Parallelization of Bin Packing on Multicore Systems

Sayan Ghosh and Assefaw H. Gebremedhin
School of Electrical Engineering and Computer Science
Washington State University
Pullman, WA, USA
{sghosh1,assefaw}@eecs.wsu.edu

***Abstract*—We study effective parallelization of approximation algorithms for the one-dimensional *bin packing* problem on a multicore platform. Bin packing is a classic combinatorial optimization problem that aims to pack a given sequence of items into a minimum number of equal-sized bins. The problem potentially serves as a model for a wide variety of applications. Examples include: packing data into chunks in a memory hierarchy in a given system to increase application performance, loading vehicles subject to weight limitations, and packing TV commercials into station breaks. Bin packing has long served as a proving ground for the analysis of approximation algorithms and played a crucial role in the development of much of the theory of approximation algorithms. Its parallelization, however, has received comparatively much less attention. In this work, we develop multiple parallel versions of an effective approximation algorithm (First Fit Decreasing) for the problem and investigate the trade-off between solution quality and execution time. We use OpenMP and Cilk Plus as mechanisms for achieving the parallelization. The new parallel algorithms obtain a speedup of more than 10x (on 32 cores) for moderate to large input sequences without sacrificing much on the quality of solution produced by the sequential algorithm — in particular, we see only about 3 to 30% increase in the number of bins compared to the sequential version. In turn, the solution obtained by the sequential First Fit Decreasing algorithm is provably almost optimal (the approximation ratio is less than 1.3).**

Keywords*—*Bin packing; First Fit Decreasing; OpenMP; Combinatorial optimization problems

I. Introduction

In the classical one-dimensional bin packing problem, we are given a sequence $L = (a_1, a_2, \ldots, a_n)$ of n items, each with a size $s(a_i) \in (0, 1]$, and are asked to pack the items into a minimum number of unit-capacity bins. Stated in other words, the problem is that of partitioning the sequence L into a minimum number m of subsets $B_1, B_2, \ldots, B_m$ such that $\sum_{a_i \in B_j} s(a_i) \leq 1, 1 \leq j \leq m$. The problem is known to be NP-hard [9] [13], and has many real-world applications. Examples include: packing data into chunks in a memory hierarchy in a given system to increase application performance, loading vehicles subject to weight limitations, and even packing TV commercials into station breaks [8], [11], [12], [17], [24]. As mentioned in [1], [8], bin packing can also model other types of computer science problems, including scheduling of tasks efficiently on multiple processing elements. For instance, in [14], a bin packing algorithm is used for load balancing in the Multi-Zone benchmark, which is a part of the NAS Parallel benchmarks (NPB) suite [3]. A similar use of bin packing can be envisioned in contemporary High Performance Computing (HPC) nodes, where there is large heterogeneity due to co-existence of different processing cores (for instance, compute nodes with GPU and general purpose processors).

Bin packing has also served as early playground for the development of much of the theory of approximation algorithms. These include determining worst-case approximation ratios [16], identifying lower bounds on the best possible online performance [18], and analyzing average-case behavior [22].

Two of the most well-studied variants of approximation algorithms for bin packing are the First Fit (FF) and Best Fit (BF) algorithms. The FF algorithm places each item, in progression, into the first bin in which it fits. The BF algorithm places each item, in progression, into the tightest (most nearly full) bin in which it fits. Both FF and BF have the same guaranteed approximation ratio of $17/10$, although the two can give strikingly different packings for individual lists [9].

To reduce the approximation ratio, in some cases, the data items are sorted by their size before packing them into bins. One variant here is to pack the largest items first. This variant of FF is called *First Fit Decreasing* (FFD), and the corresponding variant of BF is called *Best Fit Decreasing* (BFD). Both FFD and BFD algorithms have the same approximation ratio of $11/9$ [16].

Despite the differences between the FF and BF algorithms, it has been proven that there is no difference in worst and best case behaviors (or time complexities) between the two algorithms [16]. The same statement is also true of their ordered counterparts, FFD and BFD.

Although comparatively much less intensively than serial bin packing, parallel bin packing has been studied in the past [1], [4], [5], [9]. The research has proven FFD to be P-complete, indicating that the algorithm is hard to efficiently parallelize. In this work, we study a number of approaches for parallelizing bin packing algorithms within the general framework of directive-based parallelization (we employ OpenMP [10] and Intel® Cilk Plus™ [19] for the parallelizations). Our contributions are summarized as follows:

- We develop multiple strategies for parallelizing the FFD algorithm for bin packing.
- We investigate the impact of common *data* and *task* parallelism directives/keywords available in OpenMP and Cilk Plus for tuning the parallel algorithms.
- We experimentally evaluate the performance of the parallel algorithms and study the trade-offs between execution time and solution quality the algorithms

offer using test instances designed to cover a wide spectrum of input types.

Our general "take-home" finding is that our parallel bin packing implementations obtain a speedup of more than 10x (on 32 cores) for moderate to large inputs without significantly sacrificing the quality of the solution obtained by the sequential FFD algorithm. In addition, our experimental evaluation using the data/task parallel functionalities provided by OpenMP/Cilk Plus will be valuable to algorithm developers faced with the design of effective parallel algorithms for similar combinatorial problems, which generally are characterized as being irregular, memory-intensive, and low-concurrency computations — and hence challenging to parallelize.

II. Methodology and Background

In order to evaluate the quality of solution produced by the various parallel bin packing algorithms we investigate, we need a way to generate reproducible numbers (integers) within a range. By reproducible we mean that the generated numbers should be the same across software or hardware platforms. To that end, we adopted the bin packing data generation (BPPGEN) strategy proposed by Schwerin and Wascher [21]. We begin this section with a discussion of the BPPGEN model and our adoption of its implementation (Section II-A). We then review the sequential FFD algorithm that underlies all of our parallel implementations (Section II-B).

A. Bin packing data generator (BPPGEN)

BPPGEN generates data suitable for the bin packing problem by taking in four parameters:
– bin capacity (C),
– number of items to be packed (n),
– lower bound on item weights (v_1), and
– upper bound on item weights (v_2).
The bounds v_1 and v_2 are multiplied with the bin capacity C to obtain the range of items. For example, if $C = 1000$, $n = 20$, $v_1 = 0.2$ and $v_2 = 0.7$, then item weights would be uniformly distributed between $v_1 * C = 200$ and $v_2 * C = 700$. A class of problem instance is characterized by the tuple (C, n, v_1, v_2). A Fortran code that generates data using the BPPGEN model described above is given in the appendix of [21]. We translated the code to C programming language in our work (as our programs are written in C), and it is in this sense we say we adopted the model.

