Multi-Way Number Partitioning

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

Abstract

The number partitioning problem is to divide a given set of integers into a collection of subsets, so that the sum of the numbers in each subset are as nearly equal as possible. While a very efficient algorithm exists for optimal two-way partitioning, it is not nearly as effective for multi-way partitioning. We develop two new linear-space algorithms for multi-way partitioning, and demonstrate their performance on three, four, and five-way partitioning. In each case, our algorithms outperform the previous state of the art by orders of magnitude, in one case by over six orders of magnitude. Empirical analysis of the running times of our algorithms strongly suggest that their asymptotic growth is less than that of previous algorithms. The key insight behind both our new algorithms is that if an optimal k-way partition includes a particular subset, then optimally partitioning the numbers not in that set $k-1$ ways results in an optimal k-way partition.

1 Introduction: Number Partitioning

Given a set of integers, the number partitioning problem is to divide them into a collection of mutually exclusive and collectively exhaustive subsets so that the sum of the numbers in each subset are as nearly equal as possible. For example, given the integers (4, 5, 6, 7, 8), if we divide them into the subsets (4, 5, 6) and (7, 8), the sum of the numbers in each subset is 15, and the difference between the subset sums is zero, which is optimal. For partitioning into more than two subsets, the objective function to be minimized is the difference between the maximum and minimum subset sums. We focus here on optimal solutions. All the algorithms described here use space that is only linear in the number of numbers.

Number-partitioning is NP-complete, and one of the easiest NP-complete problems to describe. It is also a very simple scheduling problem, often referred to as "multi-processor scheduling"[Garey and Johnson, 1979]. For example, given a set of jobs, each with an associated completion time, and two or more identical processors, assign each job to a processor to minimize the total time to complete all the jobs.

1.1 Easy-Hard-Easy Complexity Transition

An important feature of this problem is that its difficulty shows a classic easy-hard-easy transition as the problem size increases. Let n be the number of numbers, k the number of subsets, and m the maximum possible value. Problems with small n are easy since there are only k^n partitions. As n grows, the number of partitions grows as k^n, but the number of different possible values for the difference of the subset sums grows only as $n \cdot m$. Thus, for fixed m and k and large n, there are many more partitions than subset sum differences, and hence most differences have many associated partitions.

In particular, if the sum of all the numbers is divisible by k, there can be a subset sum difference of zero, and otherwise, the minimum possible subset sum difference is one. Once such a "perfect partition" is found, search is terminated. For uniform random instances, as n grows large, the number of perfect partitions increases, making them easier to find, and the problem easier. The most difficult problems occur where the probability of a perfect partition is about one-half. Much has been written about this "phase transition", e.g. [Mertens, 1998], but it is tangential to our work, as we are concerned exclusively with algorithms for finding optimal partitions.

2 Previous Work

Here we briefly review previous work on this problem. For simplicity, we focus on two-way partitioning, but all these algorithms are extensible to multi-way partitioning as well.

2.1 Greedy Heuristic

The obvious greedy heuristic for this problem is to sort the numbers in decreasing order, and then assign each number in turn to the subset with the smaller sum so far. For example, given the numbers (8,7,6,5,4), we would assign the 8 and 7 to different subsets, the 6 to the subset with the 7, the 5 to the subset with the 8, and finally the 4 to either subset, yielding for example the partition (8,5,4) and (7,6), with a subset difference of $17 - 13 = 4$. The average solution quality of this heuristic is on the order of the smallest number.

2.2 Complete Greedy Algorithm (CGA)

This heuristic is easily extended to a complete greedy algorithm (CGA) [Korf, 1995; 1998]. First we sort the numbers in decreasing order, and then search a binary tree, where each

level assigns a different number, and each branch point alternately assigns that number to one subset or the other. Each leaf of this tree corresponds to a complete partition.

Several pruning rules can improve the efficiency of CGA: 1) If a complete partition is found with a difference of zero or one, it is returned and the search is terminated. 2) If the current difference between the two partial subset sums is at least the sum of the numbers not yet assigned, then all remaining numbers are assigned to the subset with the smaller sum, terminating that branch. 3) If both partial subsets have the same sum, the next number is only assigned to one of them, to eliminate duplicate nodes. Finally, to minimize the time to find a perfect partition, we always assign the next number to the subset with the smaller sum first.

