PARALLEL SOLUTION OF EXTREMELY LARGE KNAPSACK PROBLEMS

by

Michael C. Ferris

Computer Sciences Technical Report #842

April 1989

Parallel Solution of Extremely Large Knapsack Problems*

Michael C. Ferris†

April 1989

Abstract. We shall describe an parallel algorithm for solving the knapsack feasibility problem, also known as the subset sum problem. The use of a random branching technique is described and its implementation on a parallel processor is discussed. Computational results show this to be an effective method for solving large problems. Using this approach we have solved problems with as many as 2 million variables in an average of 800 seconds on the Sequent Symmetry parallel processor. Furthermore, a coarse parallelization overcomes some of the problems that are present when serial algorithms are used to solve the knapsack problem.

Key words. Knapsack problems, subset sum problems, parallel algorithms, random branching

Abbreviated title. Knapsack problem solution

*This material is based on research supported by National Science Foundation Grant DCR-8521228 and Air Force Office of Scientific Research Grant 86-0172

†Computer Sciences Department, University of Wisconsin, Madison, Wisconsin 53706

1 Introduction

Our principal example in this paper will be the subset sum problem[9] (also known as the knapsack problem or the 0–1 feasibility knapsack problem). The problem can be described as follows:

Given a finite set S, a size $a(i)$ in the set of positive integers Z^+ for each $i \in S$, and a positive integer b, is there a subset $S' \subseteq S$ such that the sum of the sizes of the elements in S' is exactly b?

Karp[9] has shown that the the subset sum problem is NP–complete by using a reduction from the partition problem:

Given a finite set S, a size $a(i) \in Z^+$ for each $i \in S$, is there a subset $S' \subseteq S$ such that

$$\sum_{i \in S'} a(i) = \sum_{i \in S \setminus S'} a(i)$$

A more complete discussion of the relationship between these problems and the theory of NP–completeness is given by Garey and Johnson[4].

It is noted that the magnitude of the input is crucial in the development of algorithms for the solution of the subset sum problem, since both the partition problem and the subset sum problem (which are NP–complete problems) can be solved in pseudo–polynomial time by dynamic programming[4]. (An algorithm is **pseudo–polynomial** if its time complexity can be bounded above by a polynomial in the length of the input and the magnitude of the input. For example, the length of the input for the subset sum problem is n, the magnitude is b and a dynamic programming algorithm can be produced to solve this problem in time $O(nb)$. However, this is not in P since this cannot be bounded by any polynomial in $n \log b$, the number of bits needed to encode the input.)

These problems have been investigated thoroughly in the literature, although mainly in the context of 0–1 knapsack minimization problems. The best known methods for solving these problems to optimality are the branch–and–bound methods first conceived for this problem by Kolesar[10] and Greenberg and Hegerich[5], the partition methods of Horowitz and Sahni[6] and Ingargiola and Korsh[7] and the method due to Balas and Zemel[1]. A spe-

cialized dynamic programming and branch–and–bound technique for the subset sum problem was given by Martello and Toth[12]. Most of these methods are based on branch-and-bound techniques and require a preliminary sorting of the elements, which is the computationally dominant part of the proposed algorithms. Balas and Zemel[1] have defined the concept of a core of the problem, and Fayard and Plateau[3] have presented an algorithm based on this idea. For further details on these methods, the reader is referred to a survey paper of Martello and Toth[14]. A very recent paper[15] gives an improved algorithm for the knapsack problem and does report some experience with large knapsack problems, giving results on problems up to a size of 10,000. Along with the results quoted in [1], these represent the largest problems which have been solved to the authors knowledge. The computational results given by these algorithms are impressive; however, there are two points to be noted. The first is that the reduction techniques given for the knapsack problem are not appropriate for the subset sum problem and so all the advantages from these heuristics are not appropriate. Secondly, the results reported on the subset sum problem in both of the aforementioned papers have the variables a_i generated from a uniform distribution with fixed (and rather small) upper bound. It is well known that the size of the coefficients changes the degree of difficulty of the problems, and this was also noted in our experimental work. We have chosen not to report on problems of this type as this avoids the essential difficulties associated with the subset sum problem. However, on such problems, our algorithm does perform as well as the the ones cited here.