We conclude this brief subsection with a quick side note. One of the well known candidate exact algorithms for bin packing is the Martello-Toth Packing (MTP) algorithm [18]. Meanwhile, Schwerin and Wascher [21] argue that the problem instances that were used to evaluate MTP are not adequate, and they show that for a significant number of problem classes generated using the generator BPPGEN, optimal solution was not obtained by MTP.

B. Base serial algorithm

The sequential FFD algorithm that underlies all our parallel algorithms is outlined in Algorithm 1 (SerialFFD). The item list is sorted in decreasing order, hence larger items are packed first on to the bins. Many of the notations used in Algorithm 1 are also used in other algorithms throughout the paper. We summarize the notations here for easy reference:
– $L[1, \ldots, n]$ denotes the list of n items to be packed,
– $s(a_i)$ denotes the size of each item a_i,
– $B[1, \ldots, M]$ denotes the bins,
– C denotes the bin capacity,
– the variable BC_j stores the current capacity (available space) of the jth bin.
Note that M is an upper bound on the optimal number of bins m (since the value of m cannot be known a priori).

Algorithm 1 SerialFFD: Serial implementation of FFD.
Input: List L of items $L = (a_1, a_2, \ldots, a_n)$, bin capacity C.
Output: Bins $B_1 \ldots B_{alg}$, where alg is the number of bins used by the algorithm. The variable BC_j holds the current capacity of the jth bin. M is an upper bound on alg.

```
1:  Initialize: BC_j ← C, j = 1, ..., M
2:  Initialize: B_j ← ∅, j = 1, ..., M
3:  Initialize: alg ← 0
4:  for each a_i ∈ L do
5:     for j = 1, ..., M do
6:        if s(a_i) < BC_j then
7:           B_j ← B_j ∪ {a_i}
8:           BC_j ← BC_j − s(a_i)
9:           if j > alg then
10:             alg ← j
11:          break
```

Algorithm 1 clearly has a worst-case complexity of $\mathcal{O}(n^2)$, where n is the number of items to be packed on to bins. The quadratic time complexity is due to the simple data structure (array) used to represent the bins. It is possible to reduce the complexity to $\mathcal{O}(n \log n)$ by using a tree-based data structure.[1]

III. Parallel Algorithms

A. Overview

Broadly, we distinguish between two aspects of our work: *design* of parallel versions of algorithms for the FFD bin packing algorithm and *implementation* of the parallel algorithms using existing shared memory programming models. In terms design, we have developed two classes of parallel algorithms: *direct* and *speculative*. In terms of implementation, we use OpenMP and Cilk Plus, and categorize our approaches as *data parallel* and *task parallel*. In general, our implementations parallelize the loop over a list of ordered items (Line 4 in Algorithm 1) using the worksharing or tasking constructs in OpenMP and Cilk Plus. Using worksharing constructs to divide loop iterations among threads corresponds to the data parallel approach, while spawning distinct tasks and scheduling them on the available threads, as effected by tasking constructs in OpenMP/Cilk Plus, corresponds to the task parallel approach.

The direct algorithms obtain a consistent solution with a single attempt at the expense of potentially lower concurrency. The speculative algorithms attempt to maximize concurrency at the expense of increased work; they operate by initially getting a potentially incorrect solution (in the sense that some bins are overfull) and then detecting any inconsistencies and rectifying them afterwards, iteratively.

[1] Readers interested in the details here are referred to the seminal paper of D. S. Johnson on fast algorithms for bin packing [15] (specifically Section 5).

Fig. 1. An overview of our parallel FFD implementations

```
Parallel FFD implementations
├── Data Parallel
│   ├── Direct
│   │   ├── OpenMP-Dynamic-Schedule (DIRECTDYNAMIC)
│   │   ├── OpenMP-Static-Schedule-Stride-N
│   │   │   (DIRECTSTATIC)
│   │   ├── OpenMP-Static-Schedule-Stride-1
│   │   └── Cilk-For
│   └── Speculative
│       ├── Speculate-Iterate (SPECULATEITERATE)
│       └── Speculate-Correct
└── Task Parallel
    ├── OpenMP-Dynamic-For-Task{tied/untied}
    ├── OpenMP-Dynamic-Single-Task{tied/untied}
    └── Cilk-Spawn
```

The direct approaches conform to a more straightforward usage of data/task parallel constructs available in OpenMP/Cilk Plus. We therefore prefix our notations for these algorithms with `OMP` or `CILK`, such as `OMP-DYN` and `CILK-FOR`. Figure 1 provides an overview of the parallel FFD algorithms we have developed, based on the characterizations discussed in the previous two paragraphs. The algorithms will be described in more detail in the remainder of the current section.

B. Data Parallel: Direct

Distribution of the worksharing loop iterations to threads is controlled via the "schedule" clause in OpenMP. We explore two algorithms under direct parallelization, one that employs a *dynamic* scheduling policy (iterations are distributed to threads as and when they request them) and another that uses *static* scheduling (iteration space is divided into equal sized chunks and at most one chunk is assigned to a thread). Static scheduling in OpenMP is quite useful in maximizing cache reuse, as the same threads work on the same set of data in the subsequent iterations. In Cilk Plus, the loop scheduling follows a divide-and-conquer strategy, in which loop iterations are divided recursively and executed in parallel. Cilk Plus does not support an OpenMP equivalent of "static" scheduling for mapping fixed iterations to threads.

1) Dynamic scheduling: Algorithm 2, which we call DIRECTDYNAMIC, outlines the approach that inserts an item into a bin using dynamic scheduling to distribute iterations among threads. Although Algorithm 2 is presented with explicit OpenMP directives, the Cilk Plus version works in exactly analogous manner; the needed modification is to replace the `for` loop over items (in Line 6) with a `cilk_for` extension, and ensure atomic accesses to bin capacity variable (in Line 10 and Line 18)[2].

In Algorithm 2, each thread executes a loop searching for a bin to insert an item (Lines 8-18). The variable `done` is set to `true` when an item is finally inserted into a bin (Line 15). Comparing with the serial algorithm (Algorithm 1), one

[2]Keywords for atomic/locking semantics are absent from Cilk Plus. Therefore, to protect concurrent accesses to the bin capacity variable (BC_j in Algorithm 2), we used Intel atomic intrinsics (`__sync_add_and_fetch` and `__sync_sub_and_fetch`) for atomically adding and subtracting bin capacity.