CGA is easily extended to partitioning into k subsets. Instead of a binary tree, we search a k-ary tree, where at each branch the corresponding number is assigned to one of the k subsets. The second pruning rule above is replaced by the following. Let t be the sum of all the numbers, s the current largest subset sum, and d the difference of the best complete partition found so far. If $s - \frac{t-s}{k-1} \geq d$, terminate this branch. The reason is that the best we could do would be to perfectly equalize the remaining $k-1$ subsets, and if this would result in a partition no better than the best so far, there is no reason to continue searching that path. At each branch, we place the next number in the subsets in increasing order of their sums, to minimize the time to find a good solution.

2.3 Karmarkar-Karp Heuristic (KK)

A heuristic much better than greedy was called set differencing by its authors [Karmarkar and Karp, 1982], but is usually referred to as the KK heuristic. It places the two largest numbers in different subsets, without determining which subset each goes into. This is equivalent to replacing the two numbers with their difference. For example, placing 8 in subset A and 7 in subset B is equivalent to placing their difference of 1 in subset A, since we can always subtract the same amount from both sets without affecting the solution. Swapping their positions is equivalent to placing the 1 in subset B. The KK heuristic repeatedly replaces the two largest numbers with their difference, inserting the new number in the sorted order, until there is only one number left, which is the final partition difference. In our example, this results in the series of sets (8,7,6,5,4), (6,5,4,1), (4,1,1), (3,1), (2). Some additional bookkeeping is required to extract the actual partition, which in this case is (7,5,4) and (8,6), with a partition difference of 16-14=2. The solution quality of this heuristic is the last remaining number, which is much smaller than the smallest original number, due to the repeated differencing.

2.4 Complete Karmarkar-Karp Algorithm (CKK)

We extended the KK heuristic to the complete Karmarkar-Karp algorithm (CKK) [Korf, 1998]. While the KK heuristic always places the two largest numbers in different subsets, the only other option is to assign them to the same subset. This is done by replacing the two largest numbers by their sum. CKK searches a binary tree where at each node the left branch replaces the two largest numbers by their difference, and the right branch replaces them by their sum. By searching from left to right, the first solution found is the KK solution. If we find a complete partition with a difference of zero or one, the search terminates. In addition, if the largest number is greater than or equal to the sum of the remaining numbers, we place all the remaining numbers in the opposite subset from the largest, since this is the best we can do. For two-way partitioning, CKK is slightly faster than CGA for problem instances without a perfect partition, but is much faster for problem instances with many perfect partitions, since it finds one much faster [Korf, 1998].

The extension of the KK heuristic and CKK algorithm to multi-way partitioning is more complex than for the greedy heuristic and CGA [Korf, 1998]. We describe here their extension to three-way partitioning. A state of the KK algorithm is described by a set of triples of partial subset sums, with each triple sorted in decreasing order, and the triples sorted in decreasing order of their largest sum. Initially, each number is in a separate triple, with zero for the remaining numbers. For example, the initial state of a three-way KK partition of the set (4,5,6,7,8) would be ((8,0,0),(7,0,0),(6,0,0),(5,0,0),(4,0,0)). At each step of the KK heuristic, if (a, b, c) and (x, y, z) are the triples with the largest numbers, they are replaced with $(a + z, b + y, c + x)$, which is then normalized by subtracting the smallest element of the triple from each element. This combination is chosen to minimize the largest values. In our example, this results in the set ((8,7,0)(6,0,0)(5,0,0)(4,0,0)). Combining the next two largest triples results in the set, ((8,7,6)(5,0,0)(4,0,0)), which after normalization is represented by ((5,0,0),(4,0,0),(2,1,0)). Combining the next two results in ((5,4,0),(2,1,0)), and combing the last two produces (5,5,2) or (3,3,0) after normalizing, for a final partition difference of 3, corresponding to the partition (8), (7,4), and (6,5), which happens to be optimal in this case.

