Optimally Scheduling Small Numbers of Identical Parallel Machines

Richard E. Korf and Ethan L. Schreiber
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095
korf@cs.ucla.edu, ethan@cs.ucla.edu

Abstract

Given a set of n different jobs, each with an associated running time, and a set of k identical machines, our task is to assign each job to a machine to minimize the time to complete all jobs. In the OR literature, this is called identical parallel machine scheduling, while in AI it is called number partitioning. For eight or more machines, an OR approach based on bin packing appears best, while for fewer machines, a collection of AI search algorithms perform best. We focus here on scheduling up to seven machines, and make several new contributions. One is a new method that significantly reduces duplicate partitions for all values of k, including $k = 2$. Another is a new version of the Complete-Karmarkar-Karp (CKK) algorithm that minimizes the makespan. A surprising negative result is that dynamic programming is not competitive for this problem, even for $k = 2$. We also explore the effect of precision of values on the choice of the best algorithm. Despite the simplicity of this problem, a number of different algorithms have been proposed, and the most efficient algorithm depends on the number of jobs, the number of machines, and the precision of the running times.

Introduction and Overview

Consider the following very simple scheduling problem. Given a set of n different jobs, each with an associated running time, and a set of k identical parallel machines, such as processor cores, assign each job to a machine in order to minimize the makespan of the schedule, or the time to complete all jobs. The running time of each machine is the sum of the running times of the jobs assigned to it, and the makespan is the maximum running time of the machines. For example, given five jobs with running times $4, 5, 6, 7, 8$, and two machines, assigning $4, 5, 6$ to one machine and $7, 8$ to the other results in a running time of 15 for each machine.

This is perhaps the simplest non-trivial scheduling problem. It is NP-complete, even for two machines (Garey and Johnson 1979). In operations research (OR), it is called "identical parallel machine scheduling", while in AI it is called "number partitioning".

Previous Work in Operations Research

In OR, the state-of-the-art is represented by (Dell'Amico et al. 2008), and is based on the close relationship between number partitioning and bin packing. Both problems start with a set of integers. Number partitioning packs them into a fixed number k of subsets or bins, so as to minimize the largest subset sum. A bin packer packs them into as few bins or subsets as possible, without exceeding a fixed bin capacity C of any bin, corresponding to a maximum allowable subset sum. The OR approach to number partitioning is to fix the number of bins at k, and then perform a binary search over different bin capacities C, solving a series of bin-packing problems, until the smallest feasible C is found. To solve the bin-packing problems, they use an integer programming approach called branch-and-cut-and-price (BCP).

Gleb Belov graciously provided us with his state-of-the-art BCP solver, and we compared its performance to our RNP algorithm, described below. For eight or more subsets, BCP is superior to RNP, but for seven or fewer subsets, RNP is superior. For k=7, RNP is about twice as fast as BCP, for k=6, RNP is 134 times faster, for k=5, RNP is 5,527 times faster, for k =4, RNP is 78,302 times faster, and for k=3, RNP is 589,721 times faster than BCP on the largest problems we could solve with BCP. Thus, we restrict our attention here to relatively few subsets. Many applications, such as scheduling processor cores, have few machines.

Integer Precision and Perfect Partitions

An important issue that has been overlooked in the OR literature but not in the AI community (Korf 1998) is the number of bits used to represent the integers, and the presence of *perfect partitions*. A perfect partition in one where all subset sums are equal, as in the example above, or where they differ by at most one, in those cases where the sum of all the integers is not divisible by k. A perfect partition is always optimal, and once found, search can terminate immediately.

If we fix the precision of the integers, and the number of subsets k, and increase the number of integers n, the number of partitions grows exponentially as k^n, but the number of subset sums grows only linearly with n. Thus as n increases, the likelihood of a perfect partition also increases, eventually becoming almost certain. Problem instances with perfect partitions are generally easier to solve, but the relative performance of different algorithms is different with and without perfect partitions, as we will see.

Table 1 shows where perfect partitions occur. The columns represent different numbers of subsets k, and the rows different integer precisions p. The power of ten labelling each row is the maximum possible integer. The data entries give the smallest value of n for which at least half of a set of uniform random problem instances had a perfect partition, determined experimentally. For example, the four in the top position of column 2 means that when partitioning 4 integers uniformly chosen from 0 to 9 into two subsets, at least half the problem instances had a perfect partition. Each data point represents at least 100 trials.

p / k	2	3	4	5	6	7
10^1	4	7	9	11	13	16
10^2	9	11	14	17	19	22
10^3	12	16	18	22	25	28
10^4	15	20	25	28	31	35
10^5	18	24	29	33	38	42
10^6	22	28	34	39	44	49
10^7	26	33	39	45	50	
10^8	29	37	44	51	57	
10^9	33	41	49	57		
10^{10}	36	45	55	62		
10^{11}	39	50	59			
10^{12}	42	54				

Table 1: The Approximate Sizes of the Hardest Problems

Problems of the size shown in Table 1 are among the most difficult. The empty locations represent values for which the hardest problems took too long to run. Note that the bottom rows represent integers with very high precision in practice. For example, if we use 12-digit integers to represent times in seconds, we can represent over 31.7 thousand years!