Other work has been concerned with approximation algorithms and notable results are given by Lawler[11], Sahni[17] and Martello and Toth[13]. Due to the conceptual simplicity of the subset sum problem, there are a large number of equivalent formulations for the problem. Recently, Moré and Vavasis[16] have given an $O(n \log n)$ algorithm to find a strict local solution to a concave quadratic formultaion of the subset sum problem. In contrast, our principal aim is to use simple heuristics (the ideas of which can be easily parallelized) in order to find actual solutions of the subset sum problem. For the remainder of this section will will describe the form of the problem that we shall solve and how it is used in the computational development we describe in the sequel.

We shall rewrite the problem in the following manner by introducing a vector of binary variables x_i, each of which is 1 or 0 according as the ith element of the set S is in the set S' or not:

$$\text{Find} \quad x \in \{0,1\}^n \text{ such that } a^T x = b \tag{1}$$

where $a_1, \ldots, a_n$ and b are positive integers. Note that we denote the size of the ith element by an integer a_i.

The algorithm we give in this paper is a method for finding a solution of the subset sum problem, if one exists. The method will not terminate if there is no solution to the subset sum problem but continues looking for some solution. In the examples we have tested, this fact did not create any difficulties even though the problems we generated were not guaranteed to have a solution. This approach can be theoretically justified by noting that under appropriate assumptions Karmarkar, Karp, Lueker and Odlyzko[8] have shown that the probability that the subset sum problem has solution tends to one with n.

We first generate an optimization problem from (1). Essentially, the following problem is concerned with the minimization of a residual (the error in the satisfaction of the equality) over the vertices of a box. The following definition formalizes these notions.

Definition 1 *Given a vector $a \in \mathbb{R}^n$, the* **box**, *$B(a)$, is defined by:*

$$B(a) := \{z \in \mathbb{R}^n \mid 0 \leq z \leq a\}$$

The **vertex set** *of this box, $V(a)$, is given by:*

$$V(a) := \{z \in \mathbb{R}^n \mid z_i = 0 \text{ or } z_i = a_i\}$$

Problem (1) is thus seen to be equivalent to

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}(e^T z - b)^2 \\ \text{subject to} & z \in V(a) \end{array} \tag{2}$$

with z variables related to x variables by the scale transformation

$$z_i = a_i x_i, \text{ for each } i = 1, \ldots, n \tag{3}$$

2 Serial solution of subset sum problems

In the following discussion we will describe a naive procedure to find a solution of the subset sum problem, if one exists. The technique does not address the situation where the problem has no solution, but is able to solve large problems in relatively small computational times and with linear storage.

We shall be concerned with the following notions.

Definition 2 *$\bar{x}$ is said to be a* **global** *solution of the minimization problem*

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in S \end{array}$$

if $f(\bar{x}) \leq f(x)$, for all $x \in S$. If S is a compact polyhedral set (for instance $B(a)$), then a vertex of S, $\bar{v}$, is a **local star** *solution of the problem if $f(\bar{v}) \leq f(v)$, for every adjacent vertex v of $\bar{v}$.*

The algorithm uses the formulations of the subset sum problem given in (2). It can be described as follows.

Algorithm

Initialize: Given a feasible vertex, $z \in V(a)$, set $k = 0$ and calculate the residual

$$\left| e^T z - b \right|$$

Iteration:

Search: Given an index i, move to an adjacent vertex of the box (i.e. move this particular z_i to its opposite bound) if the residual is reduced by this move. Set i to $i+1$ and repeat this procedure until every adjacent vertex has a larger residual.

Check: If the residual is zero, then stop with a feasible solution, z, which under the scale transformation (3) gives a solution, x, to the subset sum problem.