Algorithm 2 DIRECTDYNAMIC: Parallelization of SERIALFFD using OpenMP `schedule(dynamic)`.
Input: List L of items $L = (a_1, a_2, \ldots, a_n)$, bin capacity C.
Output: Bins $B_1 \ldots B_{alg}$. Other variables as in Algorithm 1. Variables BC, B and L are shared among threads, variable `done` is private. Variable `binCount` keeps track of the number of bins used by the algorithm.

```
 1: Initialize: BC_j ← C, j = 1,...,M
 2: Initialize: B_j ← ∅, j = 1,...,M
 3: Initialize: binCount ← 0
 4: #pragma omp parallel for schedule(dynamic)\
 5: reduction(max:binCount)
 6: for each a_i ∈ L do
 7:   Initialize: done ← false
 8:   for j = 1,...,M and not done do
 9:     #pragma omp atomic capture
10:     b ← BC_j ← BC_j − s(a_i)
11:     if b >= 0 then
12:       B_j ← B_j ∪ {a_i}
13:       if j > binCount then
14:         binCount ← j
15:       done ← true
16:     else
17:       #pragma omp atomic update
18:       BC_j ← BC_j + s(a_i)
19: alg ← binCount
```

can see that in Algorithm 2, we atomically decrement and increment the shared variable `BC` (Line 10 and Line 18). This determines whether a particular bin has available space to hold the item. We check if `BC` is greater than zero (Line 11: first, atomically decrement `BC`, and then capture the current value in variable `b`) because multiple threads could have tried to put items on a particular bin, and we allow an item to be inserted in a bin only if there is enough space on the bin to store it.

The atomic construct with the `capture` clause forces an atomic update of the location while also capturing the original or final value of the location. Usage of `seq_cst` subclause with an OpenMP `atomic capture` construct might be necessary, to ensure an implicit flush operation at the end of an atomic operation (in order for other threads to see the updated value).

We then check if the item were to be inserted on a bin (Line 9 and 10), whether the bin will exceed its capacity (Line 11). If we are not going beyond the capacity of the bin, then we continue with the insertion (Lines 12-15), else we skip considering that particular bin (Lines 17-18).

We point out a key difference in the Cilk Plus version of this algorithm. The Cilk Plus scheduler allows "work-stealing", which enables an idle thread to *steal* part of the work from another thread. Initially, a single thread starts executing the entire set of iterations, then an idle thread steals half of the work of the original thread, and another idle thread steals half of the work from the previous thread. This process continues until all the available threads have stolen enough work to execute the loop iterations in parallel.

2) Static scheduling (OpenMP): To prevent simultaneous access to shared variables from multiple threads, atomic operations were required in DIRECTDYNAMIC. Schweizer et al. [20] emperically measure the cost of atomic operations on a number of CPU architectures, and, report that atomic instructions are at least about 5 to $10ns$ slower than reads on Intel® Haswell™ (our evaluation platform). Therefore,

Algorithm 3 DIRECTSTATIC: Direct parallelization of SERIALFFD using OpenMP `schedule(static)`.
Input: List L of items $L = (a_1, a_2, \ldots, a_n)$, bin capacity C.
Output: Bins $B_1 \ldots B_{alg}$. Other variables as in Algorithm 1. Variables `done`, `tid` and `tnum` are private.

```
1: Initialize: BC_j ← C, j = 1,...,M
2: Initialize: B_j ← ∅, j = 1,...,M.
3: Initialize: binCount ← 0
4: #pragma omp parallel for schedule(static) \
5: reduction(max:binCount)
6: for each a_i ∈ L do
7:   Initialize: done ← false
8:   tid ← omp_get_thread_num()
9:   tnum ← omp_get_max_threads()
10:  while not done do
11:    if BC_tid >= s(a_i) then
12:      B_tid ← B_tid ∪ {a_i}
13:      BC_tid ← BC_tid + s(a_i)
14:      if tid > binCount then
15:        binCount ← tid
16:      done ← true
17:    else
18:      tid ← tid + tnum
19: alg ← binCount
```

using static scheduling, we devised a variant of FFD that avoids the need for atomic statements, while recognizing the fact that the static-scheduling version may increase the overall number of bins used. We refer to the variant that uses OpenMP static schedule to distribute fixed-size iterations among threads as DIRECTSTATIC (outlined in Algorithm 3). As mentioned earlier, there is no equivalent of a static schedule in Cilk Plus; therefore, the discussion in this section pertains only to the OpenMP data parallel versions of the algorithms.

In DIRECTSTATIC, we ensure that multiple threads do not access the same bin by associating different bin indices with each thread. As a result, the problem becomes embarassingly parallel. In particular, we avoid the possibility of multiple threads accessing the same bin by enforcing that threads only access bins indexed by a given thread number, as shown in Line 8 of Algorithm 3.

We assume a thread index (tid in DIRECTSTATIC) does not exceed the number of available bins, since the number of bins is typically much higher than the number of threads (at least for our evaluation platform). Since bins are of finite capacity, if an item cannot be placed at a bin indexed by a thread number, a stride of maximum number of threads in the OpenMP parallel region ($tnum$ in DIRECTSTATIC) is added to the current thread index (Line 18). This becomes the new bin index in which the item is inserted. Although letting threads access different bins greatly reduces execution time, it also results in an increase in number of bins used. As an attempt to reduce the number of bins used by this approach, we devised another variant of DIRECTSTATIC that combines the essence of DIRECTDYNAMIC and DIRECTSTATIC algorithms.

Like DIRECTSTATIC, in this particular variant each thread access a bin indexed by the thread number, but as the bin becomes full, instead of going to the next bin at a stride of `tnum` (Line 18), it goes to the very next bin, by incrementing `tid` by 1 (i.e., `tid ← tid + 1` instead of `tid ← tid + tnum`). We observe greatly reduced number of bins as a result of this modification compared to DIRECTSTATIC (see experimental results in Section IV-B).

Algorithm 4 SPECULATEITERATE: Iterative speculation and correction implementation.
Input: List L of items $L = (a_1, a_2, \ldots, a_n)$.
Output: Bins $B_1 \ldots B_{alg}$. Other variables as in Algorithm 1. `avoidBins` is a vector of booleans used to indicate whether or not a bin is over capacity.

```
1: Initialize: BC_j ← C, j = 1,...,M
2: Initialize: B_j ← ∅, j = 1,...,M
3: Initialize: L' = L
4: Initialize: avoidBins_j ← false, j = 1,...,M
5: while L' ≠ ∅ do
6:   binCount ← INSERTITEMSINPARALLEL(L')
7:   CHECKANDCORRECT(L', binCount) {L' gets updated}
```

C. Data Parallel: Speculative

The second class of data parallel algorithm we considered is speculative. Here, for the purpose of maximizing the items that can be processed concurrently, we allow for the possibility of a bin being overfull, tentatively. The general approach is outlined in SPECULATEITERATE (Algorithm 4).