A Note Regarding Choice of Benchmarks

All our experiments are based on uniform random problem instances, rather than real-world problem instances. The reason is that problem difficulty depends on the size n, the number of subsets k, and the precision p of the integers. Any particular set of real-world instances are likely to fall within a narrow range of these parameters, and hence not provide a very comprehensive picture of the performance of the algorithms. We choose the uniform distribution because it is the simplest, and doesn't require any additional parameters.

Overview of the Rest of This Paper

We first consider two-way partitioning, and briefly describe five different existing algorithms for this problem. We then introduce a new technique to improve the performance of most of them. Next we experimentally compare their relative performance as a function of the precision p and problem size n. We then consider the case of multi-way partitioning. We describe a new version of an existing algorithm (CKK), and a significant improvement to another algorithm (RNP). We then analyze their performance both with and without perfect partitions. Finally we present our conclusions and further work.

Two-Way Partitioning

We begin with two-way partitioning, corresponding to scheduling two machines. This is equivalent to the subset-sum problem of finding a subset of a set of integers whose sum is closest to half the sum of all the integers. We next describe five different optimal algorithms for this problem.

Complete Greedy Algorithm (CGA)

The simplest algorithm for this problem is the complete greedy algorithm (CGA) (Korf 1998). It sorts the integers in decreasing order, and searches a binary tree, assigning a different integer at each level. The left branch of each node assigns the next integer to the subset with the smaller sum so far, and the right branch assigns it to the subset with the larger sum. Thus, the first solution found is that returned by the obvious greedy heuristic for this problem. CGA keeps track of the larger subset sum of the best solution found so far, and prunes a branch when the sum of either subset equals or exceeds this value. Without perfect partitions, the time complexity of CGA is slightly better than $O(2^n)$, with the difference due to pruning. It's space complexity is $O(n)$.

Complete Karmarkar-Karp (CKK)

A better algorithm is based on a heuristic approximation originally called set differencing (Karmarkar and Karp 1982), but usually referred to as KK. KK sorts the integers in decreasing order, and at each step replaces the two largest integers with their difference. This is equivalent to separating the two largest integers in different subsets, without committing to their final placement. For example, placing 8 and 7 in different subsets is equivalent to placing a 1 in the subset with the 8. The difference is then treated as another integer to be assigned. The algorithm continues until there is only one integer left, which is the difference between the subset sums of the final two-way partition. Some additional bookkeeping is needed to construct the actual partition. The KK heuristic runs in $O(n \log n)$ time, and finds much better solutions than the greedy heuristic.

The Complete Karmarkar-Karp algorithm (CKK) is a complete optimal algorithm (Korf 1998). While KK always places the two largest integers in different subsets, the only other option is to place them in the same subset, by replacing them with their sum. Thus, CKK searches a binary tree where at each node the left branch replaces the two largest integers by their difference, and the right branch replaces them by their sum. The first solution found is the KK solution. If the largest integer equals or exceeds the sum of all the remaining integers, we terminate the branch by placing them in opposite subsets. Without perfect partitions, CKK also runs slightly faster than $O(2^n)$, again due to pruning. CKK also requires only $O(n)$ space.

Horowitz and Sahni (HS)

The Horowitz and Sahni (HS) algorithm (Horowitz and Sahni 1974) is a different approach to the subset sum problem. It divides the n integers into two "half" subsets a and c, each of size $n/2$. Then it generates all subsets from each half subset, including the empty set. The two lists of subsets are sorted in order of their subset sums. Any subset of the original integers consists of a subset of the a integers plus a subset of the c integers. Next, it initializes a pointer to the empty subset of the a list, and the complete subset of the c list. If the subset sum pointed to by the a pointer, plus the subset sum pointed to by the c pointer, is more than half the sum of all the integers, the c pointer is decremented to the subset with the next smaller sum. Alternatively, if the sum of the subset sums pointed to by the two pointers is less than half the total sum, the a pointer is incremented to the subset with the next larger sum. If the sum of the two subset sums equals half the total sum, it terminates with a perfect partition. Otherwise, HS continues until either list of subsets is exhausted, returning the best solution found.

HS runs in $O(2^{n/2}(n/2))$ time and $O(2^{n/2})$ space. This is much faster than CGA and CKK, but it's memory requirement limits it to values of n up to about 50.

Schroeppel and Shamir (SS)

The SS algorithm (Schroeppel and Shamir 1981) is based on HS, but uses much less space. Note that HS uses the subsets from the a and c lists in order of their subset sums. Rather than generating, storing, and sorting these subsets, SS generates them dynamically in order of their subset sums.

SS divides the n integers into four sets a, b, c and d, each of size $n/4$, generates all subsets from each set, and sorts them in order of their sums. The subsets from the a and b lists are combined in a min heap that generates all subsets of elements from a and b in increasing order of their sums. Each element of the heap consists of a subset of the a list, and a subset of the b list. Initially, it contains all pairs combining the empty set from the a list with each subset of the b list. The top of the heap contains the pair whose subset sum is the current smallest. Whenever a pair (a_i, b_j) is popped off the top of the heap, it is replaced in the heap by a new pair (a_{i+1}, b_j). Similarly, the subsets from the c and d lists are combined in a max heap, which returns all subsets from the c and d lists in decreasing order of their sums. SS uses these heaps to generate the subset sums in sorted order, and combines them as in the HS algorithm.

Similar to the HS algorithm, the SS algorithm runs in time $O(2^{n/2}(n/4))$, but its space complexity is only $O(2^{n/4})$, making it practical for values of n up to about 100.

