Multicore Parallelization of the PTAS Dynamic Program for the Bin-Packing Problem

Anirudh Chakravorty[1], Thomas George[2], and Yogish Sabharwal[2]

[1] Indraprastha Institute of Information Technology, Delhi
anirudh10014@iiitd.ac.in
[2] IBM Research – India
{thomasgeorge,ysabharwal}@in.ibm.com

Abstract. Dynamic Programming (DP) is an efficient technique to solve combinatorial search and optimization problems. There have been many research efforts towards parallelizing dynamic programs. In this paper, we study the parallelization of the Polynomial Time Approximation Scheme (PTAS) DP for the classical bin-packing problem. This problem is challenging due to the fact that the number of dimensions of the DP table is not known a priori and is dependent on the input and the accuracy desired by the user. We present optimization techniques for parallelizing the DP for this problem, which include diagonalization, blocking and optimizing dependency lookups. We perform a comprehensive evaluation of our parallel DP on a multicore platform and show that the parallel DP scales well and that our proposed optimizations lead to further substantial improvement in performance.

1 Introduction

Dynamic programming (DP) is a classical technique used to solve a large variety of combinatorial optimization problems in the areas of scheduling, inventory management, VLSI design, bioinformatics, etc [9,14]. The main idea behind dynamic programming is to solve complex problems by breaking them into simpler subproblems. It is applicable to problems exhibiting the *optimal substructure* and *overlapping subproblems* properties:

– *Optimal substructure* implies that the solution to a given optimization problem can be obtained by combining optimal solutions to its subproblems.
– *Overlapping subproblems* means that the space of subproblems must be small. While a recursive algorithm solving the problem would solve the same subproblems over and over again, a dynamic program solves each subproblem only once and stores the solution to be reused thereafter.

Dynamic Programming can also be applied to obtain approximation algorithms for many problems, e.g. 0-1 knapsack, bin-packing and minimum makespan scheduling[17,18]. As a matter of fact, *Polynomial Time Approximation Schemes (PTAS)* can be designed for many of these problems based on dynamic programming, wherein, we can obtain a solution having cost within a factor of $1+\epsilon$ of the optimal solution for any ϵ provided by the user. Thus, a DP can be designed to produce a solution S, such that

$$Cost(S) \quad \leq \quad (1+\epsilon) \cdot Cost(Opt)$$

where *Opt* is an optimal solution to the problem. While such accuracy is desirable in many applications, the running time of such dynamic programs can be prohibitively large, typically exponential in $1/\epsilon$. The storage requirements can also be very high. This makes parallel processing an attractive approach to implement such dynamic programs.

Many parallel applications of dynamic programming have been described in research literature; they have been generally designed for very specific problems and only run on special parallel architectures (torus, hypercube, etc.) [10,7,19,4,8,3,12,13,11,5]. Most of these problems deal with dynamic programs having fixed (generally 2) number of dimensions.

In this paper, we study the parallelization of the PTAS dynamic program for the classical bin-packing problem. The bin-packing problem is a fundamental problem in combinatorial optimization that is used as a kernel for many other optimization problems, such as minimum makespan scheduling on parallel machines. What makes this problem more challenging is the fact that the number of dimensions of the dynamic programming table is variable – it is dependent on the characteristics (*weights*) of the the input set of items to be packed and also on the desired accuracy (ϵ parameter specified by the user). Thus, computation of an entry of the DP table is dependent on a variable number of entries (not known a priori). These distinguishing characteristics make the problem more challenging and it is unclear if known and tried optimizations apply to this problem or not. In this paper, we make the following contributions:

(1) We show that the approach of filling the DP table by traversing the entries along the diagonals is well suited for parallelism. We also show that the degree of parallelism increases with increasing number of dimensions.
(2) We next show that the blocking (tiling) approach also works well in our scenario. The main reason is that due to the particular structure of the dependencies in the bin-packing problem, while filling an entry of the DP table, we require to look up a lot of entries that are packed close together. Therefore, blocking (tiling) improves cache-efficiency due to data-locality.
(3) Lastly, we propose an optimization for the bin-packing problem that takes advantage of the relationship amongst the dependencies in order to avoid some of the dependency lookups. For this, we partition the dependencies into two sets, called the *primary dependencies* and the *secondary dependencies*. While filling an entry of the DP table, we first process the primary dependencies and if all the primary dependencies are valid, we do not need to process the secondary dependencies. This reduces the random memory accesses thereby improving the running-time. We show that this can lead to up to 20% improvement in performance.
(4) We do a comprehensive study of the parallelization of the bin-packing dynamic program and report our observations.

2 Related Work

Parallel processing is an efficient approach for solving large-scale DP problems and is the subject of extensive research. Grama et. al.[9] presented a classification of dynamic programming formulations in order to characterize the kind of parallel programming techniques that can be applied in each case. Specifically, they categorize dynamic programs along two directions: (a) the first based on the proximity of dependent sub-problems that make up the overall multistage problem , (b) the second based on the multiplicity of terms in the recurrence that determines the solution to the optimization problem. A dynamic program is considered *serial* if the subproblems at all levels depend only on the results of the immediately preceding levels and *non-serial* otherwise. It is considered *monadic* if the recurrence relation contains a single recursive term and *polyadic* otherwise. Based on this classification, one can define four classes of DP formulations: serial monadic (e.g., single source shortest path problem, 0/1 knapsack problem), serial polyadic (e.g., Floyd all pairs shortest paths algorithm), non-serial monadic (e.g., longest common subsequence problem, Smith-Waterman algorithm) and nonserial polyadic (e.g., optimal matrix parenthesization problem and Zuker algorithm). Note that not all DP problems can be categorized into the above classes. The classical bin-packing problem is one such problem that does not fall into any of the above classifications discussed by Grama et. al.[9]. It is closest to the nonserial polyadic class. However, the fact that the number of terms in the recurrence is not determined a priori and is dependent on the input makes the problem stand apart from other DP problems and also makes the parallelization a challenge.