SPECULATEITERATE consists of two phases. In the first phase, items are inserted into the bins from multiple threads using OpenMP worksharing clause for loop parallelization. To reduce the chances of overfilling a bin, and also contention among threads, in each thread, the location of a bin is determined by the thread number (like DIRECTSTATIC). Due to usage of a static schedule for distributing loop iterations among threads, we cannot use Cilk Plus for implementing this approach. As mentioned earlier, Cilk Plus implements a work-stealing scheduler, which does not allow mapping threads to particular loop iterations. The discussion of SPECULATEITERATE therefore pertains to only the usage of OpenMP.

At the end of the first phase of Algorithm 4), that is, the function INSERTITEMSINPARALLEL, there may be some overfull bins (as items are inserted in bins without knowing the latest bin capacity status, unlike direct algorithms). The second phase, CHECKANDCORRECT, outlined in Algorithm 5, fixes this issue. In CHECKANDCORRECT, every bin's current capacity is compared to the maximum capacity, and items are removed from the overfull bins and added to a buffer. The buffer is then passed to INSERTITEMSINPARALLEL in the next iteration. This iteration continues until the buffer is empty, which indicates that all items are inserted into the bins and no bin is overfull.

CHECKANDCORRECT does not need to inspect all the available bins in every iteration for identifying overfull bins, since only the number of bins that INSERTITEMSINPARALLEL used for item insertion is required. Hence, INSERTITEMSINPARALLEL returns the number of bins used in inserting items at a given iteration in the variable `binCount`, which is passed to CHECKANDCORRECT. Since in INSERTITEMSINPARALLEL, each thread access bins that are separated by a fixed stride, and each iteration (in which an item is inserted into a bin) is statically distributed in a round-robin fashion, `binCount` will always return the correct number of bins used in a particular iteration to insert every item in L' (Line 6 in in Algorithm 4). Bins which get overfull need to be flagged off, to avoid insertion and removal of items from same bins over successive iterations, causing an infinite loop. During the checking phase of an iteration, CHECKANDCORRECT flags off a bin (Line 6 in Algorithm 5) if it finds it to be overfull.

Algorithm 5 CHECKANDCORRECT: Check if a bin is overfull, flag such a bin and remove items from overfull bins to create an updated L'. Other variables as in Algorithm 1. Variables BC, B, L' and `avoidBins` are shared among threads. Variable `C` is firstprivate. `binCount` is the count of bins, calculated in INSERTITEMSINPARALLEL.

```
1: #pragma omp parallel for
2: for j = 1, ..., binCount do
3:   #pragma omp atomic read
4:   b ← BC_j
5:   if b < 0 then
6:     avoidBins_j ← true
7:     while b < 0 do
8:       a ← pop an item from B_j
9:       B_j ← B_j \ {a}
10:      #pragma omp atomic capture
11:      b ← BC_j ← BC_j + s(a)
12:      L' ← L' ∪ {a} {queue item for future insertion}
```

Before inserting an item in a bin, INSERTITEMSINPARALLEL checks whether a bin was flagged off in Algorithm 5. Since items are ordered in a decreasing order, items toward the base of a bin are larger in size than items toward the top, we pop items from the top, to ensure that bins that are flagged off have capacity closer to the maximum capacity of a bin.

In INSERTITEMSINPARALLEL, like Algorithm 3, threads access different bins, and therefore there is no need for an `atomic` construct. The only difference is that in INSERTITEMSINPARALLEL, there is a check to ensure bins that were overfull at a particular iteration are not considered again in a subsequent iteration.

Over successive iterations, the number of items listed for re-insertion will get reduced, hence it could be beneficial to perform the task of re-inserting in serial after a threshold on number of items is reached. This is something we want to explore in future for SPECULATEITERATE. Further, it is possible that the number of bins that are overfull at the end of the first phase of the very first iteration of Algorithm 4 is relatively small that the iterative approach may not be worthwhile (in terms of parallel efficiency) for some input sequences.

In order to address these issues, we considered a specialized variant of SPECULATEITERATE where we at the end of the first phase, instead of iterating over the items collected from overfull bins in parallel, we handle re-insertion of those items sequentially. We call this version SPECULATECORRECT. It consists of three steps – the first two are parallel and the last is serial. In Step 1, like Algorithm 4, insertions are done in parallel (as in INSERTITEMSINPARALLEL). Step 2 is similar to Algorithm 4, wherein CHECKANDCORRECT is executed to reorganize overfull bins. The items to be inserted in new bins are identified in Step 2. These are inserted into suitable bins in serial in the final Step 3. Therefore, unlike SPECULATEITERATE, SPECULATECORRECT finishes inserting all the items in three steps in just one iteration. Although we obtained good performance from SPECULATECORRECT, the existence of a serial portion affects scalability for large input sequences on large number of threads. See Section IV (experimental results) for details.

D. Task Parallel: Direct

Instead of distributing iterations of the outermost loop in DIRECTDYNAMIC (Line 6: Algorithm 2) using `#pragma omp parallel for` directive in OpenMP or Cilk Plus's `cilk_for` keyword, an alternate strategy that could potentially improve the overall load balance is to generate a number of tasks which are dynamically executed by the available threads. We considered this approach in both the OpenMP and Cilk Plus contexts.

1) OpenMP: Tasks were introduced in OpenMP 3.0 to exploit irregular parallelism [2]. In the case of FFD, our underlying algorithm, the unit of work varies with the item being processed, which fits into the tasking model well. Each task corresponds to Lines 8-18 of Algorithm 2 (DIRECTDYNAMIC), a piece of code that searches through bins to insert an item. OpenMP task construct could be annotated with multiple clauses, to specify scope of data used in a tasking region, and also to control some aspects of the generated task. In OpenMP, by default, tasks are "tied" – once it starts executing on a particular thread, it will never suspend and resume execution on a different thread. Untied tasks are less restrictive and help load balancing for some algorithms. Untied tasks are created when the `untied` clause is specified with OpenMP task construct.

