_j. Note that the random variables for the items in G_j may not be independent and identically distributed random variables with uniform distribution. However, from Remark A.3, their corresponding types are independent and identically distributed random variables with

$$Pr(C_i = -1) = Pr(C_i = +1) = \frac{1}{2}.$$

(Recall that if $X_i > \frac{1}{2}$, then $C_i = +1$. Otherwise, $C_i = -1$.) Together with Lemma A.2, we have $Pr(M_j > 2^{i(\frac{1}{2}+\alpha)}) < e^{-n^{2\alpha}/8}$ for any positive constant α. By the *Boole's inequality*,

$$
\begin{aligned}
Pr(M_1 > 2^{i(\frac{1}{2}+\alpha)}, M_2 > 2^{i(\frac{1}{2}+\alpha)}, .., M_{2^{k-i}} > 2^{i(\frac{1}{2}+\alpha)}) &\leq 2^{k-i} e^{\frac{-n^{2\alpha}}{8}} \\
&< \frac{1}{n}, \quad \text{for sufficient large } n.
\end{aligned}
$$

Thus, with high probability, the total number of unpacked items is less than

$$\begin{aligned} 2^{i(\frac{1}{2}+\alpha)}2^{k-i} &= \frac{n}{2^{i(\frac{1}{2}-\alpha)}} \\ &= \frac{n}{D^i}, \end{aligned}$$

where $D = 2^{\frac{1}{2}-\alpha}$ is a constant. □

Theorem 3.6 *The randomized Brent's scheduling can be applied on* `PSortHashPacking` *to give an algorithm which needs $O(n \log^{-2} n)$ processors and runs in expected $O(\log^2 n)$ time.*

Proof Recall that `PSortHashPacking` consists of three parts, `Hashing`, `Sorting` and `Iteration`. Clearly, by using $n \log^{-2} n$ processors, `Hashing` needs $O(\log^2 n)$ time. By using the integer sorting algorithm, `Sorting` can be done in expected $O(\log n)$ time using $O(n \log^{-1} n)$ processors. Since $n \log^{-2} n < n \log^{-1} n$, thus it can be done in expected $O(\log^2 n)$ time using $O(n \log^{-2} n)$ processors. By Lemma 3.5, with high probability, at every iteration, the number of items remaining unpacked is reduced by a constant factor during the execution of `Iteration`. Since the number of active processors is proportional to the number of unpacked items, thus with high probability, the number of active processors is reduced by a constant factor. That is, using the notation in Theorem 3.4, $u_t(b) = n2^{-\frac{t}{2}}$. Note that $T_v(n) = \log n$. By Theorem 3.4. with with $U_s(n) = n \log^{-2} n$ simulators, we are able to obtain a randomized parallel algorithm for `Iteration` which uses $O(n \log^{-2} n)$ processors and runs in expected $O\left(\log^2 n\right)$ time. □

To reduce the work done of `PSortHashPacking`, the randomized Brent's scheduling principle is used instead of the usual Brent's scheduling principle. This is necessary because in order to apply the Brent's scheduling principle, the processor allocation must not to be a problem, that is, there must be a deterministic way for each simulator to decide which set of virtual processors it needs to simulate before the start of the simulation. However, this is not possible for `ItHashPacking` as it is impossible to determine which items will be packed before the computation. Thus, we need to apply the randomized version of Brent's scheduling principle. Furthermore, the randomized method applied in list ranking problem [9] is not applicable here because to simulate the $(t+1)$-th step, all simulators must first complete the simulation of the t-th step.

Chapter 4

Conclusion

We have presented a heuristic algorithm `HashPacking`and this algorithm has been generalized and modified for different settings. For d-dimensional on-line setting, our algorithm produces $\theta(n^{(d+1)/(d+2)})$ expected wasted space. There is an algorithm which produces $O(\sqrt{n \log n})$ expected wasted space for one-dimensional on-line setting [14]. To the best of our knowledge, there is no result on average-case analysis for higher dimensional on-line setting. However, note that for off-line setting, the best known bound is $O(n^{(d-1)/d})$ for $d \geq 2$, and $\theta(\sqrt{n})$ for $d = 1$ [8].

By repeatedly applying `HashPacking`, we obtain `ItHashPacking` which produces $O(\sqrt{n})$ expected wasted space for one-dimensional off-line bin-packing. Unfortunately, it seems that the generalization of this method to a higher dimensional setting may not achieve good bound. One draw-back of `ItHashPacking` is that it does not achieve good bound for worst-case.

An advantage of `ItHashPacking` is that it is inherently parallel. We gave a parallelization of `ItHashPacking` which achieves optimal expected work done. To obtain the expected work optimal parallel algorithm, we proposed the randomized Brent's scheduling principle, which is able to transform a parallel algorithm to a work optimal one under certain conditions.