A recent algorithm reduces this runtime to approximately $O(2^{n/3})$ (Howgrave-Graham and Joux 2010), but is probabilistic, solving only the decision problem for a given subset sum. It cannot prove there is no solution for a given sum, and doesn't return the subset sum closest to a target value.

Dynamic Programming (DP)

Finally, we consider the well-known dynamic programming (DP) algorithm for subset sum (Garey and Johnson 1979). DP allocates a two-dimensional bit array with n rows and t columns, where t is the sum of all the integers. Eventually this array will contain a one in row i and column j if it is possible to construct a subset whose sum is j, from the first i integers. The array is initialized to all zeros. In the first row, corresponding to the integer x_1, the bit whose index is x_1 is set to one, corresponding to the singleton set containing x_1. Then, for each row i, the previous row $i - 1$ is scanned. Every one bit in the previous row is copied to the next row, x_i is added to its index, and the bit at that index is set to one as well. This corresponds to excluding and including the current integer x_i in each subset of the previous integers. In the complete array, the indices of the one bits in the last row represent the possible subset sums of the original integers.

The time and space complexity of DP is $O(t \cdot n)$. Since each row is computed from only the previous row, only one row must be stored at a time, if the rows are scanned from the largest to the smallest indices. This reduces the space complexity to $O(t)$, which limits its applicability to relatively low precision integers. Processing the integers from largest to smallest significantly speeds up the algorithm in the presence of perfect partitions, since including the largest integers first reaches half the sum of all the integers quicker. In addition, keeping track of the first and last non-zero entries reduces the number of zero entries that must be scanned.

Eliminating Half the Subsets

Our first contribution to two-way partitioning starts with the simple observation that every subset has a unique complement subset, whose sum is t minus that of the original subset. If we naively search over all subsets, each two-way partition will be generated twice, once by generating the original set, and again by generating its complement.

Eliminating this redundancy is obvious for some of the above algorithms. In CGA, for example, we only put the largest integer in one of the two subsets. This redundancy does not appear in CKK. In the DP algorithm, we only allocate as many elements as half the sum of all the integers, halving both the space and time complexity.

Removing this redundancy from HS or SS is not so obvious, however. It is not mentioned in either of their papers, and eluded us until recently. We simply exclude one of the original integers from the generated sets, such as the largest one, and for each subset generated, consider both it and its complement, augmented with the excluded integer. Since the time complexity of both these algorithms is $O(n2^{n/2})$, reducing n by one should produce a speedup of slightly more than the square root of two, which is 1.414. In our experiments with high precision integers without perfect partitions, this optimization resulted in a speedup of about 1.45 for HS, and 1.53 for SS. All the experimental results below are for versions of these algorithms with this optimization included.

Experimental Results

Which is the fastest of the five different two-way partitioning algorithms described above? The answer depends on the number of integers n, and their precision p.

Conventional Wisdom We first consider asymptotic complexity. These algorithms can be divided into three groups. CGA and CKK search a binary tree, require $O(n)$ space, and slightly less than $O(2^n)$ time without perfect partitions. HS and SS require $O(2^{n/2})$ and $O(2^{n/4})$ space, respectively, and run in time $O(2^{n/2}(n/2))$ and $O(2^{n/2}(n/4))$ respectively. Finally, DP requires $O(t/2)$ space where t is the sum of all the integers, and $O(nt/2)$ time.

Given these time complexities, one would expect that given sufficient memory to run them all, DP should be the fastest, followed by HS and SS, and then CGA and CKK.

Most treatments of NP-complete problems describe the DP algorithm for numerical problems, point out its low time complexity for fixed precision integers, and imply that it is the algorithm of choice if sufficient memory is available. DP is referred to as a "pseudo-polynomial" time algorithm, and problems without such an algorithm are referred to as "strongly NP-complete", suggesting that problems with a DP solution are easier to solve because of it.

If there is insufficient memory to run DP, but enough memory to run SS or even HS, they would be expected to outperform CGA and CKK. For example, when I introduced CKK (Korf 1998), I concluded that "For problems with fewer than 100 numbers, or for problems where the numbers have low precision, there exist more efficient algorithms." (pp. 202) referring to SS and DP, respectively. Furthermore, since HS and SS have similar asymptotic time complexity, but HS is much simpler, one would expect HS to run faster than SS given sufficient memory to run both.

What we found experimentally is rather different. We first consider high-precision integers without perfect partitions, and then problems with perfect partitions.

High-Precision Integers Without Perfect Partitions To eliminate perfect partitions, we used integers with 48 bits of precision, which is over 14 decimal digits. The empirically observed time complexity of CGA is about $O(1.95^n)$, while for CKK it is about $O(1.90^n)$. By $n = 40$, CKK runs over twice as fast as CGA, due to more efficient pruning. Problems of size 40 take CKK about 36 seconds to solve.

Comparing HS and SS, we were initially surprised to find that in addition to using much less memory, SS runs about twice as fast as HS. The dominant cost in HS is the time to sort the $2^{n/2}$ subsets from each of the half lists by their subset sums. This takes $O(2^{n/2} \log 2^{n/2})$ or $O(2^{n/2}(n/2))$ time. The analogue of this sorting step in SS is the time to sort the subsets in the four quarter lists, and to insert them into their respective heaps. Since these lists and heaps are only of size $2^{n/4}$, the sorting and heap insertion times are the log of this value, for a running time of $O(2^{n/2} \log 2^{n/4})$ or $O(2^{n/2}(n/4))$. The ratio of these two running times is the factor of two we observe experimentally.