A common pattern of OpenMP tasking usage is when the master thread or a single thread in the thread team generates all the tasks. In our case such a task is created for every iteration of the outermost for loop of DIRECTDYNAMIC, which loop over all the items. This is represented by the following pseudocode, in which tasks are created by one thread (the *producer*), and are processed by multiple *consumer* threads.

```
#pragma omp parallel
for each item in L
#pragma omp single nowait
   #pragma omp task
   // task per item
```

For a large sequence of items, the number of tasks created could be quite large. There is an overhead associated with OpenMP task creation, hence we consider an alternate approach of processing multiple items (i.e., multiple iterations of the for loop over items) as a task. Since Non-Uniform Memory Access (NUMA) is ubiquitous, getting favorable performance on present day multicore systems require good data and thread affinity. Terboven et al. [23], Broquedis et al. [6] and, Chapman et al. [7] discuss the challenges of thread and data placement on NUMA systems. Rather than any one thread adding the tasks to its pool (and other threads fetching the tasks from that thread), an alternate design would entail multiple threads adding tasks to their individual pool. Since task creation is parallelized, the overhead of task creation increases slowly with the number of threads, as opposed to the other case where a single thread generates all the tasks. This is shown in the following pseudocode:

```
#pragma omp parallel shared
#pragma omp for
for each item in L
   // multiple items per task
```

```
#pragma omp task
DirectDynamic (item)
```

As a result, multiple iterations will constitute a task, instead of a task per iteration as in the case when a single thread generates all the tasks. This could result in better load balancing, as there are many more items than there are threads in the parallel region.

2) Cilk Plus: In Cilk Plus, the `cilk_spawn` keyword spawns a task. Each thread maintains a *dequeue* [3], and when a `cilk_spawn` is encountered, the task is pushed to the back of the dequeue. When a thread reaches a `cilk_sync` keyword, it pops a task from the back of its dequeue and executes it. The divide-and-conquer mechanism implemented by the Cilk Plus scheduler mentioned earlier governs the distribution of tasks to threads. When a thread find its queue to be empty, it steals tasks from the front of another thread's dequeue. The task parallel version of FFD using Cilk Plus looks like the following pseudocode, where `find_bin_for (item)` is a function that resembles the logic (without the OpenMP clauses of course) described in Lines 7-18 of Algorithm 2 (DIRECTDYNAMIC):

```
for each item in L
      cilk_spawn find_bin_for (item)
cilk_sync
```

In general (for both OpenMP and Cilk Plus), creation of too many tasks has overheads, especially when the tasks are not too fine-grained. In contrast, if tasks are too fine-grained, then the time required for spawning a task may be significantly larger than actually performing the work associated with a task. We wanted to empirically measure the impact of explicit tasks in comparison with the data parallel versions, which is the reason why we performed this exercise. Our findings are presented in the next section.

IV. EXPERIMENTAL RESULTS

Test Platform: The experiments were performed on a 2-way 36-core compute node with Intel® Haswell-EP™ Xeon® CPU (version E5-2699 v3) having 2.30GHz frequency, with 128 GB of main memory and two 45 MB of L3 cache shared by 36 cores. Each core has 256 KB private L2 cache, and 64 KB L1 cache. Intel® 16.0 compiler was used for compiling all the programs, with `-O3 -xHost` as compilation options.

Implementations Studied: We summarize below notations we use for the different parallel FFD algorithms we studied. These notations are used as key-legends in the plots presented in this section (Figures 2 – 4).

1) `SERIAL`: SERIALFFD – Algorithm 1.
2) `OMP-DYN`: DIRECTDYNAMIC – Algorithm 2.
3) `OMP-STA-STR-N`: DIRECTSTATIC – Algorithm 3. `STR` stands for stride; DIRECTSTATIC uses a stride of number of threads in the parallel region to determine next bin (Section III-B2).

[3] A *double-ended queue* is a specialized queue where items can be pushed or popped from the front (head) or the back (tail).

TABLE I. CLASSES OF DATA SIZE FOR EXPERIMENTATION - MINI, SMALL AND LARGE

Keys	Number of items	BPPGEN parameters (C, n, v_1, v_2)
mini	200	$\{1000, 200, 0.2, 0.7\}$
small	2,000/20,000	$\{50000, 2000, 0.0002, 0.2\}$/ $\{500000, 20000, 0.0002, 0.2\}$
large	200,000/400,000/ 800,000	$\{50000000, 200000, 0.0002, 0.2\}$/ $\{50000000, 400000, 0.0002, 0.4\}$/ $\{50000000, 800000, 0.0002, 0.8\}$

4) `OMP-STA-STR-1`: similar to DIRECTSTATIC, but uses a stride of 1 to determine the next bin (Section III-B2).
5) `CILK-FOR`: Cilk Plus data parallel version of Algorithm 2 (Section III-B1).
6) `ITER-SPEC-CORRECT`: SPECULATEITERATE – Algorithm 4.
7) `SPEC-CORRECT`: SPECULATECORRECT is a special case of SPECULATEITERATE in which items are collected from overfull bins and replaced in serial (Section III-C).
8) `OMP-FOR-TASK(-UTD)`: A task constitutes inserting multiple items. `UTD` denotes *untied* tasks (Section III-D). Absence of `UTD` indicates that tasks are *tied* to a particular thread (Section III-D).
9) `OMP-SNG-TASK(-UTD)`: A task corresponds to insertion of a single item (Section III-D).
10) `CILK-SPAWN`: Cilk Plus tasking version of Algorithm 2 (Section III-D).

Test Data: The test data was generated using BPPGEN (discussed in Section II-A). We generated three classes of data for our experiments; these are listed in Table I. The numbers are rounded to the nearest whole number.

Results Roadmap: We report two sets of results. The first set consists of results on execution times or speedup (of direct/speculative algorithms) compared to serial as well as results on OpenMP/Cilk Plus using single thread. We also include in the first set single-threaded results for both OpenMP and Cilk Plus, to determine respective runtime overheads. The OpenMP single-threaded version (OMP-1-THREAD) is essentially OMP-STA-STR-1 executed with one thread. We wanted to choose a simple (embarrassingly parallel) algorithm to show OpenMP overheads, hence we chose OMP-STA-STR-1 as a benchmark for single-threaded performance in this case. In Cilk Plus, the execution time of a program using 1 worker thread is referred as the *spawn overhead*, which we denote as CILK-1-THREAD.

The second set of results consists of results on the number of bins used by the parallel algorithms, and comparisons with the serial version (i.e., SERIALFFD).

A. Execution time

1) Results on Data Parallel Implementations: We compare the execution time performance of the data parallel FFD algorithms in Figure 2. We consider bin packing instances of 0.2K, 2K, 20K, 200K, 400K and 800K items (where $K = 10^3$) in these experiments. We observe very different results for packing smaller set of items (0.2K, 2K and 20K) compared to larger set of items (200K, 400K and 800K), when using serial/ OpenMP/Cilk Plus multithreaded and single-threaded versions.