For the complete CKK algorithm, at each node we combine the two triples with the largest sums in every possible way rather than just one way, branching on each combination. For three-way partitioning, there are six ways to combine two triples, corresponding to different permutations of three elements, and for k-way partitioning, there are $k!$ branches at each node. For example, given the triples (a,b,c) and (x,y,z), their different possible combinations are $(a+x, b+y, c+z)$, $(a+x, b+z, c+y)$, $(a+y, b+x, c+z)$, $(a+y, b+z, c+x)$, $(a+z, b+x, c+y)$, and $(a+z, b+y, c+x)$. Since the smallest sum of each k-tuple is always zero after normalization, we only maintain tuples of $k-1$ sums for k-way partitioning.

2.5 Pseudo-Polynomial-Time Algorithms

Technically, number partitioning is not strongly NP-complete, but can be solved in pseudo-polynomial-time by dynamic programming. This requires memory that is proportional to $n(k-1) \cdot m^{k-1}$ for k-way partitioning of n numbers with a maximum value of m. As a result, these algorithms are not practical for multi-way partitioning. For example, three-way partitioning of 40 7-digit integers requires a petabyte (10^{15}) of storage.

2.6 Previous State-Of-The-Art

For two-way partitioning, CKK is the algorithm of choice. It is slightly faster than CGA without perfect partitions, and

much faster with many perfect partitions. In fact, if we restrict the numbers to 32-bit integers, arbitrarily large instances can be solved in less than a second. The success of this algorithm probably contributed to the complete absence of papers on algorithms for optimal number partitioning in the last decade.

For three-way partitioning, CKK also outperforms CGA. To solve large problems, however, the precision of the numbers must be restricted. For 3-way partitioning of six-digit numbers, for example, CKK takes about a minute to solve the hardest random instances, which contain 30 numbers. We are not aware of any literature on optimal four-way partitioning.

For partitioning four or more ways, CGA is much more efficient than CKK. The reason is that CKK gets increasingly complex as k increases, increasing the constant time per node generation. Our implementation of CKK for three-way partitioning is specialized to three subsets, but this is impractical with more subsets, since the number of ways of combining tuples is $k!$. Specializing CGA for a particular value of k is easy, however. On problems without perfect partitions, our general implementation of CKK for four-way partitioning is ten times slower per node than our specialized CGA.

We now present our new work.

3 Principle of Optimality

The key property of multi-way partitioning that underlies both our new algorithms is the following principle of optimality. In any optimal three-way partition, the numbers in any two of the subsets must be optimally partitioned two ways. More generally, if an optimal k-way partition includes a particular subset, then optimally partitioning the numbers not in that subset $k-1$ ways will yieldy an optimal k-way partition.

To prove this, there are three cases to consider: 1) If the given subset has neither the smallest nor the largest sum, then it doesn't affect the partition difference, and the remaining numbers must be optimally partitioned $k-1$ ways to minimize the partition difference. 2) If the given subset has the smallest sum, then an optimal $k-1$ way partition of the remaining numbers minimizes the largest subset sum, thus minimizing the overall partition difference. 3) Similarly, if the given subset has the largest sum, then an optimal $k-1$ way partition of the remaining numbers maximizes the smallest subset sum, thus minimizing the overall partition difference.

Note that it is not the case that in any optimal partition, any collection of subsets must be optimally partitioned. For example, in an optimal four-way partition, the two subsets with intermediate sums need not be optimally partitioned two ways, since they don't affect the partition difference.

4 Sequential Number Partitioning (SNP)

We now introduce the first of our two new algorithms, called *sequential number partitioning* (SNP), using three-way partitioning as our first example. We first choose one complete subset, and then optimally partition the remaining numbers two ways. Since we can't identify a priori a subset in an optimal partition, we generate each subset that could possibly be part of an optimal three-way partition, and for each such subset, we use CKK to optimally partition the remaining numbers two ways to get a complete three-way partition.

The standard way to generate subsets is to search an inclusion-exclusion binary tree, in which leaf nodes represent all possible subsets. Each level of the tree corresponds to a particular number, and at each branch we either include the corresponding number in the subset, or exclude it. For example, the left subtree of the root contains all subsets that include the first number, and the right subtree of the root contains all subsets that exclude the first number.