This leaves SS and CKK as the undominated algorithms for high-precision integers, since DP is not applicable to such problems. Which is faster depends on the problem size n. For n less than 12, CKK is faster, due to its lower overhead per node. For larger values of n, SS is faster, due to its lower asymptotic complexity. The ratio of the running time of CKK to that of SS grows as n increases. For $n = 40$, for example, SS is about 500 times faster than CKK. Furthermore, without perfect partitions, the time and space complexities of both these algorithms depend only on n, and are not affected by the precision, if we ignore the additional bits needed to store high-precision values.

Fixed Precision Integers With Perfect Partitions To evaluate DP, we need to limit the precision of the integers, introducing perfect partitions. Surprisingly, we found that even our highly optimized DP is uniformly slower than CKK. For integers up to ten, CKK is only slightly faster. As the precision increases, however, the ratio of their running times increases monotonically. For integers up to one million, CKK is four orders of magnitude faster than DP.

With perfect partitions, which occur increasingly often with fixed precision values as n increases, the relative performance of SS and CKK is more complicated. The reason is that CKK is good at finding perfect partitions quickly, but SS is not. Consider the best case, where the first complete two-way partition found by each algorithm is a perfect partition. In that case, CKK runs in $O(n \log n)$ time. SS, however, has to generate and sort all the subsets of the quarter sets, and initialize the two heaps, before generating even the first complete partition. This takes $O(2^{n/4} \log 2^{n/4})$ time or $O(2^{n/4}(n/4))$. Thus, if we hold the precision fixed as n increases, at some point CKK will run faster than SS, and the ratio of their running times will increase without bound until about $n = 100$, beyond which SS is no longer feasible, due to its memory requirements.

Table 2 shows our experimental results. The top row represents the precision of the integers, in terms of number of decimal digits, with the integers ranging from zero to the power of ten in the top row. For $n < 12$, CKK is always faster than SS, regardless of the precision, due to less overhead per node. The numbers in the second row are the largest values of n for which SS is faster than CKK, for the given precision. For integers up to 10^3, CKK is always faster than SS. For integers up to 10^4, for example, CKK is faster up to $n = 12$, then SS is faster up to $n = 17$, and then CKK is faster for values larger than 17, due to the presence of perfect partitions. This pattern is repeated for each precision, with CKK being fastest for $n < 12$, and for n greater than the value in the table, while SS is fastest in the range between 12 and the table value. Note that when it is faster, SS can be orders of magnitude faster than CKK, due to its lower worst-case asymptotic complexity. Similarly, for large values of n, CKK can be orders of magnitude faster than SS, due to its lower best-case asymptotic complexity in the presence of perfect partitions.

p	4	5	6	7	8	9	10	11	12
n	17	24	29	36	43	49	56	63	71

Table 2: Largest n for Which SS is Faster than CKK

This data contrasts sharply with my claim in (Korf 1998) that SS is faster than CKK for all values of $n < 100$. Note that CKK is extremely simple to implement, compared to SS, and that it can be applied to arbitrarily large problems, since its space complexity is only $O(n)$. SS, on the other hand, is limited to problems up to size $n = 100$.

In practice, two-way partitioning problems even with 48 bits of precision are easy to solve with the right algorithm. Problems of size $n = 50$, about half of which have perfect partitions, take SS only about 2.3 seconds to solve optimally. Of course, by increasing the precision, we can always construct more difficult two-way partitioning problems.

Multi-Way Partitioning

We now consider partitioning into more than two subsets, corresponding to scheduling more than two machines. Neither HS nor SS generalize to multi-way partitioning, since they only generate single sets. DP is not practical for k-way partitioning it requires $O((t/2)^{k-1})$ space. CGA generalizes directly to k-way partitioning, searching a k-ary tree in which each integer is alternately assigned to one of k subsets, in increasing order of their subset sums.

We first present a new version of the CKK algorithm, designed to minimize the makespan. We next describe an existing multi-way partitioning algorithm called recursive number partitioning (RNP). Then we describe a new technique to limit the number of duplicate nodes in RNP, which generalizes the technique described above for two-way partitioning. Finally, we present our experimental results for multi-way partitioning, both with and without perfect partitions.

A New CKK that Minimizes the Makespan

The original multi-way CKK algorithm (Korf 1998) was designed to minimize the difference between the largest and smallest subset sums. While for two-way partitioning this is the same as minimizing the largest subset sum, for multi-way partitioning, these are different objective functions (Korf 2010). We describe here a new version of CKK that minimizes the largest subset sum.

For simplicity, we consider three-way partitioning, but the algorithm generalizes to arbitrary k-way partitioning. Rather than maintaining a list of integers, it maintains a list of ordered triples. Each element of a triple is either an original integer, or the sum of two or more original integers. A triple such as (x, y, z) represents a commitment to assign x, y, and z to separate subsets in any final three-way partition. Furthermore, whenever two or more original integers are combined into a single element of a triple, such as $x = i + j$, i and j remain together in any subsequent partition. Each triple is ordered from largest to smallest values.