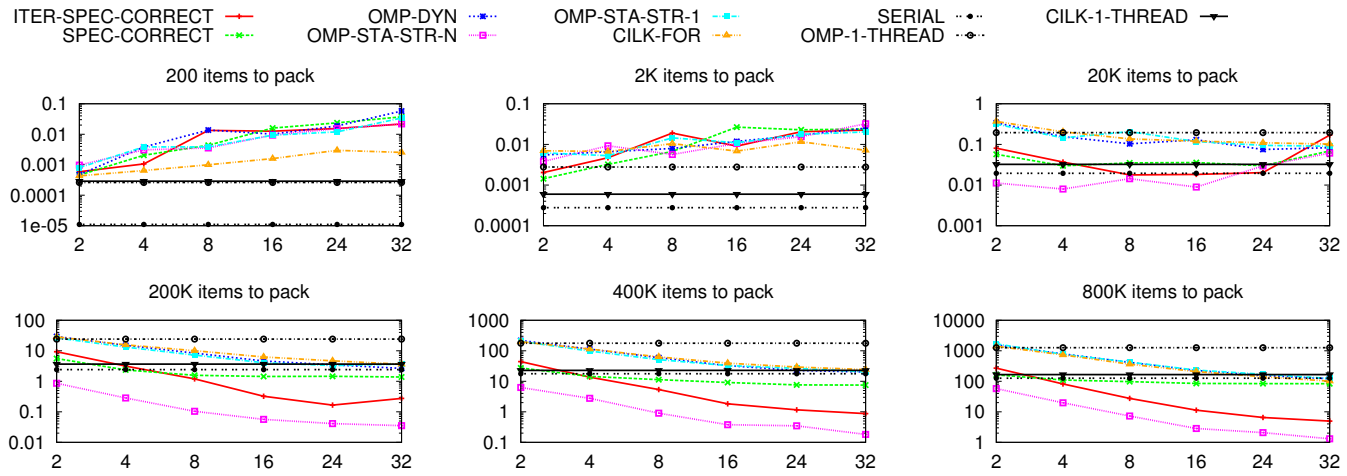

Fig. 2. Execution time of data parallel bin packing algorithms for small (top row) and large (bottom row) input sequences. **X-axis**: Number of threads. **Y-axis**: Execution time for packing items (in secs) in log-scale.

This is not surprising, because when the work is less (for mini and small data sizes), the overheads of multiple threads contending for shared resources affect overall performance. Also, OpenMP single-threaded performance was found to be about 8-10x that of the serial version. We speculate that the presence of OpenMP pragmas may inhibit some compiler optimizations, which may be the reason behind the poorer performance of the OpenMP single-threaded version compared to the serial version. In contrast, the Cilk Plus single-threaded performance was found to be close to the serial performance. The horizontal lines in Figure 2 show the performance of OpenMP/Cilk Plus single-threaded, and serial.

In Cilk Plus, task stealing can be expensive, if the time taken to execute the iterations on a single thread is considerably less. This happens particularly for the *mini* and *small* test cases, where we observe (in Figure 2) that increasing the number of threads does not result in a better overall execution time, in fact quite the opposite. For all the direct algorithms using OpenMP/Cilk Plus with mini/small input sizes, we observe a similar trait of poor multithreaded performance as compared to a single thread. This is expected for small data sizes, as there is very little work to be done in parallel, and most of the total time is comprised of OpenMP/Cilk Plus runtime overheads. With larger data sets, although there is a better load balance and we observe significant scalability, the performance of direct algorithms with dynamic loop scheduling (including tasking variants) is inferior when compared to SERIALFFD performance.

In general, for mini/small item sizes, only OMP-STA-STR-N and ITER-SPEC-CORRECT show some improvements over the serial/single-threaded versions. This is expected, since there are no contention among threads as each thread access different bins for inserting items, making the problem embarrassingly parallel.

OMP-STA-STR-1 on the other hand has better or similar execution times as compared to OMP-DYN. Letting all threads access different bins at the first time may offer a reasonable performance advantage. We observe OMP-STATIC-STR-1 performs about 10-15% better than OMP-DYN for a moderate number of items, up to 8 threads. With increasing items and/or threads, this benefit diminishes, as the number of items that could be packed with no contention at the first time (when all threads access different bins) is nominal, compared to the number of items left to be packed when many threads are contesting for bins. For large data sizes (above 200K), it can be seen that the speedup for SPEC-CORRECT flattens out after 16 threads. This is because there is a significant serial portion in SPEC-CORRECT (re-insertion of items from overfull bins), and in accordance with Amdahl's law, serial computation overwhelms the execution time for a larger input data.

2) Results on Task Parallel Implementations: We show in Table II the performance of task parallel FFD variants compared to the OMP-DYN/CILK-FOR version for packing 200K and 800K items. Our tasking experiments show mixed results. For small-moderate input data sizes, the tasking versions sometimes exhibit very similar performance compared to OMP-DYN, and, at other times it is 5-10% slower on an average. Since there are small but consistent differences between the tasking versions, we chose to tabulate the results for two data sizes from the *large* data class.

We found OMP-SNG-TASK and OMP-FOR-TASK to be similar in performance with each other, and reasonably better than the other tasking variants. We observe Cilk Plus tasking implementation to have the worst performance compared with the remaining tasking versions. The difference in Cilk Plus versions with 1 thread and the pure serial version (with no Cilk Plus keywords), also referred to as the *spawn* overhead, was found to be ∼15-20%. This suggests coarsening the tasks may help in improving the performance.

In general, the tasking versions do not perform significantly better than the data parallel versions, which could be attributed to overheads due to granularity of tasks, and the computation volume of a task.

3) Impact of scheduling on SPECULATEITERATE*:* SPECULATEITERATE (Algorithm 4) involves repeatedly executing

TABLE II. PERFORMANCE OF OPENMP/CILK PLUS TASKING IMPLEMENTATIONS AS COMPARED TO DATA PARALLEL APPROACHES ON 2-32 THREADS FOR PACKING 200K (LEFT) AND 800K (RIGHT) ITEMS. **OMP** (OMP-DYN) COMPARED WITH **F-UT** (OMP-FOR-TASK-UTD), **F-T** (OMP-FOR-TASK), **S-UT** (OMP-SNG-TASK-UTD) AND **S-T** (OMP-SNG-TASK). **CLK** (CILK-FOR) IS COMPARED WITH **C-SP** (CILK-SPAWN).