We reduce the number of subsets considered by an upper bound on their sum. To eliminate duplicate partitions that differ only by permuting the subsets, we require that the first subset have the smallest sum. If t is the sum of all the numbers, the first subset sum can be no larger than $t/3$, or one of the remaining subsets would have to have a smaller sum.

We also enforce a lower bound on the first subset sum. For a given first subset, the best we could do would be to perfectly partition the remaining numbers two ways. Thus, the difference between half the sum of the numbers excluded from the first subset, and the first subset sum, is a lower-bound on the three-way partition difference. If this difference is greater than or equal to the best three-way partition difference found so far, the corresponding first subset cannot be part of a better solution. In particular, if t is the sum of all the numbers, and d is the difference of the best three-way partition found so far, then the first subset sum must be at least $(t-2d)/3$.

Given these lower and upper bounds on the first subset sum, we prune the inclusion-exclusion tree as follows: Any branch where the sum of the numbers included so far exceeds the upper bound is pruned. Also, any branch where the sum of the included numbers, and the numbers not yet included nor excluded is less than the lower bound is also pruned.

To efficiently search this inclusion-exclusion tree, we first sort the numbers in decreasing order, and decide whether to include or exclude the largest numbers first. The reason is that the largest numbers have the greatest impact on the sum so far, and the sum of the remaining numbers, resulting in more pruning near the root of the tree.

The overall algorithm for three-way partitioning works as follows: We first run the KK heuristic to get an approximate three-way partition, and use this to compute the lower bound on the first subset sum. Next we search an inclusion-exclusion tree for subsets within the lower and upper bounds. For each such subset, we optimally partition the remaining numbers two ways using CKK. We then compute the corresponding three-way partition difference. If this solution is better than the best one found so far, we increase the lower bound on the first subset sum, and continue exploring all first subsets that could possibly lead to a better solution.

Extending this algorithm to four-way partitioning is straightforward. We first run KK to get an approximate four-way partition with a difference of d. We then select first subsets by searching an inclusion-exclusion tree with an upper bound of $t/4$, and a lower bound of $(t-3d)/4$. For each such first subset, we call SNP to optimally partition the remaining elements three ways. This involves selecting a second subset by searching another inclusion-exclusion tree, with a sum greater than or equal to that of the first subset, since the subset sums are chosen in non-decreasing order. The extension to more than four subsets follows analogously.

5 Recursive Number Partitioning (RNP)

The main drawback of SNP is that CKK is only used to determine the last two subsets, leaving the much less efficient inclusion-exclusion tree searches to determine all the previous subsets. Here we propose an alternative algorithm, which we call recursive number partitioning (RNP).

We begin with the example of four-way partitioning. We first run the KK heuristic to get an approximate four-way partition. Then we divide all the numbers into two subsets, each of which will later be partitioned two ways. This top-level partitioning is done in every way that could possibly lead to a four-way partition better than the best one found so far. For a given top-level partition, the best we could do would be to perfectly partition each of the two subsets. Thus, half of the larger subset sum minus half of the smaller subset sum is a lower bound on the difference of any four-way partition derived from a given top-level partition. This bound must be less than the current best four-way partition difference for a given top-level partition to lead to a better solution. Thus, the current best solution imposes an upper bound on the difference of the top-level partitions. The top-level partitioning is done using CKK with this upper bound, but returning all two-way partitions within the bound, rather than a single optimal partition. We use CKK instead of CGA because it finds better top-level partitions sooner.

For each such top-level partition, we use CKK to optimally partition the smaller of the two subsets. If the resulting partition, combined with a perfect partition of the larger subset, would lead to a better four-way partition, we optimally partition the larger subset as well. We compute the overall partition difference by subtracting the smallest of the four subset sums from the largest. If this four-way partition is better than the best one found so far, we update the upper bound on the initial top-level partitioning. In either case, we continue generating top-level partitions within the bound, in the order generated by CKK, until all such partitions that could lead to a better four-way partition have been explored.

For three-way partitioning, SNP and RNP are identical, but they differ for four or more subsets. For five-way partitioning, for example, RNP first searches an inclusion-exclusion tree to select a first subset, and then calls RNP to optimally partition the remaining numbers four ways. For six-way partitioning, RNP first partitions all the numbers two ways, and then calls RNP, or equivalently SNP, to optimally partition each of the two subsets into three subsets each. In general, for an even number of subsets, RNP starts with two-way partitioning at the top level, and then recursively partitions each half. With an odd number of subsets, RNP searches an inclusion-exclusion tree for a first subset, then calls RNP to divide the remaining numbers two ways, etc.