Appendix A

Lemma A.1 [11] *Let $X_1, X_2, ..., X_m$ be independent and identically distributed random variables, with $Pr[X_i = 1] = p$ and $Pr[X_i = 0] = 1 - p$. Let $E[S_m] = E[\sum X_i] = mp > 0$. Then, for any $\delta > 0$, $Pr[S_m > (1+\delta)mp] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{mp} < \left(\frac{e}{1+\delta}\right)^{(1+\delta)mp}$.*

Lemma A.2 [2] *Let $X_i, i = 1, 2, ..., n$, be independently and identical distributed random variables where $Pr(X_i = 1) = Pr(X_i = -1) = \frac{1}{2}$ for all i. Let $S_i = X_1 + X_2 + ... + X_i$, and $\hat{S}_i = S_i - E[S_i]$. Then for $y \geq 0$*

$$\begin{aligned} Pr\left(\max_{1\leq i\leq n} \hat{S}_i \geq y\sqrt{n}\right) &\leq 2e^{-\frac{y^2}{8}}, \quad \textit{and} \\ Pr\left(\max_{1\leq i\leq n} \hat{S}_i \leq -y\sqrt{n}\right) &\leq 2e^{-\frac{y^2}{8}}. \end{aligned}$$

Remark A.3 [2] We will describe an one-dimensional bin-packing algorithm `Match`. The algorithm `Match` iterates the following until all items are packed. Let S be a list of unpacked items. Choose the largest item in S, say x_i. Search for the largest item in S, say x_j, that can packed with x_i in a bin (that is, $x_i + x_j \leq 1$). If such an item does not exist, then x_i is packed in a bin and removed from S. Otherwise both items are packed in a bin and removed from S.

Suppose $l_n = \langle x_1, x_2, .., x_n \rangle$ is an input to `Match`. Let

$$e_i = \begin{cases} 1 - x_i, & x_i > \frac{1}{2}, \\ x_i, & otherwise, \quad \text{and} \end{cases}$$

$$c_i = \begin{cases} +1, & x_i > \frac{1}{2}, \\ -1, & otherwise. \end{cases}$$

Let X_i be the random variable for x_i and C_i be the random variable for c_i. Relabeled x_i such that $e_1 \leq e_2 \leq ... \leq e_n$. Call the item x_i where $x_i > \frac{1}{2}$ a big item. Then the number of big items packed alone in a bin is given by

$$\max_{1 \leq i \leq n} \sum_{k=1}^{i} c_k$$

Furthermore, if $X_1, X_2, ..., X_n$ are independent and identically distributed random variables with uniform distribution, $X_i \sim U[0,1]$, then after the relabeling, $C_1, C_2, .., C_n$ are independent and identically distributed random variables where

$$Pr(C_i = -1) = Pr(C_i = +1) = \frac{1}{2} \quad \text{for} \quad i = 1, 2, .., n.$$

Bibliography

[1] E. G. Coffman Jr, M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing - an updated survey. In G. Ausiello, M. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer System Design*. Springer-Verlag, 1984.

[2] E. G. Coffman Jr. and G. S. Lueker. *Probabilistic analysis of packing and partitioning algorithms*. Wiley-Interscience, 1991.

[3] D. Coppersmith and P. Raghavan. Multidimensional on-line bin packing: Algorithms and worst-case analysis. *Operation Research Letters*, 8:17–20, 1989.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] W. Feller. *Probability theory and its applications*. John Wiley & Sons, New York, 1952.

[6] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-Completeness*. Freeman, 1979.

[7] M. R. Garey and D. S. Johnson. *Analysis and design of algorithms in combinatorial optimization*. Springer-Verlag, 1981.

[8] R. M. Karp, M. Luby, and A. Marchetti-Spaccamela. A probabilistic analysis of multidimensional bin packing problems. *STOC*, pages 289–298, 1984.

[9] R. M. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In J. V. Leeuwen, editor, *Handbook of Theoretical Computer Science Vol A*. MIT Press, 1990.

[10] W. Knodel. *A Bin Packing Algorithm with Complexity $O(n \log n)$ and Performance 1 in the Stochastic Limit*, volume 118 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[11] P. Raghavan. Probabilistic construction of deterministic algorithms: Approximating packing integer programs. *Journal of Computer and System Sciences*, 37, 1988.

[12] J. H. Reif. An optimal parallel algorithm for integer sorting. *FOCS*, 1985.

[13] P. W. Shor. The average-case analysis of some on-line algorithms for bin packing. *Combinatorica*, 6(2):179–200, 1986.

[14] P. W. Shor. How to pack better than best fit: Tight bounds for average-case on-line bin packing. *FOCS*, pages 752–759, 1991.