Initially, each integer is placed in its own triple, with zeros for the other two values. The algorithm repeatedly combines two triples together in all different ways, until only one triple remains, representing the final three-way partition.

For example, if a and x are the two largest original integers, the two triples $(a, 0, 0)$ and $(x, 0, 0)$ will be combined into one. There are only two different ways to do this, either placing the a and x in different subsets, resulting in the triple $(a, x, 0)$, or placing them in the same subset, resulting in $(a + x, 0, 0)$. The search will branch on these two combinations, searching the first combination first.

In general, given two triples (a, b, c) and (x, y, z), there are 3! or six different ways to combine them, resulting in the triples $(a + z, b + y, c + x)$, $(a + z, b + x, c + y)$, $(a+y, b+z, c+x)$, $(a+y, b+x, c+z)$, $(a+x, b+z, c+y)$, and $(a+x, b+y, c+z)$. They are explored in this order, since this approximates the order of increasing difference between their largest and smallest values. At each node of the algorithm, we combine the two triples with the largest difference between their largest and smallest values, and replace them by each possible combination, branching up to six ways.

This is a branch-and-bound algorithm that keeps track of the largest subset sum in the best solution found so far. The search tree is pruned as follows. Any combination that results in a triple in which any value equals or exceeds the largest subset sum in the current best solution is pruned. Furthermore, since all the triples will eventually be combined into one, each value in any triple will eventually have added to it at least one value from every other current triple. Thus, each value must have added to it at least the smallest value from every other triple. Let l_i be the largest value of some triple, s_i be the smallest value of that same triple, sum be the sum of the smallest values of all current triples including triple i, and m be the largest subset sum in the best solution found so far. We prune the search when $l_i - s_i + sum \geq m$ for any triple i.

While this test may seem to require time linear in the number of triples, we can compute it incrementally in constant time. The sum of the smallest values is maintained incrementally, since each step combines just two triples into one. To test for pruning, we only have to compute the inequality for the triple i with the largest value of $l_i - s_i$. This is the triple with the largest difference between its largest and smallest values, which is also maintained incrementally, since it always participates in the next combination.

The first solution found by this algorithm is the KK approximation, and it eventually guarantees an optimal solution. The generalization to k-way partitioning is straightforward, with k-tuples replacing triples, and a maximum branching factor at any node of $k!$. The space complexity is $O(kn)$, and it's worst-case time complexity without perfect partitions or any pruning is $O(k^n/k!)$, the number of unique k-way partitions, up to permutations of the subsets.

The main difference between the original CKK algorithm and our new version, besides minimizing different objective functions, is the new pruning rule described above. In addition, the original version maintained only $k - 1$ tuples for k-way partitioning, normalizing each tuple by subtracting the smallest value from each other value.

Recursive Number Partitioning

For small k and large n without perfect partitions, Recursive Number Partitioning (RNP) (Korf 2009; 2011) is the best existing algorithm. We first describe the algorithm for three-way partitioning, and then show how to generalize it.

RNP first runs KK to get an approximate solution. For three-way partitioning, it then generates each subset that could be part of a better three-way partition. For each such first subset, it optimally partitions the complement set two ways to arrive at a three-way partition. This is based on the observation that given any optimal solution to a k-way partitioning problem, optimally partitioning the elements in any subcollection of the k subsets will still result in an optimal solution, since it can only reduce the maximum subset sum.

Let t be the sum of all the integers, and m be the largest subset sum in the best solution so far. In order for a subset to be part of a better three-way partition, its sum must be less than m, and greater than or equal to $t - 2(m - 1)$, so that it's complement could possibly be partitioned into two sets both of whose sums are less than m. To generate these first subsets, RNP uses an extension of the SS algorithm that generates all subsets whose sum lies within a given range (Korf 2011), rather than a single subset whose sum is closest to a target value. RNP is a branch-and-bound algorithm that continues to search for better solutions, until it finds and verifies an optimal solution.

For four-way partitioning, at the top-level RNP partitions the integers into two subsets, in all ways such that optimally partitioning each subset two ways could result in four subsets, each with a sum less than m. For five-way partitioning, at the top level the integers are partitioned into two subsets, one is optimally partitioned two ways, and the other is optimally partitioned three ways. In general, for k-way partitioning, at the top level the integers are partitioned into two subsets, then one subset is optimally partitioned $\lfloor k/2 \rfloor$ ways, and other subset is optimally partitioned $\lceil k/2 \rceil$ ways.

Reducing Duplicate Partitions

Unfortunately, RNP can generate duplicate partitions that differ only by a permutation of the subsets. For example, RNP could generate any given three-way partition three different ways, by constructing each of the three subsets first, and then optimally partitioning their complements two ways. To reduce this redundancy, the original version of RNP restricted the range of the first subset sum to be greater than or equal to $t/3$, and less than m. This is complete, because in any three-way partition, at least one subset sum must equal or exceed $t/3$. This doesn't eliminate the redundancy however, since two of the three subsets in a three-way partition could have sums greater than or equal to $t/3$, and hence such partitions could be generated two different ways.

We introduce here an alternative method that eliminates more duplicate partitions, and is a generalization of optimizing two-way partitioning by excluding one of the integers. For three-way partitioning, we force the largest integer to be included in the first subset. Since any given integer can only appear in one subset of a partition, this guarantees that each three-way partition can only be generated one way.