Branch: Generate a small number (say 10) of random integers lying between 1 and n, and move the corresponding z_i to their opposite bound. Calculate the new residual.

Set $k = k + 1$ and repeat the iteration.

We note that this algorithm is essentially a search procedure through the vertices of $B(a)$ to find a local star solution of (2), followed by a random branch step to allow a move away from a local star solution of (2) which has a positive residual and therefore does not correspond to a solution of the subset problem. The algorithm will not terminate if the original subset sum problem has no solution.

The algorithm needs an initial vertex as input. This is given by the heuristic

$$\text{For } i = 1, \ldots, n, \quad x_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{i-1} x_j a_j + a_i \leq b \\ 0 & \text{otherwise} \end{cases}$$

Although our first experiments generated the next vertex as the one which minimized the residual amongst all adjacent vertices to the current one, it was quickly found that, due to the overhead of finding this minimum, it is better to accept the first adjacent vertex which decreases the residual.

The problems were all randomly generated as follows. Note that $\lfloor x \rfloor$ is the integer part of x.

Input: U, n, where U is a large upper bound on the distribution of the a_i.

Procedure: For $i = 1, \ldots n + 2$, generate r_i, a sample from a uniform distribution on $[0, 1]$.

For $i = 1, \ldots, n$, let

$$a_i = \lfloor U r_i + 1 \rfloor$$

and let

$$b = \lfloor U r_{n+1} + 1 \rfloor \times \lfloor \frac{n}{2} r_{n+2} + 1 \rfloor$$

Three comments on this procedure are in order.

Size	No. probs	Average branches	Average CPU (secs)	σ for CPU
10000	80	80	22.2	20.7
20000	80	69	40.9	35.0
30000	80	78	68.8	77.4
40000	80	62	76.2	63.4
50000	80	59	91.9	80.2
60000	80	50	93.5	75.6
70000	80	58	129.3	113.9
80000	80	63	159.4	135.5
90000	80	60	173.3	135.5
100000	80	58	189.2	169.8

Table 1: Serial solution times for subset sum problems

1. b is stored as the product of two integers and since the precision of the random number generator we use is 10 decimal digits, we have 20 digits of accuraccy in the problem.