6 Experimental Results

We experimented with three, four, and five-way partitioning. Each timing shows the total time to solve 100 problem instances, in the form days:hours:minutes:seconds. The average time for a single instance is one-hundredth of these values. The numbers were uniformly distributed from zero to the maximum value. In our first set of experiments, maximum values were chosen to allow solving the hardest instances with the existing algorithms. All experiments were run on an IBM Intellistation with a two gigahertz AMD Opteron processor. All the algorithms run in space that is linear in the number of numbers n, since all searches are depth first.

6.1 Three-Way Partitioning

For three-way partitioning, CKK is the best previous algorithm, and was used for comparison. We choose a maximum value of ten million, since these are the largest integers for which CKK could solve the hardest instances. Table 1 shows our results. The first column contains the problem size N, or number of numbers. The second column gives the average optimal partition difference. The third column shows the running time for CKK to optimally solve all 100 problem instances, and the fourth column shows the running time for SNP to solve the same instances. RNP is the same as SNP for three-way partitioning. The last column shows the running time of CKK divided by the running time of SNP, which represents the speedup over the previous state of the art.

N	Diff	CKK	SNP	CKK/SNP
25	84.48	4:01	:01	212
26	51.10	10:46	:02	321
27	31.50	30:51	:04	501
28	15.38	1:17:51	:06	732
29	9.91	3:41:39	:12	1,142
30	5.68	8:52:50	:20	1,589
31	3.25	1:00:47:51	:34	2,613
32	1.98	2:01:58:24	:49	3,702
33	1.31	4:10:15:02	1:01	6,260
34	.87	5:01:52:22	:43	10,131
35	.59	5:13:23:05	:19	24,884
36	.62	3:12:43:33	:14	21,973
37	.71	1:14:44:28	:09	15,389
38	.66	1:08:39:13	:10	13,436
39	.61	1:04:20:18	:08	13,009
40	.67	:23:25:56	:07	11,619

Table 1: Three-Way Partitioning of 100 7-Digit Integers

The first thing to notice is that the running times of both algorithms exhibit the easy-hard-easy transition characteristic of number partitioning. We don't know why the most difficult problems for CKK contain 35 numbers while the most difficult problems for SNP contain only 33 numbers.

The next thing to notice is that SNP is much faster than CKK. While CKK takes over an hour on average to solve the hardest problem instances, SNP solves the hardest instances in less than a second on average. For partitioning 35 numbers, SNP runs almost 25,000 times faster than CKK.

6.2 Four-Way Partitioning

Table 2 shows our results for four-way partitioning, in the same format as Table 1. For four-way partitioning, CGA is faster than CKK, and thus CGA is used for comparison. Here we used a maximum value of a hundred thousand, as these are the largest integers for which CGA could solve the hardest instances. We also ran both SNP and RNP on these problems.

The last column shows the ratios of the running times of RNP and CGA. In the empty positions, SNP and RNP took less than half a second to solve all 100 problem instances.

N	Diff	CGA	SNP	RNP	RNP/CGA
20	79.24	:06			
21	56.55	:22			
22	33.64	1:18			
23	21.52	3:59	:01	:01	422
24	14.52	16:30	:01	:01	1,013
25	9.76	45:08	:02	:02	1,605
26	5.91	2:37:23	:03	:03	3,551
27	3.59	9:04:02	:05	:04	7,871
28	2.24	1:05:58:18	:07	:06	17,310
29	1.54	2:09:47:12	:09	:06	32,059
30	.92	4:23:25:17	:09	:04	104,197
31	.76	7:05:59:05	:09	:03	232,669
32	.70	5:03:19:33	:11	:01	383,066
33	.71	8:22:49:28	:17		1,769,720

Table 2: Four-Way Partitioning of 100 5-Digit Integers

Again we see running times going from days to seconds on the hardest problems. RNP partitions 33 numbers over a million times faster than CGA, reducing the running time from almost 9 days to less than half a second for 100 problem instances. RNP is consistently faster than SNP. The running time of SNP keeps increasing on the harder problems, while that of RNP decreases beyond 29 numbers.