In principle, we could choose any integer to include. We choose the largest integer because it has the biggest impact on the bounds on the remaining subset sum, and reduces the number of different subsets whose sum is in the correct range, compared to including a smaller integer.

For four-way partitioning, we include the two largest integers in the same subset of all the top-level two-way partitions. This is complete because the two largest integers can be in at most two subsets in any four-way partition. Unfortunately, this doesn't eliminate all duplicate partitions. For example, consider a four-way partition in which the two largest integers are in the same subset. These four subsets could be partitioned into two pairs of subsets at the top level in three different ways, combinining the subset with the two largest integers with each of the other three subsets.

In general, for k-way partitioning, at the top-level we partition all the integers two ways, into a $k1$ subset that will be optimally partitioned $k1$ ways, and a $k2$ subset that will be optimally partitioned $k2$ ways, where $k1 = \lfloor k/2 \rfloor$ and $k2 = \lceil k/2 \rceil$. Note that by forcing certain integers to be included, we can't require the sum of the set containing those integers to be larger or smaller than t/k, so the upper bound on the $k1$ subset sum is $k1(m - 1)$, and the lower bound is $t - k2(m - 1)$. Similarly, the upper bound on the $k2$ subset sum is $k2(m - 1)$ and the lower bound is $t - k1(m - 1)$.

If k is even, there are two ways to apply this technique. The first is to include the $k/2$ largest integers in the subset generated in the top-level two-way partition, and the other is to exclude them from the generated subset, which includes them in the complement set. While these may seem to be equivalent, the extended SS algorithm used to generate subsets at the top level generates them in a different order than it would generate their complements, resulting in different performance for a branch-and-bound algorithm.

If k is odd, there are four different ways to apply this technique. One is to generate the $k1$ subsets while including the $k1$ largest integers, and another is to generate the $k1$ subsets while excluding the $k2$ largest integers. Alternatively, we could generate the $k2$ subsets, either including the $k2$ largest integers, or excluding the $k1$ largest integers. All four variations perform differently for odd k, but generating the $k2$ subset while excluding the $k1$ largest integers consistently performed the best, over all k and n.

Another Optimization

RNP first partitions the integers into two subsets in all ways that could possibly lead to a better k-way partition, and then for each such top-level partition, optimally partitions each subset. This is more work than is necessary. For example, for three-way partitioning, if the first set has a sum of s, we don't need to find an optimal two-way partition of the complement set, but only one where both subset sums are less than or equal to s, since the largest of the three subset sums cannot be less than s. Similarly, with four-way partitioning, after partitioning one of subsets of the top-level partition two ways with a larger subset sum of s, when partitioning the other top-level subset two ways we can stop searching if we find a partition where both subset sums are less than or equal to s. In general, when constructing a k-way partition, if the largest sum of the final subsets constructed so far is s, any further recursive partitioning to complete that partition can terminate whenever final subsets with a sum of s or less are generated. The performance impact of this optimization is relatively minor, however, since most candidate partitions fail to improve on the best solution found so far.

Experimental Results

We now turn to experimental results for multi-way partitioning. As with two-way partitioning, the results are significantly different depending on whether or not perfect partitions exist, which depends on the precicion of the integers. We start high-precision integers without perfect partitions.

High-Precision Numbers Without Perfect Partitions To eliminate perfect partitions, we use 48-bit integers. Without

k	3-Way			4-Way			5-Way			6-Way			7-Way		
n	Old	New	R	Old	New	R	Old	New	R	Old	New	R	Old	New	R
25	.001	.000	1.37	.002	.001	2.07	.006	.002	3.30	.061	.006	9.48	.13	.02	6.22
26	.001	.001	1.55	.003	.001	1.96	.010	.003	3.84	.090	.010	8.80	.174	.015	11.9
27	.001	.001	1.37	.004	.002	2.15	.014	.003	4.24	.161	.018	9.08	.507	.034	14.7
28	.002	.001	1.64	.006	.003	2.23	.020	.005	3.70	.240	.027	8.79	.554	.045	12.2
29	.002	.002	1.33	.010	.004	2.43	.033	.008	4.27	.348	.040	8.81	1.03	.044	23.49
30	.003	.002	1.58	.016	.006	2.56	.050	.012	4.22	.606	.068	8.91	1.53	.077	19.84
31	.004	.003	1.30	.025	.009	2.79	.083	.018	4.71	.833	.101	8.27	2.39	.109	21.93
32	.007	.004	1.69	.042	.014	2.95	.129	.027	4.78	1.49	.166	8.98	4.08	.199	20.51
33	.009	.007	1.30	.063	.029	3.01	.195	.041	4.77	2.32	.248	9.36	6.04	.347	17.43
34	.014	.009	1.62	.092	.031	2.95	.335	.066	5.09	3.92	.456	8.60	10.6	.535	19.87
35	.018	.014	1.30	.132	.047	2.79	.478	.087	5.47	6.18	.740	8.35	17.6	1.04	16.93
36	.029	.017	1.66	.195	.070	2.77	.795	.138	5.76	9.86	1.15	8.57	29.4	1.42	20.76
37	.037	.029	1.26	.282	.103	2.74	1.55	.200	7.76	16.0	1.81	8.85	48.4	2.18	22.19
38	.058	.036	1.61	.431	.160	2.70	1.92	.317	6.07	26.0	3.14	8.92	80.0	3.51	22.78
39	.075	.059	1.26	.635	.234	2.71	2.95	.490	6.03	42.5	5.27	8.06	123	5.55	22.17
40	.125	.074	1.67	.978	.361	2.71	4.49	.756	5.94	73.9	8.72	8.50	228	9.64	23.62
41	.160	.125	1.28	1.40	.525	2.67	7.54	1.16	6.56	119	13.5	8.81	379	14.7	25.82
42	.251	.153	1.64	1.99	.800	2.48	11.9	1.72	6.91	202	24.8	8.12	698	26.6	26.22
43	.328	.253	1.30	3.05	1.22	2.51	18.1	2.65	6.84	315	41.0	7.68	1217	43.7	27.84
44	.528	.321	1.65	4.88	1.85	2.64	26.6	4.26	6.26	534	66.9	7.97	2197	77.0	29.70
45	.666	.529	1.26	6.68	2.64	2.53	41.9	6.30	6.65	850	104	8.21	3408	121	28.14
46	1.04	.685	1.52	10.8	4.34	2.50	65.4	9.99	6.55	1401	184	7.61	5573	208	26.85
47	1.34	1.05	1.28	16.0	6.26	2.55	110	15.8	6.94	2272	300	7.57	9878	373	26.46
48	2.22	1.36	1.63	24.1	9.42	2.55	162	24.8	6.52	4171	502	8.31		605	
49	2.75	2.24	1.23	36.9	14.3	2.57	242	36.6	6.60	6717	771	8.71		963	
50	4.33	2.81	1.54	61.0	23.3	2.62	470	59.8	8.28	10917	1384	7.89		1642	