In the past, there has been some work on parallelization of nonserial polyadic and related DP programs. However, most of the existing work generally pertains to very specific problems and is applicable to special parallel architectures (torus, hypercube, etc.) [10,7,19,4,8,3,12,13,11]. In particular, Tan and Gao[15,16] study the parallel performance of nonserial polyadic DP algorithms in the context of the RNA secondary structure prediction problem on a specific multi-core architecture. They propose parallel processing of independent triangular tiles that lie along the diagonal. Elkihel and Baz[2,6] presented parallel algorithms for the closely related 0-1 knapsack problem using a novel load-balancing strategy for distributing the workload. Alves et. al.[1] presented parallel dynamic programming algorithms for the string editing problem based on dynamic scheduling of blocks to the processors. They recursively divide the blocks into smaller blocks that are then scheduled for processing.

An important aspect that makes our problem very different from previously studied parallel dynamic programs is the number of dimensions of the dynamic programming table – while previous literature deals with two dimensional dynamic programs, the number of dimensions in our problem can be very large depending on the number of distinct item weights and the desired accuracy parameter. While our approach to parallelization via diagonalization and blocking is similar to those proposed in the past[15,16,1], we study these approaches for multi-dimensional dynamic programs for the first time. Another distinguishing factor in our problem is the large number of terms that appear in the DP recurrence. This calls for different kind of optimizations than previously known in the research literature.

3 The Bin-Packing Problem

In the bin-packing problem, we are given a set $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ of n items where each item, a_i, has a weight $0 < w(a_i) \leq 1$ associated with it. The goal is to find the minimum number of bins, each of unit capacity, required to store the items so that the total weight of the items stored in any bin does not exceed its capacity, 1. This problem is known to be NP-hard[1].

We now discuss the dynamic programming based PTAS for the Bin-packing problem[17,18]. This algorithm reduces the bin-packing problem to a special instance wherein there are only a constant number of item weights. In this specialization, called the *restricted bin-packing* problem, the weights of all the items are sampled from a fixed set $\mathcal{W} = \{w_1, w_2, \ldots, w_k\}$ of k weights, where k is a constant. That is $w(a_i) \in \mathcal{W}$ for all $a_i \in \mathcal{A}$. This special version can be solved optimally in polynomial time using dynamic programing (c.f. Section 3.1).

The pseudocode for the PTAS is presented in Figure 1. The algorithm can be divided into three phases. In the first phase we reduce the bin-packing problem to an instance of the restricted bin-packing problem after removing some items and rounding the weights of the remaining items. In the second phase, we solve the restricted bin-packing problem optimally using a dynamic program. Finally in the third phase, we augment the solution of the restricted bin-packing problem with the earlier removed items to form a feasible solution to the original bin-packing instance.

First Phase: We first remove all the items having weight less than ϵ. Let $\mathcal{A}_\epsilon$ be the set of removed items and $\mathcal{A}'$ be the resulting set of items. We partition the set $\mathcal{A}'$ into $r = \lceil log_{(1+\epsilon)} \frac{1}{\epsilon} \rceil$ groups $A_1, A_2, \ldots, A_r$ where for $1 \leq i \leq r$, A_i is the set of all the items having weight in the range $[\epsilon(1+\epsilon)^{i-1}, \epsilon(1+\epsilon)^i)$. Next, for each $1 \leq i \leq r$, we round down the weight of all the items in the group A_i to $\epsilon(1+\epsilon)^{i-1}$. Note that the instance determined by the items $\mathcal{A}'$ with the modified weights is an instance of the restricted bin packing problem with at most r different weights.

Second Phase: In this phase we solve the restricted bin-packing problem optimally on the instance $\mathcal{A}'$ with the modified weights. The dynamic program for achieving this is formally described in Section 3.1. Let the solution thus obtained be S', i.e., S' is a partition of the items of $\mathcal{A}'$ into sets $R_1, R_2, \ldots, R_t$ where R_j specifies the set of items packed into bin j.

Third Phase: In the third phase we augment the solution returned by the restricted bin-packing problem to a solution for the original instance of the bin-packing problem. This is achieved by introducing back the items removed in phase 1 using the *first-fit* algorithm. For this, we consider the items of $\mathcal{A}_\epsilon$ in an