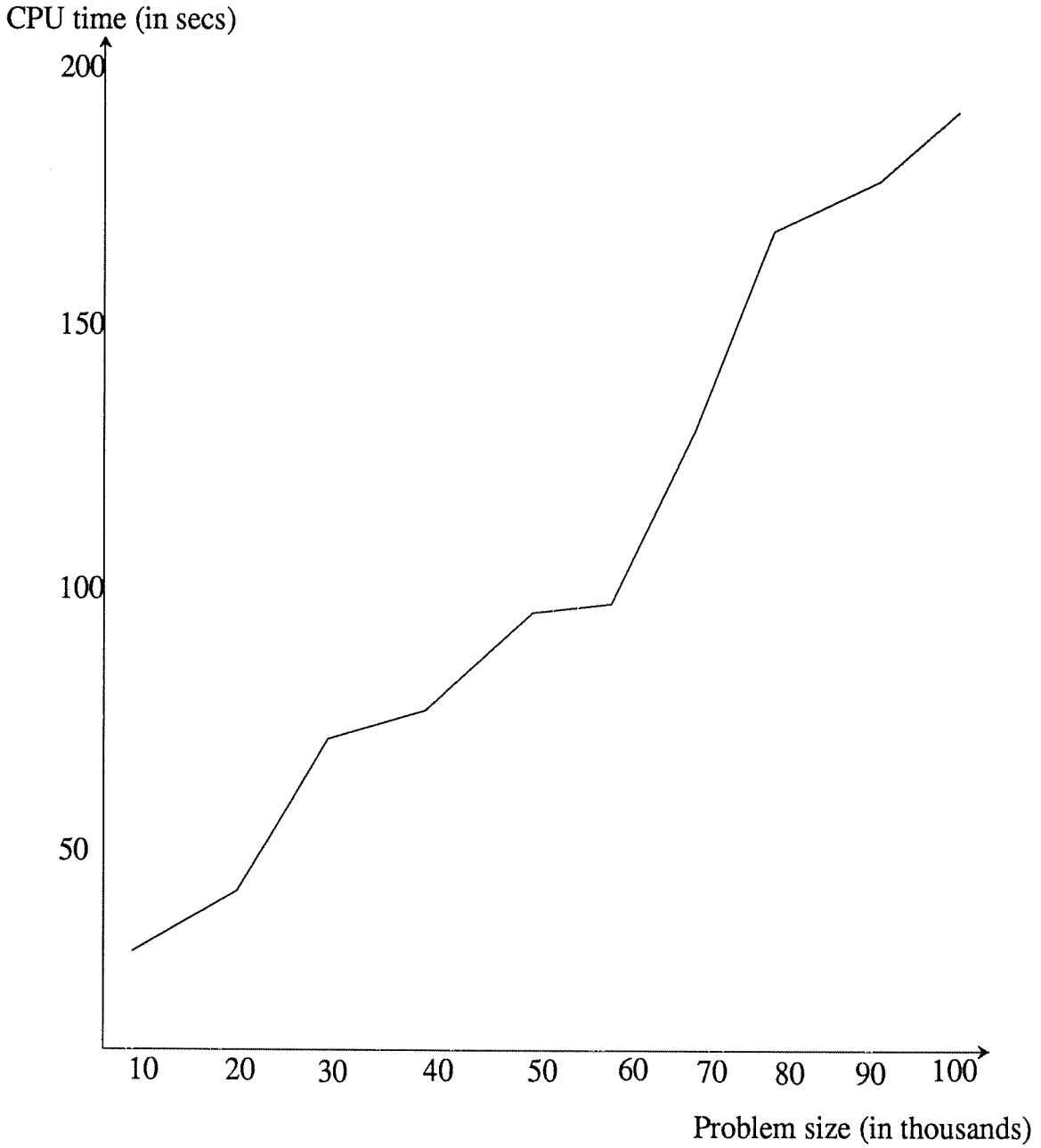
2. The problem is not necessarily solvable.

3. In all the experiments U was chosen as $30n$ to make sure that as n increases the problems are not becoming easier to solve. This scheme guarantees that the density of the a_i is constant in their range of possible values.

A sample of the computational results is given in Table 1. We note that σ represent the standard deviation observed in our experiments. A graph of the solution times for some of these experiments is given in Figure 1. These results were obtained using a Microvax II, and are averaged over a set of 80 problems. The average number of branches is noted to be approximately constant independent of the size of the problem.

It should therefore be pointed out that the value of the right hand side b, was generated as the integer product in order to generate problems which are not solvable in pseudo-polynomial time by dynamic programming. For large n, all of the problems were observed

Figure 1: Serial solution times for subset sum problem



to have solutions even though the generation scheme **did not** guarantee this. Theoretically, this can be explained by the result of to Karmarkar, Karp Lueker and Odlyzko[8] on the probabilistic analysis of optimum partitioning which shows that the probability of no solution tends to zero with n under appropriate assumptions on the problem data.

The algorithm given above is a random algorithm. The number of random branch steps needed to solve the larger problems means that the computation becomes rather expensive. We shall attempt to use a parallel architecture in the next section to limit the number of these branch steps and so effect the solution of very large problems in much smaller CPU times.

3 Random branching procedures

In the previous section we gave an algorithm which found a local star solution of the subset sum problem and then used a random branching technique, if needed, to give a new vertex from which to continue the search for a global solution. In this section we shall use a parallel architecture in order to effect the solution of very large problems by doing many random branches in parallel. The efficacy of this technique relies heavily on two factors. The first is that the search procedure used to calculate a local star solution must be efficient, and the second is that the memory needed by each individual processor must be relatively small. The former was demonstrated in the previous section. To overcome the latter, we note that a solution can be stored as a bit pattern and hence each processor only needs to hold its current residual and the bit pattern of its solution, the problem data being accessed from a shared memory.