6.3 Five-Way Partitioning

Table 3 shows our results for five-way partitioning, in the same format as Tables 1 and 2. As for four-way partitioning, CGA is more efficient than CKK, and thus CGA is used for comparison. Here we used a maximum value of ten thousand, since these were the largest integers for which CGA could solve the hardest problems. We also ran both SNP and RNP on these problems. The last column shows the ratios of the running times of RNP and CGA.

N	Diff	CGA	SNP	RNP	RNP/CGA
20	38.95	:11	:01	:01	18
21	24.01	:28	:01	:01	31
22	17.08	1:24	:03	:02	50
23	10.16	7:20	:05	:03	156
24	6.89	31:02	:11	:06	334
25	4.84	2:15:35	:22	:10	828
26	2.98	9:01:11	:54	:20	1,640
27	1.98	1:10:19:00	2:10	:39	3,197
28	1.40	3:01:09:18	4:26	1:12	3,666
29	.99	6:04:27:31	10:14	2:08	4,183
30	.83	9:00:10:16	19:23	4:14	3,066

Table 3: Five-Way Partitioning of 100 4-Digit Integers

Again we see orders of magnitude speedup, although not as large as for three and four-way partitioning. The reason may be the inability of CGA to solve problems with larger values. For partitioning 29 numbers, RNP runs over four thousand times faster than CGA. Here we see a larger difference between the performance of RNP and SNP. For example, RNP partitions 29 or 30 numbers almost five times faster than SNP.

6.4 Empirical Asymptotic Time Complexity

These dramatic performance improvements, and the fact that the ratios of the running times increase with increasing problem difficulty, suggest that both SNP and RNP are asymptotically faster than CKK and CGA. Due to the pruning rules employed, however, these algorithms are very difficult to analyze. The next best thing is to empirically estimate their asymptotic complexity. We do this by computing the ratios of the running times for pairs of successively larger problem instances. To do this, we chose problem instances without many perfect partitions, since as perfect partitions become more common, the running times of these algorithms decrease. Furthermore, without perfect partitions, the running times of these algorithms are insensitive to the size of the numbers. Thus, we chose integers uniformly distributed from zero to 100 million, since these are the largest integers we could handle using 32-bit arithmetic with up to 40 numbers.

Table 4 shows the results of partitioning these numbers three, four, and five ways. The left-most column shows the number of numbers. The next four columns show results for three-way partitioning, the middle four for four-way partitioning, and the last four for five-way partitioning. Each group of columns starts with the total running time for the best previous algorithm to solve 100 uniform random problem instances, which is CKK for three-way partitioning, and CGA for four and five-way partitioning. The next column in each group gives the running time for the given number of numbers divided by the running time for one less number, from the line immediately above it. This ratio is the geometric growth between the two levels. The next two columns in each group give the corresponding data for RNP.

For three-way partitioning, the geometric growth rate for CKK ranges between about 2.5 and 3. The corresponding growth rate for RNP stabilizes at about 1.8 for up to 35 numbers. For more numbers, the growth rate decreases as the problems get easier because perfect partitions become more common. This shows that RNP can optimally solve arbitrarily large three-way problems with up to 8-digit numbers.

For four-way partitioning, the geometric growth of CGA ranges between about 3 and 4. The corresponding growth rate for RNP varies from about 1.8 to 1.9. This data oscillates between even and odd problem sizes. Again, we see clear evidence of an asymptotic improvement. If we compare three-way to four-way partitioning with RNP, four-way partitioning is only slightly more expensive for hard problems.

For five-way partitioning, the geometric growth rate of CGA ranges between about 3 and 5. The corresponding growth rate for RNP ranges between about 2 and 2.4 for hard problems, with consecutive outliers at 2.98 and 1.76. The geometric mean of these two values is 2.29.

These data strongly suggest that the asymptotic growth rate of RNP is lower than that of CKK or CGA. Since these values represent the base of the exponential complexity of the algorithms, any significant difference between them will eventually produce arbitrarily large constant factor speedups.