[1] The Bin-packing problem is weakly NP-hard; it is solvable in polynomial time when the input is in unary.

```
Input: Item Set, $\mathcal{A}$; weight function, $w$; and approximation parameter, $\epsilon$

Phase 1:
Let $\mathcal{A}_\epsilon$ be the set of items having weight $< \epsilon$, i.e., $\mathcal{A}_\epsilon = \{a \in \mathcal{A} : w(a) < \epsilon\}$
Let $\mathcal{A}'$ be the remaining set of items, i.e., $\mathcal{A}' = \mathcal{A} \setminus \mathcal{A}_\epsilon$
For $i = 1$ to $\lceil log_{(1+\epsilon)} \frac{1}{\epsilon} \rceil$
    Let $A_i = \{a \in \mathcal{A}' : \epsilon(1+\epsilon)^{i-1} \leq w(a) < \epsilon(1+\epsilon)^i\}$
    For all $a \in A_i$, define $w'(a) = \epsilon(1+\epsilon)^{i-1}$

Phase 2:
Let $S' = \{R_1, R_2, \ldots, R_t\}$ be the solution obtained on invoking
    Restricted-Bin-Packing( $\mathcal{A}'$, $w'$, $k$ )

Phase 3:
Initialize $S = S'$
For each $a \in \mathcal{A}_\epsilon$
    If $\exists R \in S$, such that $\sum_{a' \in R} w(a') + w(a) \leq 1$
        add $a$ to $R$
    else
        create a new bin $R_{|S|+1}$ and add $a$ to $R_{|S|+1}$

Return $S$
```

Fig. 1. PTAS for the bin-packing problem

arbitrary order. For each item, we find the first bin in which it fits using the original weights. If we find such a bin, the item is added to that bin. If no such bin exists, we create a new bin and add the item to this new bin.

3.1 Restricted Bin-Packing Problem

We now discuss a dynamic program that solves the restricted bin-packing problem optimally in polynomial time. Let $\mathcal{A}'$ be the set of all the items. We fix an ordering on the item weights say $w'_1, w'_2, \ldots w'_k$. Let n_i be the number of items of weight w'_i for $1 \leq i \leq k$. Note that an instance, I, of the packing problem can be defined by a k-tuple $(p_1, p_2, \ldots, p_k)$ specifying the number of items for each weight. We set up a k-dimensional table $BINS$ of size $(n_1+1, n_2+1, \ldots, n_k+1)$. $BINS(p_1, p_2, \ldots, p_k)$ denotes the minimum number of bins required to pack the items of $I = (p_1, p_2, \ldots, p_k)$, i.e., p_1 items of weight w'_1, p_2 items of weight w'_2, …, p_k items of weight w'_k. Note that the total weight for an instance, $I = (p_1, p_2, \ldots, p_k)$, is given by $\sum_{i=1}^{k} p_i \cdot w'_i$. We first compute the set Q of all instances for which the total weight is at most 1 – clearly only one bin is required to fit these items. Thus

```
Input: Item Set, $\mathcal{A}'$; weight function, $w'$; number of distinct item weights, $k$

For $i = 1$ to $k$
    Let $A'_i = \{a \in \mathcal{A}' : w(a) = w'_i\}$
    Let $n_i = |A'_i|$

Let $\mathcal{U} = \{(p_1, \ldots, p_k) : 0 \leq p_i \leq n_i \; \forall \; 1 \leq i \leq k\}$ be the set of all valid $k$-tuples
Let $\mathcal{Q} = \{(p_1, \ldots, p_k) : 0 \leq p_i \leq n_i \; \forall \; 1 \leq i \leq k$ and $\sum_{i=1}^{k} p_i \cdot w_i \leq 1\}$

For all $(q_1, q_2, \ldots, q_k) \in \mathcal{Q}$
    set $\mathcal{BINS}(q_1, q_2, \ldots, q_k) = 1$
    set $\mathcal{Q}_{BINS}(q_1, q_2, \ldots, q_k) = (q_1, q_2, \ldots, q_k)$

For all $(p_1, p_2, \ldots, p_k) \in \mathcal{U} \setminus \mathcal{Q}$
    Let $(\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_k) =$
                 $argmin_{(q_1, q_2, \ldots, q_k) \in Q : q_i \leq p_i \forall i} \mathcal{BINS}(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k)$
    set $\mathcal{BINS}(p_1, p_2, \ldots, p_k) = 1 + \mathcal{BINS}(\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_k)$
    set $\mathcal{Q}_{BINS}(p_1, p_2, \ldots, p_k) = (\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_k)$

Initialize solution $S' = \phi$.
Let $(p_1, p_2, \ldots, p_k) = (n_1, n_2, \ldots, n_k)$
While $(p_1, p_2, \ldots, p_k) \neq (0, 0, \ldots, 0)$
    Let $(\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_k) = \mathcal{Q}_{BINS}(p_1, p_2, \ldots, p_k)$
    Let $R$ be a subset of $\mathcal{A}'$ such that $R \cap A'_i = \hat{q}_i \; \forall 1 \leq i \leq k$
    Add $R$ to $S'$ and update $\mathcal{A}' = \mathcal{A}' \setminus R$
    Update $(p_1, p_2, \ldots, p_k) = (p_1 - \hat{q}_1, p_2 - \hat{q}_2, \ldots, p_k - \hat{q}_k)$

Return $S'$
```

Fig. 2. Restricted bin-packing problem

$$Q = \{(p_1, p_2, \ldots, p_k) : 0 \leq p_i \leq n_i \; \forall \; 1 \leq i \leq k \text{ and } \sum_{i=1}^{k} p_i \cdot w'_i \leq 1\}$$

For all instances $(q_1, q_2, \ldots, q_k) \in Q$, we initialize $BINS(q_1, q_2, \ldots, q_k) = 1$.

Now we can use the following recurrence for computing the remaining entries:

$$BINS(p_1, p_2, \ldots, p_k) = 1 + \min_{\substack{(q_1, q_2, \ldots, q_k) \in Q: \\ q_i \leq p_i \;\; \forall i}} BINS(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k)$$

In order to trace back the optimal solution, we maintain another k-dimensional table Q_{BINS} of size $(n_1 + 1, n_2 + 1, \ldots, n_k + 1)$. For an entry $BINS(p_1, p_2, \ldots, p_k)$, we store in $Q_{BINS}(p_1, p_2, \ldots, p_k)$ the corresponding entry of Q that yields the optimal solution for this entry of $BINS$.