We note that although the algorithms we propose are for the particular instance of the subset sum problem, it is hoped that the technique of random branching can be extended to further problems where a local star solution can be found easily but a (known) global solution is required.

The pattern of work for each processor in the algorithmic schema is as follows:

Initialize: Generate an initial vertex.

Iterate:

Search for a local star solution of the problem and calculate the residual of this solution.

Reset the current solution.

Branch from the current solution and repeat the iteration.

Some comments are in order. The **generate** procedure used was the heuristic given in the previous section and the **branch** and **search** procedures are also described there. There are many different ways to implement such schema on a given parallel architecture. The different algorithms are constructed by carrying out the **reset** step in different ways. Our experiments were confined to the following different techniques, where the description given below details only the **reset** step of each algorithm.

Manyruns: Check for an exit condition. If any of the processors has found a solution, then exit. (This means the procedure is almost the same as running the serial algorithm several times except that we are able to terminate very quickly after a solution is found.)

Syncmin: Wait until every processor has found a local star solution. Compare the local star solutions to find the best one. If this is a solution of the problem, then exit. Otherwise, reset all the processor solutions to the best local star solution.

Syncsome: Wait until every processor has found a local star solution. Compare the local star solutions to find the best one. If this is a solution of the problem, then exit. Otherwise, reset the processor solutions to the best local star solution if they have a residual which is more than twice the residual of the best one.

Async: Keep the best solution stored in shared memory. When a processor has found a local star solution, check this solution against the best solution immediately. If the processor solution is better than the best one, then update the best solution. If the processor solution has a residual more than twice as big as the best residual, then copy the best solution into the processor. If the best solution has residual zero, then exit.

Procs	Manyruns		Syncmin		Syncsome		Async	
r	CPU	E(r)	CPU	E(r)	CPU	E(r)	CPU	E(r)
1	17.16	–	*	–	*	–	17.71	–
2	9.15	0.93	8.95	–	9.51	–	9.65	0.92
3	8.60	0.67	5.27	1.13	7.71	0.82	6.01	0.98
4	6.45	0.67	6.91	0.65	6.92	0.69	5.28	0.84
5	4.19	0.82	4.33	0.83	5.67	0.67	3.59	0.99
6	3.68	0.78	4.22	0.71	4.60	0.69	2.54	1.16
7	3.30	0.74	3.77	0.68	5.56	0.49	2.82	0.90
8	3.22	0.67	3.52	0.64	3.71	0.64	2.51	0.88
9	3.20	0.60	4.11	0.48	3.99	0.53	2.46	0.80
10	3.04	0.56	3.98	0.45	4.08	0.47	2.26	0.78