Thd	OpenMP					Cilk Plus	
	OMP	**F-UT**	**F-T**	**S-UT**	**S-T**	**CLK**	**C-SP**
2	29.32	29.94	30.41	29.90	30.23	28.53	31.68
4	15.09	15.39	15.33	15.26	15.24	15.61	18.79
8	8.39	8.24	8.41	8.29	8.27	10.08	10.97
16	4.61	4.55	4.70	4.57	4.44	6.27	6.54
24	3.48	3.22	3.29	3.24	3.15	4.66	4.79
32	2.64	2.58	2.91	2.56	3.02	3.63	3.69

Thd	OpenMP					Cilk Plus	
	OMP	**F-UT**	**F-T**	**S-UT**	**S-T**	**CLK**	**C-SP**
2	1551.48	1586.61	1598.47	1592.22	1592.85	1406.32	1643.18
4	790.65	814.21	803.40	802.78	814.28	732.66	914.93
8	416.97	431.48	432.34	424.89	423.03	363.92	466.82
16	227.15	229.14	227.21	225.28	227.30	208.15	245.16
24	154.89	168.48	167.64	184.47	159.99	141.09	164.56
32	121.77	139.31	123.98	122.84	122.62	101.61	155.34

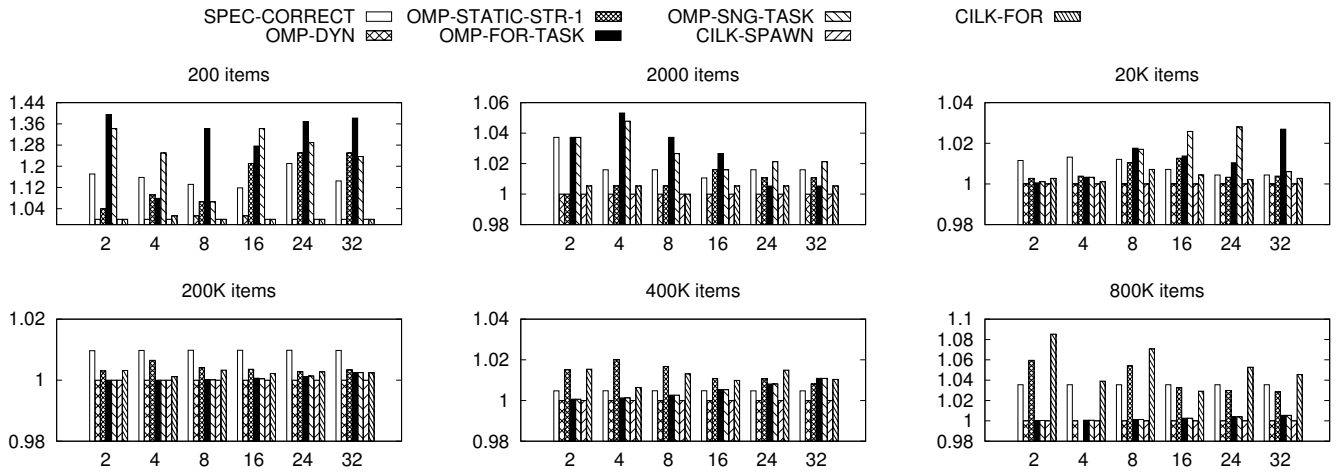

Fig. 4. Normalized number of bins used in parallel bin packing algorithms for different input sequences. The normalizing factor is the number of bins used by SERIALFFD. (refer to Table III for comparison with lower bound). **X-axis**: Number of threads. **Y-axis**: Ratio of bins used by parallel bin packing algorithms to SERIALFFD.

INSERTITEMSINPARALLEL and CHECKANDCORRECT for a number of iterations until all items are inserted into bins. To optimize the insertion of items, INSERTITEMSINPARALLEL uses the strategy mentioned in DIRECTSTATIC (Algorithm 3), which ensures two threads will not access the same bin at a time. In order for this logic to work, static scheduling of loops is needed to map a thread to a bin. However, the second step of SPECULATEITERATE, i.e., CHECKANDCORRECT could either use static[4] or dynamic scheduling to access bins. Figure 3 shows the comparative performance of SPECULATEITERATE when CHECKANDCORRECT uses static or dynamic scheduling. Static schedule could be useful in improving the memory access latency as the same thread touches the same data. We observe static schedule in CHECKANDCORRECT results in overall ∼10-30% better performance for SPECULATEITERATE on greater than 16 threads, as compared to dynamic schedule.

B. Number of bins used

Let $FFD(L)$ denote the number of bins used by the FFD bin packing algorithm when applied on a list L of items, and let L^* denote optimal number of bins. Johnson *et al.* [16] show that the maximum value obtained by the ratio $\frac{FFD(L)}{L^*}$, $R_{FFD}(k)$, exhibits this property in the limit: $\lim_{k\to\infty} R_{FFD}(k) = \frac{11}{9}$.

[4]CHECKANDCORRECT in the form that we have shown (in Algorithm 5), requires a minor modification to support static scheduling.

TABLE III. NUMBER OF BINS USED BY SERIALFFD AND LOWER BOUNDS (LBFFD)

Sequence size	SERIALFFD	LBFFD
200	76	62
2000	188	154
20,000	1,820	1,489
200,000	19,789	16,191
400,000	78,676	64,371
800,000	310,016	253,649

We can substitute $FFD(L)$ in the expression $\frac{9}{11} \cdot FFD(L)$ with the number of bins used for SERIALFFD and obtain a lower bound on L^*. For instance, for 200 data items, the lower bound on L^* comes to 62.18; we denote this by LBFFD (Lower Bound for FFD). In Table III, we list the number of bins used by SERIALFFD, and the respective lower bounds for the input data classes. The LBFFD values are rounded off to nearest integers.

Due to non-deterministic execution of threads, the parallel versions of the FFD algorithm we have studied are likely to use more number of bins compared to serial FFD. In Figure 4 we plot the number of bins used by some of the parallel implementations for a range of processor cores.

OMP-DYN and CILK-SPAWN always produced the same number of bins as SERIALFFD for 98% of the cases. This is due to the fact that a number of threads could contest for a single bin (as per Lines 8-10 of Algorithm 2), which insure the number of bins to be similar to SERIALFFD. Since ITER-