Once the DP table is completely filled, we can trace back and retrieve the optimal solution as follows. The entry $BINS(n_1, n_2, \ldots, n_k)$ tells us the optimal number of bins required. The entry $Q_{BINS}(n_1, n_2, \ldots, n_k)$ tells us the contents of one bin of this optimal solution, i.e., the number of items of each weight that fit in one bin in the optimal solution. Let $Q_{BINS}(n_1, n_2, \ldots, n_k)$ be $(\hat{q}_1, \hat{q}_2, \ldots, \hat{q}_k)$. Thus we can now create one bin of the optimal solution by selecting any $\hat{q}_1$ items of weight w'_1, $\hat{q}_2$ items of weight w'_2, ..., $\hat{q}_k$ items of weight w'_k from $\mathcal{A}'$. We add this bin to the solution and then remove these items from $\mathcal{A}'$ so that they are not considered for any further bins. We then process $BINS(n_1 - \hat{q}_1, n_2 - \hat{q}_2, \ldots, n_k - \hat{q}_k)$ in a similar manner. We proceed recursively like this, adding bins to the solution until no more items remain.

To analyze the complexity of the dynamic program, we see that computing each entry takes $O(n^k)$ time. Thus, the entire table can be computed in $O(n^{2k})$ time as Q is of size $O(n^k)$. The final solution is obtained in the entry $BINS(n_1, n_2, \ldots, n_k)$.

4 Multicore Parallelization and Optimization

In this section, we discuss our strategies for parallelizing the dynamic program for the bin-packing problem efficiently on multicore platforms.

4.1 Parallelization via Diagonalization

The maximum time in the dynamic program is spent in computing the entries of the k-dimensional table $BINS$. Recall that the dynamic program recurrence is given by

$$BINS(p_1, p_2, \ldots, p_k) = 1 + \min_{\substack{(q_1, q_2, \ldots, q_k) \in Q: \\ q_i \le p_i \ \forall i}} BINS(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k)$$

We note that we cannot parallelize the computation of any two entries that lie along the same dimension, this is because the entries of Q may have 0's in some of their entries implying that the entries of $BINS$ lying along the same dimension may be dependent on each other.

Consider the sum function, $\mathcal{Z}$, that maps the k-tuple index of each entry of the $BINS$ array to the sum of the indices in the k-tuple, i.e., for any index $(p_1, p_2, \ldots, p_k)$ of the $BINS$ array, $\mathcal{Z}((p_1, p_2, \ldots, p_k)) = \sum_{i=1}^{k} p_i$. An important observation from the recurrence above is that the set of entries that a particular entry of $BINS$ depends on, all have a smaller value of $\mathcal{Z}$ associated with them; this is because $\mathcal{Z}((q_1, q_2, \ldots, q_k)) > 0$ for all entries $(q_1, q_2, \ldots, q_k) \in Q$. This suggests that we can process the entries of the table $BINS$ in increasing order of their $\mathcal{Z}$ values. That allows us to process all the entries of $BINS$ that have the same $\mathcal{Z}$ value in parallel. Thus we process the entries of $BINS =$ in $n_1 + n_2 + \ldots + n_k$ iterations; in iteration i, we process all the entries for which the associated $\mathcal{Z}$ value is i. This is the same as processing the entries of $BINS$ diagonally as the sum of the indices of all the entries lying along the same diagonal is the same. (see Figure 5 (a) for an illustration in 2 dimensions).

```
For $i = 1$ to $n_1 + n_2 + \ldots + n_k$

    # pragma omp parallel for ...
    For $j = 1$ to $size(\mathcal{Z}ptrlist[i])$
        Retrieve index of $BINS$ entry pointed to by $\mathcal{Z}ptrlist[i][j]$
        Compute DP entry for this index using the DP recurrence
```

Fig. 3. OpenMP parallelization using diagonalization

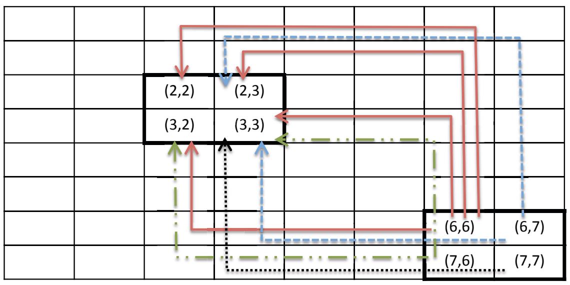


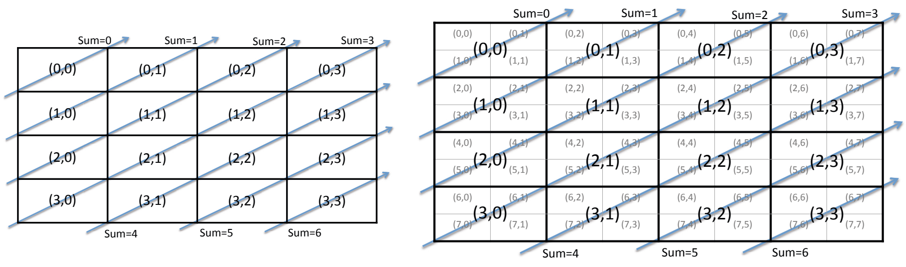
Fig. 4. Dependencies of entries within blocks

Before getting into the dynamic program, we preprocess and create some additional data structures to allow diagonal parallelization of the dynamic program. We create $n_1 + n_2 + \ldots + n_k$ lists (arrays), $\mathcal{Z}ptrlist[i]$, one for each possible value that the function $\mathcal{Z}$ can take. In list i, we store pointers to all the entries of $BINS$ for which $\mathcal{Z}(\cdot) = i$. The DP can now be updated using two loops; the outer loop runs over all the possible values that $\mathcal{Z}$ can take in increasing order and the inner loop runs over all the entries of $BINS$ having the corresponding $\mathcal{Z}$ value and process them as specified by the DP. The inner loop can now be parallelized using OpenMP (See Figure 3).