Table 2: Efficiencies for subset sum solution

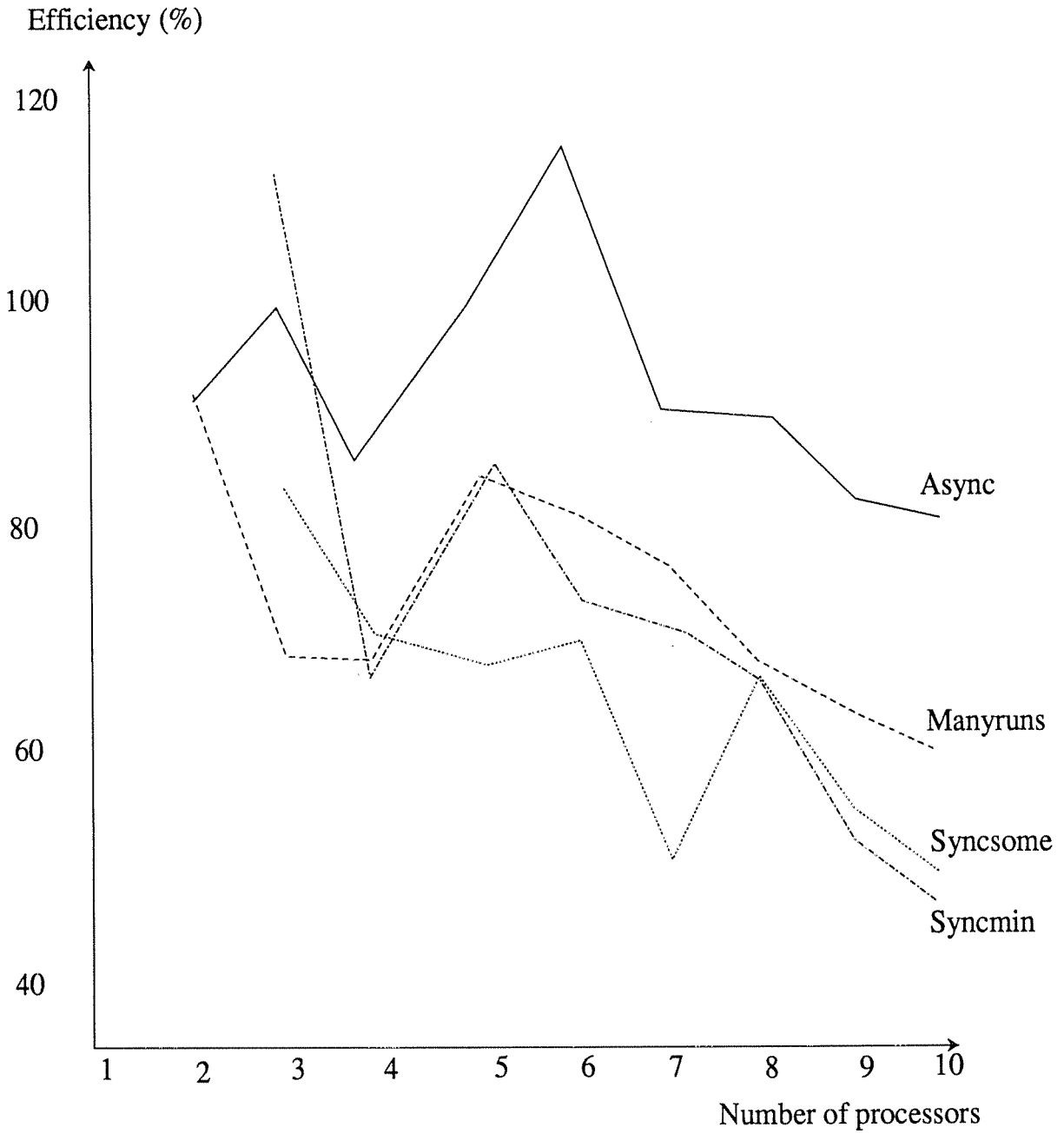
(Note that the implementation of this algorithm requires that the best solution can be read by many processors, but only written by one at a time. This is implemented using the technique of semaphores described by Dijkstra[2].)

The above algorithms constitute efficient methods for solving the subset sum problem. Table 2 and the following graph (Figure 2) support this observation with some computational results performed on a Sequent Symmetry 81 multiprocessor. The figures given are averaged over a set of 40 problems, each of which has $10,000$ variables. The definition of efficiency which we use is speedup efficiency, $E(r)$, which is defined as follows:

$$E(r) := \frac{T(1)}{rT(r)}$$

where $T(r)$ is the total time for solving a given problem using r parallel processors. In Table 2, a $*$ indicates some of the problems in the set of 40 test problems were not solved by the specified algorithm. This means that in some cases the speedup efficiencies have to be

Figure 2: Parallel solution efficiences for subset sum problem



Size	No. probs	Average branches	Average CPU (secs)	σ for CPU
100000	40	7	25.2	17.9
200000	40	9	64.0	52.9
300000	67	8	85.1	56.7
400000	80	9	130.9	114.0
500000	40	7	132.6	85.4
600000	40	9	209.8	140.9
700000	40	9	222.6	160.6
800000	40	7	212.5	128.9
900000	80	8	277.9	203.3
1000000	40	7	264.1	157.0
1500000	40	8	508.1	440.7
2000000	40	10	801.2	701.5