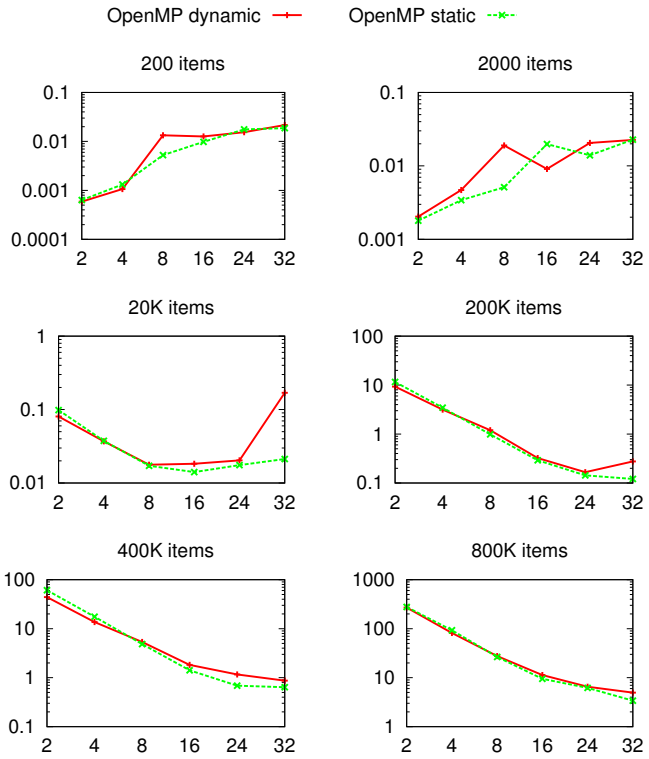

Fig. 3. Comparison between SPECULATEITERATE (ITER-SPEC-CORRECT) versions with OpenMP *static* and *dynamic* loop scheduling in CHECKANDCORRECT. **X-axis**: Number of threads. **Y-axis**: Execution time (in secs) in log-scale.

SPEC-CORRECT and OMP-STA-STR-N used significantly more bins (on an average, 20% more bins compared to SERIALFFD) than the rest of the parallel versions, we chose to not display them on Figure 4.

As discussed in Section III-C, to reduce the number of bins in ITER-SPEC-CORRECT, we introduced SPEC-CORRECT, which handles the re-insertion of items from overfull bins in serial. SPEC-CORRECT improves the amount of bins (in the worse case, only 3.5% extra bins are used, as compared to SERIALFFD), by accepting reasonable penalty in scalability (due to the serial overhead, which is unaffected by the number of threads used).

Cilk Plus exhibits very different results for data and task parallel versions. For instance, the tasking version of Cilk Plus (which uses `cilk_spawn` and `cilk_sync`) produced the same number of bins as SERIALFFD every time; whereas, the data parallel version (using `cilk_for`) uses ∼0.5%-7% more bins.

We observed no difference in the number of bins between OpenMP tied and untied tasking versions, hence we only show the tied versions in Figure 4. Although the OpenMP tasking versions (OMP-SNG-TASK and OMP-FOR-TASK) are based in OMP-DYN, they almost always required more bins than OMP-DYN. This may be an artifact of threads executing explicit tasks in a certain order, because task scheduling in OpenMP is implementation specific.

V. CONCLUDING REMARKS

We have devised multiple algorithms for the bin packing problem, and demonstrate usage of data and task parallel approaches for their parallelization. Broadly, the parallel algorithms are categorized into *direct* and *speculative* approaches. In the direct approach, OpenMP/Cilk Plus directives/keywords are used to obtain the solution in a single step. In the speculative strategies, concurrency is maximized at the expense of using multiple iterations. While we understand that Cilk Plus keywords are more suitable for programs with serial semantics, the performance of our parallel FFD versions (using Cilk Plus) gives us enough motivation to explore recursive bin packing algorithms, which will perhaps be a better fit for task-based programming models. We compared relative performances of these versions in terms of solution quality, and our results agree with theoretical expectations. Overall, among the parallel algorithms we studied, SPECULATEITERATE (Algorithm 4) provides the highest performance and scalability, and SPECULATECORRECT offers the best runtime-solution quality tradeoff. In future, we plan to extend this work by evaluating and analyzing the parallel versions on other hardware architectures and programming models (e.g. software transactional memory approaches).

ACKNOWLEDGEMENTS

This work is supported by NSF CAREER award IIS-1553528. The experiments were performed on the Aeolus cluster at the Voiland College of Engineering and Architecture at Washington State University. The authors thank Priyanka Ghosh for discussions on OpenMP tasking.

REFERENCES

[1] Richard J Anderson, Ernst W Mayr, and Manfred K Warmuth. Parallel approximation algorithms for bin packing. *Information and Computation*, 82(3):262–277, 1989.

[2] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[3] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[4] J. O. Berkey and P. Y. Wang. A systolic-based parallel bin packing algorithm. *IEEE Trans. Parallel Distrib. Syst.*, 5(7):769–772, July 1994.

[5] Judith O Berkey. Massively parallel computing applied to the one-dimensional bin packing problem. In *Frontiers of Massively Parallel Computation, 1988. Proceedings., 2nd Symposium on the Frontiers of*, pages 317–319. IEEE, 1988.

[6] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic task and data placement over numa architectures: an openmp runtime perspective. In *Evolving OpenMP in an Age of Extreme Parallelism*, pages 79–92. Springer, 2009.

[7] B Chapman, Fréderic Bregier, Amit Patil, and Achal Prabhakar. Achieving performance under openmp on ccnuma and software distributed shared memory systems. *Concurrency and Computation: Practice and Experience*, 14(8-9):713–739, 2002.

[8] Edward G Coffman, Jr, Michael R Garey, and David S Johnson. An application of bin-packing to multiprocessor scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[9] Edward G Coffman Jr, Michael R Garey, and David S Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1996.

[10] Leonardo Dagum and Rameshm Enon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[11] Emanuel Falkenauer and Alain Delchambre. A genetic algorithm for bin packing and line balancing. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 1186–1192. IEEE, 1992.

[12] Michael R Garey, Ronald L Graham, and Jeffrey D Ullman. Worst-case analysis of memory allocation algorithms. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 143–150. ACM, 1972.

[13] Dorit S Hochbaum. *Approximation algorithms for NP-hard problems*. PWS Publishing Co., 1996.

[14] Haoqiang Jin and Rob F Van der Wijngaart. Performance characteristics of the multi-zone nas parallel benchmarks. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 6. IEEE, 2004.

[15] David S Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[16] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

[17] William Leinberger, George Karypis, and Vipin Kumar. Multi-capacity bin packing algorithms with applications to job scheduling under multiple constraints. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 404–412. IEEE, 1999.

[18] Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete applied mathematics*, 28(1):59–70, 1990.

[19] Arch D Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

[20] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the cost of atomic operations on modern architectures.

[21] Petra Schwerin and Gerhard Wäscher. The bin-packing problem: A problem generator and some numerical experiments with ffd packing and mtp. *International Transactions in Operational Research*, 4(5-6):377–389, 1997.

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

[23] Christian Terboven, Dirk Schmidl, Henry Jin, Thomas Reichstein, et al. Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, pages 377–384. ACM, 2008.

[24] Eduardo C Xavier and Flávio Keidi Miyazawa. The class constrained bin packing problem with applications to video-on-demand. *Theoretical Computer Science*, 393(1):240–259, 2008.