4.2 Block Decomposition for Cache Efficiency

Blocking is a well-known technique that has been used in optimization of numerical linear algebra implementations on high performance computing platforms. Instead of operating on entire rows or columns of a matrix, blocked algorithms operate on submatrices (called blocks). Operating on blocks leads to improved data locality as the blocks loaded into the faster levels of memory hierarchy result in better data reuse in comparison to the data loaded in case of operations performed on entire rows or columns.

In this section, we present a blocking optimization for our dynamic program and illustrate how it can benefit the performance of the DP. Consider the



(a) (b)

Fig. 5. Processing of (a) *entries* and (b) *blocks* having same sum of indices in parallel

restricted bin-packing problem with 2 weights. Thus, the $BINS$ array in this case is a 2-dimensional array. Suppose that the Q array contains the entries $(4,4)$, $(4,3)$, $(3,4)$ and $(3,3)$. Now consider the dependencies for entries $(6,6)$, $(6,7)$, $(7,6)$ and $(7,7)$. These are illustrated in Figure 4. Entry $(6,6)$ depends on all the four entries $(2,2)$, $(2,3)$,$(3,2)$ and $(3,3)$; entry $(6,7)$ depends on the entries $(2,3)$ and $(3,3)$; entry $(7,6)$ depends on the entries $(3,2)$ and $(3,3)$; entry $(7,7)$ depends on the entry $(3,3)$. Thus if we create blocks of 2×2 in this example, we see that we can benefit from data locality of the entries of $BINS$ that are fetched; this is because the Q array has entries that are packed close together. More precisely, whenever $(q_1, q_2, \ldots, q_k) \in Q$, then all the entries dominated by this entry also belong to Q. For instance in 2 dimensions, if $(4,4) \in Q$, then $(4,3), (3,4), (3,3) \in Q$ as well.

We next discuss how blocks can be used in conjunction with the diagonalization optimization discussed in the previous section. Consider the case of $BINS$ array being 2-dimensional. Figure 5 (b) illustrates the $BINS$ array divided into blocks of size 2×2. We can number the blocks in 2-dimensions as illustrated in the figure. We note that the entries for which the indices of the block numbers are the same are independent of each other. Thus, we can process all the blocks having the same sum of indices in parallel. This is the same as before, except that the sum of indices is now computed over the blocks instead of the elements. However, note that the entries within a block are not independent. Therefore, we need to process them in increasing order of the sum of indices as before.

Note that since in the bin-packing problem the DP can have large number of dimensions, the blocks are themselves multi-dimensional and hence large. For instance, even with a modest value of $k = 6$ (unique item weights), a block size of $2 \times \ldots \times 2$ has $2^6 = 64$ elements. Thus we shall use small block sizes for our DP so that the blocks fit in the L1 cache.

4.3 Optimizing Dependency Lookups

Recall that the dynamic program recurrence is given by

$$BINS(p_1, p_2, \ldots, p_k) = 1 + \min_{\substack{(q_1, q_2, \ldots, q_k) \in Q: \\ q_i \le p_i \ \forall i}} BINS(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k)$$

This typically requires random memory accesses that can be very costly. The size of the Q array can be very large and therefore this operation is very time consuming. In this section, we discuss how we can possibly filter out some of the Q entries in order to reduce the number of memory accesses.

We say that a vector $(p_1, p_2, \ldots, p_k)$ dominates another vector $(p'_1, p'_2, \ldots, p'_k)$ if $p'_i \leq p_i \; \forall \; 1 \leq i \leq k$. We first make a simple observation: if x bins are required to pack $(p_1, p_2, \ldots, p_k)$ items, then we require no more than x bins to pack $(p'_1, p'_2, \ldots, p'_k)$ items, where $p'_i \leq p_i \; \forall \; i$. Note that whenever $(q_1, q_2, \ldots, q_k) \in Q$, then every entry dominated by this entry is also in Q, i.e. $(q'_1, q'_2, \ldots, q'_k) \in Q$ for all entries such that $q'_i \leq q_i \; \forall \; i$. This follows because, if q_1 items of weight w_1, q_2 items of weight w_2, …, q_k items of weight w_k can fit in a bin, then so can any subset of these items. Now, if while computing an entry $(p_1, p_2, \ldots, p_k)$ of $BINS$, the entry $(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k)$ is valid, i.e., $p_i - q_i \geq 0 \; \forall 1 \leq i \leq k$, then we do not need to lookup the entries $(p_1 - q'_1, p_2 - q'_2, \ldots, p_k - q'_k)$, where $(q'_1, q'_2, \ldots, q'_k) \in Q$ is dominated by $(q_1, q_2, \ldots, q_k)$. This is because the number of bins required to pack $(p_1 - q'_1, p_2 - q'_2, \ldots, p_k - q'_k)$ items cannot be anymore than required to pack $(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k)$ items. Thus, while updating an entry of $BINS$, if the lookup for some entry of Q results in a valid entry of $BINS$, then we do not need to perform lookup for the entries of Q that are dominated by this entry (however, we cannot throw away all the dominated entries as lookups for the dominating entry may be an invalid entry of $BINS$).