Table 3: Asynchronous solution times for subset sum problems

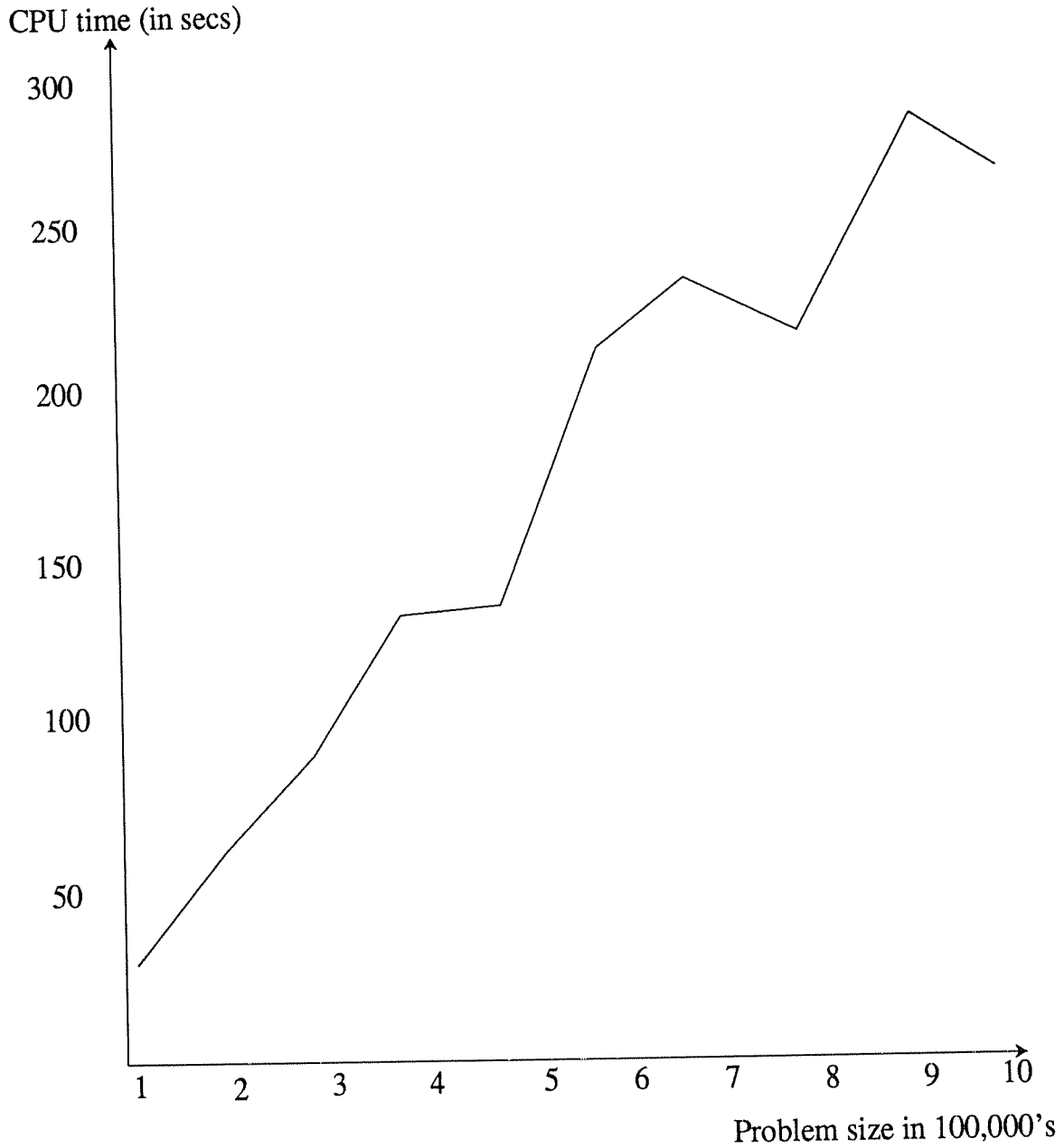
calculated from a base of 2 processors instead of 1. Note that failures occurred only when using a single processor. This is an indication that even a coarse parallelization with two processors circumvents such difficulties.