Table 4: Average Time in Seconds to Optimally Partition 48-bit Integers 3, 4, 5, 6, and 7 Ways

k	3	4	5	6	7	8	9	10
n	10	12	15	16	17	19	21	22

Table 3: Largest n for Which CGA is Faster than RNP

perfect partitions, the performance of all our algorithms, except DP, is independent of the precision. The candidate algorithms for multi-way partitioning are CGA, our new version of CKK, and our improved version of RNP.

For multi-way partitioning without perfect partitions, CGA outperforms CKK for all values of n and $k > 2$, since it is much simpler and has a lower constant time per node. For three-way partitioning, CGA is only 12% faster than CKK, but for four-way partitioning it is about twice as fast, and for five-way partitioning it is about three times faster.

For small values of n, CGA also outperforms RNP, again due to its simplicity and lower overhead per node. Table 3 shows the largest value of n, as a function of k, for which CGA is faster than RNP with our new duplicate pruning method. Thus, to solve small recursive subproblems, RNP calls CGA. For values of n larger than those in Table 3, RNP outperforms CGA, and the ratio of their running times increases without bound as n increases. The differences between Table 3 and the corresponding Table 1 of (Korf 2011) reflect our improved version of RNP.

Table 4 shows the performance of RNP with our new duplicate pruning method, compared to our previous best version, described in (Korf 2011). This data is based on 48-bit integers, rather than the 31-bit integers used previously, to eliminate perfect partitions. We only show results for partitioning three through seven ways, since beyond seven the bin-packing approach described at the beginning of this paper outperforms RNP. Each data point represents the average over 100 instances of the time in seconds to solve a random instance for a given value of k and n, where the integers are uniformly distributed from zero to $2^{48} - 1$. Each group of three columns represents a different value of k, and gives the running time of our previous code (Old), our improved version (New), and the ratio (R) of the two. The ratios don't always equal the ratios of the table values, since they are based on the full precision of our data, which exceeds that presented in the table. These experiments were run on an Intel Xeon X5680 CPU running at 3.33GHz. There is relatively little variation among the 100 instances in each set of problems. The three empty entries for $k = 7$ represent problems where our previous code took too long to run.

Our new version of RNP is significantly faster than our previous version, and the difference increases with larger k. For 3-way partitioning, our new version alternates between about 1.25 and 1.5 times faster. For 4-way partitioning, it is about 2.5 times faster, for 5-way it is about 7 times faster on the larger problems, for 6-way it is about 8 times faster, and for 7-way it is over 26 times faster on the larger problems.

Fixed-Precision Numbers With Perfect Partitions For problems with perfect partitions, the relative performance of our algorithms depends on problem size. For small values of n, CGA is fastest, for intermediate values of n, RNP outperforms it, and for large values of n, our new CKK is fastest.

Consider three-way partitioning, for example. For integers up to 10, CGA dominates both CKK and RNP, which perform similarly. For integers up to 100, CGA is fastest for n up to 10. For $n = 11$ through $n = 15$, RNP is fastest. For all values of $n > 15$, CKK is the fastest algorithm. For values up to 1000, CGA is again fastest for n up to 10. For $n = 11$ through $n = 22$, RNP is fastest, and for larger values of n, CKK is fastest. For integers with four decimal digits, RNP is fastest from $n = 11$ through $n = 35$, and CKK is faster for larger values of n. For integers with five decimal digits, RNP is fastest from $n = 11$ through $n = 50$, and CKK is faster for larger values of n.