In order to address this problem, we divide the entries of Q into two parts, called the *primary* and the *secondary* entries of Q. The primary entries correspond to entries that are not dominated by any other entry of Q. The secondary entries correspond to entires of Q that are dominated by some entry of Q. We can now filter the entries of Q as follows. We first determine the minimum by looking up all the primary entries of Q. In case any of these is invalid, then we also lookup the minimum using the secondary entries of Q. If all the primary entries of Q are valid, then there is no need to lookup any secondary entry of Q. The pseudocode for this procedure is illustrated in Figure 6. Note that it may seem that we can design more optimal algorithms for optimizing on all the dependencies amongst the Q entries; however, we emphasize here that what we are trying to save by avoiding one lookup is a single memory access – a more complicated scheme will require maintaining additional information that will also need to be looked up from the memory and result in additional memory accesses. Therefore, we choose to implement a simple strategy that does not require any additional data structures to be maintained.

5 Experimental Evaluation

In this section, we describe experimental evaluation of the proposed parallelization and optimization strategies.

5.1 Experimental Setup

Data. The characteristics of a restricted bin packing problem instance are largely controlled by three properties: (a) number of distinct weights, (b) item-weight

```
Let $numP$ = number of primary entries of $Q$
Let $numS$ = number of secondary entries of $Q$

For $i = 1$ to $numP$
    Let $(q_1, q_2, \ldots, q_k)$ be the $i^{th}$ primary entry of $Q$
    If $(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k) \geq (0, 0, \ldots, 0)$
        Increment $numAccessed$
        Let $\mathcal{X} = BINS(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k) + 1$
        If $\mathcal{X} < BINS(p_1, p_2, \ldots, p_k)$
            $BINS(p_1, p_2, \ldots, p_k) = \mathcal{X}$

If ( $numAccessed \neq numP$ )
    For $i = 1$ to $numS$
        Let $(q_1, q_2, \ldots, q_k)$ be the $i^{th}$ secondary entry of $Q$
        If $(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k) \geq (0, 0, \ldots, 0)$
            Let $\mathcal{X} = BINS(p_1 - q_1, p_2 - q_2, \ldots, p_k - q_k) + 1$
            If $\mathcal{X} < BINS(p_1, p_2, \ldots, p_k)$
                $BINS(p_1, p_2, \ldots, p_k) = \mathcal{X}$
```

Fig. 6. Dependency lookup optimization of the Q array in the DP

generating distribution, (c) number of items. For the current study, we evaluated the different DP variants on synthetic problem instances with varying number of distinct weights $(2, 3, 4, 5, 6, 7, 8)$ and varying number of items $(50, 100, 200, 300)$. We also considered two different distributions (uniform and Zipf) for generating item weights. The weights were chosen to be contiguous.

Hardware Configuration. The experiments were performed on a Dell Precision T7600 system with a 8 core Intel Xeon processor running 64 bit Ubuntu 12.04 LTS. This machine has 20MB cache for all the 2GHz processor cores, supports hyper-threading, and has 256 GB DDR3 RAM. The bin packing code was compiled using GNU gcc compiler with -O3 optimization and openMP was used for shared memory parallelization. To study the benefits of parallelization, each problem instance was solved using varying number of parallel threads $(1, 2, 4, 8, 16)$.

5.2 Results and Discussion

Parallel Speedup with Increasing Number of Distinct Weights. Figure 7 shows the speedup relative to the single thread implementation for problem instances with a fixed (=100) number of items and varying number of distinct weights $(4, 6, 8)$ for both uniform and Zipf distributions. The performance times in this case correspond to the parallelization without any optimizations. The range for the number of distinct weights was chosen so that the problem sizes are large enough to benefit from parallelization. We observe that a larger number of

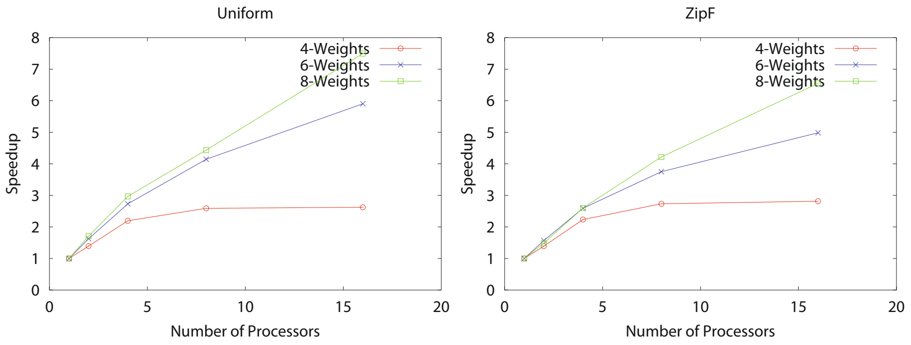


Fig. 7. Speedup with increasing number of distinct weights for a fixed(=100) number of items

distinct weights results in higher speedups for both the distributions. In particular, for the case of 8 distinct weights with uniform distribution, we obtain speedups of around 7.5× with 16 threads. This increased speed-up is to be expected as the problem size becomes exponentially large as we go to higher number of distinct weights. Specifically, when k is the number of distinct weights, the size of parallelizable work, (i.e., the number of blocks in the intersection of the "diagonal" hyperplane of the form $p_1+\cdots+p_k = r$ and the k-dimensional cuboid to be explored) increases exponentially with k allowing more opportunity for parallelization.