The culmination of the computational effort is given in Table 3 and illustrated in Figure 3, where some extremely large subset sum problems are solved on the Sequent Symmetry 81 multiprocessor using the asynchronous version of the algorithm. All of these results are for the case when we have 10 processors working. We note that the increase in time is **approximately linear** in n (and the storage required by the algorithm is obviously linear). Clearly, the use of parallelism has enabled the number of branches to be kept very small on average (compare Tables 1 and 3).

We note the variance in the results for the asynchronous version of the algorithm is due to two factors. These are:

1. Variance due to the asynchronous nature of the algorithm.

Figure 3: Parallel solution times for subset sum problem



2. Variance due to the varying degree of difficulty of the problems.

In order to ascertain which of these factors dominates (if any), a problem of size 400,000 was run 40 times when other machine activity was high. The resulting variance in the solution times was very small, leading to the conclusion that the variance is due mainly to the varying problem difficulty.

The author knows of no computational results for subset sum problems of this magnitude at the time of writing this paper.

References

[1] E. Balas and E. Zemel. An algorithm for large zero–one knapsack problems. *Operations Research*, 28:1132–1154, 1980.

[2] E.W. Dijkstra. Cooperating sequential processes. Technical Report EWD–123, Technological University, Eindhoven, The Netherlands, 1965.

[3] D. Fayard and G. Plateau. An algorithm for the solution of the 0–1 knapsack problem. *Computing*, 28:269–287, 1982.

[4] M.R. Garey and D.S. Johnson. *Computers and Intractability, A Guide to the Theory of NP–Completeness.* W.H. Freeman and Company, San Francisco, 1979.

[5] H. Greenberg and R.L. Hegerich. A branch search algorithm for the knapsack problem. *Management Science*, 16:327–332, 1970.

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

[7] G.P. Ingargiola and J.F. Korsh. A reduction algorithm for the zero–one single knapsack problem. *Management Science*, 20:460–463, 1973.

[8] N. Karmarkar, R.M. Karp, G.S. Lueker, and A.M. Odlyzko. Probabilistic analysis of optimum partitioning. *Journal of Applied Probability*, 23:626–645, 1986.

[9] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[10] P.J. Kolesar. A branch and bound algorithm for the knapsack problem. *Management Science*, 13:723–735, 1967.

[11] E.L. Lawler. Fast approximations algorithms for knapsack problems. *Mathematics of Operations Research*, 4:339–356, 1979.

[12] S. Martello and P. Toth. A mixture of dynamic programming and branch–and–bound for the subset–sum problem. *Management Science*, 30:765–771, 1984.

[13] S. Martello and P. Toth. Worst–case analysis of greedy algorithms for the subset sum problem. *Mathematical Programming*, 28:198–205, 1984.

[14] S. Martello and P. Toth. Algorithms for knapsack problems. In S. Martello, G. Laporte, M. Minoux, and C. Ribeiro, editors, *Surveys in Combinatorial Optimization.* Annals of Discrete Mathematics 31, North–Holland, Amsterdam, 1987.

[15] S. Martello and P. Toth. A new algorithm for the 0–1 knapsack problem. *Management Science*, 34(5):633–644, 1988.

[16] J.J. Moré and S.A. Vavasis. On the solution of concave knapsack problems. Mathematics and Computer Science Division Report ANL/MCS–P40–1288, Argonne National Laboratory, Argonne, Illinois, 1988.

[17] S. Sahni. Approximate algorithms for the 0/1 knapsack problem. *Journal of the ACM*, 22(1):115–124, 1975.