The reasons for these results are as follows. For multi-way partitioning without perfect partitions, CGA outperforms CKK because of it's simplicity and lower constant factors. With perfect partitions, however, CKK outperforms CGA because it finds better solutions and hence perfect partitions faster. The performance crossover between CGA and CKK occurs at values of n for which RNP outperforms both algorithms, however. For intermediate values of n, RNP is faster than both CGA and CKK because it has lower asymptotic complexity. With many perfect partitions, however, it must do $O(2^{n/4}(n/4))$ work before generating even its first partition, while CKK only has to do $O(n \log n)$ work before generating complete partitions, and thus CKK performs best for larger values of n. Note that since RNP uses the extended SS algorithm, it is not feasible for $n > 100$. At that point, without perfect partitions, CGA is the best algorithm available for multi-way partitioning, while with large numbers of perfect partitions, CKK is the fastest algorithm.

Conclusions and Further Work

We have explored a very simple scheduling problem. For two-way partitioning, we introduced a simple optimization that eliminates half the nodes generated and speeds up the HS and SS algorithms by a factor of about 1.5. We then experimentally compared five different algorithms for this problem, systematically varying the problem size and precision of the integers. We showed that CKK dominates CGA, and SS dominates HS. If we hold the precision fixed and increase the problem size n, we found that CKK performs best for small problems and large problems, while SS performs best for problems of intermediate size, with the specific cross-over point depending on the precision of the integers. Surprisingly, we found that the well-known dynamic programming algorithm is dominated by CKK.

For multi-way partitioning, we introduced a new version of CKK that minimizes the makespan, instead of the difference between the largest and smallest subset sums, and a new pruning rule. We also introduced a new method to reduce the number of duplicate partitions generated by RNP. For problems without perfect partitions, this improves the performance of RNP for all numbers of subsets k, with the improvement increasing with increasing k, with speedups of over a factor of twenty-six for seven-way partitioning. Holding the precision and number of subsets fixed, while increasing the problem size n increases the number of perfect partitions. In that case, CGA performs best for small values of n, RNP is the best choice for intermediate values of n, and CKK performs best for large values of n.

A natural question to ask is where are the hardest problems? First, it depends on the algorithm, since the hardest problems for one algorithm are not the hardest for another. In general, the more perfect partitions there are of, the easier the problem instance is, since a perfect partition can be returned immediately as optimal. The lower the precision of the integers, the more common perfect partitions become, making the problems easier. To eliminate perfect partitions, we can always increase the precision of the integers. Thus, even hard two-way instances can be constructed by making the precision high enough. If there are no perfect partitions, however, then increasing the precision has no effect on the performance of any of these algorithms, except for dynamic programming, which we showed is not competitive. If we hold the precision p and number of subsets k fixed, and increase the problem size n, then problems get harder until perfect partitions appear, and then they get easier. Table 1 shows the values of n, as a function of p and k, where approximately half the problems have perfect partitions. In general, for fixed p and fixed n, problems with larger k are more difficult, until k begins to approach n, at which point they get easier again.

One of the surprising results of this work is that in spite of the simplicity of the problem, particularly two-way partitioning, the choice of the best algorithm for finding optimal solutions is rather complex, and significant progress is still being made. There are several undominated algorithms, and the best choice depends on n, k, and the precision p of the integers. Since most real scheduling problems are much more involved, and many of them contain this problem as a special case, it is likely that designing the best algorithms for those problems will be at least as complex. In particular, this work highlights the important role that precision may play in any combinatorial problem involving numerical quantities.

Since the submission of this paper, we have been exploring the bin-packing approach to this problem, using improvements to an alternative bin-packing algorithm called bin-completion (Korf 2002; 2003). Our preliminary results indicate that this outperforms RNP for $k = 7$ and $k = 6$, but is more than an order of magnitude slower than RNP for $k = 5$, and orders of magnitude slower for smaller values of k. This work will be described in a forthcoming paper.

Acknowledgments

This work was supported by NSF grant IIS-0713178. Thanks to Gleb Belov for providing his BCP code to us and for other valuable assistance.

References

Dell'Amico, M.; Iori, M.; Martello, S.; and Monaci, M. 2008. Heuristic and exact algorithms for the identical parallel machine scheduling problem. *INFORMS Journal on Computing* 20(23):333–344.

Garey, M. R., and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman.

Horowitz, E., and Sahni, S. 1974. Computing partitions with applications to the knapsack problem. *Journal of the ACM* 21(2):277–292.

Howgrave-Graham, N., and Joux, A. 2010. New generic algorithms for hard knapsacks. In *Proceedings of EUROCRYPT 2010*, 235–256. LNCS 6110.

Karmarkar, N., and Karp, R. M. 1982. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, C.S. Division, University of California, Berkeley.

Korf, R. E. 1998. A complete anytime algorithm for number partitioning. *Artificial Intelligence* 106(2):181–203.

Korf, R. E. 2002. A new algorithm for optimal bin packing. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-02)*, 731–736.

Korf, R. E. 2003. An improved algorithm for optimal bin packing. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-03)*, 1252–1258.

Korf, R. E. 2009. Multi-way number partitioning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-09)*, 538–543.

Korf, R. E. 2010. Objective functions for multi-way number partitioning. In *Proc. of the Symposium on Combinatorial Search (SOCS-10).*

Korf, R. E. 2011. A hybrid recursive multi-way number partitioning algorithm. In *Proc. of IJCAI-11*, 591–596.

Schroeppel, R., and Shamir, A. 1981. A $T = O(2^{n/2}), S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal of Computing* 10(3):456–464.