Parallel Speedup with Increasing Number of Items. Figure 8 shows the speedup relative to the sequential case for a fixed (=5) number of distinct weights with varying number of items $(100, 200, 300)$ and two distributions (uniform and Zipf). As in the previous case, the performance times correspond to parallelization without additional optimizations. We observe that relative to the previous case with varying number of distinct weights, there is only a modest increase in speedup with increasing number of items. For instance, in case of uniform distribution, with 16 threads and 5 distinct weights , there is a jump in speedup from 4.2× to 4.8× when the number of items goes from 100 to 200. On the other hand as we observed in Figure 7, with 16 threads and 100 items generated using uniform distribution, there is a jump in speedup from 5.8× to nearly 7.8× when the number of distinct weights goes from 6 to 8. The increase in speedup is also much more pronounced in case of uniform distribution than Zipf distribution. This behavior can be explained by the fact that the size of parallelizable work (i.e., the number of blocks in the intersection of the "diagonal" hyperplanes and the k-dimensional cuboid to be explored) increases polynomially with the number of items n in case of uniform distribution. However, in case of Zipf distribution, the cuboid is very skewed in a few dimensions so that the size of the intersection with the diagonal hyperplanes is much smaller as a result of which there is barely any increase in the speedup.

Effect of Optimizations: Next, we study the effect of the proposed optimizations on the performance of the parallel DP. We consider four different variants of the code:

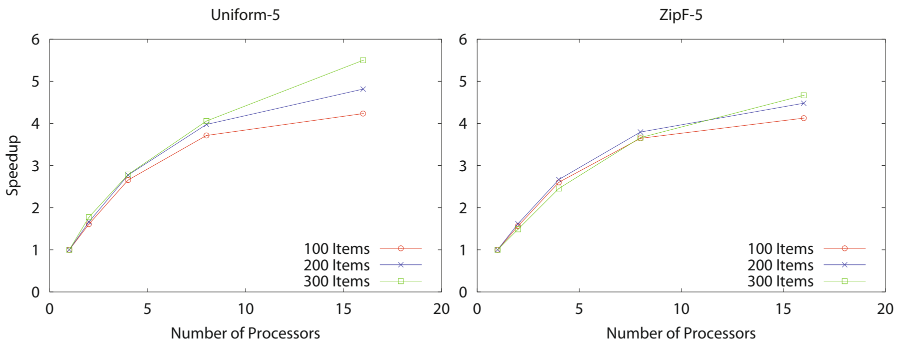


Fig. 8. Speedup with increasing number of items for a fixed(=5) number of distinct weights

(i) `NOOPT`: This is the base code without any optimizations, i.e., the block sizes are $1 \times \ldots \times 1$ and all the dependencies from the Q array are looked up;
(ii) `BLK2`: This is the code with the blocking optimization wherein the block sizes are $2 \times \ldots \times 2$ and all the dependencies from the Q array are looked up;
(iii) `DEP`: This is the code with the optimization wherein the dependencies are split into primary and secondary and the block sizes are $1 \times \ldots \times 1$;
(iv) `DEP-BLK2`: This is the code with both the optimizations, i.e., block sizes are $2 \times \ldots \times 2$ and the dependencies are split into primary and secondary.

Figure 9 shows the performance improvement of the optimized versions relative to `NOOPT` with varying number of threads and a fixed (=100) number of items. The different plots correspond to two different choices $(6, 8)$ for the number of distinct weights and two different (uniform and Zipf) item-weight generating distributions.

In all the four cases, we observe that for a fixed block size, splitting up dependencies into primary and secondary results in a performance improvement, i.e., `DEP-BLK2` is superior to `BLK2` and `DEP` is superior to `NOOPT`. For uniform distribution and 6 distinct weights, `DEP` and `DEP-BLK2` provide roughly $10-20\%$ improvement over `NOOPT` and `BLK2` respectively. The relative improvement seems to be similar for both uniform and Zipf distribution.

From the plots, we also observe the blocking optimization is beneficial to some extent in all the four cases. However, the relative improvement depends on the problem size and the number of parallel threads and even the choice of the distribution. On an average, there is substantial increase in the improvement as the number of distinct weights goes up. For instance, with uniform distribution and 16 threads, we observe a 10% improvement with 6 distinct weights, but nearly 30% improvement with 8 distinct weights. In case of Zipf distribution, the improvements for equivalent scenarios are much smaller, but the trend holds. For a fixed choice of distribution and problem size, we also notice a decrease in improvement due to larger block size as the number of threads increase. This behavior can be readily explained by the fact that larger block sizes result in fewer