N	Three-Way Partitioning				Four-Way Partitioning				Five-Way Partitioning			
	CKK	ratio	RNP	ratio	CGA	ratio	RNP	ratio	CGA	ratio	RNP	ratio
20	:02				:07				:06		:01	
21	:05	2.36			:20	2.96			:24	3.77	:01	1.88
22	:13	2.65			1:06	3.39			1:21	3.40	:02	1.47
23	:33	2.62			3:32	3.19			5:01	3.72	:03	1.75
24	1:25	2.58			11:17	3.20			17:35	3.51	:06	2.00
25	4:01	2.83	:01		44:33	3.95	:02		1:30:09	5.13	:10	1.70
26	10:01	2.50	:02	1.75	2:07:17	2.86	:03	1.64	4:40:18	3.11	:27	2.63
27	28:51	2.88	:04	1.82	7:51:33	3.70	:05	1.74	20:58:41	4.49	:36	1.32
28	1:23:21	2.89	:07	1.80	1:06:34:13	3.89	:09	1.71	4:10:23:28	5.07	1:33	2.57
29	3:29:42	2.52	:11	1.75	3:23:52:38	3.14	:15	1.74			3:06	2.00
30	10:22:52	2.97	:22	1.89			:27	1.80			6:15	2.02
31	1:01:47:27	2.48	:39	1.83			:48	1.78			12:55	2.07
32			1:13	1.87			1:29	1.85			26:15	2.03
33			2:11	1.78			2:38	1.78			55:38	2.12
34			3:51	1.78			4:50	1.83			2:46:04	2.98
35			6:54	1.79			8:37	1.78			4:52:20	1.76
36			11:16	1.63			16:14	1.88			11:38:42	2.39
37			15:10	1.35			29:13	1.80			2:02:33:05	2.35
38			12:32	.83			55:05	1.89				
39			6:42	.53			1:38:56	1.80				
40			2:54	.43			3:09:34	1.92				

Table 4: Asymptotic Growth of Three, Four, and Five-Way Partitioning of 100 8-Digit Integers

7 Further Work

To optimize performance, the implementations of all these algorithms were specialized for partitioning into three, four, and five subsets. The next step is to partition numbers into more subsets, using general implementations of these algorithms, to see if the same trends continue. In addition, a challenging task will be to prove theoretically that our new algorithms are in fact asymptotically faster than CKK and CGA.

8 Conclusions

We have introduced two new algorithms for multi-way number partitioning, sequential number partitioning (SNP) and recursive number partitioning (RNP). Both algorithms rely on the insight that if an optimal k-way partition of a set of numbers includes a particular subset, then optimally partitioning the remaining numbers not in that subset $k-1$ ways results in an optimal k-way partition. Both algorithms dramatically outperform the best previous algorithms for this problem, the complete greedy algorithm (CGA) and the complete Karmarkar-Karp algorithm (CKK), by orders of magnitude. For four-way partitioning of 33 5-digit numbers, for example, RNP runs over a million times faster than CGA. For three-way partitioning of eight-digit numbers, SNP, or equivalently RNP, can solve the hardest random problem instances in less than ten seconds on average. For partitioning into four or more sets, RNP outperforms SNP. An empirical exploration of the performance of RNP on problem instances without perfect partitions strongly suggests that it is asymptotically faster than both CGA and CKK. In addition to the specific contributions to number partitioning, this work demonstrates that better search techniques can still achieve enormous speedups over the previous state-of-the-art in simple combinatorial problems.

Acknowledgments

This research was supported by NSF grant No. IIS-0713178. Thanks to Satish Gupta and IBM for providing the machine these experiments were run on,

References

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

[Karmarkar and Karp, 1982] Narendra Karmarkar and Richard M. Karp. The differencing method of set partitioning. Technical Report UCB/CSD 82/113, Computer Science Division, University of California, Berkeley, 1982.

[Korf, 1995] Richard E. Korf. From approximate to optimal solutions: A case study of number partitioning. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 266–272, Montreal, Canada, Aug 1995.

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

[Mertens, 1998] Stephen Mertens. Phase transition in the number partitioning problem. *Physical Review Letters*, 81(20):4281–4284, November 1998.