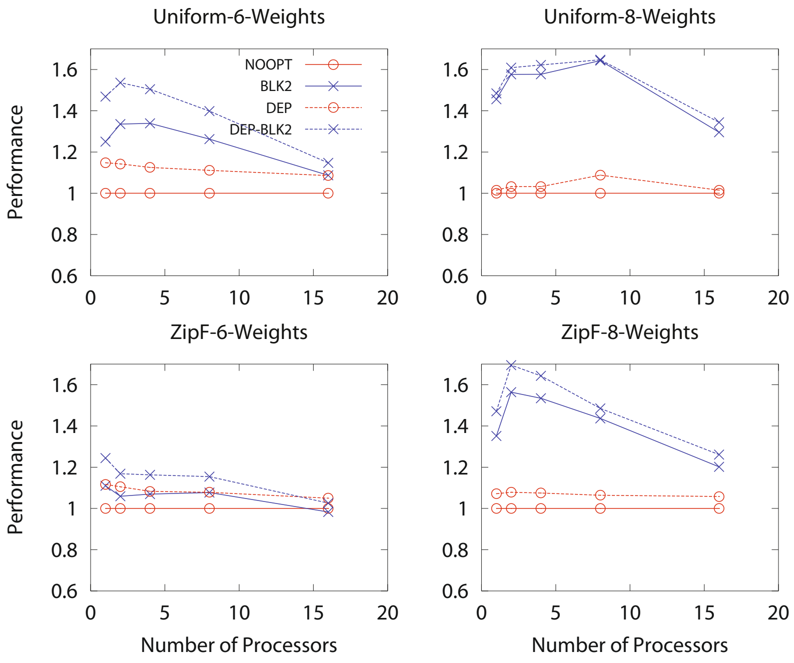


Fig. 9. Performance improvement with the proposed optimizations – blocking and secondary Q array

opportunities for parallelization resulting in poorer performance with increasing number of threads.

Overall, we notice that combining the blocking optimization and dependency partitioning can result in a substantial improvement. For instance, in case of uniform distribution with 8 distinct weights, we observe that `DEP-BLK2` provides up to 65% improvement over `NOOPT` algorithm.

6 Conclusions

We presented a study on shared-memory parallelization of the PTAS dynamic program for the classical bin-packing problem, one of the fundamental problems in combinatorial optimization. Parallelizing multi-dimensional DP where each entry in the DP table depends on variable and potentially large number of other entries is highly challenging. To the best of our knowledge, this paper is the first work that attempts to address this problem for the scenario where the number of dimensions is greater than two. We demonstrated that certain tried and tested techniques such as diagonal traversal and blocking (tiling) perform well. We also propose a novel technique to optimize dependency lookups that arise in the bin-packing problem by partitioning them into two categories- primary and secondary, with the latter ones required only when at least one of the primary lookups is not valid. Experimental evaluation on synthetic data indicates significant performance gains due to the proposed optimizations (up to 65 % in certain cases). There is scope for much more improvement in optimizing these dependency lookups especially in case of distributed memory systems.

References

1. Alves, C.E.R., Cáceres, E., Dehne, F.K.H.A.: Parallel dynamic programming for solving the string editing problem on a cgm/bsp. In: SPAA, pp. 275–281 (2002)
2. El Baz, D., Elkihel, M.: Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem. J. Parallel Distrib. Comput. 65(1), 74–84 (2005)
3. Bradford, P.G.: Efficient parallel dynamic programming (1994)
4. Calvet, J.-L., Viargues, G.: Parallelism in dynamic programming: an issue to feedback control. In: First IEEE Conference on Control Applications, vol. 2, pp. 1063–1064 (1992)
5. Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming algorithms for multicores. In: SPAA, pp. 207–216. ACM (2008)
6. Elkihel, M., El Baz, D.: Load balancing in a parallel dynamic programming multimethod applied to the 0-1 knapsack problem. In: PDP, pp. 127–132. IEEE Computer Society (2006)
7. Huang, S.H.S., Liu, H., Viswanathan, V.: Parallel dynamic programming. IEEE Trans. Parallel Distrib. Syst. 5(3), 326–328 (1994)
8. Karypis, G., Kumar, V.: Efficient parallel mappings of a dynamic programming algorithm: A summary of results. In: IPPS, pp. 563–568 (1993)
9. Kumar, V., Grama, A., Gupta, A., Karypis, G.: Introduction to Parallel Computing. Benjamin/Cummings (1994)
10. Larson, R.E., Tse, E.: Parallel processing algorithms for optimal control of nonlinear dynamic systems. IEEE Trans. Comput. 22(8), 777–786 (1973)
11. Lau, K.K., Kumar, M.J.: Parallel implementation of the unit commitment problem on nows. In: High Performance Computing on the Information Superhighway, HPC Asia 1997, pp. 128–133 (1997)
12. Lewandowski, G., Condon, A., Bach, E.: Asynchronous analysis of parallel dynamic programming. In: SIGMETRICS, pp. 268–269 (1993)
13. Rodríguez, C., Roda, J.L., García, F., Almeida, F., González, D.: Paradigms for parallel dynamic programming. In: EUROMICRO, p. 553. IEEE Computer Society (1996)
14. Smith, T.F., Waterman, M.S.: Identification of common molecular subsequences. Journal of Molecular Biology 147(1), 195–197 (1981)
15. Tan, G., Feng, S., Sun, N.: Biology - locality and parallelism optimization for dynamic programming algorithm in bioinformatics. In: SC, p. 78. ACM Press (2006)
16. Tan, G., Sun, N., Gao, G.R.: A parallel dynamic programming algorithm on multicore architecture. In: SPAA, pp. 135–144. ACM (2007)
17. Vazirani, V.V.: Approximation algorithms. Springer (2001)
18. Williamson, D.P., Shmoys, D.B.: The Design of Approximation Algorithms. Cambridge University Press (2011)
19. Xu, H.H., Hanson, F.B., Chung, S.L.: Optimal data parallel methods for stochastic dynamical programming. In: ICPP (3), pp. 142–146 (1991)